
Exploring 1-Wire Devices revised 5/25/14

This document describes a method to interface the Dallas Semiconductor high-precision one-wire digital thermometer (DS18S20) device with the EVB001 evaluation board. The hardware and software interface, discussed in this app note, uses arrayFORTH and polyFORTH, respectively. This illustrates one way to prototype such an interface from the manufacturer's one-wire protocol documentation.

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 Franklin Amador.

CONTENTS

1	Introduction	2
2	The one-wire Interface	2
2.1	Interface Circuit	2
2.2	One-wire Protocol Overview	3
2.3	Sending Data	4
2.4	Receiving Data	6
3	arrayFORTH Implementation	8
3.1	The one-wire Kernel	8
3.2	Loading	9
3.3	Initialization	9
3.4	SoftSim Integration	9
3.5	Testing on the Chip	10
4	polyFORTH Implementation	11
4.1	Integrating the one-wire kernel	11
4.2	Accessing the one-wire kernel from polyFORTH	11
4.3	Integration into polyFORTH	12
4.3.1	Accessing GA144 node	12
4.3.2	ROM Search Algorithm	13
4.3.3	High Level one-wire polyFORTH Code	15
4.3.4	polyFORTH Example 1	16
5	Conclusions	17
6	References	17

1 Introduction

This application note attaches a Dallas one-wire Temperature sensor (DS18S20) to the GreenArrays' evaluation board (EVB001) using a simple custom level shifting circuit. An off-the-shelf level shifter solution was decided upon by using Texas Instruments TXS0102 chip which already contained the necessary pull-up resistors for the one-wire interface. However, due to the no-lead DQE package ordered, it was impossible to wire leads to the chip without fully designing a surface mount board. As a result, a simple proof-of-concept circuitry was created and bread boarded with readily available components from the local electronics stores.

The one-wire kernel only uses 1 of 144 nodes for sending and receiving bytes over the one-wire network. The code does not include CRC error checking, but could be added at a later time for real applications. This applications note shows the basic bare-bone mechanisms needed to both interface and communicate via one-wire bus.

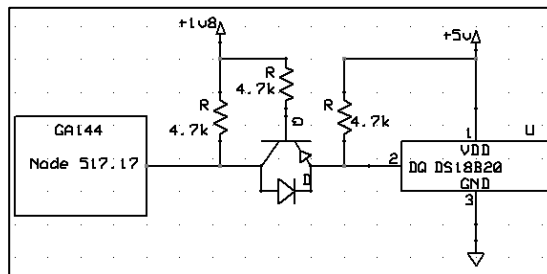
This effort was possible through Brad Rodriguez's one-wire high level Forth implementation on CamelForth [6] with GNU GPL v3. What is new and not part of CamelForth's one-wire implementation is the arrayFORTH's one-wire kernel and polyFORTH's interface to the arrayFORTH's one-wire kernel. As a result, the one-wire kernel work in this document is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. For commercial license interest, please contact the author for further details (fdamador@comcast.net).

2 The one-wire Interface

The one-wire protocol is discussed in every one-wire datasheet [7] and fully detailed in the "Book of iButton Standards" AN937 application note [8].

2.1 Interface Circuit

The interface circuit between the one-wire bus and GA144 chip is similar to SparkFun's logic level Bi-directional converter (BOB-12009). However, the major difference is the pnp transistor used as oppose to the N-Channel MOSFET proposed by SparkFun. Lastly, the base current needs to also be limited as it can cause the circuit to overheat and burnout and an additional bypass diode was added since the transistor package did not include one.



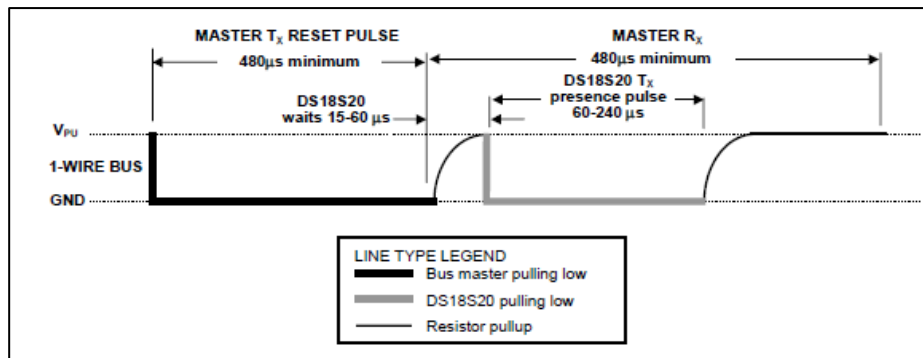
Powering the DS18S20 with an External Supply

Exploring 1-Wire Devices

The results from the digital switching between low level at 1.8Volts to high level at 5.0Volts worked as expected.

2.2 One-wire Protocol Overview

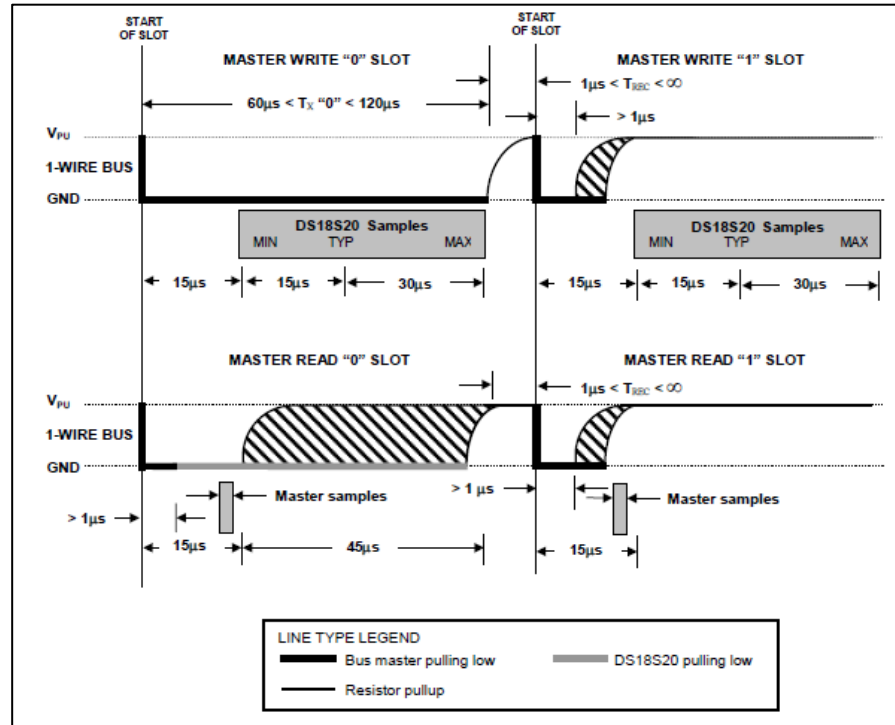
The one-wire protocol runs on master-slave architecture with a minimum of 6 types of signals; reset pulse, presence pulse, write 0, write 1, read 0, and read 1. All communication is initiated with a Reset pulse from the master and then waits for a presence pulse from any slaves on the network. If no presence pulse is detected then no data is transmitted by the high level polyFORTH code.



Initialization Timing

If a presence pulse is detected, then the master streams bytes of data at 1 bit per read/write time slots. Depending on the commands sent then 0 bytes, 8 bytes or 9 bytes are read back from the slave.

Exploring 1-Wire Devices



Read/Write Time Slot Timing Diagram

2.3 Sending Data

Every one-wire device has a set of unique command protocols that pertains to that particular device. In this particular device, the DS18S20 has the following unique protocol commands.

Function Command Set Table

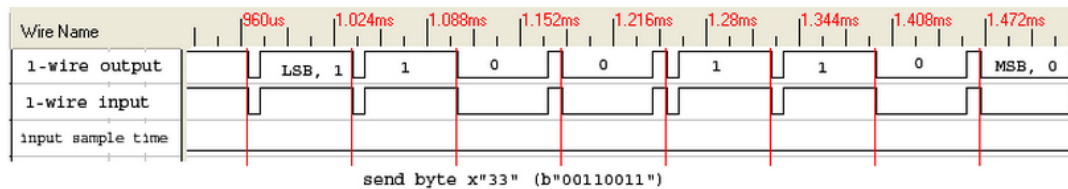
Exploring 1-Wire Devices

COMMAND	DESCRIPTION	PROTOCOL	1-Wire BUS ACTIVITY AFTER COMMAND IS ISSUED	NOTES
TEMPERATURE CONVERSION COMMANDS				
Convert T	Initiates temperature conversion.	44h	DS18S20 transmits conversion status to master (not applicable for parasite-powered DS18S20s).	1
MEMORY COMMANDS				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18S20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2 and 3 (T_H and T_L).	4Eh	Master transmits 2 data bytes to DS18S20.	3
Copy Scratchpad	Copies T_H and T_L data from the scratchpad to EEPROM.	48h	None	1
Recall E ²	Recalls T_H and T_L data from EEPROM to the scratchpad.	B8h	DS18S20 transmits recall status to master.	
Read Power Supply	Signals DS18S20 power supply mode to the master.	B4h	DS18S20 transmits supply status to master.	

- Note 1:** For parasite-powered DS18S20s, the master must enable a strong pullup on the 1-Wire bus during temperature conversions and copies from the scratchpad to EEPROM. No other bus activity may take place during this time.
- Note 2:** The master can interrupt the transmission of data at any time by issuing a reset.
- Note 3:** Both bytes must be written before a reset is issued.

At this point, it is emphasized that the term one-wire becomes blurred when an additional strong pullup is needed when in parasitic wire mode. Without either a strong pull up or separate external volt supplied to the DS18S20 device, we cannot get temperature conversion or Copy from ScratchPad (Protocol 44h or 48h) functions to work properly.

Sending data is as simple as writing a “0” or “1” slot as shown on the previous page. During a send command, we write the sequential binary data starting from LSB to MSB in 8 bit intervals. While the Master node is in sending mode, the one-wire devices listen to the data being sent on the bus and do nothing with the bus. As a result, the bus shows just the data being sent from the master. In the example below the Master is sending a hex byte of 33h (00110011b).



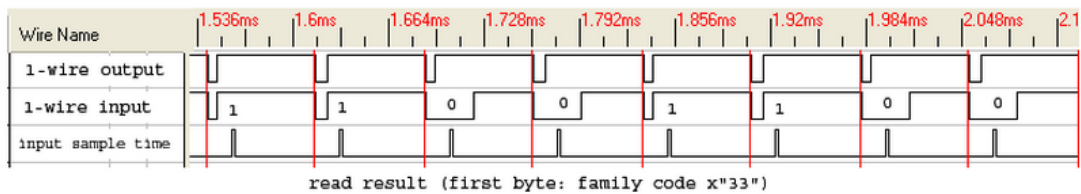
Sending Transmission Packet Timing

Exploring 1-Wire Devices

2.4 Receiving Data

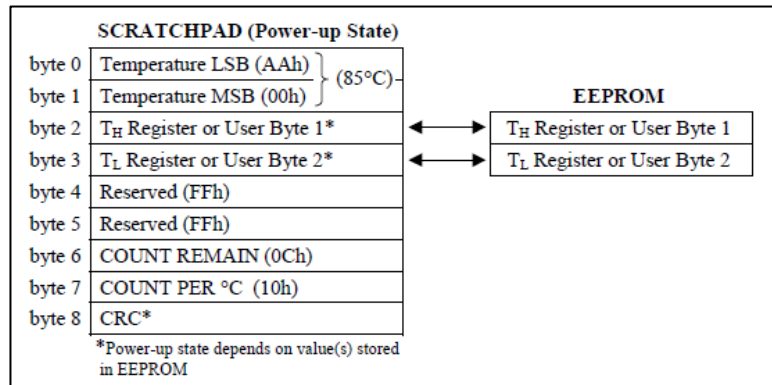
As the Master node is the one always initiating the transmission and receiving of data, the master has to first send bytes of ffh (11111111b) to sequence the data bits being sent back to the master from the slave. This lets the particular one-wire slave device synchronize when it's time to send its own data bits.

In the particular case below, the master always sends ffh data and the slave device waits for the initial pull down of the line to send it's 1s or 0s by either release or holding the bus line down respectively. Again, in the example below the Master is sending a hex byte of ffh (11111111b) and the slave is sending back a hex byte of 33h (00110011b). In the receiving data cycle, the master will wait a predetermined time (less than 15us) to sample the bus line for the bit state.



Receiving Transmission packet Timing

For a DS18S20 read scratch pad command (hex byte BEh) we'll receive a total of 9 bytes of data as outlined below.



Receiving Scratch Pad packets frames

Bytes 0 & 1 are further detailed in the Temperature Register Format list below. The LS Byte bit 0 is the .5 degrees Celsius. The MS Byte is only used for signed information of Positive (S=0) or Negative (S=1) temperature values.

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
LS BYTE	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	2 ⁻¹
	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
MS BYTE	S	S	S	S	S	S	S	S

S = SIGN

Exploring 1-Wire Devices

Temperature Register Format

The actual analog temperature scaling received from LSB (byte 0) and MSB (byte 1) is further defined in the Temperature/Data Relationship table.

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	DIGITAL OUTPUT (HEX)
+85.0*	0000 0000 1010 1010	00AAh
+25.0	0000 0000 0011 0010	0032h
+0.5	0000 0000 0000 0001	0001h
0	0000 0000 0000 0000	0000h
-0.5	1111 1111 1111 1111	FFFFh
-25.0	1111 1111 1100 1110	FFCEh
-55.0	1111 1111 1001 0010	FF92h

*The power-on reset value of the temperature register is +85°C.

Temperature/Data Relationship

It is interesting to note that a Hex-to-Decimal conversion and multiplied with 5 will give the temperature in Celsius with the decimal point moved to the right (multiplied by 10) or if we divide by 2 or right shift one bit we also arrive at a non-decimal resolution.

For detailed temperature fractions readings other than a 0.5 resolution the COUNT REMAIN (byte 6) and COUNT PER C (byte 7) from the scratch pad reading can be used with the fraction equation below. However, the Temperature reading (byte 0) will need truncated to eliminate the 0.5 resolution prior to using the value as TEMP_READ.

$$TEMPERATURE = TEMP_READ - 0.25 + (COUNT_PER_C - COUNT_REMAIN)/COUNT_PER_C$$

Fraction Temperature equation

The table below outlines the possible values from 5 *, 2/, and Temperature conversion.

Temperature	HEX	HEX2DEC	5 *	2/	TEMPERATURE =	TEMP_READ - 0.25 + (COUNT_PER_C - COUNT_REMAIN)/COUNT_PER_C
85.0	AA	170.00	850	85	85.6875 =	85 - 0.25 + (16 - 1)/ 16
25.0	32	50.00	250	25	25.625 =	25 - 0.25 + (16 - 2)/ 16
0.5	1	1.00	5	0.5	0.9375 =	0.5 - 0.25 + (16 - 5)/ 16
0.0	0	0.00	0	0	0.375 =	0 - 0.25 + (16 - 6)/ 16
-0.5	FFFF	-1.00	-5	-0.5	-0.25 =	-0.5 - 0.25 + (16 - 8)/ 16
-25.0	FFCE	-50.00	-250	-25	-25.0625 =	-25 - 0.25 + (16 - 13)/ 16
-55.0	FF92	-110.00	-550	-55	-55.125 =	-55 - 0.25 + (16 - 14)/ 16

3 arrayFORTH Implementation


One F18 computer (Node 517) with 49 words (31h) is used to implement the onewire – kernel in F18 code. No clock is needed for this application. The slaves basically wait until the master sends the necessary high/low commands for both synchronization and logic functions. Luckily for us, it is quite easy to figure out the right (200) delay with “**for . . unext**” commands to create an approximation to 1 usec. As a result, to approximate 500 usec we just multiply 500 with 200 to arrive at the necessary time delay that will be used in the kernel.

3.1 The one-wire Kernel

Two important words are used for the kernel; 1) **reset** and 2) **slot**. Both words follow the timing diagram as shown on the one-wire protocol overview section 2.2.

The reset word is in charge of initializing the beginning of data transfer to/from the master/slave. It basically transmits a master reset and waits to receive a presence pulse from the one-wire network. As a result, a high signal results in no one-wire device present and a low signal results in a one-wire device present. This data is inverted as 0 for no device or -1 for one device present and the result is sent back to polyFORTH.

The slot word is in charge of the Master Write and Master Read for a given 1/0 slot. In the slot word, we first pull the line low for 6 usec and see if the bit to be sent will be a 1 or 0. If the bit is a 1 then we just hold the line high otherwise we keep the line low. Then we continue with a 16 usec delay and sample input pin and or the result with the original 8 bit word to bit position 9. Lastly, we right shift one bit, wait for 35usec, pull the line high and delay for 2usec. This function completes both the Master Write and Read 1/0 slot commands. Please note that during Master Write the pin is sampled to finally end up with the same 8 bit word we originally wanted to send and that we discard this word in polyFORTH.

<pre>onewire dallas-1wire protocol, copyright 2014 franklin amador br dly timed usec loop delay wat wait 500 usec ?pin read pin.17 value hiz set pin.17 to high-impedance low set pin.17 low reset -n init device, presence pulse br slot read/write lbit to/from 1wire bus br send sample pin.17, if x100 xor else br receive right-shift main word, delay pfrset send presence pulse back pfslot send slot result back pftouch touch read/write @/! from node 516</pre>	<pre>864 list onewire -- kernel, copyright 2014 franklin amador, 517 node 0 org delay 00 for . . unext ; wait 02 100000 delay ; ?pin 04 @b 20000 and ; hiz 06 0 !b ; low 08 20000 !b ; reset 0a -n low wait hiz 15000 delay ?pin wait 11 if dup or ; 13 then - ; slot 14 n-n low 1200 delay dup 1 and ., 19 if hiz 1b then, send 1b drop 3800 delay ?pin, if if over 100 or over . . . 23 then, recv 23 drop 2/ 7000 delay hiz 400 delay ; pfrset 29 -n reset ! ; pfslot 2b n-n @ slot ! ; pftouch 2d n-n @ 7 for slot next ! ; 32</pre>	
---	---	---

In the end, node 517 basically waits for commands from node 516 which has Glanglia code to send its data back to polyFORTH via Snorkel. These network commands are integrated in 1) **pfrset**, 2) **pfslot** and 3) **pftouch** where pf stands for polyFORTH. As you can see, **pfrset** just sends data back to node 516 once initiated. Also, both **pfslot** and **pftouch** receive data (via Fetch thru a register) to be sent on the one-wire bus and once done then it transmit data (via Store thru a register) back to node 516.

3.2 Loading

A loading block is used to load the one-wire kernel. Block 200 must contain the instruction **862 Load**.

```
862 list
onewire -- loader,
reclaim,
864 load,
reclaim,
```

Loading Block

3.3 Initialization

Block 860 is used for the node initialization. This can be either used for SoftSim, loading the code into the GA144 chip, or integrating it into PolyFORTH boot stream.

```
860 list
onewire -- code loader,
517 +node 517 /ram right /a io /b 0 0 0 0 0 0
0 0 0 0 10 /stack right /p,
```

Initialization Block

/stack defines the values on the stack at chip initialization. This is helpful to load and not waste energy and pass the default **15555** stack data while the kernel is running. Notice that the communication port to node 516 is the right port. As a result, both the **/a** register and the **/p** register are loaded to look and wait for commands from polyFORTH at all times. The **/b** register is used to read/write to the node 517.17 pin.

3.4 SoftSim Integration

If we want to test our interface with SoftSim then block 866 is used to define the environment.

```
866 list
onewire -- softsim starter
/wave softbed assign time @ 10000 / 1 and ?v p
17v ! ;,
517 !node /wave,
```

SoftSim Integration

A **866 load** must be placed in block 216 to include our example in SoftSim. The bus on Pin 17 is changed every 10000 ticks. Note, after using SoftSim and before using the code on the chip we must comment out the **866 load** command.

3.5 Testing on the Chip

For testing on the host chip we create a block that loads the code onto the chip and another for interactive testing.

```
868 list
one-wire loader,
empty compile host load loader load,
0 517 hook 0 -hook,
860 load,
2 ship
```

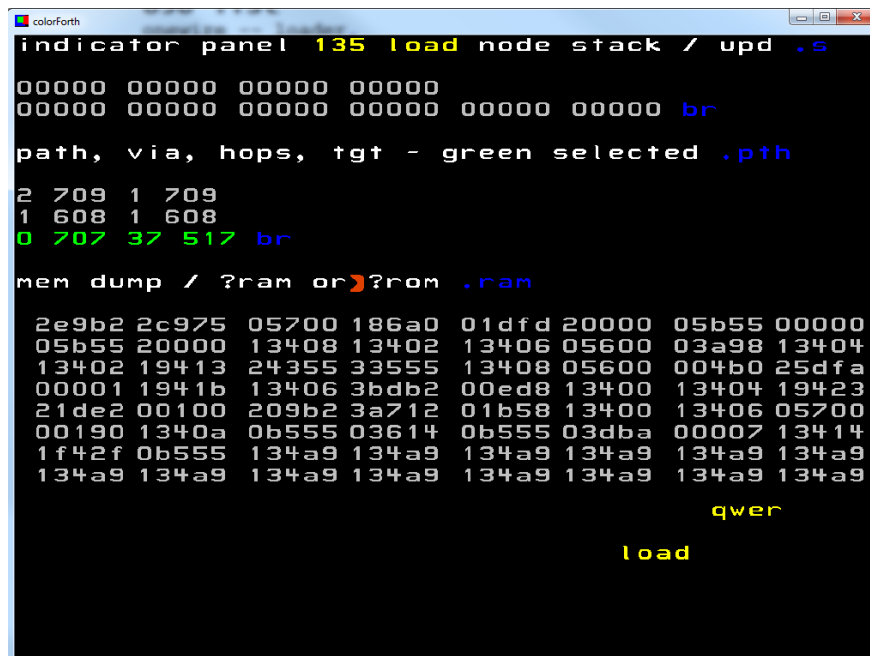
Loading code onto the chip

First we type **868 load**. After some time the code is loaded into the chip and started. Now we must use **858 load** to reset the chip and display the ROM/RAM.

```
858 list
onewire -- loader,
empty compile host load,
talk panel,
0 517 hook,
upd ?ram
```

Resetting the chip and displaying ROM/RAM.

We should now see the following screen:



```
colorforth
indicator panel 135 load node stack / upd .s
00000 00000 00000 00000
00000 00000 00000 00000 00000 00000 br
path, via, hops, tgt - green selected .pth
2 709 1 709
1 608 1 608
0 707 37 517 br
mem dump / ?ram or ?rom .ram
2e9b2 2c975 05700 186a0 01dfd 20000 05b55 00000
05b55 20000 13408 13402 13406 05600 03a98 13404
13402 19413 24355 33555 13408 05600 004b0 25dfa
00001 1941b 13406 3bdb2 00ed8 13400 13404 19423
21de2 00100 209b2 3a712 01b58 13400 13406 05700
00190 1340a 0b555 03614 0b555 03dba 00007 13414
1f42f 0b555 134a9 134a9 134a9 134a9 134a9 134a9
134a9 134a9 134a9 134a9 134a9 134a9 134a9 134a9
qwer
load
```

Interactive Panel IDE

4 polyFORTH Implementation

For using the full potential of the one-wire interface, we can integrate it into a virtual machine that takes care of the higher level aspects. The larger memory of the polyFORTH virtual machine is better suited for configuring the one-wire and interpreting the response.

4.1 Integrating the one-wire kernel

If we want to load our one-wire kernel code together with the polyFORTH virtual machine, then we can place a **860 load** into block 368 (or 478 for older versions), where the additional I/O for the virtual machine is loaded:

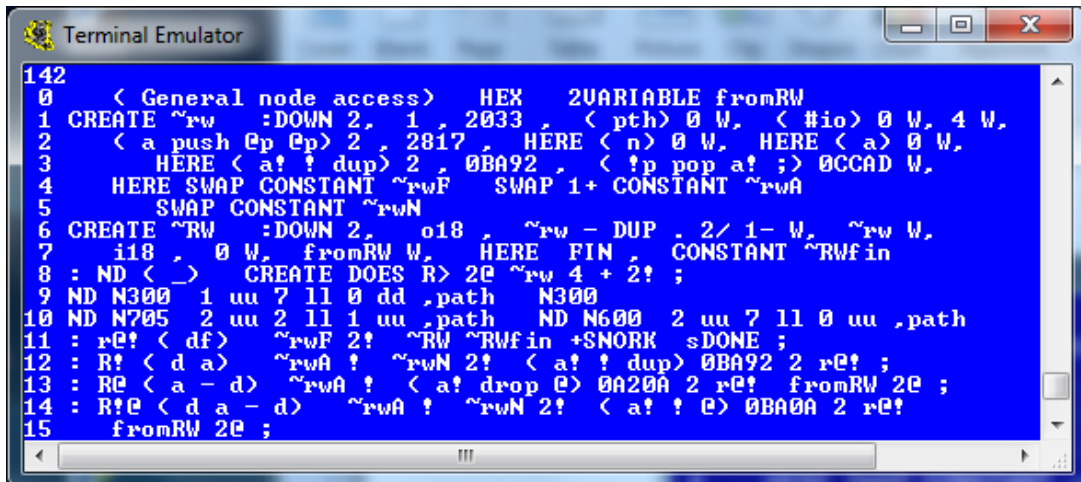
```
368 list
-- additional i/o,
spi 705 +node 1606 /ram io /b a9 /p,
async bootable 708 +node aa /p,
onewire 860 load
```

Listing 12 Integrating our code into the polyFORTH boot stream

When polyFORTH is started then our code will also be loaded into node 517. That is true whether you start polyFORTH from the IDE (450 load) or you install the polyFORTH boot stream in the flash (460 load).

4.2 Accessing the one-wire kernel from polyFORTH

After starting polyFORTH on the evaluation board, we load the snorkel and ganglia mechanism by doing a **142 load** command.

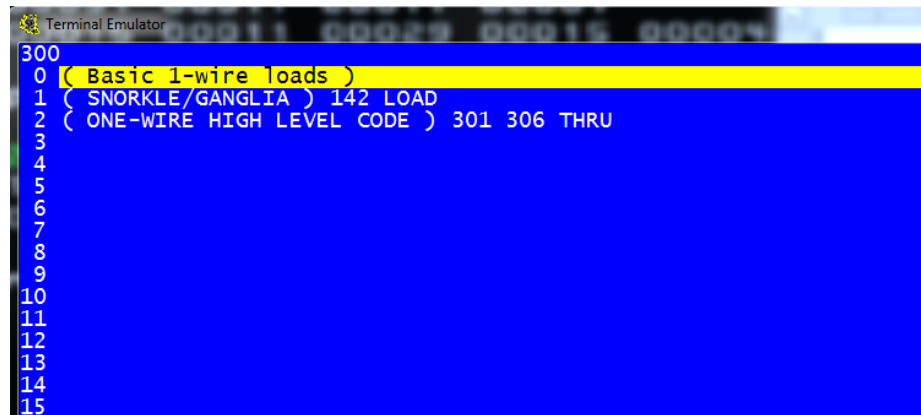


```
142
0 < General node access> HEX 2VARIABLE fromRW
1 CREATE ~rw :DOWN 2, 1, 2033, < pth> 0 W, < #io> 0 W, 4 W,
2 < a push @p @p> 2, 2817, HERE < n> 0 W, HERE < a> 0 W,
3 HERE < a! ! dup> 2, 0BA92, < !p pop a! ;> 0CCAD W,
4 HERE SWAP CONSTANT ~rwF SWAP 1+ CONSTANT ~rwA
5 SWAP CONSTANT ~rwN
6 CREATE ~RW :DOWN 2, 018, ~rw - DUP . 2/ 1- W, ~rw W,
7 i18, 0 W, fromRW W, HERE FIN, CONSTANT ~RWfin
8 : ND < _> CREATE DOES R> 2@ ~rw 4 + 2! ;
9 ND N300 1 uu 7 ll 0 dd ,path N300
10 ND N705 2 uu 2 ll 1 uu ,path ND N600 2 uu 7 ll 0 uu ,path
11 : r@! < df> ~rwF 2! ~RW ~RWfin +SNORK sDONE ;
12 : R? < d a> ~rwA ! ~rwN 2! < a! ! dup> 0BA92 2 r@! ;
13 : R@ < a - d> ~rwA ! < a! drop @> 0A20A 2 r@! fromRW 2@ ;
14 : R?@ < d a - d> ~rwA ! ~rwN 2! < a! ! @> 0BA0A 2 r@!
15 fromRW 2@ ;
```

arrayFORTH's block 142

4.3 Integration into polyFORTH

Until now we have used our one-wire interface with arrayFORTH. In order to fully support the one-wire protocols and its features we must implement a little driver that converts the one-wire commands to Read/Write bytes. To simplify the task we load block 300 for loading the one-wire driver.

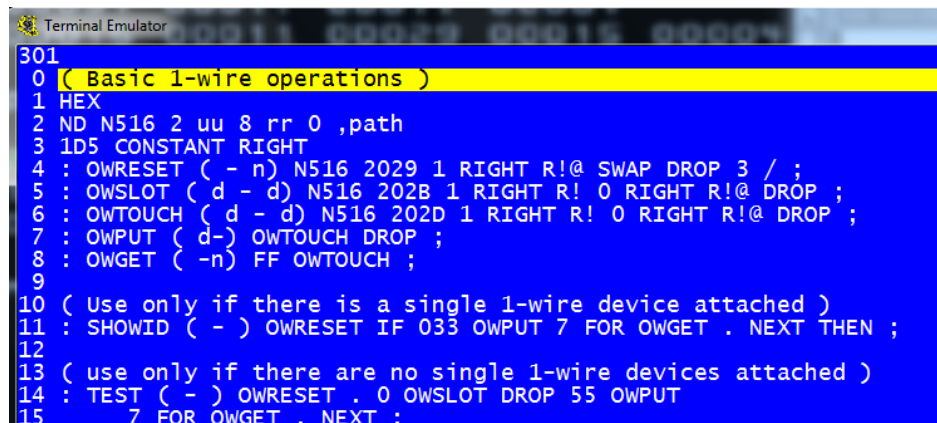


```
Terminal Emulator
300
0 ( Basic 1-wire loads )
1 ( SNORKLE/GANGLIA ) 142 LOAD
2 ( ONE-WIRE HIGH LEVEL CODE ) 301 306 THRU
3
4
5
6
7
8
9
10
11
12
13
14
15
```

One-wire loading

4.3.1 Accessing GA144 node

Block 301 defines the words (OWRESET, OWSLOT, OWTOUCH, OWPUT, and OWGET) for actually accessing the one-wire kernel read and write to 517.17 io pin.



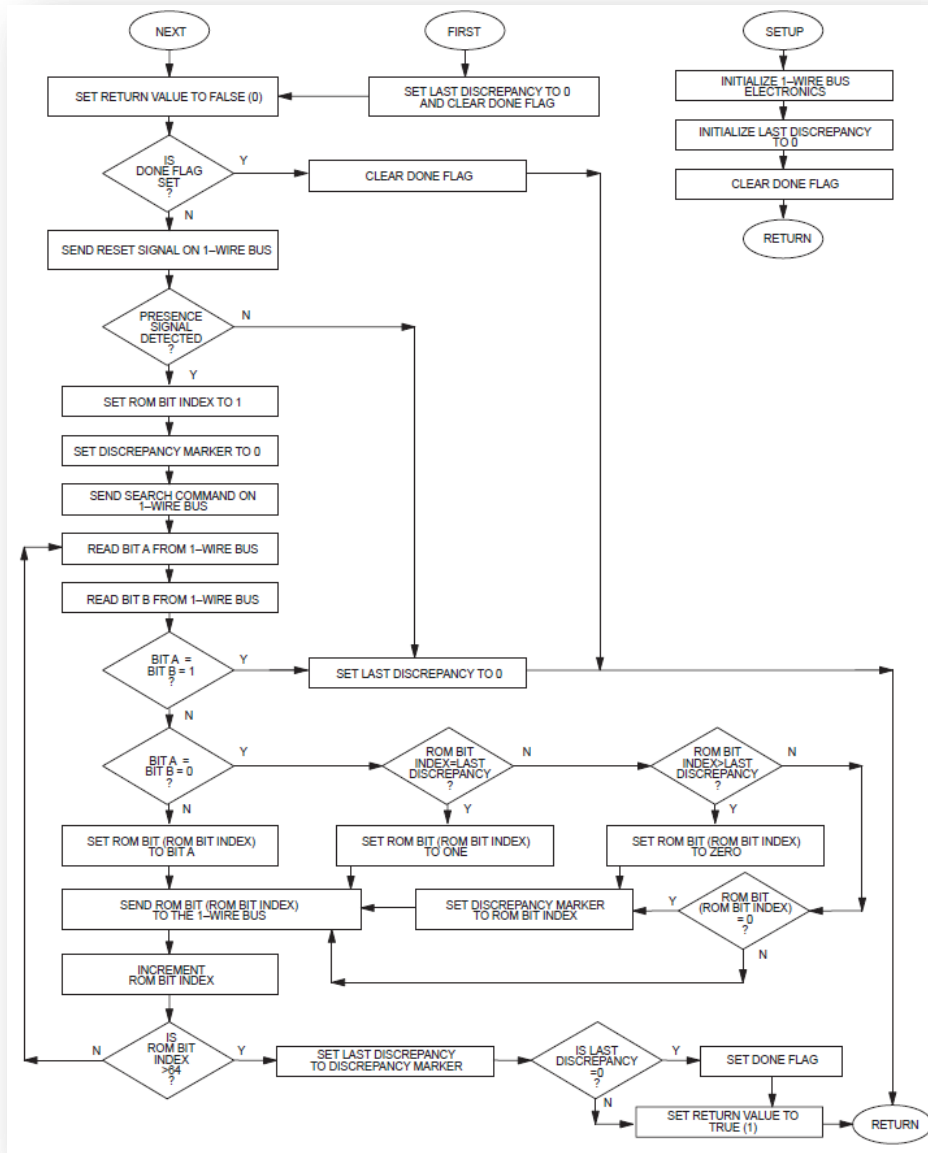
```
Terminal Emulator
301
0 ( Basic 1-wire operations )
1 HEX
2 ND N516 2 uu 8 rr 0 ,path
3 1D5 CONSTANT RIGHT
4 : OWRESET ( - n) N516 2029 1 RIGHT R!@ SWAP DROP 3 / ;
5 : OWSLOT ( d - d) N516 202B 1 RIGHT R! 0 RIGHT R!@ DROP ;
6 : OWTOUCH ( d - d) N516 202D 1 RIGHT R! 0 RIGHT R!@ DROP ;
7 : OWPUT ( d-) OWTOUCH DROP ;
8 : OWGET ( -n) FF OWTOUCH ;
9
10 ( Use only if there is a single 1-wire device attached )
11 : SHOWID ( - ) OWRESET IF 033 OWPUT 7 FOR OWGET . NEXT THEN ;
12
13 ( use only if there are no single 1-wire devices attached )
14 : TEST ( - ) OWRESET . 0 OWSLOT DROP 55 OWPUT
15 : 7 FOR OWGET . NEXT ;
```

polyFORTH Block 301

Note that we first define a path to Node 516 which will finally talk to Node 517. The wire path to Node 516 uses Snorkel and Ganglia via the polyFORTH virtual machine. Then we further re-define each one-wire words in Node 517 for polyFORTH VM interpretation. I leave it to the reader to study Application Note 009 for other existing examples of using Node 142 for R!, R@ and R!@ commands.

4.3.2 ROM Search Algorithm

To search for available devices over the one-wire bus, Dallas Semiconductors has provided a search algorithm outline on their Book for iButton Standards [8] Figure 5-3.



ROM Search Algorithm

Blocks 302, 303 and 304 referenced below are used from 4e4th one-wire implementation by Brad Rodrigues [6]. What is not included in [6] are the LSHIFT and RSHIFT commands that I implemented in polyFORTH.

Exploring 1-Wire Devices

Block 302 defines the all the variables bytes for the ROM Search Algorithm. Here we also define missing Left Shift (LSHIFT) and Right Shift (RSHIFT) from the polyFORTH VM.

```
Terminal Emulator
302      0 Refs      0 Other blocks
0 ( Maxim 1-wire ROM Search algorithm )
1 HEX
2 VARIABLE LASTDISC
3 LASTDISC 1+ CONSTANT DONEFLAG
4
5 VARIABLE ROMBIT
6 ROMBIT 1+ CONSTANT DISCMARK
7
8 VARIABLE ROMID 6 ALLOT
9
10 : LSHIFT DUP 0 > IF 1- FOR 2* NEXT ELSE DROP THEN ;
11 : RSHIFT DUP 0 > IF 1- FOR 2/ NEXT ELSE DROP THEN ;
12
13
14
15
```

polyFORTH Block 302

Block 303 defines the ROMBIT fetch and store routines and reset of

```
Terminal Emulator
303
0 ( Maxim 1-wire ROM Search algorithm continued )
1 HEX
2 : !ROMBIT ( f -- ) ROMBIT C@ 1- 8 /MOD
3   ROMID + 1 ROT LSHIFT ROT
4   IF
5     OVER C@ OR SWAP C!
6   ELSE
7     INVERT OVER C@ AND SWAP C!
8   THEN ;
9
10 : @ROMBIT ( -- f ) ROMBIT C@ 1- 8 /MOD
11   ROMID + C@ 1 ROT LSHIFT AND ;
12
13 : NEWSEARCH 0 LASTDISC ! ;
14
15
```

polyFORTH Block 303

Block 304 defines the ROM SEARCH algorithm [8] as outlined by the logical diagram Figure 5-3.

```
Terminal Emulator
304
0 HEX : ROMSEARCH ( -- f ) 0 DONEFLAG C@
1 IF 0 DONEFLAG C! EXIT
2 THEN OWRESET
3 IF 1 ROMBIT C! 0 DISCMARK C! F0 OWPUT
4 BEGIN 03 OWSLOT OWSLOT DUP
5   C0 = IF DROP 0 LASTDISC C! EXIT
6   ELSE DUP 0= IF DROP ROMBIT C@ LASTDISC C@ = IF 1 !ROMBIT
7     ELSE ROMBIT C@ LASTDISC C@ > IF 0 !ROMBIT ROMBIT C@
8     DISCMARK C!
9     ELSE @ROMBIT 0= IF ROMBIT C@ DISCMARK C! THEN THEN THEN
10    ELSE 40 AND !ROMBIT THEN THEN
11    @ROMBIT IF 1 ELSE 0 THEN OWSLOT DROP ( SEND TO BUS )
12    ROMBIT C@ 1+ DUP ROMBIT C!
13    40 > UNTIL DISCMARK C@ DUP LASTDISC C!
14    0= IF 1 DONEFLAG C! ELSE DROP 1 THEN
15 ELSE 0 LASTDISC C! THEN 0 DONEFLAG C! ;
```

polyFORTH Block 304

4.3.3 High Level one-wire polyFORTH Code

Block 305 finally defines the last of the high level polyFORTH interface to the one-wire protocol. The SHOWIDS uses the ROMSEARCH algorithm to display any device ID on the network. SENDID focuses on that particular device for further commands. READSCRATCH, OWCONVERT and READTEMP are specific to the DS18S20 temperature device.

```

Terminal Emulator
305
0 ( Maxim 1-wire high level operations )
1 HEX
2 : SHOWIDS ( -- ) NEWSEARCH
3 BEGIN ROMSEARCH CR ROMID 8 + ROMID DO I C@ 3 U.R LOOP
4 0= UNTIL CR ;
5
6 : SENDID ( addr -- ) OWRESET
7 IF 55 OWPUT 8 OVER + SWAP DO I @ OWPUT LOOP
8 ELSE ." failed" DROP THEN ;
9
10 : READSCRATCH ( a - ) SENDID BE OWPUT 8 FOR OWGET . NEXT ;
11
12 : OWCONVERT ( a -- ) SENDID 44 OWPUT ;
13
14 : READTEMP ( a -- n ) SENDID BE OWPUT OWGET OWGET 8 LSHIFT OR ;
15
    
```

polyFORTH Block 305

Although NOT coded verbatim from reference [6], the “TEMP>PAD” is now refined for this polyFORTH implementation.

```

Terminal Emulator
306
0 ( Maxim 1-wire high level operations )
1 HEX
2 : TEMP>PAD ( n -- ) 5 * DECIMAL DUP ABS 0
3 <# # 2E HOLD # #S SIGN #> TYPE SPACE ;
4
5 : OWID ( --addr ) 8 OVER + SWAP DO I @ . LOOP ;
6
7 HEX CREATE SENSOR1 10 , FB , 24 , B8 , 2 , 8 , 0 , 22 ,
8 HEX CREATE SENSOR2 10 , CA , B7 , B6 , 2 , 8 , 0 , 14 ,
9
10 : FINAL1 SENSOR1 OWCONVERT 750 MS SENSOR1 READTEMP TEMP>PAD ;
11
12 : FINAL2 SENSOR2 OWCONVERT 750 MS SENSOR2 READTEMP TEMP>PAD ;
13
14 : EXAMPLE1 SENSOR1 OWCONVERT 750 MS SENSOR1 READSCRATCH ;
15
    
```

PolyFORTH Block 306

It's my hope that in a future Application Note that I can implement a Floating Point library for polyFORTH.

4.3.4 polyFORTH Example 1

The following sequence of steps are coded block 306 as the define EXAMPLE1 polyFORTH word. As you can see, the simplicity of Forth is apparent when viewed in the top down hierarchical forth word command arrangements.

MASTER MODE	DATA (LSB FIRST)	COMMENTS
TX	Reset	Master issues reset pulse.
RX	Presence	DS18S20s respond with presence pulse.
TX	55h	Master issues Match ROM command.
TX	64-bit ROM code	Master sends DS18S20 ROM code.
TX	44h	Master issues Convert T command.
TX	DQ line held high by strong pullup	Master applies strong pullup to DQ for the duration of the conversion (t_{conv}).
TX	Reset	Master issues reset pulse.
RX	Presence	DS18S20s respond with presence pulse.
TX	55h	Master issues Match ROM command.
TX	64-bit ROM code	Master sends DS18S20 ROM code.
TX	BEh	Master issues Read Scratchpad command.
RX	9 data bytes	Master reads entire scratchpad including CRC. The master then recalculates the CRC of the first eight data bytes from the scratchpad and compares the calculated CRC with the read CRC (byte 9). If they match, the master continues; if not, the read operation is repeated.

Below are some of the working commands for the one-wire protocol. The first command "SHOWIDS" will query the one-wire bus for available devices. I had a bug in the ROMSEARCH algorithm that took more than one time to actually get all the devices ID, but this is now corrected. In my particular one-wire network I only have two DS18S20 devices.

```

SHOWIDS
 10 FB 24 B8 2 8 0 22
ok
SHOWIDS
 10 CA B7 B6 2 8 0 14
 10 FB 24 B8 2 8 0 22
ok
FINAL1 21.0 ok
FINAL2 20.5 ok
ok
EXAMPLE1 42 0 75 70 255 255 15 16 64 ok
HEX EXAMPLE1 2A 0 4B 46 FF FF F 10 40 ok
    
```

Working One-wire commands

Lastly, we show the "FINAL1" & "FINAL2" working words that receives the temperature of both devices in degrees centigrade, respectively. Lastly, we show the data received back from the example 1 above. The scratchpad data from Sensor1 displays all the available information as described section 2.4.

5 Conclusions

We have shown how to interface the Dallas Semiconductor one-wire master protocol in the multi-computer architecture of the GA144 chip. Combining F18A machine level language using arrayFORTH and high level polyFORTH virtual machine gives any novice or experienced developer the flexibility to design both in Hardware and Software. It is further emphasized that the one-wire kernel, although considered a slow protocol by GreenArrays standard, was implemented using a mere 50 words on 1 out of 144 nodes.

The author of this document is a system integrator for a controls engineering company. This is the author's 1st attempt at implementing forth in any application. With the help of the documents and staff from GreenArrays it was possible to write the driver software in a few weeks. Although arrayFORTH and polyFORTH are very different than environments most programmers are used to, GreenArrays tools were complementary to its architecture 1) simple and 2) modular.

This application note was finished at nights while carrying the author's 9 month year old on a baby carrier. The author wants to thank for the help and patience he received from the GreenArray staff, Charley Shattuck.

6 References

- [1] "ArrayFORTH User's Manual, for G144A12 and EVB001", Green Arrays DB004
- [2] "polyFORTH Reference manual", GreenArrays Data Books: DB005
- [3] "G144A12 polyFORTH supplement to DB005", GreenArrays Data Books: DB06
- [4] "F18A Technology Reference", GreenArrays Data Books: DB001
- [5] "Attaching a PS/2 Keyboard", GreenArrays App Note: AN009
- [6] <http://www.camelforth.com/download.php?view.27>
- [7] <http://datasheets.maximintegrated.com/en/ds/DS18S20.pdf>
- [8] <http://pdfserv.maximintegrated.com/en/an/AN937.pdf>
- [9] <http://en.wikipedia.org/wiki/1-Wire>