TMS320C27x
Core Software Simulator Interface (CSSI)
User's Guide

Literature Number: SPRU316
July 1999

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Read This First

About This Manual

This user's guide discusses the characteristics of the Core Software Simulator Interface (CSSI).

This document contains the following chapters:

- Chapter 1 provides an overview of the CSSI.
- Chapter 2 describes the components required for a simulation system with CSSI and the stages involved in simulation and how to use CSSI to integrate a user-defined system.
- Chapter 3 describes the interfaces provided in the CSSI definition.
- Chapter 4 describes the usage of the CSSI interface as supported by the C2700B0 simulator (sim27x).

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is a sample program listing:

  ```
  0011  0005  0001        .field    1, 2
  0012  0005  0003        .field    3, 4
  0013  0005  0006        .field    6, 3
  0014  0006              .even
  ```

  Here is an example of a system prompt and a command that you might enter:

  ```
  C:  csr –a /user/ti/simuboard/utilities
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect **"***section name***",** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

  LALK *16–bit constant [, shift]*

  The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here's an example of a list:

  `{ * | *+ | *– }`

  This provides three choices: `*`, `*+`, or `*–`.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

  .byte *value*$_1$ *[, ... , value*$_n$*]*

  This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Related Documentation From Texas Instruments

The following books describe the TMS320C27x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C27x Assembly Language Tools User's Guide (literature number SPRU211) describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C27xx device.

TMS320C27x Optimizing C Compiler User's Guide (literature number SPRU212) describes the TMS320C27xx C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the TMS320C27xx device.

TMS320C27xx C Source Debugger User's Guide (literature number SPRU214) tells you how to invoke the TMS320C27xx emulator and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C27xx DSP CPU and Instruction Set Reference Guide (literature number SPRU220) describes the central processing unit (CPU) and the assembly language instructions of the TMS320C27xx 16-bit fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

Code Composer User's Guide (literature number SPRU296) explains how to use the Code Composer development environment to build and debug embedded real-time DSP applications.

Code Composer Studio Software Developer's Kit User's Guide (literature number SPRU320) describes the Code Composer Studio software developer's kit (SDK), which allows you to execute custom plug-ins and debug them line by line. Covered are open architecture, host side automation servers, ActiveX Control clients, and programming considerations.

Code Composer Studio Application Programming Interface (API) Reference Guide (literature number SPRU321) describes the Code Composer Studio application programming interface, which allows you to program custom plug-ins for Code Composer.

Trademarks

To Help Us Improve Our Documentation . . .

If you would like to make suggestions or report errors in documentation, please send us mail or email. Be sure to include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251–1443

Email: comments@books.sc.ti.com

Contents

Chapter 1

Introduction

The TMS320C27x Core Software Simulator Interface (CSSI) provides access to the programmable instruction set simulator (ISS). The complexity of core and system designs requires fast simulation and a high degree of visibility. These needs have driven the development of an interface definition that helps extend the simulator to incorporate a user-defined system. The *Core Software Simulator Interface (CSSI)* provides that interface definition.

Topic **Page**

1.1 Development of CSSI

Simulation is an important part of design. It helps evaluate, debug, and validate a product. CSSI defines an interface to the programmable core, or processor instruction set simulator (ISS). This interface extends the simulator to incorporate a peripheral set and to link into another simulation environment or any other form of stimulus generation. It does this while still providing an instruction set architecture (ISA) level abstraction and high-level debug capabilities.

Note:

This is only an interface definition to enable such mechanisms.

1.2 Key Benefits

The CSSI interface definitions produce the following key benefits:

- Provides a means to define and measure a system through the use of simulation
  - Defining a system: Every necessary element of the system — the memory, peripherals, interrupt generation, complex memory controllers — is simulated. Defining a system may also require configuring an existing definition example such as setting up the cache size or the on-chip memory size. There can be any level of definition of the system — chip, board, and so forth.
  - Measuring system performance: To measure system performance, you must be able to analyze how well the defined system makes use of system resources. This analysis may involve measuring resource usage such as the number of accesses to a particular range of memory, the nature of such accesses, or the peripheral usage.
- Provides a target independent standard

  CSSI is a target and simulator implementation-independent interface definition.
- Enables a tradeoff between performance and functionality

  Having a standard interface to CSSI enables you to choose the memory system you want. You can choose to have a large amount of functionality which will be implemented slowly, or to have only the essential functions implement at a very rapid pace.

  Example: While modeling the memory system, you could model all the components, such as memory system controllers, to a great degree of accuracy. This would provide information on the performance of the system, e.g., wait states, cache misses, and so on. On the other hand, if only a functional model is required, a simple memory model which does not model the performance impacts, would suffice. This could dramatically influence simulation speed. CSSI provides a standard interface wherein any of the above models could be linked dynamically. This allows for functionality vs. speed tradeoff. Additionally, the use of CSSI brings in a great deal of modularity which could also influence reuse of the models in various systems.

Although these requirements include a broad usage scope, the CSSI definition does not replace logic simulation environment or any other system simulation environment. CSSI is an interface definition, not a separate system.

Chapter 2

Integrating a User-Defined System Through CSSI

A simulation system with CSSI contains the following components:

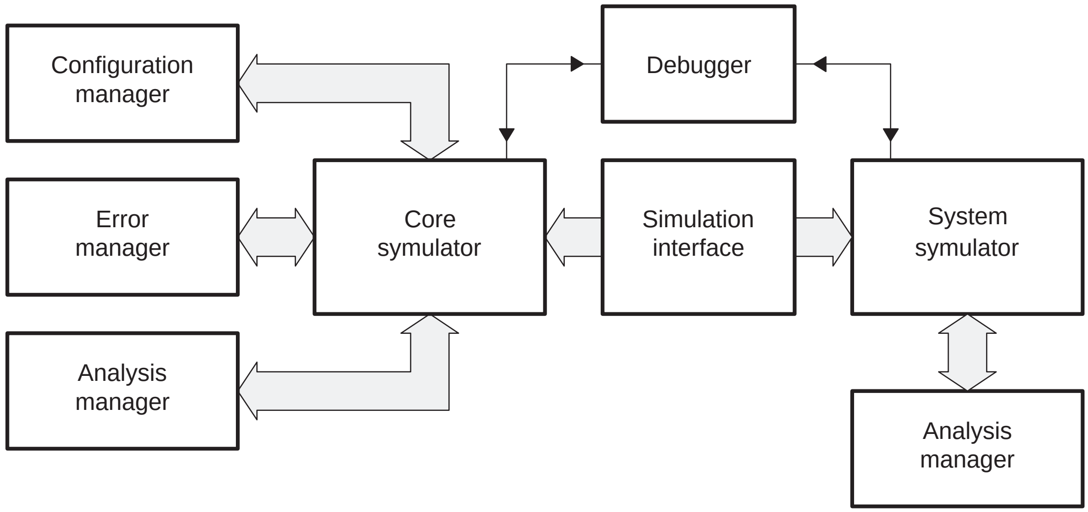
- ❑ Open Target Interface Standard (OTIS)
- ❑ Device core simulator
- ❑ System simulator
- ❑ Simulation interface
- ❑ Configuration manager
- ❑ Analysis manager
- ❑ Error manager

These components must adhere to the interface definitions of CSSI.

Topic **Page**

2.1 CSSI Components

Figure 2–1. Simulation Interface with CSSI



Open Target Interface Standard (OTIS)

The open target interface standard (OTIS) provides an interface from the target (simulation core) to the external world (debuggers, application programming interface, etc.). It provides visibility into the core and the system, as well as some basic control in terms of step, run, and breakpoints.

Device Core Simulator

The device core simulator mimics the core of the device, such as the TMS320C2700 megamodule. The core does not need to be a megamodule without any memory system. It can be any level of abstraction; for example, it can be a specific device like the TMS320C2700 with all its memory and peripherals. There is one “device core” identifiable for every instance of CSSI. Many such abstractions can exist simultaneously. For the ’C2700 simulator, a CSSI interface at the megamodule boundary implements any ’C2700 derivative. Another CSSI interface can be placed at the ’C2700 chip boundary.

The device core simulator maintains the processor state in terms of register contents and some memory elements.

System Simulator

The system simulator mimics the system; it holds the memory system, peripherals, and so forth. This is provided by the user of the CSSI interface. It can be at various levels of abstraction. Parts of it can even be implemented in another simulation environment.

Simulation Interface

The simulation interface, the key part of the CSSI definition, helps share information between the core and the rest of the system. The simulation interface also provides the synchronization between the core and the user system.

Configuration Manager

The configuration manager helps configure the system. It receives input in the form of a configuration file. This file is loaded into the internal data structures of the configuration manager so the core simulator and the system simulator can be configured as desired. The information available can include:

- Size and location of various memory elements
- Configuration of certain peripherals
- Set of supported analysis events
- Name of the shared object or dynamic link library for peripherals
- Name of the initialization function

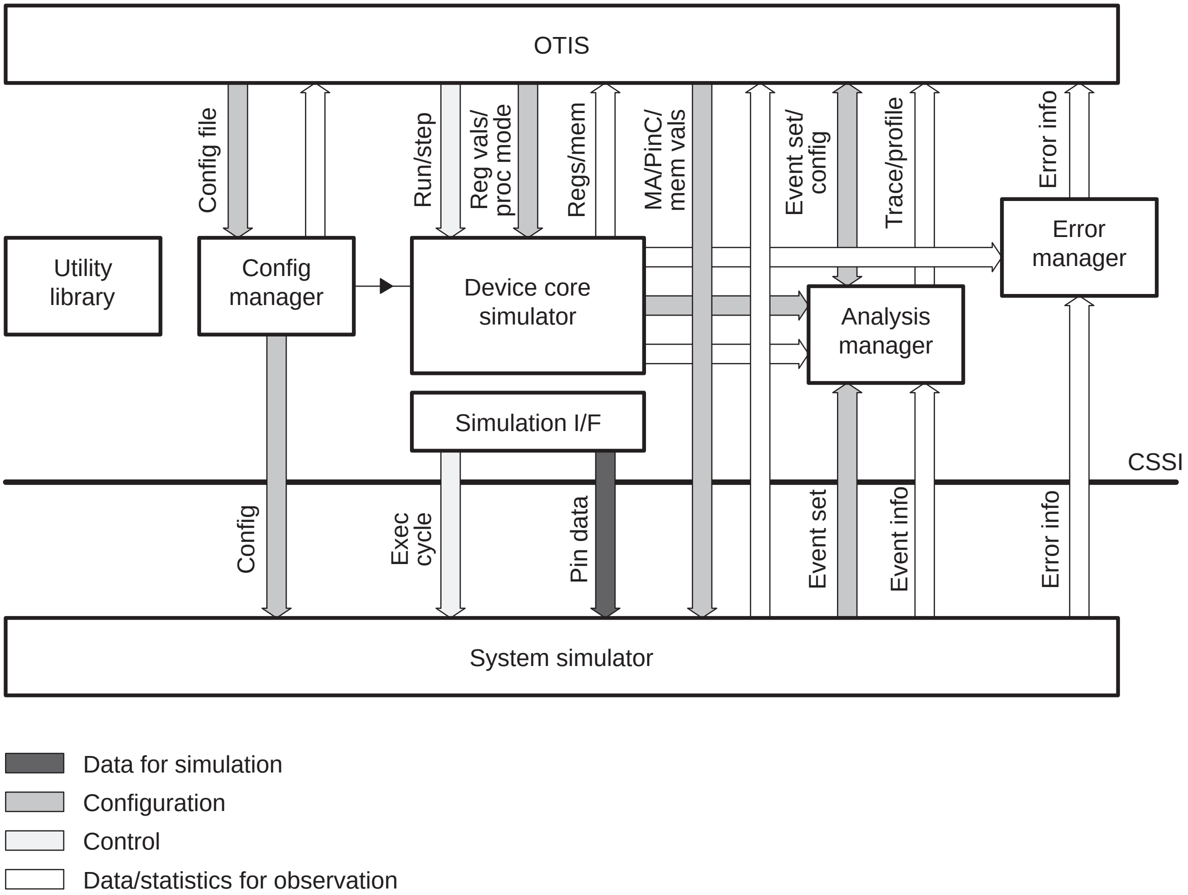
Analysis Manager

The analysis manager helps monitor system performance by gathering information from the core simulator and the system. This information pertains to the set of events supported and any occurrences of these events. The analysis manager also collates all the events happening in the system and presents them through OTIS in the form of trace, event counts, profiles, and so forth.

Error Manager

The error manager provides a uniform, centralized mechanism for reporting errors. In doing this, the error manager compiles each error that occurs in the system, formats the information, and presents it to the user.

Figure 2–2. CSSI Components



2.2 Stages in Simulating a User System

A typical simulation is comprised of the following steps:

Step 1: Initialize the system, which involves constructing the system and configuring it as needed.

Step 2: Register and notify the system of relevant events, such as the beginning of a new cycle, memory access, and system reset. The exact set of events is determined by the core to which you are interfacing.

Step 3: Determine the current inputs, such as data address and read/write control, when notified with an event. The set of values and how they are exported may depend on the core. Values, which are exported through registers, obtain register information from the core. Having obtained the inputs, set the outputs at a defined point in simulation. The outputs are read data, interrupts, and so forth.

Step 4: Observe the new state periodically. As the simulation proceeds, the state of the system changes. The user who performs the simulation must periodically observe the new state. To do this, there must be visibility into the memory contents and the register values.

Step 5: Analyze events with special value, suce as cache hit and databank conflict.

Step 6: Notify the user of any error conditions, as they occur.

Step 7: Terminate the simulation, which must happen smoothly.

These simulation stages can be directly mapped on a CSSI-based implementation.

2.3 Using CSSI to Integrate a User-Defined System

CSSI has the flexibility to let you define a user-implemented system and then interface it with the standard software simulator. To do this, perform the following steps:

Step 1: Define the user system.

Step 2: Configure the system as required by the configuration database.

Step 3: Register the events the system must handle with the core.

Step 4: Notify the events. This step associates each event with a particular function.

Step 5: Obtain access to the interface values and registers.

Step 6: Register any analysis events.

Step 7: Provide visibility into the memory and registers that are implemented by the user system.

Step 8: Provide memory map information like start address and length.

Step 9: When relevant events occur, obtain the input values and set new outputs.

Step 10: Provide notification of all errors using the *IErrors* interface.

Defining the System

A user-defined system in the CSSI paradigm is essentially an implementation of the interface defined through *ICssi*. The user-defined system can be implemented in C++ by deriving the system implementation from ICssi.

```
class UserSystem : public ICssi
{
   ...
};
```

CSSI defines methods to simulate the system and provide visibility into its state. (Note: It is permissable, however, for you to partially implement a system around a core because those methods/functions not defined are implied and will default to the appropriate functions).

After defining the interface the system is to present, the user must link to the core.

Configuring Through the Configuration Database

The user system can be configured according to simulation needs by setting up construction parameters such as memory size, start address, and location of a link library. The configuration may be specified through a configuration file that is parsed by the core simulator and provided to the system as a configuration database through *IConfig*.

Registering Events

After the basic system is defined, it must be notified whenever specific events occur. This set of supported events varies with the core being interfaced. The event set and relevant access information are obtained from *IPortMap*. The core can then be instructed to give notification of the occurrences of relevant events through *ICore*.

Accessing Interface Values

Simulation requires frequent exchanges of data with the core. This data exchange happens through registers in the ’C27xx simulator. The set of interface values and details are obtained through IRegisters (described on page 3-26). The data values can be accessed through ICore (described on page 3-7) at any point in simulation.

Performing Analysis

Analysis can be performed through the *IAnalysis*. Events to analyze can be registered with the Analysis Manager through IAnalysis. The Analysis Manager is later notified when the event occurs. The Analysis Manager is responsible for performing further analysis.

In addition, you can also perform your own analyses.

Adding Visibility into the System

The implementer of the user system can choose the extent of visibility provided by implementing various members of ICssi which are derived from *IMemory* and *IRegisters*.

Advancing Simulation

Simulation is advanced through the notification of relevant events, such as beginning of cycle and system reset. The user system implementation must synchronize itself with the core based on these events. The core itself directs these notifications based on the control instructions received through OTIS.

Notifying Errors

Notification of errors can be sent to the error manager via the IErrors interface. The error manager holds a centralized queue of all errors that have occurred in the system since the simulation began.

2.4 CSSI Example

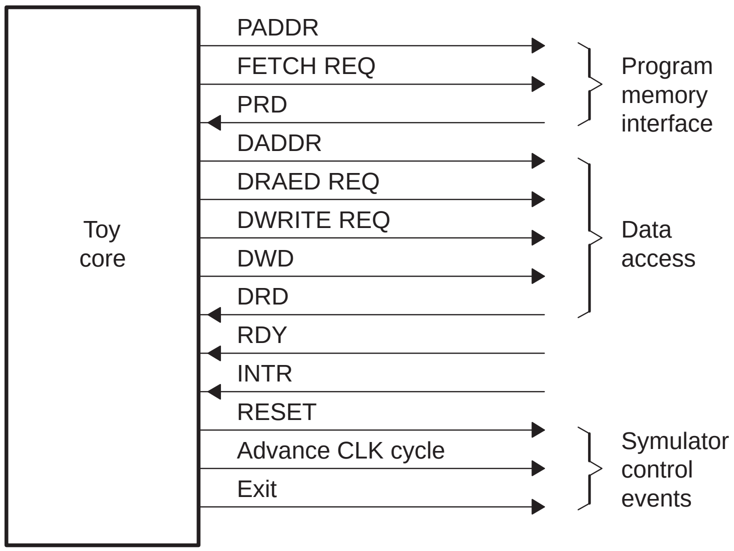
This example illustrates how a user-defined system can be integrated with a simulator. A simple processor is defined for this example for the purposes of illustration. This example highlights most of the aspects of CSSI. In practice, the usage may be more complex due to more sophisticated error handling and more realistic interface protocols.

The Toy System

The toy system that we are considering in Figure 2–3 is based upon a fictitious core. The core has a Harvard architecture with one program fetch bus and one data access bus. All memory accesses happen in a single cycle and follow a very simple access protocol. The processor has one interrupt.

A user system is constructed around this fictitious core. This system implements memory. The program and data memory are treated as one unified address space. Each time memory accesses a specified range of memory, the system generates interrupts to the core. The range of memory that triggers these interrupts can be located in the configuration file.

Figure 2–3. The Toy System



Constructing the User System

The system is constructed using C++. The system implements the ICssi interface.

```
#include <cssi.h>
```

```
class System : public ICssi
```

At construction, the system implementation:

- ❑ Obtains access to all the interface values. These interface values are used to exchange information with the core on a regular basis. These include address buses, data buses, control status, interrupts and anything else. The interface values supported by our example core are:

Value	Definition
FETCH	Program fetch control
PADDR	Program address
PRD	Program fetch data
DREAD	Read on data bus control
DWRITE	Write on data bus control
DADDR	Data address
DRD	Read data
DWD	Write data
RDY	Ready control for timing
INTR	Interrupt input

- Registers the event handlers. Most of the simulation advances are notified through various events. The events in our system are:

Value	Definition
CYCLE	CPU clock tick
EXIT	End of world predictor
RESET	System reset (debugger initiated)
FETCH	Program fetch
DREAD	Read on data bus
DWRITE	Write on data bus

- Configures the system based on the configuration data base. The example user system supports just one element of configuration. A range of addresses may be configured to trigger the CPU interrupt.

Getting Interface Values

```
port_map.getValueID(”FETCH”,  fetch_val_id);
port_map.getValueID(”PADDR”,  paddr_val_id);
port_map.getValueID(”PRD”,    prd_val_id);
port_map.getValueID(”DREAD”,  dread_val_id);
port_map.getValueID(”DWRITE”, dwrite_val_id);
port_map.getValueID(”DADDR”,  daddr_val_id);
port_map.getValueID(”DRD”,    drd_val_id);
port_map.getValueID(”DWD”,    dwd_val_id);
port_map.getValueID(”RDY”,    rdy_val_id);
port_map.getValueID(”INTR”,   intr_val_id);
```

The value identifiers obtained through getValueID would later be used to exchange data with the core.

Adding Event Notifiers

```
port_map.getEventID(”RESET”, reset_evnt_id);
core.notifyOnEvent(reset_evnt_id, this, reset);

port_map.getEventID(”CYCLE”, cycle_evnt_id);
core.notifyOnEvent(cycle_evnt_id, this, cycle);

port_map.getEventID(”EXIT”, exit_evnt_id);
core.notifyOnEvent(exit_evnt_id, this, exit);

port_map.getEventID(”DREAD”, dread_evnt_id);
core.notifyOnEvent(dread_evnt_id, this, dataAccess);

port_map.getEventID(”DWRITE”, dwrite_evnt_id);
core.notifyOnEvent(dwrite_evnt_id, this, dataAccess);
```

getEventID provides an identifier of the event. Once a function is registered to be called through the notifyOnEvent (id, this, func) call, the member function would be invoked whenever the even id occurs. The function would be called with id as one of its parameters.

Configuring the System

Configuration File

```
...
MODULE MYSYSTEM;
   INTR_ADDR_START 1000;
   INTR_ADDR_END   1100;
END MYSYSTEM;
...
```

Configuring the Interrupt’s Address Space

```
if( config.getValue(”INTR_ADDR_START”, intr_addr_start) ){
  if( config.getValue(”INTR_ADDR_END”, intr_addr_end) ){
    if( intr_addr_end > intr_addr_start ){
      interrupts_enabled = 1;
    }
    else{
      ...
    }
  }
  else{
    ...
  }
}
```

Registering the analysis event

```
analysis.registerEvent("Interrupt Range Hit",
intr_addr_hit_evnt_id);
```

Doing Some Simulation

Simulation proceeds through a series of event notifications. The notifier functions are set up at the time of construction. Typical notification includes beginning of cycle, reset, program fetch, and so forth. The notifier must read the respective interface values, perform the necessary evaluations, and send out the required set of outputs. The implementation of the program fetch notifier is shown below.

```
int System::progFetch(CSSI_EventID)
{
  CSSI_Value fetch_addr;
  CSSI_Value fetch_data;
  core.getValue(paddr_val_id, fetch_addr);
  fetch_data = memory[fetch_addr % MAX_MEM_SIZE];
  core.setValue(prd_val_id, fetch_data);
  return 1;
}
```

Interrupts and Other Stimulus

Interrupts and other stimulus to the CPU core are simply notifications at the apropriate instance. The example system sends an interrupt to the CPU every time a data access is performed to a specified range of memory. The interrupt is sent to the CPU a few cycles after this access. A data access to the interrupt address range is also considered as an analysis event.

```
int System::dataAccess(CSSI_EventID id)
{
  CSSI_Value addr, data;
  core.getValue(daddr_val_id, addr);
  if( IS_INTR_ADDR(addr) ){
    if( intr_analysis_enabled )
      analysis.eventOccured(intr_addr_hit_evnt_id); // do
it only if we are told to
    count_to_intr = 5;
  }
  if( id == dread_evnt_id ){
    /* read access */
    ...
  }
  else{
    /* write access */
    ...
  }
  return 1;
}
int System::cycle(CSSI_EventID)
{
  if( count_to_intr > 0 ){
    count_to_intr––;
  }
  else if( count_to_intr == 0 ){
    core.setValue(intr_val_id, 1); // fire
    count_to_intr = –1;            // reset event counter
  }
  return 1;
}
```

Adding Debugger Visibility

System visibility is a very important aspect of simulation. CSSI provides a standard mechanism for visibility into the memory implemented by the system. It also provides a standard mechanism for defining memory maps.

```
int System::store(CSSI_Address addr, CSSI_Value data)

{

  memory[addr % MAX_MEM_SIZE] = data;

  return 1;

}

int System::fetch(CSSI_Address addr, CSSI_Value& data)

{

  data = memory[addr % MAX_MEM_SIZE];

  return 1;

}
```

Building and Linking Up

The user system must finally be built as a dynamic library (DLL). The library is then dynamically linked with the simulator at run time. The configuration file is used to specify which dynamic library to use. An entry point must be chosen for the DLL. The entry point function is then used to obtain a link into the user system.

The example entry point function:

```
extern ”C” ICssi *NewSystem(CSSI_Name name, ICore& core,
IPortMap&
                            port_map, IConfig& config, IA-
nalysis&
                            analysis, IErrors& errors)
{
  return new System(name, core, port_map, config, analy-
sis, errors);
}
```

The previous configuration file now becomes:

```
/*
 * Top Level Module
 */
MODULE MAIN;
        CSSI_MODULES MYSYSTEM;       // indicate all the
cssi submodules
END MAIN;

/*
 * The Toy System
 */
MODULE MYSYSTEM;
        CSSI_LIBRARY    example.dll;  // DLL which imple-
ments the system
        INIT_FUNCTION   NewSystem;    // Initialisation
Function
        INTR_ADDR_START 1000;         // Configure the in-
terrupt range
        INTR_ADDR_END   1100;         //
END MYSYSTEM;
```

The Complete Source

Example 2–1. example.h

```
/*
 * example.h : Example CSSI system
 */
#include <cssi.h>
#ifndef example_h
#define example_h
#define MAX_MEM_SIZE 1024
#define IS_INTR_ADDR(x) (interrupts_enabled &&\
                       ((x) >= intr_addr_start) && \
                       ((x) <= intr_addr_end))
class System : public ICssi
{
  /*
   * Set of value ids that would be needed
   */
  CSSI_ValueID fetch_val_id, paddr_val_id, prd_val_id, dread_val_id,
    dwrite_val_id, daddr_val_id, drd_val_id, dwd_val_id, rdy_val_id,
    intr_val_id;
  /*
   * Set of event ids that would be needed
   */
  CSSI_EventID dread_evnt_id, dwrite_evnt_id;
  CSSI_Value memory[MAX_MEM_SIZE];       // simple simulated memory
  int count_to_intr;                     // keep track of when to intr
                                         // reset –1; 0 => interrupt;
                // >0 => decr
  int        interrupts_enabled;
  CSSI_Value intr_addr_start, intr_addr_end;
  /*
   * analysis event ids
   */
  CSSI_AnalysisEventID intr_addr_hit_evnt_id;
  int                  intr_analysis_enabled;

public:
  /*
```

Example 2–1. example.h (Continued)

```
    * Constructor & Destructor
    */
  System(CSSI_Name _name,ICore&, IPortMap&, IConfig&, IAnalysis&, IEr-
rors&);
  ~System();
  /*
    * Event Handlers
    */
  int reset(CSSI_EventID);
  int cycle(CSSI_EventID);
  int progFetch(CSSI_EventID);
  int dataAccess(CSSI_EventID);
  /*
    * Memory Visibility
    */
  int store(CSSI_Address, CSSI_Value);
  int fetch(CSSI_Address, CSSI_Value&);
  /*
    * Analysis
    */
  int analysisEventControl(CSSI_AnalysisEventID id, int enable);
};
#endif
```

Example 2–2. example.cpp

```
 * example.cpp : Example CSSI system
 */

#include <example.h>

/*
 * Entry point into the DLL
 */
extern ”C” ICssi *NewSystem(CSSI_Name _name, ICore& core, IPortMap&
             port_map, IConfig& config, IAnalysis&
             analysis, IErrors& errors)
{
  return new System(name, core, port_map, config, analysis, errors);
}

/*
 * Constructor & Destructor
 */
System::System(CSSI_Name _name, ICore& _core, IPortMap& _port_map,
          IConfig& _config, IAnalysis& _analysis, IErrors& _errors):
  ICssi(_name, _core, _port_map, _config, _analysis, _errors)
{
  /* obtain the value ids            */
  port_map.getValueID(”FETCH”,  fetch_val_id);
  port_map.getValueID(”PADDR”,  paddr_val_id);
  port_map.getValueID(”PRD”,    prd_val_id);
  port_map.getValueID(”DREAD”,  dread_val_id);
  port_map.getValueID(”DWRITE”, dwrite_val_id);
  port_map.getValueID(”DADDR”,  daddr_val_id);
  port_map.getValueID(”DRD”,    drd_val_id);
  port_map.getValueID(”DWD”,    dwd_val_id);
  port_map.getValueID(”RDY”,    rdy_val_id);
  port_map.getValueID(”INTR”,   intr_val_id);

  /* set up the event notifiers      */
  CSSI_EventID reset_evnt_id, cycle_evnt_id, fetch_evnt_id,
    exit_evnt_id;

  port_map.getEventID(”RESET”, reset_evnt_id);
  core.notifyOnEvent(reset_evnt_id, this, (CSSI_EventFunc)reset);

  port_map.getEventID(”CYCLE”, cycle_evnt_id);
  core.notifyOnEvent(cycle_evnt_id, this, (CSSI_EventFunc)cycle);

  port_map.getEventID(”EXIT”, exit_evnt_id);
  core.notifyOnEvent(exit_evnt_id, this, (CSSI_EventFunc)exit);

  port_map.getEventID(”FETCH”, fetch_evnt_id);
  core.notifyOnEvent(fetch_evnt_id, this, (CSSI_EventFunc)progFetch);
```

Example 2–2.example.cpp (Continued)

```
   port_map.getEventID(”DREAD”, dread_evnt_id);
   core.notifyOnEvent(dread_evnt_id, this, (CSSI_EventFunc)dataAccess);

   port_map.getEventID(”DWRITE”, dwrite_evnt_id);
   core.notifyOnEvent(dwrite_evnt_id, this, (CSSI_EventFunc)dataAccess);

   /* read the config data base       */
   /*
    * The only configurable aspect is the range of memory address that
    * once accessed flags off an interrupt
    * The config is optional – but if pressent must be complete (start,
    * end) & correct end > start
    * This also demos the error reporting in CSSI
    * An access to the interrupt range is also to be analysed through
    * the analysis manager
    */
   interrupts_enabled    = 0;
   intr_analysis_enabled = 0;   // by default analysis disabled
   if( config.getValue(”INTR_ADDR_START”, intr_addr_start) ){
     if( config.getValue(”INTR_ADDR_END”, intr_addr_end) ){
       if( intr_addr_end > intr_addr_start ){
    interrupts_enabled = 1;
    /* all is fine – set up analysis events */
    analysis.registerEvent(”Interrupt Range Hit”, intr_addr_hit_evnt_id);
       }
       else{
    CSSI_ErrorSource err_src;
    err_src.name = ”INTR_ADDR_END”;
    errors.queueError(CSSI_ERR_CFG_INVALID, CSSI_ERR_CFG_INVALID,
            err_src, ”INTR_ADDR_END must be greater than
 INTR_ADDR_START”);
       }
     }
     else{
       CSSI_ErrorSource err_src;
       err_src.name = ”INTR_ADDR_END”;
       errors.queueError(CSSI_ERR_CFG_INVALID, CSSI_ERR_CFG_INVALID,
          err_src, ”INTR_ADDR_END missing”);
     }
   }
```

Example 2–2.example.cpp (Continued)

```
  /* initialize some internal states */
  reset(reset_evnt_id);
}

System::~System()
{
}

/*
 * Event Handlers
 */
int System::reset(CSSI_EventID)
{
  count_to_intr = –1;
  return 1;
}

int System::cycle(CSSI_EventID)
{
  if( count_to_intr > 0 ){
    count_to_intr––;
  }
  else if( count_to_intr == 0 ){
    core.setValue(intr_val_id, 1); // fire
    count_to_intr = –1;            // we don’t want false triggers
  }

  return 1;
}

int System::progFetch(CSSI_EventID)
{
  CSSI_Value fetch_addr;
  CSSI_Value fetch_data;
  core.getValue(paddr_val_id, fetch_addr);
  fetch_data = memory[fetch_addr % MAX_MEM_SIZE];
  core.setValue(prd_val_id, fetch_data);
  return 1;
}
```

Example 2–2.example.cpp (Continued)

```
int System::dataAccess(CSSI_EventID id)
{
  CSSI_Value addr, data;

  core.getValue(daddr_val_id, addr);
  if( IS_INTR_ADDR(addr) ){
    if( intr_analysis_enabled )
      analysis.eventOccured(intr_addr_hit_evnt_id); // do it only if
                           // we are told to
    count_to_intr = 5;
  }
  if( id == dread_evnt_id ){
    /* read access */
    data = memory[addr % MAX_MEM_SIZE];
    core.setValue(drd_val_id, data);
  }
  else{
    /* write access */
    core.getValue(dwd_val_id, data);
    memory[addr % MAX_MEM_SIZE] = data;
  }
  return 1;
}

/*
 * Memory Visibility
 */
int System::store(CSSI_Address addr, CSSI_Value data)
{
  memory[addr % MAX_MEM_SIZE] = data;
  return 1;
}

int System::fetch(CSSI_Address addr, CSSI_Value& data)
{
  data = memory[addr % MAX_MEM_SIZE];
  return 1;
}

/*
 * Analysis
 */
int System::analysisEventControl(CSSI_AnalysisEventID id, int enable){
  if( interrupts_enabled && id == intr_addr_hit_evnt_id )
    intr_analysis_enabled = enable;
  return 1;
}
```

Chapter 3

CSSI Interfaces

The CSSI interfaces enable easy extendibility of the software simulator. These interfaces also provides a standard interface across processor families. CSSI interfaces provide a uniform means of managing configuration, errors, and analysis. Along with these benefits, core implementation is hidden with the use of CSSI interfaces. The nine CSSI interfaces provided in the CSSI definition are as follows:

- ICssi
- ICore
- IConfig
- IAnalysis
- IError
- IMemory
- IRegisters
- IPinIO
- IPortMap

Topic **Page**

3.1 Interfaces Overview

The nine CSSI interfaces constitute the CSSI definition.

- **ICssi**: The most important of the interfaces, ICssi is the interface into the user system and must be implemented by the user. Through this interface, the user system also has access to the other classes.

  ICssi provides members functions to help determine the configuration of the system. These members query supported register sets, pins, and memory maps.

  ICssi enables configuration of map settings and pin controls in a system. It also provides a window into the system to view memory contents and register values. ICssi is used to notify the user system of the events of interest, such as clock cycle boundaries and changes of relevant values. These events help provide synchronization to the user system.

- **ICore**: ICore provides visibility into the core simulator via the user system. It accesses and influences all the interface values. Memory space and register set held by the core simulator can also be accessed through this interface.

  Members functions are provided to ascertain the set of interface values, core identification, and register set.

- **IConfig**: IConfig provides access to the configuration database. The configuration is managed through {tag, value} pairs. The values can be queried by supplying appropriate tags. The values can be strings, numbers, list of values, or a new configuration database itself. Minimal constraint is imposed on the set of tag and value valid space.

- **IAnalysis**: IAnalysis is used to extract useful performance metrics in the system. The interface helps register events to the analysis manager and then give notification of those events. The analysis manager helps provide higher level analysis, like stopping on certain events and counting event occurrences.

- **IErrors**: IErrors provides a consistent framework for reporting and handling errors. Members are defined for queueing errors into a centrally maintained error queue.

- **IMemory**: IMemory provides a memory abstraction of a system. IMemory can be used to access the memory, define and query memory maps, and implement the port-connect-to-file feature.

- **IRegisters** : IRegisters provides an abstraction to access the registers in the design. It provides the getFirstRegisterInfo and getNextFileConnectInfo member functions to query the set of registers. IRegisters also provides the getRegisterValue and setRegisterValue member functions to access register values for reading or writing

- **IPinIO**:IPinIO provides an interface to support the pin-connect-to-file feature. It provides member functions to query the set of pins (getFirstPinInfo and getNextPinInfo). It also provides for an interface to connect and disconnect pins to files.

- **IPortMap**: IPortMap is used to determine the ids for the interface values and events supported by the simulator core. It provides member functions that help query the set of values and events supported. Member functions are also provided to translate string names of the interface entities ids. These ids are used later when communicating to the core through ICore.

3.2 ICssi

The ICssi interface is implemented by the user of the CSSI-based system. It is the window into the user system. The user system derives from this interface. At construction, it provides the user system access to the other relevant interfaces. The interface provides visibility into the user system through access to the memory and registers that the system may implement. The user system may be notified on the occurrence of relevant events. The user system must register what events it would like to be notified on. These events, which help in the simulation process, include:

- Reset
- Beginning of cycle
- Exit
- Memory access

For details on using the CSSI interface to implement a user system, see *section 2.3, Using CSSI to Integrate a User-Defined System.*

In order to provide the visibility required, ICssi is derived from the following interfaces:

- IMemory
- IRegisters
- IPinIO

Constructor

Accesses the core, interface values, error queue, analysis manager

Syntax

ICssi(CSSI_Name& name, IPortMap& port_map, IConfig& config, IAnalysis& analysis, IErrors& errors)

Description

Constructor provides access to the core, interface values, error queue, analysis manager, and the configuration database. It can use the configuration database to configure the system. The user system is notified of the simulation events through CSSI events. The construcor must be used to register the required set of events. Important events include reset and beginning a new cycle.

Once constructed, the system is notified whenever a relevant event occurs. The system can obtain the current state through the various values exposed by the core. The system can also set up any values requred.

Input

name	Name of the system to construct. Usually, this is picked from the configuration database. It can be used as a configuration parameter, too. For example, the names PROC1 and PROC2 can be used to identify what processor type is being modeled.
core	This is a reference to the core implementation. This would be used to set up event notifiers, obtain interface values, and obtain visibility into the core state.
port_map	The portmap provides a mapping between string names on the interface values and events and their identifiers. It can also be used to query the set of interface values.
config	Access to the configuration database.
analysis	Access to the Analysis Manager.
errors	Access to the error manager. This helps provide a consistent error reporting mechanism.

Output

none

Return Value(s)

none

analysisEvent-Control

Notify when a registered analysis event status changes

Syntax

int analysisEventControl (CSSI_AnalysisEventID id, int enable);

Description

analysisEventControl is used to notify all CSSI systems when a previously registered analysis event is enabled or disabled.

Input

id	Id of the analysis event.
enable	Enable control. A non-zero value indicates the event must be enabled. A zero value indicates the event must be disabled.

Output

none

Return Value(s)

A non-zero return status indicates successful completion.
A zero return status indicates unsuccessful completion.

3.3 ICore

ICore provides a window into the core simulator. The purpose of this interface is to provide:

- A mechanism for sharing values with the rest of the system for the purpose of simulation
- Visibility into the internal state of the core

However, the purpose of this visibility is primarily for observing the state only. Using it to alter the state may create unexpected results. For example, if the user-implemented system attempts to alter the contents of a register which is being used by an instruction, it is not guaranteed that the instruction will see the updated value.

ICore derives from the IMemory and IRegisters interfaces in order to provide the visibility required.

The ICore interface members are:

- getValue
- setValue
- notifyOnEvent
- denotifyOnEvent

getValue

Retrieve the interface value

Syntax int getValue(CSSI_ValueID, CSSI_Value&)

Description Obtain the interface value. The value can be set through setValue. Every interface value has a unique ID which can be obtained through getValueID.

Input id — Identifcation of the interface value that is desired.

Output value — Current value

Return Value(s) A non-zero return status indicates successful completion.
A zero return status indicates an unsuccessful completion.

setValue

Set the interface value

Syntax int setValue(CSSI_ValueID, CSSI_Value)

Description Set up the interface value. The value can be obtained through getValue. Every interface value has a unique ID which can be obtained through getValueID.

Input id — Identification of the interface value that is desired.

Output value — New value.

Return Value(s) A non-zero return status indicates successful completion.
A zero return status indicates an unsuccessful completion.

notifyOnEvent

Nofity user system of an event

Syntax

int notifyOnEvent(CSSI_EventID when, ICssi* who, CSSI_EventFunc how)

Description

Configure the core to notify the user system of the occurence of a particular event. The notification is done through invoking the member function pointer. The notifier is usually set up during the construction of the user system. The set of events provided by the core can be obtained through:

- getFirstEventInfo
- getNextEventInfo
- getEventID

Input

when	The event on which to notify.
who	The interface to notify.
how	The member function to call

Output

none

Return Value(s)

A non-zero return status indicates successful completion.
A zero return status indicates an unsuccessful completion.

denotifyOn-Event

Denofity user of an event

Syntax

int denotifyOnEvent(CSSI_EventID when, ICssi* who, CSSI_EventFunc how)

Description

Configure the core to denotify the user system of the occurrence of a particular event. The denotification is done through invoking the member function pointer. This function can be used when peripherals are not triggered for an event.

Input

when	The event on which to notify.
who	The interface to notify.
how	The member function to call

Output

none

Return Value(s)

A non-zero return status indicates successful completion.
A zero return status indicates an unsuccessful completion.

3.4 IConfig

IConfig provides an interface to the configuration database. The configuration holds dynamic aspects of the system (like the location of libraries used) and various parameters (like the cache size). The database is managed as a set of tag and value pairs. The database provides the value given the tag. The value may be a:

- ❑ Numeric value
- ❑ Name
- ❑ List of values
- ❑ Configuration itself. This helps provide for a hierarchical system.

While supporting a hierarchical system, the query database provides a concept of scope for the access. The scope may be local or global. When a local scope is used, only the current database is queried. When the global scope is used, the current database is queried first. If the tag does not find a match, the parent database (if one exists) is queried. This procedure is followed recursively.

The IConfig interface members are:

- ❑ getName
- ❑ getType
- ❑ getParent
- ❑ getValue{name}
- ❑ getValue{Value List}
- ❑ getValue{numeric value}
- ❑ getValue{configuration database}
- ❑ getFirstValue {name}
- ❑ getFirstValue {numeric value}
- ❑ getNext
- ❑ getNextValue {name}
- ❑ getNextValue {numeric value}
- ❑ setvalue

getName — *Get the configuration database name*

Syntax	const CSSI_Name getName()	
Description	Obtain the name of the configuration database.	
Input	none	
Output	return value	This function provides the database name.
Return Value(s)	none	

getType — *Get the configuration database type*

Syntax	const CSSI_Name getType()	
Description	Obtain the type of the given configuration database.	
Input	none	
Output	return value	The name of the configuration database type. This is the same as the block name for the configuration.
Return Value(s)	none	

getParent — *Get the parent configuration database*

Syntax	const IConfig* getParent()	
Description	Obtain the parent configuration database, if one exists.	
Input	none	
Output	return value	Access to the parent database, if no parent, returns NULL.
Return Value(s)	none	

getValue{name}

Find matching string value

Syntax
int getValue(CSSI_Tag, CSSI_Name&, ConfigScope=CFG_SCOPE_LOCAL)

Description
Finds the matching string value for a given tag, if one exists.

Input

	tag	Configuration tag for which to look.
	scope	Look either in the local database, or in both the local database and the parent database

Output

	value	Value corresponding to the supplied tag.

Return Value(s)
A non-zero return value indicates a successful find. This value will correspond with the name you supplied.
A zero return value indicates that the name could not be located in the database.

getValue{ValueList}

Find matching value list

Syntax
int getValue(CSSI_Tag, IConfigValueListPtr&, ConfigScope=CFG_SCOPE_LOCAL)

Description
Find the matching value list for a given tag, if one exists.

Input

	tag	Configuration tag for which to look.
	scope	Look either in the local database, or in both the local database and the parent database.

Output

	value	Value corresponding to the supplied tag.

Return Value(s)
A non-zero return value indicates a successful find.
A zero return value indicates that the value could not be located in the database.

getValue-{numeric value}

Find matching numeric value

Syntax	int getValue(CSSI_Tag, CSSI_Value&, ConfigScope=CFG_SCOPE_LOCAL)	
Description	Find the matching numeric value for a given tag, if one exists.	
Input	tag	Configuration tag for which to look.
	scope	Look either in the local database, or in both the local database and the parent database.
Output	value	Value corresponding to the supplied tag.
Return Value(s)	A non-zero return value indicates a successful find. This value will correspond to the numeric value you supplied. A zero return value indicates that the numeric value could not be located in the database.	

getValue-{configuration database}

Find matching configuration database

Syntax	ing getValue(CSSI_Tag, IConfigPtr&, ConfigScope=CFG_SCOPE_LOCAL)	
Description	Find the matching configuration database for a given tag, if one exists.	
Input	tag	Configuration tag for which to look.
	scope	Look either in the local database, or in both the local database and the parent database.
Output	value	Value corresponding to the supplied tag.
Return Value(s)	A non-zero return value indicates a successful find. This value will correspond to the configuration database you supplied. A zero return value indicates that the configuration database could not be located.	

GetFirst-Value{name}

Get first name

Syntax int getFirstValue(CSSI_Name&)

Description Obtain the first name from a list.

Input none

Output return value Returns name if found.

Return Value(s) A non-zero return value indicates a successful find. This value will correspond to the name you supplied.
A zero return value indicates that the name could not be located in the list.

GetFirstValue {numeric value}

Get the first numberic value

Syntax int getFirstValue(CSSI_Value&)

Description Obtain the first numeric value from a list.

Input none

Output return value Returns numeric value if found.

Return Value(s) A non-zero return value indicates a successful find. This value will correspond to the numeric value you supplied.
A zero return value indicates that the numeric value could not be located in the list.

getNext

Get the next member pointer

Syntax ConfigValueList* getNext() const;

Description Obtain the pointer to the next member of the list, if one exists.

Input none

Output return value Returns the pointer to the next member in the list.

Return Value(s) none

GetNextValue {name}

Get next name in list

Syntax int getNextValue (CSSI_Value&)

Description Obtain the next name from a list.

Input none

Output return value Returns name if found.

Return Value(s) A non-zero return value indicates a successful find. This value will correspond to the name you supplied.
A zero return value indicates that the name could not be located in the database.

GetNextValue {numeric value}

Get the next numeric value

Syntax int getNextValue (CSSI_Value&);

Description Obtain the next numeric value from a list.

Input none

Output return value Returns numeric value if found.

Return Value(s) A non-zero return value indicates a value was found. This value will correspond with the numeric value you supplied.
A zero return value indicates a value was not found.

setValue

Set the value of a member

Syntax void setValue (ConfigValue* value);

Description Set the value of a member in the Config Value list.

Input none

Output none

Return Value(s) none

3.5 IAnalysis

IAnalysis is used to extract useful performance metrics in the system. This interface is still under definition.

The IAnalysis interface members are:

- registerEvent
- getFirstEventInfo
- getNextEventInfo
- eventOccurred
- pendingError
- stdDescription
- dequeueError

registerEvent

Register an event to be analyzed by the analysis manager

Syntax int registerEvent(CSSI_Name name, CSSI_AnalysisEventID& id)

Description Can be used to register an event to be analyzed with the analysis manager. The id obtained at the registry can later be used by eventOccurred to give notification of the occurrence of the event. analysisEventControl would be called for all the registered events when the event is enabled or disabled.

Input name — Analysis event name

Output id — Assigned id for the event to be used while notifying at a later time.

Return Value(s) A non-zero return value indicates success.

getFirstEvent-Info

Access information on supported events

Syntax int getFirstEventInfo(CSSI_Name& name, CSSI_AnalysisEventID& id)

Description getFirstEventInfo and getNextEventInfo provide access to the information on all the supported events. Information on the first supported event can be obtained with getFirstEventInfo. Information on the subsequent events can later be accessed through a sequence of calls to getNextEventInfo.

Input none

Output name — Name of the event

id — Id of the event

Return Value(s) A zero return value indicates there were no more events.
A non-zero return value indicates a successful read.

getNextEvent-Info

Access information of supported events

Syntax int getNextEventInfo(CSSI_Name& name, CSSI_AnalysisEventID& id)

Description getFirstEventInfo and getNextEventInfo provide access to the informatin on all the supported events. Informatin on the first supported event can be obtained with getFirstEventInfo. Informatin on the subsequent events can later be accessed through a sequence of calls to getNextEventInfo.

Input none

Output

name	Name of the event
id	Id of the event

Return Value(s) A zero return value indicates there were no more events.
A non-zero return value indicates a successful read.

eventOccurred

Notify event manager of event occurrence

Syntax int eventOccurred(CSSI_AnalysisEventID id)

Description eventOccurred is used to nitify the event manager of occurrences of an event. The event must have been previously registered through registerEvent. analysisEventControl is used to notify all the CSSI systems of whether an event is enabled or disabled. When disabled, the event need not be notified to the analysis manager. This is not mandatory, but may have performance implications.

Input

id	Id of the event which occurred

Output none

Return Value(s) A non-zero return status indicates successful completion.

pendingError

Gives information about pending error

Syntax
int pendingError(CSSI_ErrorClass&, CSSI_Error&, CSSI_ErrprSource&, CSSI_Name& description) const;

Description
Gives detail about a pending error.

Input
none

Output
none

Return Value(s)
If any error is pending, returns 1.
If no errors pending, returns 0.

stdDescription

Gives standard error message

Syntax
int stdDescription(CSSI_ErrorClass, CSSI_Error, CSSI_ErrorSource&, CSSI_Name& description) const;

Description
Gives standard error message for the defined error classes; otherwise, it gives the default error description

Input
none

Output
none

Return Value(s)
Always returns 1.

dequeueError

Deques an error

Syntax
int dequeueError () const;

Description
Deques an error from the error que.

Input
none

Output
none

Return Value(s)
Always returns 1.

3.6 IErrors

IErrors provides a uniform and centralized mechanism for notifying errors.

The IErrors interface members are:

- hasPendingError
- queueError

hasPending-Error

Determines if error is to be dequeued

Syntax

int hasPendingError()

Description

Determines if there is an error pending to be dequeued. This is usually used to abort operations in the event of pending errors.

Input

none

Output

none

Return Value(s)

A non-zero return value indicates there is some error pending to be dequeued. A zero return value indicates there are no errors to be dequeued.

queueError

Places an error in the error queue

Syntax

int queueError(CSSI_ErrorClass, CSSI_Error, CSSI_ErrorSource, const CSSI_Name description)

Description

Enqueue an error into the error queue. These errors are arranged into error classes. These are target independent. The target or implementation can provide an implementation-specific error number.

While queueing the error, the source of the error can also be indicated. Finally, a string description of the error can also be provided.

Input

class	Class of errors to which this error belongs.
error	An optional implementation-specific error code; this may not be interpretable by the applications.
source	Source of the error; this could be the address of the defaulting instructions, the configuration string in error, etc.
description	An optional string description of the error; this enables a detailed context-specific error.

Output

none

Return Value(s)

none

3.7 IMemory

IMemory provides a memory abstraction of a system. IMemory can be used to access the memory, define and query memory maps, and implement the port-connect-to-file feature.

The IMemory interface members are:

- mapCheckingControl
- mapAdd
- mapDelete
- mapDeleteAll
- getFirstMapInfo
- getNextMapInfo
- Store
- Fetch
- fileConnect
- fileDisconnect
- fileDisconnectAll
- getFirstFileConnectInfo
- getNextFileConnectInfo

mapChecking-Control

Determines if map checking performed on all accesses

Syntax int mapCheckingControl(int check_map)

Description Controls whether the map checking must be performed on all accesses. When access check is disabled, every memory access can be assumed to be valid.

Input check_map A boolean map checking control. When a non-zero return value occurs, it implies that map checks can be performed.

Output none

Return Value(s) A non-zero return value indicates successful completion.

mapAdd — *Add a new memory map*

Syntax	int mapAdd(CSSI_Address start_addr, CSSI_Value byte_length, CSSI_MapAttr)	
Description	Add a new memory map.	
Input	start_addr	The starting address of the map.
	byte_length	The size of the map in bytes.
	attr	Access control definition (read/write access control).
Output	none	
Return Value(s)	A non-zero return value indicates successful completion.	

mapDelete — *Delete a memory map*

Syntax	int mapDelete(CSSI_Address start_addr)	
Description	Attempt to delete an existing memory map.	
Input	start_addr	The starting address of the map to delete.
Output	none	
Return Value(s)	A non-zero return value indicates successful completion.	

mapDeleteAll — *Delete all defined memory map*

Syntax	int mapDeleteAll()
Description	Delete all memory maps that have been defined.
Input	none
Output	none
Return Value(s)	A non-zero return status indicates successful completion.

getFirstMapInfo — *Get information on first defined memory map*

Syntax
int getFirstMapInfo(CSSI_Address& start_addr, CSSI_Value& byte_length, CSSI_MapAttr&)

Description
Obtain information regarding the first defined memory map. Subsequent memory maps are obtained using getNextMapInfo.

Input
none

Output

start_addr	The starting address of the map.
byte_length	The size of the map in bytes.
attr	Access control definition (read/write access control)

Return Value(s)
A non-zero return value indicates that at least one map has been defined.

getNextMapInfo — *Get information on the next defined map*

Syntax
int getNextMapInfo(CSSI_Address& start_addr, CSSI_Value& byte_length, CSSI_MapAttr&)

Description
Obtain information on the next map defined. The sequence can be started using getFirstMapInfo. All subsequent maps after the first map are accessed using getNextMapInfo.

Input

start_addr	The starting address of the map.
byte_length	The size of the map in bytes.
attr	Access control definition (read/write access control)

Output
none

Return Value(s)
A non-zero return value indicates that a map was found.
A zero return value indicates that there are no more maps to query.

store — *Store data*

Syntax int store(CSSI_Address, CSSI_Value)

Description Store data at the indicated address.

Input

address	The address at which the store must be performed.
value	The value to be stored.

Output none

Return Value(s) A non-zero return value indicates successful completion

fetch — *Fetch data*

Syntax int fetch(CSSI_Address, CSSI_Value&)

Description Fetch data from the indicated address.

Input

address	The address from which to fetch.

Output

value	The value stored at the address.

Return Value(s) A non-zero return value indicates successful completion.

fileConnect — *Connect memory range to a file*

Syntax int fileConnect(CSSI_Address start_addr, CSSI_Value byte_length, CSSI_MapAttr, CSSI_Name file_name);

Description Connect a range of memory to a file for input or output. If connected for read, all reads to the specified range of memory are done from the specified file. If connected for write, all writes to the specified range are written to the file and the memory location.

Input

start_addr	Starting address of the memory range.
byte_length	The byte length of the memory range to be connected.
attr	Attributes with which to connect (must be read or write). There may be separate file connects to the same range of memory for reading and writing.
file_name	The file which is to be connected.

Output none

Return Value(s) A non-zero return value indicates successful completion.

fileDisconnect

Disconnect a file from a memory range

Syntax int fileDisconnectAll()

Description Disconnect the file connected to a memory range through a fileConnect.

Input

address	The starting address of the file connect to delete.
attr	Choose the attribute file connect to delete. Read and write connects must be deleted independently.

Output none

Return Value(s) A non-zero return value indicates successful completion.

fileDisconnect-All

Disconnect all files

Syntax int fileDisconnectAll(CSSI_Address, CSSI_MapAttr)

Description Disconnect all files connected through fileConnect.

Input none

Output none

Return Value(s) A non-zero return value indicates successful completion.

getFirstFile-ConnectInfo

Get information on the first port connect

Syntax int getFirstFileConnectInfo(CSSI_Address& start_addr, CSSI_Value& byte_length, CSSI_MapAttr&, CSSI_Name& file_name);

Description Obtain information on the first port connect. Information on subsequent port connects can be obtained using getNextFileConnectInfo.

Input none

Output

start_addr	Starting address of the memory range.
byte_length	Byte length of the memory range to connect
attr	Attributes with which to connect (must be read or write). There may be separate file connects to the same range of memory for reading and writing.
file_name	The file which is to be connected.

Return Value(s) A non-zero return value indicates a successful read.
A zero return value indicates that there are no file connects defined.

getNextFileConnectInfo

Get information on next port connect

Syntax

int getNextFileConnectInfo(CSSI_Address& start_addr, CSSI_Value& byte_length, CSSI_MapAttr&, CSSI_Name& file_name)

Description

Obtain information on the subsequent port connect. Information on the first port connect can be obtained using getFirstFileConnectInfo. All subsequent queries after the first are made using getNextFileConnectInfo.

Input

none

Output

start_addr	Starting address of the memory range.
byte_length	Byte length of the memory range to connect.
attr	Attributes with which to connect(must be read or write). There may be separate file connects to the same range of memory for reading and writing.
file_name	The file which is to be connected.

Return Value(s)

A non-zero return value indicates successful completion.
A zero return value indicates there are no more file connects into which you can look.

3.8 IRegisters

IRegisters provides an abstraction to access the registers in the design. It provides the getFirstRegisterInfo and getNextFileConnectInfo members to query the set of registers. IRegisters also provides the getRegisterValue and setRegisterValue members to access register values for reading or writing.

The IRegisters interface members are:

- getRegisterInfo
- getFirstRegisterInfo
- getNextRegisterInfo
- getRegisterValue
- setRegisterValue

getRegisterInfo

Get information about a register

Syntax int getRegisterInfo(CSSI_Name& long& bit_width, CSSI_RegID& id) const;

Description Obtain information about a particular named register.

Input name — Register name

Output id — Identification for the register. This must be used for all accesses to the register.

bit_width — Size of the register in bits.

Return Value(s) A non-zero return value means a register with the given name has been found.

getFirstRegisterInfo

Get information on the first register

Syntax int getFirstRegisterInfo(CSSI_RegId& ID, long &bit_width, CSSI_Name &name)

Description Obtain informatin on the first available register. Information on subsequent registers can be obtained using getFirstRegisterInfo.

Input none

Output id — Idenfitication for the register. This must be used for all accesses to the register.

bit_width — Size of the register in bits.

name — The name assigned to the register.

Return Value(s) A non-zero return value indicates successful completion.
A zero return value indicates there are no registers defined.

getNextRegisterInfo

Get information on the next register

Syntax int getNextRegisterInfo(CSSI_RegId& ID, long &bit_width, CSSI_Name &name)

Description Obtain information on a subsequent register. The first register is accessed using getFirstRegisterInfo.

Input none

Output

id	Identification for the register. This must be used for all accesses to the register.
bit_width	Size of the register in bits.
name	The name assigned to the register.

Return Value(s) A non-zero return value indicates successful completion.
A zero return value indicates there are no more registers into which you can look.

getRegisterValue

Get the register value

Syntax int getRegisterValue(CSSI_RegId, CSSI_Value&)

Description Obtain the value stored in a register. The value can be set using setRegisterValue. The set of register IDs can be viewed with getFirstRegisterInfo and getNextRegisterInfo.

Input

id	Identification for the register. This can be obtained through the getFirstRegisterInfo and getNextRegisterInfo members.
value	The value held by the register.

Output

value	The value held by the register.

Return Value(s) A non-zero return value indicates successful completion.

setRegisterValue

Store a register value

Syntax int setRegisterValue(cSSI_RegId, CSSI_Value)

Description Store a value to a register. The value can be read using getRegisterValue. The set of register ids can be viewed using getFirstRegisterInfo and getNextRegisterInfo.

Input

	id	Identification for the register. This can be obtained through the getFirstRegisterInfo and getNextRegisterInfo members.
	value	The value held by the register.

Output none

Return Value(s) A non-zero return value indicates successful completion.

3.9 IPinIO

IPinIO provides an interface to support the pin-connect-to-file feature. It provides members to query the set of pins (getFirstPinInfo and getNextPinInfo). It also provides for an interface to connect and disconnect pins to files.

The IPinIO interface members are:

- getFirstPinInfo
- getNextPinInfo
- connect
- disconnect

getFirstPinInfo

Access to information on supported pins

Syntax int getFirstPinInfo(CSSI_PinId& ID, CSSI_Name& name)

Description getFirstPinInfo and getNextPinInfo provide access to the information on all the pins supported. getFirstPinInfo provides information on the first supported pin. Information on subsequent pins can be accessed using getNextPinInfo.

Input none

Output

id	PinID to be used in other accesses to the pin
name	Name associated with the pin

Return Value(s) A non-zero return value indicates successful completion.
A zero return value indicates there are no pins into which you can look.

getNextPinInfo

Access information on supported pins

Syntax int getNextPinInfo(CSSI_PinId& ID, CSSI_Name& name)

Description getFirstPinInfo and getNextPinInfo provide access to the information on all the pins supported. getFirstPinInfo provides information on the first supported pint. Information on subsequent pins can be accessed using getNextPinInfo.

Input none

Output

id	PinID to be used in other accesses to the pin.
name	Name associated with the pin.

Return Value(s) A non-zero return value indicates successful completion.
A zero return value indicates there are no pins into which you can look.

connect

Connect supported pins to a file

Syntax int connect(CSSI_PinID, CSSI_Name file_name)

Description The member connect can be used to connect a supported pin to a file for stimulus. The member disconnect can later be used to disconnect the pin.

Input

id	PinID. The set of PinIDs can be obtained through calls to getFirstPinInfo and getNextPinInfo.
file_name	The name of the system file to which you are supposed to connect.

Output none

Return Value(s) A non-zero return value indicates successful completion.

disconnect

Disconnect pins

Syntax int disconnect(CSSI_PinId);

Description The member disconnect can be used to disconnect an already connected pin.

Input

id	Identification of the pin to be disconnected. The set of pin ids can be obtained through getFirstPinInfo and getNextPinInfo.

Output none

Return Value(s) A non-zero return value indicates successful completion.

3.10 IPortMap

IPortMap provides an interface to the user system for querying the interface values offered. It can be used to ascertain the set of interface values and the notify events supported. It provides the user system with the reference IDs which may later be used in the course of simulation for talking to the core through ICore.

The IPortMap interface members are:

- getFirstValueInfo
- getNextValueInfo
- getValueID
- getFirstEventInfo
- getNextEventInfo
- getEventID

getFirstValue-Info

Access information on supported values

Syntax

int getFirstValueInfo(CSSI_Name& name, CSSI_ValueID& id)

Description

getFirstValueInfo and getNextValueInfo provide access to the information on all the values suppoted. getFirstValueInfo provides information on the first supported value. Information on the subsequent values can later be accessed using getNextValueInfo.

Input

none

Output

name	Name of the value
id	An identificatin of the value to be used in future references.

Return Value(s)

A non-zero return value indicates successful completion.
A zero return value indicates there are no values supported.

getNextValueInfo

Access information on supported values

Syntax

int getNextValueInfo(CSSI_Name& name, CSSI_ValueID& id)

Description

getFirstValueInfo and getNextValueInfo provide access to the information on all the values supported. getFirstValueInfo provides information on the first supported value. Information on the subsequent values can later be accessed using getNextValueInfo.

Input

none

Output

Output	name	Name of the value.
	id	An identification of the value to be used in future references.

Return Value(s)

A non-zero return value indicates successful completion.
A zero return value indicates there are no more values into which you can look.

getValueID

Get the ID for a value

Syntax

int getValueID(CSSI_Name name, CSSI_ValueID&)

Description

Given a string token, getValueID can be sued to obtain the ID for a value to be used to converse with ICore.

Input	name	Name of the value
Output	id	An identification of the value to be used in future references.

Return Value(s)

A non-zero return value indicates successful completion.

getFirstEventInfo

Access information on supported events

Syntax int getFirstEventInfo(CSSI_Name& name, CSSI_EventID& id)

Description getFirstEventInfo and getNextEventInfo provide access to the information on all supported events. getFirstEventInfo provides information on the first supported event. Information on subsequent events can be accessed using getNextEventInfo.

Input none

Output

name	Name of the event.
id	An identification of the event to be used in future references through ICore.

Return Value(s) A non-zero return value indicates a successful read.
A zero return value indicates there were no events.

getNextEventInfo

Access information on supported events

Syntax int getNextEventInfo(CSSI_Name& name, CSSI_EventID& id)

Description getFirstEventInfo and getNExtEventInfo provide access to the information on all supported events. getFirstEventInfo provides information on the first supported event. Information on subsequent events can be accessed using getNextEventInfo.

Input none

Output

name	Name of the event.
id	An identification of the event to be used in future references through ICore.

Return Value(s) A non-zero return value indicates a successful read.
A zero return value indicates there were no more events.

getEventID

Get the ID for an event

Syntax

int getEventID(CSSI_Name name, CSSI_EventID&)

Description

Given a string token, getEventID can be used to obtain the ID for an event to be used to convers with ICore.

Input

name	The name of the event for which you are looking.

Output

id	An identification of the event to be used in future references through ICore.

Return Value(s)

A non-zero return value indicates successful completion.

Chapter 4

CSSI Interface for the ’C27xx Core Simulator

This chapter describes the usage of the CSSI interface as supported by the ’C27xx simulator, and an example of a timer which can be used as a guidline to design a user system for the ’C27xx core simulator. The interface values and the protocols followed by the ’C27xx simulator core are described here.

Topic **Page**

4.1 Interface Registers and Events

The C27xx core simulator interface exports values through registers. The following table shows a list of the registers and their descriptions.

Register	Updated by	Description
CLOCK_CYCLE	Core	Contains clock cycle number
DRAB	Core	Data Read Address Bus
DRDB	Peripheral	Data Read Data Bus
DRDY	Peripheral	Read Done
DROREADY	Peripheral	Read Ready
DWAB	Core	Data Write Address Bus
DWDB	Core	Data Write Data Bus
DWOREADY	Peripheral	Write Ready
EALLOW	Core	1 – if EALLOW instrucion in execute 0 – if EDIS instrucion in execute
IFR	Peripheral	Interrupt Flag Register
NMI	Peripheral	Interrupt Nmi occurred
PAB	Core	Program Address Bus
PC	Core	Program counter
READ_TYPE	Core	8, 16, or 32-bit Read
REASON_FOR_STALL	Core	Register containing reason for stall when pipeline stall occurs.
RSN	Peripheral	Reset through hardware
VMAP	Peripheral	Sets vmap_in to 1 or 0
WRITE_TYPE	Core	8, 16, or 32-bit Write

The following table shows events supported by the C27xx core simulator.

Event	Affected Register	Description
CPU_IDLE		CPU Stalling
CPUSTATLOW		cpuStat signal low
DREAD_REQ	DRAB, READ_TYPE	Data Read Request
DWRITE_REQ	DWAB, DWDB, WRITE_TYPE	Data Write Request
EALLOW		EALLOW/EDIS in execute phase
EXIT		Exit Core Simulator
FETCH_OCCURED		Default Fetch
IACK		IACK instruction in execute stage
IDLE_END		End of Idle instruction
IFSTATLOW		FetchStatus signal low
NEG_CLKEDGE	CLOCK_CYCLE	Negative Clock Edge
PC_DISCONTINUITY		Program Counter Discontinuity
PREAD_REQ	PAB, READ_TYPE	Program Read Request
PWRITE_REQ	PAB,READ_TYPE	Program Write Request
RESET		Reset Core Simulator
VECT_FETCH		Vector Fetch

4.2 Data Memory Access

The simulator models the pipelined accesses of the CPU core. The protocol used is:

1) The read request will be seen by the peripheral at the READ1 stage of the pipeline.
2) The request event will trigger a function in the peripheral.
3) The Peripheral can get the address from DRAB register and the type from the READ_TYPE register.
4) The Peripheral must set DROREADY high one cycle before the data is ready.
5) On the next cycle, the peripheral will set DRDY high, and write the data in DRDB.
6) If the data is in wait-stated memory, it must pull DROREADY low in order to tell the core to wait for the data in the positive edge of the clock.
7) Similarly, the write request is seen by the peripheral in the WRITE1 stage of the pipeline.
8) The peripheral can get address and type, from DWAB and WRITE_TYPE registers.
9) If memory is ready for the request, it must set DWOREADY high. If the memory is slow, it must set DWOREADY low untill it is ready

4.3 Assumptions

1) Peripherals must allocate their own memory.

2) The core gets information about peripheral memory using the getFirstMapInfo function call. It then expects the peripheral to return the start address and length so the core can compute the end address as:

   end = start +length –1

3) If the peripheral memory is slow it is the peripherals responsibility to synchronize by pulling down ready signals by the required number of cycles.

4) For the debugger to be visibile, peripherals must implement fetch and store functions. Additionaly, they must ”map off” using the command file or debugger command window.

5) The core allows seven types of read/write. This information will be available during simulation through read_type and write_type registers, as in the following example:

Code	Description	Value
ALL32BITS	32-bit data read/write	1
ODD16BIT	16-bit data from odd address location	2
ODD16BITMSB	8-bit data from MSB of odd address location	3
EVEN16BIT	16-bit data from even address location	4
EVEN16BITMSB	8-bit data from MSB of even address location	5
ODD16BITLSB	8-bit data from LSB of odd address location	6
EVEN16BITLSB	8-bit data from LSB of even address location	7

4.4 Writing CSSI Modules

1) Define the system

The user defined system can be implemented in C++ by deriving the system from ICssi . In the following example, ICssi defines methods that can be used to simulate a system.

```
class UserSystem : public ICssi

{
...
};
```

2) Write the initialization function

Write an initialization function which will serve as an entry point to the shared object. The ICSSI initialization function must be written in C, and use the following parameters:

- CSSI_Name& _name,
- ICore _core,
- IPortMap& _port_map,
- IConfig& _config,
- IAnalysis& _analysis,
- IErrors& _errors ,

3) Register Events

Register necessary Cssi Events and get Event IDs using the **getEventID** IPort function.

4) Get Register ID

Get register IDs using the **getRegisterInfo** ICore function.

5) Notify Events

Use the **notifyOnEvent** function to register the function names to be called when an event occurs.

6) Parse the configDB pointer to get the start address , length, or any other attribute.

7) Write your own virtual function with **getFirstMapInfo**, which passes start_address, length, and attribute information to the core. Length must be such that an end address can be computed as:

   end = start + length –1;

8) Every function must return 1 for successful completion.

9) For every CSSI function, make sure the function prototype matches that in the header file.

10) Synchronize with the core by setting Ready Signals properly. For example, pull droready low untill the system is ready to send data in the next cycle. Set drdy prior to the data send.

11) Use **getRegisterValue** to read from the register, and use **setRegisterValue** to write into a register.

4.5 Building and Linking the User System With CSSI

Build the user system using the following specifications:

- Development Platform: SUN5
- Language: C++
- Compiler Version: 4.1

Create a .so library. (mysystem.so) In the configuration file, add the following lines:

```
module main;
cssi_modules mysystem;
end main;

module mysystem;
cssi_library "../lib/mysystem.so" //Give full path
init_function newSystem     //An init function in C must
                            //give the entry point
< any other attributes >
end mysystem;
```

4.6 A Timer example

The timer has four registers at addresses 0x100, 0101, 0x102,and 0x103. The register at address 0x100 contains the initial value of the timer. The register at address 0x101 reflects the decremented value each cycle. The interrupt generated cycles from interrupt 1 to interrupt 4. The register ad address 0x102 reflects the cumulative clock, when the interrupt is generated. The register at address 0x103 controls the timer. It has to be set for the timer to be start ticking. Its default value is 0.

Constructing the user system:

The system is constructed using C++.
#include <cssi.h>
class Timer : public ICssi

1) Create a local memory map

   Query the config database and get map information to create a local memory map .

```
if(config.getValue(”INTR_ADDR_START”, start))
{
if(config.getValue(”INTR_ADDR_END” , end ))
{
length = end – start + 1;
mapAdd(start,length,CSSI_MAP_PORT) ;
}
else length = 0x4 ;
}
```

2) Give map information to core

   Implement getFirstMapInfo function to give map information to the core.

```
int Timer::getFirstMapInfo (CSSI_Address& start_addr,
CSSI_Value& byte_length, CSSI_MapAttr& attr)
{
start_addr = MEMMAP_START ;
byte_length = MEMMAP_SIZE ;
                /* send the length in such a way that
                the core can compute an end address as
                end_addr = start_addr+byte_length+1 */
attr = CSSI_MAP_PORT ;
return 1;
}
```

3) Obtain information on the following registers of interrest.

Register names:

CLOCK_CYCLE
DWAB
DWDB
DRAB
DRDB
DWOREADY
DROREADY
DRDY
WRITE_TYPE
READ_TYPE
IFR

```
core.getRegisterInfo(”CLOCK_CYCLE”,bitWidth,cssi_clock-
CycleId) ;
core.getRegisterInfo( ”DWAB”,bitWidth,cssi_dwAddrId);
core.getRegisterInfo( ”DWDB”,bitWidth,cssi_dwValId);
core.getRegisterInfo( ”DRAB”,bitWidth,cssi_drAddrId);
core.getRegisterInfo( ”DRDB”,bitWidth,cssi_drValId);
core.getRegisterInfo( ”DWOREADY”,bitWidth,cssi_wRdyId);
core.getRegisterInfo( ”DROREADY”,bitWidth,cssi_rRdyId);
core.getRegisterInfo( ”DRDY”,bitWidth,cssi_drdyId);
core.getRegisterInfo( ”WRITE_TYPE”,bitWidth,cssi_write-
TypeId);
core.getRegisterInfo( ”READ_TYPE”,bitWidth,cssi_read-
TypeId);
core.getRegisterInfo( ”IFR”,bitWidth,cssi_intrId);
```

4) Obtain event Id’s for the following events :

RESET
NEG_CLKEDGE
DWRITE_REQ
DREAD_REQ
EXIT

```
port_map.getEventID( ”RESET” , resetEventId ) ;
port_map.getEventID( ”NEG_CLKEDGE” , negClkEventId ) ;
port_map.getEventID( ”DWRITE_REQ” , dwEventId ) ;
port_map.getEventID( ”DREAD_REQ” , drEventId ) ;
port_map.getEventID( ”EXIT” , exitEventId ) ;
```

Write functions to be called for each event and notify on event

```
core.notifyOnEvent(resetEventId, this, (CSSI_EventFunc)
reset);
core.notifyOnEvent(negClkEventId, this, (CSSI_Event-
Func) negClkEdge);
core.notifyOnEvent(dwEventId, this, (CSSI_EventFunc)
memWrite);
core.notifyOnEvent(drEventId, this, (CSSI_EventFunc)
memRead);
core.notifyOnEvent(exitEventId, this, (CSSI_EventFunc)
exit);
```

5) Write fetch and store functions to give debugger visibility

6) Interrupt the Core Simulator

Interrupt the Core Simulator by jamming the IFR register. For example:

```
core.setRegisterValue( cssi_intrId , 1<<i)
```

7) Build and Link the system

The user system must build a shared object/dynamic link library(dll). The library is then dynamically linked by the simulator at run time . The configuration file is used to specify which dynamic library to use. An entry point must be chosen for the DLL. The entry point is then used to obtain a link to the user system. For example:

```
extern ”C” ICssi* initTimer(CSSI_Name&  _name,
             ICore&     _core,
             IPortMap&  _port_map,
             IConfig&   _config,
             IAnalysis& _analysis,
             IErrors&   _errors)
{
    return  new Timer( _name , _core , _port_map,
                     _config, _analysis,_errors)  ;
```

In the siminit.cnf file add the following lines:

```
module main ;
   cssi_modules timer ;
end main ;
module timer ;
   cssi_library /tmp/v2.00/documents/example/timer.so ;
   init_function initTimer ;
   intr_addr_start 0x100 ;
   intr_addr_end  0x103;
end timer ;
```

Source code for the timer example

```
timer.h
=======
#include ”timerMem.h”


/*******************************/
/* Initialisation function to be */
/* specified in cnf file         */
/*******************************/
extern ”C” {
ICssi* initTimer(CSSI_Name&  _name,
             ICore&     _core,
             IPortMap&  _port_map,
             IConfig&   _config,
             IAnalysis& _analysis,
             IErrors&   _errors);

};
/***************************************************/
/* Macros used wrt periph memory mapped registers */
/***************************************************/

#define R_INIT  MEMMAP_START
#define R_NEXT  MEMMAP_START+1
#define R_CLK   MEMMAP_START+2
#define R_CNTRL MEMMAP_START+3
#define N_INTRS 4
#define memVal(addr)   (memMap–>val[addr–MEMMAP_START])
#define jamIntr(i) core.setRegisterValue( cssi_intrId ,
1<<i )


class Timer : public ICssi
{
    protected:

        ushort intrJammed ;
        ushort disabled ;
        ushort eventsNotified ;
        TimerMem *memMap ;
        ulong   MEMMAP_START ;
        ulong   MEMMAP_SIZE ;
        /* Declare EventId’s corresponding   */
        /* to each event of timer’s interest */
        CSSI_EventID resetEventId ;
        CSSI_EventID negClkEventId ;
        CSSI_EventID dwEventId ;
        CSSI_EventID drEventId ;
        CSSI_EventID exitEventId ;

        CSSI_RegID cssi_clockCycleId ;
```

```
        CSSI_RegID cssi_dwAddrId ;
        CSSI_RegID cssi_dwValId ;
        CSSI_RegID cssi_drAddrId ;
        CSSI_RegID cssi_drValId ;
        CSSI_RegID cssi_wRdyId ;
        CSSI_RegID cssi_rRdyId ;
        CSSI_RegID cssi_drdyId ;
        CSSI_RegID cssi_intrId ;
        CSSI_RegID cssi_writeTypeId ;
        CSSI_RegID cssi_readTypeId ;

    public:

        Timer( CSSI_Name&  _name, ICore&     _core, IPort-
Map&  _port_map,
            IConfig&   _config, IAnalysis& _analysis, IEr-
rors&   _errors)
            : ICssi( _name , _core , _port_map , _config ,
                     _analysis , _errors)
        {
            initTimer( ) ;
        }

        ~Timer( )
        {
            delete memMap ;
        }

        // Functions corresponding to each event
        int reset( CSSI_EventID event_id ) ;
        int negClkEdge( CSSI_EventID event_id  ) ;
        int memWrite( CSSI_EventID event_id ) ;
        int memRead( CSSI_EventID event_id ) ;
        int exit( CSSI_EventID event_id ) ;

        int fetch(const CSSI_Address& , CSSI_Value& ) ;
        int store(const CSSI_Address& , CSSI_Value ) ;
        int mapAdd(const CSSI_Address&
start_addr,CSSI_Value byte_length, CSSI_M
apAttr);
        int mapCheckingControl(int check_map);
        int mapDeleteAll();
        int getFirstMapInfo(CSSI_Address& start_addr,
CSSI_Value&
                        byte_length, CSSI_MapAttr& attr);


         // timer’s internal functions

        void initTimer( ) ;
        void initTimerMembers() ;
        void notifyEvents() ;
        void denotifyEvents() ;
```

```
        void jamIntrAndUpdate() ;
} ;

timerMem.h
===========
#include ”timerDefs.h”

class TimerMem
{
    public:

        ulong startAddr ;
        ulong endAddr ;
        ushort waitStates ;
        ushort *val ;

        ushort dwReq ;
        ushort drReq ;
        ushort dwoRdy ;
        ushort droRdy ;
        ulong  dwAddr ;
        ulong  drAddr ;
        ushort  drType ;
        ushort dwVal ;
        ushort dataRead ;
        ushort wWaitStates ;
        ushort rWaitStates ;

        TimerMem( ) ;
        TimerMem(uint _start, uint _memMapSize, ushort
_waitStates  ) ;
        ~TimerMem() ;
        int negClkEdge( ) ;
        int memWrite( ulong& addr , ulong& val , ushort&
type) ;
        int memRead( ulong& addr , ushort& type) ;
        ushort decode ( ulong& val, ushort& type) ;
        ulong encode ( ushort& val, ushort& type) ;
} ;

timerDefs.h
=========
#include ”cssi.h”
#include ”cssi_event.h”
#include ”cssi_regs.h”
#include ”cssi_value.h”
#include ”config.h”
#include ”cssi_core.h”

#define ushort unsigned short
#define ulong unsigned long
#define uint unsigned int
```

```
#define ALL32BITS    1
#define ODD16BIT      2
#define ODD16BITMSB   3
#define EVEN16BIT      4
#define EVEN16BITMSB   5
#define ODD16BITLSB   6
#define EVEN16BITLSB   7

timer.cpp
=======
#include ”timer.h”

/* This function is the entry point of timer */
/* This function name has to be specified in */
/* the config file as INIT_FUNCTION          */

ICssi* initTimer(CSSI_Name&  _name,
             ICore&     _core,
             IPortMap&  _port_map,
             IConfig&   _config,
             IAnalysis& _analysis,
             IErrors&   _errors)
{
    return  new Timer( _name , _core , _port_map,
                     _config, _analysis,_errors)  ;
}


int Timer::reset( CSSI_EventID event_id )
{
    initTimerMembers() ;
    return 1 ;
}
int Timer::negClkEdge( CSSI_EventID event_id  )
{
    if (! memVal(R_CNTRL) )
    {
        disabled = 1 ;
//      return 1;
    }

else    if (disabled)
    {
        memVal(R_NEXT) = memVal(R_INIT) ;
        disabled = 0 ;
    }
    else
    {
        if (! memVal(R_NEXT))
            jamIntrAndUpdate() ;
        else
            –– memVal(R_NEXT) ;
    }
```

```
    memMap->negClkEdge() ;


    /***********************************/
    /* If the memory READY strobes are  */
    /* pulled low, then set the corresp */
    /* CSSI register to zero.           */
    /***********************************/
    if (! memMap->dwoRdy )
    {
        core.setRegisterValue( cssi_wRdyId , 0 )  ;
     }
    /***********************************/
    /* If the read READY signal is high */
    /* check if data was ready to be put*/
    /* on the data bus. This is denoted */
    /* by the flag memRead. Thus, the   */
    /* reg DR_VAL is valid only after   */
    /* a READ_REQ event, when the READ_RD*/
    /* is high                           */
   /***********************************/
    if (! memMap->droRdy )
        core.setRegisterValue( cssi_rRdyId , 0 )  ;
    else
        if (memMap->dataRead)
        {
         CSSI_Value val ;

          val = memMval(memMap->drAddr) ;
           val = memMap->encode(val, memMap->drType) ;
            core.setRegisterValue( cssi_drValId , val) ;
            core.setRegisterValue( cssi_drdyId , 1) ;
            memMap->dataRead = 0 ;
        }
    return 1 ;
}


int Timer::memWrite( CSSI_EventID event_id )
{
    CSSI_Value addr , val ,type;
    core.getRegisterValue( cssi_dwAddrId , addr ) ;
    core.getRegisterValue( cssi_dwValId , val ) ;
    core.getRegisterValue( cssi_writeTypeId , type ) ;
    memMap->memWrite( addr , val ,type) ;
    return 1;
}


int Timer::memRead( CSSI_EventID event_id )
{

    CSSI_Value addr ,type;
```

```
    core.getRegisterValue( cssi_drAddrId , addr ) ;
    core.getRegisterValue( cssi_readTypeId , type ) ;
    memMap->memRead( addr , type) ;
    return 1 ;
}
int Timer::exit( CSSI_EventID event_id )
{
    denotifyEvents( ) ;
    return 1 ;
}


int Timer::fetch(const CSSI_Address& addr , CSSI_Value&
val)
{
    if ((addr >= MEMMAP_START) && (addr <= MEM-
MAP_START+MEMMAP_SIZE))
    {
        val = memVal(addr) ;
        return 1;
        }
        else return 0;
}

int Timer::store(const CSSI_Address& addr , CSSI_Value
val)
{
    if ((addr >= MEMMAP_START) && (addr <= MEM-
MAP_START+MEMMAP_SIZE))
    {
        memVal(addr) = val ;
        return 1;
        }
        else return 0;
}

int Timer::mapAdd(const CSSI_Address& start_addr  ,
                  CSSI_Value byte_length, CSSI_MapAttr)
{

  MEMMAP_START = start_addr ;
  MEMMAP_SIZE  = byte_length ;
  return 1;

 }

int Timer::getFirstMapInfo(CSSI_Address& start_addr,
CSSI_Value&
                            byte_length, CSSI_MapAttr&
attr)
{
 start_addr = MEMMAP_START ;
```

```
 byte_length = MEMMAP_SIZE ; /* send the length in such a
way that
                               core can compute end ad-
dress as
                                end_addr =
start_addr+byte_length+1 */
 attr = CSSI_MAP_PORT ;
 return 1;
}
 int Timer::mapDeleteAll()
 {
  MEMMAP_START = 0;
  MEMMAP_SIZE = 0;
  return 1;
  }

  int Timer::mapCheckingControl(int check_map)
  {
   return 1 ;
   }


void Timer::initTimer( )
{
    CSSI_EventID eventId ;

    ulong bitWidth;
    CSSI_Value start , length , end,waitstates;

    if(config.getValue(”INTR_ADDR_START”, start))
    {
      if(config.getValue(”INTR_ADDR_END” , end ))
      {
       length = end – start + 1;
        mapAdd(start,length,CSSI_MAP_PORT) ;
        }
        else length = 0x4 ;
      }
     else start = 0x100 ;
     R_INIT = MEMMAP_START ;

    /* Check if waitstates specified */
    if (!(config.getValue(”WAITSTATES” , waitstates)))
    waitstates = 0 ;


    memMap = new TimerMem(R_INIT,MEMMAP_SIZE,waitstates) ;
    initTimerMembers() ;

    /************************/
    /* Get CSSI register IDs */
    /* CAUTION : Register name*/
    /* should match with core */
```

```
    /************************/
    core.getRegisterInfo(”CLOCK_CYCLE”,bit-
Width,cssi_clockCycleId) ;
    core.getRegisterInfo(”DWAB”,bitWidth,cssi_dwAddrId) ;
    core.getRegisterInfo(”DWDB”,bitWidth,cssi_dwValId) ;
    core.getRegisterInfo( ”DRAB”,bitWidth,cssi_drAddrId) ;
    core.getRegisterInfo( ”DRDB”,bitWidth,cssi_drValId) ;
    core.getRegisterInfo( ”DWOREADY”,bitWidth,cssi_wRdyId)
;
    core.getRegisterInfo( ”DROREADY”,bitWidth,cssi_rRdyId)
;
    core.getRegisterInfo( ”DRDY”,bitWidth,cssi_drdyId) ;
    core.getRegisterInfo( ”WRITE_TYPE”,bitWidth,cssi_wri-
teTypeId) ;
    core.getRegisterInfo( ”READ_TYPE”,bitWidth,cssi_read-
TypeId) ;
    core.getRegisterInfo( ”IFR”,bitWidth,cssi_intrId) ;
    /***********************/
    /* Set notify on events */
    /* CAUTION: Event names */
    /* should match with core*/
    /***********************/
    port_map.getEventID( ”RESET” , resetEventId ) ;
    port_map.getEventID( ”NEG_CLKEDGE” , negClkEventId ) ;
    port_map.getEventID( ”DWRITE_REQ” , dwEventId ) ;
    port_map.getEventID( ”DREAD_REQ” , drEventId ) ;
    port_map.getEventID( ”EXIT” , exitEventId ) ;

    notifyEvents() ;
}

void Timer::initTimerMembers( )
{
    memVal(R_INIT) = 0 ;     // Initial counter Value
    memVal(R_NEXT) = 0 ;     // Cycles to interrupt
    memVal(R_CLK) = 0 ;     // Cycle of last intr
    memVal(R_CNTRL) = 0 ;   // Control. 0=disable 1=en-
abled

    // Once disabled, couter is reloaded when timer is en-
abled
    disabled = 1 ;
    intrJammed = 0 ;     // Last intr jammed
    eventsNotified = 0 ;
}

/* This function tells which function to */
/* call if a particular event occured    */
void Timer::notifyEvents()
{
    if (eventsNotified)
        return ;
    core.notifyOnEvent( resetEventId , this ,
```

```
                            (CSSI_EventFunc) reset ) ;
    core.notifyOnEvent( negClkEventId , this ,
                            (CSSI_EventFunc) negClkEdge ) ;
    core.notifyOnEvent( dwEventId , this ,
                            (CSSI_EventFunc) memWrite ) ;
    core.notifyOnEvent( drEventId , this ,
                            (CSSI_EventFunc) memRead ) ;
    core.notifyOnEvent( exitEventId , this ,
                            (CSSI_EventFunc) exit ) ;
    eventsNotified = 1 ;
}

void Timer::denotifyEvents()
{
    if (!eventsNotified)
        return ;
    core.denotifyOnEvent( resetEventId , this ,
                            (CSSI_EventFunc) reset ) ;
    core.denotifyOnEvent( negClkEventId , this ,
                            (CSSI_EventFunc) negClkEdge ) ;
    core.denotifyOnEvent( dwEventId , this ,
                            (CSSI_EventFunc) memWrite ) ;
    core.denotifyOnEvent( drEventId , this ,
                            (CSSI_EventFunc) memRead ) ;
    core.denotifyOnEvent( exitEventId , this ,
                            (CSSI_EventFunc) exit ) ;
    eventsNotified = 0 ;
}

void Timer::jamIntrAndUpdate( )
{
    CSSI_Value clk ;

    intrJammed = (intrJammed+1) % N_INTRS ;
    jamIntr( intrJammed ) ;
    core.getRegisterValue(cssi_clockCycleId , clk) ;
    memVal(R_CLK) = clk;
    memVal(R_NEXT) = memVal(R_INIT) ;
}


timerMem.cpp
============
#include ”timerMem.h”

TimerMem::TimerMem()
{
}

TimerMem::TimerMem(uint _start,
                  uint _memMapSize ,ushort _waitState )
                  : startAddr(_start),
                    waitStates(_waitState)
```

```
{
    if (_memMapSize<1) _memMapSize = 1 ;
    endAddr = _start + _memMapSize – 1 ;
    val = new ushort[_memMapSize] ;
    memset( val , _memMapSize , 0 ) ;
    dwReq = 0 ;
    drReq = 0 ;
    dwoRdy = 1 ;
    droRdy = 1 ;
    dwAddr = 0 ;
    drAddr = 0 ;
    dwVal = 0 ;
     dataRead = 0 ;
//  waitStates = 0 ;
}

TimerMem::~TimerMem()
{
    if (val) delete val ;
}

int TimerMem::negClkEdge( )
{
    if (dwReq)
    {
        if (!wWaitStates)
        {
        /* Use the value of dwoRdy to modify reg READ_RDY
*/
            dwoRdy = 1 ;
            dwReq = 0 ;
            val[dwAddr] = dwVal ;
        }
        else{
            wWaitStates–– ;}
    }
    else if (drReq)
    {
        if (!rWaitStates)
        {
            droRdy = 1 ;
            drReq = 0 ;
            /* Timer checks if this is high, then puts val
in reg */
            dataRead = 1 ;
        }
        else
            rWaitStates–– ;
    }
    return 1 ;
}
```

```
int TimerMem::memWrite( ulong& addr , ulong& wVal ,ushort&
type)
{
    if (!((addr<startAddr) || (addr>endAddr)))
    {

        dwReq = 1 ;
        dwAddr = addr – startAddr;
        dwVal = decode(wVal,type) ;
        dwoRdy = 0 ;
        wWaitStates = waitStates ;
        return 1 ;
    }

    return 0 ;
}

int TimerMem::memRead( ulong& addr ,ushort& type)
{
    if (!((addr<startAddr) || (addr>endAddr)))
    {
        drReq = 1 ;
        drAddr = addr ;
        drType = type ;
        droRdy = 0 ;
        rWaitStates = waitStates ;
        return 1 ;
    }
    return 0 ;
}

ushort TimerMem::decode ( ulong& val, ushort& type)
{
 ushort value ;
 switch(type)
 {
    case EVEN16BIT : value = val & 0x0000ffff;
                    break ;

    case ODD16BIT: value = (val & 0xffff0000) >> 16 ;
                    break ;

    case EVEN16BITMSB : value = val << 16 ;
                      value = (value & 0xff00) >> 8 ;
                        break ;
    case EVEN16BITLSB : value = val << 16 ;
                      value = value & 0x00ff ;
                        break ;
    case ODD16BITMSB : value = val >> 16 ;
                      value = (value & 0xff00) >> 8 ;
                        break ;
    case ODD16BITLSB : value = val >> 16 ;
                      value = value & 0x00ff ;
```

```
                        break ;

    default         : break ;
    }
    return value ;
}
ulong TimerMem::encode ( ushort& val, ushort& type)
{
 ulong value;
 switch(type)
 {
    case ODD16BIT : value = val;
                    value = value << 16;
                    break ;

    case EVEN16BIT: value = val;
                    break ;

    case EVEN16BITMSB : value = val & 0xff00 ;
                        break ;
    case EVEN16BITLSB : value = val  ;
                        break ;
    case ODD16BITMSB : value = (val & 0xff00) ;
                        value = value << 16 ;
                        break ;
    case ODD16BITLSB : value = (val & 0x00ff)  ;
                        value = value << 16;
                      break ;

    default         : break ;
    }
    return value ;
}
```

Appendix A

Configuration File

The configuration of the system can be stored in the configuration file. The configuration file is read by the simulator at the time of initialization. This sets up the configuration database which can later be accessed through the IConfig interface.

The configuration file provides information related to the system parameters. These are all the parameters which must be known at run time to dynamically construct the complete simulator. These include:

- The system implementations that must be used
- The DLLs that must be used for system implementation
- The initialization functions that must be called for system implementation and for DLLs
- Dynamic parameters are possible for each system. These may be specified through the configuration file.

The configuration file is structured into blocks. Each block constitutes a configuration database. The file can have many blocks. Each block can have sub blocks. Currently, the configuration manager supports the following block types:

- Module
- Memory
- Register
- Constraints
- Attributes

The ’C2700B0 simulator identifies the configuration for the ’C2700B0 core through the ’C27x module. The tag CSSI_MODULES provides the list of modules that must be linked through CSSI to the ’C2700B0 simulator. The simulator then looks for the configuration modules with the same name for information on the individual modules. The simulator looks at the dynamic link library (DLL) information on the module through the CSSI_LIBRARY tag for the initialization function to invoke. Please note that the INIT_FUNCTION must be an exported dynamic link library function.

Here is a sample configuration file for the’C2700B0 simulator:

```
MODULE ’C27x;      //Top level ’C2700B0 simulator descrip-
tion
   CSSI_MODULES MY_CSSI; //Says that we are using MY_CSSI
   CSSI module
END ’C27x
MODULE MY_CSSI;
   CSSI_LIBRARY  mycssi.dll; //look for mycssi.dll for im-
   plementation
   INIT_FUNCTION  my_cssi_init;  //call my_cssi_init to
   create a new system
END MY_CSSI;
```

All blocks are treated identically by the configuration manager. They are named differently for adding readability to the configuration file. Only the syntax is defined by the configuration manager. The interpretation is left to the individual modules that read the configuration.

A block is accessed by its name. For example:

```
MODULE module1;
        ...
END module1;
```

Is accessed by *module1*.

```
   ...
   IConfigPtr module1_db;
   config.getValue(‘‘module1’’, module1_db);
   ...
```

The configurations contained in *module1_db* may be accessed in the local scope. While using module1_db to access values, values in the top level database (config) can be accessed in the global scope.

The configuration database manager automatically constructs a list of all blocks at each level. The following would return a list of values in *module_list*.

```
   IConfigValueList module_list;
   config.getValue(‘‘MODULE’’, module_list);
```

Each block can have some individual configuration items which can be accessed as *{tag, value}* pairs. The values can be numbers, strings, or a list of values.

The configuration file supports C++ sytle comments.

```
config_file:
   {block}+

block:
   module_block|memory_block|register_block|
   constraints_block|attributes_block

module_block:
   MODULE name‘;’block_body END name‘;’

register_block:
   REGISTER name‘;’block_body END name‘;’

memory_block:
   MEMORY name‘;’block_body END name‘;’

constraints_block:
   CONSTRAINTS name‘;’block_body END name‘;’

attributes_block:
   ATTRIBUTES name ‘;’block_body END name‘;’

block_body:
   {config}*

config:
   block|line_config

line_config:
   name value‘;’

value:
   {single_value}+

single_value:
   number|name|path

value_list:
   single_value{‘;’single_value}*
```

CSSI Types

This appendix lists the different CSSI types that can be associates with the various CSSI interfaces. The following types are defined in CSSI. These are defined in the cssi.h file.

- CSSI_Value
- CSSI_ValueID
- CSSI_Name
- CSSI_Tag
- CSSI_EventID
- CSSI_AnalysisEventID
- CSSI_MemPage
- CSSI_Address
- CSSI_CoreType
- CSSI_CoreName
- CSSI_InstanceID
- CSSI_MapAttr
- CSSI_RegID
- CSSI_PinID
- Cssi_CoreID
- IConfigValueList
- ConfigScope
- CSSI_ErrorClass
- CSSI_Error
- CSSI_ErrorSource

Appendix C

Error Classes

CSSI defines the following error classes. For details on using the error classes while enqueuing errors with the error manager, please see the description of IErrors member function queueError, found on page 3-19.

- CSSI_ERR_NONE
- CSSI_ERR_NOT_IMPLEMENTED
- CSSI_ERR_INIT_FAILED
- CSSI_ERR_CFG_BAD_SYNTAX
- CSSI_ERR_CFG_NOFILE
- CSSI_ERR_CFG_INVALID
- CSSI_NO_MEM
- CSSI_ERR_FILE_OPEN_READ
- CSSI_ERR_FILE_OPEN_WRITE
- CSSI_ERR_MEMC_BAD_MEM
- CSSI_ERR_MEMC_NOT_MAPPED
- CSSI_ERR_MEMC_CONFLICT
- CSSI_MEMC_TOO_MANY
- CSSI_ERR_MEMD_NOT_CONNECTED
- CSSI_ERR_PINC_BAD_PIN
- CSSI_ERR_PINC_CONFLICT
- CSSI_ERR_PIND_NOT_CONNECTED
- CSSI_ERR_BRK_CONFLICT
- CSSI_ERR_BRK_BAD_MEM
- CSSI_ERR_BRK_TOO_MANY
- CSSI_ERR_BRK_CLR_NOT_SET
- CSSI_ERR_REG_BAD_ID
- CSSI_ERR_REG_BAD_VALUE
- CSSI_ERR_MEM_BAD_ACCESS
- CSSI_ERR_MEM_BAD_ALIGH
- CSSI_ERR_MEM_BAD_MEM
- CSSI_ERR_MAP_BAD_MEM
- CSSI_ERR_MAP_CONFLICTS
- CSSI_ERR_MAP_TOO_MANY
- CSSI_ERR_BAD_INSTR
- CSSI_ERR_RESOURCE_CONFLICT
- CSSI_ERR_BAD_MODE

Index

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

PRELIMINARY