BL233C New Features

Contents

1 Introduction

BL233C enhances BL233B, while remaining a drop in replacement in most cases. There are many small improvements, but the additional features are:

- Quoted strings for text.
- Easy reading and writing of I2C EEProm
- Stack Data processing
- Sleep Mode
- “;” (printable) resume character

- W2812B Serial LED strips (Adafruit NeoPixel etc)

If you do not send the new command characters, then all existing commands behave as they did before. Default EEProm contents are different.

2 Sleep

Sleep mode is disabled by default, until the eeprom flag fSeral_ is set. To set it send “F708 61”. The change happens after next reset (or send “X5A”)

Sleep is useful for:

- Power-down sleep for battery powered applications e.g. WebI2C Router control
- Power-down sleep to wait for RTC clock or other external interrupt
- Wait-for-string sleep for multidrop operation
- Wait-for-string sleep to ignore Linux boot messages when connected to console/ttyS0

Sleep is entered with command “Z5A <Sleep Time>”. As the BL233C will be stopping its uart, you must use the pause sequence to ensure that all commands, including the post wakeup ones, are inside the BL233 before it executes sleep.

```
T"Going to sleep for 1hour" :Z5A 061D T"Awake Now!";
```

2.1 Powerdown Sleep “Z”

Sleep time is N+1 * 2.3sec. BL233C stops its oscillator, empties the TX buffer, and enters sleep. It wakes on

- Timeout
- INT pin is pulled low
- Change on RTS pin
- Change on Port 3 pins SDA3,SCL3, CS3/P5

Commands are ignored during sleep. You cannot wake it from sleep via the serial RX pin. However changing the RTS pin state will wake it.

Power consumption is only a few microamps.

2.2 Wait for ID-String Sleep

The BL233C stalls (at normal power), waiting for a string that matches the stack contents. This can be used to

- Multidrop BL233C
- Ignore boot messages from Linux when connected to TTYS0 (linux console) on routers and similar devices.

Power consumption is normal - BL233C is running.

3 Easy I2C EEProm Command “K”

BL233C can manage the process of reading and writing I2C EEProms, handling the data pointer and the paged writes for you. This turns the reading and writing into simple sequential reads and writes. The data pointer is persistent, making it possible to resume reads and writes, and (with the stack commands) to move data from an I2C device directly into an I2C memory.

I2C EEProms have up to 20 bits of data pointer. They use a one or two byte pointer, and up to 3 bits mixed into the lower bits of the address byte. When writing they have write pages from 1 to 256bytes long. The Control Byte configures K command for 1 or 2 address bytes and the write page size.

The K command sets up the pointers. R & W commands read and write the eeprom. The comand is: K [I2C base address + Hi 3 Pointer Bits] [Pointer Byte M] [Pointer byte L] [Control Byte]

I2C address defaults to 0xA0. The merging of data pointer bits into the address is taken care of for you.

```
K R02          ~ resume reading from previous data pointer
K 02 R04       ~ ControlByte=02, Default Data Pointer to 0, use default address c
K 40 02 RFF ~ CB=02, DP=0x40, read 256bytes
K 21 40 0D RFE ~CB=0D, DP= 0x02140, read 512 bytes
K 03 21 40 0D RFD ~CC=0D, DP= 0x32140, read 1024 bytes
```

3.1 Read Command

The number of bytes to read is modified from the normal read command. From 1 to 0xF3 it is simply the number of bytes to read, as normal. From FF to F4 it goes in binary steps: FF=256, FE=512, ... F4=512k.

Thus the whole memory can be read with a single command.

Note: Reading large memories will take some time. You may need to change baud rate.

3.2 Write Command

Before performing long write operations you should check that the memory is present.

All paging and data pointers are taken care of, you can write the whole memory in a single operation if you wish.

W [data bytes] stoP

Writing is executed when a write page boundary is crossed, or a *stop* command is sent. The eeprom goes offline while writing, and the BL233 loops addressing it until it returns, or times out.

4 Stack Operations

The BL233C adds a small data stack, and a set of 16bit maths primitives. Those familiar with RPN calculators will be familar with the idea (if not - ask your dad). Using this you can:

- Swap byte order (Big endian / little endian)
- Convert results to Decimal strings, with (-ve) and decimal points.
- dump unwanted data bytes that must be read as a block.
- Scale ADC values to human readable units e.g. 0x0400 -> 10.23
- Average multiple readings to reduce noise / improve resolution.
- Pass data directly from one I2C device to another e.g sensor to display, or sensor to I2C-eeprom

4.1 Stack Commands

BL233C has a 6 byte data stack. and a 2 byte store.

	Name	Function
!		
(	stkPush	Push byte onto stack (from serial)
#	PrintHex	Pull byte off stack, and print as hex
=	PrintAscii	Pull char/byte off stack, and print as Ascii/binary
@		R/W src/dest is Stack not serial port
)	Pop/Roll	Pop 1 byte off stack (remove it) X Y Z -> Y Z X
’	Roll4	Roll top 4 elements of stack X Y Z -> Y X Z
/	Swap	Swap top 2 bytes Xlo,Xhi -> Xhi Xlo
’ ’	SwapD	Roll4 twice swaps words
+	Add	Add X+Y -> X
-	Neg	Negate X -> -X
-+	Sub	Subtract Y-X ->X
*	Mul	Multiply+Shift
&	And	And
$		Bin2Dec. Convert 1 digit to ASCII
%		Abs / Sign., leave Sign as Ascii ’-’ or ’ ’ on stack
[	STO	Store 2 bytes, leaves on stack
]	RCL	RCL 2 bytes, push on stack
\	ShowStack	Prints whole stack - for debugging use

4.2 Data to-and-from the Stack

Data is pushed onto the stack from the serial by (command

When you want to use the stack with Read and Write commands, use @. Each @ is one byte, eg R@@ reads two bytes *onto* the stack. W@@ pulls the two data bytes *from* the stack. Beware that bytes reverse when push/pulled. If you want them in the same order, use Swap /. For example to read 2 bytes from on address and write to another, in the same order....

```
S41 @@ ~read 2 bytes onto stack
/      ~swap bytes
S42 @@ ~write back out from stack
```

Note that the 16bit maths operations are performed little-endian on the stack. This means you push the data in, in the obvious big-endian way. When you read a 16bit number you need to swap bytes before reading using /

The ShowStack \ (backslash) command, prints the stack leaving it unchanged. It is only for debugging purposes.

4.3 Swap Endian

Little Endian data (Least Significant Byte first) is a pain for to read for humans, and programs like Excel. To swap 2 byte data from big endian to little endian, you only have to push it on the stack (@), then pull it off (#). The endianness reverses.

```
S41R@@ ##
```

e.g. The Dallas DS1820 temperature sensor returns data Little Endian. Ignoring the setup, the data is read onto the stack by R@@. We can swap the bytes. Now it is readable.

```
SWCC44L0400 ~initialise sensor
SWCCBE R@@  ~read 2 bytes (1 word) onto stack
##                ~return 2 bytes from stack as hex
```

The first part initialises the chip, the second reads the data

4.4 Averaging Data

A common need is to average sensor data to reduce noise, or to increase resolution.

```
SWCC44L0400 ~initialise sensor
SWCCBE R@@/ ~{Read first temp onto stack. Swap bytes as DS18B20 data is little-
SWCCBE R@@/ ~{read 2nd temp
+ ))    ~add together and pop 2 bytes off stack. Now sum is at top of stack
SWCCBE R@@/ +))  ~and keep doing this as many times as you like....
/##     ~swap and print as hex
```

So we can take two readings and add them. Note that we Swap the data bytes after each read, as DS1820 was little-endian.

If you want to increase resolution, you *need* noise in the readings. If the readings are always the same, then the avarage will be unchanged. Noise is reduced by the square root of the number of readings. To add 1 bit of resolution, average 4 readings, 2 bits - 16 readings and so on. Obviously that makes it clear that averaging 4 readings is the minimum to be useful.

4.4.1 Running Average

If your sensor has electrical noise, then doing an average at once (above) is simple. However if you want to smooth the data over longer times, then a running average might be better.

```
SWCC44L0400 ~initialise sensor
SWCCBER@@/ []+]+]+]+]+]+]+ [  ~initialise running average total
] 2000 00 *   ~divide  by 8
- ] +   ~subtract so we have 7/8th of running total
SWCCBER@@/ + [  ~add new value in to running total and store
] 2000 00 *   ~divide  by 8
/##              ~return the result
```

4.5 Scaling Data

Most sensors return integer ADC values that are not meaningful. In the general linear case Value = ADC * m + c.

The DS18B20 returns a value of 0x7D0 (2000) for 125 deg C, and 0 at zero. We can scale it up to 1250 i.e. 10ths of a degree.

```
SWCC44L0400
SWCCBER@@/
A000 00 *
/##
```

Each count is 1/16th degree. So to get degrees multiply by 1/16 * 65536. But we want to get 10ths of a degree, so mutliply it by 10/16 * 65536 = 40960 (A000)

To scale to 100ths of a degree, 100/16*65536 = 409600. But this is too big for 16 bits. But 409600 = 51200 * 8.

4.5.1 Multiply/Shift

Integer multiply is a 16 x 16 = 32 process, where the bottom 16 bits of the result is dropped. To keep data in range, the multiply is followed by a shift left of 0-16 bits

X Y Numshifts

$X = X * \frac{Y}{65536} * 2^{NumShifts}$

4.6 Decimal Conversion

Decimal conversion is done using a primitive that converts a single digit at a time. By having a single digit primitive, you can format the data as you want - convert more or less digits, insert decimal points and so on. The conversion returns an ascii character on the stack, you can print, or write to a display device.

A macro to print 5 digit Bin2Dec is in BL233C eeprom. see below

For signed integers, use the Abs/Sign function first. This makes the data +ve, and puts Ascii '-' or ' ' on stack.

If you want to insert a decimal point, push 2E between digits.

```
(07D0 %= (D8F0 ~$= FC18 $= FF9C $= 2E= FFF6 $= FFFF $=

(07D0  ~push a sample value, or read ADC onto stack
%=  ~ show sign, make -ve values +ve, print ascii value from stack
(D8F0 ~$= ~ push -10k onto stack and convert first digit and print
FC18 $= ~thousands
FF9C $= ~hundreds
2E=     ~insert a decimal point ascii 2E
FFF6 $= ~tens
FFFF $= ~ones
```

$ subtracts this digit. For each digit, the numbers are D8F0=-10000, FC18= -1000, FF9C= -100 etc.

% returns the sign. = pulls the digit character off the stack.

You can just convert that part of the number you want. For example if you have a temperature in 10th of a degree, where 650=65C, you might just convert the hundreds and tens, to get just the two digit number 65.

4.6.1 Decimal Conversion Macro

This macro to print signed and unsigned numbers as decimal is in macro memory at address x15 for signed, x17 unsigned:

```
%=(D8F0$=FC18$=FF9C$=FFF6$=   FFFF$=<
```

To use it:

```
(8004   ~Push data
[       ~STO a copy
>15     ~print signed
]       ~RCL
>17     ~print unsigned
]
>1E
```

You can insert decimal points after the =. There is even a space for one before the last digit.

```
V30CF"2E="  ~write 2E= into the gap in the macro to get a decimal point
```

You can also call just the last part when you only want a couple of digits

5 W2812B

Support for the single wire, coloured led driver W2812B is added. Please contact us if this is useful to you - this feature may not make it into final versions of BL233C.

A single channel is on Bus B Due to the timing requirements of W2812B, only ?13? LEDs (39bytes) can be driven.

```
GB  ~select W2812 bus
SW
81 82 83 ~3bytes per led
74 85 86 ~ up to ?16? leds/?48bytes?
P               ~data is written when P is sent
```

Like OneWire, the timing is dependent on clock. Note that the timing follows

6 Revision History

Rev	Date	Changes
0	15 Aug 16	First Release
1		
2		
3		