SystemC

Version 2.0 User's Guide

Update for SystemC 2.0.1

Contents

Chapter 2: Starting with a Simple Example

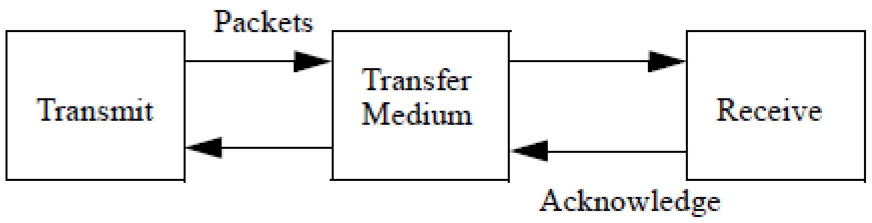
Page 9 - 38

Simplex Data Protocol

The simplex data protocol is a simple data protocol used to transfer data from one device to another in a single direction. (A duplex data protocol would transfer data in both directions.) The simplex data protocol can detect transfer errors, and it can resend data packets to successfully complete the data transfer if errors are detected.

The basic design consists of a transmitter, a receiver, and a model representing the data transfer medium (or channel). The data transfer medium can model wired and wireless networks. It can be a simple or complex model of data and error rates to match the actual physical medium.

A block diagram of the system is shown below:



The transmitter sends data packets to the data transfer medium. The data transfer medium receives those packets, and sends them on to the receiver. The data transfer medium can introduce errors to represent the actual error rate of the physical medium.

The receiver receives the data packets from the data transfer medium and analyzes the data packets for errors. If the data packet has no errors, the receiver generates an acknowledge packet and sends the acknowledgement packet back to the data transfer medium. The data transfer medium receives the acknowledge packet and sends this packet to the transmitter. The data transfer medium can introduce errors when sending the acknowledge packet that causes the acknowledge packet to not be properly received. After the transmitter has received the acknowledge packet for the previously sent data packet, the transmitter sends the next packet. This process continues until all data packets are sent.

This protocol works well for sending data in one direction across a noisy medium.

An example model that implements this system in C/C++ is shown below:

```
frame data;    //global data frame storage for Channel

void transmit(void) {      //Transmits frames to Channel
    int framenum;          // sequence number for frames
    frame s;               // Local frame
    packet buffer;         // Buffer to hold intermediate data
    event_t event;         // Event to trigger actions
    //in transmit

    framenum = 1;          // initialize sequence numbers

    get_data_fromApp(&buffer); // Get initial data from Application
    while (true) {
        s.info = buffer;              // Put data into frame to be sent
        s.seq = framenum;             // Set sequence number of frame
        send_data_toChannel(&s);      // Pass frame to Channel to be sent

        start_timer(s.seq);           // Start timer to wait for acknowledge

        // If timer times out packet was lost
        wait_for_event(&event);       // Wait for events from channel and timer
        if (event==new_frame) {       // Got an event, check which kind
            get_data_fromChannel(s);  // Read frame from channel
            if (s.ack==framenum){     // Did we get the correct acknowledge
                get_data_fromApp(&buffer);
                // Yes, then get more data from application, else send old packet again

                inc(framenum);        // Increase framenum for new frame
            }
        }
    } // while
} // transmit

void receiver(void) {  // Gets frames from channel
    int framenum;      // Scratchpad frame number
    frame r,s;         // Temp frames to save information
    event_t event;     // Event to cause actions in receiver

    framenum = 1;      // Start framenum at 1

    while (true) {
        wait_for_event(&event);       // Wait for data from channel
        if (event==new_frame){        // Event arrived see if it is a frame event
            get_data_fromChannel(r);  // If so get the data from channel
            if (r.seq==framenum) {    // Is this the frame we expect
                send_data_toApp(&r.info); // Yes, then data to application
                inc(framenum);            // Get ready for the next frame
            }
            s.ack = framenum -1;
            // Send back an acknowledge that frame was received properly

            send_data_toChannel(&s);      // Send acknowledge
        }
    }
}

void send_data_toChannel(frame &f) {  // Stores data for channel
    data = f;                         // Copy frame to storage area
}

void get_data_fromChannel(frame &f) { // Gets data from channel
    int i;
    i = rand();    // Generate a random number to cause receive errors

    if ( i > 10 && i < 500) {
        // If the random number is between 10 and 500
        // mess up the sequence number in the packet
        data.seq = 0;
        // This will cause the packet reception to
    }   // fail - protocol should resend packet

    f = data;  // Copy data out of channel
}
```

The C/C++ model contains a transmit function, a receiver function, and two data transfer medium (or channel) functions. These channel functions get data from and put data to the channel (data transfer medium). This description is not a complete implementation of the entire algorithm but only a fragment to show the typical style of a C/C++ model. Some of the model complexity is hidden in the wait_for_event() function calls. These calls are needed to take advantage of a scheduling mechanism built into the operating system, or you can implement a user defined scheduling system. In either case, this is a complex task.

The transmit function, at the beginning of the C/C++ model, has local storage to keep frames and local data, and then it calls the function get_data_fromApp(). This function gets the first piece of data to send from the transmitter to the receiver.

The next statement is a while loop that continuously sends data packets to the receiver. In a real system, this while loop would have a termination condition based on how many packets were sent. However, in this example the designer wants to determine the data rate with varying noise on the channel, rather than sending real packets from one place to another.

The statements in the while loop fill in the data fields of the packet, the sequence number of the packet, and send the packet to the channel. The sequence number is used to uniquely identify the data packet so the correct acknowledge packets can be sent.

After the transmitter sends the packet to the channel, a timer is started. The timer allows the receiver to receive the frame and send back an acknowledge before the the timer times out. If the transmitter does not receive an acknowledge after the timer has timed out, then the transmitter determines that the data frame was not successfully sent, and it will resend the packet.

When the transmitter sends a data packet and starts the timer, the transmitter waits for events to occur. These events can be timeout events from the timer, or they can be new_frame events from the channel. If the event received is a new_frame event, the transmit function gets the frame from the channel and examines the sequence number of the frame to determine if the acknowledge is for the frame just sent. If the sequence number is correct, the frame has been successfully received. Then, the transmitter gets the next piece of data to send and increments the sequence number of the frame. The transmitter sends the data frame and waits again for events.

If the timeout event was received, the test for a new_frame event fails and the transmitter resends the frame. This process continues until the frame is successfully sent. The receiver function also has temporary storage to keep track of local data. At the first invocation, it initializes the frame sequence number to 1, similar to the transmitter function. This allows the two functions to get synchronized.

The receiver function has a main loop that waits only for new_frame events. After a new_frame is received, the receiver gets the frame from the channel and analyzes the contents.

If the sequence number of the frame matches the framenum variable, then the expected frame was sent and received properly. The receiver increments the framenum to get ready for the next frame.

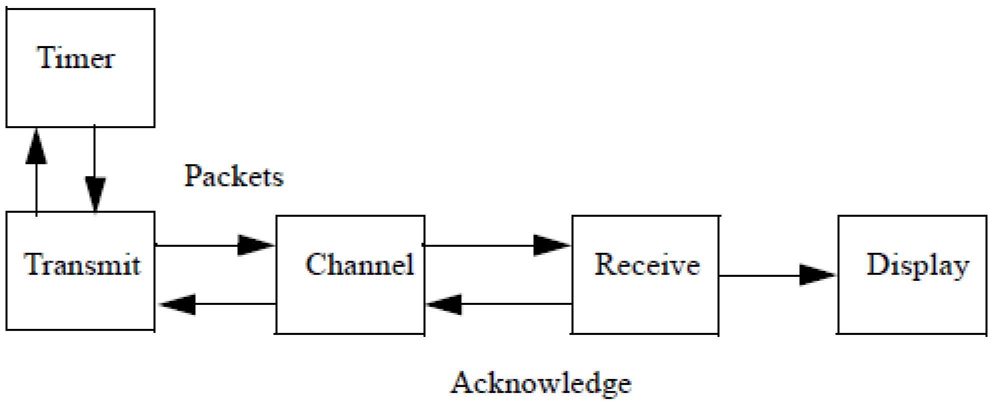
The receiver generates an acknowledgement frame containing the sequence number minus 1. Because the frame sequence number is already incremented, the acknowledgement frame needs to subtract 1 from the framenum to acknowledge the last frame received. If the wrong frame was received, the acknowledgement contains an improper sequence number to inform the transmitter that the proper frame was not correctly transmitted.

The last two functions in the C/C++ model send data to the channel and get data from the channel. These two functions are very simple in this model, but they could be complex, depending on the factors to be analyzed. Function send_data_to_channel() simply copies the received frame from the transmitter to a local variable. Function get_data_from_channel() reads the data from the local variable, but adds noise to the data so some frames are not passed intact. Noise is generated by a random number generator that selectively zeroes the sequence number of the frame. The amount of noise is dependent on the total range of the sequence numbers and the range of numbers that cause the sequence number to be zeroed.

Using the C/C++ model, the designer can analyze the total data rate, effective data rate, error recovery, error recovery time, and numerous other factors. The designer can modify parameters such as frame rate size, error range size, data packet size, timer length to verify that the protocol works, and analyze the effects of these parameters.

SystemC Model

Using SystemC the designer can design at a high level of abstraction using C++ high level techniques, and refine the design down to a level that allows hardware or software implementation. The block diagram for the SystemC implementation is shown below:

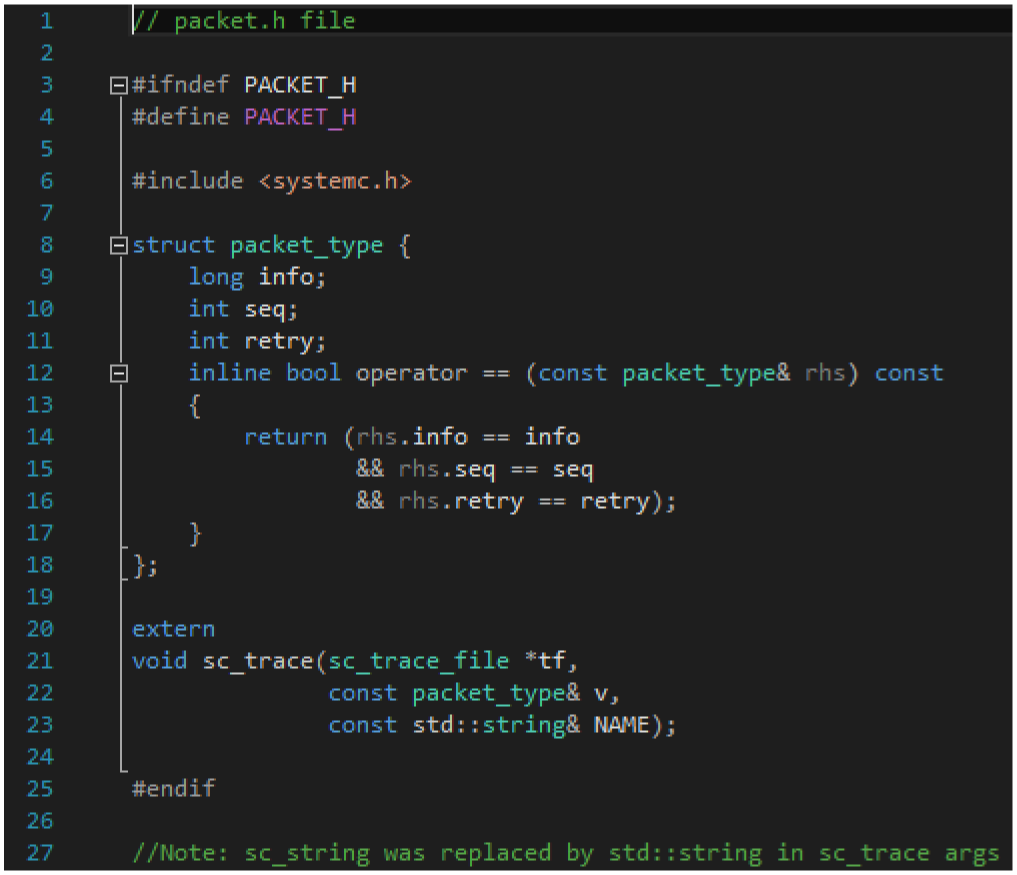


This block diagram is slightly different than the C model because the SystemC implementation is a more complete model. The SystemC description contains the transmit block, the receiver block, the channel block, a timer block, and a display block. The transmit, the receiver, and the channel blocks are the same as the C++ implementation. The display block emulates the application interface on the receiver side and the timer block generates timeout events. Packets are generated by a function in the transmit block and are sent through the channel to the receiver block. The receiver block sends data to the display block where the data is displayed.

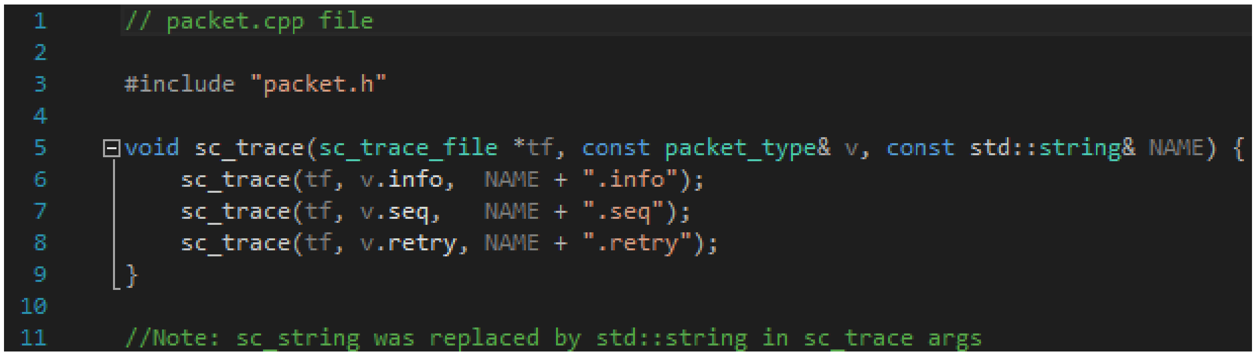
Let’s examine each block to see the descriptions and how they work.

User Defined Packet Type

Before we describe the blocks, we need to look at the underlying packet data structure that passes data from module to module. The packet type is defined by a struct as shown below:



```cpp
// packet.h file

#ifndef PACKET_H
#define PACKET_H

#include <systemc.h>

struct packet_type {
    long info;
    int seq;
    int retry;
    inline bool operator == (const packet_type& rhs) const
    {
        return (rhs.info == info
                && rhs.seq == seq
                && rhs.retry == retry);
    }
};

extern
void sc_trace(sc_trace_file *tf,
              const packet_type& v,
              const std::string& NAME);

#endif

//Note: sc_string was replaced by std::string in sc_trace args
```



```cpp
// packet.cpp file

#include "packet.h"

void sc_trace(sc_trace_file *tf, const packet_type& v, const std::string& NAME) {
    sc_trace(tf, v.info,  NAME + ".info");
    sc_trace(tf, v.seq,   NAME + ".seq");
    sc_trace(tf, v.retry, NAME + ".retry");
}

//Note: sc_string was replaced by std::string in sc_trace args
```

The struct has three fields, info, seq, and retry. Field info carries the data sent in the packet. The goal of this simulation is to measure the protocol behavior with respect to noise, not the data transfer characteristics. Therefore, the info field for data is of type long. Future versions of this data packet type could use a struct type for the data.

The second field is named seq and represents the sequence number assigned to this packet. For better error handling, this number will uniquely identify the packet during data transfers.

The third field in the packet is the retry field. This field contains the number of times the packet has been sent.

Other constructs in the packet.h and packet.cpp files will be discussed later.

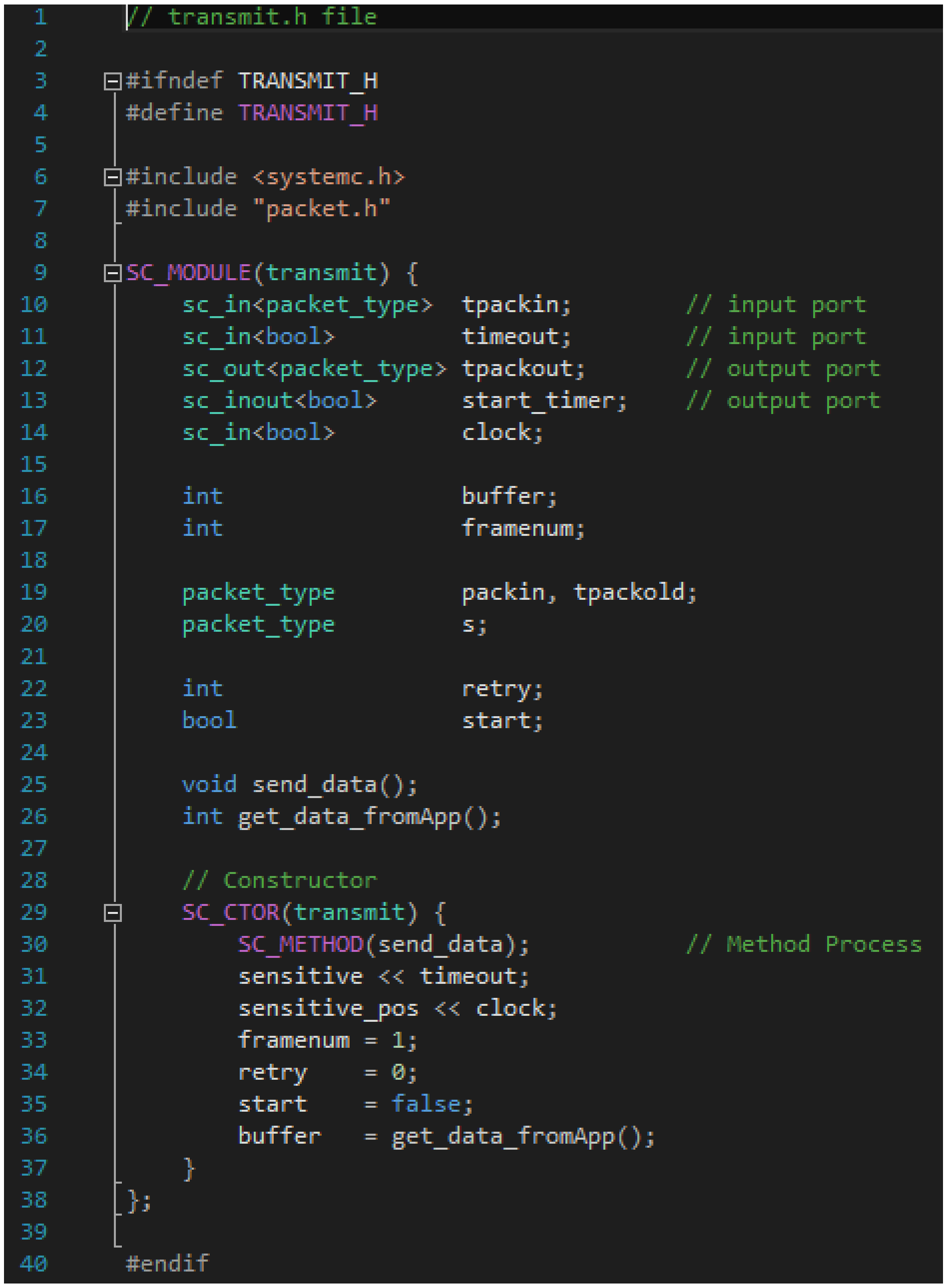
Let’s now take a look at the first block, the transmit block.

Transmit Module

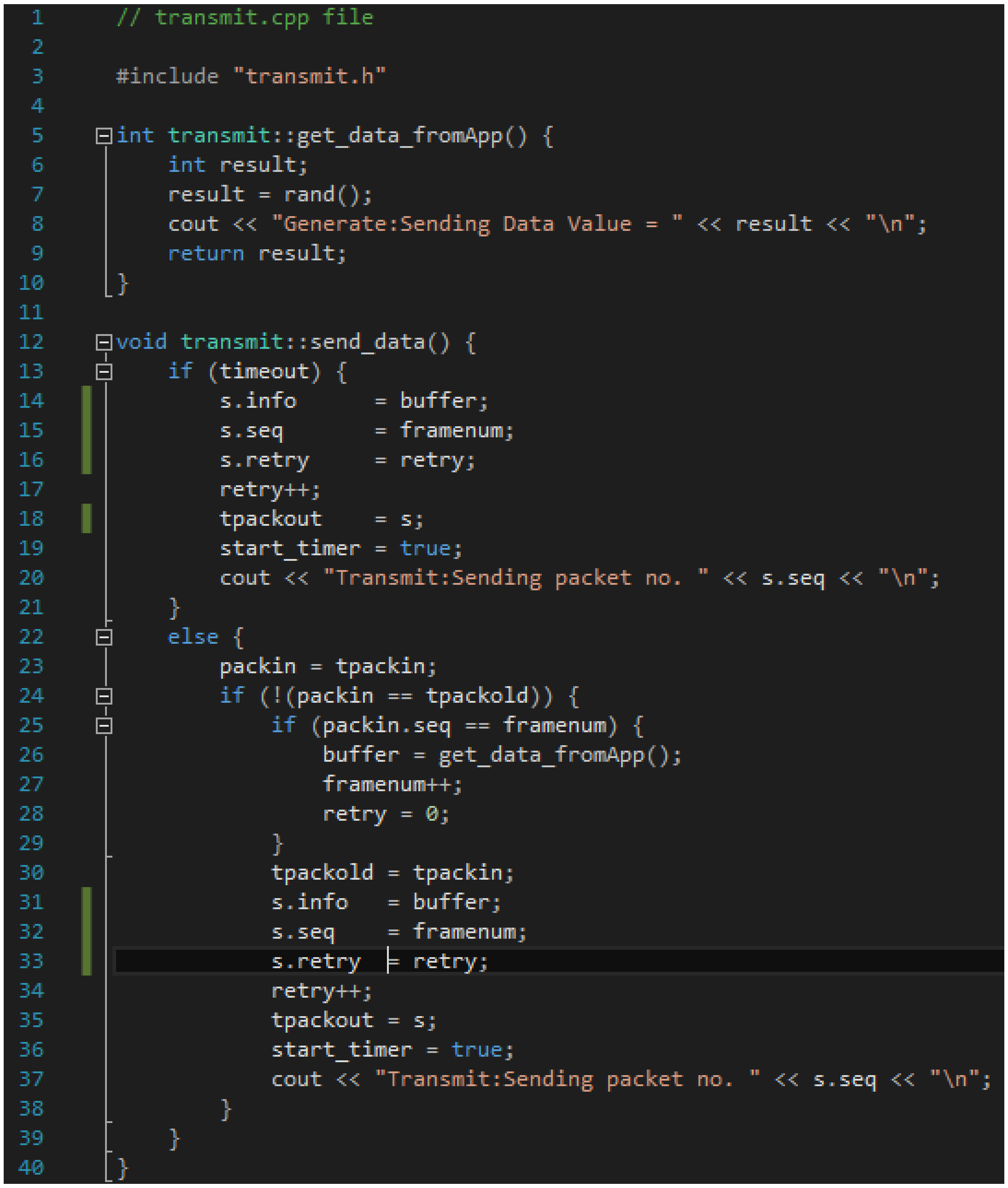
Notice that the transmit module includes the packet.h file which includes systemc. h. The systemc.h file gives the design access to all of the SystemC class methods and members. The packet.h file gives the design access to the packet definition and methods associated with the packet.

Note: In C++, function members are similar to C functions and data members are similar to C variables.

The SystemC description of the transmit module, described in the sections that follow, is shown below:



```
// transmit.h file

#ifndef TRANSMIT_H
#define TRANSMIT_H

#include <systemc.h>
#include "packet.h"

SC_MODULE(transmit) {
    sc_in<packet_type>  tpackin;        // input port
    sc_in<bool>         timeout;        // input port
    sc_out<packet_type> tpackout;       // output port
    sc_inout<bool>      start_timer;    // output port
    sc_in<bool>         clock;

    int                 buffer;
    int                 framenum;

    packet_type         packin, tpackold;
    packet_type         s;

    int                 retry;
    bool                start;

    void send_data();
    int get_data_fromApp();

    // Constructor
    SC_CTOR(transmit) {
        SC_METHOD(send_data);           // Method Process
        sensitive << timeout;
        sensitive_pos << clock;
        framenum = 1;
        retry    = 0;
        start    = false;
        buffer   = get_data_fromApp();
    }
};

#endif
```



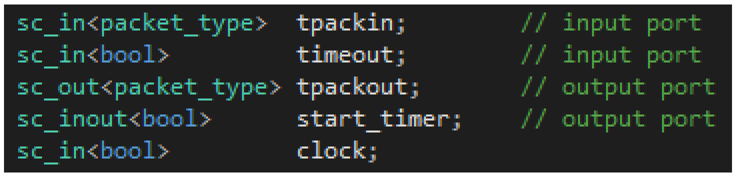
```
// transmit.cpp file

#include "transmit.h"

int transmit::get_data_fromApp() {
    int result;
    result = rand();
    cout << "Generate:Sending Data Value = " << result << "\n";
    return result;
}

void transmit::send_data() {
    if (timeout) {
        s.info      = buffer;
        s.seq       = framenum;
        s.retry     = retry;
        retry++;
        tpackout    = s;
        start_timer = true;
        cout << "Transmit:Sending packet no. " << s.seq << "\n";
    }
    else {
        packin = tpackin;
        if (!(packin == tpackold)) {
            if (packin.seq == framenum) {
                buffer = get_data_fromApp();
                framenum++;
                retry = 0;
            }
            tpackold = tpackin;
            s.info   = buffer;
            s.seq    = framenum;
            s.retry  = retry;
            retry++;
            tpackout = s;
            start_timer = true;
            cout << "Transmit:Sending packet no. " << s.seq << "\n";
        }
    }
}
```

Module

A module is the basic container object for SystemC. Modules include ports, constructors, data members, and function members. A module starts with the macro SC_MODULE and ends with a closing brace. A large design will typically be divided into a number of modules that represent logical areas of functionality of the design.

Ports

Module transmit has three input, one output, and one inout ports as shown below:



```
sc_in<packet_type>  tpackin;      // input port
sc_in<bool>         timeout;      // input port
sc_out<packet_type> tpackout;     // output port
sc_inout<bool>      start_timer;  // output port
sc_in<bool>         clock;
```

Port tpackin is used to receive acknowledgement packets from the channel. Port timeout is used to receive the timeout signal from the timer module and lets the transmit module know that the acknowledge packet was not received before the timer times out. The clock port is used to synchronize the different modules together so that events happen in the correct order.

Output port tpackout is the port that module transmit uses to send packets to the channel. Inout port start_timer is used by the transmit module to start the timer after a packet has been sent to the channel.

Data and Function Members

After the port statements, local data members used within the module are declared.

The function members send_data() and get_data_fromApp() are declared in the transmit.h file and implemented in the transmit.cpp file. This is the standard way to describe functionality in C++ and SystemC.

Constructor

The module constructor identifies process send_data() as an SC_METHOD process, which is sensitive to clock and timeout. The constructor also initializes the variables used in the module. This is an important step. In HDL languages such as VHDL and Verilog, all processes are executed once at the beginning of simulation to initialize variable and signal values. In SystemC the constructor of a module is called at initialization, and all initialization that needs to be performed is defined in the constructor.

The constructor initializes variable framenum to 1, which is used as the sequence number in all packets. Variable retry is initialized to 0, and variable start is initialized to false. Then the buffer, used to hold data is initialized by getting the first piece of data.

Implementation of Methods

Let’s now take a look at the transmit.cpp file. This file implements the two methods declared in the header file. One method is a process, i.e. it executes concurrently with all other processes, the other is not.

Method get_data_fromApp() is a local method that generates a new piece of data to send across the channel. In the implementation you can see that this method calls the random number generator rand() to generate a new data value to send. Because we are validating the effects of noise on the protocol, the data values sent are not important at this stage of the design.

The method send_data() is a process because it is declared as such in the constructor for module transmit.The process checks first to see if timeout is true. Timeout will be true when the timer has completed.

Next the process checks to see if the current value of port tpackin is equivalent to the old value. This check is used to see if an acknowledgement packet was received from the channel.

If the values differ an acknowledgement packet has been received from the channel. Notice that tpackin was copied to local variable packin. Using a local variable allows you to access to the packet fields that cannot be accessed directly from the port.

The sequence number of the packet is checked against the last sent packet to see if they match. If they match, a correct acknowledgement was received and the next piece of data can be sent. The buffer will be filled with the next piece of data and the framenum is incremented. Field retry is also reset to 0, which is the initial value.

If no acknowledgement packet was received, the process was triggered by an event from the timeout port. This means the timer block completed its count or "timed out". If a timeout event occurs, that means the packet was sent, but no acknowledge was received. In this case, the packet must be transmitted again.

A local packet named s has all of its fields filled with the new data values to be sent, and it is assigned to tpackout. Notice that retry is incremented each time the packet is sent. This number is required to ensure the uniqueness of each packet.

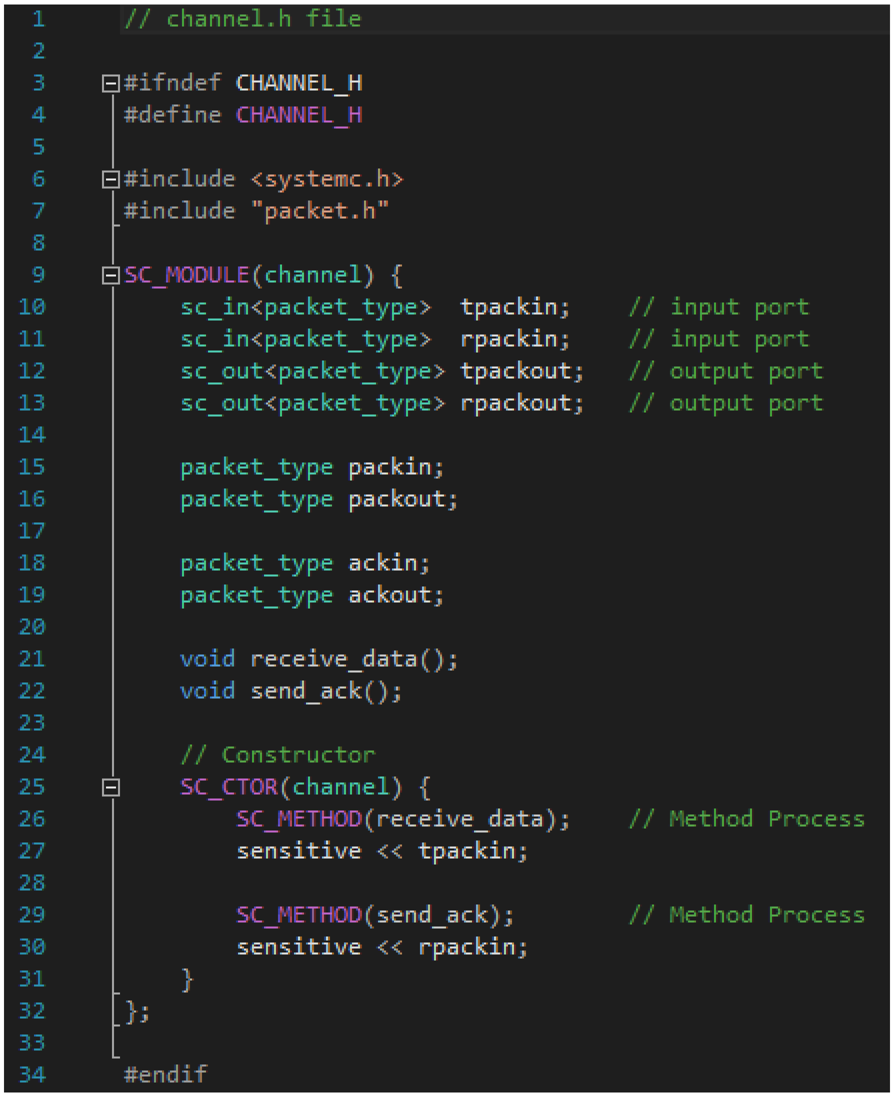
Whenever a packet is written, the timer is started by the transmit process by setting the start_timer signal to true.

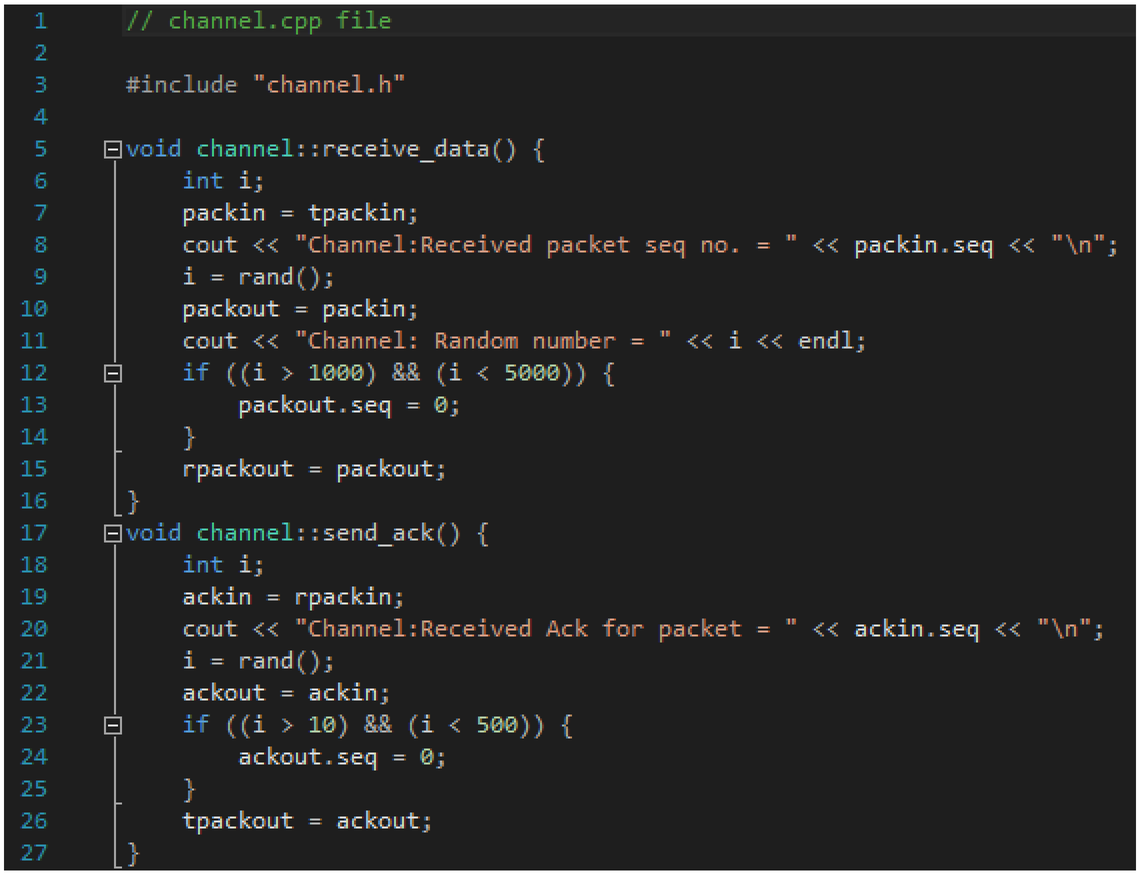
In summary, the transmit module sends a new packet to the channel. The timer is started to keep track of how long ago the packet was sent to the channel. If the acknowledge packet does not return before the timer times out, then the packet or acknowledge were lost and the packet needs to be transmitted again.

Channel Module

The channel module accepts packets from the transmitter and passes them to the receiver. The channel also accepts acknowledge packets from the receiver to send back to the transmit module. The channel adds some noise to the transmission of packets to model the behavior of the transmission medium. This causes the packets to fail to be properly received at the receiver module, and acknowledge packets fail to get back to the transmit module. The amount of noise added is dependent on the type of transfer medium being modeled.

The SystemC channel description is shown below:


```
// channel.h file

#ifndef CHANNEL_H
#define CHANNEL_H

#include <systemc.h>
#include "packet.h"

SC_MODULE(channel) {
    sc_in<packet_type>  tpackin;    // input port
    sc_in<packet_type>  rpackin;    // input port
    sc_out<packet_type> tpackout;   // output port
    sc_out<packet_type> rpackout;   // output port

    packet_type packin;
    packet_type packout;

    packet_type ackin;
    packet_type ackout;

    void receive_data();
    void send_ack();

    // Constructor
    SC_CTOR(channel) {
        SC_METHOD(receive_data);    // Method Process
        sensitive << tpackin;

        SC_METHOD(send_ack);        // Method Process
        sensitive << rpackin;
    }
};

#endif
```



```cpp
// channel.cpp file

#include "channel.h"

void channel::receive_data() {
    int i;
    packin = tpackin;
    cout << "Channel:Received packet seq no. = " << packin.seq << "\n";
    i = rand();
    packout = packin;
    cout << "Channel: Random number = " << i << endl;
    if ((i > 1000) && (i < 5000)) {
        packout.seq = 0;
    }
    rpackout = packout;
}
void channel::send_ack() {
    int i;
    ackin = rpackin;
    cout << "Channel:Received Ack for packet = " << ackin.seq << "\n";
    i = rand();
    ackout = ackin;
    if ((i > 10) && (i < 500)) {
        ackout.seq = 0;
    }
    tpackout = ackout;
}
```

Ports

The channel description contains four ports, two input ports and two output ports. Port tpackin accepts packets from the transmit module and port rpackin accepts acknowledge packets from the receiver. Port tpackout sends acknowledge packets to the transmit module and port rpackout sends data packets to the receiver.

Data and Function Members

Notice the four local packet variable declarations to hold the values of the packet ports. Local variables are necessary so you can access the packet internal data fields.

The channel description has two processes. Process receive_data() is used to get data from the transmit module and pass the data to the receiver module. Process send_ack() gets acknowledge packets from the receiver module and sends them to the transmit module.

Process receive_data is sensitive to events on port tpackin. Thus when a new packet is sent from the transmit module, process receive_data() is invoked and analyzes the packet. Process send_ack() is sensitive to events on port rpackin. When the receiver module sends an acknowledge packet to the channel module, the value of port rpackin is updated and causes process send_ack to be invoked.

Constructor

In the channel constructor, we can see that both processes are SC_METHOD processes.

Implementation of Methods

Let’s take a closer look at process receive_data. The first step is to copy tpackin to a local variable so that the packet fields can be accessed. A message is printed for debugging purposes. A random number is generated to add noise to the channel. The packet is assigned to the output packet and another debugging message is printed displaying the random number value. Finally, an if statement determines whether the packet passes through as received, or if the packet is altered by adding noise.

If the random number is within the specified range, the sequence number of the packet is set to 0. This means the packet was corrupted. The last two statements of the process receive_data copies the altered or unaltered packet to output port rpackout

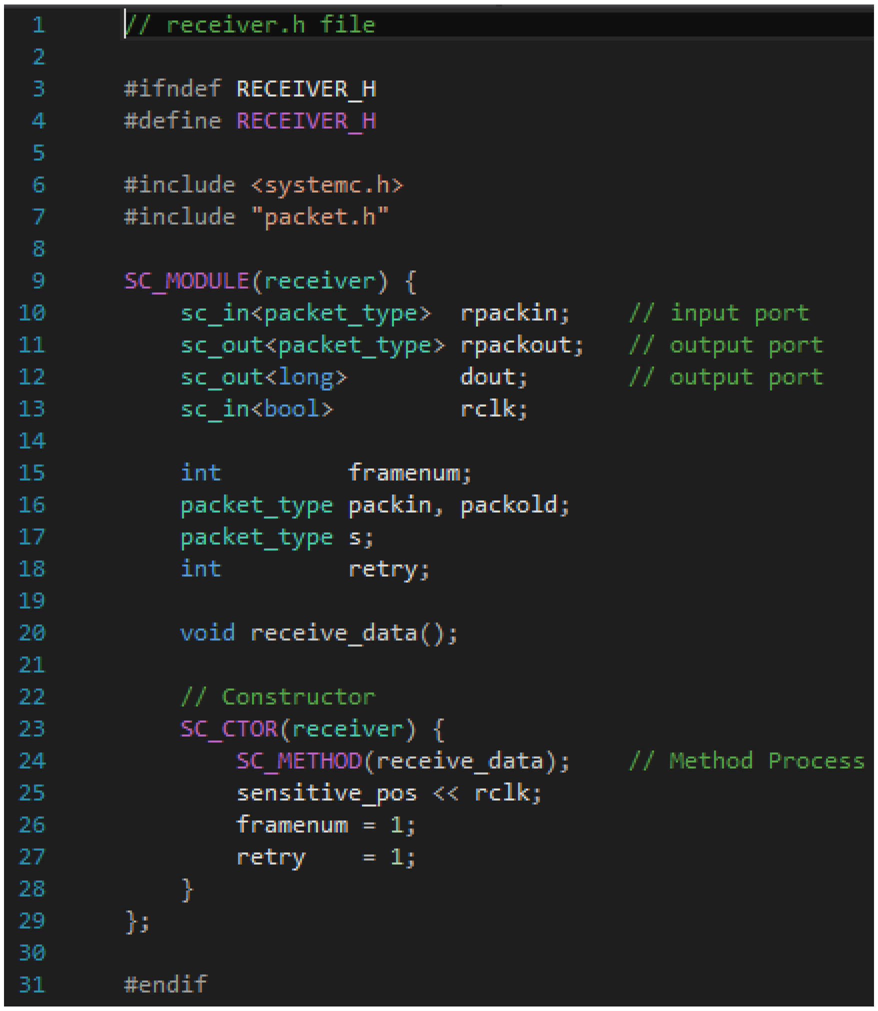
You can modify the range of random numbers generated and the range of numbers that modify the packet sequence number to control the amount of noise injected.

The send_ack process, triggered by events on port rpackin, is very similar to the receive_data process. It assigns rpackin to the local packet ackin so the fields of the packet can be examined. Next, a debug message is written, and a random number representing the noise in the channel for acknowledgements is generated. This process also uses a specified range to modify the sequence number field of the packet. Finally, the packet is assigned to tpackout where it will be passed to the transmit module.

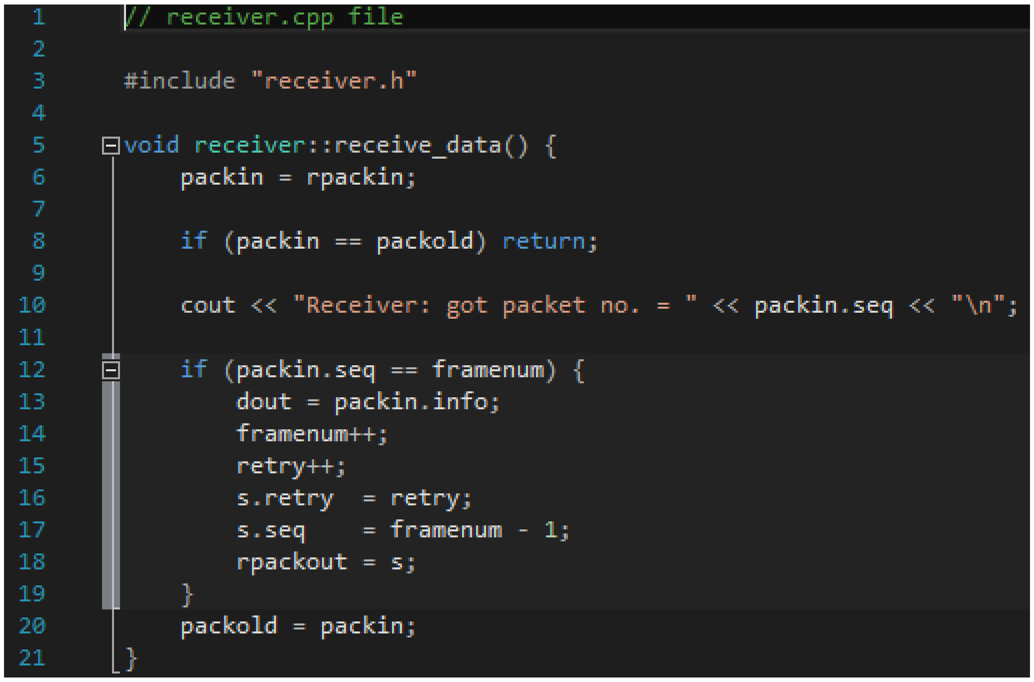
Receiver Module

The receiver module accepts packets from the channel module and passes the data received to the virtual application. In this design the virtual application is a display, modeled in the display module. When the receiver module successfully receives a packet, it send an acknowledgement packet back to the transmit module. Incoming packet sequence numbers are compared with an internal counter to ensure the correct packets are being transmitted.

The receiver module is shown below:



```cpp
// receiver.h file

#ifndef RECEIVER_H
#define RECEIVER_H

#include <systemc.h>
#include "packet.h"

SC_MODULE(receiver) {
    sc_in<packet_type>  rpackin;    // input port
    sc_out<packet_type> rpackout;   // output port
    sc_out<long>        dout;       // output port
    sc_in<bool>         rclk;

    int         framenum;
    packet_type packin, packold;
    packet_type s;
    int         retry;

    void receive_data();

    // Constructor
    SC_CTOR(receiver) {
        SC_METHOD(receive_data);    // Method Process
        sensitive_pos << rclk;
        framenum = 1;
        retry    = 1;
    }
};

#endif
```



```cpp
// receiver.cpp file

#include "receiver.h"

void receiver::receive_data() {
    packin = rpackin;

    if (packin == packold) return;

    cout << "Receiver: got packet no. = " << packin.seq << "\n";

    if (packin.seq == framenum) {
        dout = packin.info;
        framenum++;
        retry++;
        s.retry  = retry;
        s.seq    = framenum - 1;
        rpackout = s;
    }
    packold = packin;
}
```

Ports

The receiver module has four ports, two input port, and two output ports. Input port rpackin accepts packets from the channel. Input rclk is the receiver block clock signal. Output port rpackout is used to send acknowledge packets to the channel where they may be passed to the transmit module. Output port dout transfers the data value contained in the packet to the display module for printing.

Constructor

The receiver module contains one SC_METHOD process named receive_data, which is sensitive to positive edge transitions on input port rclk. Notice in the receiver module constructor that variable framenum is initialized to 1. Module transmit also initializes a framenum variable to 1, so both transmit and receiver are synchronized at the start of packet transfer.

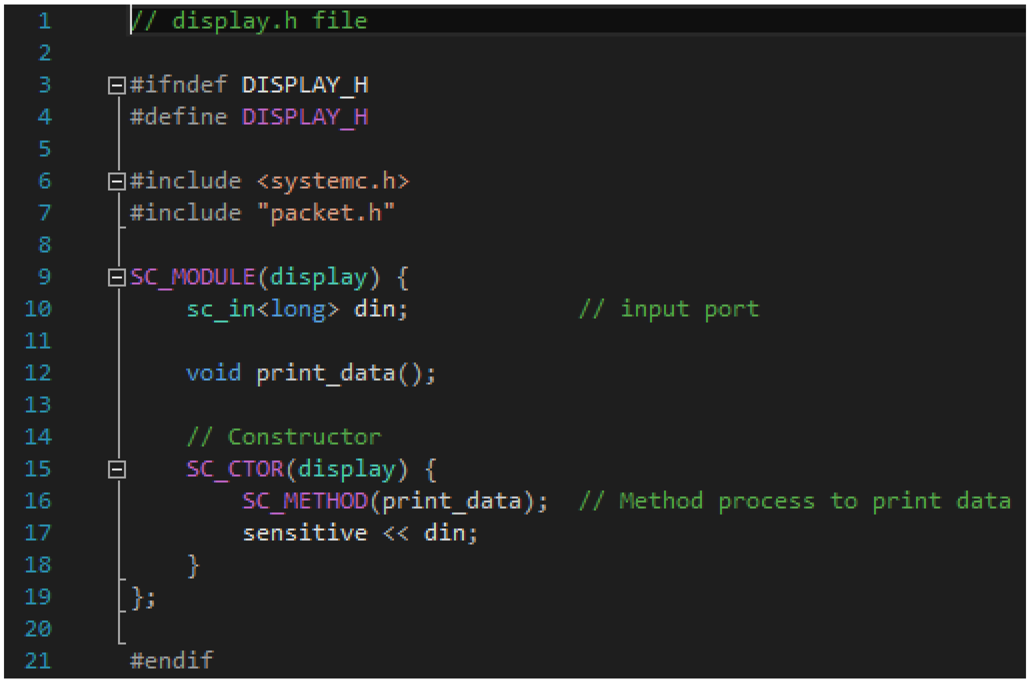
Implementation of Methods

As we have already seen, process receive_data is invoked when a new packet arrives on the rpackin port. The first step in the process is to copy rpackin to the local variable packin for packet field access. The receiver block, like the transmit block, compares the new packet value with the old packet value to determine if a new packet has been received. A debug message is printed and the process checks to see if the sequence number of the incoming packet matches the expected framenum. If so, the packet data is placed on the dout port where it will be sent to the display module. Next, the framenum variable is incremented to reflect the next framenum expected.

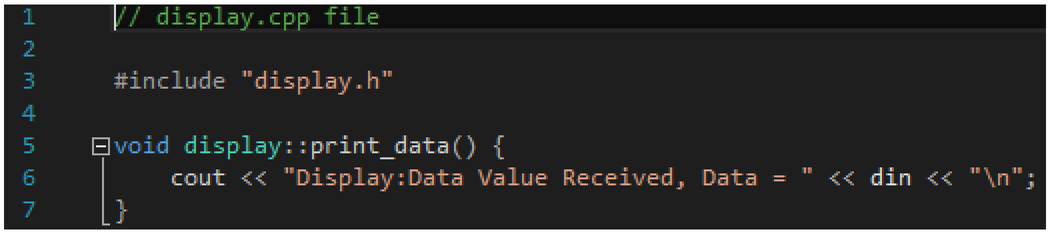
If a packet is successfully received an acknowledge packet needs to be sent back to the transmit module. A local packet named s has its sequence number filled in with framenum. After the sequence number field is updated, packet s is assigned to port rpackout.

Display Module

The display module is used to format and display the packet data received by the receiver module. In this example the data is a very simple type long value. This makes the display module very simple. However, as the simplex protocol is more completely implemented, the data being sent could grow in complexity to the point that more complex display formatting is needed. The display module is shown below:



```cpp
// display.h file

#ifndef DISPLAY_H
#define DISPLAY_H

#include <systemc.h>
#include "packet.h"

SC_MODULE(display) {
    sc_in<long> din;            // input port

    void print_data();

    // Constructor
    SC_CTOR(display) {
        SC_METHOD(print_data);  // Method process to print data
        sensitive << din;
    }
};

#endif
```



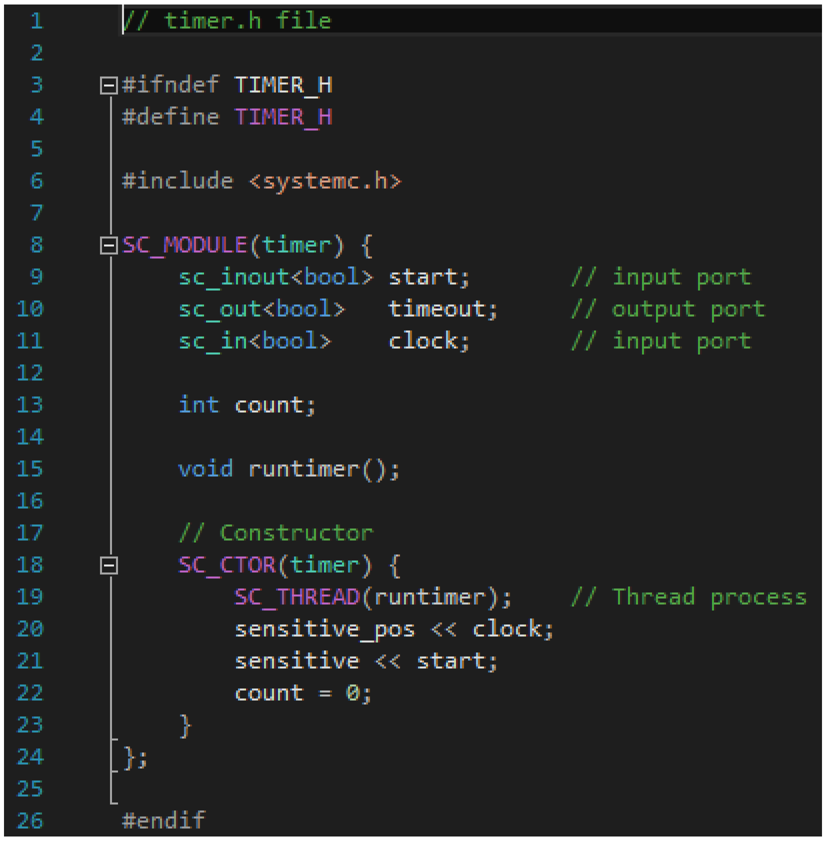
```cpp
// display.cpp file

#include "display.h"

void display::print_data() {
    cout << "Display:Data Value Received, Data = " << din << "\n";
}
```

This module only has one input port named din. It accepts data values from the receiver module. When a new value is received, process print_data is invoked and writes the new data item to the output stream.

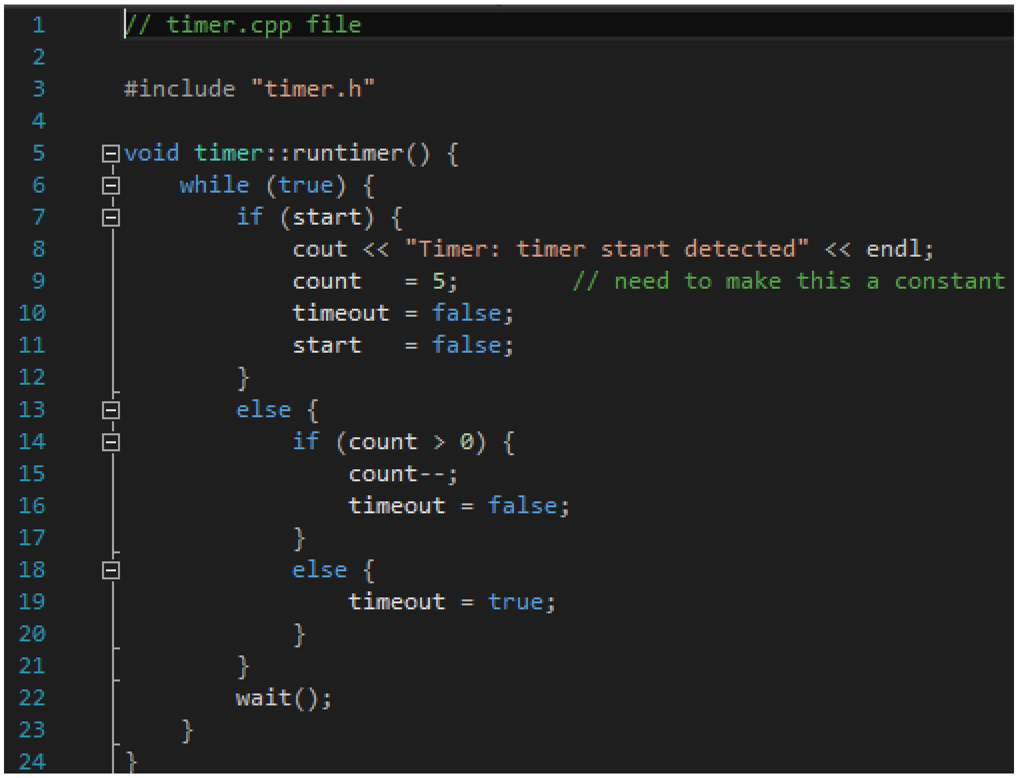
Timer Module

The timer module implements a timer for packet retransmission. The delay allows a packet to propagate to the receiver and the acknowledge to propagate back before a retransmit occurs. Setting the delay properly is a key factor in determining the maximum data rate in a noisy environment. Without the timer, the transmitter would not know when to retransmit a packet that was lost in transmission. The delay value is a parameter that can easily be modified to find the optimum value.

The timer module is shown below:



```
// timer.h file

#ifndef TIMER_H
#define TIMER_H

#include <systemc.h>

SC_MODULE(timer) {
    sc_inout<bool> start;       // input port
    sc_out<bool>   timeout;     // output port
    sc_in<bool>    clock;       // input port

    int count;

    void runtimer();

    // Constructor
    SC_CTOR(timer) {
        SC_THREAD(runtimer);    // Thread process
        sensitive_pos << clock;
        sensitive << start;
        count = 0;
    }
};

#endif
```



```
// timer.cpp file

#include "timer.h"

void timer::runtimer() {
    while (true) {
        if (start) {
            cout << "Timer: timer start detected" << endl;
            count   = 5;        // need to make this a constant
            timeout = false;
            start   = false;
        }
        else {
            if (count > 0) {
                count--;
                timeout = false;
            }
            else {
                timeout = true;
            }
        }
        wait();
    }
}
```

Ports

The timer module has one input port, one inout port, and one output port. Port start is an inout port of type bool. The transmit module activates the timer by setting start to true. The timer module starts the count and resets the start signal to false. Port clock is an input port of type bool that is used to provide a time reference signal to the timer module. Output port timeout connects to the transmit module and alerts the transmit module when the timer expires. This means that either the packet or acknowledge were lost during transmission and the packet needs to be transmitted again.

Constructor

The timer module contains one process called runtimer. This process is sensitive to the positive edge of port clock.

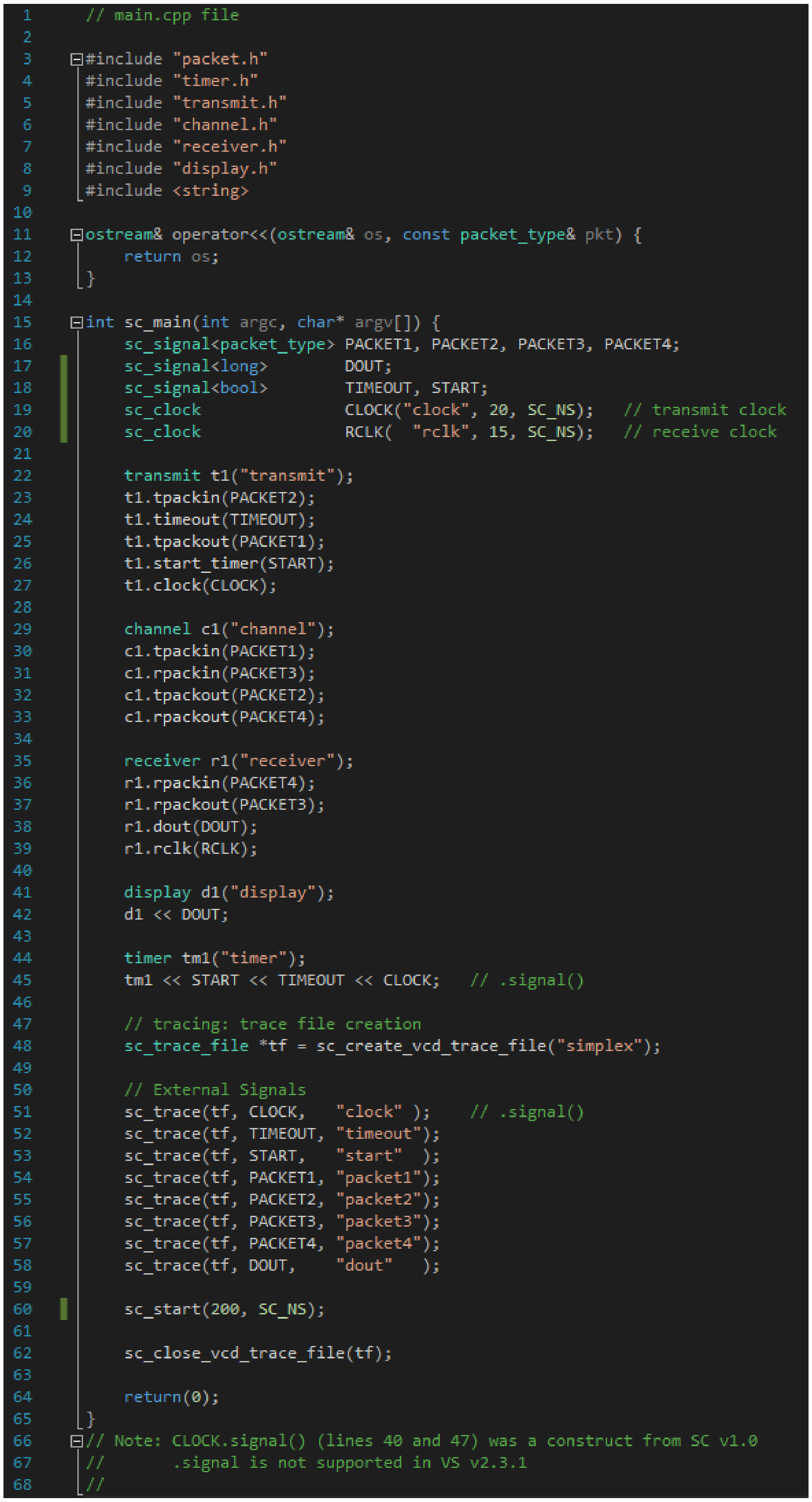
Implementation *of Methods*

If the value of the start signal is 1 the timer is started. When the timer is started, a debug message is written out, the value of variable count is set to the timer delay value, and output port timeout is set to false. The false value signifies that the timer is running and has not timed out yet. The timer process then resets the start signal to 0.

If variable count is greater than 0, the timer is still counting down. Variable count is decremented and port timeout stays false. If variable count is equal to 0, the timer has expired and timeout is set to true. At this point the transmit module knows that the packet was lost and retransmits the packet.

Putting it all together - The main routine

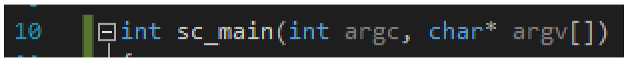
The sc_main routine is the top-level routine that ties all the modules together and provides the clock generation and tracing capabilities. The sc_main routine is shown below:



```cpp
// main.cpp file

#include "packet.h"
#include "timer.h"
#include "transmit.h"
#include "channel.h"
#include "receiver.h"
#include "display.h"
#include <string>

ostream& operator<<(ostream& os, const packet_type& pkt) {
    return os;
}

int sc_main(int argc, char* argv[]) {
    sc_signal<packet_type> PACKET1, PACKET2, PACKET3, PACKET4;
    sc_signal<long>        DOUT;
    sc_signal<bool>        TIMEOUT, START;
    sc_clock               CLOCK("clock", 20, SC_NS);   // transmit clock
    sc_clock               RCLK(  "rclk", 15, SC_NS);   // receive clock

    transmit t1("transmit");
    t1.tpackin(PACKET2);
    t1.timeout(TIMEOUT);
    t1.tpackout(PACKET1);
    t1.start_timer(START);
    t1.clock(CLOCK);

    channel c1("channel");
    c1.tpackin(PACKET1);
    c1.rpackin(PACKET3);
    c1.tpackout(PACKET2);
    c1.rpackout(PACKET4);

    receiver r1("receiver");
    r1.rpackin(PACKET4);
    r1.rpackout(PACKET3);
    r1.dout(DOUT);
    r1.rclk(RCLK);

    display d1("display");
    d1 << DOUT;

    timer tm1("timer");
    tm1 << START << TIMEOUT << CLOCK;   // .signal()

    // tracing: trace file creation
    sc_trace_file *tf = sc_create_vcd_trace_file("simplex");

    // External Signals
    sc_trace(tf, CLOCK,   "clock" );    // .signal()
    sc_trace(tf, TIMEOUT, "timeout");
    sc_trace(tf, START,   "start"  );
    sc_trace(tf, PACKET1, "packet1");
    sc_trace(tf, PACKET2, "packet2");
    sc_trace(tf, PACKET3, "packet3");
    sc_trace(tf, PACKET4, "packet4");
    sc_trace(tf, DOUT,    "dout"   );

    sc_start(200, SC_NS);

    sc_close_vcd_trace_file(tf);

    return(0);
}
// Note: CLOCK.signal() (lines 40 and 47) was a construct from SC v1.0
//       .signal is not supported in VS v2.3.1
//
```

Include Files
Notice that the sc_main file includes all of the other modules in the design. You instantiate each of the lower level modules and connect their ports with signals to create the design in sc_main. To instantiate a lower level module, the interface of the module must be visible. Including the .h file from the instantiated module provides the necessary visibility.

Argument to sc_main
The sc_main routine takes the following arguments:



```
int sc_main(int argc, char* argv[])
```

The argc argument is a count of the number of command line arguments and the argv is an array containing the arguments as char* strings. This is the standard C++ way of parsing command line arguments to programs.

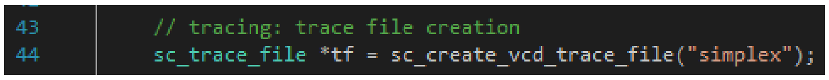
Signals
After the sc_main statement, the local signals are declared to connect the module ports together. Four signals are needed for packet_type to cross connect the transmit, receiver, and channel modules. There are two clock declarations, clock and rclk. Clock is used as the transmitter clock and will synchronize the transmit block and the timer block. Rclk is used as the receiver clock and will synchronize the receiver block and the display block.

Module Instantiation
After the declaration statements, the modules in the design are instantiated. The transmit, channel, receiver, display, and timer are instantiated and connected together with the locally declared signals. This completes the implementation of the design.

Positional and Named Connections
In the sc_main file two different types of connections were used to connect signals to the module instantiations. Modules transmit, channel, and receiver used named connections. A named connection connects a port name to a signal name. Notice that the port names were in lowercase and the signal names in uppercase.

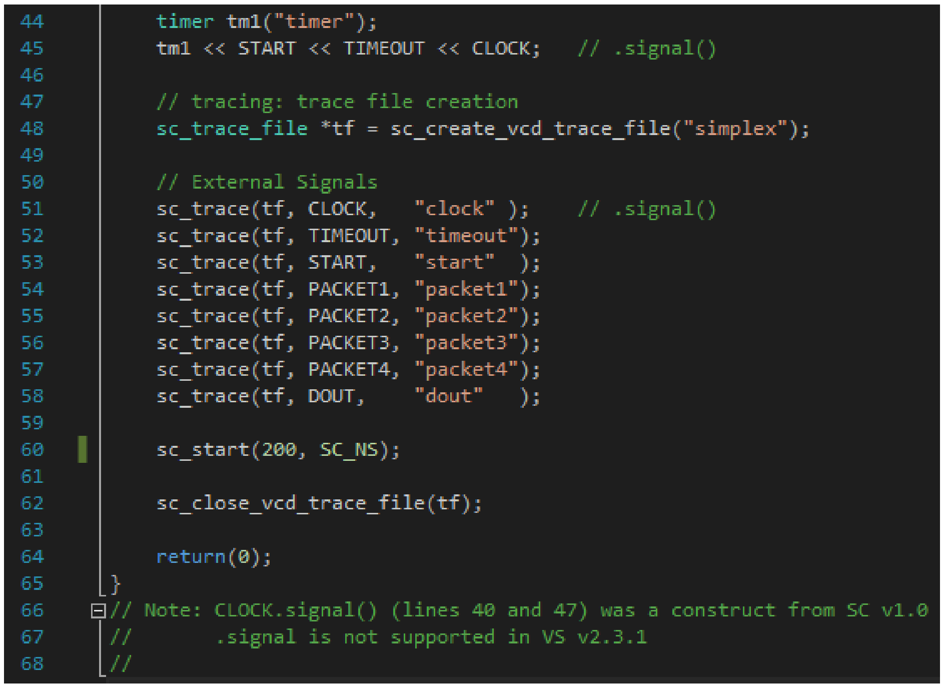
Modules display and timer used positional connections to connect signals to the module instantiations. With this style of connection a list of signals is passed to the instantiation and the first signal in the list connects to the first port, the second signal to the second port, etc.

Using Trace
The program can now be built and run. To make is easier to determine if the design works as intended, you can create a trace file with the built-in signal tracing methods in SystemC. The first trace command, shown below, creates a trace file named simplex.vcd into which the results of simulation can be written:



```
    // tracing: trace file creation
    sc_trace_file *tf = sc_create_vcd_trace_file("simplex");
```

Next, a set of sc_trace commands trace the signals and variables of a module, as follows:



```
44         timer tm1("timer");
45         tm1 << START << TIMEOUT << CLOCK;   // .signal()
46 
47         // tracing: trace file creation
48         sc_trace_file *tf = sc_create_vcd_trace_file("simplex");
49 
50         // External Signals
51         sc_trace(tf, CLOCK,   "clock" );    // .signal()
52         sc_trace(tf, TIMEOUT, "timeout");
53         sc_trace(tf, START,   "start"  );
54         sc_trace(tf, PACKET1, "packet1");
55         sc_trace(tf, PACKET2, "packet2");
56         sc_trace(tf, PACKET3, "packet3");
57         sc_trace(tf, PACKET4, "packet4");
58         sc_trace(tf, DOUT,    "dout"   );
59 
60         sc_start(200, SC_NS);
61 
62         sc_close_vcd_trace_file(tf);
63 
64         return(0);
65     }
66     // Note: CLOCK.signal() (lines 40 and 47) was a construct from SC v1.0
67     //       .signal is not supported in VS v2.3.1
68     //
```

These commands write the value of the signal specified to the trace file previously created. The last argument specifies the name of the signal in the trace file. After simulation is executed, you can examine the results stored in the trace file with a number of visualization tools that generate waveforms and tables of results.

Simulation Start

After the trace commands, the following function call instructs the simulation kernel to run for 10,000 default time units and stop:

```
sc_start(10000);
```

Alternatively, you can use an sc_start value of -1, as shown below:

```
sc_start(-1);
```

This command tells the simulation to run forever.

After the example is completely described in SystemC, the commands to build the simulator need to be specified. The following sections provide procedures for compiling under UNIX and Windows.

Compiling the Example for UNIX

The following steps are needed to compile the design for the UNIX environment:

1. Create a new directory for the design and create all the design files in it.
2. Copy file Makefile and Makefile.defs from the SystemC installation examples directory into the new directory.
3. Edit the Makefile so that the list of files includes all of the design source files.An example Makefile is shown below:

   ```
   TARGET_ARCH = gccsparcOS5
   ```

```
MODULE = demo
SRCS = channel.cpp display.cpp packet.cpp receiver.cpp timer.cpp transmit.cpp main.cpp
OBJS = $(SRCS:.cpp=.o)

include ./Makefile.defs
```

Edit the SRCS line to list all of the source files in the design. Don't remove the line "include ./Makefile.defs". The MODULE line specifies the name of the executable to run when the compilation is done. In this example, the compilation creates a program named demo.

4. Open the Makefile.defs and make sure that the SYSTEMC line points to the current location of the SystemC class libraries. An example is shown below:

```
TARGET_ARCH = gccsparcOS5
CC      = g++
OPT     =
DEBUG   = -g
SYSTEMC = /remote/dtg403/dperry/systemc-2.0
INCDIR  = -I. -I.. -I$(SYSTEMC)/include
LIBDIR  = -L. -L.. -L$(SYSTEMC)/lib-$(TARGET_ARCH)
CFLAGS  = -Wall $(DEBUG) $(OPT) $(INCDIR) $(LIBDIR)
LIBS    = -lsystemc -lm $(EXTRA_LIBS)
// rest of file not shown
```

5. By default the simulation is built with debugging turned on. Modify the DEBUG line to turn on or off the debugging options as desired.

6. To compile the design, enter the following in the command line:

```
unix% gmake
```
or
```
unix% make
```

Compiling the Example for Windows

The SystemC distribution includes project and workspace files for Visual C++. If you use these project and workspace files the SystemC source files are available to your new project. For Visual C++ 6.0 the project and workspace files are located in directory:

```
...\systemc-2.0\msvc60
```

This directory contains two subdirectories: systemc and examples.

The systemc directory contains the project and workspace files to compile the systemc.lib library. Double-click on the systemc.dsw file to launch Visual C++ with the workspace file. The workspace file will have the proper switches set to compile for Visual C++ 6.0. Select "Build systemc.lib" under the Build menu or press F7 to build systemc.lib.

The examples directory contains the project and workspace files to compile the SystemC examples. Go to one of the examples subdirectories and double-click on the .dsw file to launch Visual C++ with the workspace file. The workspace file will have the proper switches set to compile for Visual C++ 6.0. Select "Build <example>. exe" under the Build menu or press F7 to build the example executable.

To create a new design, first create a new project by using the "New" menu item under the File menu. Select the Projects tab on the dialog box that appears and select Win32 Console Application. Create an empty project.

For your own SystemC applications, make sure that the Run Time Type Information switch is on by using the "Settings..." menu item under the Project menu. Select the C/C++ tab, and select the C++ Language category. Make sure that the Enable Run Time Type Information (RTTI) checkbox is checked.

Also make sure that the SystemC header files are included by selecting the C/C++ tab, selecting the Preprocessor category, and typing the path to the SystemC src directory in the text entry field labeled "Additional include directories". The examples use e.g. "../../../src".

Next add the source files to the project by using the "Add To Project>Files..." menu item under the Project menu. Make sure that the files are added to the new project directory just created. Do the same for the systemc.lib library before building your SystemC application.

Now use the Compile and Build menu selections to compile and build the SystemC application. When the application has been built, the design can be run from Visual C++ to debug the application.

Executing the Example

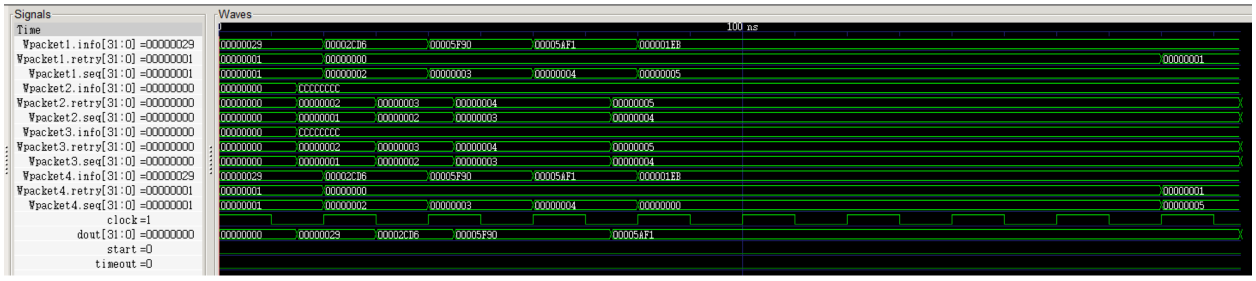
After the simulation executable is built, you run the simulation by executing the simulation executable created in the compilation step. The simulation executable is a batch program that executes the simulation. For example, to run a simulation for a module named demo, simply type demo at the command prompt and press return. If you built a console application in Visual C++ you can run the application in a Windows Command Prompt window by typing the name of the project created.

The duration of the simulation is specified by method sc_start in the sc_main module. The data created by the simulation is specified with the sc_trace commands in the sc_main module.

When the simulation is complete, a trace file of the traced signals is created. You can use tools to view waveforms and tables from this data and analyze the results of simulation and determine whether or not the simulation succeeded.

Viewing Output File

Using GTK Wave to view the simplex.vcd file allows you to visually verify timing:



```
Time
 Wpacket1.info[31:0] =00000029
Wpacket1.retry[31:0] =00000001
  Wpacket1.seq[31:0] =00000001
 Wpacket2.info[31:0] =00000000
Wpacket2.retry[31:0] =00000000
  Wpacket2.seq[31:0] =00000000
 Wpacket3.info[31:0] =00000000
Wpacket3.retry[31:0] =00000000
  Wpacket3.seq[31:0] =00000000
 Wpacket4.info[31:0] =00000029
Wpacket4.retry[31:0] =00000001
  Wpacket4.seq[31:0] =00000001
             clock =1
       dout[31:0] =00000000
             start =0
           timeout =0
```