

Using System C Exports and Hierarchical Channels for Efficient Data Conversion and Loopback



Electronic **S**ystem **L**evel **X**perts

Rosamaria Carbonell Mann & David C Black



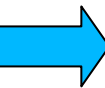
www.ESLX.com

Info@ESLX.com

Version 1.0



Agenda

- 
- Goals
 - Overview of exports
 - Overview of hierarchical channels
 - Data Conversion example
 - Data Conversion Solution
 - Loopback example
 - Loopback Solution
 - Suggestions for other useful channels
 - Summary
 - Q&A



Goals

- Review exports and hierarchical channels in System C
 - Purpose
 - Practical Uses
- Demonstrate real examples
- Suggest other uses



Overview of exports in System C

- Difference between `sc_ports` & `sc_exports`
 - Ports point to channels from the outside
 - `sc_exports` export a pointer from the inside
- Motivation
 - Ease of use (can put channels anywhere)
 - Makes using IP more convenient
 - ◆ Don't have to instantiate IP and channels
- Example
 - TLM channels use exports



SC_EXPORT



"exports the interface of the channel"

Direction of call reversed.

modA mA

```
sc_export<INTERFACE> pA
```

```
CHANNEL c;
```

```
write()...
read()...
```

A_thread

```
c.write(v);
```

modB mB

```
sc_port<INTERFACE> pB
```

B_thread

```
v=pB->read();
```

Pointer Access

Overview of hierarchical channels

- Purpose
 - Model complex communications
 - ◆ Separate communications from implementation
 - Adapt between models at different abstraction levels
 - ◆ Transactors are simply pin-level adaptors to TLM
- Motivation
 - Simplify comparison of bus architectures
 - Enable fast simulations, performance analysis & detailed implementation of complex buses
- Example
 - Processor bus



Channels Defined

- Channels safely communicate between processes.
- From IEEE Std 1666
 - A primitive channel is a non-abstract class derived from one or more interfaces and also derived from the class **sc_prim_channel**.
 - A hierarchical channel is a non-abstract class derived from one or more interfaces and also derived from the class **sc_module**.
 - ◆ **sc_channel** is a **typedef** (alias) for **sc_module**.
- Characteristics
 - Primitive channels are supposed to be fast and light-weight
 - Hierarchical channels are often complex, but not always
- Capabilities (unique to each type)
 - Primitive channels are allowed to call **request_update()**
 - Hierarchical channels are allowed to have ports processes, and hierarchy (sub-modules/channels)



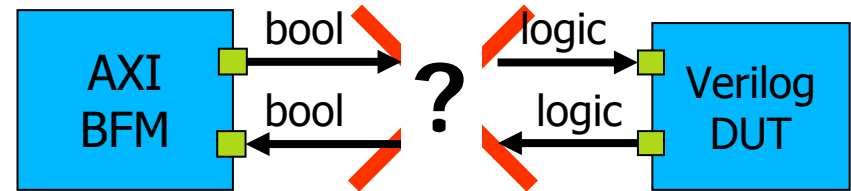
Agenda

- Goals
- Overview of exports
- Overview of hierarchical channels
- ● Data Conversion example
- Data Conversion Solution
- Loopback example
- Loopback Solution
- Suggestions for other useful channels
- Summary
- Q&A

Data Conversion Example

- Problem

- Transactor has bool ports
- DUT has logic ports
- Expert Verilog users, first exposure to System C and in some cases, C++

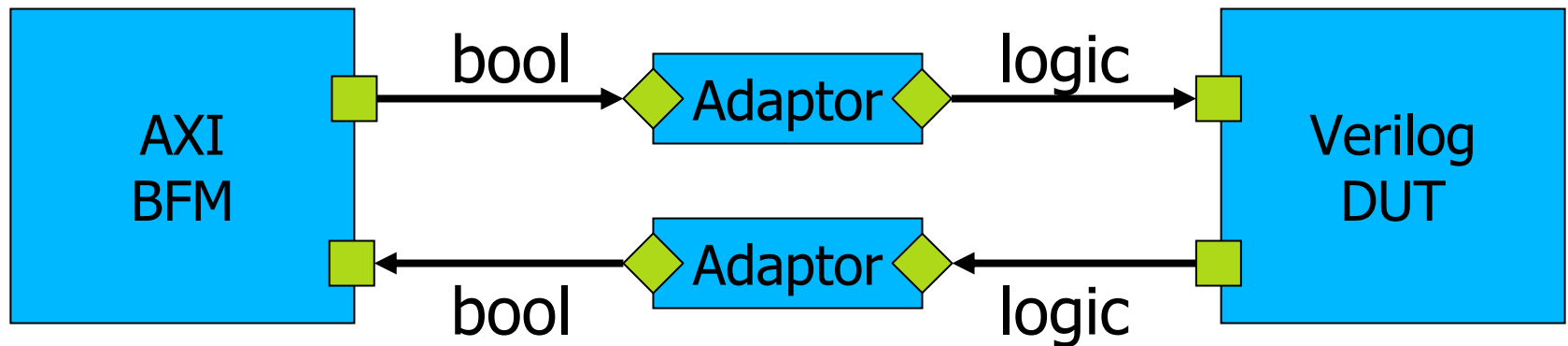


- Solution

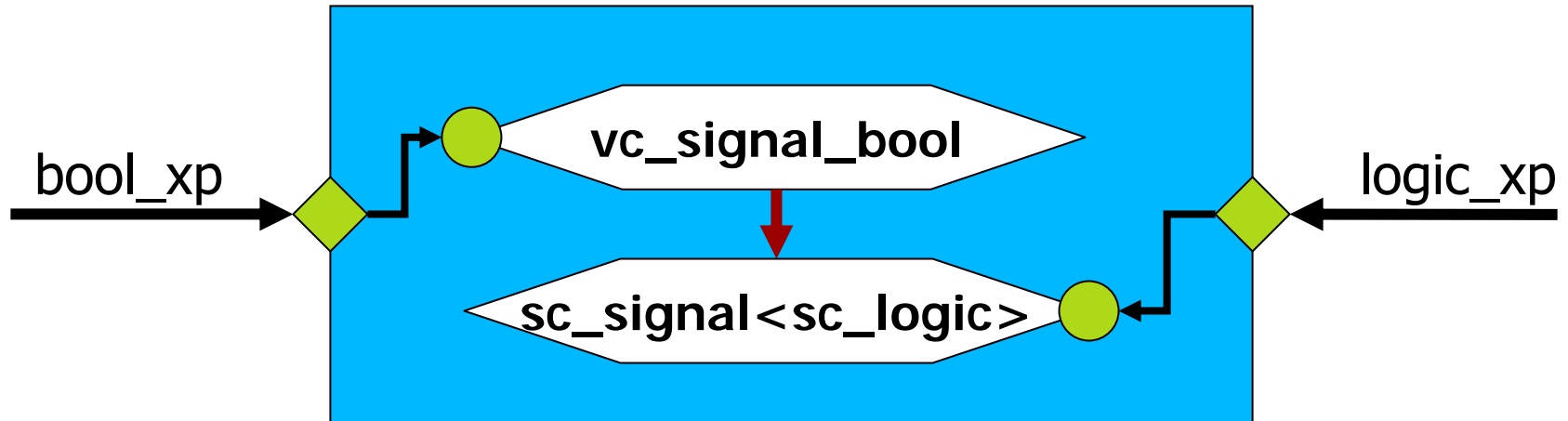
- Easy to understand, adaptor
- Self-contained channel
- Easy to use, no extra channels required to connect to Verilog
- Easy to re-use - template based

Data Conversion Solution

Create a unique hierarchical channel to adapt



Loopback Solution graphically





Supporting sub-channels definitions (1 of 2)

```
// First we define two supporting signals to embed
```

```
typedef sc_signal<sc_logic> vc_signal_logic;
```

```
// Definition of supporting signal where work is done
```

```
// - Store all information in other channel (vc_signal_logic),
```

```
// and convert all accesses here...
```

```
class vc_signal_bool : public sc_signal_inout_if<bool>
    , public sc_prim_channel {
```

```
public:
```

```
vc_signal_bool() // Constructor
```

```
:sc_prim_channel(sc_gen_unique_name("vc_signal_bool")) {}
```

```
// Implement ALL methods as calls to m_logic_sig-> versions
```

```
void register_port(sc_port_base& pb, const char* cp)
```

```
{ m_logic_sig->register_port(pb, cp); }
```

```
void operator() (vc_signal_logic& logic_sig)
```

```
{ m_logic_sig = &logic_sig; }
```

```
const sc_event& default_event() const
```

```
{ return m_logic_sig->default_event(); }
```

```
...
```



Supporting sub-channel Read & Write (2 of 2)

```

void write(const bool& val) {
    m_logic_sig->write(sc_logic(val));
    m_bool_value = val;
}

const bool& read() const {
    if (m_logic_sig->read() == SC_LOGIC_0) m_bool_value = false;
    else m_bool_value = true;
    return m_bool_value;
}

operator const bool&() const {
    if (m_logic_sig->read() == SC_LOGIC_0) m_bool_value = false;
    else m_bool_value = true;
    return m_bool_value;
}

private:
    vc_signal_logic* m_logic_sig; // points to the sc_signal
    mutable bool m_bool_value;
}; //end vc_signal_bool

```

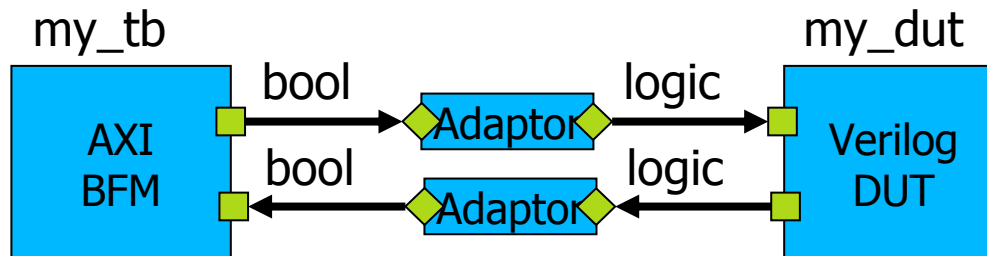


Data Conversion: usage

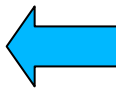
```

DUT my_dut; // logic
TRANSACTOR my_tb; // bool
logic_to_bool bool2logic_ch;
logic_to_bool logic2bool_ch;
bool2logic_ch.bool(my_tb.input_pi);
bool2logic_ch.logic(my_dut.input_pi);
logic2bool_ch.bool(my_tb.input_pi);
logic2bool_ch.logic(my_dut.input_pi);

```



Agenda

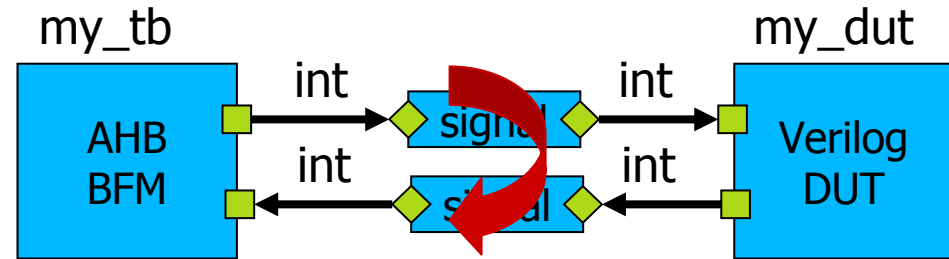
- Goals
- Overview of exports
- Overview of hierarchical channels
- Data Conversion example
- Data Conversion Solution
- Loopback example 
- Loopback Solution
- Suggestions for other useful channels
- Summary
- Q&A



Loopback Example

● Problem

- Self-test transactors
- Do not modify DUT
 - ◆ just change execution
 - ◆ mode from loopback to pass-through
- Reduce context switching (slight)



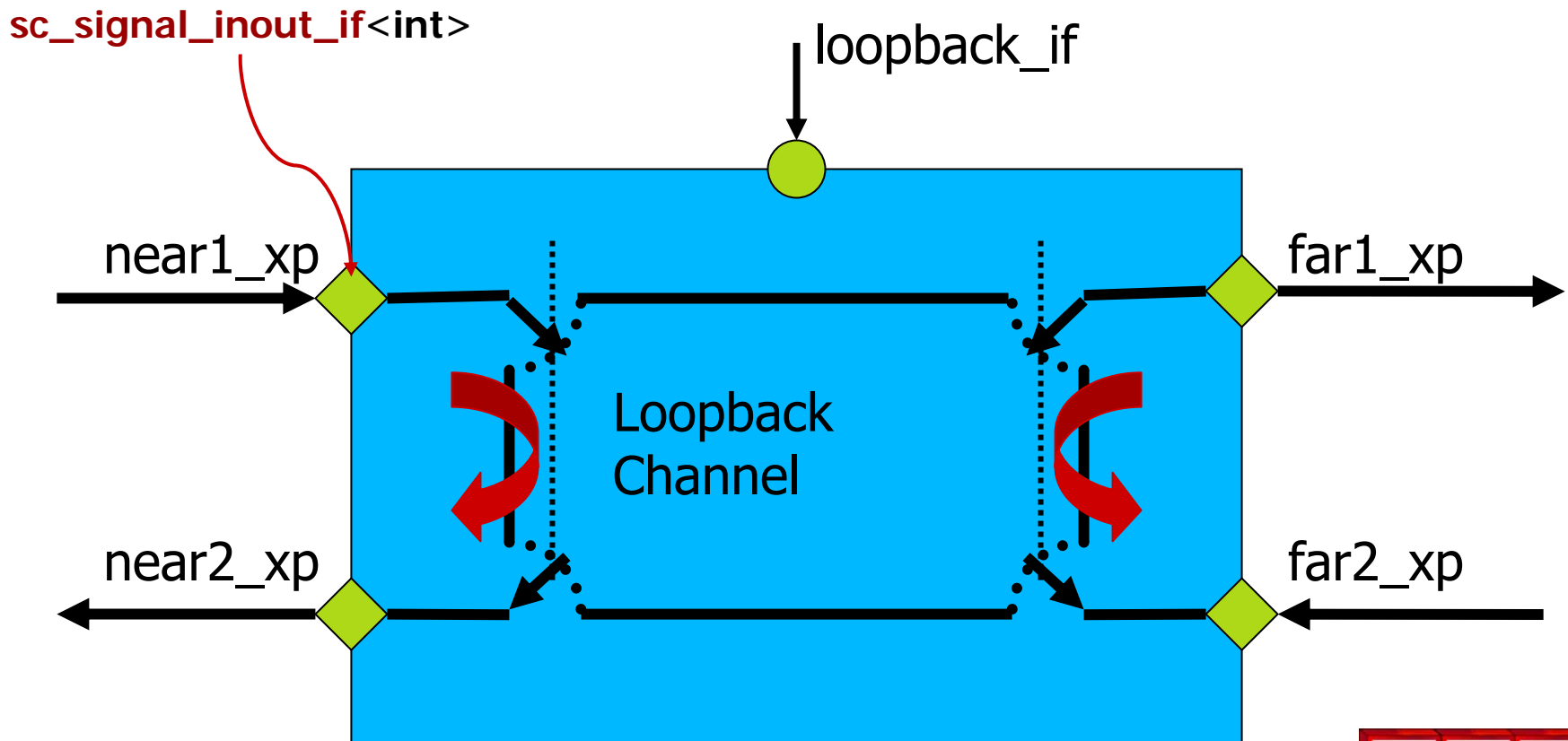
● Solution

- Easy to understand
- Self-contained channel
- Easy to use
- Easy to re-use - template based



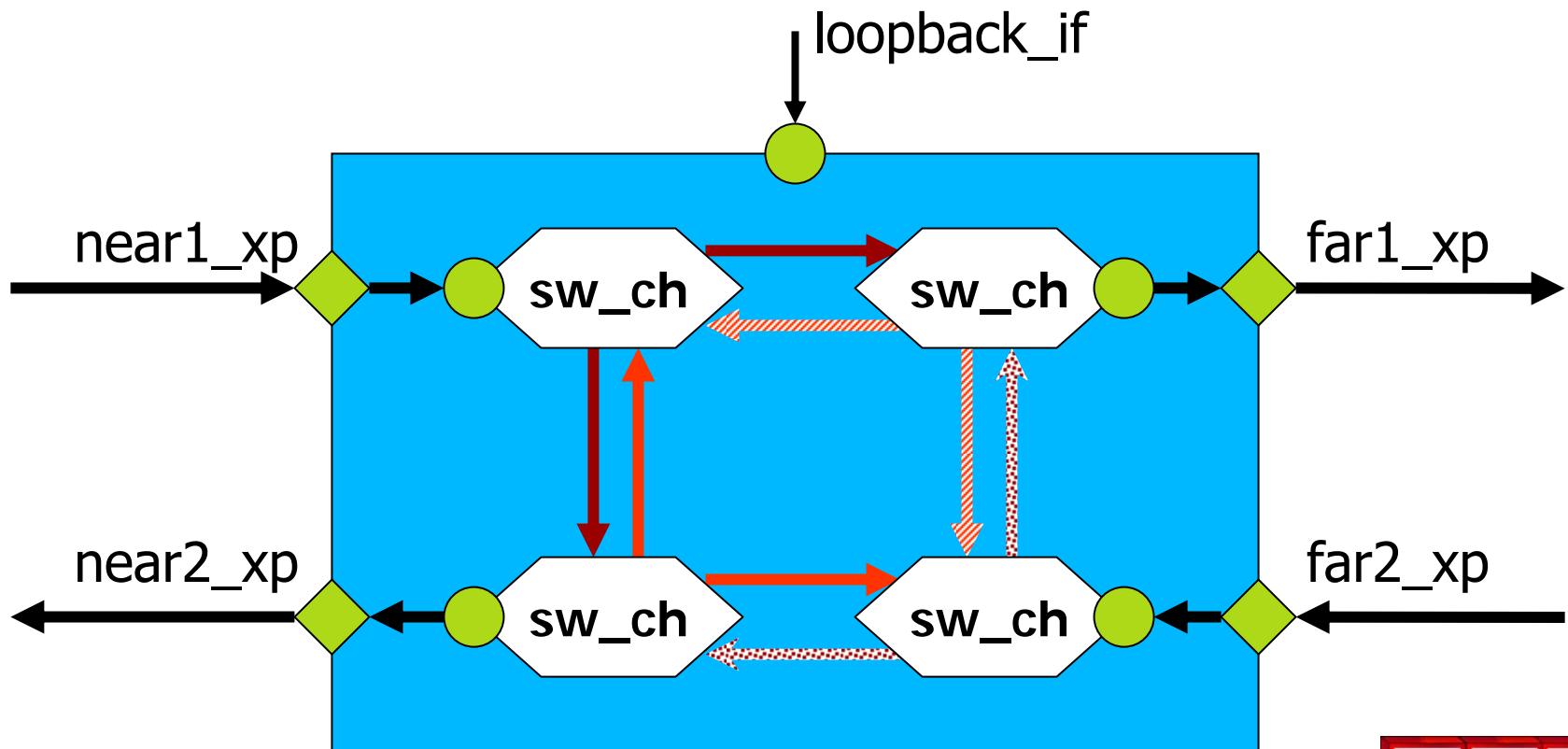
Loopback - external view

Data Flow (double pole, double throw)



Solution graphically

- Custom primitive channel, `sw_ch` implements `loopback_if` and `sc_signal_inout_if<int>`



Loopback: Mode method

```
enum mode_t { LOOPBACK, PASS_THRU };
```

```
void set_mode(mode_type mode) {  
    m_mode = mode;  
}
```

```
mode_type get_mode(void) const {  
    return m_mode;  
}
```



Loopback: Read method

- Local copy of current value

```
const int& read() const {  
    return m_curr;  
}
```

Loopback: Write method

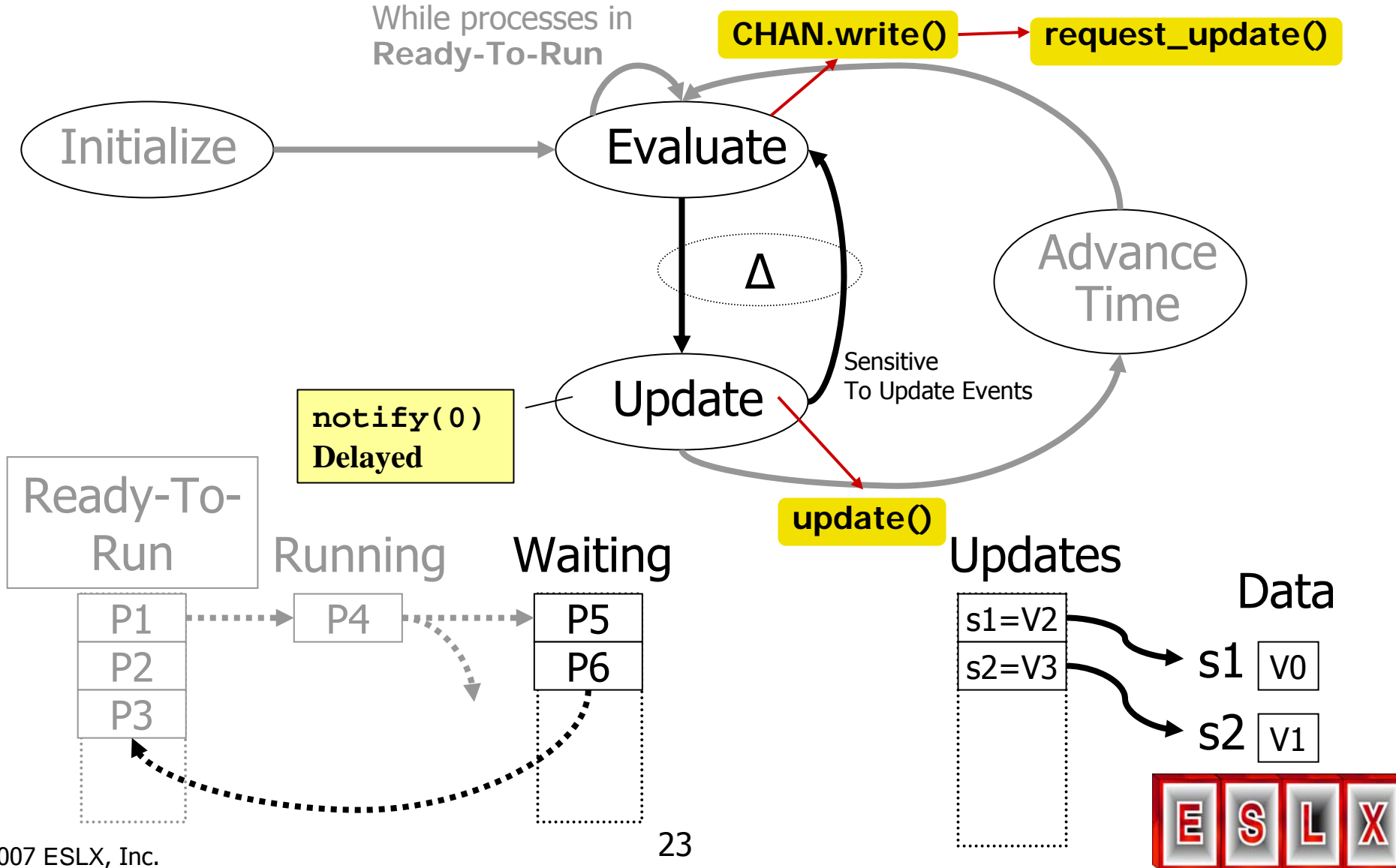
- Real work occurs in update() - next 3 slides

```
void write( const int& new_value) {  
    if ( ! (m_curr == new_value)) {  
        m_next = new_value;  
        request_update();  
    }  
}
```



Simulation Engine

While processes in
Ready-To-Run

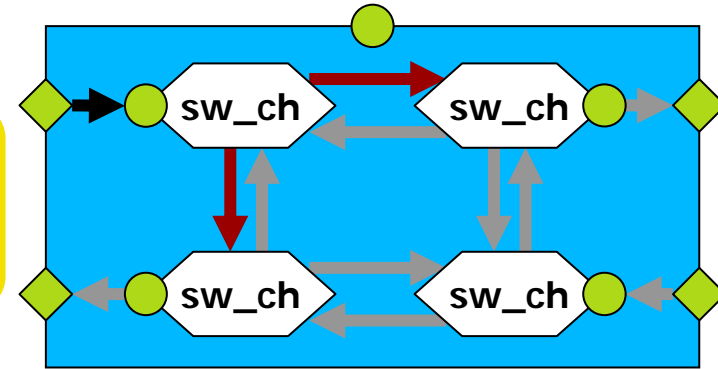


Loopback: Update method

```

void update() {
    update_value(
        m_next, m_curr, m_changed_evt
        m_evt_rqst, m_delta
    );
    if (m_mode == PASS_THRU) {
        update_value(
            m_next, *m_pass_curr_ptr, *m_pass_evt_ptr,
            *m_pass_evrq_ptr, *m_pass_delta_ptr
        );
    } else {
        update_value(
            m_next, *m_loop_curr_ptr, *m_loop_evt_ptr,
            *m_loop_evrq_ptr, *m_loop_delta_ptr
        );
    }
}

```



Loopback: Update_value helper

```
void update_value(  
    int new_value, int& curr_value,  
    sc_event& changed_evt, bool ev_rqst,  
    sc_dt::uint64& delta  
) {  
    if ( ! (curr_value == new_value) ) {  
        curr_value = new_value;  
        if (ev_rqst) changed_evt.notify();  
        delta = sc_delta_count();  
    } //endif  
}
```



Loopback: value_changed_event

- Each SW_CH has its own event

```
const sc_event&  
value_changed_event() const  
{  
    event_requested = true;  
    return m_changed_evt;  
}
```



Loopback: usage

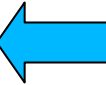
- Simply instantiate and connect

```
SC_MODULE(TOP) {
    M1 m1_inst;
    M2 m2_inst;
    loopback lpbk_ci;
    SC_CTOR(TOP), m1_inst("m1_inst")
    , m2_inst("m2_inst"), lpbk_ci("lpbk_ci")
    {
        m1_inst.p1(lpbk_ci.near1_xp);
        m1_inst.p2(lpbk_ci.near2_xp);
        m2_inst.p1(lpbk_ci.far1_xp);
        m2_inst.p2(lpbk_ci.far2_xp);
        lpbk_ci.set_mode(LOOPBACK);
    }
};
```



Agenda

- Goals
- Overview of exports
- Overview of hierarchical channels
- Data Conversion example
- Data Conversion Solution
- Loopback example
- Loopback Solution
- Suggestions for other useful channels
- Summary
- Q&A



Suggestions for other useful applications of `sc_export`

- Data conversion between any types
 - Many of these can be templates
- Bit splitting (8 bits taken out of 32)
 - Warning: May indicate too low level thinking
- Busses needing differentiated ports using a common interface
 - Reduces adding channels between ports
 - Increase performance



Summary

- Exports do not require an external channel for binding and may be left unbound.
- Hierarchical channels can be used to connect ports and implement specialized behavior in an easy to use and re-use self-contained module.
- Sample code will be available:
www.eslx.com/Library_open_area/register.html



Questions?

- Goals
- Overview of exports
- Overview of hierarchical channels
- Data Conversion example
- Data Conversion Solution
- Loopback example
- Loopback Solution
- Suggestions for other useful channels
- Summary
- Q&A

