# Improving the UVM Register Model: Adding Product Feature based API for Easier Test Programming

Krishnan Balakrishnan, Courtney Fricano, and Kaushal Modi
Analog Devices, Inc. 3 Technology Way Norwood, MA 02062

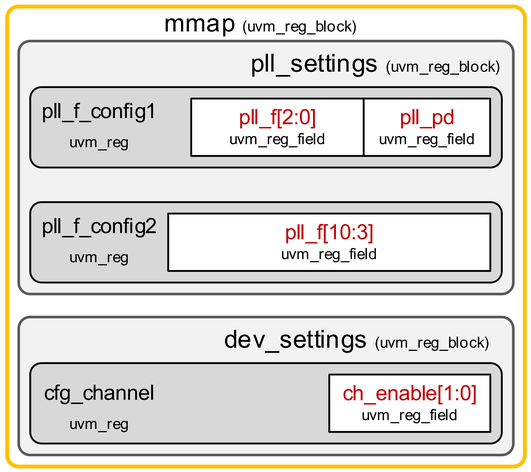
ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

## Problem Statement

- Users prefer an API that clearly highlight the feature of the device being programmed.
  - **UVM**: Design verification
  - **GUI**: lab evaluation
  - **Software**: firmware and customer
- Specifying product features as fields in registers hides product features under layers of register abstractions
  - Features that span multiple registers get split into multiple fields with no mapping from field(s) back to product features.
  - The IP-XACT specification, and the UVM environment present a hardware centric view of a device

## Example of register based environment

- Device contains three product features - pll_f[10:0], pll_pd, and ch_enable[1:0] in three 8-bit registers.




- Programming the device in UVM environment

```
// pll_f = 11'b10101010101          pll_pd = 1'b0      ch_enable = 2'b11
uvm_status_e status;

// Set pll_pd
mmap.pll_settings.pll_f_config1.pll_pd.set          (1'b0);

// Set pll_f - The value of pll_f has to be split manually by user into the two uvm_reg_field set methods.
mmap.pll_settings.pll_f_config1.pll_f.set           (3'b101);
mmap.pll_settings.pll_f_config2.pll_f.set           (8'b10101010);

// Set ch_enable
mmap.dev_settings.cfg_channel.ch_enable.set         (2'b11);

// Update mmap. Not ideal as it iterates through all registers (and writes all registers with volatile fields)
mmap.update(status);
```

## Programming Impediments

- The user needs to have a detailed knowledge of the implementation details of the product features.
  - **Complicated coding style**: User needs to traverse the register layer hierarchy of the device to access a feature.
  - **Manual overhead**: Configuration / Read-back of product features that span multiple registers is a manual process (splitting feature into multiple fields / merging multiple fields into a feature) that is very prone to user errors.
  - **Reduced Portability**: Code is not easily portable as it is susceptible to changes in field width, position, or moving the field to a different register.

## Solution – Product feature based API

- API is independent of the implementation of product features as field(s) within register(s)
  - Obviates user errors and improves readability and portability.
  - The API coexists with the existing UVM model.
  - Product features are implemented as BitFields ('adi_bitfield' objects).
  - All BitFields are held in a container class ('adi_bitfield_block')
    - The object for this class is created at the root of the UVM memory map for easy access to product features

## 'adi_bitfield' class

- The Class implements a product feature (BitField)
  - Can have arbitrary length, not restricted by register widths
  - Contains methods similar to uvm_reg_field
    - Set, get, update, write, read
  - Contains an array of 'slices' that maps portions of the product feature to uvm_reg_field(s) across uvm_reg object(s).
    - Each member of the 'slice' array is a handle to a uvm_reg_field, a MSB position, and a LSB position that define the mapping

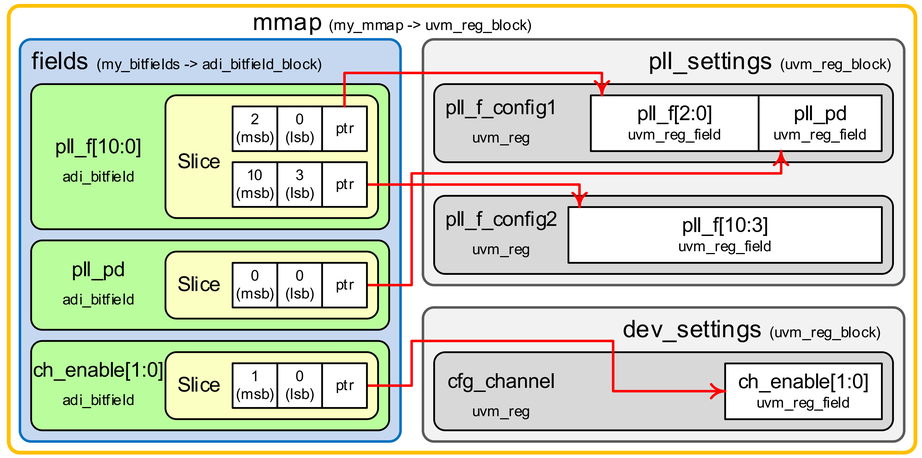
```
class adi_bitfield extends uvm_object;

  typedef struct {              // Structure containing pointers to the position of the bitfield slice
    uvm_reg_field ptr;          // Handle to the sub-bitfield slice
    int lsb;                    // LSB of the slice in the bitfield
    int msb;                    // MSB of the slice in the bitfield
  } slice_t;

  // Members
  slice_t slices[int];          // Collection of pointers to support bitfields spanning multiple registers
  adi_bitfield_block m_parent; // Pointer to parent fields block

  // Methods
  new(..);
  configure_slice (..);         // Configure slice function sets the ptr, MSB, and LSB fields for a single slice
  set(..);                      // Set the desired value for this field by iterating through the slices
  uvm_reg_data_t get (..);      // Get the desired value for this field by iterating through the slices
  bit needs_update(..);         // Check if the abstract model contains different desired and mirrored values.
  update (..);                  // Update the content of the field in the design to match the desired value
  write (..);                   // Write the specified value in this field
  read (..);                    // Read the specified value in this field
endclass
```

## 'adi_bitfield_block' class

- The class is a container for all adi_bitfield objects
  - Instance of this class is created at the root of the UVM memory map to be easily accessible to the user API.
- This class contains a non-iterative implementation of the 'update' method.
  - The method updates only the adi_bitfield objects that have been modified using the adi_bitfield's 'set' method.
  - When an adi_bitfield's set method is invoked it adds itself to the 'modified_fields' array. The 'update' method in this class only iterates through this array, and not through all BitFields.

```
class adi_bitfield_block extends uvm_object;

  // Members
  adi_bitfield modified_fields[string];   // Array of fields modified through the 'set' method

  // Methods
  new(..);
  set_modified_field (..);    // adi_bitfield's 'set' calls this to add itself to the modified_fields array
  update (..);      // Iterates through modified_fields, and calls 'update' only for adi_bitfield objects in array
endclass
```

## BitField based environment




## Configuring Bitfield based API

```
// extend adi_bitfield_block class to add the BitFields and a function to configure them
class my_bitfields extends adi_bitfield_block;

  // BitFields
  adi_bitfield ch_enable;
  adi_bitfield pll_f;
  adi_bitfield pll_pd;

  // The configure task sets up the mapping between BitField and the uvm_reg_fields
  function void configure (my_mmap blk_parent);
    // map ch_enable
    ch_enable = adi_bitfield::type_id::create("ch_enable");
    ch_enable.configure_slice(this, blk_parent.dev_settings.cfg_channel.ch_enable);

    // map pll_f      pll_f = adi_bitfield::type_id::create("pll_f");
    pll_f.configure_slice(this, blk_parent.pll_settings.pll_f_config2.pll_f, 10, 3);
    pll_f.configure_slice(this, blk_parent.pll_settings.pll_f_config1.pll_f, 2, 0);

    // map pll_pd     pll_pd = adi_bitfield::type_id::create("pll_pd");
    pll_pd.configure_slice(this, blk_parent.pll_settings.pll_f_config1.pll_pd);
  endfunction
endclass

// Class for the memorymap
class my_mmap extends uvm_reg_block;
  // The RegsiterMaps
  rand my_dev_settings dev_settings;
  rand my_pll_settings  pll_settings;

  // The Bitfields environment
  rand my_bitfields fields;

  virtual function void build();
    // Build the RegisterMaps as in the existing UVM environment
    ...
    // Build the BitFields environment
    this.fields = my_bitfields::type_id::create("fields");
    this.fields.configure(this);
  endfunction
endclass
```

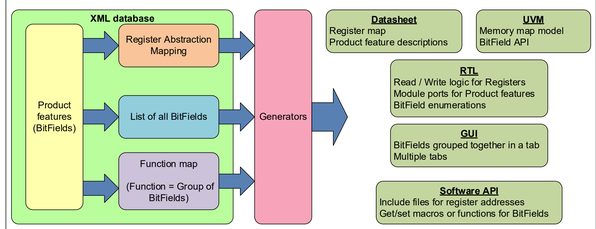
## Programming using the BitField API

- API accesses product features directly
  - Code is independent of implementation.
  - No manual overhead splitting/merging features to fields.
  - Portable and understandable code

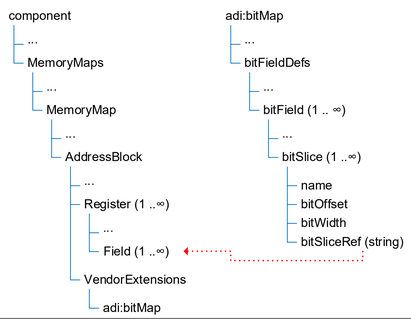
```
// pll_f = 11'b10101010101          pll_pd = 1'b0      ch_enable = 2'b11
uvm_status_e status;

// Set pll_pd
mmap.fields.pll_pd.set          (1'b0);

// Set pll_f.
mmap.fields.pll_f.set           (11'b10101010101);

// Set ch_enable:
mmap.fields.ch_enable.set       (2'b11);

// Update all modified features.
mmap.fields.update(status);
```

## Automating BitField Model Creation

- ADI uses an internally developed database tool with scripts to generate the required BitField to uvm_reg_field mappings
  - BitField is the principal element
  - Can group BitFields to create mappings other than the register abstraction, that improve Time-To-Market for ADI's products and provide great value addition to the customers




- Vendor extensions in IP-XACT standard can also be used to define the Bitfield to register mapping.




## Conclusion

- BitField based API has several advantages over Register based UVM environment
  - Integrated and co-existing with existing UVM environment
  - Fully automated process to generate BitField environment
  - Test patterns using the BitField API are easy to code, better documented, and immune to changes in the BitField address, width or position during the course of the project.