

Programming the TLD Sequencer

John Doty
Noqsi Aerospace Ltd.

Copyright 2007, Noqsi Aerospace Ltd.

This work is licensed under the Creative Commons Attribution-Share Alike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Introduction

The Trakimas-Larosa-Doty sequencer was developed at MIT in 2003 as a means to control the clock and measurement timing of CCD imagers in development. Hardware details, and the programs described here, are available at <http://www.noqsi.com/engineering.html>. The compiler/interpreter for the LSE64 programming language used is available at <http://www.noqsi.com/software.html>. The zip package includes introductory and reference documentation on the language itself.

This document is intended as a programming tutorial for those who need to write their own microcode for this sequencer.

A Simple Program

Here's what is perhaps the simplest useful TLD sequencer program:

```
" TLDsequencer.lse" load

15 2 seq{
  cycle :block{
    0 bit high step
    0 bit low step
  }block
}seq

1000 cycle goSeq repeat
```

Here's what it does:

```
" TLDsequencer.lse" load
```

loads the LSE64 definitions for the microcode generator. The space character after the first " is required (this is like other Forth dialects).

```
15 2 seq{
```

This resets the sequencer, sets the clock divider to 15, and sets the block size (also known as "sequence length" in the hardware spec) to 2. As I'm using a 30 MHz sequencer, this means that the sequencer will step at 2 MHz, and each block will take 1 μ s to execute.

```
cycle :block{
```

This starts generation of a block named "cycle".

```
0 bit high step
```

This generates a microcode word with bit 0 high, then steps to the next microcode location.

```
0 bit low step
```

This generates a microcode word with bit 0 low, then steps to the next microcode location. That location is beyond the end of the block, so the block is complete.

```
}block
```

This checks to make sure you've exactly filled the block, exiting if you haven't. It also checks that you've left the bit state the same at the end as at the beginning, and warns you if you haven't (more details below). It issues pad words to the sequencer to fill the unused memory up to the next block boundary.

```
}seq
```

This issues pad words to fill the remaining unused sequencer memory. At this point, the sequencer is ready for commands.

```
1000 cycle goSeq
```

This issues the sequencer command to perform the block "cycle" 1000 times. In this case, the result is 1 ms of 1MHz square waves on sequencer output bit 0.

```
1000 cycle goSeq repeat
```

The LSE64 function "repeat" just jumps back to the start of the line, so this line keeps issuing these commands indefinitely, and thus generates a continuous 1MHz square wave until the program is interrupted.

What the Hardware Sees

The control computer connects to the sequencer using a standard PC "parallel printer

port". The command "15 2 seq{" first pulses the INIT line on the parallel port to place the sequencer in the initial state, ready to accept program data. Then it issues two bytes:

01 00 (hex)

these represent the 16 bit hexadecimal number 0001 (multibyte fields go least significant byte first here). This tells the sequencer that the block size is two: it is always one more than the number here.

It then issues one more byte:

0E

This tells the sequencer to divide its master clock by 15 (decimal) when stepping through memory. Again, the divider is one more than the number here. For a 30MHz master clock, this means the sequencer will step through its microcode memory at a 2 MHz rate.

"cycle :block{" sets up to generate a block of code, but doesn't send any data to the sequencer. "0 bit high step" sends four bytes:

01 00 00 00

This represents the state of the 32 sequencer output bits for the first step in the "cycle" block. The least significant byte is first, so this has bit 0 set to 1, all others set to 0 by default. "0 bit high step" sends four more bytes:

00 00 00 00

making the state of all bits 0 for the second step in the block.

"}block" fills the 1022 unused locations in the block with an arbitrary bit pattern:

DE AD C0 DE

"}seq" does the same for the remaining 64512 locations in the sequencer memory. With its memory full, the sequencer automatically enters the "idle" state, ready to accept commands.

"1000 cycle goSeq" sends two bytes to the sequencer:

E7 03

representing the 16 bit hexadecimal number 03E7. The six most significant bits of this are the six most significant bits of the block start address (the rest are always zero).

The remaining 10 bits represent the repeat count, 3E7(hex) = 999 (decimal), and the actual count is one more than this number.

For this program, the control computer will continue to issue these command bytes until the sequencer asserts the BUSY bit on the parallel port, indicating that its input FIFO memory is full. It will then wait for BUSY to go low, and then issue more command bytes until BUSY goes high again, continuing this indefinitely.

Practical Timing Considerations

My Pentium workstation can issue about 100,000 bytes/second on its parallel port, or a two byte sequencer command every 20 μ s. The command FIFO can hold 32 commands: if it becomes empty, the sequencer will enter the idle state and stop generating new outputs until a new command comes in. It is often desirable to avoid the idle state. This requires that the necessary command rate be low enough to keep the FIFO from becoming empty. In this simple case, with a 30 MHz sequencer, a clock divider of 15 (making the step time 0.5 μ s), a block length of 2 (making the block time 1 μ s), and a repeat count of 1000, each command takes 1 ms, and it takes 32 ms to empty a full FIFO. That's adequate to stay out of the idle state even without a real time kernel.

A more realistic program will alternate between high repeat counts (e.g. CCD pixel readout) and low repeat counts (e.g. parallel to serial transfers). Usually, the high repeat count commands take most of the time, so there is little difficulty here.

Using the LSE Language

The microcode generator is just a set of definitions in LSE64. You can use them with the rest of the LSE language to parameterize programs, name things, and do other things to clarify the code.

The sequencer I have on my bench has a 30 MHz master clock, but future versions may have different clock rates. The TLDsequencer.lse file defines a constant, seqTicks/s, that represents the clock rate. So, the expression

```
seqTicks/s 2000000 /
```

evaluates to 15, the clock divider we want for a 2 MHz step rate. Note that symbols ("word names") names in LSE can contain any non-blank character, so the "/" in "seqTicks/s" is just part of the symbol, but the "/" at the end of the expression, separated by whitespace means "divide".

You can define your own constants.

```
seqTicks/s 2000000 / clockDivider :constant
```

This evaluates "seqTicks/s 2000000 /" and creates a new constant, "clockDivider". Below, we'll also compute the 2000000 from the desired frequency.

The word "bit" simply generates a bitmask from its argument. "0 bit" is 1, "4 bit" is 16, etc. Rather than write down bit numbers or masks all of the time, it is usually clearer to define named constants:

```
0 bit wave :constant
```

And finally, the word "#" introduces a comment, which continues to the end of the line. So, the program above may be written:

```
# Program to generate a 1 MHz square wave on sequencer bit 0

" TLDsequencer.lse" load

1000000 Hz :constant      # 1 MHz
2 steps/block :constant # must match the block definition below
seqTicks/s Hz steps/block * / clockDivider :constant

0 bit wave :constant

clockDivider steps/block seq{ # generate the microcode
    cycle :block{
        wave high step
        wave low step
    }block
}seq

1000 cycle goSeq repeat      # run the microcode
```

Clocking a CCD

Imagine we have a simple 200 by 200 full frame 3 phase CCD with one output and a serial register that extends 10 pixels beyond the bottom of the imaging area. We want to expose for 1 s, read out at 1 Mpix/s. Here's one way to clock it.

```
" TLDsequencer.lse" load

1000000 pix/s :constant # readout rate
6 steps/block :constant # must match the block definitions below
seqTicks/s pix/s steps/block * / clockDivider :constant

# CCD structure

200 rows :constant
200 columns :constant
10 extpix :constant      # serial extended pixels

# number of pixel cycles per row (derived constant)
```

```

extpix columns + pix/row :constant

0 bit IA1 :constant          # imaging area
1 bit IA2 :constant
2 bit IA3 :constant
3 bit S1  :constant          # serial register
4 bit S2  :constant
5 bit S3  :constant
6 bit RG  :constant          # reset gate

clockDivider steps/block seq{ # generate the microcode

    IA1 high          # establish clock state between pixels
    IA2 high
    IA3 low
    S1  high
    S2  high
    S3  low
    RG  low
    setinitstate

    pixel :block{
        RG high S1 low step
        RG low S3 high step
        S2 low step
        S1 high step
        S3 low step
        S2 high step
    }block

    down :block{
        IA1 low step
        IA3 high step
        IA2 low step
        IA1 high step
        IA3 low step
        IA2 high step
    }block

    idle :block{
        step step step step step step
    }block
}seq

# Microcode is done, now program the control computer

expose : pix/s idle goSeq          # idle for one second
readRow : 1 down goSeq pix/row pixel goSeq # read out one row
readout : rows readRow iterate     # read out all rows

# Everything is ready, just do a bunch of frames:

expose readout repeat

```

This program introduces several useful ideas. Before the first block, it establishes the "initial state" the outputs will be assumed to be in between pixels. It uses "high" and "low" commands followed by "setinitstate" to accomplish this. Each ":block{" command implicitly assumes that's the state before changes are commanded with "high" or "low". "}"block" will issue a warning if the state at the end of the block differs from the initial state. This isn't an error, but it may cause outputs to change without an explicit "high" or "low" command if you execute a block that leaves the outputs in an unexpected state followed by a block that expects them to be in the initial state.

We now have three different blocks, whose execution needs to be put together in a complicated rhythm. This will require a more complex sequence of commands from the host computer. There are three LSE definitions to help here.

```
expose : pix/s idle goSeq
```

This defines a new LSE "word" that issues commands to execute the "idle" block 1 million times. Because the hardware repeat count is limited to 1024, goSeq will actually issue 975 sequencer commands (0BFF hex) with a count of 1024, and one (0A3F hex) with a count of 576, when this definition is executed.

```
readRow : 1 down goSeq pix/row pixel goSeq
```

This word issues the sequencer command (0400 hex) to clock one row down, followed by the sequencer command (00d1 hex) to clock the serial register 210 times.

```
readout : rows readRow iterate
```

This word invokes readRow 200 times to read out the entire CCD. Here, I introduce the word "iterate". Its usage is similar to "goSeq", except "goSeq" repeatedly executes a sequencer block, while "iterate" repeatedly executes an LSE definition.

Note that excluding unused padding, the microcode only occupies 18 words of the 65536 word sequencer memory. The "pixel" block is:

```
00000053
00000033
00000023
0000002b
0000000b
0000001b
```

The "down" block is:

```
0000001a
0000001e
```

```
0000001c
0000001d
00000019
0000001b
```

And the "idle" block is:

```
0000001b
0000001b
0000001b
0000001b
0000001b
0000001b
```

A More Complicated Example

Imagine now that the parallel clocking circuitry is slower than the serial clocking circuitry. Here's a modification of the program above that splits the parallel clocking into six blocks:

```
" TLDsequencer.lse" load

1000000 pix/s :constant # readout rate
6 steps/block :constant # must match the block definitions below
seqTicks/s pix/s steps/block * / clockDivider :constant

# CCD structure

200 rows :constant
200 columns :constant
10 extpix :constant          # serial extended pixels

# number of pixel cycles per row (derived constant)

extpix columns + pix/row :constant

0 bit IA1 :constant          # imaging area
1 bit IA2 :constant
2 bit IA3 :constant
3 bit S1  :constant          # serial register
4 bit S2  :constant
5 bit S3  :constant
6 bit RG  :constant          # reset gate

clockDivider steps/block seq{ # generate the microcode

    IA1 high          # establish clock state between pixels
    IA2 high
    IA3 low
    S1  high
```

```

S2 high
S3 low
RG low
setinitstate

pixel :block{
    RG high S1 low step
    RG low S3 high step
    S2 low step
    S1 high step
    S3 low step
    S2 high step
}block

down1 :block{
    IA1 low step step step step step
    setinitstate
}block

}down : 6 step iterate setinitstate }block # convenience

down2 :block{
    IA3 high
}down

down3 :block{
    IA2 low
}down

down4 :block{
    IA1 high
}down

down5 :block{
    IA3 low
}down

down6 :block{
    IA2 high
}down

idle :block{
    6 step iterate
}block

}seq

# Microcode is done, now program the control computer

expose : pix/s idle goSeq # idle for one second
readRow : 1 down1 goSeq \
          1 down2 goSeq \
          1 down3 goSeq \
          1 down4 goSeq \
          1 down5 goSeq \

```

```

        1 down6 goSeq \
          pix/row pixel goSeq
readout : rows readRow iterate          # read out all rows

# Everything is ready, just do a bunch of frames:

expose readout repeat

```

Look at the definition of the block "down1". All It does is set IA1 low. It uses "setinitstate" to change the initial state to its end state, so that "down2" will start from that point. This is useful when you always intend to execute a sequence of blocks in order, as in this case.

Since these blocks all have a pattern to them, I went ahead and captured most of that pattern in the LSE definition "down". This illustrates the fact that the words that build the microcode are simply LSE definitions that emit microcode when executed. You can embed them in higher level LSE definitions, and they'll do the same thing. The exception is "block{", a "parsing word" that operates on its preceding token when compiled. Don't put "block{" in a definition.

"readRow" is now too complicated to fit on one line, so "/" must be used to continue the definition across lines.

One disadvantage of this approach is that "readRow" now issues 7 commands per row, rather than just two. That's over 30,000 commands/s. Depending on the nature of the computer controlling the sequencer and the load other software places on it, that may be a problem.

A More Efficient Approach

We can reduce the command load on the control computer by making better use of the sequencer's large memory. The last version had 8 blocks of 6 words each, thus using less than 0.1% of the 65536 word memory. But blocks can be much longer: the only restrictions are that all blocks must be the same length, each block occupies the next largest multiple of 1024 words, and the total memory available is 65536 words. So, if your block size is 2500 words, each block actually occupies 3072 words, and you may have up to 21 blocks.

```

" TLDsequencer.lse" load

1000000 pix/s :constant # readout rate
6 steps/pix :constant
seqTicks/s pix/s steps/pix * / clockDivider :constant
6 steps/pclock :constant

# CCD structure

200 rows :constant

```

```

200 columns :constant
10 extpix :constant          # serial extended pixels

# number of pixel cycles per row (derived constant)

extpix columns + pix/row :constant

# and now we can calculate steps/block

pix/row steps/pix * 6 steps/pclock * + steps/block :constant

# approximate blocks/s
seqTicks/s clockDivider / steps/block / blocks/s :constant

0 bit IA1 :constant          # imaging area
1 bit IA2 :constant
2 bit IA3 :constant
3 bit S1  :constant          # serial register
4 bit S2  :constant
5 bit S3  :constant
6 bit RG  :constant          # reset gate

# generate microcode for one pixel

pixel :      RG high S1 low step \
             RG low S3 high step \
             S2 low step \
             S1 high step \
             S3 low step \
             S2 high step

# long step for parallels

pstep : steps/pclock step iterate

clockDivider steps/block seq{ # generate the microcode

    IA1 high          # establish clock state between pixels
    IA2 high
    IA3 low
    S1  high
    S2  high
    S3  low
    RG  low
    setinitstate

    row :block{
        IA1 low pstep
        IA3 high pstep
        IA2 low pstep
        IA1 high pstep
        IA3 low pstep
        IA2 high pstep

```

```

        pix/row pixel iterate
    }block

    idle :block{
        steps/block step iterate
    }block
}seq

# Microcode is done, now program the control computer

expose : blocks/s idle goSeq          # idle for one second
readout : rows row goSeq              # read out all rows

# Everything is ready, just do a bunch of frames:

expose readout repeat

```

Here, I've made a LSE definition, "pixel" that generates the microcode for one pixel. It's rather long: LSE is designed for one-line definitions. I have to suppress the newlines in it with "\". There's also a "pstep" for the multiple steps between parallel clock transitions. But then, my "row" block just does the parallel transfer and then iterates the "pixel" word to get the whole row out.

Note that steps/block is now 1296, much larger than previous versions, but only two blocks are required, "row" and "idle". A single command suffices to read out the entire chip (200 iterations of "row"), and five commands suffice for the exposure delay (4629 iterations of "idle"). That's only 6 commands/s. But the exposure is no longer exactly one second: if there was a hard requirement here I'd want to reconsider the block size (perhaps putting in "overclocked" pixels) and possibly the clock divider in order to make the block duration exactly divide one second.

Conclusion

Of course, microcode for a real CCD is more complicated. This code assumes an unusually simple CCD. Often you'll want to support other kinds of transfers (charge injection, pixel summation). The microcode usually needs to provide timing for the measurement chains, too. But I think I've covered the basic ideas behind the microcode design and implementation.