MULTI-QUEUE FLOW-CONTROL DEVICE SERIAL PROGRAMMING

APPLICATION NOTE AN-303

By Stewart Speed & Jack Deans

INTRODUCTION

The multi-queue flow-control device family from IDT are user programmable devices, the user can program a given device to have anywhere from 1 to the maximum number of queues available. The user can also program each queue to be any depth, assigning memory blocks from the total pool of memory within the device to the queues independent of each other. That is, queue sizes are flexible and can be set to any depth on a per queue basis. The size of a queue must comprise of memory blocks, therefore a queue can be configured down to a resolution of 1024 x 9, 512 x 18 or 256 x 36.

The user can also set Almost Full and Almost Empty flags to any location within the depth of a queue, again on a per queue basis, so that the Almost Full and Almost Empty flags within individual queues of a device are at different locations.

Please note, that queue widths per device are not independent, that is, all queues within a given device will have the same data input bus width and the same data output bus width.

As one can imagine, this requires a large amount of configuration registers within the device to store and hold the set-up information of a Multi-Queue device. The registers can be configured independently by the multi-queue flow-control device using the default programming mode. Here a sequence of WCLK cycles must be provided to the device after a master reset, these WCLK cycles loading the registers internally. The default programming will configure a multi-queue flow-control device in a predetermined manner that is always the same, the maximum number of queues will be set-up, all queues will be equal depths. The total memory of the device will be shared equally between all queues. The Almost Empty and Almost Full flag offsets of all queues will set to either 8 or 128 depending on the DF input at master reset.

The set-up registers of the Multi-Queue devices are also user programmable via the serial port. As opposed to default configuration of the device, serial programming can be performed by the user allowing the configuration of the number of queues, between 1 and the maximum, the depth of queue's, a minimum of 1024 x 9; 512 x 18; 256 x 36, the flag offsets at any point within a respective queue depth. One may also modify the serial bitstream file manually instead of using the serial bitstream generator for minor changes. A section describing the bitstream register values are defined in this document.

THE SERIAL PORT OF THE MULTI-QUEUE FLOW-CONTROL DEVICE

The serial port comprises the following inputs/outputs:

1. Serial Clock, SCLK.
2. Serial Input Enable, $\overline{SENI}$
3. Serial Data Input, SI
4. Serial Output Enable, $\overline{SENO}$
5. Serial Data Output, SO

The user serially loads data onto the SI input, this data is clocked into the device on the rising edge of SCLK provided that $\overline{SENI}$ is active, LOW. The $\overline{SENO}$ output of a device is HIGH until programming of the device is fully complete at which point the $\overline{SENO}$ output follows the $\overline{SENI}$ input, therefore with $\overline{SENI}$ being LOW the $\overline{SENO}$ output should go LOW. With $\overline{SENO}$ LOW, if the user continues to clock data into a device serially on SI, this data will be passed through to the

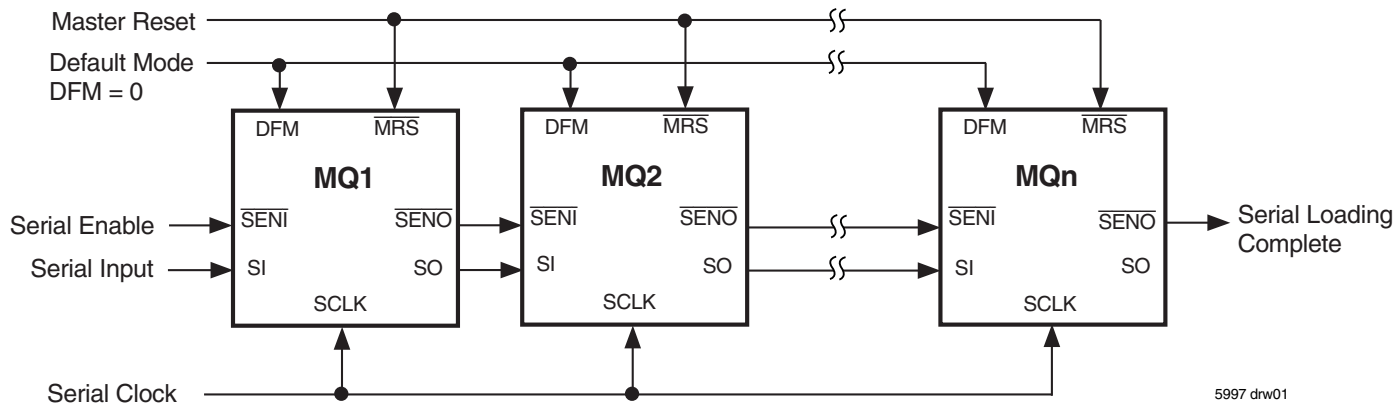


Figure 1. Serial Port Connection

SO output of the device. Therefore the $\overline{SENO}$ output has 2 purposes, firstly it can be used when devices are connected in expansion mode. In expansion mode the $\overline{SENO}$ output of device N connects directly to the $\overline{SENI}$ input of device N+1, the SO output of device N connects directly to the SI input of device N+1. This can be seen from Figure 1, *Serial Port Connection*. The $\overline{SENI}$ and SI inputs of the first device within a chain is controlled by the user and serial data required for all devices loaded here. The first device programmed is the first device in the chain, when the first device is programmed the second will be and so on. This is done by virtue of the fact that the $\overline{SENO}$ of device N connects to $\overline{SENI}$ of N+1 and SO of N to SI of N+1. So when device N is programmed its $\overline{SENO}$ goes LOW driving $\overline{SENI}$ of N+1 LOW, serial data being passed from SO of N to SI of N+1.

The second purpose of the $\overline{SENO}$ is providing the user with a programming "Done" signal, whether the Multi-Queue device is used in single device mode or expansion mode, the $\overline{SENO}$ output of the single device (or last device in a chain) should be used to indicate that programming of the Multi-Queue(s) is complete, when the final $\overline{SENO}$ goes LOW (provided that $\overline{SENI}$ is LOW) programming has been completed and the user should stop serial programming, and take $\overline{SENI}$ inactive, HIGH. This can be seen in Figure 2, *Serial Programming*. Remember, that when programming of a device is complete, the $\overline{SENO}$ output will follow the $\overline{SENI}$ input and the SO output will follow the SI input.

Therefore, in Serial programming mode the user has to provide a serial bitstream to the multi-queue flow-control device(s). This bitstream varies in length depending on the number of queues required within a device and how many devices are connected together. The number of bits required for a device can be calculated from:

Number of bits = 19 + (Q x 72), where Q is the number of queues required.

When multiple devices are connected in expansion the number of bits required will accumulate accordingly such that:

Total Number of bits = [19 + (Qa x 72)] + [19 + (Qb x 72)] + ……, where Qa is the number of queues in the first device and Qb is the number of queues in the second, and so on.

This means that the number of serial bits required can vary from 91 bits for a single device, single queue setup, to 18,5384 bits for 8 devices with 32 queues in each device. The user must generate this bitstream and store it to be dumped into the Multi-Queue(s) when required. There are multiple ways this bitstream may be generated and stored, this application note will discuss one such method open to the designer.

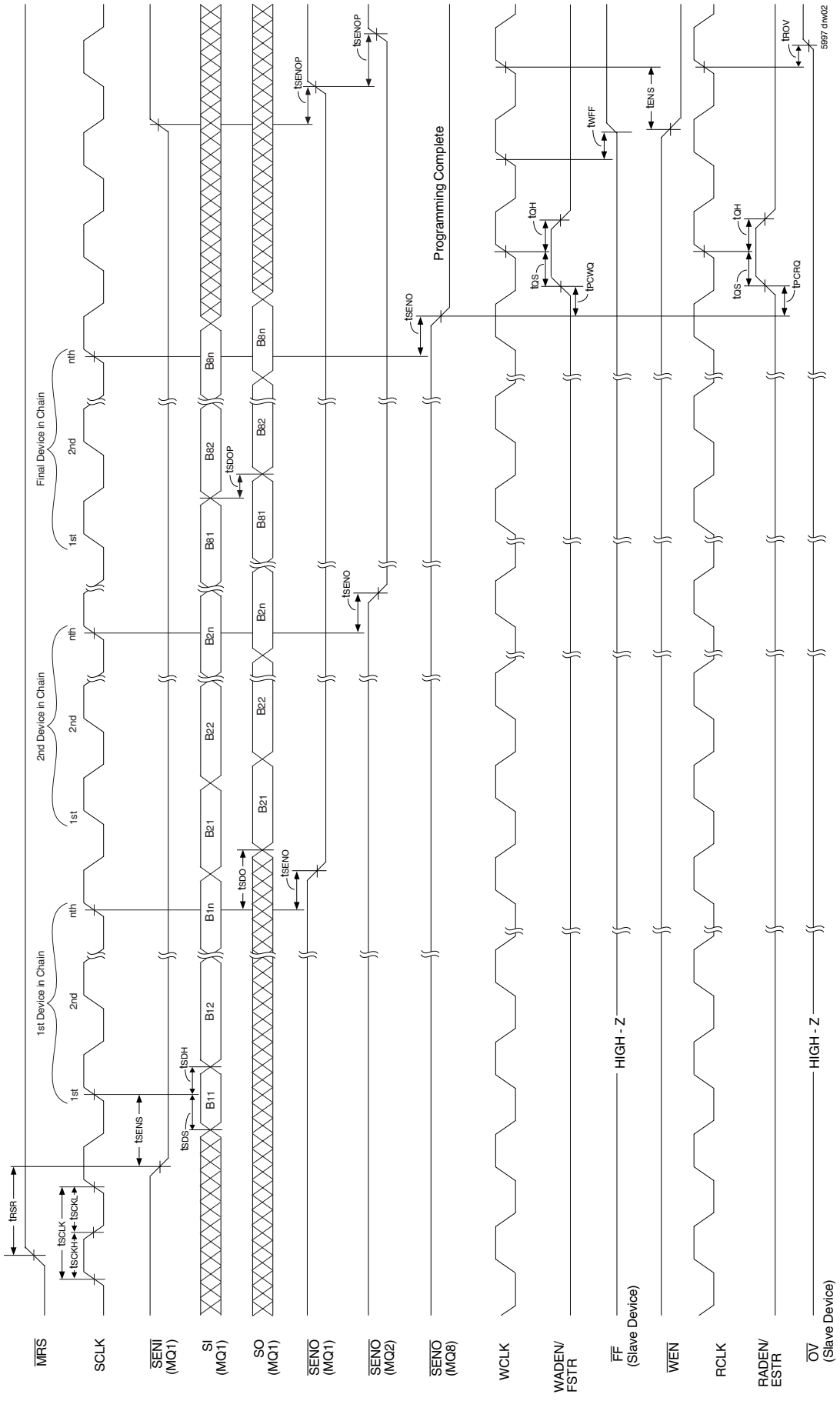


Figure 2. Serial Programming

DESIGN METHODOLOGY

IDT can provide a "Serial Bitstream Generator", which is a 'C' program that takes a user defined input file containing the user's set-up requirements. These requirements include the following:

1. The number of devices to be programmed.
2. The number of queues within each device.
3. The depth of every queue.
4. The Almost Empty and Almost Full offset values for every queue.

The generator when run will create an output file which is basically a bit file containing logical 1's and 0's that should be stored and dumped to the Multi-Queue when programming takes place. (Please contact the IDT Flow-Control Management application group for this "Serial Bitstream Generator").

The example discussed in this application note utilizes a Xicor X5323 32Kbit Serial (SPI) EEPROM that stores the serial bit file to be loaded into the Multi-Queue(s). This bit file has at some time earlier been loaded (written) into the EEPROM, this loading of the EEPROM is not discussed in this application note. This application note also utilizes an FPGA (or PLD) as the interface between the EEPROM and the Multi-Queue device(s). Also included in the application note is the associated Verilog code that shows how the interface is controlled.

Figure 3, *Hardware Set-Up for Serial Programming*, is shown below. This diagram shows that anywhere between 1 and 8 Multi-Queue devices can be connected in expansion mode, with the associated serial ports cascaded. Some important points to note from this figure are:

1. The FPGA interfaces between the EEPROM and the 1. Multi-Queue(s).
2. This set-up is the same for a single Multi-Queue device application or multiple Multi-Queue's cascaded together.
3. The $\overline{SENO}$ of the single device or the last device in a chain must feedback to the controlling FPGA and be used as a "done" signal that causes programming to cease when $\overline{SENO}$ goes LOW.
4. The serial port inputs of the first device within a chain, $\overline{SENI}$ and SI connect to the FPGA, the FPGA programs all devices in a chain via these input pins.
5. The serial bit file stored in the EEPROM is fed through the FPGA to the Multi-Queue(s).
6. The $\overline{SENO}$ of device 1 connects to the $\overline{SENI}$ of the next device, so that when programming of device 1 is complete the $\overline{SENO}$ output of device 1 takes the $\overline{SENI}$ of the next device LOW and data from SO of device 1 is written into the next device. Remember, this data is being passed through device 1, the FPGA writing data in to SI of device 1.
7. The FPGA requests serial data from the EEPROM, this is discussed in more detail later. This serial data is read from the EEPROM's SO output and in this example is gated through the FPGA from its input s_so to its output m_si.
8. Note that all the pin names of the FPGA are taken from the sample Verilog code.

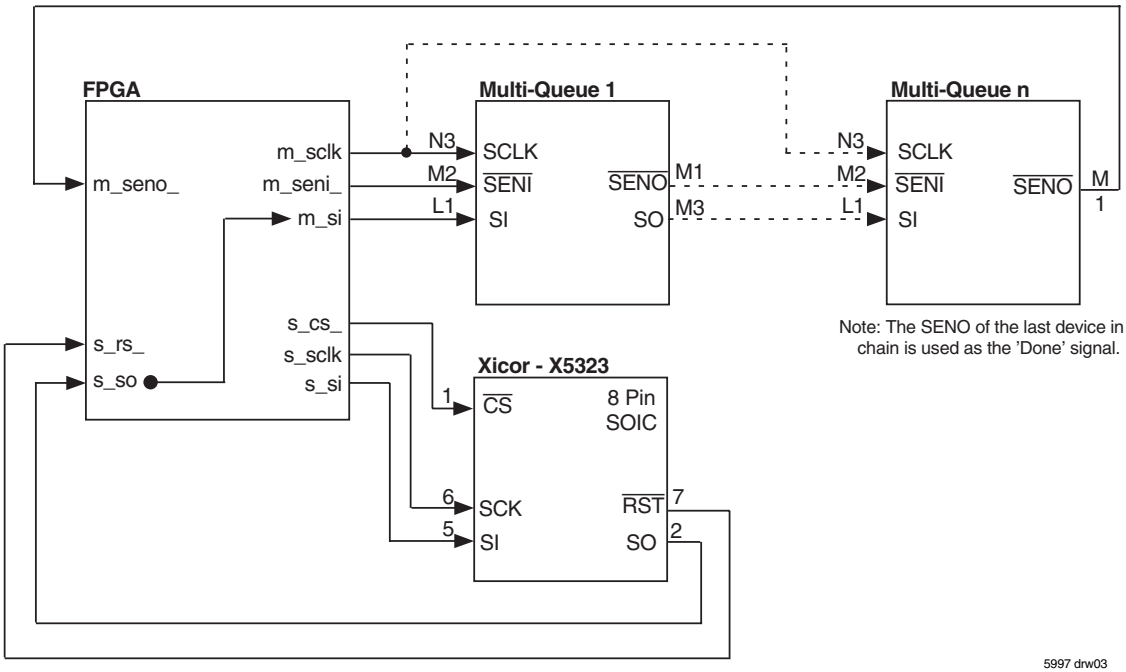


Figure 3. Hardware Set-Up for Serial Programming

PROPAGATION OF SERIAL DATA INPUT

As explained above, when Multi-Queue devices are connected in expansion mode their serial ports are cascaded. Serial data intended to program the final device in a chain must propagate through all devices. This propagation delay therefore, is accumulative and at worst there may be 8 devices connected. However, this propagation delay does not affect the performance of the serial port. This can be seen from the example below, which is based on using the slowest speed grade device:

1. The serial port for all Multi-Queue devices is rated at 10MHz, a 100ns cycle time.
2. The maximum propagation delay for a Multi-Queue device is 4ns
3. With 8 devices cascaded (maximum allowable), the total propagation delay of data written into the SI input of device 1 to the SI input of device 8 is: 7 x 4ns = 28ns (not counting any trace delays).

Therefore, we can see that there will always be a large amount of margin provide on the set-up time for serial data regardless of how many devices are cascaded. In fact there will be 52ns margin on the data set-up of the final device even with 8 devices in cascade. This is given from:

Margin = Cycle Time – (Total Propagation Time + Data Set-up Time, t_{SDS})

Margin = 100ns – (28ns + 20ns) = 52ns

THE XICOR EEPROM REQUIREMENTS

The Xicor X5323 EEPROM is a 32Kbit serial EEPROM device, it is used to store the bit file required to program the Multi-Queue(s). A bit file previously loaded into the EEPROM can be read from the EEPROM via the FPGA and used to program the Multi-Queue device(s). A read sequence is performed on the X5323 by first pulling the $\overline{CS}$ (pin 1) LOW, selecting the device. The 8-bit "read" instruction followed by the 16 bit address is transmitted to the device via the SI input (pin 5). This 16 bit address is the start address within the EEPROM from where serial data will begin reading. The user should be aware that the serial bit stream output form the X5323 will be MSB first, i.e. bit 7 of that address.

All instructions and addressing are done with respect to a rising edge of the SCK (pin 6) input. Once this has been done serial data will be output from SO (pin 2) with respect to a falling edge of SCK (pin 6). When read operations from the X5323 are complete the $\overline{CS}$ input should be taken HIGH. Figure 4 "X5323 Read Timing" shown below illustrates the read operation from the Xicor EEPROM. The instruction for a read from memory is "0000 0011". Please refer to the Xicor X5323 data sheet for more details.

The Xicor X5323 with 32Kbit memory, can easily accommodate any Multi-Queue device(s) configuration file, the largest bit file being 18.5Kbits (as mentioned). However, one could also consider using the EEPROM to store multiple files, each starting at their own respective address space in memory.

The designer should note that the maximum clock frequency of the Xicor X5323 is 2MHz.

FPGA INTERFACE

Following is sample code developed for an FPGA to deal with the interfacing between the Xicor X5323 EEPROM and the Multi-Queue device(s). This design is based on using an X5323 device with a Vcc Range of 2.7V-5.5V, this device having a maximum clock cycle time of 500ns (2MHz). This design is also based on the X5323 clock running at 1MHz, (the FPGA takes a 4MHz clock input and divides by 4 to produce a 1MHz SCLK output). The suggested part number for the Xicor EEPROM is: X5323S8-2.7

The code design may need to be re-evaluated when running at higher clock speeds or when using a different EEPROM, to ensure that there are no timing conflicts.

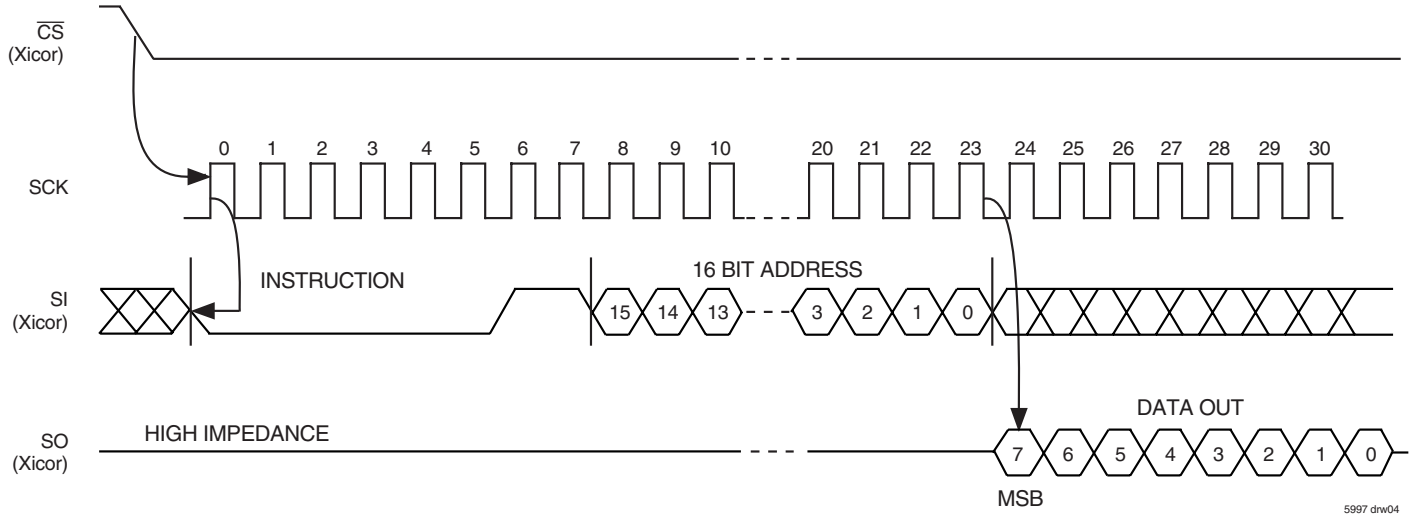


Figure 4. Read EEPROM Array Sequence

Verilog Code

```
//Mqwrite verilog synthesis file
//Serial EEPROM functions for Multi-Queue Validation Board.

//Module to divide down serial clock
module divides clk(reset_,clkin,clkout);
        input reset_;
        input clkin;                                    //4.096MHz input clk
        output clkout;
        reg [1:0] div;                                  //for clk divider
        wire clkout;
        //Xicor X5323 maximum clk=2MHz so divide clkin by 4, about 1MHz
        assign clkout = div[1];
        always @ (posedge clkin or negedge reset_)
        begin
                if (!reset_)
                        div <= 2'b00;                   //clear registers
                else
                        div <= div + 2'b01;             //count up
        end
endmodule //divides clk

//************************************************************
//This module interfaces the serial SPI EEPROM, Xicor X5323
//or similar, to the Multi-Queue serial programming port.
//Assume that the EEPROM has already been programmed.
//Initially, after master reset, read serial bitstream out
//of the EEPROM and into the Multi-Queue device(s) until SENO_ goes
//low.  This should also work in default mode.
//************************************************************

module serialprog(reset_,mrs_,sclk,s_rs_,s_so,s_sclk,s_cs_,s_si,
            m_so,m_seno_,m_sclk,m_seni_,m_si);
        input reset_;                                   //master reset input
        output mrs_;                                    //master reset to Multi-Queue
        input sclk;                                     //input clk
        input s_rs_,s_so;                               //EEPROM outputs
        output s_sclk,s_cs_,s_si;                       //EEPROM inputs
        input m_so,m_seno_;                             //Multi-Queue serial outputs
        output m_sclk,m_seni_,m_si;                     //Multi-Queue serial inputs

        wire s_sclk,m_sclk;
        wire res_sync_;
        reg s_cs_,s_si;
        reg mrs_,m_seni_;
        reg [3:0] bitcount;                             //bit counter
        reg [2:0] state;                                //state machine bits
//state definitions
        parameter start =       3'b000;
        parameter r1 =          3'b001;
        parameter r7 =          3'b011;
        parameter r8 =          3'b010;
        parameter addr =  3'b110;
        parameter data =  3'b111;
        parameter idle =  3'b101;
```

```
assign s_sclk = sclk;                          //EEPROM and Multi-Queue both on sclk
assign m_sclk = sclk;
assign m_si   = s_so;                          //Pass EEPROM data to Multi-Queue

always @ (posedge sclk or negedge reset_)
        if (!reset_)
                state <= start;
        else case (state)
                start:
                begin
                        s_cs_ <= 1;
                        s_si <= 0;        //start of EEPROM READ b'00000011(msb 1st)
                        mrs_ <= 0;                      //reset Multi-Queue
                        m_seni_ <= 1;
                        bitcount <= 4'd0;
                        if (reset_)                     //wait for reset to end
                                state <= r1;
                        else
                                state <= start;
                end
                r1:                                     //next bit of EEPROM READ
                begin
                        s_cs_ <= 0;                     //start selecting EEPROM
                        s_si <= 0;                //bits 1-5 of EEPROM READ b'00000011
                        mrs_ <= 1;
                        m_seni_ <= 1;
                        bitcount <= bitcount + 1;
                        if (bitcount < 4'd5)            //send bits 1-6
                                state <= r1;            //stay here
                        else
                                state <= r7;
                end
                r7:                                     //next bit of EEPROM READ
                begin
                        s_cs_ <= 0;
                        s_si <= 1;                      //bit 7 of EEPROM READ b'00000011
                        mrs_ <= 1;
                        m_seni_ <= 1;
                        state <= r8;
                end
                r8:                                     //next bit of EEPROM READ
                begin
                        s_cs_ <= 0;
                        s_si <= 1;                      //bit 8 of EEPROM READ b'00000011
                        mrs_ <= 1;
                        m_seni_ <= 1;
                        bitcount <= 4'd0;
                        state <= addr;
                end
                addr:                                   //address bits of EEPROM READ
                begin
                        s_cs_ <= 0;
                        s_si <= 0;                      //all address bits=0
                        mrs_ <= 1;
                        m_seni_ <= 1;
                        bitcount <= bitcount + 1;
                        if (bitcount < 4'd15)           //16 address bits
```

```
                        state <= addr;
                else
                        state <= data;
        end
        data:                                   //send data bits
        begin
                s_cs_ <= 0;
                s_si <= 0;
                mrs_ <= 1;
                m_seni_ <= 0;                   //enable Multi-Queue serial in
                bitcount <= 4'd0;
                if (m_seno_)                    //wait for seno_=0 from Multi-Queue
                        state <= data;
                else
                        state <= idle;          //done
        end
        idle:                                   //idle state, stay here until reset
        begin
                s_cs_ <= 1;
                s_si <= 0;
                mrs_ <= 1;
                m_seni_ <= 1;
                bitcount <= 4'd0;
                state <= idle;
        end
        default:
        begin
                s_cs_ <= 1;
                s_si <= 0;
                mrs_ <= 1;
                m_seni_ <= 1;
                bitcount <= 4'd0;
                state <= idle;
        end
    endcase

endmodule //serialprog
```

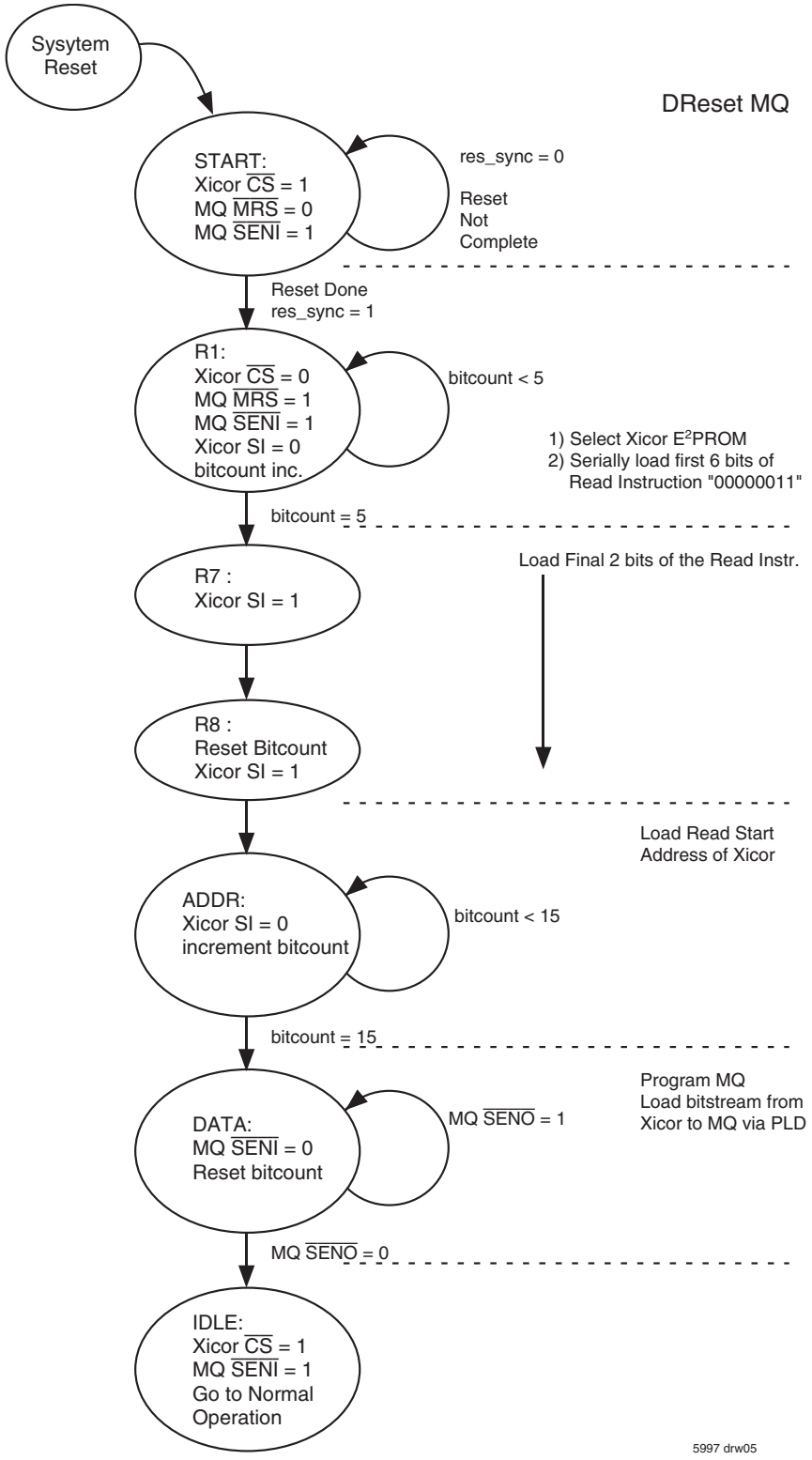


Figure 5. Serial Programming State Diagram

SERIAL PROGRAMMING REGISTERS DEFINED

The serial bitstream file generated from the C program is nothing more than a binary file. By defining the bits in the bitstream file pertaining to the internal registers, the user can modify certain parameters (such as $\overline{PAE}$, $\overline{PAF}$ offset values), without having to use the serial bitstream generator.

Because the number of bits required to program a Multi-Queue are relatively few, storing the bitstream in an FPGA is a viable alternative. The number of bits needed to program a Multi-Queue depends on the number of queues desired. The minimum number of bits needed to program the Multi-Queue is 91 bits to program a single queue configured device. The maximum number of bits needed to program the Multi-Queue is 2323 bits for a full 32 queues. The required number of bits for any single chip application is calculated using the following formula: Total number of bits = 19 + (N x 72) where N is the number of queues desired.

BITSTREAM REGISTER VALUES DEFINED

The registers in the Multi-Queue are 18 bits wide. The first 18 bits of the serial bitstream are defined as the Header register, the next four 18 bit registers define the first queue parameters the, second four 18 bit registers define the second queue parameters and so on for each queue. The final bit of the serial bitstream is a single stop bit.

To program multiple devices simply concatenate each devices header register and queue registers in series to the first device's string. A stop bit is required at the end of each device's configuration section. Figure 6 shows the format structure of the bitstreams and identifies the sections in the bitstream. Note that the first bit in the serial bitstream is on the far left (MSB) progressing to the right (LSB) through each register in succession.

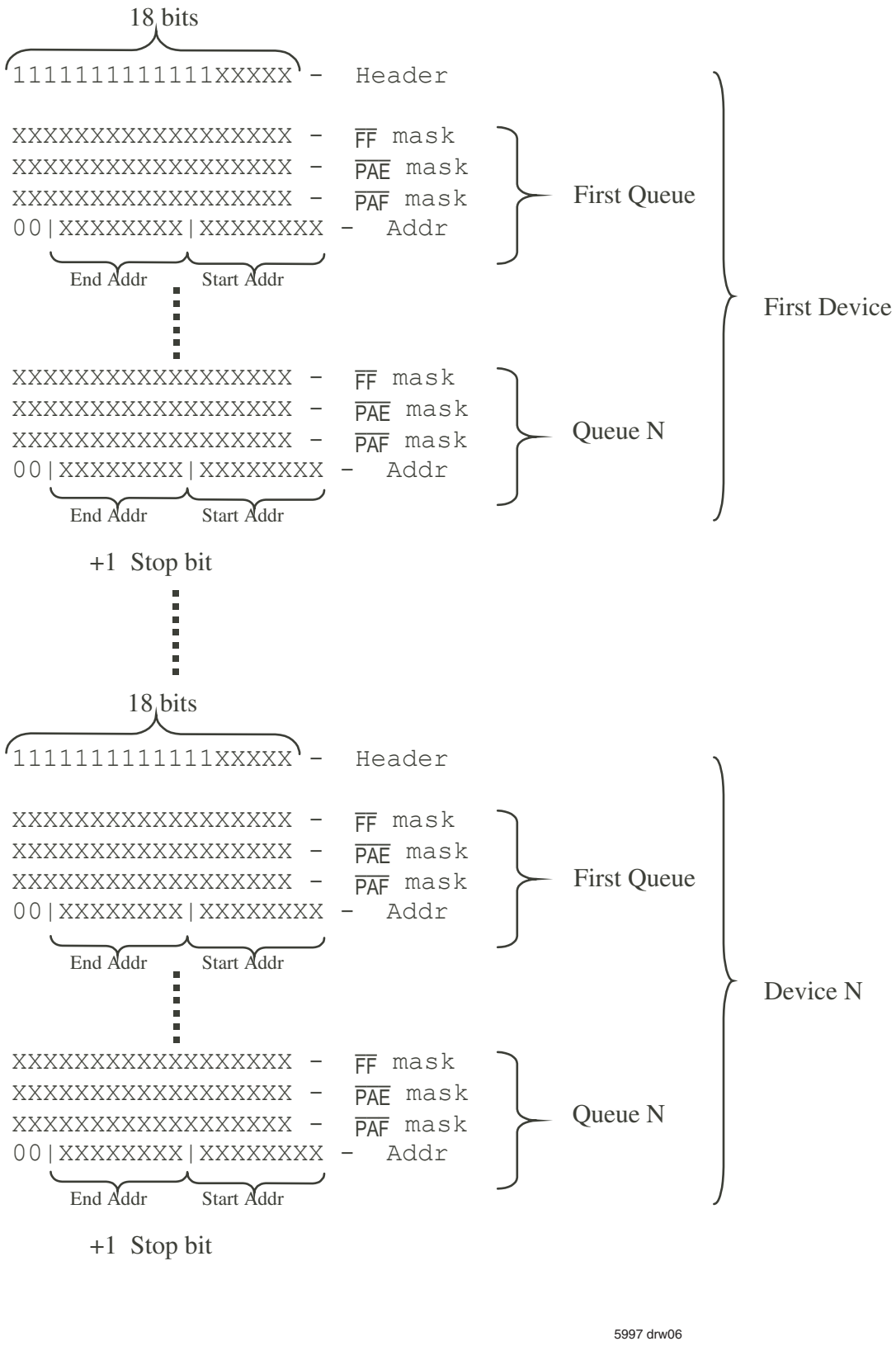


Figure 6. Bit Streams Format Structure

EACH OF THE 18 BIT WIDE REGISTERS ARE FURTHER DESCRIBED BELOW:

Header: This is an 18-bit word and has the following components. 13 ones as the MSB, which are used for error detection, followed by a five bit LSB indicating the number of queues to be programmed (equal to (#Q-1)). Ex: Five queues requires the binary value "00100". The header is needed only once for each device.

1	1	1	1	1	1	1	1	1	1	1	1	1	Q4	Q3	Q2	Q1	Q0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$\overline{\text{FF}}$ mask: This is an 18-bit word and represents the Full Flag mask. It is equal to ~(Qdepth-2). Each queue requires a separate FF mask. (~ Denotes inversion). Qdepth is the number of words in queue where word width is the maximum of the input or output port width. Ex: input port = x9, output port = x18, and queue depth = 3k (3072x18) then subtract two and invert. Resultant serial stream is shown in the figure below.

1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$\overline{\text{PAE}}$ mask: This is an 18-bit word and represents the partial empty mask. It is equal to ~(PAE Offset). Each queue requires a separate PAE mask. PAE Offset is the number of words above the empty queue where word width is the maximum of the input or output port width. Ex: input port = x18, output port = x9, and PAE Offset = 256 words (x18 wide) from empty queue. Resultant serial stream is shown in the figure below.

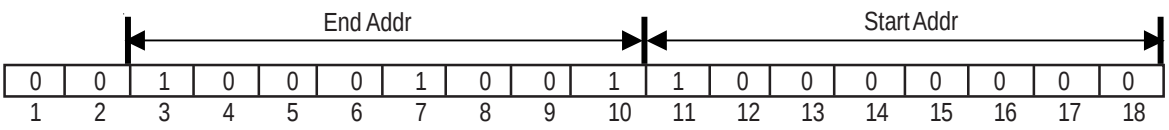
1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$\overline{\text{PAF}}$ mask: This is an 18-bit word and represents the partial full mask. It is equal to ~(Qdepth-PAF Offset). Each queue requires a separate PAF mask. PAF Offset is the number of words below the full queue where word width is the maximum of the input or output port width. Ex: input port = x9, output port = x18, Qdepth is 5k (1024x5=5120), and PAF Offset = 224 words (x18 wide) from full queue. Resultant serial stream is shown in the figure below.

1	1	1	1	1	0	1	1	0	0	1	1	0	1	1	1	1	1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Start Address: This is an 8-bit word and represents the start address of each queue in memory. Start addresses are specified in increments of 1Kx9 words. 2Mbit Multi-Q devices start at binary address "00000000", 1Mbit Multi-Q at binary address "10000000", 512kbit Multi-Q at binary address "11000000", and 256kbit Multi-Q at binary address "11100000". The first queue should always start at this address, subsequent queues should start at an address just above the previous queue's end address.

End Address: This is an 8-bit word and represents the end address of each queue in memory. End addresses are specified in increments of 1K x 9 words. The end address for a particular queue should be the number of 1Kx9 blocks (minus one) which will cover the FF Mask value of that queue. For example, if the queue depth is to be 8Kx9 then add binary "111" to the start address, if the queue depth is to be 1Kx9 then add zero to the start address to obtain the end address value. Ex: a single queue 10Kx9 depth queue in a 1Mbit Multi-Q device is shown in the figure below. Note: Under this example the next queue in this device would start at binary address "10001010".



CONCLUSION

The IDT Multi-Queue requires a serial bitstream for configuring the device prior to operation if other than the default queue parameters are desired. A binary bitstream file can be generated using a C program provided by IDT. In the cases where the user wishes modify minor configurations in the Multi-Queue without having to generate a bitstream file every time, the user can modify or create the binary file directly using the information detailed in this application note.