GreenArrays®

Application Note AN008 *Revised 5/10/12*

Exploring a 3-axis Accelerometer

In this document Peter Milford, its contributor, explores a new sensor using high level polyFORTH® on the Host chip of an EVB001 evaluation board. This illustrates one way to rapidly prototype such an interface while validating the understanding gained from the manufacturer's documentation, and serves to make the point that familiar, interactive Forth methods are applicable to our chips.

The text assumes you have familiarized yourself with our hardware and software technology by reading our other documents on those topics. The current editions of all GreenArrays documents, including this one, may be found on our website at http://www.greenarraychips.com . It is always advisable to ensure that you are using the latest documents before starting work

The work described herein is that of Peter Milford.

Contents

1. Introduction

This application note introduces using high level polyFORTH running on a GA144 eval board (EVB-001) to access input output ports on nodes with IO facilities. The nodes with IO are nodes positioned at the edges of the processor array. Some of these nodes have digital IO pins, some have analog IO pins and some have special purpose IO pins.

Several examples are presented, working up to interfacing to the CMA3000 3 axis accelerometer over an SPI bus. No low level coding is needed. The 'ease' of access of the remote nodes across the chip comes from using a well-developed communication library: 'Snorkel' based on 'Ganglia' and the interface words to polyFORTH named R!, R@ and the path defining word ND.

The code does not include error checking that might be included in a real application. This note demonstrates how to quickly use polyFORTH to build and test a hardware driver from the bottom up, discovering how the hardware behaves in the process. polyFORTH is a high level application running on the GA144 and provides an interactive application development environment. It was developed originally by FORTH Inc. for a variety of processor architectures. See [2] for details. In this document source code is presented in block form with shadow screens on the left.

polyFORTH on the GA144 evaluation board is controlled via a USB port that mixes terminal IO with DISKIO. The current implementation of the driver software runs on a Windows PC and is coded in X86 polyFORTH. arrayFORTH is another development environment for the GA144, documented in [1].

polyFORTH on the GA144 is a full featured 16 bit FORTH. The development board can boot polyFORTH from a serial flash. polyFORTH uses nine F18 nodes, including a static RAM interface, plus additional nodes for the serial interface. The polyFORTH virtual machine uses four F18 nodes. Some of the nodes have names, such as BITSY and STACK. Additional nodes are used to support external static RAM and remote node access. Several nodes are available to extend the virtual machine with F18 instructions. Instructions, data stack, return stack and variables are all stored in external SRAM. The instructions consist of tokens that are decoded into a node ID (1 of 6) and an address of F18 code to execute within the node. The stacks and dictionary can be 'large' in size, limited by RAM size only.

Item	polyFORTH	arrayFORTH	Comments
Word width	16 bits	18 bits	
Stack length	Large	10 elements	
Return Stack length	Large	9 elements	
Approx time to duplicate Top of stack	RAM read instruction Decode & run instruction RAM write stack element ~ 2 SRAM read access times.	~ 1 instruction time	The SRAM mounted on the development board is specified to be a 55ns access time SRAM. The current code accesses for this are slower. So two accesses take about 500 nsec. 1 instruction time in an F18 is ~ 1.4nsec. possibly overlapped with a 5 nsec. next instruction word fetch.
Approximate easy program size 'limits' (arbitrary but indicative)	~10k FORTH words	~1k FORTH words. (A few nodes of instructions)	Using several nodes, but not entire chip.

Table 1 polyFORTH arrayFORTH comparison

Note that polyFORTH is an example of a large arrayFORTH application. Time critical sections of code could be ported to arrayFORTH/F18 machine code to execute.

2. Port Access

	302
Create named paths to io nodes	(SPI PIN ROUTING ASSUMING START AT N107)
MISO master in slave out at node 517	ND MISO 3 uu 8 rr 0 ,path (N517)
MOSI master out slave in at node 417	ND MOSI 2 uu 8 rr 0 ,path
INT interrupt at node 317	ND INT 1 uu 8 rr 0 ,path
CLK clock at node 217	ND CLK 8 rr 0 0 ,path
CS chip select at node 117 using the DAC	ND CS 4 rr 1 dd 4 rr ,path
DAC values xored with x155, CSR is high CSA is low	HEX 01FF 155 XOR CONSTANT CSR 155 CONSTANT CSA
	(Initial setup of ports for SPI)
	: SPII
To initialize set INT low, CLK and CS high	INT 00 0 IO R!
	CLK 00 3 IO R!
	CS CSR 0 IO R! ;

The code running the polyFORTH virtual machine is not directly running in an F18 node with input output pins. In order to access the F18 processor nodes that have IO pins, polyFORTH has a high level mechanism available that uses resources in the intermediate nodes to access other nodes. This system is called 'SNORKEL' and is supported by routines running in the SRAM interface nodes.

2.1 Set up Node Access

The GA144 is configured as an array of 18 by 8 processors (144 nodes) in a regular grid. In general, nodes can communicate with the 4 adjacent nodes directly. For code running in one node to access resources in another node, messages must be passed from the initial node to/from the destination node. The F18 processor uses an elegant method of 'port' execution to enable much of this access.

2.1.1 To use SNORKEL system

ND defines a node and its relative path to the start of the 'snorkel' node, N108. The Snorkel routine creates a message passing path through a series of nodes to deliver one or more messages to the node at the end of the path. The first step is always a single step up to N208.

Use the primitives **ND** (a defining word) and several path defining words: **uu***,* **dd***,* **ll** and **rr**.

(n) uu	(number of nodes up relative to current node)
(n) dd	(number of nodes down relative to current node)
(n) ll	(number of nodes left relative to current node)
(n) rr	(number of nodes right relative to current node)

The path words are used to build an 18 bit message, with 3 six bit fields. Each field has a 2 bit direction code: up down left or right. Note that right is 00 so a path of 0 right cannot be encoded this way. Each field also has a 4 bit count.

The word **,path** takes the path and stores it on the end of the current definition.

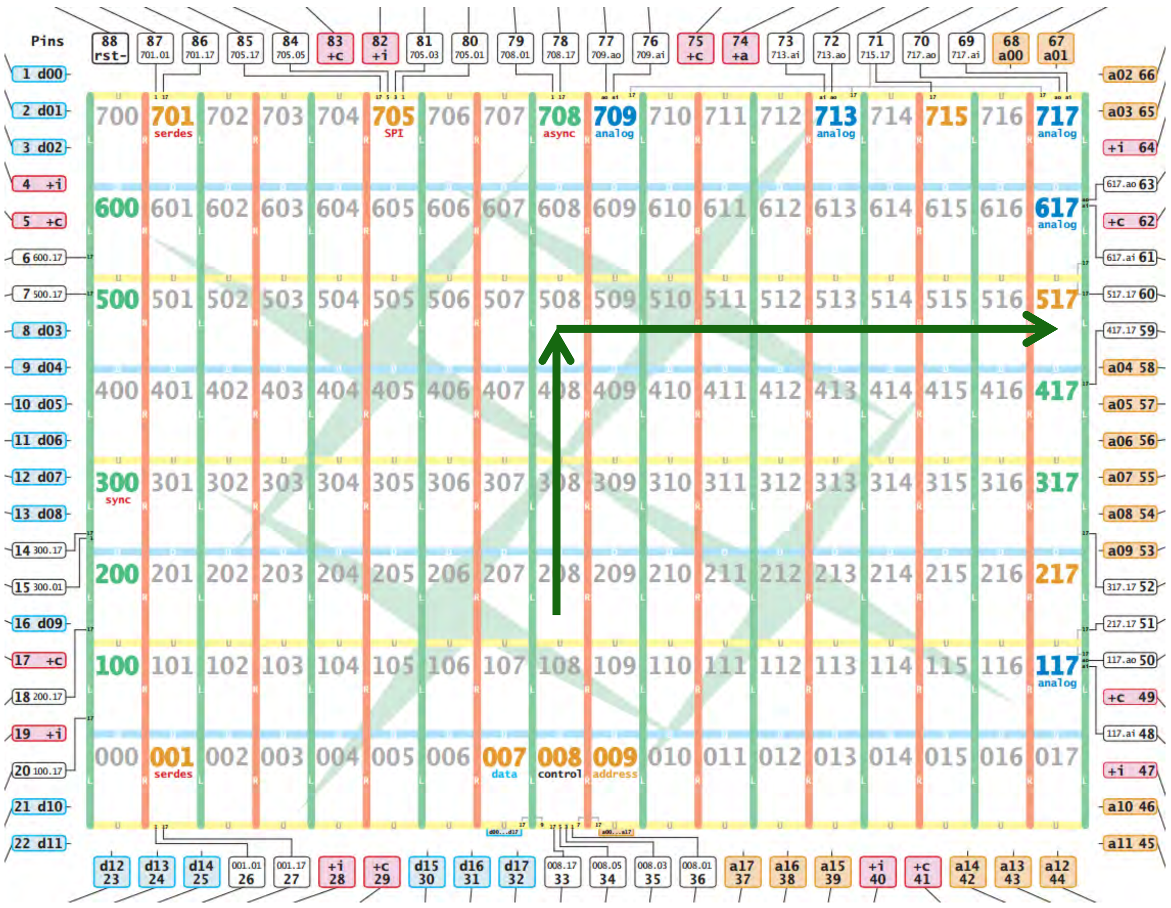
For example:

```
ND N517   3 uu   8 rr   0 ,path
```

Note the path has 3 items:

3 uu	3 nodes up.
8 rr	8 nodes to the right
0	No op.

When the word N517 is executed, it will setup a path through intermediate nodes in the given directions so messages can be sent to node 517.



3. Read and Write Remote Node Memory

`Give the address of the IO register the name IO`	`301`
	`( SPI INITIALIZATION)`
	`HEX 15D CONSTANT IO`

There are two main polyFORTH words for reading or writing a memory location in a remote node. Both require a path to be already setup using a path defined by an ND word then enabled by running the defined path word.

- **`( a - d) R@`** - fetch the data at address a over the current path.
- **`( d a) R!`** – store the double (18 bit) data at address a over the current path.

Note, d is a double number on the stack, high 2 bits on TOS, low 16 bits below. For example 0 3 represents an 18 bit word with the most significant 2 bits set to 1. [It is a double polyFORTH word as polyFORTH is based on 16 bit words and the F18 is 18 bit words natively]

Frequently, accesses are to the IO register on a node, its address is the same on all nodes, defined as constant: x15d.

So to read the IO register on node 517: **`N517 IO R@`**
This returns an 18 bit word packed in the top two stack entries, bits 16&17 in the low order bits of the top of stack, bits 0-15 in the next word on the stack.

Now would be a good time to connect the GPIOpin of node 517 to your oscilloscope so you can see what’s happening.

Note: Node 517 is a single bit GPIO node. Other nodes have for example analog input, analog output, or parallel i/o. The definition of bits in the IO register differs for each type of node.

3.1 Toggle GPIO

To setup the digital GPIO on node 517 for output and make it low.

For each IO pin there are two control bits. The high order bit determines whether or not the pin is an output or an input. If the high bit is 0, for input, then the low order bit determines whether the pin is tri-state or weakly pulled down. When the high bit is 1, for output, then the low bit determines whether the pin is driven high or low. The actual state of the pin can be read in any case.

As an 18 bit value, store x30000 in the IO register to output a 1 and x20000 to output a 0.

Bit 17 in input mode is connected to the pin for reads.

To output a low on pin 517.17, set bits 16&17 of the IO register (high 2 bits) to: 17:1 16:0.

`ND N517  3 uu 8 rr 0 ,path` (Define the path from current node (108) to edge node)

To setup the path execute the newly defined word,

`N517` (then...)

`0 3 IO R!` (This would take pin 517.17 high)

`0 2 IO R!` (This would take pin 517.17 low)

3.2 Send a bit to current node GPIO

Define a word to send the low order bit of a word .The word TBIT is sending information across several GA144 nodes, using the path setup in the definition of N517.

```
: TBIT ( n)  1 AND  2 OR  0 SWAP  IO R! ;
```

Set the path and send a 1:
N517 1 TBIT

And now a 0
0 TBIT

3.3 Send a byte to a port

Send byte LSB first

```
: TBYTE ( n)  7 FOR DUP TBIT 2/ NEXT DROP ;
```

```
N517 HEX AA TBYTE
```

3.4 Sample Use of PORT Out

To turn on/off an LED. Connect an LED between 1.8V and Node 517.17 GPIO pin. (Pin 60 on 88 pin package). More conveniently connect to J21, Pin 6 on the eval board.There is also a VDD pin available (V1P8) on a nearby pin/hole.

Turn on LED
```
N517 0 TBIT
```

Now turn it off:
N517 1 TBIT

Note: If nothing has changed the routing in the intermediate nodes, then the second call above to N517 is redundant.

To send alternating high and low in an infinite loop:

```
: OSC  N517 BEGIN  1 TBIT  0 TBIT AGAIN ;
```

```
OSC
```

If the LED is still attached, how fast will it flash? How would you insert a delay to slow down the flashing?

```
Test with an oscilloscope
TBIT( n) changes low bit to a value for IO
TBYTE ( b) sends 8 bits LSB first

OSCTEST sends a pattern to the oscilloscope
```

```
303
( TEST CODE)
: TBIT 1 AND 2 OR 0 SWAP IO R! ; ( SEND ONE BIT OUT
CURRENT PATH)
: TBYTE 7 FOR DUP TBIT 2/ NEXT DROP ;
HEX
: OSCTEST N517 BEGIN 0AA TBYTE ?KEY UNTIL ;
```

`TBIT` is defined in block 303 to copy the low bit in the word on the stack to an io pin. Type `N517 1 TBIT` and then `0 TBIT` and watch the scope. You should see the pin go high and then low. The word TBIT is sending information across several GA144 nodes, using the path setup in the definition of N517. Send a byte LSB first using TBYTE from block 303. This is more interesting if you set up a loop so the image will persist on the oscilloscope. You can check timing and bit patterns this way. The word `OSCTEST` sets up such a loop. Run OSCTEST at the command line and look at the pattern produced on the scope. Can you see the number HEX AA in the pattern? Now connect an LED between 1.8V and Node 217.17 GPIO pin (Pin 60 on 88 pin package). More conveniently connect to J21, Pin 3 on the eval board. There is also a VDD pin available (V1P8) on a nearby pin/hole. Turn on the LED via `N517 0 TBIT` then turn it off via `1 TBIT`. Note: If nothing has changed the routing in the intermediate nodes, then a second call to N517 would be redundant. To send alternating high and low in an infinite loop run `OSCTEST` from the command line.

4. Accessing an SPI Device

Using the same basic primitives, a program to access a device with an SPI interface will be developed. The sample device is a 3 axis accelerometer made by VTI, a CMA3000. See for example [4] for details of using and programming the device. The CMA3000 has an SPI interface and operates at 1.8V. On the CMA3000 all transfers are bi-directional 16 bit transfers. (Some SPI devices use 8 bit transfers, some unidirectional.)

The goal is to initialize the operating mode of the CMA3000 then read the x, y and z accelerations. The CMA3000 measures acceleration and reports an 8 bit signed value for each axis. The values are read by reading a particular register for each axis.

SPI signal on CMA3000 dev board	Pin on dev board	CMA3000 Fn	Ga144 node	Reg bits	Pin	Header location on GA144 dev board.
CS	13	In active Lo	117	AO 0-8	50 (AO)	J21.7 Note AO is current source: needs resistor to ground ~1k Ohms.
MISO	12	Out	517	16-17	60	J21.3
MOSI	11	In	417	16-17	59	J21.4
CLK	10	In active Hi	217	16-17	51	J21.6
INT	3	Out	317	16-17	52	J21.5
GND	1, 9, 15					J21B.5
AVDD	14					J21B.1
DVIO	8					Connected on board header. Should be isolated from AVDD

Table 2 CMA3000 SPI interface connections

4.1 Paths to Nodes

To access the GPIO pins on the various processor nodes attached to pins, remember the paths defined in block 301 above. The path stays the same until changed and could be accessed several times without rerunning the path word. This SPI application however is accessing a number of different pins on different nodes and rarely uses the same path twice in a row. Refer to block 304 for the next bit of testing as we build up an SPI interface.

```
1BITO ( n) use MOSI path to set one bit of output
data
1BITI ( - 0|1) use MISO path to read one bit of
input data
CSL take chip select low  CSH take chip select
high

CLKL take clock low  CLKH take clock high

1BYTELSB ( b) send a byte LSB first
1BYTEMSB ( b) send a byte MSB first

1BYTEIN ( - b) read a byte, MSB first

OSC2 send another pattern for the scope
OSC3 receive a pattern for the scope
```

```
304
( PSEUDO CLOCKED OUTPUT SAMPLE)
: 1BITO MOSI 1 AND 2 OR 0 SWAP IO R! ; ( SETUP PORT
WITH DATA)
: 1BITI MISO  IO R@ SWAP DROP 2/ ;
: CSL CS CSA 0 IO R! ;  : CSH CS CSR 0 IO R! ;
: 1CLOCK CLK 0 3 IO R! 0 2 IO R! ; ( PULSE A CLOCK)
: CLKL CLK 0 2 IO R! ; : CLKH CLK 0 3 IO R! ;

: 1BYTELSB 7 FOR DUP 1BITO 1CLOCK 2/ NEXT DROP ;
: 1BYTEMSB 7 FOR DUP 128 AND  IF 1 ELSE 0 THEN
1BITO 1CLOCK
  2* NEXT DROP ;
: 1BYTEI 0 7 FOR 2* CLKH 1BITI CLKL OR NEXT ;
HEX
: OSC2 BEGIN 0F2 1BYTEMSB ?KEY UNTIL ;
: OSC3 BEGIN 1BYTEI  0FF =    ?KEY OR UNTIL ;
```

4.2 One Bit In and Out

Block 304 contains some of the primitives that will be used to clock bits out through the MOSI pin and in through the MISO pin. The word `1BITO` sends a one or a zero to MOSI. The word `1BITI` reads the IO register in the MISO node and translates that value into a 1 or a 0 for the state of pin 517.17. Other words are expected to enable chip select and manipulate bits read and written.

4.3 Chip Select

As chip select is assigned here to an analog out pin, the signal is using the DAC to set the output levels. Chip select is active low. To set the output levels on the 9 bit DAC, the values to program are xor'd with x155. So for low output, xor 0 and x155 to get x155, and for high output xor x1ff and x155 to get xaa for high. For readability, these values were created as the constants CSR and CSA in block 302.

To enable the CMA3000 to be read or written and start an access cycle, chip select must be enabled. It is active low. So to enable access to the chip execute `CSL` and to disable execute `CSH`. Normally the CMA3000 is selected during 16 bit transfers of data and disabled otherwise. Type CSH and then CSL at the command line. If the scope probe is connected to the chip select pin you will see it go high then low.

4.3.1 Digital to Analog converter (DAC) Note

Note that chip select is attached to a digital to analog converter (DAC) pin. See for example [6] for discussion of using the DAC. Note that an external resistor of ~1kOhms is connected from the DAC to ground, as the DAC is a current source. We are using the DAC to output digital values, but similar code could be used to output analog values. We have connected the DAC pin as a digital out for two reasons: To demonstrate the DAC usage as a digital out and to reserve the other digital GPIO pins for other uses.

4.4 Clock Words

To set the clock line high or low use the words `CLKH` and `CLKL` which set a path to node 217 where the clock pin resides and takes pin 217.17 high or low. MOSI bit is modified during clock low and MISO is read on clock high. Connect a scope probe to the clock pin and type CLKH then CLKL. You should see the pin go high and then low.

4.5 Another Test Pattern

Block 304 has a few test words for purpose of checking patterns on the oscilloscope. The word `OSC2` sends a pattern of bits driving both the clock pin and the MOSI pin. Connect the scope probes to those two pins and run OSC2. The phrase `0F2 1BYTEMSB` is executed in a loop, which writes a bit and toggles the clock eight times for a byte.

4.6 Single Bit Transfer.

Now that we're satisfied that we can manipulate the right bits it's time to make the driver for an SPI interface. Each bit consists of a read and a write operation as data are clocked out and in. The transfers on ports MISO (master in slave out) and MOSI (Master out slave in) using `1BITO` and `1BITI` from block 304 were tested above. They are combined in the word `ONEBIT` defined in block 305. In this word a bit is written to MOSI when clock is low and read from MISO when clock is high.

```
ONEBIT ( n – 0|1) writes low bit of n then reads a
bit

WORDIO ( n1- n2) clock out MSB and shift in bit
read to LSB


OSC4 another test pattern for the scope

(RREG) ( a – n) read back contents of register at
address a
WREG ( n a) write into register at address a
RREG ( a – n) mask off 16 bits (not needed as this
is a 16 bit Forth)
```

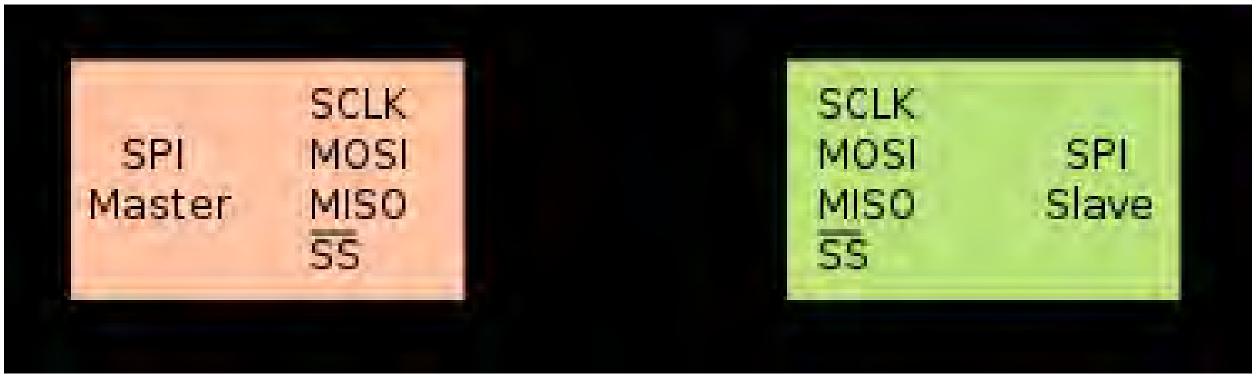
```
305
( VTI CMA3000 3 AXIS ACCELEROMETER SPI. 16 BITS R/W
AT  )
: ONEBIT 1BITO CLKH 1BITI CLKL ;
HEX
: WORDIO CSL 0 0F FOR 2* OVER  8000 AND IF 1 ELSE 0
THEN
       ONEBIT OR SWAP 2* SWAP NEXT SWAP DROP CSH ;

: OSC4 BEGIN 0AA01 WORDIO  0FFFF = ?KEY OR UNTIL ;

: (RREG) 2* 2*  ><  WORDIO ; : WREG 2* 2* 2 OR >< OR
WORDIO ;
: RREG (RREG) 0FF AND ;
```

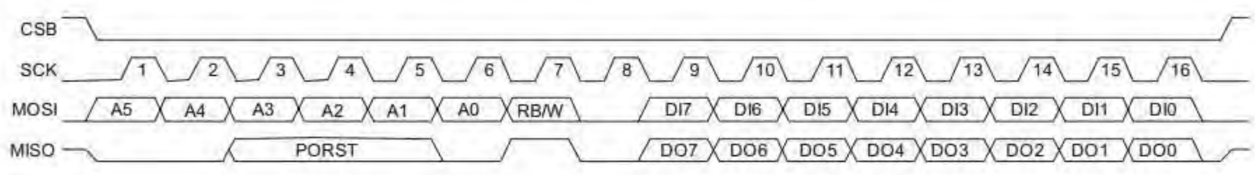
4.7 Word Transfer

Each transaction for the CMA3000 sends 16 bits to the chip and reads back 16 bits. `WORDIO` takes a 16 bit word as input, clocks it out 1 bit at a time, shifting in 1 bit of input on each clock till all 16 bits of input and output are processed. The order required for the CMA3000 is most significant bit first to be output. WORDIO is the most complex word developed here, it is manipulating two 16 bit numbers and shifting them both… can you find a way to improve it?

4.7.1 Note on Timing:

The CMA3000 has a number of timing specifications. This app note does not correctly address them. The app note assumes that the code is executing slowly enough to meet all the timing requirements of the SPI interface on the CMA3000. Experience with the oscilloscope indicates that this is true. Check for yourself by running `OSC4` and measuring clock pulses on the scope.





5. CMA3000

Now we're almost ready to read and display acceleration data from a CMA3000 chip. It's just a matter of assembling the primitives we've already coded and tested.

5.1 Register Access

Using the CMA3000 requires accessing on-chip registers over the SPI bus. Each transaction consists of 16 clock transitions, consisting of an 8 bit command, 8 bits of data transfer, reference the timing diagram. For convenience, create read register and write register words that convert register numbers to packed register access commands then call WORDIO to carry out the transaction. (These are CMA3000 specific register commands [4]). Note: `><` is byteswap and is used in `(RREG)` to position the bits as required by the CMA3000 chip. `WREG` and `RREG` are defined in terms of WORDIO in block 305.

5.2 Initialization

	306
Give names to some CMA3000 registers	(CMA3000 COMMANDS) 6 CONSTANT DOUTX 7 CONSTANT DOUTY 8 CONSTANT DOUTZ 2 CONSTANT CTRL
INITCMA3000 assemble bits and write into control register	: INITCMA3000 1 6 OR 128 OR CTRL WREG DROP ; (2G 40HZ NO INT)
SEXT (ub – n) sign extend a byte into a 16 bit word	: SEXT (ub--n) >< 2/ 2/ 2/ 2/ 2/ 2/ 2/ 2/ ;
RXYZ (- z y x) read 3 axes onto stack	: RXYZ (---nnn) DOUTZ RREG SEXT DOUTY RREG SEXT DOUTX RREG SEXT ;
TEST read and display acceleration data for x y z	: TEST BEGIN RXYZ . . . CR ?KEY UNTIL ;

The control register is set to no ints (128), 100 Hz (6), 2g mode (1). For this example we will not be using the INT pin, though we have reserved it and created a path to it. INITCMA3000 writes bits in the control register of the CMA3000 to set its initial state.

5.3 Read xyz Accelerations.

The acceleration results are stored in CMA3000 registers x, y & z. Let's name them along with the control register as constants in block 306. The CMA3000 returns 8 bit signed quantities for the acceleration. These are sign extended to form 16 bit quantities using the word `SEXT`. The word `RXYZ` reads all three accelerometer axes, leaving them in reverse order on the stack to be popped off in order later.

Initialize the SPI interface and stare the CMA3000 in the correct operating mode with `SPII CMA3000INIT`. Read and report the acceleration using the TEST loop until a key is pressed. The CMA3000 in 8G mode will output +1g as ~16 and -1g as ~16. And in 2G mode, ~+/- 64

6. References

[1] "ArrayFORTH User's Manual, for G144A12 and EVB001", Green Arrays DB004

[2] "polyFORTH II Reference manual", FORTH Inc.

[3] http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

[4] http://www.vtitechnologies.jp/documents/data_sheet/cma3000-d0x_product_family_specification_8281000a.02.pdf

[5] "G144A12 polyFORTH", Green Arrays App Note: DB06

[6] "F18A Technology Reference", Green Arrays DB001

IMPORTANT NOTICE