

GreenReg

Device & Register Framework

v-2.0

API Overview

WARNING:

All image texts have not yet been renamed:

DRF → GreenReg
prefix dr_ → gr_



- Introduction
 - What is GreenReg?
 - Ambitious Goals
 - What is missing?
- Concepts & Architecture behind GreenReg
 - Bridging Model Complexity
 - Registers as Stimuli
 - Device Framework
 - Register Framework
 - Notification Rule Execution Path
- API Overview
 - Device
 - Register Container & Register
 - Bit Range & Bit
 - Notification Rules & DRF_Port<master>
 - Attribute & Switch

API Not Covered
State Machine Support



Introduction



What is GreenReg?

GreenReg - GreenSocs (Device &) Register Framework

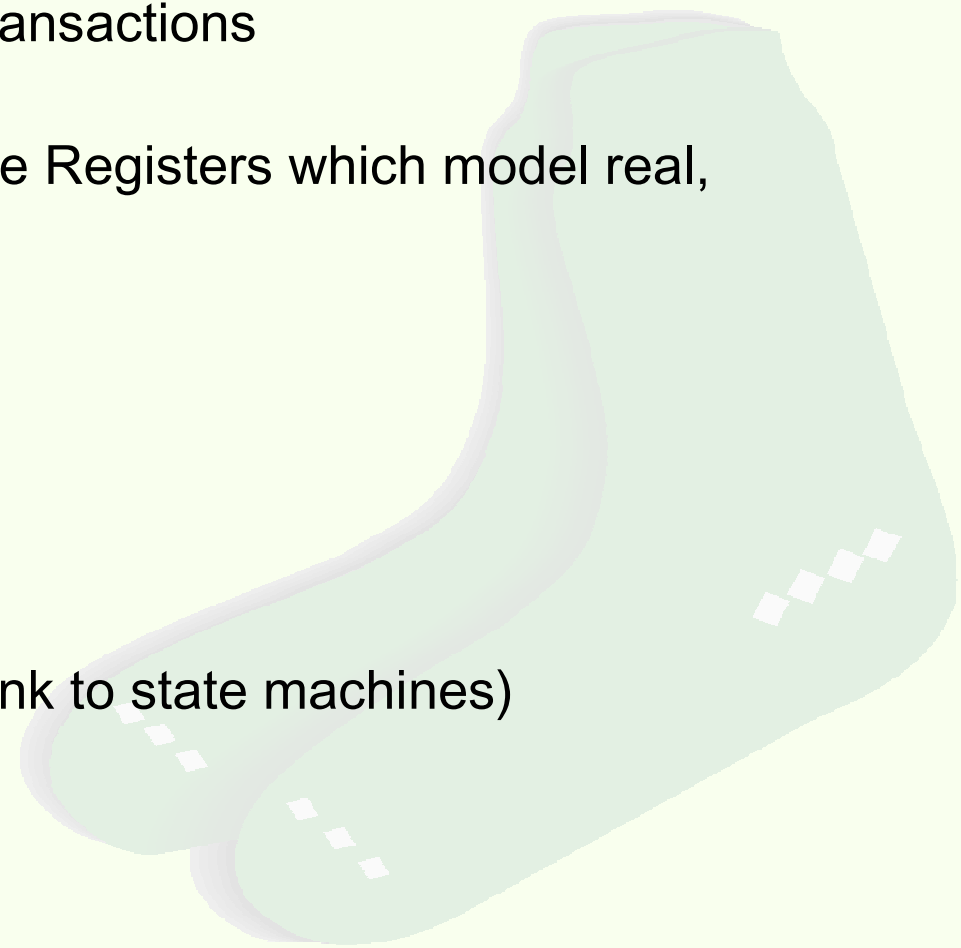
- Refinement of Intel DRF
- Developed to address several deficiencies in the OSCI language.
 - Clear hierarchical accessibility of blocks and their elements
 - Simplified communication over chosen bus
 - Complete register implementation with simplified accessibility as apposed to current industry solutions
 - Bit level decode
- Intended for rapid development & maintainability of ESL models.
- GreenReg has APIs for the following:
 - **attribute**
 - **switch**
 - **device** (may have registers)
 - **sub-device** (may not have registers)
 - **registers**
 - **event C++ callback** (bypass SC kernel)
 - **bit ranges**
 - **bits**
 - **consistent accessibility**
 - (global, local, hierarchical)
 - **analysis collection** (for power, emi, or other)
 - **simplified sockets**

Deficiencies in SystemC which GreenReg attempts to address

- Re-use of code in PV, PVT & AV models (methodology of implementation) - WIP
- TLM independence with simplified transactions (OSCI TLM, GSGPSocket, “XYZ”)
- Intelligent Attributes & Highly Flexible Registers which model real, complex data scenarios

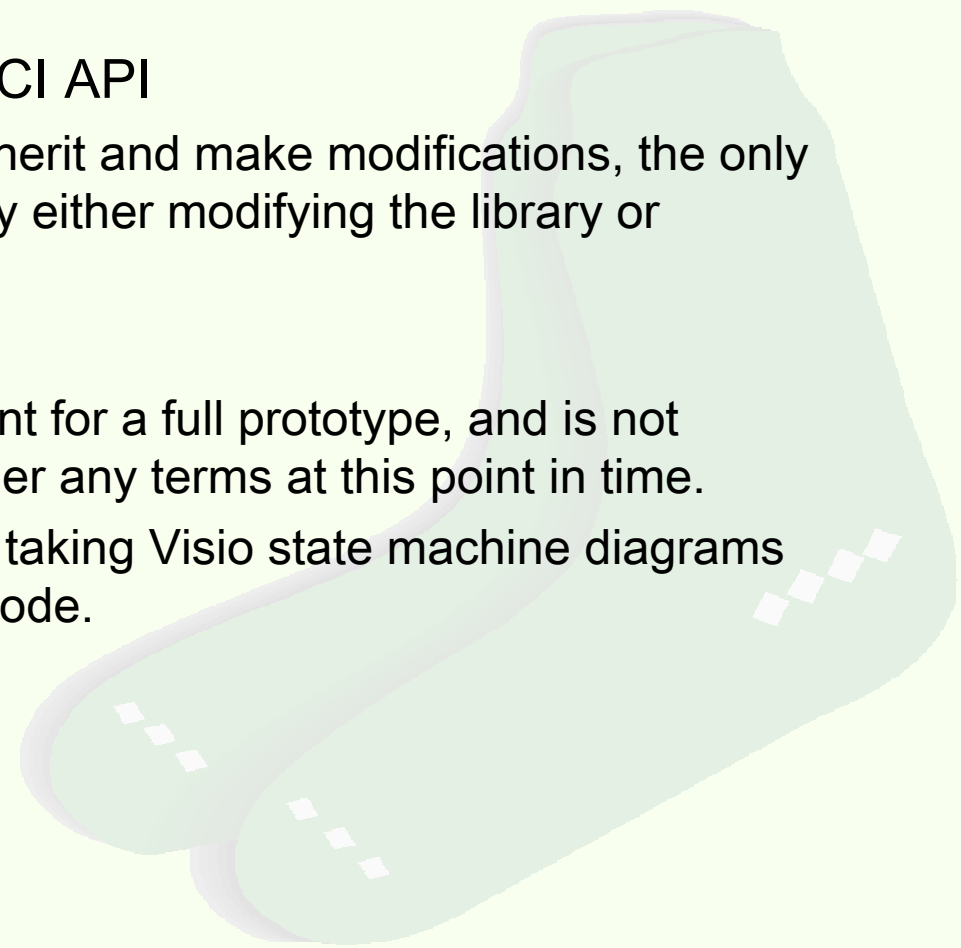
Key aspects to GreenReg

- Simple User API
- Access to Data
- Manipulation of Data
- Data Driven Changes (SC missing link to state machines)
- Extensible “Contained Capabilities”



What is Missing?

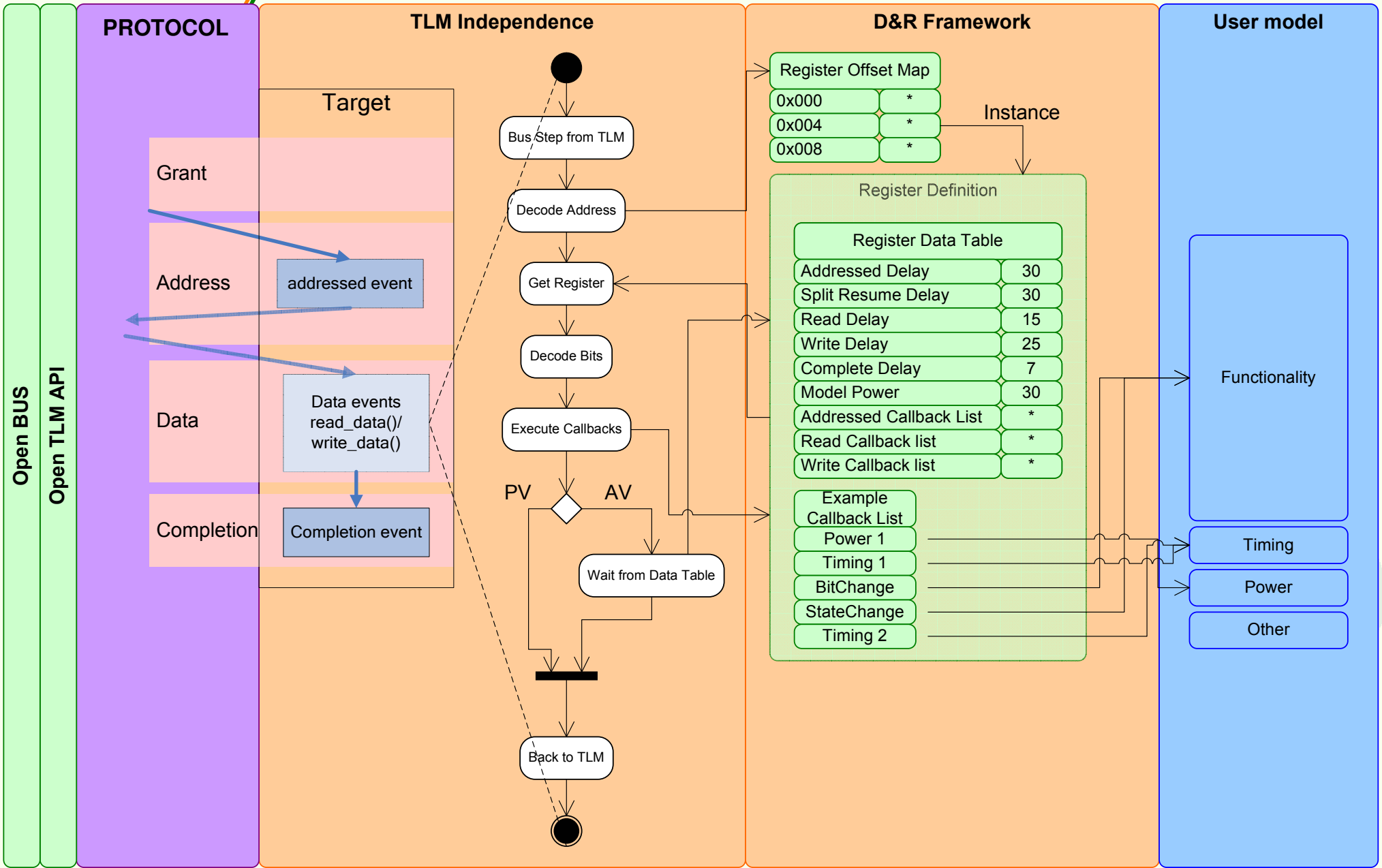
- Endianness
 - Can be addressed on back end but...
 - do not care, and IA target running on an IA host does not need said overhead.
- Complete enhancement into the OSCI API
 - SystemC is too tightly wrapped to inherit and make modifications, the only way to make real improvements is by either modifying the library or building an exoskeleton.
- State machine kernel
 - Still under investigation / development for a full prototype, and is not available for release in any form under any terms at this point in time.
 - Positive progress has been made in taking Visio state machine diagrams and generating compiler compliant code.



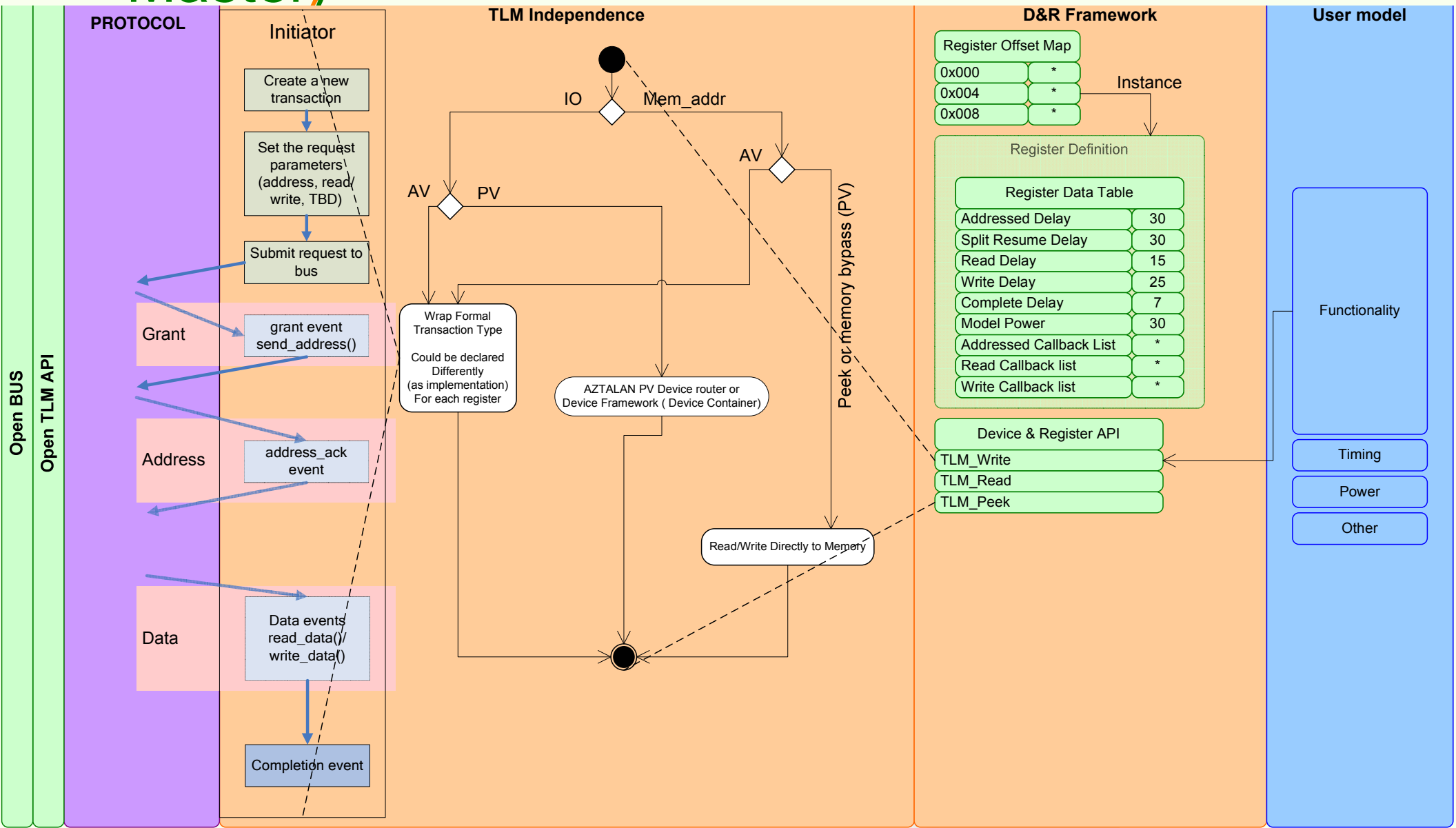
Concepts & Architecture Behind GreenReg



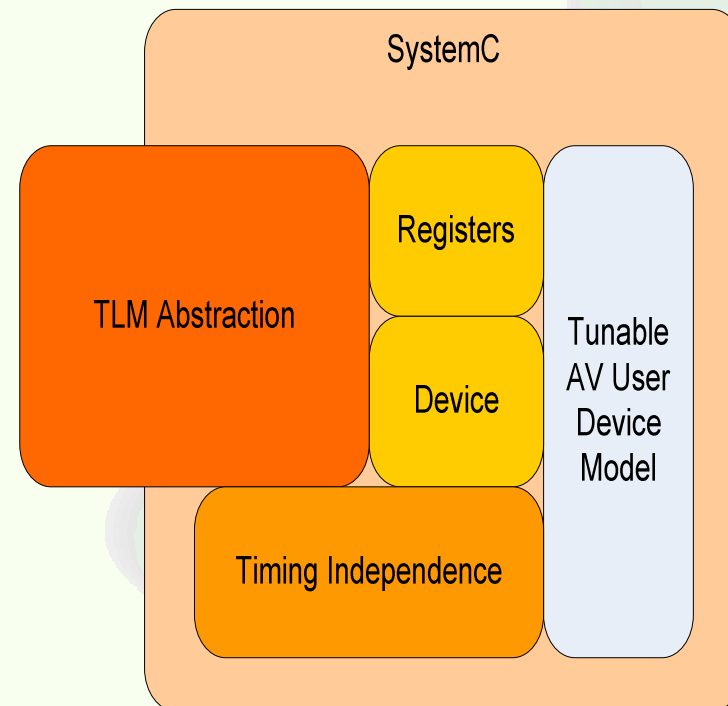
Registers as Stimuli (TLM Independence Slave)



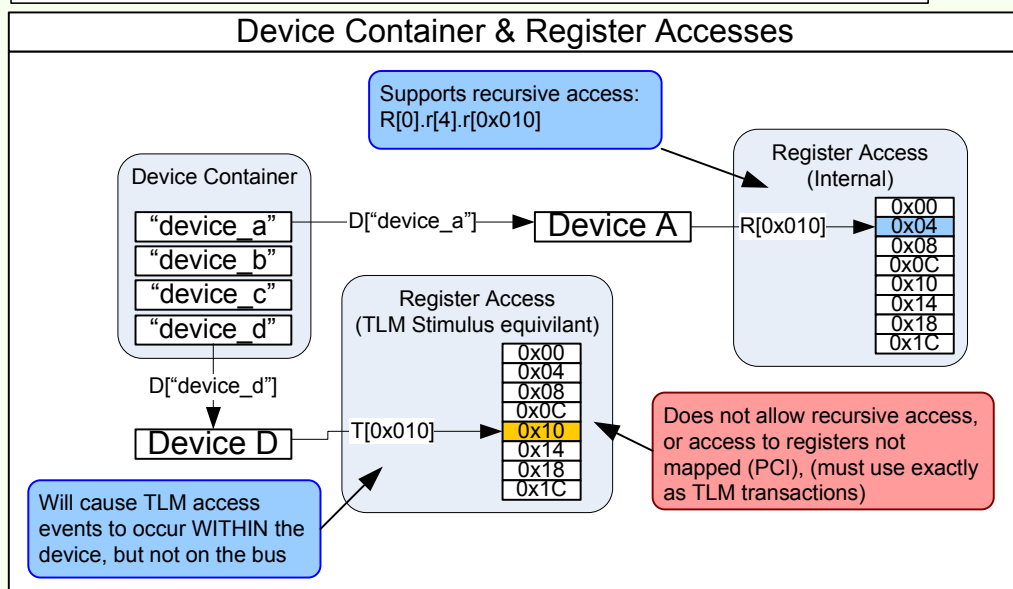
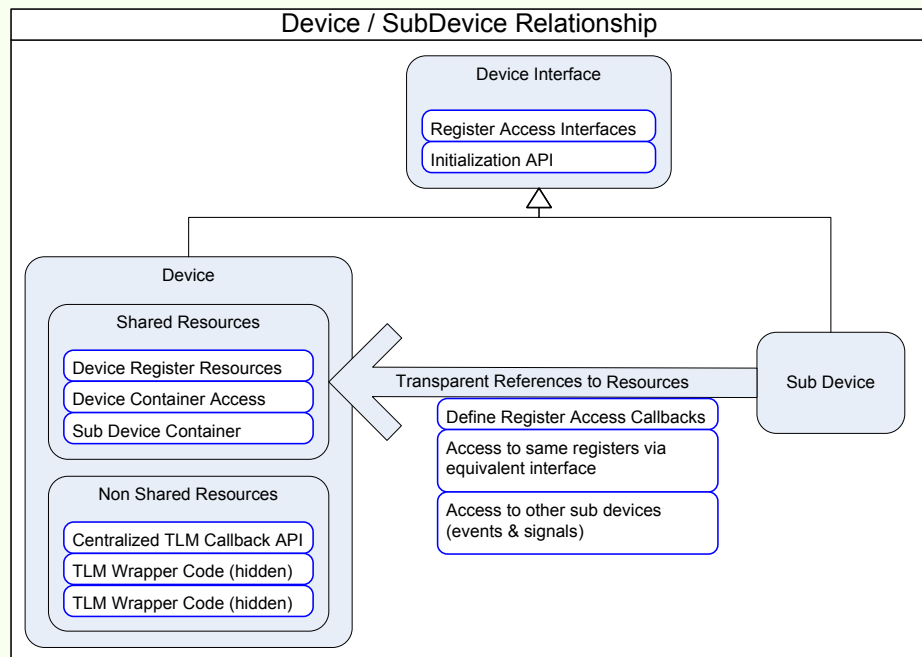
Registers as Stimuli (TLM Independence Master)



- Enhance Development while achieving TLM & Timing Independence
 - Support Complex Register Interfaces & Events
 - Enable Reusable Models
 - Provide a Device – Sub Device Model Code Base
 - Improve Source Code Readability
 - Code Faster

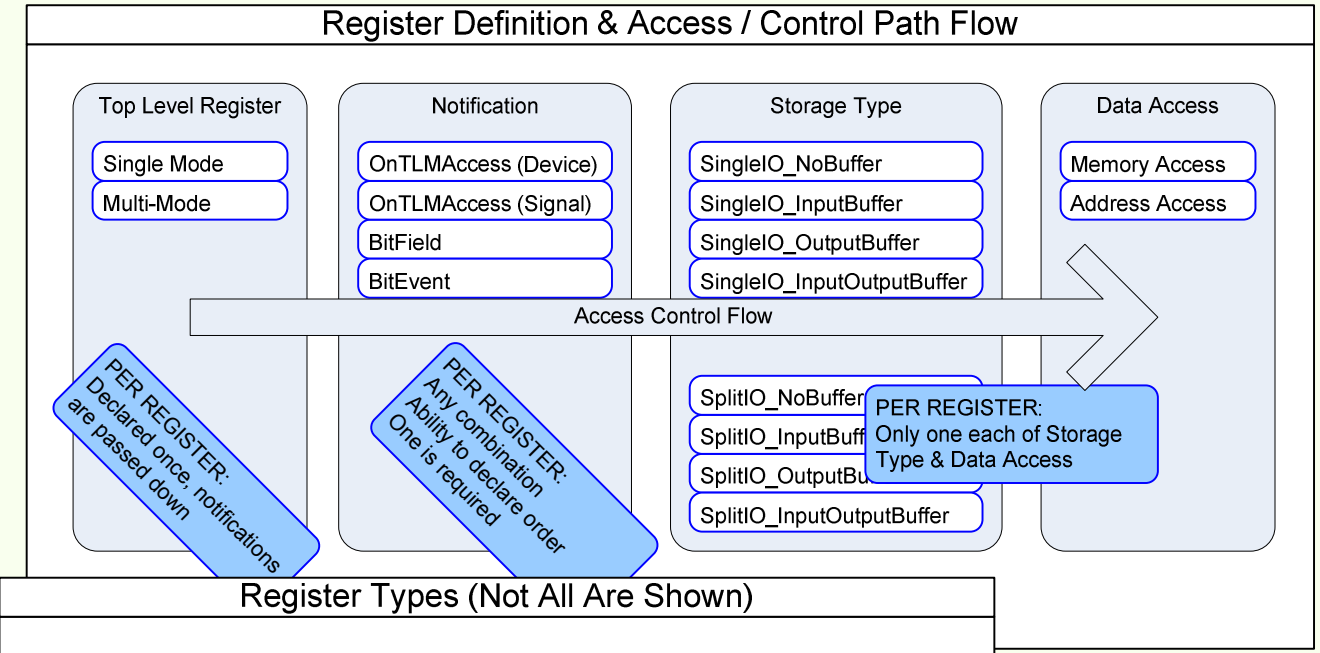


Device Framework

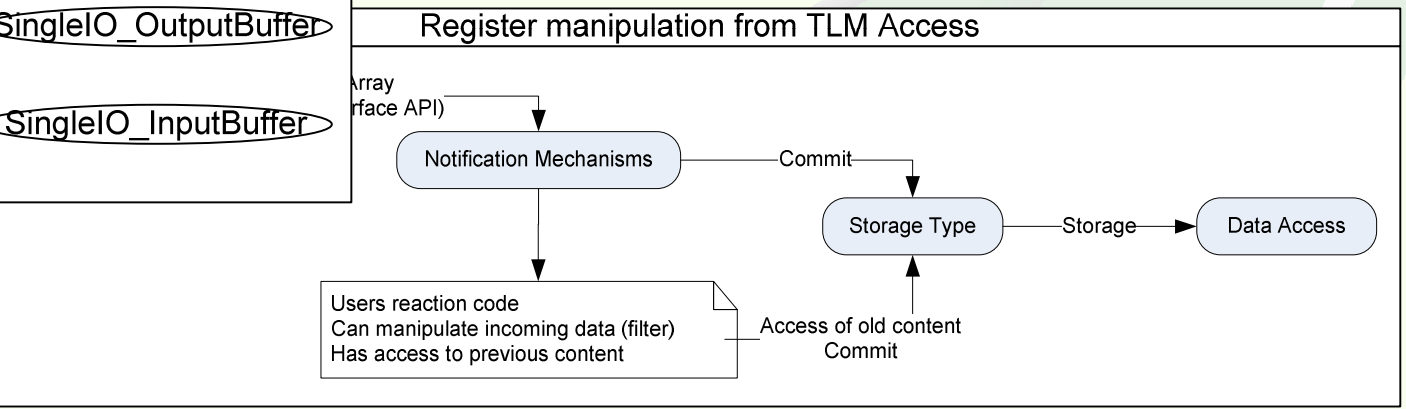
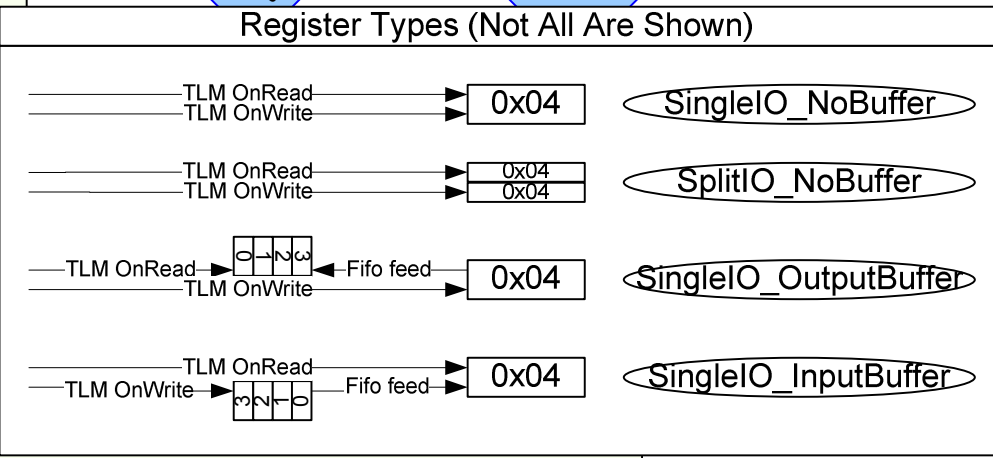


- Device class provides center point of access
- Sub-Device enables modular development of complex models
- Heavy weight ports allows TLM transactions between models w/o TLM access

Register Framework (Register Types - concept)

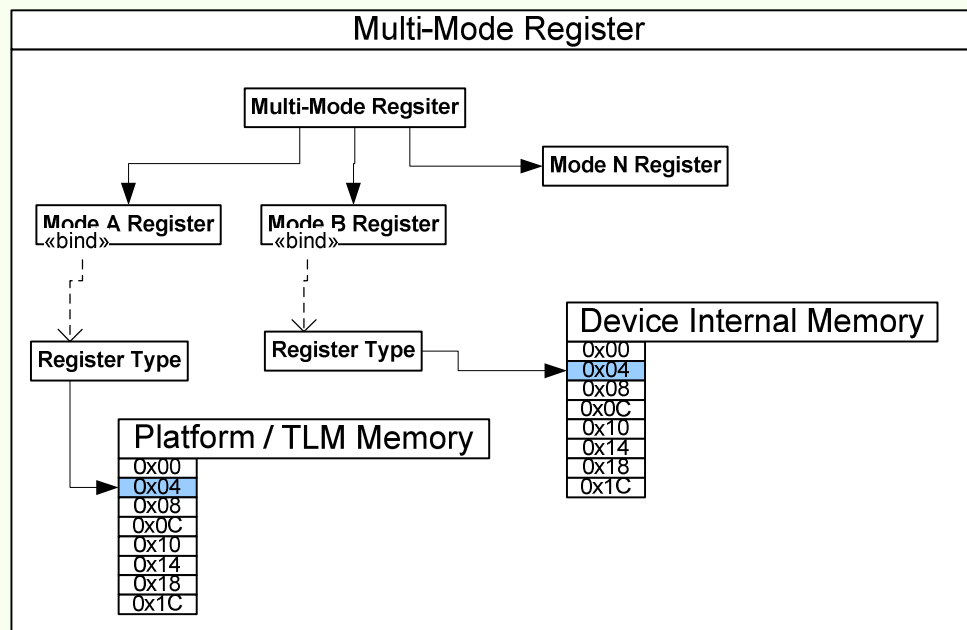


- Multiple Change Notification Types
- Complex Input Output Chain (Fifo register)
- Only NoBuffer implementations are complete.



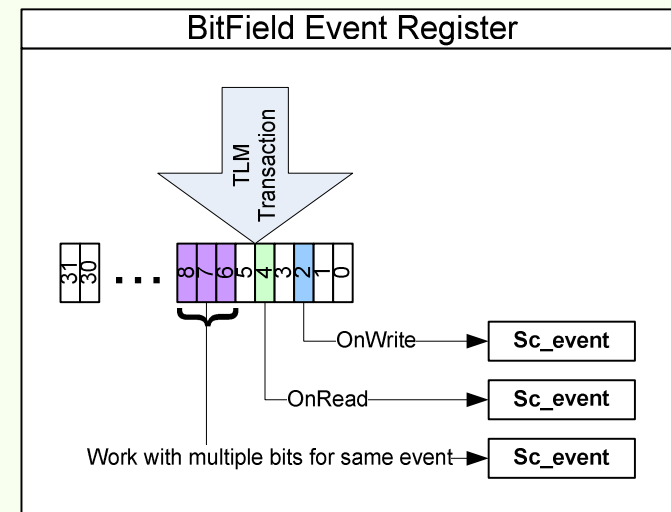
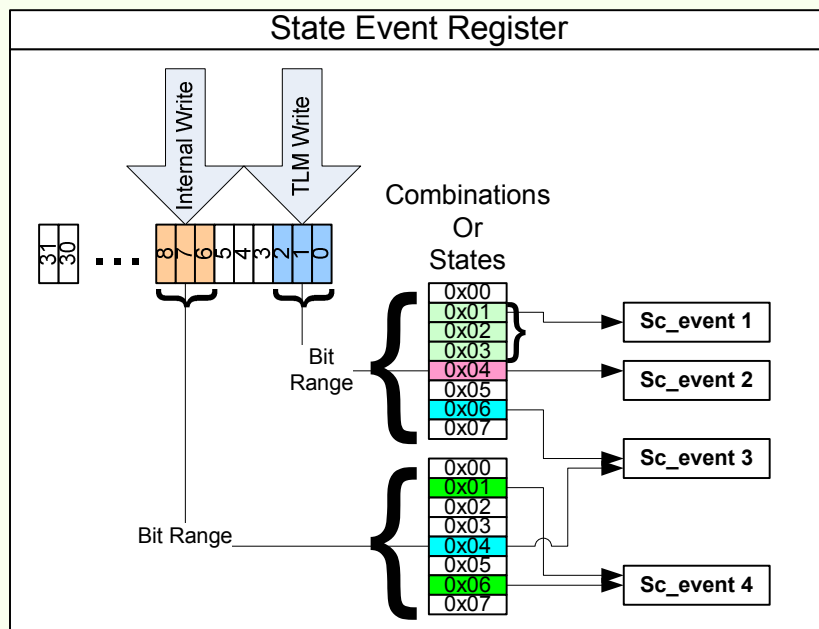
Register Framework (Multi-Mode) – Not Available

- Support mode change based on event (using a switch)
- Easily enable development of multi-memory devices (flash)



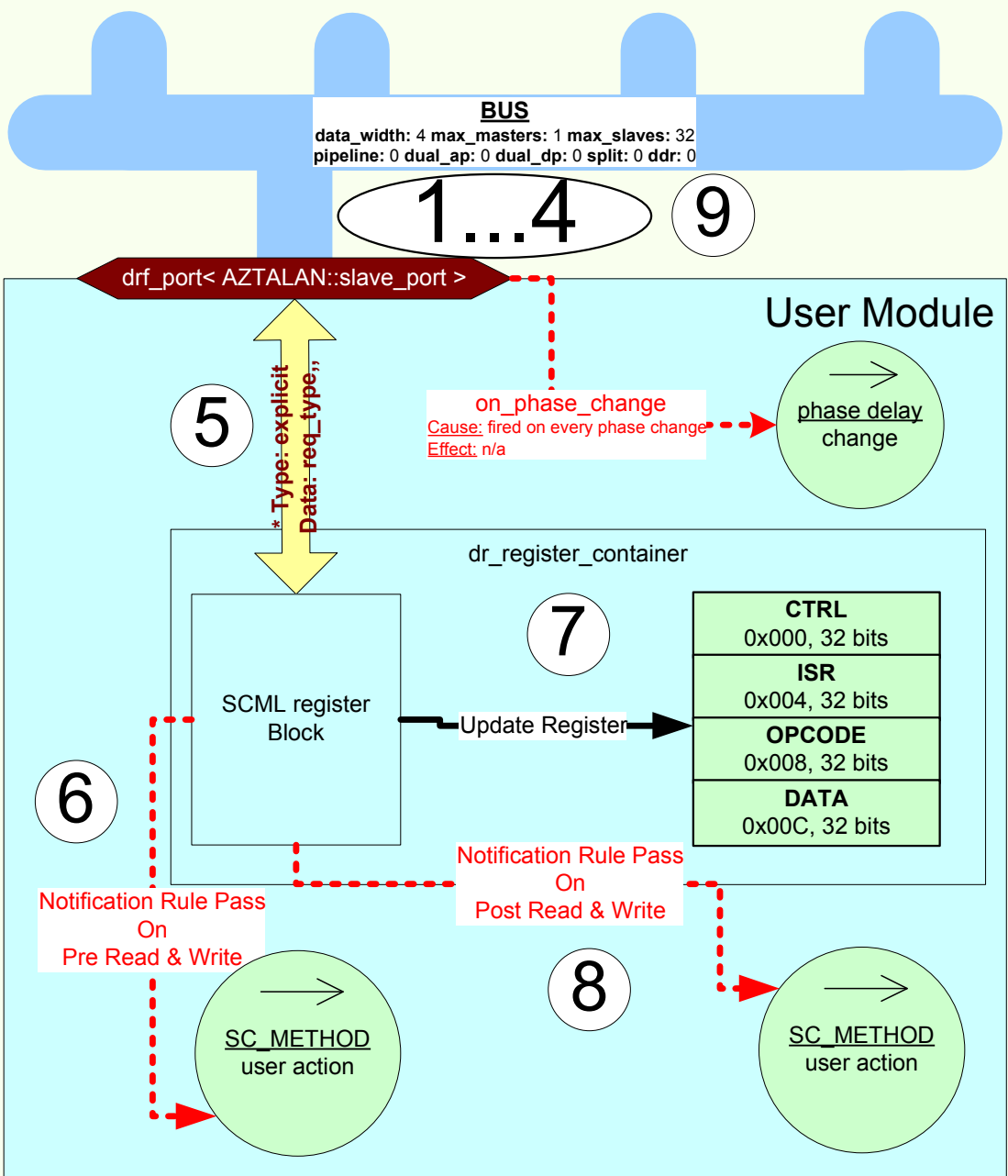
Register Framework (Notification Rules)

- Register Notification Event Rules
 - TLM->Register Pre & Post Read Events
 - TLM->Register Pre & Post Write Events
 - Bit Change
 - Pattern Match (on change)



DRF Enables Second Level Decode through Notification Rules

Notification Rule Execution Overview



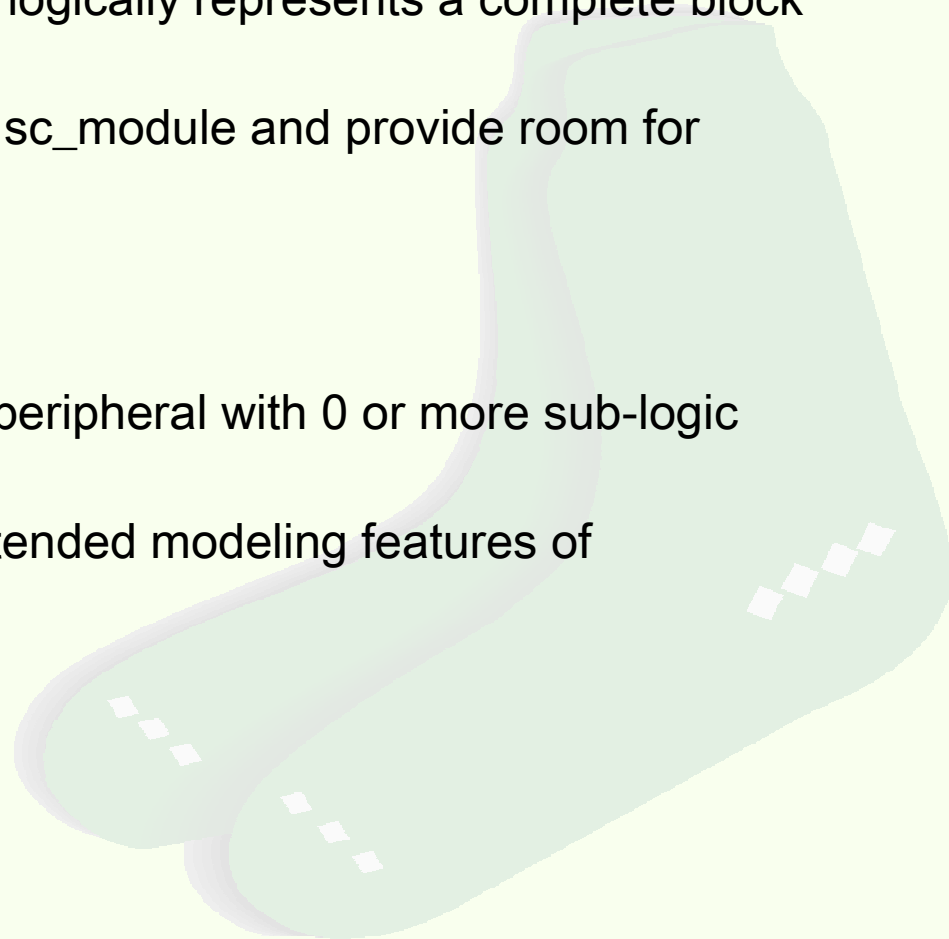
- 1-4: Bus phases based on port type
- 5: Read or Write request at appropriate phase
- 6: Pre – notification rules execute
 - true evaluations stimulate model
 - false evaluations do nothing
 - Purpose:
 - Reads which calculate register data
 - Prep model for incoming writes
 - enable / disable other notification rules in post-write
- 7: Read register data or Write register data
- 8: Post – notification rules execute
 - true evaluations stimulate model
 - false evaluations do nothing
 - Purpose:
 - Stimulate model based on write data change
 - Stimulate model after bus has read current data (flush)
- 9: drf_port completes transaction with bus

GreenReg API (Device)

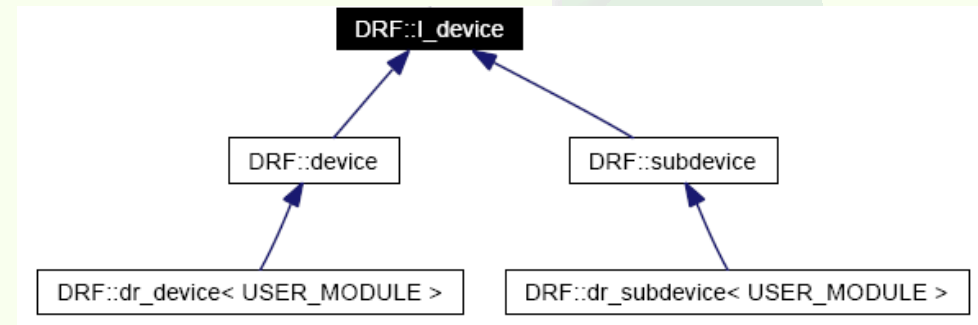


Device Package

- Scope
 - Provide separation, containment and traversal of specialized modeling elements that are not part of the Device Package.
 - Enable a unified hierarchy that more logically represents a complete block of capability.
 - Reduce overhead of instantiating an `sc_module` and provide room for future SystemC based automation.
- Use Cases
 - Any model representing a complete peripheral with 0 or more sub-logic blocks.
 - Any model which will utilize other extended modeling features of GreenReg.



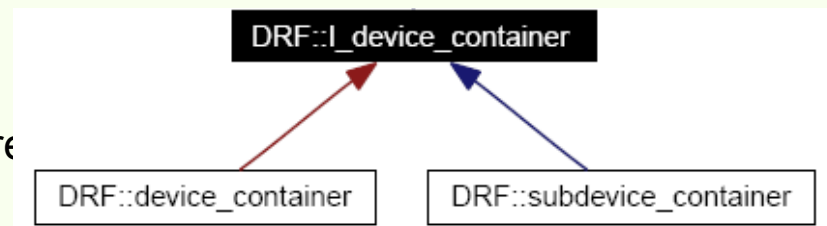
- GreenReg Internal
 - I_device
 - Common Interface for device relationships
 - device
 - Responsible for feature implementation of device hierarchy functionality
 - subdevice
 - Responsible for feature implementation of subdevice hierarchy functionality
- GreenReg User Interface
 - gr_device
 - Top level encapsulation of complete functional unit
 - Abstracts device containment structure binding
 - gr_subdevice
 - Recursive logical block containment node.
 - Abstracts device containment structure binding



Device Container Classes

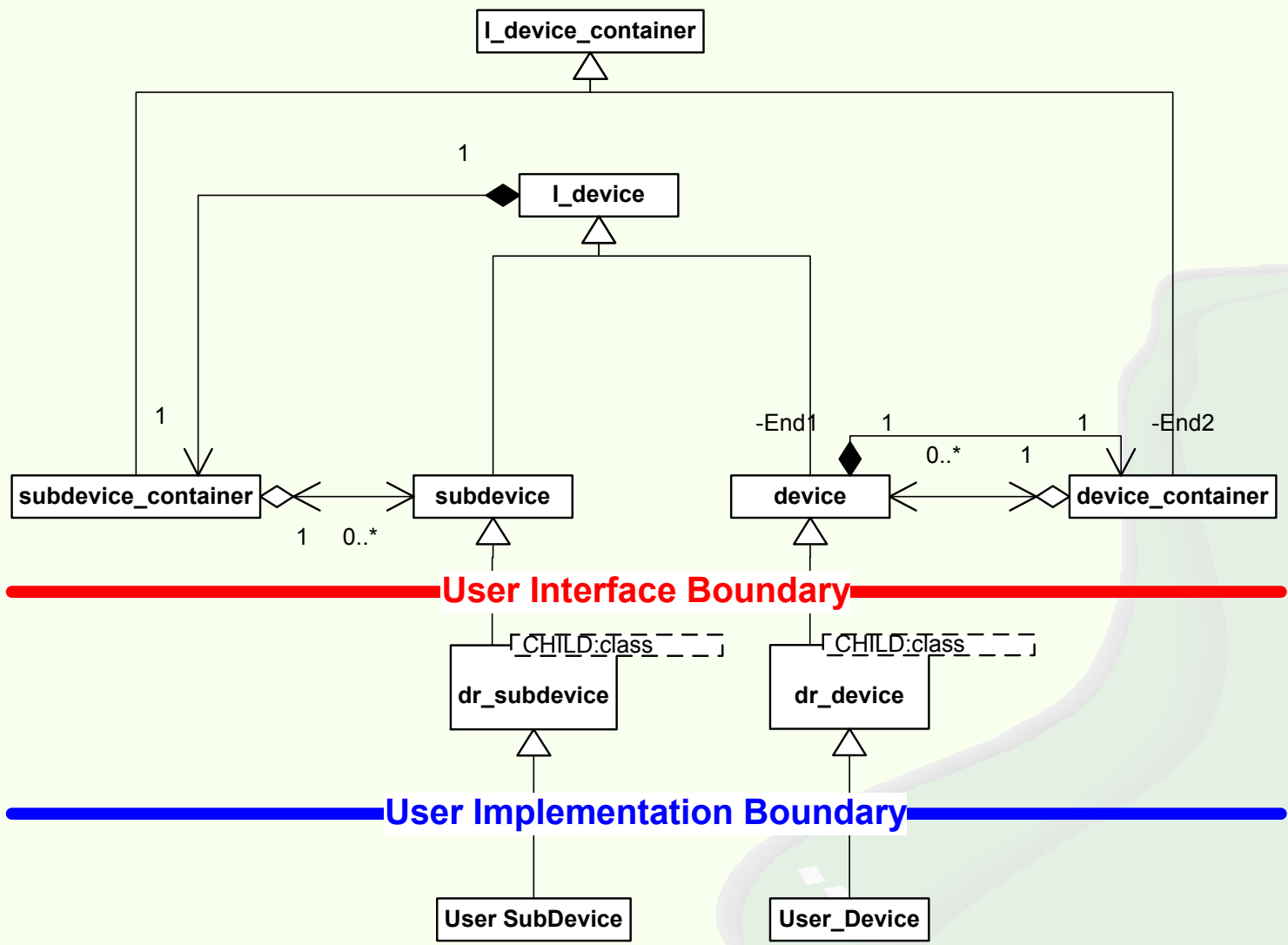
- GreenReg Internal

- I_device_container
 - Common Interface for device container feature
- subdevice_container
 - Responsible for feature implementation of device hierarchy functionality
 - Instantiated in both the device & subdevice classes
- device_container
 - May contain only devices
 - Each device has one to enable hierarchical containment of devices with register blocks



Global Device Container: `gs::reg::g_gr_device_container`

Device Inheritance & Containment



User Interface (gr_device)

- Description
 - Defines a top level encapsulation of reference for a complete functional unit
 - Provides containment structures for other GreenReg elements
 - Provides configuration accesses
- Inheritance
 - `class my_device : public gr_device`
- Constructor
 - Params: `sc_module_name`, address mode, dword size (ALIGNED) or # registers (INDEXED) of register block, parent
 - parent is either the parent `gr_device`, or NULL if it is a top level device
 - `: gr_device(_name, ALIGNED_ADDRESS, 2, NULL)`
- Accessibility
 - Attributes, components and sub-devices are accessed by ["name"]
 - Registers are accessed by either ["name"] or [0xoffset]
 - 'a' – attribute & switch container
 - 'c' – component container
 - 'r' – register container
 - 'rc' – register container "containment"
 - 'sd' – sub-device container
 - 'd' – device container
 - `m_parent` – pointer to parent device, this pointer CAN BE NULL, so check before use

User Interface (gr_device)

- register_container_addressing_mode_e
 - ALIGNED_ADDRESS – strict alignment forcing 32 bit registers to 32 bit boundaries, 16 to 16, 8 to 8
 - INDEXED_ADDRESS – Allows non-aligned offsets to have any size register (32 bit max for now)



User Interface (dr_subdevice)

- Description
 - Defines a recursive encapsulation of sub-functional units
 - Provides containment structures for other DRF elements
 - Provides configuration accesses
- Inheritance
 - User sub-device must inherit from gr_subdevice as a type of itself
 - `class my_subdevice : public gr_subdevice`
- Constructor
 - Params: `sc_module_name` from user device, parent node (`I_device`)
 - `: gr_subdevice(_name, _parent)`
- Accessibility
 - Attributes, components and sub-devices are accessed by ["name"]
 - Registers are accessed by either ["name"] or [0xoffset]
 - 'a' – attribute & switch container
 - 'da' – attribute & switch container – top devices attribute container
 - 'c' – component container
 - 'r' – register container – reference to top devices register container, NOT LOCAL, may not be overwritten.
 - 'rc' – register container "containment" – reference to top devices rc, NOT LOCAL, may not be overwritten.
 - 'sd' – sub-device container
 - `m_parent` – reference access to parent node (`I_device`)

Using the Device Package

- Global or External Access

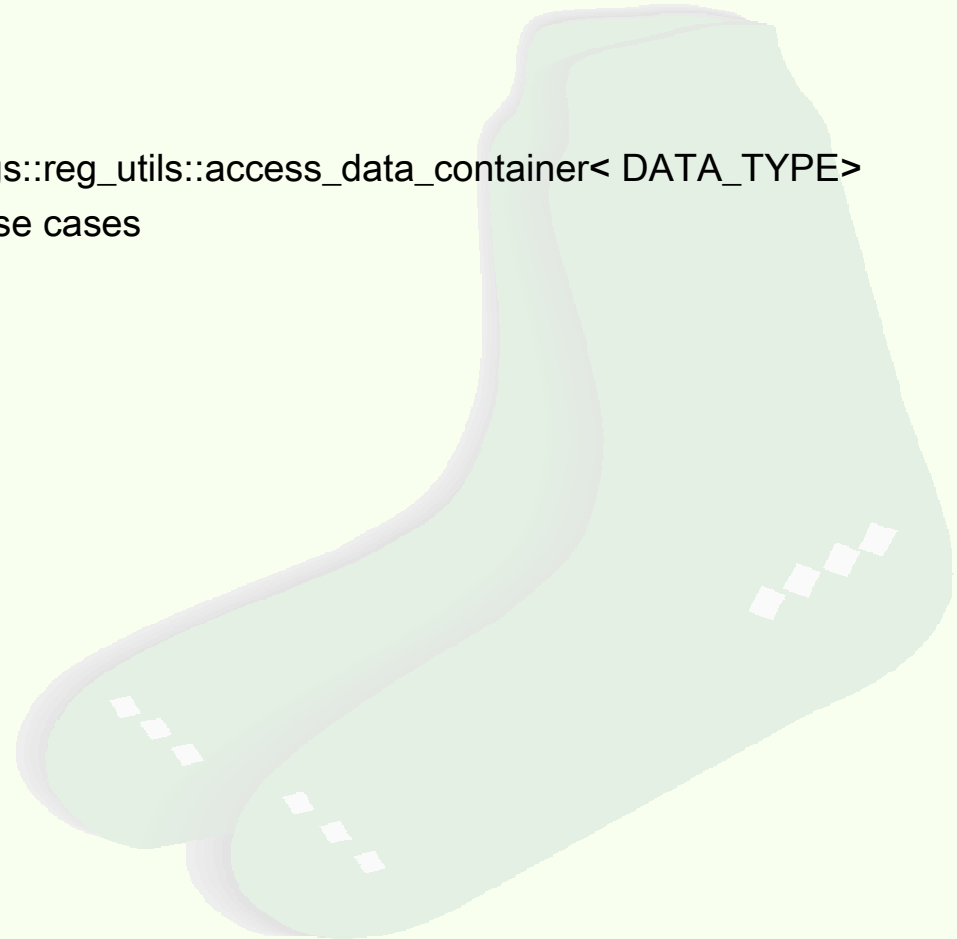
- `gs::reg::g_gr_device_container["device_name"].sd["subdevice_name"].sd["subdevice_name"].{Accessibility}`
- `gs::reg::g_gr_device_container["device_name"].d["encapsulated_device_name"].{Accessibility}`

- Internal Access

- `sd["subdevice_name"].{Accessibility}`
- `m_parent.sd["subdevice_name"].{Accessibility}`

- Container API's

- `device_container` & `subdevice_container` inherit from `gs::reg_utils::access_data_container< DATA_TYPE >`
 - `DATA_TYPE` is either `device` or `subdevice` in these cases
- `DATA_TYPE` & operator `[]` (`id` or `"key"`)
- `bool has_data_elements()`
- `unsigned int number_of_data_elements()`
- `vector< unsigned int > get_all_data_ids()`
- `std::string get_data_key(id)`
- `DATA_TYPE * get_data (id or "key")`



GreenReg API

(RegisterContainer & Register)



Register Package

- Scope
 - Provide containment, flexible register description, and intuitive access methodology not available in leading industry solutions.
 - Enable an Industry First in 3rd tier register decoding
 - 1st tier encompasses identification of the owning device (BUS)
 - 2nd tier encompasses identification of a specific register, and applying write masks
 - 3rd tier encompasses rule based bit decoding and pattern matching
 - Usually implemented in a giant switch
 - Provide Industry with a Register API that can bolt to vendor API's easily (under the hood).
 - Excellent reusability potential
- Use Cases
 - Any logic block that needs registers
 - Register use model where data changes will drive other logic blocks in the model

A Note on GreenReg Registers

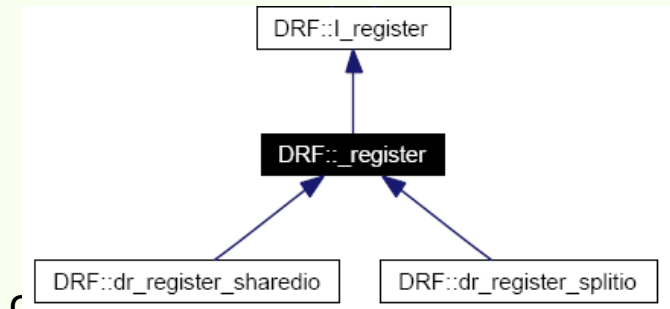
- Capabilities Enablement
 - Focuses more heavily on accessibility and stimulus generation than any other Industry API.
 - Size & Scope of the entire GreenReg Register API is too large to cover in a single training session.



Register Classes

- GreenReg Internal

- `_register`
 - Implementation of common access methods.
- `gr_register_sharedio`
 - Specialized implementation for shared data buffer.
 - May not be created and bound by the user.
- `gr_register_splitio`
 - Specialized implementation for separate input and output data buffers.
 - May not be created and bound by the user.

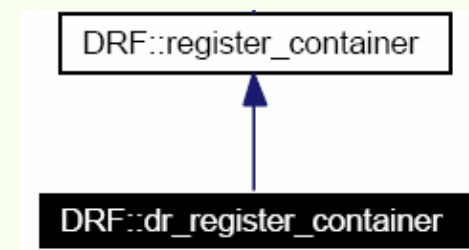


- GreenReg User Interface

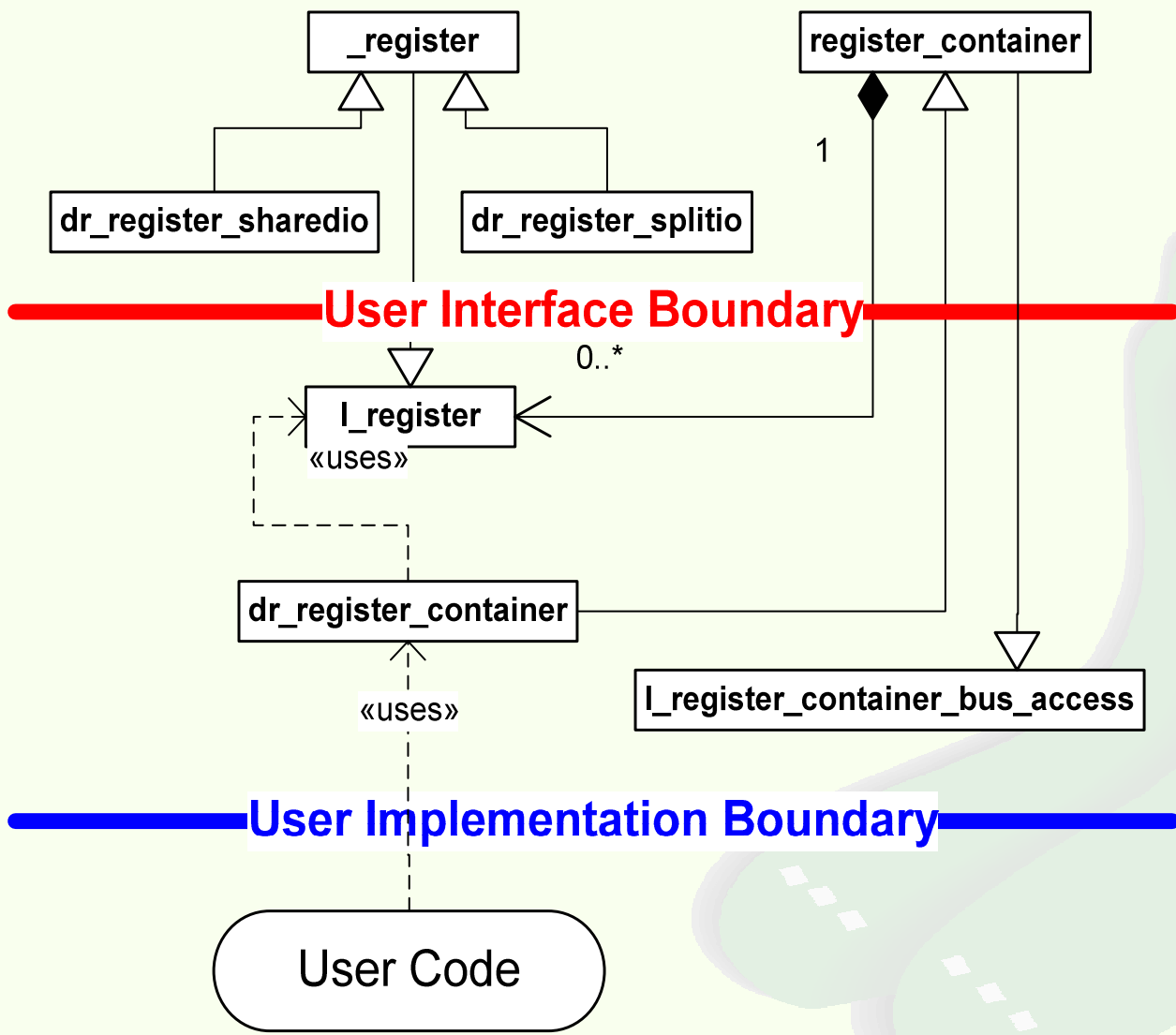
- `I_register`
 - Interface defining all public access methods.

Register Container Classes

- GreenReg Internal
 - register_container
 - Container of I_register *
- GreenReg User Interface
 - gr_register_container
 - Provides only API to create registers
 - I_register_container_bus_access
 - Provides a simple interface for bus accesses to the registers
 - Causes bus side stimulus to occur



Register Inheritance & Containment



User Interface (gr_register_container)

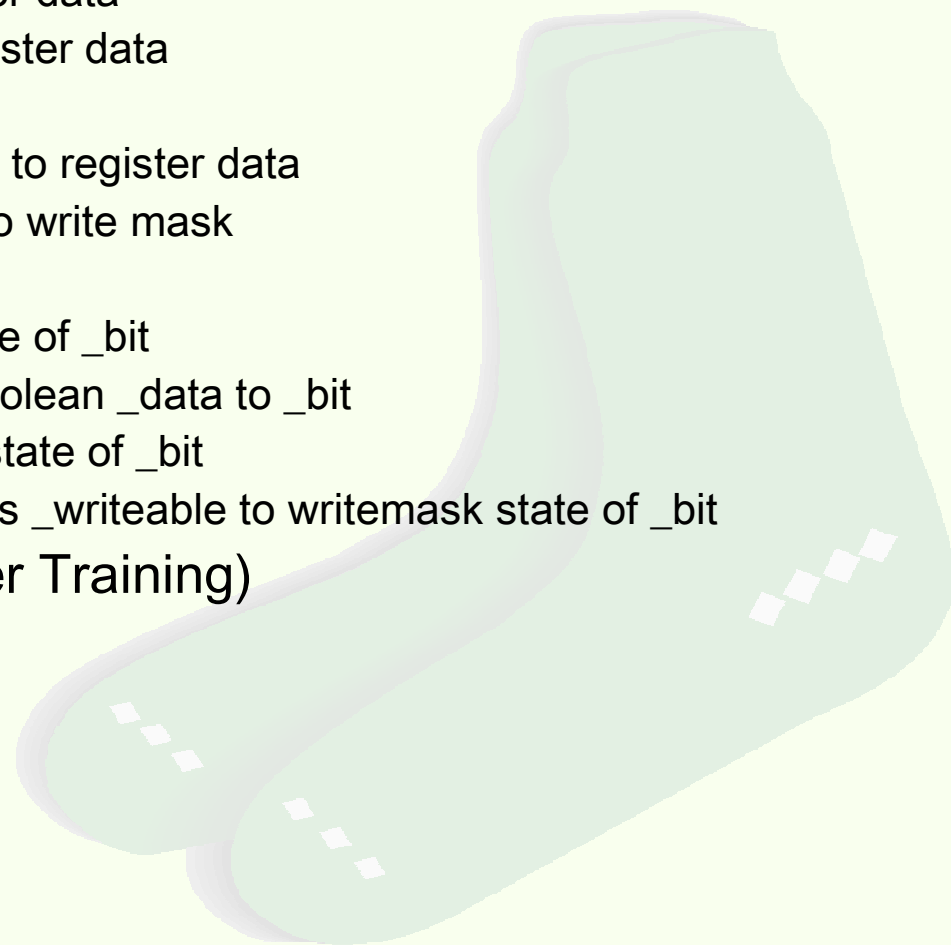
- Instantiation
 - `gs::reg::dr_register_container my_container;`
 - By default 1 register container already exists in `gr_device`, accessible by 'r'
- Constructor
 - Params: `sc_module_name`, address mode, dword size (ALIGNED) or # registers (INDEXED) of reg. block
 - `my_container("registers", ALIGNED_ADDRESS, 2)`
- Methods of Interest
 - `create_register("reg", "desc", 0x01, gs::reg::SPLIT_IO, 0xab, 0xff, 32, 0x0);`
 - creates a single register
 - Params: name, description, offset, configuration, initial value, write mask, reg width, lock mask (ignore)
 - `create_register_block("desc", 0x02, 0x4c, gs::reg::SINGLE_IO, 0x00, 0xff, 32, 0x0);`
 - creates a block of registers
 - Params: description, start offset, end offset, configuration, initial value, write mask, reg width, lock mask (ignore)
- Container API's
 - `register_container` inherit from `gs::reg_utils::addressable_owning_container< KEY, VALUE>`
 - KEY - an unsigned integer representing the offset of the register
 - DATA_TYPE - `I_register`
 - VALUE & operator `[] (id)`

Configuration Enums (gr_register_container)

- Configuration parameter in Create Register is a binary 'or' combination of the following:
 - register_type_e (NOT YET AVAILABLE)
 - STANDARD_REG – (default) describes a normal register
 - MULTI_MODE_REG – describes a container of registers at specified offset
 - register_io_e
 - SINGLE_IO – (default) reads and writes access same buffer
 - SPLIT_IO – reads and writes access separate buffers
 - register_buffer_e (NOT YET AVAILABLE)
 - SINGLE_BUFFER – (default) buffer is only 1 level deep, a read or write triggers logic immediately
 - DOUBLE_BUFFER – buffer is 2 levels deep, a write pushes data to the second buffer, triggering logic
 - register_data_e (NOT YET AVAILABLE, MAY BE REMOVED)
 - FULL_WIDTH – (default) register uses code base for width optimized accesses
 - BIT_ONLY – register uses code base for bit optimized accesses
 - BIT_RANGE – register uses code base for bit_range optimized accesses

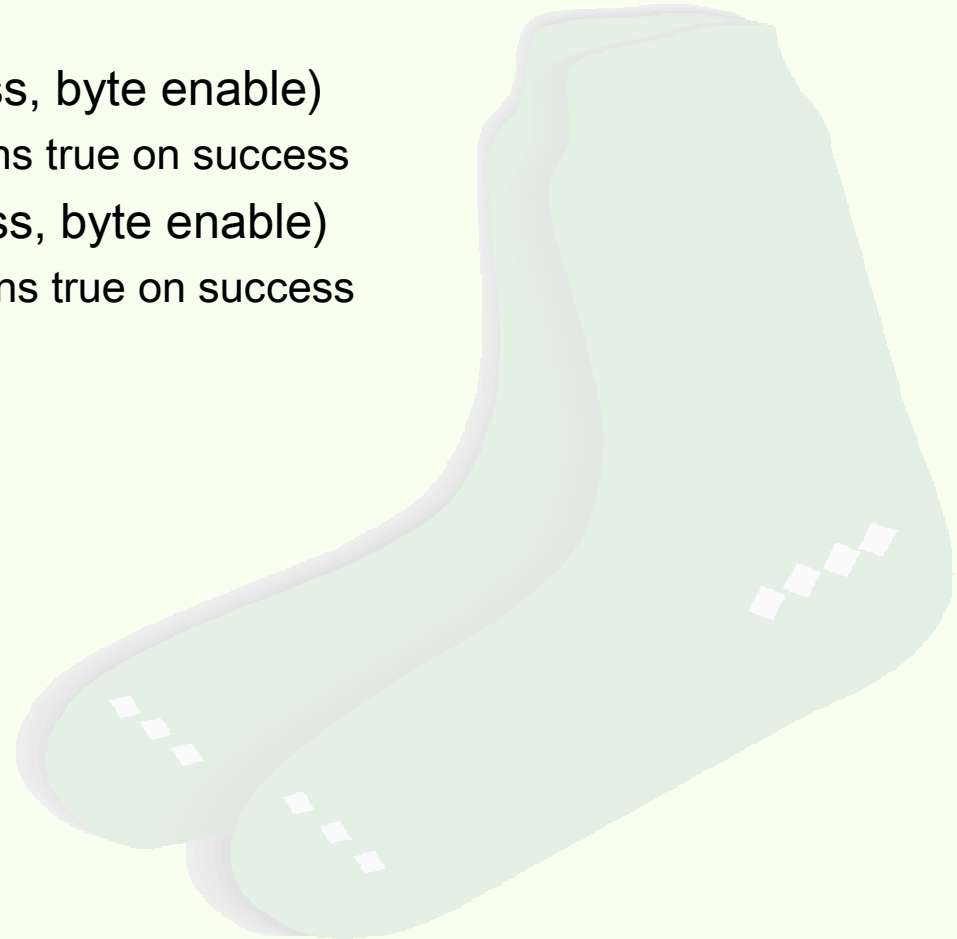
User Interface (I_register)

- Methods of Interest
 - `get_register_type()` - returns an unsigned int representing the configuration
 - `register_type_e`, `register_io_e`, `register_buffer_e`, `register_data_e`
 - `get_width()` – returns the width in bits of the register
 - `operator uint32 ()` – returns copy of register data
 - `operator = (uint32)` – assigns value to register data
 - `get()` – returns copy of register data
 - `set(uint32) & put(uint32)` – assigns value to register data
 - `set_write_mask(uint32)` – assigns value to write mask
 - `get_write_mask()` – returns write mask
 - `bit_get(uint32 _bit)` – returns boolean state of `_bit`
 - `bit_set(_bit, _data) & bit_put` – assigns boolean `_data` to `_bit`
 - `bit_is_writable(_bit)` – returns writemask state of `_bit`
 - `bit_set_writable(_bit, _writeable)` – assigns `_writeable` to writemask state of `_bit`
- Accessibility (Refer to Advanced Register Training)
 - 'i' – input buffer
 - 'o' – output buffer
 - 'b' – bit container
 - 'br' – bit range container



User Interface (`I_register_container_bus_access`)

- Notes
 - Inherited by `register_container`
 - Documented separately because it is a fundamental access API
- Methods of Interest
 - `bus_read(data, word aligned address, byte enable)`
 - stimulates a bus side read and returns true on success
 - `bus_write(data, word aligned address, byte enable)`
 - stimulates a bus side write and returns true on success



Using the Register Package (Register Allocation)

- Word Alignment
 - GreenReg registers support 8, 16, & 32 bit registers in the same block
 - A tight (and logical) convention must be followed:
 - 32 bit registers can only be assigned to dword aligned addresses
 - 16 bit registers can only be assigned to word aligned addresses
 - 8 bit registers can only be assigned to byte aligned addresses



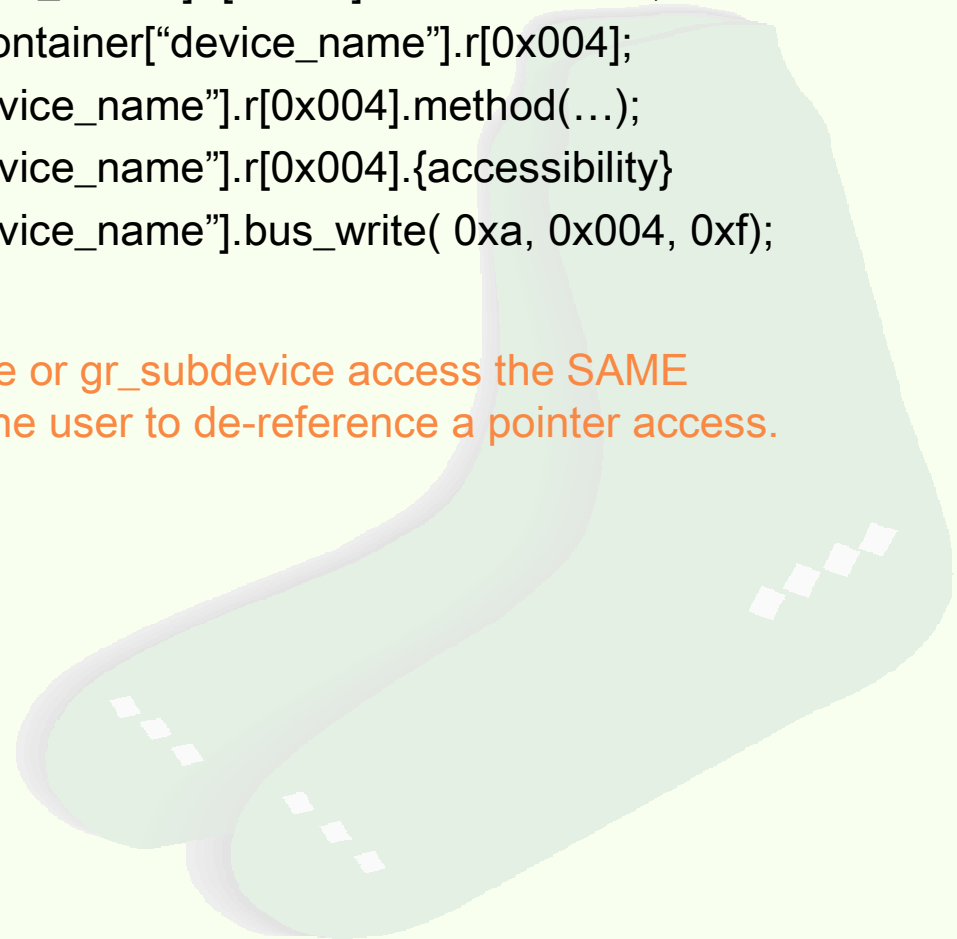
Using the Register Package

- Global or External Access

- WARNING:** This access description works with the registers from the module, not the bus perspective!
- `gs::reg::g_gr_device_container["device_name"].r[0x004] = 0x12345678;`
- `uint my_int = gs::reg::g_gr_device_container["device_name"].r[0x004];`
- `gs::reg::g_gr_device_container["device_name"].r[0x004].method(...);`
- `gs::reg::g_gr_device_container["device_name"].r[0x004].{accessibility}`
- `gs::reg::g_gr_device_container["device_name"].bus_write(0xa, 0x004, 0xf);`

- Internal Access

- NOTE:** All accesses from a `gr_device` or `gr_subdevice` access the SAME register container without requiring the user to de-reference a pointer access.
- `r[0x004] = 0x12345678;`
- `uint my_int = r[0x004];`
- `r[0x004].method(...);`
- `r[0x004].{accessibility}`
- `r.bus_write(0xa, 0x004, 0xf);`



GreenReg API

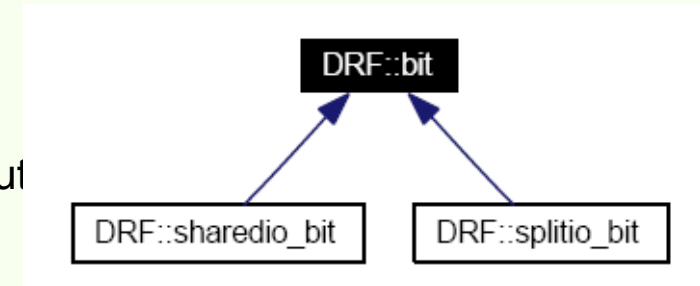
(BitRange & Bit)



- Scope
 - Intuitive bit level access to register data
 - Dynamic on the fly allocation reduces memory impact without speed loss
- Use Cases
 - Any case where bit use is heavily required




- GreenReg Internal
 - sharedio_bit
 - specialized implementation for shared data buffer
 - splitio_bit
 - specialized implementation for separate input and output
- GreenReg User Interface
 - bit
 - class providing bit level method implementation



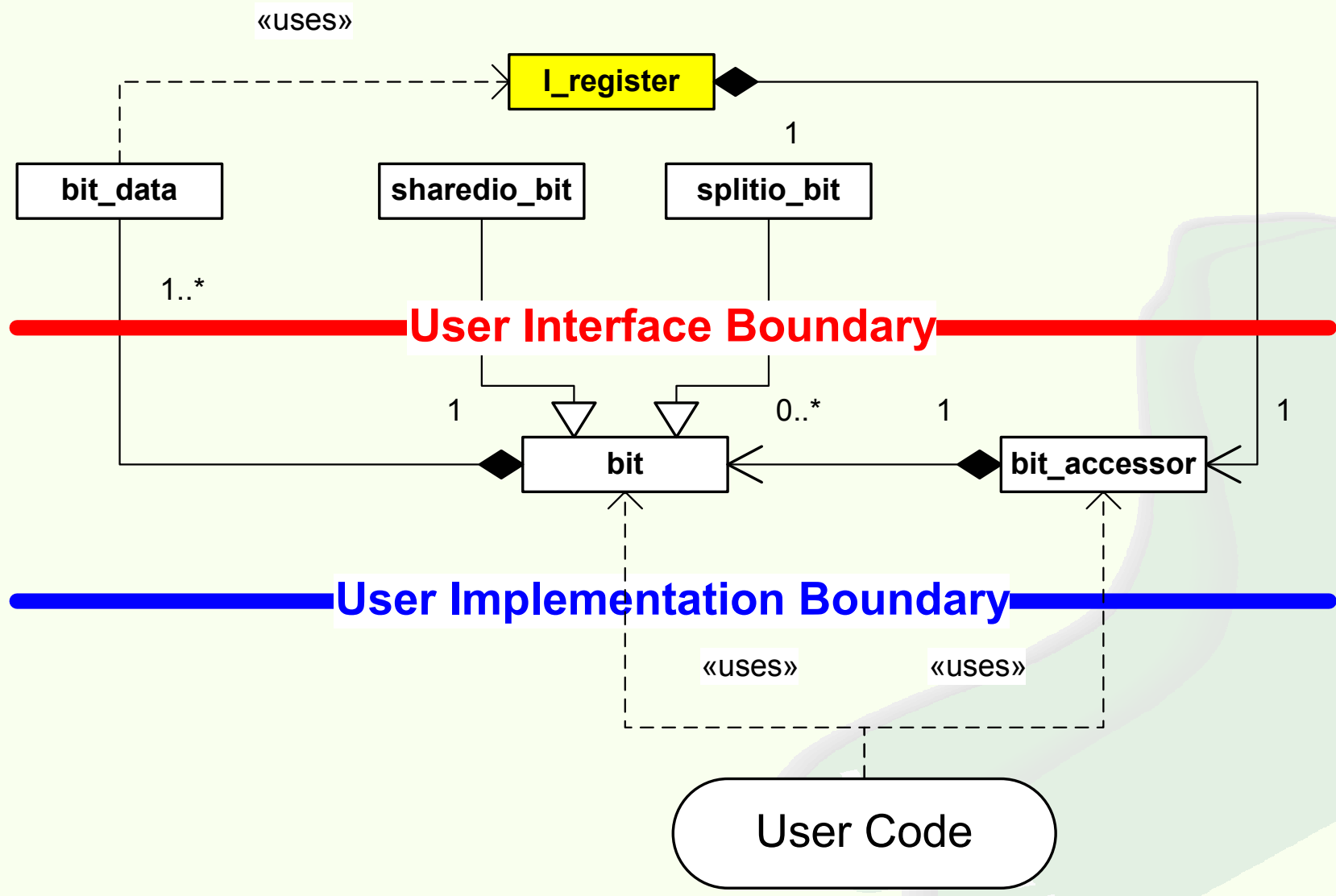
Bit Container Classes

- GreenReg User Interface
 - bit_accessor
 - Container of bits
 - API dynamically allocates bits that are used upon first access

DRF::bit_accessor



Bit Inheritance & Containment



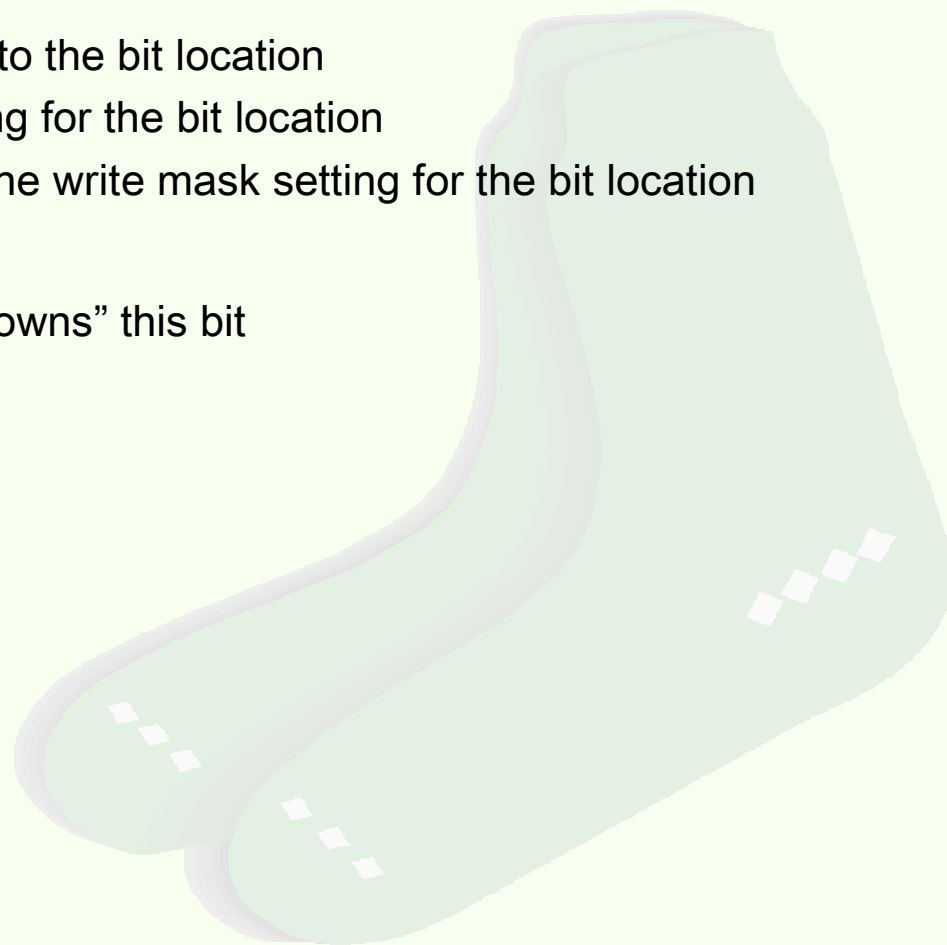
User Interface (bit_accessor)

- Instantiation
 - bit_accessor has a symbiotic relationship with I_register, this is the only place of instantiation possible.
- Methods of Interest
 - bit & operator [] (unsigned int) – returns a bit class at the specified bit index



User Interface (bit)

- Methods of Interest
 - operator `bool ()` – returns the value of the bit location
 - `bit & operator = (bool)` – applies the value to the bit location
 - `get()` – returns the value of the bit location
 - `set(bool) & get(bool)` – applies the value to the bit location
 - `is_writable()` – returns the write mask setting for the bit location
 - `set_writable(bool)` – applies the value to the write mask setting for the bit location
- Accessibility
 - `m_register` – pointer to the register which “owns” this bit
 - `'i'` – input data buffer
 - `'o'` – output data buffer



Using the Bit Package

- Global or External Access
 - **WARNING:** This access description works with the registers from the module, not the bus perspective!
 - `gs::reg::g_gr_device_container["device_name"].r[0x004].b[3] = true;`
 - `bool val = gs::reg::g_gr_device_container["device_name"].r[0x004].b[3];`
 - `gs::reg::g_gr_device_container["device_name"].r[0x004].b[3].method(...);`
- Internal Access
 - `r[0x004].b[3] = true;`
 - `bool val = r[0x004].b[3];`
 - `r[0x004].b[3].method(...);`



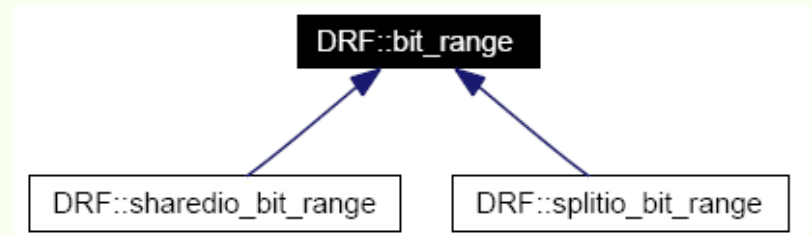
Bit Range Package

- Scope
 - Intuitive Bit Range access to data
 - Intuitive Bit accesses relative to Bit Range starting offset
 - Facilitate 3rd tier register decode with notification rule support in the bit ranges
 - Support multiple overlapping bit ranges
- Use Cases
 - Control & Status Registers



Bit Range Classes

- GreenReg Internal
 - sharedio_bit_range
 - specialized implementation for shared data buffer
 - splitio_bit_range
 - specialized implementation for separate input and output data buffers
- GreenReg User Interface
 - bit_range
 - class providing bit range method implementation



Bit Range Container Classes

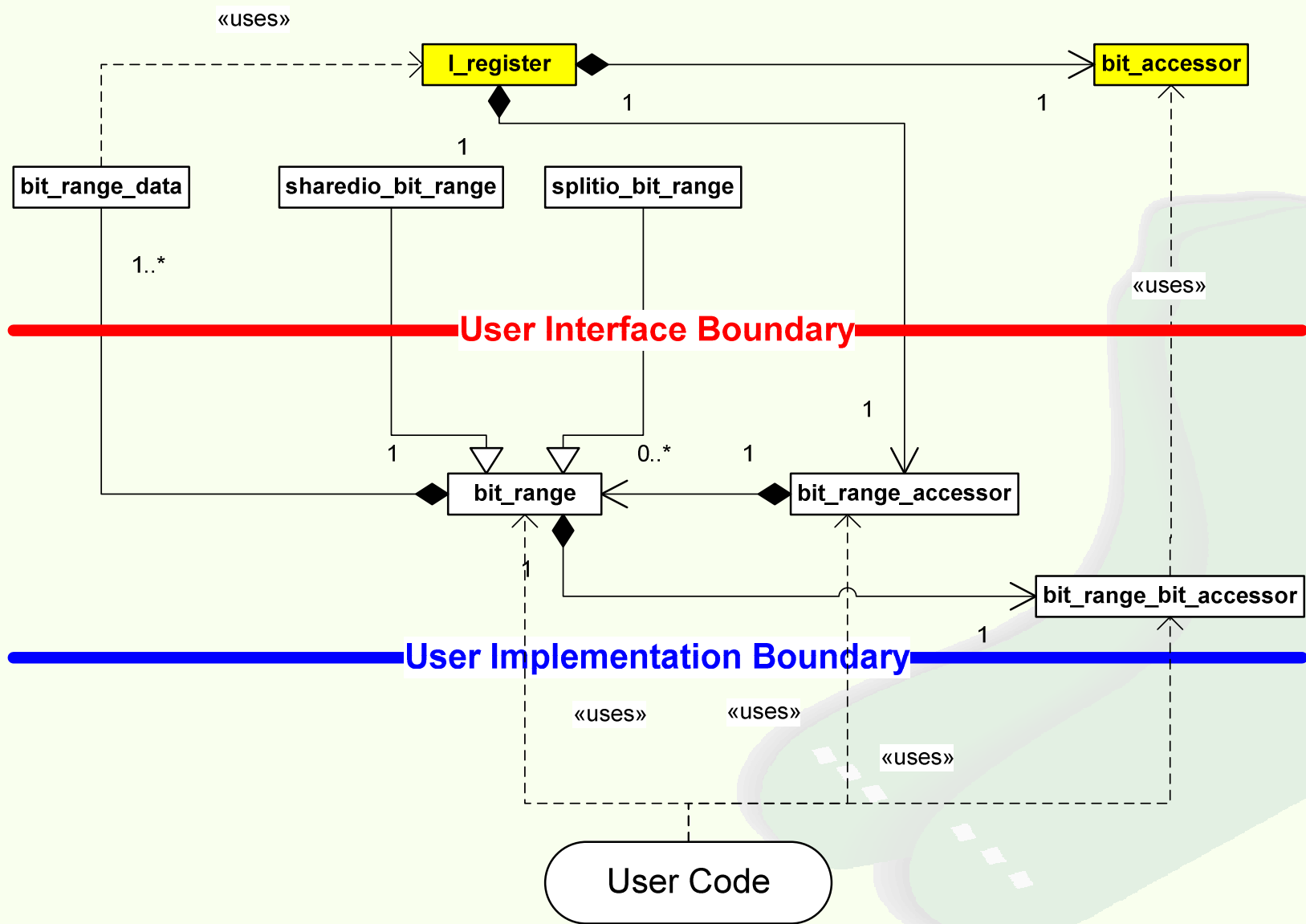
- GreenReg User Interface
 - bit_range_accessor
 - Container of bit_ranges
 - bit_range_bit_accessor
 - Container of bit references
 - Accesses bits relative to bit_range starting offset

DRF::bit_range_accessor

DRF::bit_range_bit_accessor

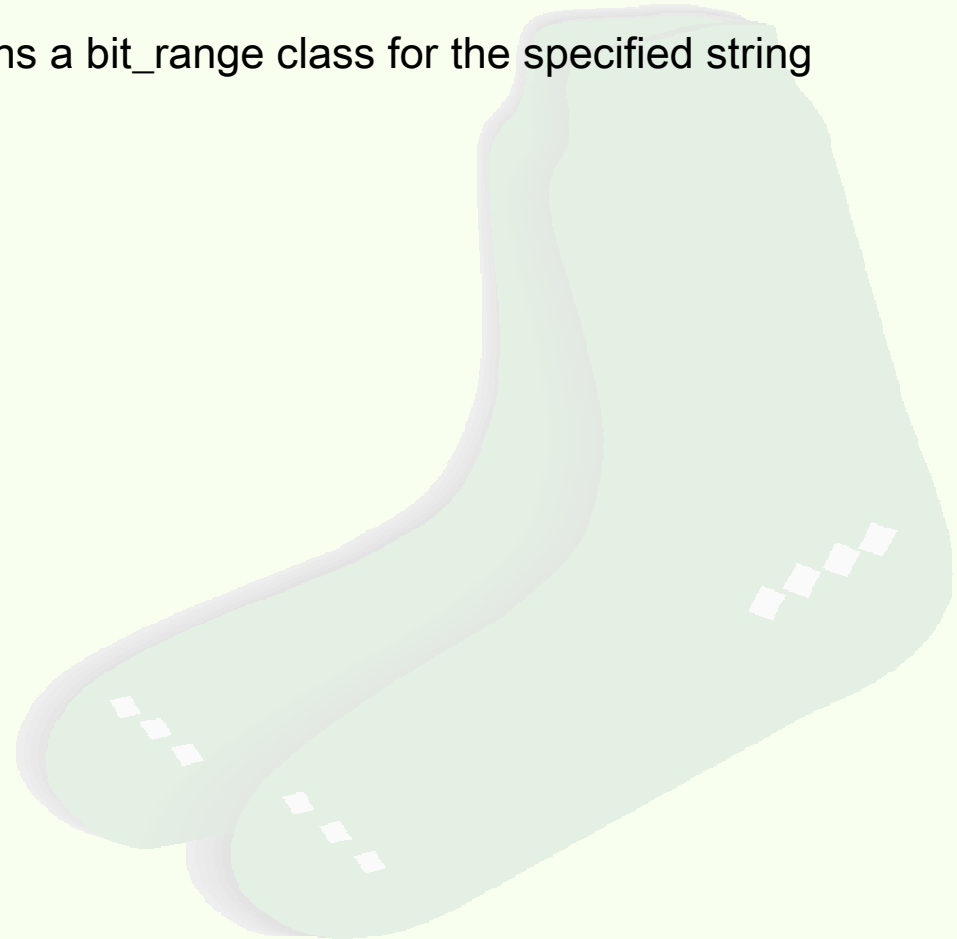


Bit Range Inheritance & Containment



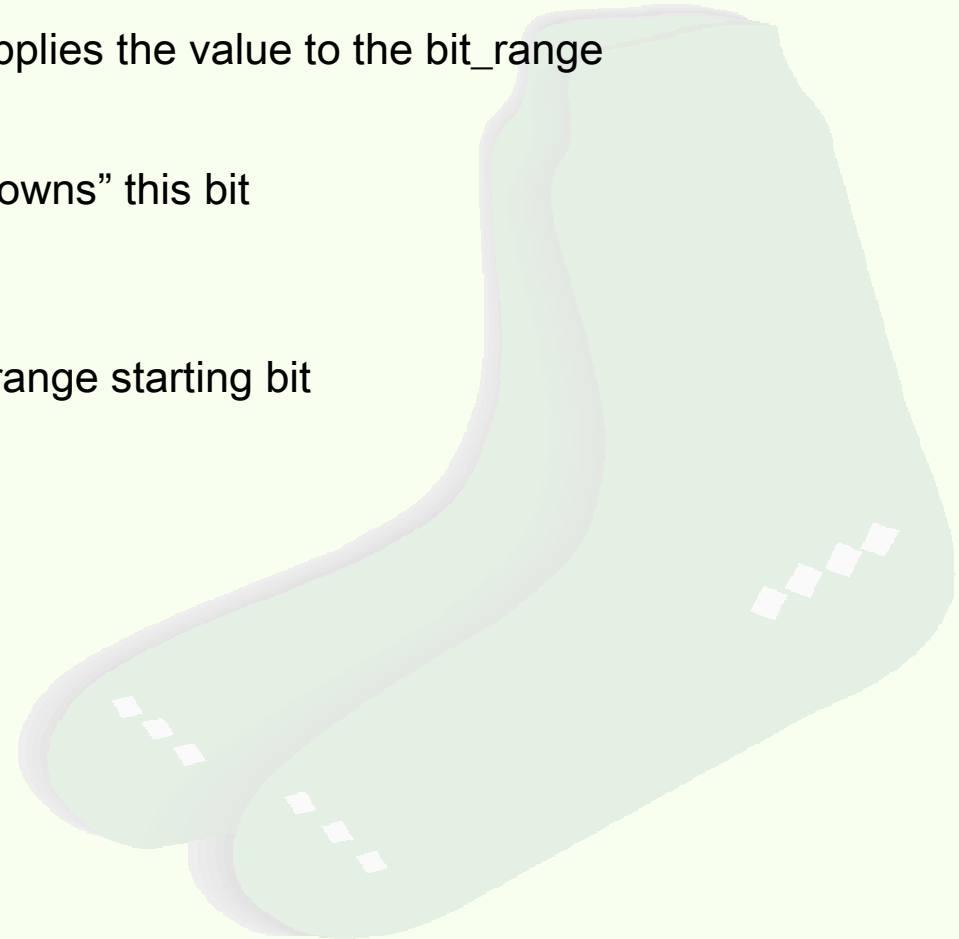
User Interface (bit_range_accessor)

- Instantiation
 - bit_range_accessor has a symbiotic relationship with I_register, this is the only place of instantiation possible.
- Methods of Interest
 - bit_range & operator [] (std::string) – returns a bit_range class for the specified string
 - create(“name”, 3, 7);
 - Params: “name”, starting bit, ending bit



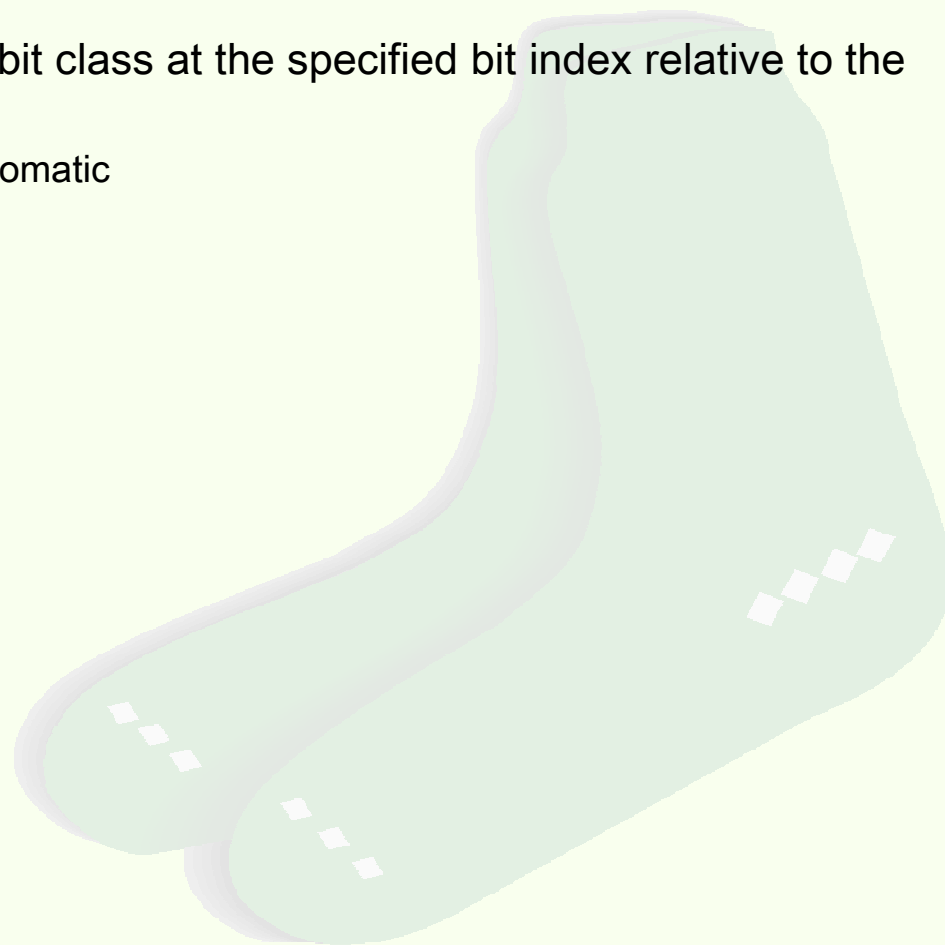
User Interface (bit_range)

- Methods of Interest
 - operator unsigned int () – returns the value of the bit_range
 - bit_range & operator = (unsigned int) – applies the value to the bit_range
 - get() – returns the value of the bit_range
 - set(unsigned int) & get(unsigned int) – applies the value to the bit_range
- Accessibility
 - m_register – pointer to the register which “owns” this bit
 - ‘i’ – input data buffer
 - ‘o’ – output data buffer
 - ‘b’ – bit_range_bit_accessor relative to bit range starting bit



User Interface (`bit_range_bit_accessor`)

- Instantiation
 - `bit_range_bit_accessor` has a sole relationship with `bit_range`, this is the only place of instantiation possible.
- Methods of Interest
 - `bit` & operator `[]` (`unsigned int`) – returns a bit class at the specified bit index relative to the `bit_range`
 - Implies the adjustment to the register is automatic



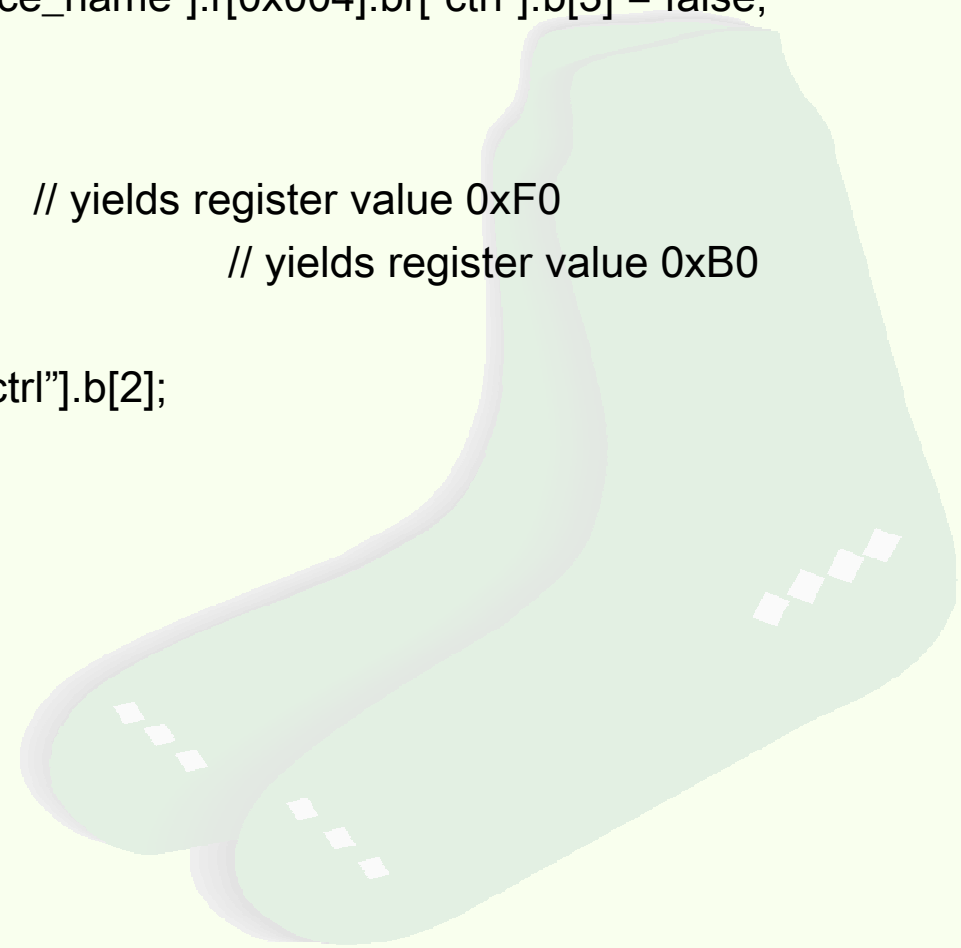
Using the Bit Range Package

- Global or External Access

- `gs::reg::g_gr_device_container["device_name"].r[0x004].br.create("ctrl", 3, 7);`
- `gs::reg::g_gr_device_container["device_name"].r[0x004].br["ctrl"] = 0xF;`
- `gs::reg::g_gr_device_container["device_name"].r[0x004].br["ctrl"].b[3] = false;`

- Internal Access

- `r[0x004].br.create("ctrl", 4, 8);`
- `r[0x004].br["ctrl"] = 0xF;` `// yields register value 0xF0`
- `r[0x004].br["ctrl"].b[3] = false;` `// yields register value 0xB0`
- `unsigned int var = r[0x004].br["ctrl"];`
- `unsigned int boolean = r[0x004].br["ctrl"].b[2];`
- `r[0x004].br["ctrl"].method(...);`
- `r[0x004].br["ctrl"].b[3].method(...);`



GreenReg API (Functions)



GR_FUNCTIONS etc.

- Instead of SC_METHOD, use
 - GR_FUNCTION
 - Called by immediate callback instead of event (more efficient!) or called delayed (see notification rule slides)
 - Do not use SC_METHOD/SC_THREAD specific elements inside
 - Does not need (cannot handle) dont_initialize()
 - GR_FUNCTION_PARAMS
 - Like GR_FUNCTION but gives parameters to called function:
 - `void function(gs::reg::transaction_type* &tr, const sc_core::sc_time& delay)`
 - TODO future: also give bool delayed to identify *if* delayed
 - GR_METHOD (not recommended)
 - Like GR_FUNCTION but is an SC_METHOD additionally
 - Needs dont_initialize()
 - SC_METHOD (not recommended)
 - Possible with notification rule/register with enabled event witch (see slide event switch)

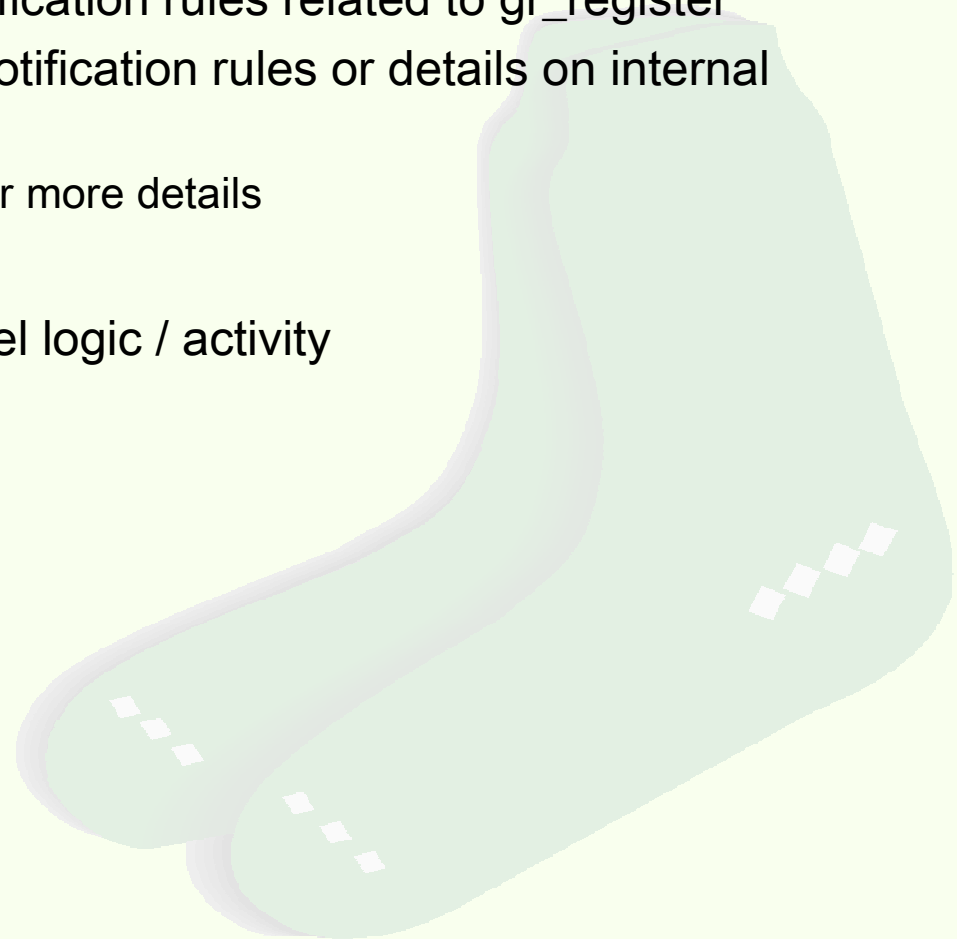
GreenReg API

(Notification Rules & `greenreg_socket<master>`)



Register Package (notification rules)

- Scope
 - 3rd tier decode encompasses rule based bit decoding and pattern matching
 - Creation and use of pre-existing notification rules related to gr_register
 - Does not cover creation of custom notification rules or details on internal mechanisms
 - Refer to Notification Rule Training for more details
- Use Cases
 - Any registers that will stimulate model logic / activity



Using Notification Rules with I_register

- Note:
 - Not all features of I_register were covered in Register Training, please refer back if needed.
 - These methods apply to GreenReg registers and bit ranges
- Methods of Interest
 - add_rule(POST_WRITE, “Post Write CWR”, NOTIFY, ...)
 - Creates a rule internally based on parameters and returns an event associated with the rule.
 - Params: gr_reg_rule_container enum, unique name, gr_reg_rule_type, rule parameters.
 - SHOULD BE DEFINED IN “end_of_elaboration”
 - The following methods return the appropriate notification_rule_container
 - Notification Rule Containers can be used to:
 - enable/disable specific rules (by name)
 - add new rules
 - get_pre_write_rules() – returns container for rules executing before a write to register
 - get_post_write_rules() – returns container for rules executing after a write to register
 - get_pre_read_rules() – returns container for rules executing before a read from register
 - get_post_read_rules() – returns container for rules executing after a read from register
 - get_user_ibuf_write_rules() – returns container for rules executing after write to i buffer
 - get_user_obuf_write_rules() – returns container for rules executing after write to o buffer

Configuration Enums (l_register::add_rule)

dr_reg_rule_container

- PRE_READ – add rule to pre read notification rule container
- POST_READ – add rule to post read notification rule container
- PRE_WRITE – add rule to pre write notification rule container
- POST_WRITE – add rule to post write notification rule container
- USR_IN_WRITE – add rule to model stimulated input buffer write notification rule container
- USR_OUT_WRITE – add rule to model stimulated output buffer write notification rule container

dr_reg_rule_type

- NOTIFY – simplest rule, no parameters, simply notifies for every access
- WRITE_PATTERN_EQUAL – notifies if the written data matches a specified pattern
- READ_PATTERN_EQUAL – notifies if the data to be read matches a specified pattern
- PATTERN_STATE_CHANGE – notifies if the old data to the new data matches a set of change patterns
- BIT_STATE – notifies if a bit state is true (does not care if it was true prior to access)
- BIT_STATE_CHANGE – notifies if bit state changed

Using Notification Rules with I_register

- Global or External Access

- `gs::reg::g_gr_device_container["device_name"].r[0x000].get_post_write_rules().method(...);`

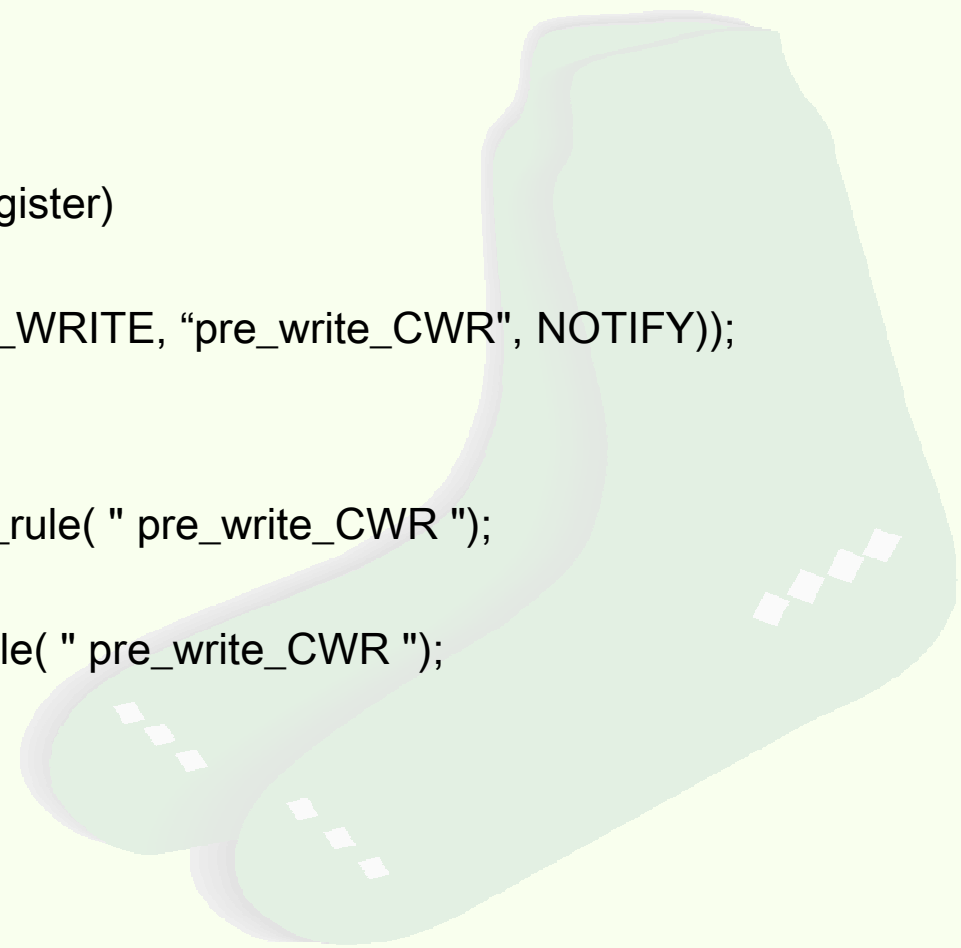
- Internal Access

- Declaring a pre-existing rule

```
void my_module::end_of_elaboration()
{
    // Pre-Write to the CWR (Control Word Register)
    GR_FUNCTION( pre_write_cwr );
    GR_SENSITIVE(r[0x000].add_rule( POST_WRITE, "pre_write_CWR", NOTIFY));
}
```

- User Code

- `r[0x000].get_post_write_rules().deactivate_rule(" pre_write_CWR ");`
 - ...
 - `r[0x000].get_post_write_rules().activate_rule(" pre_write_CWR ");`



Using Notification Rules with bit_range

- Global or External Access

```
gs::reg::g_gr_device_container["device_name"].r[0x000].br["test"].get_post_write_rules().method(...);
```

- Internal Access

- Declaring a pre-existing rule

```
void my_module::end_of_elaboration()  
{  
    // Pre-Write to the CWR (Control Word Register)  
    GR_FUNCTION( isr_bits );  
    GR_SENSITIVE(r[0x000].br["test"].add_rule( POST_WRITE, "isr_bits_CWR", BIT_STATE, 2, true));  
}
```

- User Code

- r[0x000].br["test"].get_post_write_rules().deactivate_rule("post_write_isr_bits_CWR ");
 - ...
 - r[0x000].br["test"].get_post_write_rules().activate_rule("post_write_isr_bits_CWR ");

Delayed Notification Rules

- Internal Access

- Declaring a delayed rule

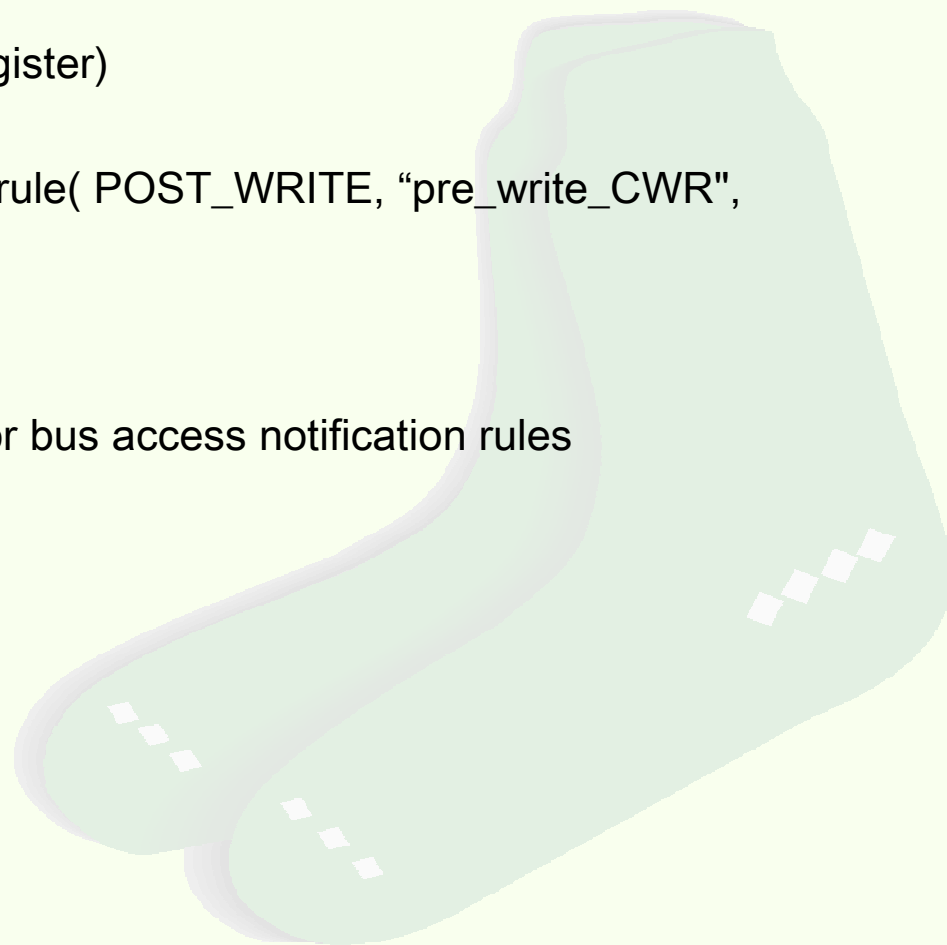
```
void my_module::end_of_elaboration()  
{  
    // Pre-Write to the CWR (Control Word Register)  
    GR_FUNCTION( pre_write_cwr );  
    GR_DELAYED_SENSITIVE(r[0x000].add_rule( POST_WRITE, "pre_write_CWR",  
    NOTIFY), sc_time(10, SC_NS));  
}
```

- Remark:

- Delayed only for bus accesses, only use for bus access notification rules

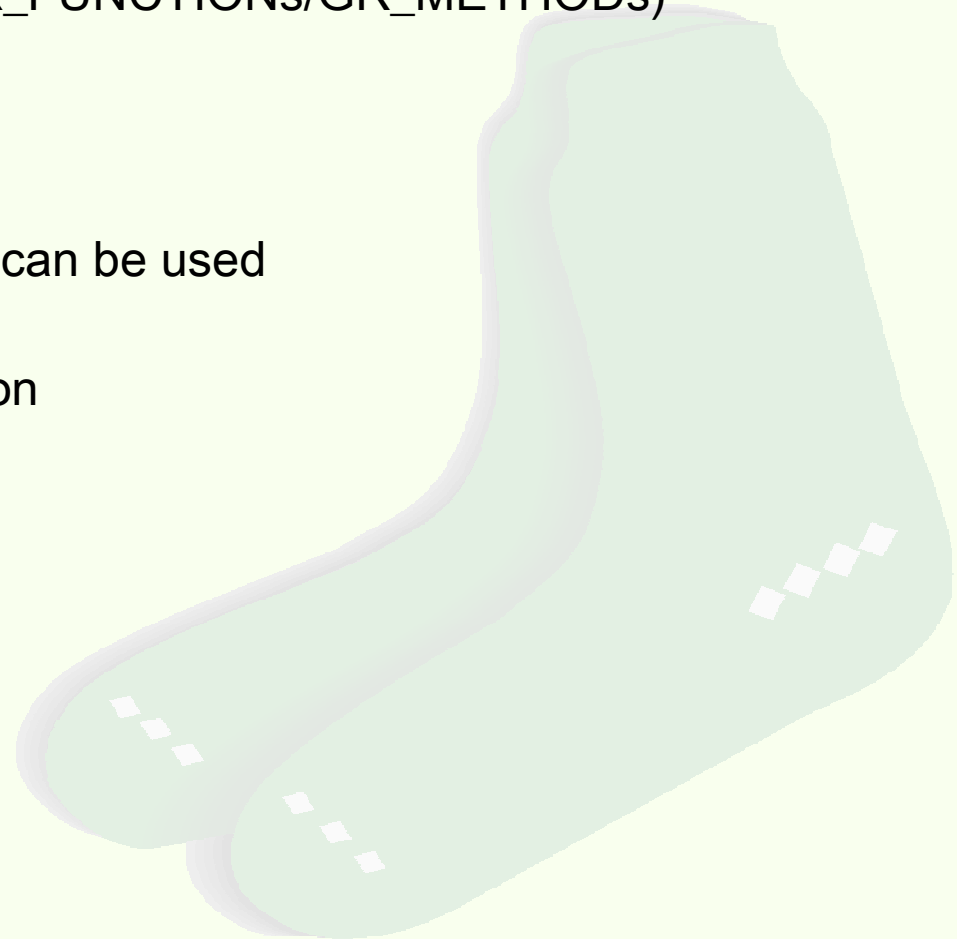
- Switch (enable/disable) delay

- Can be switched off per socket
- m_slave_socket.disable_delay();
m_slave_socket.enable_delay();



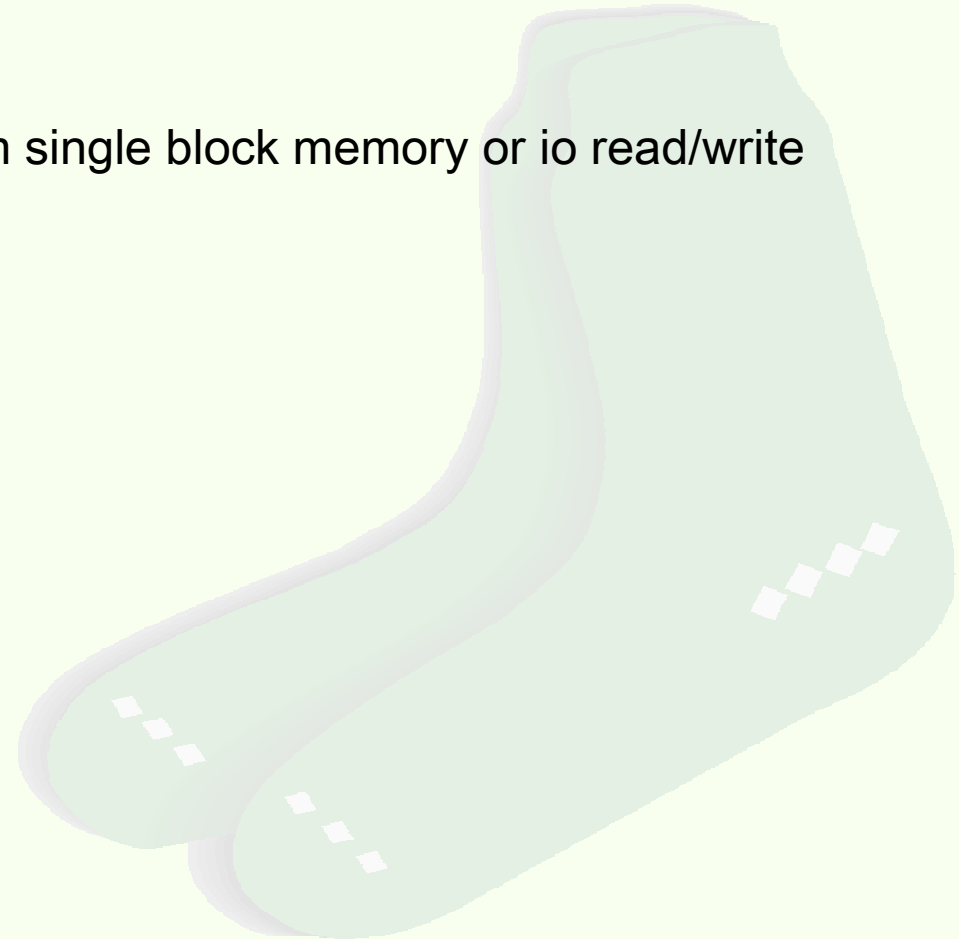
Event Switch

- Register-wide switch enables event notification of gr_event (fired by notification rules)
 - Disabled by default (this means notification rules by default never fire their event but only make callbacks to GR_FUNCTIONS/GR_METHODs)
 - Enabled register-wide
 - `r[0x01].enable_events();`
 - `r[0x01].disable_events();`
 - With enabled switch SC_METHODs can be used (if needed for some reason)
 - Enabled switch slows down simulation



greenreg_socket<master> Package

- Scope
 - Simplify the action of creating/completing read and write transaction across a bus
 - Enable TLM Independence
- Use Cases
 - Any module using the bus to perform single block memory or io read/write transaction



drf_port<master > Classes

- GreenReg User Interface
 - greenreg_socket< *::master >
 - Master port simplifies bus read(s) and write(s)

```
DRF::drf_port< AZTALAN::master >
```



User Interface (greenreg_socket <master>)

- Instantiation
 - `greenreg_socket < master> my_master_port;`
- Constructor
 - `my_master_port("my_port")`
 - Params: name (sc_module_name)
- Methods of Interest
 - `->read(0x4f003, 1);` - returns data from offset for the specified size in bytes
 - Pointer reference is using an operator overload, even though the port was not declared as a pointer
 - Params: Absolute address, byte width (must align properly with target register expectations)
 - `->write(0x4f003, 0xc2, 1);`
 - Pointer reference is using an operator overload, even though the port was not declared as a pointer
 - Params: Absolute address, byte width (must align properly with target register expectations)
 - `disable_delay()` (only slave socket)
 - Disables usage of delay for delayed notification rules (see GR_DELAYED_SENSITIVE)
 - `enable_delay()` (only slave socket)
 - Enables usage of delay for delayed notification rules (see GR_DELAYED_SENSITIVE)
 - `bool delay_enabled()`
 - If the delay for delayed notification rules is enabled

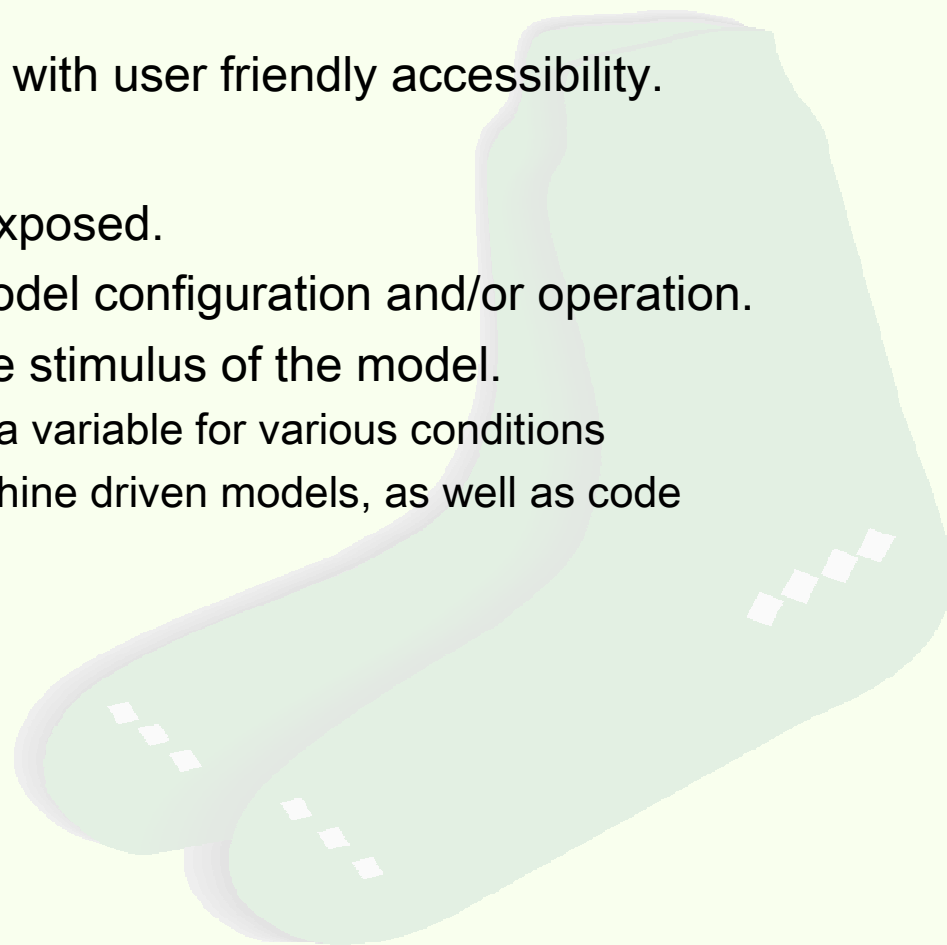
GreenReg API

(Attribute & Switch)



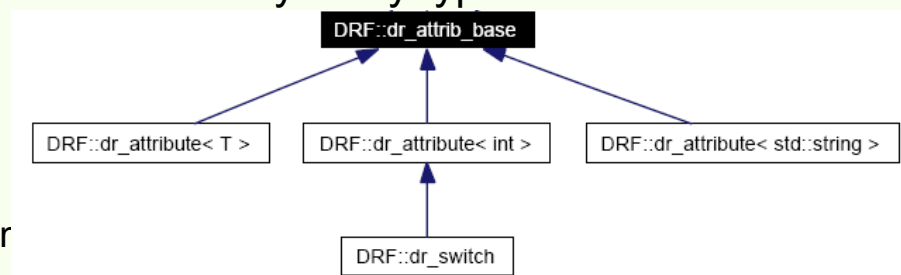
Attribute & Switch Package

- Scope
 - Enable data elements that are easily accessible and configurable.
 - Facilitate GreenReg modeling methodology by enabling stimulus capabilities in data elements.
 - Build upon and enhance `sc_attribute` with user friendly accessibility.
- Use Cases
 - Any data element that needs to be exposed.
 - Exposed switches that will modify model configuration and/or operation.
 - Any variable that will be used to drive stimulus of the model.
 - As apposed to code that would pole a variable for various conditions
 - This is a key step towards state machine driven models, as well as code automation



Attribute & Switch Classes


- GreenReg Internal
 - gr_attrib_base – many C++ operators to do some relatively tricky type resolution for the descending templated class.
- GreenReg User Interface
 - gr_attribute< T >
 - T is int, unsigned int, string or user specific in
 - Default event fires for every modification (disabled by default)
 - Dumps to configuration file
 - gr_attribute< unsigned int>
 - SPECIAL: Only implementation that has Pre-Read & Post-Write notification rules
 - gr_switch
 - A specialized implementation of gr_attribute<int>
 - A switch can have 2 or more pre-defined states
 - Each state has an associated event that is fired when the state becomes active
 - Dumps to configuration file (pre-defined switches will also dump possible options)



Attribute & Switch Container Classes

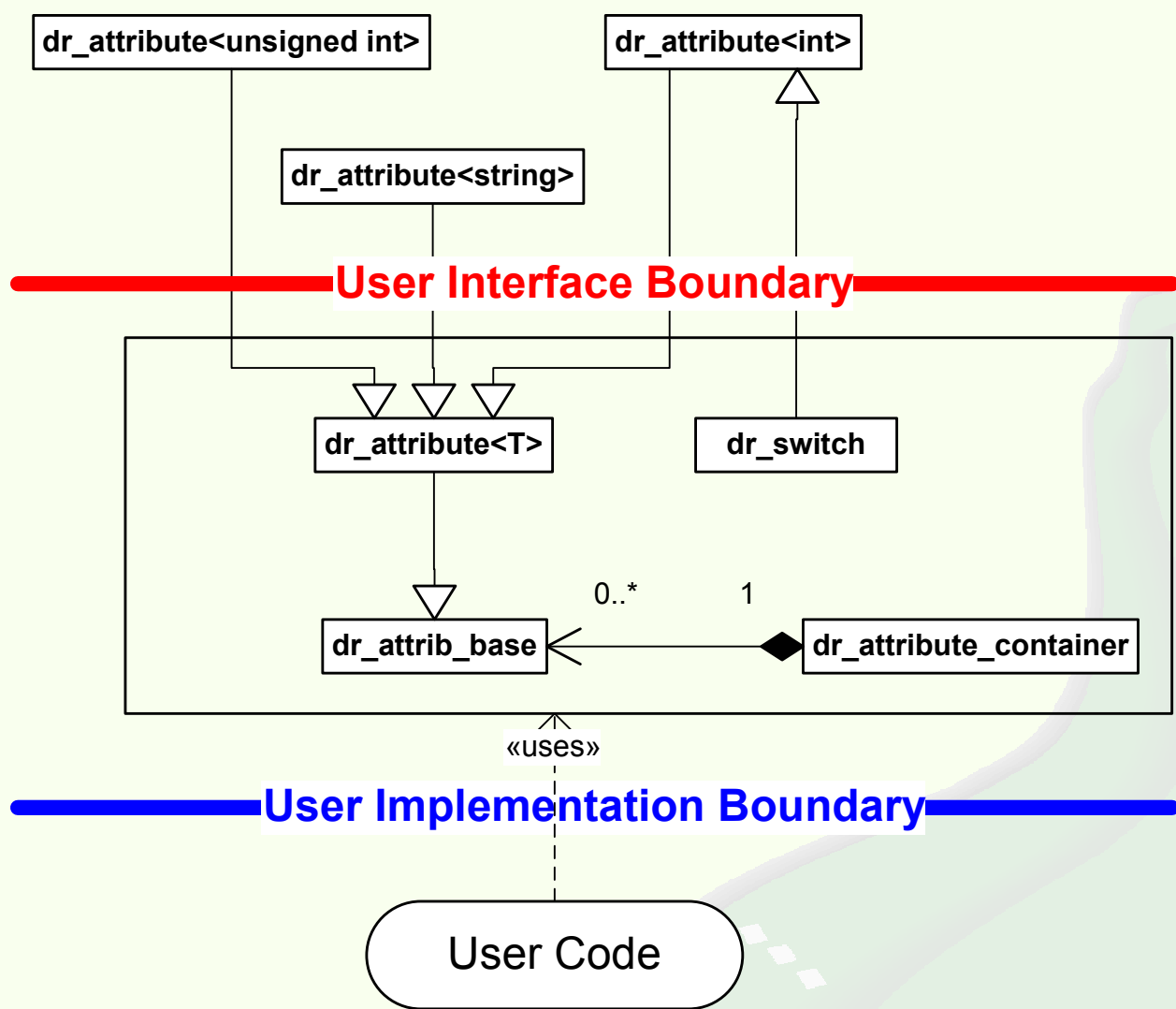
- GreenReg User Interface

- `gr_attribute_container`
 - storage container for `gr_attrib_base` objects (i.e `gr_attribute<T>` & `gr_switch`)
 - `gr_device` & `gr_subdevice` have `gr_attribute_container` declared as 'a'
 - inherits from `gs::reg_utils::access_data_container` for accessibility
- `gr_switch_container` (DEPRICATED)



DRF::dr_attribute_container

Attribute & Switch Inheritance & Containment



User Interface (gr_attribute)

- Instantiation
 - `gr_attribute<int> my_int;`
 - `gr_attribute<unsigned int> my_uint;`
 - `gr_attribute<std::string> my_string;`
 - NOTE: other template types will require additional template specialization implementations.
- Constructor
 - `my_uint("counter", "variable holding counter value", a, 0x0);`
 - Params: name, description, attribute container, initial value
- Methods of Interest
 - Notification Rule Methods
 - `add_rule(ATTRIB_PRE_READ, "on_read_counter", NOTIFY, ...)`
 - Creates a rule internally based on parameters and returns an event associated with the rule.
 - Params: `gr_reg_rule_container` enum, unique name, `gr_reg_rule_type`, rule parameters.
 - SHOULD BE DEFINED IN **"end_of_elaboration"**
 - `get_pre_read_rules()` – returns the pre-read rule container.
 - `get_post_write_rules()` – returns the post-write rule container.
 - Default Write Event Methods
 - `default_event()` – returns the default write event (the event will not fire unless explicitly enabled, i.e. default disabled)
 - `enable_event()` – enables the default write event if it were disabled
 - `disable_event()` – disables the default write event if it were enabled
 - `has_event()` – returns true if the default write event is enabled, false otherwise
 - Access Methods
 - `dr_attrib_base & operator = (_value)` – write data operator (executes post_write notification rules if T=unsigned int)
 - `operator T ()` – read data operator (must be accessed via explicit <T> definition)
(executes pre_read notification rules if T=unsigned int)
 - `set_value(_value)` – writes value (executes post_write notification rules if T=unsigned int)
 - `get_value()` – returns value (executes pre_read notification rules if T=unsigned int)

User Interface (gr_switch)

- Instantiation
 - `gr_switch my_toggle_switch;`
- Constructor
 - `my_toggle_switch("bypass_algorithm", "bypasses decode algorithm to run faster when on", a, gr_options_bool_toggle, gs::reg::OFF)`
 - NOTE: used for pre-defined switch types
 - Params: name, description, dr_attribute_container, pre-defined switch, initial switch position.
 - `my_toggle_switch("bypass_algorithm", "description", a, 0)`
 - NOTE: used for user-definable switches
- Methods of Interest
 - `set_value(int id)` – set the switch to the specified id, an event will be fired for this id (even if already set)
 - `operator = (int id)` – same as `set_value`
 - `get_switch_event(int id)` – returns the `sc_event` associated with the id
- Methods applicable only to the user-definable switch use case:
 - `add_entry("confused", 1)` – creates a switch placement for the specified key at the specified id
 - `get_switch_event("confused")` – returns the switch event from the string key
- Room for Improvement:
 - switches do not have the ability to obtain their active state

Using the Attribute & Switch Package

- Global or External Access
 - `gs::reg::g_gr_device_container["device_name"].a["my_uint"] = 32;`
 - `unsigned int uint_a =`
`gs::reg::g_gr_device_container["device_name"].a["my_uint"].attrib<unsigned int>();`
- Internal Access
 - `a["my_uint"] = 32;`
 - `unsigned int uint_a = a["my_uint"].attrib<unsigned int>();`
 - `a["my_switch"].method(...)`

