AN5279
Application note

A bypass method to utilize the three-phase gate driver of the STSPIN32F0

Dennis Nolan

Introduction

While the STSPIN32F0 includes a powerful STM32F0 microcontroller core which is capable of performing all of the basic computation and control tasks of either a field oriented (sinusoidal) of six-step (trapezoidal) BLDC motor drive, situations may arise where a different microcontroller is required. Perhaps a more powerful microcontroller is required for the application or a legacy controller is already in place. In these instances, the full-featured three-phase gate driver functionality of the STSPIN32F0 along with its built in analog and switch mode power supplies can still be utilized in a motor drive design in a very cost effective manner. A well-integrated overcurrent protection comparator and four high quality op amps also come with the package.

In order to utilize the high performance “six-pack” gate driver built into the STSPIN32, a very small utility program must be loaded into the STM32 core to effectively “pass through” the six gate drive control signals to the gate drivers. We wish to do this with minimal propagation delay. The port pin assignments of the gate driver inputs are hard wired within the STSPIN32 and are as follows:

PA10 HS3

PA9 HS2

PA8 HS1

PB15 LS3

PB14 LS2

PB13 LS1

The best choice for the six input port pins to control the gate drivers are bits 0-5 of port A. Since a straight-forward programming approach would require logic and bit shifting operations to translate the format of the input data to the required format of the output data, a table lookup approach is chosen as it allows for arbitrary logic translation and is the fastest execution technique.

Port configuration is as follows:

Port A, bits 8, 9 and 10 as push-pull output. All other bits as general purpose inputs.

Port B, bits 13, 14 and 15 as push-pull output. All other bits as general purpose inputs.

The program itself could not be simpler. The first line reads a 16 bit data element from table A and writes that data to port A. Table A is a data table in flash memory which contains 256 16 bit data values. The index (pointer) into the 256 element table is provided by the value read from port A. Since port A is logically a 16 bit wide port it must be cast using (unsigned char) so that we only use the low byte. Actually we are only using bits 0 through 5 of port A as relevant inputs and do not care about the state of bits 6 and 7.

For execution expediency we do not mask these bits. This can all be taken care of with the arbitrary logic of the lookup table. In fact, if the final lookup table data is closely examined we see the pattern that the full 256 element array is actually a 64 element block repeated

four times. This gives us the same data, regardless of which of the 4 sub-blocks the bit field [pa7,pa6] may be pointing to. It is a truly “don't care” situation.

In a similar manner, the second line provides the data for port B. Since the “hardwired” situation is that the six gate driver pins are split up between ports A and B, we must have two lines of code. Of course, the data in table B is different (it still has the 4 repeating blocks). The great advantage of the table lookup method is that this logic is quite arbitrary and we can make it anything we like. Even faced with the ridiculous situation that five of the logic inputs are positive logic and one is inverted, it presents no problems. It may be a bit tedious to craft the modified data for the tables, but it is certainly possible.

C language program

```
While(1) // endless loop
{
GPIOA->ODR = tablea[ (unsigned char) GPIOA->IDR ];
GPIOB->ODR = tableb[ (unsigned char) GPIOA->IDR ];
}
...............
```

Test results

Since, as noted, the complete construction of the lookup tables is a bit tedious, test tables have been defined in order to allow for the true propagation delay to be measured. The tables are as follows:

```
const unsigned short tablea[256] = {
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
};
const unsigned short tableb[256] = {
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
```

```
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,0,b13,
};
```

Table A is all zeroes. This is because, for this test, we aim to not use any of the high side gate drives, which are all on port A. For the purposes of this test, we just want the input PA0 to control gate drive LS1 (which is controlled by PB13). The tables are 16 rows of 16 elements for a total of 256. For table B we started with all zeroes (like table A) and then went back and changed all of the odd elements. This is because PA0 being low corresponds to the even table elements and PA0 high corresponds to the odd elements. Each odd element of the table has been set to the symbol b13. As a programming convenience, the symbol b13 was previously defined as the 16 bit data word with all zero bits except bit 13.

The tests were conducted by applying a 20 KHZ, 50% duty cycle, square wave from a lab signal generator to PA0 and observing the output at LS1. *Figure 1* shows the overall response. *Figure 2* is taken at fast sweep to show the propagation delay on rising edge. It appears to be typically around 0.5 uS. *Figure 3* is the same as figure 2 but taken with long persistence envelop trigger mode to show the range of propagation delay.

Figure 4 and *5* complete the story for falling edge. This appears quite similar to the rising edge case, as expected.

Conclusions

This program provides the required functionality, with expected propagation delay in the range of 0.2 to 0.6 microseconds for both rising and falling edges.

Figure 1. Channel 1 (yellow) LS1 gate drive output at 5 volts per division (zero at center) Channel 2 (blue) PA0 input at 2 volts per division (zero at -3 divisions) Sweep speed at 10 microseconds per division

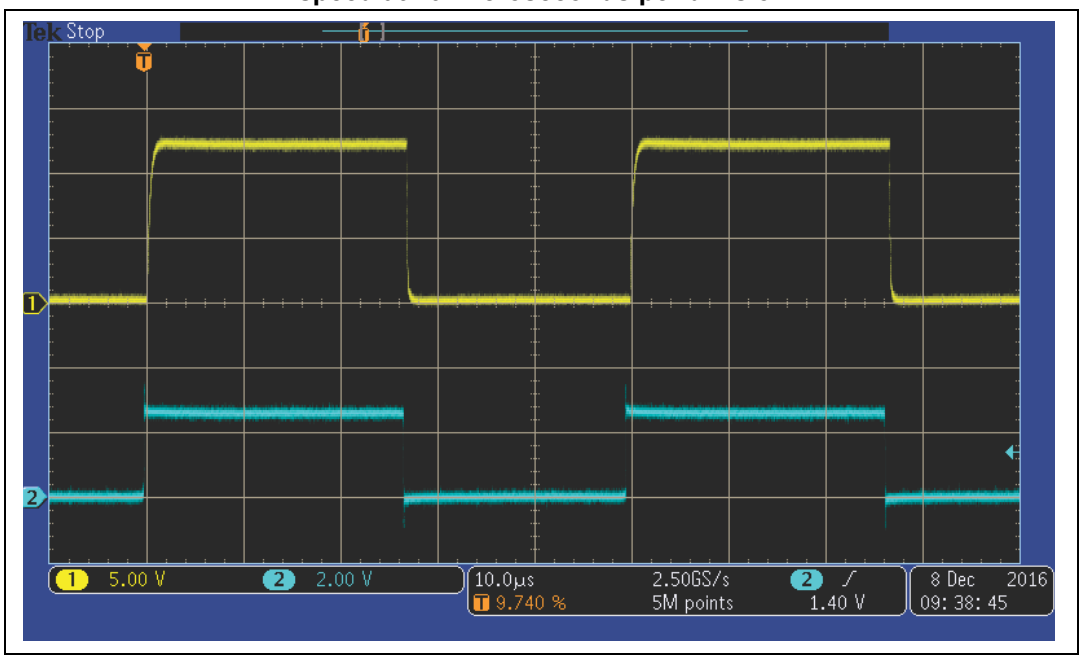


Figure 2. Channel 1 (yellow) LS1 gate drive output at 5 volts per division (zero at center) Channel 2 (blue) PA0 input at 2 volts per division (zero at -3 divisions) Sweep speed at 1 microseconds per division

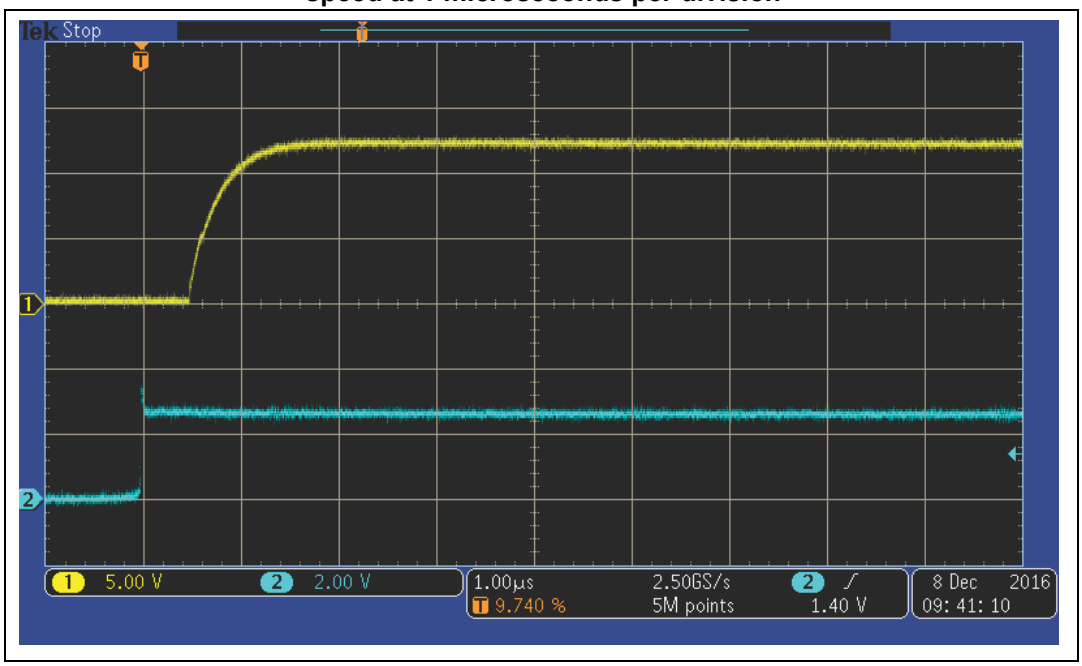


Figure 3. Channel 1 (yellow) LS1 gate drive output at 5 volts per division (zero at center) Channel 2 (blue) PA0 input at 2 volts per division (zero at -3 divisions) Sweep speed at 1 microseconds per division

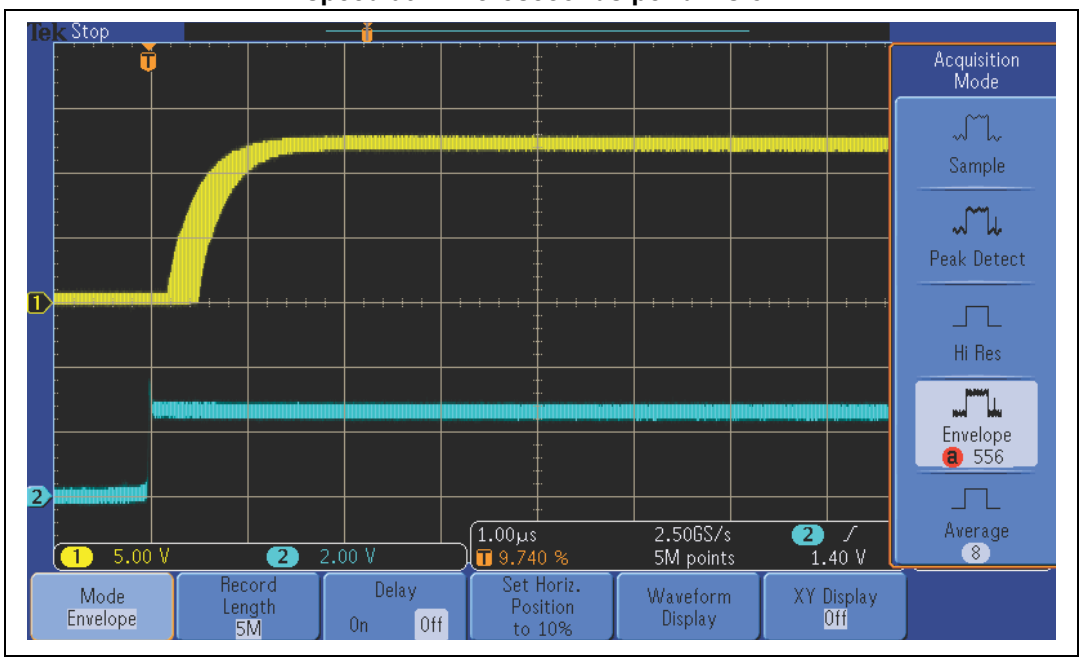


Figure 4. Channel 1 (yellow) LS1 gate drive output at 5 volts per division (zero at center) Channel 2 (blue) PA0 input at 2 volts per division (zero at -3 divisions) Sweep speed at 1 microseconds per division

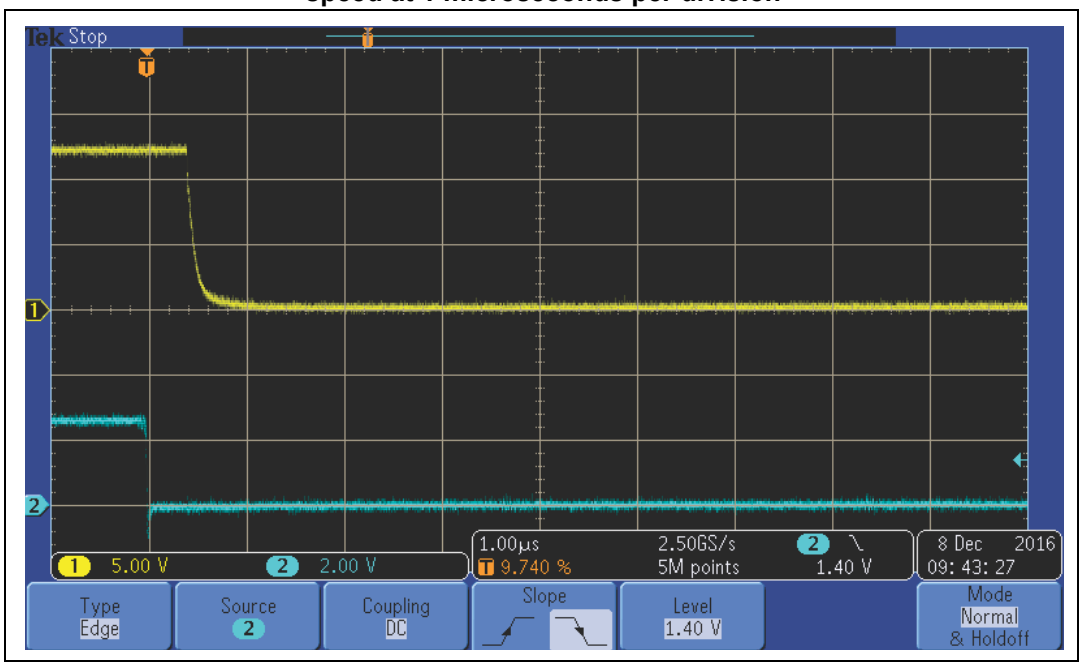
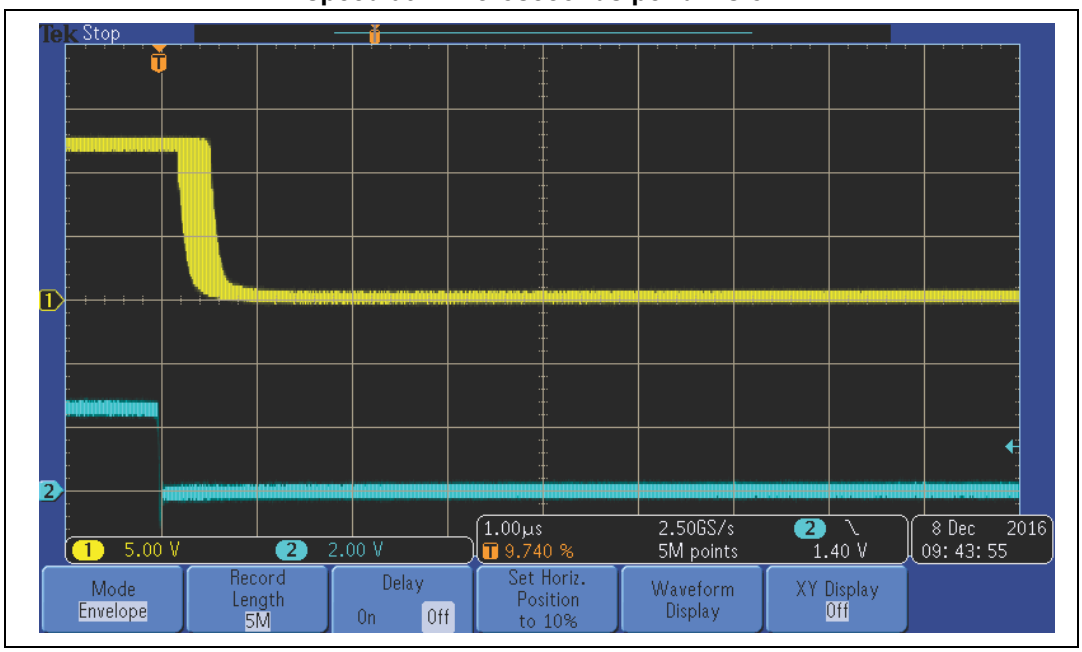


Figure 5. Channel 1 (yellow) LS1 gate drive output at 5 volts per division (zero at center) Channel 2 (blue) PA0 input at 2 volts per division (zero at -3 divisions) Sweep speed at 1 microseconds per division



Appendix A the real tables

Now let's present the real lookup tables. The mapping of input pins to gate drives is as follows:

PA0 --> PB13 (LS1)

PA1 --> PB14 (LS2)

PA2 --> PB15 (LS3)

PA3 --> PA8 (HS1)

PA4 --> PA9 (HS2)

PA5 --> PA10 (HS3)

In the following code #define definitions, a symbol of the form bx means a 16 bit value where all of the bits are zero except bit x. The expression b10+b9+b8, for example, is then the 16 bit value where just bits 10, 9, and 8 are ones and the rest zeros. We also use positive logic (PA0 high for LS1 to go high).

```
*/

#define hs1     b8
#define hs2     b9
#define hs21    b9+b8
#define hs3     b10
#define hs31    b10+b8
#define hs32    b10+b9
#define hs321   b10+b9+b8
const unsigned short tablea[256] = {
0,0,0,0,0,0,0,0,hs1,hs1,hs1,hs1,hs1,hs1,hs1,hs1,
hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs21,hs21,hs21,hs21,hs21,hs21
,hs21,hs21,
hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs31,hs31,hs31,hs31,hs31,hs31
,hs31,hs31,
hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs321,hs321,hs321,hs3
21,hs321,hs321,hs321,hs321,
0,0,0,0,0,0,0,0,hs1,hs1,hs1,hs1,hs1,hs1,hs1,hs1,
hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs21,hs21,hs21,hs21,hs21,hs21
,hs21,hs21,
hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs31,hs31,hs31,hs31,hs31,hs31
,hs31,hs31,
hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs321,hs321,hs321,hs3
21,hs321,hs321,hs321,hs321,
0,0,0,0,0,0,0,0,hs1,hs1,hs1,hs1,hs1,hs1,hs1,hs1,
```

```
hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs21,hs21,hs21,hs21,hs21,hs21
,hs21,hs21,
hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs31,hs31,hs31,hs31,hs31,hs31
,hs31,hs31,
hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs321,hs321,hs321,hs3
21,hs321,hs321,hs321,hs321,

0,0,0,0,0,0,0,0,hs1,hs1,hs1,hs1,hs1,hs1,hs1,hs1,
hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs2,hs21,hs21,hs21,hs21,hs21,hs21
,hs21,hs21,
hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs3,hs31,hs31,hs31,hs31,hs31,hs31
,hs31,hs31,
hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs32,hs321,hs321,hs321,hs3
21,hs321,hs321,hs321,hs321,
};
#define ls1     b13
#define ls2     b14
#define ls21    b14+b13
#define ls3     b15
#define ls31    b15+b13
#define ls32    b15+b14
#define ls321   b15+b14+b13
const unsigned short tableb[256] = {
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,

0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
```

```
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,

0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
0,ls1,ls2,ls21,ls3,ls31,ls32,ls321,0,ls1,ls2,ls21,ls3,ls31,ls
32,ls321,
};
```

Revision history

Table 1. Document revision history

Date	Revision	Changes
06-Mar-2019	1	Initial release.