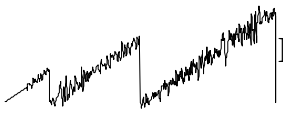
Draft



NEES at CU Boulder

01000110 01001000 01010100

The George E Brown, Jr. Network for Earthquake Engineering Simulation

CU-NEES-06-7

Hybrid Simulation Hardware Interface

by

Dr. Gary Haussmann

Software Engineer

September 2006

Department of Civil Environmental and Architectural Engineering
University of Colorado
UCB 428
Boulder, Colorado 80309-0428

Contents

1 Overview

The Scramble library is a set of software interfaces for use with the Scramnet Shared-Memory Network

2 Installation

The HSHI software is available from NEESForge at neesforge.nees.org/projects/scramble. From there, you can download and compile the software library:

1. You must have the Software Configuration Management tool "Subversion"" to access the source code files for HSHI; Subversion is available from subversion.tigris.org. A Windows client is available at tortoisesvn.tigris.org

2. You can get the relevant files by "checking out" the current source code as the user "anonymous". From the command line, this would be done as:
`svn checkout https://scm.neesforge.nees.org/svn/scramble` if you are using the TortoiseSVN windows client you would instead right click in the windows explorer, select `SVN Checkout` and enter the relevant URL and module "scramble" into the relevant fields.

3. Once you have the source code, you can include the project file scram_hybrid into your solution. Once you modify your project to have a dependency on the scram_hybrid project, you will be able to access functions and classes in the HSHI library.

3 Scramnet Functionality

The Scramnet Shared-Memory Network works by using a shared-memory space on multiple computers. Each computer contains a scramnet card, and is connected to the other computers, typically using fiber optic cable.

When data in the shared-memory space is modified by any computer, the scramnet protocol will communicate that modification to the other computers. Thus, the shared-memory area will appear to contain the same data on every computer in the scramnet network.

The main advantage of sending data between computers using scramnet is guaranteed latency: upon modifying memory one computer, the data is sent to all the other computers within a specific, short, time window. By contrast, many networking protocols such as ethernet do not have any guarantee that the data is sent in a specific amount of time, or even that the data will be sent at all. For hybrid simulations, especially real-time, or scaled-real-time simulations, the transmission and updating of data is typically required to occur on very short guaranteed timeframes.

4 Hybrid Simulation

Hybrid simulation here refers to the simulation of an event by combining both software computation and actual lab mechanisms/measurements. The specific example here is simulating the vibration of a simple structure. In this example, part of the simulation is performed using a Finite-Element model in computer software. Certain quantities of the FE model (such as displacement and/or force) are coupled to an actual physical structure. This coupling of the software model and physical model produce a combined simulation or so-called hybrid of the process or event that we wish to describe, Fig.refScramble-Illo1.

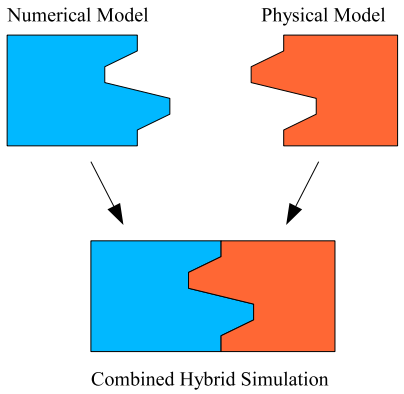


Figure 1: Hybrid simulation combines experimental tests and numerical simulation

5 Using Scramnet for Hybrid Simulation

The scramnet network allows you to couple together multiple computers with guaranteed latency. In a typical hybrid simulation one computer will perform the software computation while another computer drives and measures the physical model. These two computers are coupled together using the scramnet network; more computers can also be connected via scramnet in order to process the simulation results for storage or visualization.

6 The scramnetobject Interface

The scramnetobject interface provides almost direct access to the scramnet hardware. The scramnet hardware is abstracted into a class scramnetobject which provides the following functionality, Fig. 2:

1. Automatic initialization, which resets the card and checks for connectivity errors.
2. Access to read and write the shared-memory area and the control registers (CSRs).
3. Processing and routing of interrupts. You can specify a certain function to be called when an interrupt occurs for a specific memory location.

7 Typical Scramnet Usage

Typical usage consists of several steps:

1. Construct a scramnetobject.
2. Initialize the object using *startscramnet*.
3. Add interrupt handlers for the memory locations for which you want to handle interrupts, using *addMemoryHandler* to specify the function which will be called when an interrupt occurs.
4. Enable interrupt processing for particular memory locations using *enableMemoryInterruptAt*.
5. Enable interrupt processing using *interrupt_enable*.
6. Begin reading and writing shared-memory data using *writeSharedMem* and *readSharedMem*.

An example of typical C++ code:

```
scramnetobject scram;

if (!scram.startscramnet())
{
    aborttest("could not initialize scramnet hardware");
    return 0;
}
```

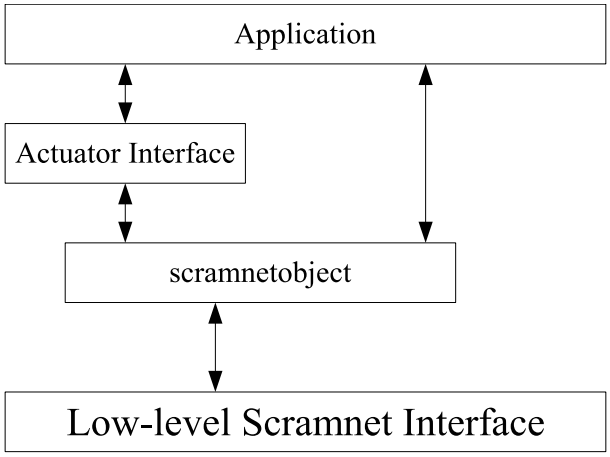


Figure 2: Diagram of interactions between SCRAMnet Software components and your application

```
    // put in an interrupt handler for location 252
    scram.addMemoryHandler(252, InterruptHandler);

    // turn on interrupts for that location and in general
    scram.enableMemoryInterruptAt(252,false,true);
    if (!scram.interrupt_enable())
    {
        aborttest("can't enable interrupts");
        return 0;
    }
```

8 Cross-platform Initializer and State

Internally the *scramnetobject* uses two objects, the *ScramnetInitializer* and *ScramnetState* objects; the relationship between these classes and the *scramnetobject* is shown in Fig. 3. These objects represent internal state and their actual implementation is dependent on the computing platform on which the library is compiled. Therefore, when you see an instance of these classes be aware that you cannot make any assumptions about what data or functionality is stored within those objects, since the class declaration and internal implementation can change drastically when you switch to a different computing platform (for instance, switching from a Windows OS to the ETS embedded system).

9 Initializating and Accessing the Scramnet Object

The scramnetobject is created by constructing it, either explicitly using new/delete or by creating a local variable in your main function.

Typical code for manual creation and deletion:

```
int main(int argc, char *argv[]) {
    scramnetobject *scram = new scramnetobject;
    if (!scram->startscramnet())
    {
        // error code here...
    }
    // code here to access scramnet...
    scram->writeSharedMem(0, datavalue1);

    delete scram;
}

typical code for automatic creation and deletion:

int main(int argc, char * argv[]) {
    scramnetobject scram;
    if (!scram.startscramnet())
    {
```

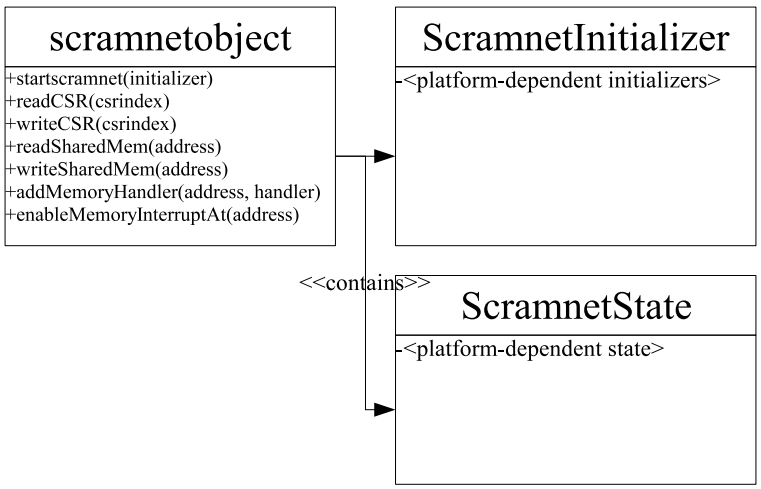


Figure 3: The *scramnetobject*class refers to platform-specific version of the initializer and state classes.

```
        // error!
        return 0;
    }

    // use scramnet object...
    float datavalue1 = scram.readSharedMem(23);

    // the scramnet object will be automatically
    // destroyed here, at the of the end of the function
}
```

If at any point in your program you need to access the scramnet object, you can acquire a pointer to it via the *getScramnetObject()* function:

```
scramnetobject *scram = scramnetobject::getScramnetObject();
scram->writeSharedMem(102, 1.5f);
```

10 Reading and Writing Scramnet Shared-Memory

To read and write from specific locations in the Scramnet shared-memory space, you can use the methods *readSharedMem* and *writeSharedMem*. The first parameter to both functions is the memory address, which represents a location in the Scramnet shared memory space. The address range starts at zero and goes up to the size of the memory installed on the Scramnet hardware. Addresses are in bytes, so if you are reading and writing four-byte data (such as single-precision floating point values) you will be using addresses that are a multiple of four such as 0,4,8, etc.

For reading and writing you can use numbers represented as either floating-point values or integers. Both will be written using the native machine format for these numbers, which is typically big-endian 32-bit integers and IEEE Standard floating-point representations. Note that if, for example, you write out a value as an integer and read it back from the same location as a floating-point number you will usually get garbage or a different value. For each location in Scramnet shared-memory you should decide ahead of time which number type (integer or floating-point) will be stored there.

```
scram->writeSharedMemory(12, 3); // write integer 3 to location 12
scram->writeSharedMemory(24, 3.0f); // write floating-point 3 to
location 24 int x = scram->readSharedMemory(24); // x is an integer,
which does not
            // match the format previously writen to
            // location 24!
            // x will probably have a garbage value in it
```

11 Handling Scramnet Interrupts

The Scramnet hardware can generate interrupts whenever a particular memory location is modified, possibly by another computer on the Scramnet network. This allows you to write functions that respond to memory updates in certain memory locations. For instance, you may be drawing a graph of the data values in Scramnet memory and want to redraw the graph every

time those values change. In this case you can specify a function to call every time the data at a specific location is changed. However, you must also enable interrupts, which is a two-step process. You must enable interrupts for the specific memory location(s) that will trigger an interrupt, and you must turn on the switch that enable or disables ALL interrupts generated by the Scramnet card.

```
void myinterruptfunction(UINT32 memorylocation) {
    // code to update/redraw goes here
    // the memorylocation is passed in so that you can have
    // one function handle interrupts for multiple memory locations
}

//
// .. in your startup code ...
//
scram->addMemoryHandler(104, myfunction);
scram->enableMemoryInterruptAt(104, false, true);
scram->interrupt_enable();
// at this point the function "myfunction" will be called
// every time another computer modifies Scramnet memory
// at location 104
```

12 The Actuator Interface

TBA

13 The FEM Elements

TBA