AttoBASIC Application Notes

For Version 2.31+

Table of Contents

AttoBASIC Application Notes
Version 2.31+

AttoBASIC Overview

AttoBASIC is the creation of Dick Cappels, a retired analog engineer whose career history includes developing circuitry for APPLE and RAYETHON.

Development of this "tiny" BASIC began in early 2002 with ATMEL's AT90S2313, the 2nd generation of ATMEL's AVR-8 series of low-power, low-cost microcontrollers. The AT90S2313 has 2KB of FLASH memory, 128 bytes of RAM and 128 bytes of EEPROM. Mr. Cappels' motivation was simply to see if he could write a BASIC interpreter to fit into a 2KB code space … and it was rather attractive to see a 20-pin MCU that could run a BASIC interpreter that was hardware-orientated.

In choosing a name for this BASIC, "Tiny BASIC" was widely used and seemed to be "too large" for the project. With only 2KB of program space available, even "Pico BASIC" seemed "too large", hence the (de)evolution into the name AttoBASIC, wherein "atto" is two (2) orders of magnitude smaller than "pico".

Mr. Cappels' prior experience as a FORTH programmer led to his design of an interpretive BASIC, "FORTH'ish" in nature. "FORTH'ish" because data submitted to and returned from commands is passed on a "data stack" and the interpreter engine is recursive, which allows subsequent commands to pass their data back to a prior command. This method has the effect of allowing most commands to act as "functions" to other commands, much like those of C/C++ and other "high-level" programming languages. [*Programming examples of the recursive nature of AttoBASIC are demonstrated in sample programs throughout this application note.*]

Mr. Cappels was successful in implementing AttoBASIC with primitive commands providing access to I/O, the PWM and the analog comparator. Communications with AttoBASIC was via a host's serial communications program (terminal emulator) and the AT90S2313's on-chip USART. Programs were limited to 72 characters, which meant that the AttoBASIC of that era could execute only a few instructions in a program. It's power, however, was in direct command mode where direct real-time manipulation the PWM, I/O ports and pins could be easily accomplished. [*This was the primary reason why this author chose to acquire and evolve AttoBASIC into it's current incarnation.*]

Within a year, Mr. Cappels developed version for the AT90S8515 and Atmega163, adding support for the on-chip Analog-to-Digital Converter.

In mid-2011, this author acquired AttoBASIC and with Mr. Cappels' blessing, gave it a "face-lift". It was initially ported to the ATmega88/168 and ATmega32U4 and was released as Version 2.0.

The writing entitled "*AttoBASIC V2.xx Revision History*" contains the complete revision history of AttoBASIC since Version 2.00.

After data file support was added in version 2.20, a path for a natural extension was created to use AttoBASIC as the heart of a programmable data recorder (see *AVR Data Recorder project*).

It has been suggested that AttoBASIC has outgrown its name and perhaps should be renamed to something more "size and feature appropriate". Perhaps "MegaBASIC"?

Mr. Cappels hosts a web site containing many free projects, with schematics and source code (when MCU-based). The web site is http://www.cappels.org.

Console I/O

Interfacing with a host computer

AttoBASIC communicates with the console via a serial I/O interface. On AVR flavors that use a USART , such as the ATmega88/168/328/2560, the interface to a host computer is usually through a user-supplied level shifter (MAXIM MAX203/221/222, etc. type) or a USB to Serial bridge (FTDI FTD-232, CP-2102, etc. type). ARDUINO and compatibles usually have this ability on-board.

On AVR flavors that have an on-chip USB interface, such as the Atmega32U4 and AT90USB1286, Atto-BASIC contains routines that enable the AVR's USB interface to act as a "Virtual Communications Port" (VCP).

In either case, Linux platforms and most WINDOWS® platforms support these interfaces. If a USB driver is required for the WINDOWS® platform, there are some available in the *_USB_Drivers* folder.

Using the proper baud rate

AttoBASIC comes pre-build with HEX files for clock speeds of 4, 8, 16 and 20 MHz. However, one may wish to build AttoBASIC from the source code using a clock frequency less than 4 Mhz.

Depending on the clock speed of the target MCU, the baud rate chosen is the fastest with the lowest bit error, typically no more than ±0.16%. Therefore, the baud rate for the different clock frequencies of the pre-built HEX files will be as follows:

38.4K baud for clock frequencies of 8, 16 and 20 MHz.
19.2K baud for clock frequencies of 4 Mhz

For custom builds, the baud rate automatically chosen for no more than ±0.16% error will be:

9600 baud for clock frequencies of 2 MHz.
4800 baud for clock frequencies of 1 MHz.

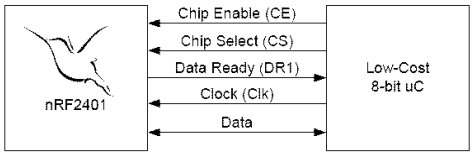
Default MCU Pinouts for Hardware Support Functions

Function	Atmega88/168/328	Atmega32U4	Atmega2560	AT90USB1286
PWM (OC1A/B)	PB1 (OC1A) only	PB5 & PB6	PB5 & PB6	PB5 & PB6
ICP	PD5 (T1)	PD6 (T1)	PE6 (T3)	PC3 (T3)
DDS	PD6	PD5	PD5	PE4
DHT	PD7	PB2	PB2	PB2
TWI	PC4 & PC5	PD0 & PD1	PD0 & PD1	PD0 & PD1
SELF-START	PC3	PD7	PD7	PD7
SPI	PB[5..2]	PB[3..0]	PB[3..0]	PB[3..0]
DS	PD3 & PD4	PC6 & PC7	PC4 & PC5	PC4 & PC5
NRF24L01 CE	PB0	PB4	PB4	PB4
1-WIRE	PD3	PB5	PB5	PB5

Modules enabled for default HEX file builds

Function	Atmega88	Atmega168	Atmega328	Atmega32U4	Atmega2560	AT90USB1286
DEBUG		■	■	■	■	■
DDS		■	■	■	■	■
DS			■	■	■	■
PWM	■	■	■	■	■	■
SPI	■	■	■	■	■	■
DATA FILE		■	■	■	■	■
TWI		■	■	■	■	■
1-WIRE	■	■	■	■	■	■
ADC	■	■	■	■	■	■
INT REG's		■	■	■	■	■
SELF-START	■	■	■	■	■	■
ICP		■	■	■	■	■
EFS		■	■	■	■	■
LOW PWR		■	■	■	■	■
NRF24L01		■	■	■	■	■
DHT		■	■	■	■	■
RTC		■	■	■	■	■
HELP	■	■	■	■	■	■

nRF24L01(+) RF Transceiver Support (Version 2.31+)



As of version 2.31, AttoBASIC adds language extensions to support the NORDIC nRF24L01(+) RF Transceiver IC. NORDIC SEMICONDUCTOR has created a superb RF transceiver that is easy to interface with, easy to program and simply works "out of the box" with little configuration. The best part about it is that nRF24L01-based transceiver modules can be acquired from an eBay vendor for US$3.00 or less each!

Note that for purposes of this writing: 1) the nRF24L01 and nRF24L01+ devices are the same, 2) the abbreviation "TX" means "transmitter" and the abbreviation "RX" means "receiver", 3) the term "payload" refers to a packet of data either transmitted or received.

Use the proper band for the designated jurisdiction:
If operating under "U.S. jurisdiction", use only channels 0 to 83 otherwise, agents of the U.S. FCC may be upset and wish to levy "charges" in the form of "fines" for violations. Consult the "local rules" for the "jurisdiction" in which the nRF24L01(+) will be used.

Reading payload data:
With the use of the **DATA, READ** and **REST** commands, AttoBASIC implements a simplistic approach to the task of reading various sizes of payload data from the various nRF24L01(+) receiver pipes whose data is in the FIFO (first-in-first-out) buffer. The **RFX** command determines which of the receiver pipes has data in the RX FIFO buffer(s), transfers it to the DATA statement's buffer and returns the number of the pipe the data was transferred for. In this way, the user can not only read the pipe's data but easily determine which pipe the data came from. AttoBASIC keeps track of the varying sizes of the payloads transferred to the DATA statement's buffer, but it is the user's responsibility to use the **REST** command to determine a payload's size if payload sizes vary between RX pipes.

Configuration of the nRF24L01(+):
At the very minimum, after initial power-up (or an **RFI 3** command), the device needs to be configured as a transmitter (TX) or receiver (RX). Since the power-up defaults pre-select channel 2, a 2 mbps data rate and a TX output level of 0 dBm, a single **RFI** command (or **RFI 1 2 1**) is all that is needed to configure the device as a transmitter. Once configured, data can be transmitted (to a listening receiver) with a single **RFT 1 2 3 4** command, for example.

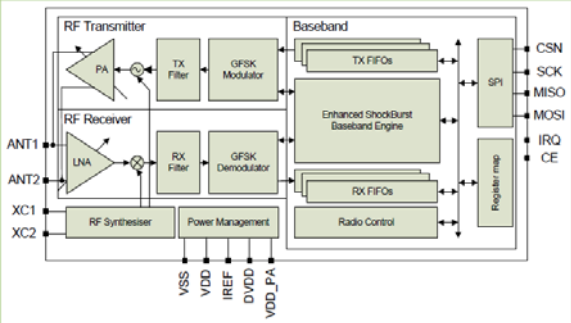
NORDIC's *ShockBurst*^{TM} technology is inherent in the nRF24L01(+) devices. As a power-up default, *ShockBurst*^{TM} is enabled with auto-acknowledgment, automatic retransmissions and CRC for all RX pipes and payloads. Although those features may be manipulated with the **RFW** and **RFR** commands, they are transparent to the AttoBASIC command set and any extra measures to insure robust data transmission are unnecessary.

The AttoBASIC command set does not directly support the "dynamic payload" and "Payload with ACK" features of *Enhanced ShockBurst*^{TM}, however, with the use of the **RFW** and **SPW** commands, one may make use of them.

Returning the nRF24L01(+) to power-up defaults:
Due to the fact that the nRF24L01(+) I.C. does not provide a physical pin to perform a hardware reset, the only means to reset the internal registers to factory defaults is to power-cycle the part. As a means to that end though, the **RFI 3** command will reset all registers to their power-up state.

Checking the TX/RX status:
The nRF24L01(+) provides several means to determine if the three (3) transmit or receive FIFO buffers are full or empty and if data was transmitted reliably, if at all. Consult the device's datasheet to determine the meanings of the various bits in the registers.

The use of the **RFR** command without any command line parameters will always return the contents of register $17, "FIFO_STATUS". Keep in mind that the TX and RX FIFO buffers are only three levels (3) deep, so any additional writes, either in TX mode (with the **RFT** command) or data received in RX mode, will delete the oldest packet in the respective FIFO buffers. The exception is when a payload is not received and blocks the TX FIFO buffer from further transmissions, in which case, consult the subsection entitled "*Recovering from errors*". Fortunately, TX FIFO overrun is not likely to be a problem with AttoBASIC since the time it takes to transmit a payload is typically less than the time it takes AttoBASIC to execute sequential commands. Tests indicate 15 mS between back-to-back transmissions on a 16 MHz ATmega2560.



When configured as a receiver, polling the "FIFO_STATUS" register will allow one to determine if there is data in the receive FIFO buffers (RX FIFO not empty flag) and if the receive FIFO buffers are full (RX FIFO full flag). Register $07, "STATUS", may also be polled, which will also allow one to determine if data has been received and which pipe's data is currently first in the FIFO buffer. Polling the "STATUS" register for the RX pipe that received data isn't really necessary since the **RFX** command provides that after transferring the data from the RX FIFO buffer.

When configured as a transmitter, polling the "FIFO_STATUS" register will allow one to determine if there is data still in the transmit FIFO buffers (TX FIFO not empty flag) and if the transmit FIFO buffers are full (TX FIFO full flag). Register $07, "STATUS", may also be polled, which will also allow one to determine if the data was successfully transmitted (TX_DS flag) or if it was unsuccessful due to too many TX retransmits (MAX_RT flag). Polling the "FIFO_STATUS" register isn't really necessary since the **RFT** command provides the status of the TX FIFO buffer or an indicator that the last transmission failed.

Transmitter example:

The following example program illustrates a simple transmitter, which sends the 32-bit value of the AttoBASIC Real-Time Counter as a four (4) byte payload packet to a listening receiver. This program could be used, for instance, to synchronize two (2) AttoBASIC host's real-time counters.

```
10 C:=0                          # INIT PAYLOAD COUNTER
15 RFI 1 0 1                     # POWER UP IN TX MODE, CHANNEL 0, 2 MBPS
20 RFB 0 4                       # 4 BYTES OF DATA RX0
25 RFF                           # FLUSH RX AND TX BUFFERS
30 PRINT "----------"            # PRINT A PERFORATION
35 PRINT "PAYLOAD " ; PRINT C    # PRINT A HEADER
40 GOSUB 50 ; C:= C + 1          # CALL TRANSMIT ROUTINE AND INCREMENT PAYLOAD COUNTER
45 SLP 7 ; GOTO 30               # SLEEP 1 SECOND AND LOOP
50 M:=PEEK VPG@RTC3              # FETCH THE 4 RTC BYTES
55 N:=PEEK VPG@RTC2
60 O:=PEEK VPG@RTC1
65 P:=PEEK VPG@RTC0
70 PRINT "D:" ; PRX M            # SHOW THE TIMESTAMP TO USER
75 PRINT "D:" ; PRX N
80 PRINT "D:" ; PRX O
85 PRINT "D:" ; PRX P
90 RFT M N O P                   # SEND THE RTC DATA
95 RET                           # RETURN TO CALLER
```

Receiver example:

To configure as a receiver, an **RFI 2 2 1** command, for example, is executed along with an **RFB 0 4** command (to set pipe 0 to receive 4 bytes of payload, for example) and the **RFE 1** command to enable the device's receiver. Once configured, periodically polling the device's STATUS or FIFO_STATUS register will allow one to determine if data has been received. The following example program illustrates a simple receiver, which receives the four (4) byte RTC value sent by the transmitter program above and prints them to the user's console.

```
10 C:= 0                          # INITIALIZE A PAYLOAD COUNTER
15 RFI 2 0 1                      # POWER UP IN RX MODE, CHAN 0, 2 MBPS
20 RFB 0 4                        # 4 bytes of data for RX0
25 RFF ; RFE 1                    # FLUSH RX AND TX FIFO AND ENABLE RX MODE
30 A:= RFR; A:= A AND 1           # FETCH THE STATUS OF THE RX FIFO
35 PRI "." ; SLP 3                # PRINT A STATUS CHARACTER AND DELAY 128mS
40 IF !A THEN GOTO 30             # CHECK FOR FIFO NOT EMPTY
45 PRINT "~PAYLOAD # "; PRINT C   # PRINT SOME INFO FOR THE USER
50 C:= C+1                        # INCREMENT THE PAYLOAD COUNTER
55 P:= RFX                        # TRANSFER THE RX DATA TO THE DATA BUFFER AND
                                  #   ASSIGN THE PIPE # TO VARABLE P
60 FOR N= 1 4                     # SET UP THE LOOP COUNTER FOR 4 BYTES
65 PRI "D:" ; PRX READ N          # PRINT THE DATA RECEIVED
70 NEXT                           # LOOP FOR ALL FOUR DATA BYTES
75 PRI "----------~"              # PRINT A PERFORATION
80 GOTO 25                        # LOOP
```

MultiCeiver™ examples:

The MultiCeiver™ capability of the nRF24L01(+) allows one to implement a 6 transmitter, 1 receiver "star network" using a single RF channel. The AttoBASIC command set supports the MultiCeiver™ mode using the **RFA** and **RFB** commands. The following example programs illustrate a basic transmitter and receiver combination. For these examples, the device's default MAC addresses are used. For each transmitter, the TX pipe and RX pipe 0's MAC address is changed to the MAC address of the desired RX pipe on the receiver. For this example, transmission to the receiver's pipe 3 is used. The receiver program is the same as the one shown above with the exception that receive pipe 3 is enabled with a buffer size of 4 bytes, which allows our transmitter to send the data to pipe 3. The relevant changes for each program are emphasized in bold text.

```
5  REM nRF24L01 MULTICEIVER-RECEIVER
10 C:=0                           # INIT PAYLOAD COUNTER
15 RFI 2 0 1                      # POWER UP IN RX MODE, CHAN 0, 2 MBPS
20 RFB 3 4                        # 4 bytes of data for RX3
25 RFF ; RFE 1                    # FLUSH RX AND TX FIFO AND ENABLE RX MODE
30 A:= RFR; A:= A AND 1           # FETCH THE STATUS OF RX FIFO
35 PRI "." ; SLP 3                # PRINT A STATUS CHARACTER AND DELAY
40 IF !A THEN GOTO 30             # CHECK FIFO
45 PRINT "~PAYLOAD # "; PRINT C   # PRINT SOME INFO
50 C:= C+1                        # INCREMENT THE PAYLOAD COUNTER
55 PRINT "FIFO: " ; PRINT RFX     # TRANSFER THE RX DATA TO THE DATA BUFFER AND PRINT
                                  # THE PIPE NUMBER THAT RECEIVED THE DATA
60 FOR N= 1 4                     # SET UP THE LOOP COUNTER FOR 4 BYTES
65 PRI "D:" ; PRX READ N          # PRINT THE DATA RECEIVED
70 NEXT                           # LOOP SOME MORE
75 PRI "----------~"              # PRINT A PERFORATION
80 GOTO 25                        # LOOP

5  REM nRF24L01 MULTICEIVER-TRANSMITER ON PIPE 3'S ADDRESS
10 C:=0                           #INIT PAYLOAD COUNTER
15 RFI 1 0 1                      # POWER UP IN TX MODE, CHANNEL 0, 2 MBPS
20 RFA 0 $C2 $C2 $C2 $C2 $C3      # SET PIPE 0 TO MAC FOR PIPE 3 (for ShockBurst)
25 RFA 7 $C2 $C2 $C2 $C2 $C3      # SET TX TO MAC FOR PIPE 3
30 RFW 4 $13                      # ADJUST AUTO-REXMIT DELAY TO 500US
35 RFF                            # FLUSH RX AND TX BUFFERS
40 PRINT "PAYLOAD " ; PRI C       # PRINT A HEADER
45 RFT RND RND RND RND            # TRANSMIT RANDOM DATA
50 C:= C + 1                      # INCREMENT PAYLOAD COUNTER
55 SLP 7 ; GOTO 40                # SLEEP 2 SECONDS AND LOOP
```

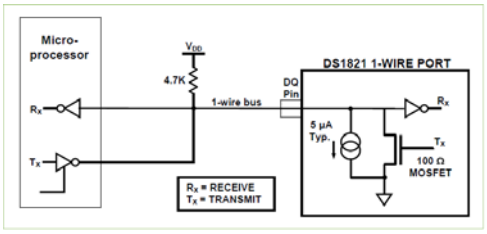
Recovering from errors:
The nRF24L01(+) seems to be very reliable in terms of data robustness. Depending on the data rate, channel selected, transmitter power level and noise in the 2.4GHz ISM band, transmission and reception errors may occur.

In particular is during transmission of a packet. If a receiver did not receive a packet and the MAX_RT flag is set in the "STATUS" register, then that last packet is "stuck" in the FIFO buffer, effectively blocking any packets scheduled for transmission after it. There are two (2) ways to resolve this problem:

1. Execute the `RFF 1` command and resend the failed payload, including the payloads that were "stuck" in the FIFO buffer behind the failed payload. This method has the disadvantage that one needs to keep track of the payloads to resend in case they are lost due to the execution of the `RFF 1` command. An advantage is that it is easy to do with few commands.
2. Monitor the status of the `RFT` command and take the appropriate action to (continually) resend the failed payload thereby releasing the other payloads "stuck" behind the failed payload when the failed payload is finally successfully received. This method has the disadvantage that if one is continuously sending data to a receiver and the receiver failing to receive the payload never becomes able to receive, then a continuous software loop may be executed resulting in a software "lockup". An advantage is that under AttoBASIC's control, longer delays of an unavailable receiver can be tolerated.

   One way to handle this is to implement a delay within a software loop counter and exit when the predetermined count is exceeded. It is worth while to note that this is exactly what the nRF24L01(+) does using the "ARD" and "ARC" bits in register $04, "SETUP_RETR" using hardware instead of software. Of course, setting the "ARC" bits in register $04 to $15 (using the `RFW` command) would increase the number of retries to 15 instead of the default of 3 would accomplish this as well.

DALLAS 1-Wire® Support (Version 2.31+)



As of version 2.31, AttoBASIC supports the DALLAS/MAXIM 1-Wire® protocol using three (3) primitive commands; The `0WI` command resets the 1-Wire® bus and detects the presence of devices, the `0WW` command writes data to a device on the 1-Wire® bus while the `0WR` command reads data from a device on the 1-Wire® bus.

Note that AttoBASIC enables the AVR port pin's pull-up to supply a positive potential for the bus. This author has not encounters data corruption problems using only the port pin pull-up but design guidelines suggest using a separate 4.7 KΩ pull-up in addition to the port pin pull-up.

The use of the `0WI`, `0WW` and `0WR` commands is straight-forward as the following example program illustrates being used to read the temperature from a DS1821 temperature sensor:

```
5  REM DS1821 TEMPERATURE READ
10 IF 0WI = 0 THEN GOTO 40              # initialize bus and test for device present
15 0WW $EE                              # start a temperature conversion
20 0WW $AA                              # initiate a read
25 PRINT "Temperature: " ; PRINT 0WR    # print the temperature read
30 SLP 9 ; GOTO 10                      # sleep for 8 seconds and loop forever
35 END
40 PRINT "NO DEVICES!~" ; END           # inform user of error and end
```

Another method of transferring data to a 1-Wire® device makes use of the DATA and READ statements as the following example illustrates writing data to the scratchpad RAM of a DS2431 EEROM.

```
5  REM DS2431 Write SP Test
10 DATA $CC $0F $20 $00 1 2 3 4 5 6 7 8        # Setup commands and data to write
```

```
15 IF OWI = 0 THEN PRINT "No Devices!~" ; END  # Reset bus, test for device and
                                               #   error if none detected
20 for N= 1 RES ; OWW REA ; NEXT               # Write loop all data, REST returns
                                               #   DATA buffer size ( # of elements)
25 PRX OWR ; PRX OWR                           # Print the CRC-16 returned by the device
30 END
```

A variation of the **OWR** command allows one to transfer the data read from a 1-Wire® device into the DATA statement's data buffer by specifying the number of bytes to read from the device. The data can then be read with the READ command. The following example program illustrates this concept by reading the 8-byte ROM code from a DS2431 EEPROM device:

```
5  REM
10 IF OWI = 0 THEN GOTO 45       # initialize bus and test for device present
15 OWW $33 ; OWR 8               # request ROM code read and transfer 8 bytes from
                                 #   the device to the DATA statement buffer
20 FOR N= 1 7                    # initiate a loop counter
25 PRINT "Data: " ; PRINT READ   # print each data value
30 NEXT                          # continue looping for the 1st seven data bytes
35 PRINT "CRC: " ; PRINT READ 7  # then print the CRC8
40 END
45 PRINT "NO DEVICES!~" ; END    # inform user of error and end
```

Although AttoBASIC supports multi-drop 1-Wire® buses, there is no direct provision to execute a "Search ROM" function to isolate and detect specific devices on the bus. Therefore, each device's address code should be known before attempting to communicate with any specific device. This can be accomplished by attaching one device at a time to the bus, executing a "Read ROM" command then logging the resulting ROM code for each device.

Determining the Answer to the Ultimate Question of Life (Version 2.31+)

As of version 2.31, AttoBASIC calculates in real-time, to exact precision the answer to life, the universe and everything. Don't Panic, the answer correlates to that provided with infinite majesty and calm by *Deep Thought*. The following example command in direct mode illustrates this. The **WTF** command may also be used in a program and the answer assigned to a variable.

```
PRINT WTF                        # Print's the answer to life, the universe
                                 #   and everything.
```

Using nested FOR-NEXT and GOSUB-RETURN (Version 2.30+)

As of version 2.30, AttoBASIC supports nested loops and subroutine calls up to four (4) levels deep. GOSUB's can be embedded within a FOR-NEXT loop. The following sample program illustrates nesting of FOR-NEXT loops:

```
10 Z:=0                # Zero a counter
15 FOR I=1 2           # 1st level loop setup
20 FOR J=1 3           # 2nd level loop setup
25 FOR K=1 4           # 3rd level loop setup
30 FOR L=1 5           # 4th level loop setup
35 PRI "I=" ; PRINT I  # Print value of I
40 PRI "J=" ; PRINT J  # Print value of J
45 PRI "K=" ; PRINT K  # Print value of K
50 PRI "L=" ; PRINT L  # Print value of L
55 Z:=Z+1              # Increment the counter
60 NEXT;NEXT;NEXT;NEXT # LOOP value increment for each level
65 PRI "--------~"     # Print a perforation
70 PRI "Z=";PRI Z      # Print the cumulative total, Z = ( 2 * 3 * 4 * 5 )
75 END
```

The following sample program illustrates nesting GOSUB-RETURN subroutine calls:

```
10 GOSUB 20                          # Call the 1st level subroutine
15 END                               # End the program
20 PRINT "L1~" ; GOSUB 25 ; RETURN   # Print "L1", call 2nd level subroutine, return
25 PRINT "L2~" ; GOSUB 30 ; RETURN   # Print "L2", call 3rd level subroutine, return
30 PRINT "L3~" ; GOSUB 35 ; RETURN   # Print "L3", call 4th level subroutine, return
35 PRINT "L4~" ; RETURN              # Print "L4" then return
```

Using DATA, READ and RESTore (Version 2.30+)

As of version 2.30, AttoBASIC supports storing up to eight (8) 8-bit values in RAM using the **DATA**, **READ** and **REST**ore commands. The intended use is to store constants or elements in a look-up table for later retrieval and use. Since the data is stored in RAM, it remains untouched unless over-written by another **DATA** statement or if the nRF24L01 routines are enabled, execution of the **RFX** command, whether executed within a program or in immediate mode.

The following sample program illustrates use of the DATA, READ and RESTore commands. The sample program creates a DATA table then the FOR-NEXT loop READS a value and prints it:

```
10 DATA 10 20 30 40    # Store four values for later
20 FOR N=1 4           # Start FOR-NEXT loop at 1, end at 4
30 A:= READ : PRINT A  # Assign data table value to variable A then print it
30 NEXT                # End of FOR-NEXT loop
40 REST                # RESTORE resets the DATA pointer to the beginning
50 GOTO 20             # Go back and do it some more
```

An alternative use allows one to index directly into the DATA table as an array by supplying a parameter with the READ statement. The same sample program creates a DATA table then the FOR-NEXT loop READS the value indexed by the variable N and then prints it. Note that the DATA table pointer is "zero inclusive", meaning that the 1st data element is at position "0" and the 8th at position "7":

```
10 DATA 10 20 30 40    # Store four values for later.
20 FOR N=0 3           # Start FOR-NEXT loop at 0, end at 3.
30 A:= READ N          # Assign data table value indexed by N to variable A
40 PRINT A             # then print it.
40 NEXT                # End of FOR-NEXT loop
50 REST                # RESTORE resets the DATA pointer to the beginning
60 GOTO 20             # Go back and do it some more
```

There may be circumstances where one may wish to store the results of other AttoBASIC commands in a DATA table. For instance, readings from desired channels of the ADC. As is typical of many AttoBASIC commands, directly assigning the results of such commands to a DATA statement will produce the desired results:

```
10 DATA ADC0 ADC1 ADC2# Assign the results of ADC0, 1 and 2 to the DATA statement
```

Although there is no real advantage, an alternative method involves using the variables **A..Z** for intermediate storage. The following program statements create a DATA table from the readings from the desired channels of the ADC:

```
10 A:=ADC0 : B:=ADC1 : C:=ADC2   # Store the readings in variable first
20 DATA A B C                    # Now store the results in the DATA table
```

EEPROM Support (Version 2.30+)

As of version 2.30, AttoBASIC adds language extensions to access the AVR's on-chip EEPROM using the EER and EEW commands. Also as of version 2.30, AttoBASIC supports using the AVR's on-chip EEPROM as a file system (EFS) to save and load user programs. This includes support for the self-start feature, which will load a program from file number "0".

When the EFS builds are used, all but 16 bytes of the EEPROM are used to support the EFS. However, if need be, the unused 16 bytes can be used to store constants and configuration information. They are accessed at locations 16 ($10) through 31 ($FF).

The following sample program illustrates use of the EER and EEP commands. The sample program reads a temperature sensor on ADC channel 0 and multiplies a calibration constant to the reading before displaying the value:

```
10 A:= $10 : EEW 3 A   # 1st byte of user accessible EEP, store 3 to it.
20 0:= EER A           # store the value in the address held in A to 0
30 ADR 0               # Select the ADC's internal reference
30 PRI EER A * ADC 0   # print the value on ADC channel 0 * the calibration constant
```

```
40 SLP 9 : GOT 30     # wait 8 seconds and loop forever
```

EEPROM File System – structure (Version 2.30+)

The EFS takes a rather simple approach in its implementation. It does not support file names but rather file numbers or "handles". The EEPROM is divided into 32 byte blocks and uses an 8-bit pointer to address each block. The 8-bit block pointer yields an 8KB addressable range, which is currently not a size supported even on AVR devices such as the Atmega2560, which has 4KB of EEPROM.

Block 0 is designated as the file index "header" block and does not contain user program data, only the starting block of each file handle. The number of file handles is dependant on the target MCU's amount of EEPROM divided by 256 with a maximum of eight (8) file handles. The file handle, being a number between 0 and 1, 3 or 7 inclusive, allows one to easily specify the file number as a parameter with the save or load command. Without a file number is the same as file "0" so that the self-start feature's code remains unmodified.

As stated, each file handle uses a single byte per file handle so a maximum of 8 bytes of EEPROM is used for the "header". Each file's header byte contains "0" if it unused or the block number of the 1st block of data. Each 32 byte block uses 2 bytes of "overhead" dedicated to keeping track of the blocks allocated to each file. The 1st byte of a data storage block specifies the type of block, either 0x01 for data or 0xFF for unused. The 2nd through 31st byte contains the program data and the 32nd byte contains the pointer to the file's next used block ("link list") or "0" if it is the last block of the file. Thus 30 bytes per block are available for program storage.

On a 512 byte EEPROM, the file system uses 62 bytes of overhead or 12.1% (450 bytes free). For a 1K byte EEPROM, the file system uses 94 bytes of overhead or 9.2% (930 bytes free) and on a 4K byte EEPROM, the file system uses 286 bytes of overhead or 6.9% (3810 bytes free).

In addition to saving and loading user programs, the EFS supports "cataloging" the contents of the file system using the CAT command. The CAT command displays a list of each file number, the file's size and the 1st line of the program. The total bytes in use and remaining are also printed.

To facilitate "initializing" the EEPROM for use with the EFS, the INIT command is used. Invoking the INIT command insures that the file system structures are properly formatted. It should be invoked on a freshly programmed AVR where the EEPROM is likely in an erased state, or on an AVR that was used on another project where the EEPROM may have "old data". If so desired, the EEPROM may be bulk-erased with the EEW command before using INIT.

Since the CAT command displays the file's 1st program line number, it may be helpful to optionally make the 1st line number of a program bear some sort of short description. Therefore, the REM command can be used to allow the 1st line of a program to become a file description. Using the REM command does, however, use precious program memory.

As with all file systems, the EFS allows individual files to be erased using the ERA command, thereby freeing a files blocks for reuse by the file system.

The EFS is disabled on the Mega88 due to code size but it is available on the Mega168 even though there is only 512 bytes of EEPROM.

DHTxx Humidity and Temperature Sensor (Version 2.22+)

As of version 2.22, AttoBASIC adds language extensions to support the DHT21/22 series of low-cost humidity and temperature sensors. Connection to the AVR is simple, VCC (>= 3.3V), GND and DATA. An external pull-up resistor on the DATA line is optional as AttoBASIC enables the pin's internal pull-up resistor, which is sufficient for short cable lengths.

Since the DHT interface commands take advantage of the AVR's pin-change interrupts, the DATA pin must be connected to a port pin that supports pin-change interrupts. For the Mega88/168/328 (ARDUINO©), all ports support pin-change interrupts, while only PORTB supports pin-change interrupts on the ATmega2560, ATmega32U4 and AT90USB1286.

Per the DHT2x datasheet, consecutive readings should be taken no less than 2 seconds apart. If an attempt is made to take another reading in less than 2 seconds, the DHT and DHH commands will simply return the prior reading.

The status can be checked using the DHR command, which will return "1" when the DHT is "busy" and "0" when "ready".

The sign is ignored when taking readings in Fahrenheit. However, when taking readings in Celsius, the sign is recognized and an offset of 128 is added to the temperature reading. Testing for and subtracting 128 from the reading will yield the temperature below 0° degrees Celsius.

```
10 DHS 1                          # set temperature readings to include the sign
15 PRINT "%RH: " ; PRINT DHH      # print the leading string then the humidity
20 PRINT "*C : " ; PRINT DHT 1    # print the leading string then the temperature in C
25 SLP 7                          # sleep for ~2 seconds
30 PRINT "BUSY: " ; PRINT DHR     # print the leading string then the status
35 PRINT "=================~"    # print a separator
40 GOTO 15                        # loop for more readings
```

Low-power SLEEP till interrupt (Version 2.20+)

As of version 2.20, AttoBASIC adds language extensions to support the AVR's "sleep" instruction using the "SLP" command. Upon execution of this command, AttoBASIC enters the low-power sleep mode and awaits a hardware interrupt event. The AVR's "idle mode" is selected so the peripheral clocks are still running. In this way, any supported interrupt source may be used as the event trigger. Proper use of the "SLP 0" command requires that the user enable the desired interrupt source(s) before execution. The **POKE** statement allows one to program the desired Interrupt Control Register and Interrupt Mask Registers to setup the desired interrupt source.

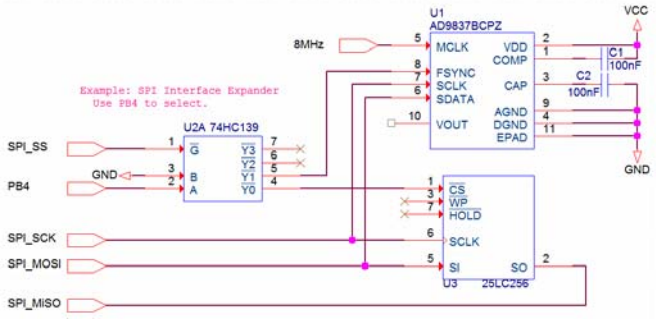
In addition to having the ability to use a hardware interrupt as an event source, the user may also use the AVR's WATCHDOG timer as the interrupt source. When invoked in this manner, the watchdog timer is implemented in **"Interrupt Mode"** where the WATCHDOG timer's prescaler value is the value of the parameter passed with the SLP command (excluding "0") allowing a pre-determined delay to be executed before exiting the SLP command. Keep in mind that the AVR's watchdog timer is driven by the internal RC oscillator and has an inherent error associated with it, therefore the delay duration may be inaccurate.

The following sample program illustrates use of the SLP instruction to sample analog channel 0 every second and print the results to the console:

```
10 ADR 1            # select external Vref source
20 SLP 6            # sleep for 1 second
30 PRI ADC 0        # print the value on ADC channel 0
40 GOT 20           # loop forever
```

External EEPROM for data recording (Version 2.20+)

As of version 2.20, AttoBASIC adds language extensions to support the ability to use an externally connected serial EEPROM (such as MICROCHIP 25LC256) as a data file using the "Data File" commands. In analog or digital values at pre-determined intervals and ne.



AVR. A device with up to 64K bytes of storage is dected to the SPI pins, then the ability to use another SPI lware and software are implemented. The circuit shown

allows more than one SPI device to be used with the Data File and SPI commands. Using a 74HC139 decoder will support up to four (4) devices while using a 74HC138 will support up to eight (8) devices.

The following sample program illustrates implementing an additional device, an AD9837 DDS generator, while using the data file commands:

```
10 ADR 1; DFI 0 4                  # Vref is ext, init 1st 1k (4 pages) of data file
11 SPM ; RTI 1                     # init SPI interface to mode "0", RTC @ 10mS
12 SDB 4                           # PB4 as output
13 SBB 4; SPW $21; SPW $00; SPS 1# select U1, reset the DDS chip, raise SS pin
20 SLP 9; SLP 7                    # sleep for 10 (8+2) seconds
21 CBB 4                           # clear PB4 to select U3 (datafile)
22 DFL PEEK VPG@RTC1               # store lower 16-bits of RTC (time-stamp)
23 DFL PEEK VPG@RTC0
25 DFL ADC 0; DFL ADC 1            # store the value of ADC channel 0 and 1
30 SBB 4                           # set PB4 to insure U1 is selected
31 SPW $40; SPW $A8                # set the frequency register on the DDS chip
32 SPW $40; SPW $00
33 SPW $20; SPW $68; SPS 1         # enable the DDS chip and raise the SS pin
40 GOT 20                          # loop forever
```

Real-time Counter (Version 2.20+)

As of version 2.20, AttoBASIC adds language extensions to support a 32-bit Real-time Counter (RTC) implementation much like that of the ARM processor cores. This feature was added as an enhancement to the aforementioned *Data File* commands so that one can print relative time-stamp information to the console or the data file when logging data. The internal resolution of the RTC is 1mS at power-up and hardware/software reset thus the RTC increments 1000 times per second yielding a duration of 49.71 days before a counter roll-over occurs. The *RTI* command allows one to set the increment rate to 1mS, 10mS or 100mS thus the RTC increments 1000, 100 or 10 times per second, yielding a duration of 49.71, 497.1 or 4971 days respectively.

The following sample program illustrates a simple data-logging to the serial console using the RTC to time-stamp the data:

```
10 ADR 1; RTI 1                    # Vref is external; RTC increment to 100mS
20 FOR I= 1 10; DEL 100; NEX       # 10 iterations of loop for 10 seconds delay
25 RTP                             # print the RTC as a time-stamp
30 PRINT ADC 0; PRB PINC           # print value of ADC 0 and PINC (on separate lines)
40 GOTO 20                         # loop forever
```

The lower 8-bits of the RTC can be directly assigned to a variable or if one needs access to more digits of the RTC, using the PEEK statement will accomplish this as the following sample program illustrates:

```
10 A:= PEEK VPG@RTC3 ;B:= PEEK VPG@RTC2    # RTC registers MSB to LSB
20 C:= PEEK VPG@RTC1; D:= PEEK VPG@RTC0
30 PRX D; PRX C; PRX B; PRX A              # Print all 32 bits in hex
```

If one needs to directly read or modify the RTC registers, the PEEK and POKE commands can be used to accomplish this. Refer to the VPG, @ and RTC[N] commands.

SPI interface (Version 2.0+)

As of version 2.00, AttoBASIC supports the AVR's SPI interface in Master mode at speeds of up to (F_clk / 2). The command set allows selection of the Master mode sample edges and the data order (MSB or LSB first). Usage of the interface is fairly straight forward as the following sample program illustrates:

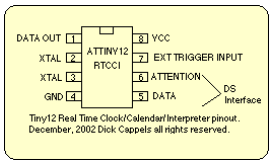
```
10 SPM 2; SP0 1                    # init the SPI interface in Master mode 2, MSB 1st
15 SPW $21; SPW $58                # write $21 and $58 to the slave
20 PRI SPR                         # print the data read from the slave
25 SPS 1                           # deselect the slave by raising the SS pin
```

TWI (I^2C) interface (Version 2.0+)

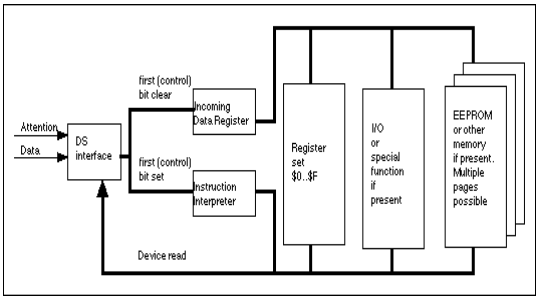
As of version 2.00, AttoBASIC adds language extensions to support the AVR's TWI (I^2C) interface in polled mode at bus speeds of up to 400Kbps. Usage of the interface is fairly straight forward as the following sample program (with no bus error check) illustrates:

```
10 TWI 0                          # init the TWI interface at 400kbps
15 TWS: TWA $58                   # assert a START condition and address the slave
20 TWW $00: TWW $FE               # write $00 and $FE to slave
25 TWP                            # assert STOP condition to signal end
```

DS Protocol using a two-wire interface (Version 1.0+)



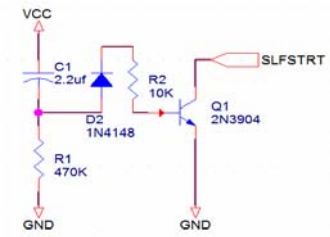
The DS Protocol (Data transfer via handshake) is a firmware-based protocol for low priority data transfer between a host and a slave processor with limited resources.



The DS protocol was designed to provide firmware-based bidirectional host-to-slave inter-processor communications for situations in which no hardware solution is available (TWI, SPI, CAN. 1-WIRE®, etc.) and the host and/or slave are incapable of tending to the interface in real-time. Designed to be a chip-to-chip (within the same device) protocol, the only specialized hardware required is two (2) bidirectional I/O ports on each chip, or alternatively two (2) input ports and two (2) tristate-able output ports may be used.

AttoBASIC V1.x has always directly supported the DS protocol. Version 2.00 has supported the DS protocol when the routines are enabled. Starting with Version 2.30, when the routines are enabled, the DS command set detects and reports a non-responsive DS slave. The DS protocol was designed and developed by Dick Cappels. A complete technical description, sample programs and complete projects based on the DS protocol may be found his web site at: http://www.cappels.org/dproj/dspage/ds.htm

Self-Start Feature (Version 1.0+)



AttoBASIC supports a "self-start" feature wherein if the designated SLFSTRT pin is held "low" for at least 250mS during power-up, the contents of the EEPROM will be loaded and executed by the interpreter. A program started this way may be stopped with a "CTRL-C" keyboard combination received via the serial console input. A weak internal pull-up is applied to the SLFSTRT pin before testing it. Once the pin is tested, the weak pull-up is released and it can be used for analog or digital I/O purposes. One might implement a simple transistor circuit as shown to create a short-duration "low pulse" on the SLFSTRT pin at device power-up. The extra diode on the base is to raise the base-emitter bias turn-on voltage to ~1.5 volts.

Renumbering AttoBASIC Programs

As one programs and debugs under AttoBASIC, additions and deletions of program lines can become somewhat difficult to read. Most programmers prefer to “beautify” their programs in some manner. As yet, AttoBASIC does not provide a native program renumbering command.

However, if one uses a text editor to write programs and uploads them through their host OS’s terminal emulator, then an external renumbering utility can be used to “beautify” before uploading. Be sure to type **NEW** before uploading the newly renumbered program or left-over program lines may cause errors when the program is run.

Included with the AttoBASIC source code is a renumbering program that has been compiled to run as a command line program under Linux and WINDOWS™. The utility(s) are located in the "*_Applications/Saved_Programs/Renumber/*" folder. The Linux version is named "**renumber**" and the WINDOWS™ version is named "**renumber.exe**".

The syntax is the same for each operating system and is as shown below; where `infile` is the original file to be renumbered, `outfile` is the file to be written to with the renumbered program, `start#` is the desired starting number and `increment` is the desired increment between program line numbers. `start#` and `increment` are optional and default to 5 and 5 if not provided.

```
renumber [infile] [outfile] [start#] [increment]
```

An example is as follows:

```
renumber TestFile.txt _TestFile.txt 10 5
```

Programming idiosyncrasies

1. As of Version 2.20, the relational operators, =, != (<>), > and < return "1" if the test is true, and "0" if the test is false. This is opposite from prior versions wherein a "0" was returned if the test is true, and "1" if the test is false.

2. IF-THEN structures can be used with the GOTO or GOSUB commands or multiple commands can be concatenated on a single program line number using the " ; " separator. There are two forms of structures that can be used, as follows:
   - All on one line, such as;
   ```
   10 IF A = B THEN GOTO 100           # if A = B then jump to line 100
   20 [next statement]                 # execute if A != B

   10 IF A = B THEN PRINT A ; PRINT B  # if A = B then print A then B
   20 [next statement]                 # execute if A != B
   ```
   - If the THEN keyword is omitted on the line containing the IF keyword and the keyword to be executed if the condition tests true is moved to the next program line, then that next program line (following the IF keyword) only executes if the condition in the previous program line is true, as in:
   ```
   10 IF A = B                                 # compare; A = B ?
   20 GOTO 100 [true statement]                # execute if A = B
   30 GOTO 200 [not true statement]            # execute if A <> B

   10 IF A = B                                 # compare; A = B ?
   20 PRINT A ; PRINT B  [true statement]      # execute if A = B
   30 PRINT "Not Equal~" [not true statement]  # execute if A <> B
   ```

3. AttoBASIC passes data between commands using a "stack-based" approach (much like FORTH). Therefore, whether in a program or in direct mode, commands are processed sequentially from right to left. In other words, any command that returns a value will return it's value as a parameter to the previous command. Variables are handled differently in AttoBASIC and thus can exist peacefully on the left side of a relational operator command. Results from hardware-orientated command cannot. An example or two is in order to demonstrate this concept and point out some tricks that can be used to accomplish certain tasks.

   In the following example, the inverted value of the 8-bit CRC is assigned to the variable **B** as the value returned by the **CRC** command becomes a parameter for the **COM** command, which returns a value to the assignment command (**:=**).

   ```
   B:= COM CRC $11 $22 $33 $44          # assign the inverted value of the CRC-8
   ```

   In the following example, if the value returned from the **RND** command is less than 128 then the

word “Low” is printed, otherwise, the word “High” is printed. The value returned by the **RND** command is used by the greater-than relational operator command “>”.

```
10 IF 128 > RND THEN PRINT “Low~” ; GOTO 10   # print “Low” if RND < 128, loop
20 PRINT “High~” ; GOTO 10                    # print “High” if RND > 128, loop
```

In the above example, if the value returned by the **RND** command was on the left side of the greater-than relational operator, then AttoBASIC would have produced and error stating that there was no value available because the value returned by the **RND** command would not be available on “the stack” for the greater-than relational operator to use.

In the following example, a reset is applied to the 1-Wire® bus and the presence of a device is tested (using the **OWI** command). If no device is detected then print an error, otherwise, write some commands to the device and print the data read from it, looping for more.

```
10 IF 0 = OWI THEN GOTO 40       # if there are no 1-Wire® devices, goto error
20 OWW $CC $AA ; PRX OWR         # write $CC and $AA to the 1-Wire® device
                                 # and print the value
30 SLP 6 ; GOTO 10               # delay for 1 second and loop
40 PRINT “No Device!~” ; END     # inform user and exit
```

4. The result of a complicated relational operator test can be broken down by assigning intermediate values to variables first then the final result can be tested using these variables. The following program tests for PORTB’s pin 4, 5, 6 and 7 all being “1”. Of course, there is an easier way to accomplish this task.

```
10 A:= INB6 = INB7           # Assign the result of INB6 = INB7 to A
20 B:= INB4 = INB5           # Assign the result of INB4 = INB5 to B
30 C:= A = B                 # Assign the result of A = B to C
40 IF C = 1                  # Test is INB4 = INB5 = INB6 = INB7
50 PRINT “All high!~”        # If true, execute this statement
60 PRINT “All low!~”         # If false, execute this statement
```

5. The "$" is an operator and must be formally delimited from other operators such a +, -, =, <, !, etc.
6. The "TO" statement is included as a convenience, therefore the "TO" statement may be replaced with one or more spaces, thus the following program lines are equivalent:
   - `FOR A = 1 TO 9; NEXT`
   - `FOR A = 1 9; NEXT (implied "TO")`
7. A RETurn command without a corresponding GOSub has the same effect as the END statement in that program execution will stop.

The following table lists various common errors and the reason for the error along with acceptable usage examples:

Acceptable Usage	Erroneous Usage	Reason for error
`A := 255`	`A := 256`	Exceeds the value of an 8-bit number.
`FOR A =`	`FORA =`	AttoBASIC sees this as a command named "FORA", which is undefined.
`A:= $3A`	`A:=$3A`	AttoBASIC requires a space between the "=" and "$" characters.
`PRINT A`	`PRINT A, B`	AttoBASIC will only print the last value on a line.

```
10 IF 1 = OWI THEN GOTO 30
20 PRINT “FALSE~” ; END
30 PRINT “TRUE~” ; END
```

```
10 IF 1 = OWI
20 PRINT “TRUE~” ; END
30 PRINT “FALSE~” ; END
```

```
A:= RND                                                  No error.
A:= $10
A:= $10+10
A:= A + $10
A:= COM CRC 1 2 3 4
A:= PEEK VPG@DFS0
B:= A > $80
C:= $0A (or $A)
PRINT 1 – 2 – 6 [as 1–(2–6)]
PRX A+B
PRX $2A
PRX $F + 10
PRX $2A + 9 + $C + 21
PRI 10 COM or PRI COM 10
PRI PEEK VPG@RTC0
DATA 1 2 3 4 5 6 7 8
DATA A B C D E F G H

GOTO 25
GOTO $20                                                 Valid syntax but not necessary
GOTO A
GOTO A+B
GOTO 2 + 2 + 5
```

Rolling your own customized version of AttoBASIC (Version 2.00+)

Starting with AttoBASIC version 2.20, conditional assembly was added to support the building of an assortment of HEX files for a selection of MCU's, clock speeds and console I/O media type. There may however, arise a desire or need for one to extend AttoBASIC or perhaps use a different clock speed than what is supplied in the pre-built HEX files. Thus the need to perform a custom assembly.

This section will explain the details of how to accomplish such a task. It is assumed that the reader has the development software/equipment and is familiar with:

- Programming in assembly language on the ATMEL© 8-bit AVR® series of controllers
- Using ATMEL's AVR STUDIO **4.19**

The directory structure of the AttoBASIC project directory is as follows:

```
AVR_Specific_Builds   ; contains the MCU specific build HEX files
Include               ; contains the subroutine modules, various definitions and data
_Applications         ; contains sample programs
_LUFA-110528          ; contains Dean Camera's LUFA bootloader.
_USB_Drivers          ; contains the WINDOWS® USB drivers for LUFA and USB SIO
_USB_Serial           ; contains the source code for PJRC's USB serial I/o
_Bootloaders          ; contains the source code and HEX files for various USART bootloaders
```

For the most part, the reader need not be concerned with these subdirectories as the root project folder contains the AVR STUDIO project file, named "*AttoBASICVxxx.aps*", where "xxx" is the version number multiplied by 100. Once the project is opened in AVR STUDIO, the main assembly file, "*AttoBASICVxxx.asm,* is displayed along with a file from the "*Include*" subdirectory named "*Defs_Constants.inc*". "*Defs_Constants.inc*" is the file where the specific options relating to MCU, clock speed and module ex/inclusion is performed.

Selecting an MCU:

The supported MCU's are specified in the part definition files. Simply comment/uncomment the target MCU's definition file.

Selecting a bootloader:

One may wish to build with or without a bootloader. The line below enables or disables the bootloader in the build. The bootloader used is determined by the MCU defined.

```
#define BTLDR 1       ;"1" to include boot loader code.
```

Selecting USB or USART serial I/O:

One may wish to build with or without USB support. The line below enables or disables USB and USART support. If USB is enabled, the USART support code is disabled and all serial I/O is passed to the USB serial I/O routines. Note that USB is only available on MCU's with a hardware USB controller. If mistakenly selected for a non-USB MCU, the USB support code is disabled and the USART support code is used instead.

```
#define USB 1          ;"1" to include USB support code
```

Selecting a TEENSY2.0++ target:

One may wish to build for a TEENSY2.0 ++ target. The line below enables or disables support for the TEENSY2.0++ product, particularly the "*HALFKAY loader*". Note that selecting this option disables the bootloader code, enables the USB serial I/O support and sets the MCU clock to 16MHz.

```
#define TEENSY 0       ;set to "1" to enable TEENSY++ 2.0 build
```

Selecting the main clock speed:

The line below defines the main clock speed derived from either the internal resonator or an external source. It is not necessarily the MCU's core clock speed as the MCU's core clock speed is derived from this value and the FCLK_PS described below.

```
#define FCLK (8000000/FCLK_PS)   ;set Fclk (preprocessor variable)
```

Selecting the MCU's clock prescaler:

Although unlikely, one may wish to use the MCU's system clock pre-scaler to divide the main clock to drive the target MCU's core. The line below sets the system clock pre-scaler register of the target MCU to the value specifies. It also sets the target MCU clock to the value of FCLK divided by the value of FCLK_PS so that the proper timing and baud rate is used.

```
#define FCLK_PS  1          ;System clock prescaler
```

Assigning a version number to the build:

It is good practice to assign a version for target builds that stray from the main target versions. The version number is reflected in AttoBASIC sign-on message. The format of the version number is [Mj].[Mn][Sb], where [Mj] is the majot version, [Mn] is the minor version and [Sb] is the subversion of the build. The following line in "*Defs_Constants.inc*" sets the value of the build's version. Note that the version is left justified by two (2) character positions (multiplied by 100).

```
.equ    Version = 230    ;version 2.30 (100x so its an integer)
```

Enabling and disabling support modules:

For all MCU's with FLASH memory of 32KB of larger, all software support modules are enabled. Due to FLASH size constraints on the 8KB and 16KB Atmega88 and ATmega168 parts, certain software modules have been disabled. Refer to the section entitled "*Additional modules enabled for default HEX file builds*" for specifics.

Enabling and/or disabling of specific software modules is performed within the "*Defs_Constants.inc*" file. For all MCU's having 32KB or more of FLASH, the line containing the text `";~~~~~ Code enabling feature ~~~~~~"` is the starting point. Simply set the modules name to "1" to enable a module or a "0" to disable a module. The example below enables the DDS and DHT routines while disabling the SPI routines.

```
.set  DDS        = 1        ;"1" to enable DDS routines (+710 bytes)
.set  DHT        = 1        ;"1" to enable DHTxx routines (+886 bytes)
.set  SPI        = 0        ;"1" to enable SPI routines (+260 bytes)
```

For the Atmega88 and Atmega168, the line containing the text `";~~~~~~~ MCU Specific Restrictions ~~~~~~~~~~~~~~~"` is the starting point to further disable modules for those classes of MCU. As

before, simply set the modules name to "1" to enable a module or a "0" to disable a module. The example below counters the DDS and SPI settings from above by re-enabling the SPI routines and disabling the DDS routines, while leaving the setting of the DHT routines alone.

```
.set  DDS         = 0         ;"1" to enable DDS routines (+710 bytes)
.set  SPI         = 1         ;"1" to enable SPI routines (+260 bytes)
```

Notes:

1. Enabling some modules will inherently enable other modules for underlying support. An example is the nRF24L01(+) module enables the SPI module for support.
2. The byte values for each module are typically for an Atmega2560, which has 3 byte calls and returns and generally uses a larger code footprint than the MCU's with 64KB or less FLASH. Thus, for other MCU's, the values will likely be less than those listed in the "*Defs_Constants.inc*" file so always insure that the FLASH memory capacity is not exceeded.

Building the project:

Once the desired options for the target build are selected, the AttoBASIC project should build without error. Care was taken to catch contradictory or invalid build options. The resultant assembly listing will be available in the file "*AttoBASICVxxx.lst*" and the HEX file will be "*AttoBASICVxxx.hex*".