Serial Prog Class Tutorial for Matlab

Michael H. Perrott (www.cppsim.com)

Table of Contents

Setup

The Serial Prog Class has been set up with a mex file interface to Matlab, and must be compiled within Matlab in conjunction with an outside C++ compiler such as Microsoft Visual C++. Be sure that you have this C++ compiler installed on your system before doing the operations below.

While purchasing the Microsoft Visual C++ compiler is the preferred approach since it allows the easiest distribution of the mex function associated with the Serial Prog Class, you can also obtain a free express version of the Microsoft C++ compiler as described at:
http://www.mathworks.com/matlabcentral/fileexchange/22689

The key files involved in creating the mex function associated with the Serial Prog Class are:

- compile_serial_state_mex.m
  - This is a simple Matlab script that includes the mex compile instructions
- serial_state_mex.cpp
  - This is the mex wrapper code that interfaces Matlab to the underlying C++ Serial Prog Class code.
- serial_prog_class_mex.cpp
  - This is the C++ Serial Prog Class code, which has been slightly altered to conform to Matlab mex code standards. Specifically, all **printf** commands have been changed to **mexPrintf** commands, and all **exit** commands have been changed to **mexErrMsgTxt** commands.
- serial_prog_class_mex.h
  - This is the header file associated with the C++ Serial Prog Class code. No changes were required to utilize this within this mex application
- cppsim_classes_mex.cpp
  - This provides supporting C++ classes from the CppSim package (available at http://www.cppsim.com) which provide Lists and expression evaluation used in the C++ Serial Prog Class code.
- cppsim_classes_mex.h
  - This is the header file associated with the C++ classes from the CppSim package.

To perform the compile operation, do the following steps within a Matlab command window:

- >> cd MatlabSerialProgClass
  - Change the directory such that you are in the same location as this document.
- >> ls
  - Verify that the file 'compile_serial_state_mex.m' is in the current directory
- >> mex –setup
  - Choose the Microsoft C++ compiler
- >> compile_serial_state_mex
  - This should compile the C++ code into a mex function called 'serial_state'

Introduction

The Serial Prog Class has been created as part of the CppSim/VppSim ecosystem to serve the following purposes for ICs that contain a serial programming pin:

- Allow a more seamless transfer of device programming knowledge from CMOS design to characterization to test by utilizing a C++ class to contain such knowledge. The benefit of using C++ code is that:
  - It can be incorporated in CMOS design using CppSim and VppSim for functional verification, which allows the programming information encoded in the class to be verified and debugged by the actual CMOS designers
  - It can be incorporated in characterization by embedding it within a Matlab mex file, as is the focus of this document
  - It can be incorporated by directly in test instruments as a C++ routine.
- Provide a layer of abstraction above the NVM and Register bits used in a chip and instead provide a set of Definitions and Tokens that are more human readable. Aside from improving readability of code, the extra layer of abstraction allows NVM and Register maps to be reconfigured if necessary without impacting higher levels of code. This includes register re-assignments as well as XOR operations on individual bits for the purposes of achieving appropriate default conditions.

In the provided code, the NVM and Register map is organized within an 8-bit address space that is composed of 16-bit registers such that:

- The first 32 registers correspond to operational modes that are mainly used for testing and burning of the NVM. However, a few registers are used for selected functionality modes, and will therefore be exposed to the end user if such personalities are programmed into the part. We will refer to this set of registers as the “reg” section.
- The next set of 32 registers correspond to “shadow NVM” which are assumed to directly control various components of the IC. To simplify notation, we will refer to this set of registers as the “nvm” section.

Listing Key Modes and Resetting the Class State in Matlab

This section provides details of listing key modes and resetting the Serial Prog Class when encapsulated as a Matlab mex function. All commands below are assumed to take place within a Matlab command window.

A. Listing Available Options

```
>> serial_state
```

Generates an error message which provides a quick overview of various command options.

B. Clear out the state of the Serial class

```
>> serial_state('end')
```

Resets the state of the serial class such that all nvm and reg bits are assumed to be zero.

Read Modes in Matlab

This section provides details of the various Matlab read modes of the Serial Prog Class when encapsulated as a Matlab mex function. All commands below are assumed to take place within a Matlab command window.

A. Listing All Definitions and their current Token values

```
>> out = serial_state('nvm','get')
```

Displays a list of all definitions and their current token values within the "nvm" register space.

```
>> out = serial_state('reg','get')
```

Displays a list of all definitions and their current token values within the "reg" register space.

B. Displaying the Current Token value for a specific Definition

```
>> out = serial_state('definition','get')
```

Displays the current token value for a specific definition.

- Examples:
  ```
  >> out = serial_state('reg_cap','get')
  ```
  Displays the current token value for this definition

  ```
  >> out = serial_state('gain1','get')
  ```
  Displays the current value for this definition as defined by formula and inverse formula definitions in the C++ code for the Serial Prog Class.

C. Listing all available Tokens for a specific Definition

```
>> out = serial_state('definition','get_tokens')
```

Provides a list of all possible tokens for the given definition. In the case where a formula is used, the returned list includes 'formula' as the first entry followed by the min and max input values that are supported.

- Examples:
  ```
  >> out = serial_state('reg_cap','get_tokens')
  ```
  Displays the set of 8 possible token values which are supported for this definition

```
>> out = serial_state('gain1','get_tokens')
```
Indicates 'formula' as its first entry and then lists the min and max values that are supported.

D. Displaying the Current Bit Pattern for specific reg or nvm register bits

- Examples:

  ```
  >> out = serial_state('nvm4[14:12]','get')
  ```
  Displays the binary values corresponding to this set of bits in the "nvm" section

  ```
  >> out = serial_state('nvm11[0]','get')
  ```
  Displays the binary value corresponding to this bit in the "nvm" section

  ```
  >> out = serial_state('reg0[15:0]','get')
  ```
  Displays the binary values corresponding to this set of bits in the "reg" section

  ```
  >> out = serial_state('reg12[15]','get')
  ```
  Displays the binary value corresponding to this bit in the "reg" section

E. Obtaining a matrix of the entire reg or nvm sections

In cases where outside routines need access to the bit patterns of the "reg" or "nvm" (i.e., Shadow NVM) sections, a matrix output of size 32 rows by 16 columns with binary values corresponding to individual bit values can be obtained:

```
>> out = serial_state('nvm','get_bin');
```
For the above output, out(j,k) = nvm(j-1)[k-1], where 1 <= j <= 32 and 1 <= k <= 16.

```
>> out = serial_state('reg','get_bin');
```
For the above output, out(j,k) = reg(j-1)[k-1], where 1 <= j <= 32 and 1 <= k <= 16.

- Examples:

  ```
  >> out = serial_state('nvm','get_bin')
  >> out(5,4)
  ```
  Displays the binary value of nvm4[3]

  ```
  >> out(1,:)
  ```
  Displays the binary values of nvm0[15:0]

  ```
  >> out = serial_state('reg','get_bin')
  >> out(7,1:3)
  ```
  Displays the binary values of reg4[3:1]

F. Obtaining a text listing of the entire reg or nvm sections

In cases where a quick check is desired of the nvm or reg state, one can get a text listing of the bit patterns of each of these sections:

```
>> out = serial_state('nvm','get_bin_verbose');
```
Displays text corresponding to the bit patterns of the entire "nvm" (i.e., Shadow NVM) section.

```
>> out = serial_state('reg','get_bin_verbose');
```
Displays text corresponding to the bit patterns of the entire "reg" section.

<u>Write Modes in Matlab</u>

This section provides details of the various Matlab write modes of the Serial Prog Class when encapsulated as a Matlab mex function. All commands below are assumed to take place within a Matlab command window.

A key issue to bear in mind when writing to the Serial Class state is that the contents are only updated after the "prog-send" command is issued:
>> out = serial_state('prog','send');

If multiple write commands are issued before the "prog-send" command above, then they are compacted together in order to minimize the number of bits that must be sent to the prog pin of the physical Serial chip. Due to this compacting operation, there is no guarantee that the order of commands will be preserved for multiple commands issues before a "prog-send" command. As such, if the order of commands matter, one should issue multiple "prog-send" commands such that compacting will not impact the relevant command order..

A. Setting the Token value for a given Definition

The preferred operating mode when using the Serial Prog Class is to only deal with definition and tokens rather than registers and bits:
>> serial_state('definition','token')
Sets the given definition to the specified token value. Recall from the previous section that you can see what tokens are supported by typing:
>> out = serial_state('definition','get_tokens')

- Examples:
  >> serial_state('reg_cap','3.0')
  Sets definition 'reg_cap' to token value '3.0'. Note that in this case, 8 possible tokens were specified and the token value must match one of these values. The possible token values are displayed by typing
  >> out = serial_state('reg_cap','get_tokens')

  >> serial_state('gain1','11.419')
  Sets definition 'gain1' to token value '11.419'. Note that in this case, a formula was specified for this definition so that any real number in the range of 0 to 16 can be entered. The possible range of values are displayed by typing
  >> out = serial_state('gain1','get_tokens')

B. Setting the bit pattern for a given reg or nvm register

While the preferred means of changing values is through the use of definitions and tokens, there are times when more direct bit level operations are required.

- Examples:
  >> serial_state('nvm5[9:1]','010011111')
  Sets the binary bit pattern for nvm5[3]

  >> serial_state('nvm10[14]','0')
  Sets the binary value for nvm10[14]

>> serial_state('reg2[12:8]','10100')
Sets the binary bit pattern for reg2[12:8]

>> serial_state('reg15[0]','1')
Sets the binary value for reg15[0]

C. Setting bit values for the entire reg or nvm sections

In cases where outside routines need to explicitly set the bit patterns of the entire "reg" or "nvm" (i.e., Shadow NVM) sections, a matrix of size 32 rows by 16 columns with binary values corresponding to individual bit values can be input into the Serial Prog Class. Note that the "prog-send' command must be applied to update the "reg" or "nvm" contents according to the input matrix.
>> input_matrix = zeros(32,16);
The above input matrix corresponds to the 32 16-bit registers of either the "reg" or "nvm" sections

>> serial_state('nvm',input_matrix);
Sets the entire "nvm" section according to the binary values contained in "input_matrix". Note that nvm(j-1)[k-1] = input_matrix(j,k), where 1 <= j <= 32 and 1 <= k <= 16.

>> out = serial_state('nvm',input_matrix);
Same as the previous command, but with the addition that the prior "nvm" contents are sent to the out matrix. Recall that a "prog-send" command must be sent to update the "nvm"contents.

>> serial_state('reg',input_matrix);
Sets the entire "reg" section according to the binary values contained in "input_matrix". Note that nvm(j-1)[k-1] = input_matrix(j,k), where 1 <= j <= 32 and 1 <= k <= 16.

>> out = serial_state('reg',input_matrix);
Same as the previous command, but with the addition that the prior "reg" contents are sent to the out matrix. Recall that a "prog-send" command must be sent to update the "reg"contents.

Physically Writing to the Serial Chip from Matlab

The actual IC is assumed to have a serial interface pin that needs a particular sequence input into it in order to update its internal state. In the provided code, the protocol for such updates is assumed to consist of a 16-bit header sequence '0C0C', 8-bit address sequence corresponding to the "reg" or "nvm" register address to be updated, and 16-bit data sequence corresponding to the "reg" or "nvm" register value to be updated.
The 'prog-send' command will generate this programming sequence in concatenated manner (i.e., header,address,data,header,address,data,…) to update the "reg" and/or "nvm" registers of the IC according to the previous write commands that were applied to the Serial Programming Class since the last 'prog-send' command. As mentioned earlier, it is important to note that the previous write commands that occurred since the last 'prog-send' command will be compacted such that the order of executing each of those commands will not be preserved.

Also, while the 16-bit header is nominally '0C0C', the third nibble can be altered from its default value of '0' to any value between '0' and 'F' by setting the NVM bits using the command:
>> serial_state('chip_address','x'); // where x is a decimal value between 0 and 15

In order to change the header coming from the Serial prog class, you must also run
>> serial_state('header_address_for_programmer','x'); // where x is a decimal value 0 to 15

It is expected that some other Matlab function will then take the output of the 'prog-send' sequence and turn into the physical signals which are sent to the prog pin of the physical Serial chip.

There are two primary ways of executing the 'prog-send' command:

>> out = serial_state('prog','send');
Creates a binary sequence that consists of concatenated programming commands (i.e., header, address, data, header, address, data, …). As such, a valid output will have length of N*40, where N is an integer value greater than 0. If no write commands occurred since the last 'prog-send' command, then the output will have length 1 to indicate that no prog commands need be sent to the physical Serial chip.

>> out = serial_state('prog','send',previous_sequence);
Also creates a binary sequence that consists of concatenated programming commands, but also concatenates the binary sequence from input 'previous_sequence' to the output vector 'out'. This may prove useful where you want to assemble Matlab functions that need to execute multiple 'prog-send' commands in order to preserve order between a set of write commands.

Adding New Definitions and Tokens to the Serial Prog Class

The Serial Prog Class allows easy addition and modification of definitions and their respective tokens. However, any such updates must be done through modification of the C++ code in 'serial_prog_class_mex.cpp' and then 'compile_serial_state_mex' must be run in Matlab to generate an update version of the mex function 'serial_state'.

There are 5 key methods that are involved in adding or modifying definitions and tokens in the 'serial_prog_class_mex.cpp' file: **add_definition()**, **add_tokens()**, **add_formula()**, **add_inv_formula()**, and **add_xor()**. Examination of the 'serial_prog_class_mex.cpp' file provides several examples of using these methods.

Conclusion

This document provides information on the mex version of the Serial Prog Class as implemented by the 'serial_state' function in Matlab. This approach to doing serial pin programming of ICs will hopefully provide meaningful benefits in efficiency. The first is to allow a more seamless transfer of device programming knowledge from CMOS design to characterization to test by utilizing a C++ class to contain such knowledge. The second is to provide a layer of abstraction above the actual Shadow NVM and Register bits used in the chip and instead provide a set of Definitions and Tokens that are more human readable. It is hoped that this C++ Class based approach will be generally useful as part of the CppSim/VppSim ecosystem.