

# AttoBASIC Application Notes

## Version 2.34+

### Table of Contents

AttoBASIC Overview.....	3
Console I/O.....	4
Power supply voltage .....	4
Notes about the example programs.....	4
Default MCU Pinouts for Hardware Support Functions .....	5
Language Extension Modules enabled for default HEX file builds .....	6
The traditional "Hello World" program .....	7
ATtiny84 and ATtiny85 Support (Version 2.34+) .....	7
ATtiny2313 Support (Version 2.34+) .....	8
AT90S8515 and ATmega163 Support (Version 2.34+).....	8
nRF24L01(+) RF Transceiver Support (Version 2.31+).....	8
DALLAS 1-Wire® Support (Version 2.31+) .....	11
Determining the Answer to the Ultimate Question of Life (Version 2.31+).....	12
Using nested FOR-NEXT and GOSUB-RETURN (Version 2.30+) .....	12
EEPROM – Non-volatile configuration (Version 2.34+) .....	13
EEPROM – Reserved Application Space (Version 2.34+).....	13
Using DATA, READ and RESTore (Version 2.30+) .....	13
EEPROM Support (Version 2.30+) .....	14
EEPROM File System – structure (Version 2.30+) .....	14
System Clock Prescaler and internal RC oscillator adjustment (Version 2.34+).....	15
DHTxx Humidity and Temperature Sensor (Version 2.22+) .....	16
Low-power SLEEP till interrupt (Version 2.20+).....	16
External EEPROM for data recording (Version 2.20+).....	17
DDS - Direct Digital Synthesis (Version 2.00+).....	17
Real-time Counter (Version 2.20+) .....	18
SPI interface (Version 2.00+).....	18
TWI (I <sup>2</sup> C) interface (Version 2.00+) .....	19
DS Protocol using a two-wire interface (Version 1.0+) .....	20
Self-Start Feature (Version 1.0 to 2.33) .....	21
Self-Start Feature (Version 2.34+).....	21
Saving program space by compressing characters .....	21

# AttoBASIC Application Notes

## Version 2.34+

Renumbering AttoBASIC Programs.....	22
Programming idiosyncrasies.....	22
Rolling your own customized version of AttoBASIC (Version 2.00+).....	25
Interfacing with AttoBASIC on a Linux platform (Version 2.00+).....	27

# AttoBASIC Application Notes

Version 2.34+

## AttoBASIC Overview

AttoBASIC is a hardware-oriented “byte-wide” BASIC for the ATMEL 8-bit AVR microcontroller series. It 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 2<sup>nd</sup> generation of ATMEL’s AVR-8 series of low-power, low-cost microcontrollers. The AT90S2313 had 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 in AttoBASIC are passed on a “data stack” and the interpreter engine is recursive, which allows subsequent commands on a program line (or command line) 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 computer’s serial port, the host’s communications program (a terminal emulator) and the AT90S2313’s on-chip USART. Programs were limited to seventy-two (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 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 a version for the AT90S8515 and Atmega163, adding support for the on-chip Analog-to-Digital Converter and external RAM in the case of the AT90S8515/8535.

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. Language extensions in the form of software modules were added to support the various on-chip peripherals as well various inter-device communications protocols. Support for the ATmega328P, the ARDUINO platforms and a boot-loader was added later.

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 the *AVR Data Recorder project*).

This author has also released to Mr. Cappels, a version 1.1 of the original AttoBASIC for the AT90S2313, which also supports the Attiny2313 and takes advantage of the AVR’s low-power idle mode.

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>.

# AttoBASIC Application Notes

Version 2.34+

## 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 ATmega16/32/88/168/328/644/1284/2560, 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 the Attiny85 (i.e. ADAfruit *Trinket* and OLIMEX *OLIMEXINO-85{S/BC/ASM}*) platforms, if the platform does not already support it, the user must supply a level-shifter or a USB to Serial bridge as aforementioned.

On AVR flavors that have an on-chip USB interface, such as the Atmega32U4 and AT90USB1286, AttoBASIC contains routines that enable the AVR's native USB interface to act as a "Virtual Communications Port" (VCP). Upon initialization, AttoBASIC will wait up to five (5) minutes for a host to connect before it gives up and enters an idle mode. If the "self-start" option is enabled, the timeout occurs after five (5) seconds so as not to inhibit the self-start program from executing.

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 that will likely work.

### 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, if the hardware UART is used then the baud rate chosen is the fastest with the lowest bit error, typically no more than  $\pm 0.16\%$ . 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 with a lower clock speed, the baud rate automatically chosen for no more than  $\pm 0.16\%$  error will be:

9600 baud for clock frequencies of 2 MHz.

4800 baud for clock frequencies of 1 MHz.

For the ATtiny84 and ATtiny85 in particular, due to low-pin-count, AttoBASIC is intended to be used with the on-chip 8 MHz or 16 MHz (PLL) oscillator. Therefore, the baud rate for a bit-rate error under 0.16% will be 38.4K baud for clock frequencies of 8 and 16 MHz. If one so desires, an external crystal or clock source may be used if a more accurate system clock is required or if one wishes to use a lower clock frequency than the internal 8 or 16 MHz clock. Custom builds at frequencies lower than 4 MHz are also supported.

## Power supply voltage

AttoBASIC has been tested on MCU's running with power supply-voltages of 3 and 5 volts. Although most versions of the AVR can run at power-supply voltages as low as 1.8 volts, AttoBASIC has not been tested at that power-supply voltage level. Although there are no reasons why AttoBASIC cannot run at that lower voltage, one MUST use a lower clock frequency and the analog conversion module may not be fully functional if the particular MCU's internal analog reference is used and it is not 1.1 volts.

## Notes about the example programs

The example programs shown in this application note are for reference purposes and have been tested by the author and actual hardware. While certain programming liberties have been exercised for illustration purposes, there are usually more compact and efficient methods to perform the same function(s). The reader is encouraged to experiment.

# AttoBASIC Application Notes

Version 2.34+

## Default MCU Pinouts for Hardware Support Functions

Function	AT90S2313 & Attiny2313	Attiny84	Attiny85	AT90S8515	ATmega163	ATmega16/32	AT-mega88/168/328	ATmega32U4	AT90USB1286	ATmega644/1284	ATmega2560
PWM (OC1A/B)	PB3 (OC1A) only	PA5 & PA6	PB4 (OC1B) only	PD5 (OC1A) only	PD5 (OC1A) only	PD4 & PD5	PB1 (OC1A) only	PB5 & PB6	PB5 & PB6	PD4 & PD5	PB5 & PB6
ICP						PB1 (T1)	PD5 (T1)	PD6 (T1)	PC3 (T3)	PB1 (T1)	PE6 (T3)
DDS		PB2	PB2			PB0	PD6	PD5	PD5	PB0	PE4
DHT							PD7	PB2	PB2	PB3	PB2
TWI						PC0 & PC1	PC4 & PC5	PD0 & PD1	PD0 & PD1	PC0 & PC1	PD0 & PD1
SELF-START (if using I/O pin)	PD0	PB1 (momentary)	PB1 (momentary)	PD0	PD0	PD7	PC3	PD7	PD7	PD7	PD7
SPI						PB[4..7]	PB[5..2]	PB[3..0]	PB[3..0]	PB[4..7]	PB[3..0]
DS		PB3(D) & PB4(A) (if enabled)	PB3(D) & PB4(A) (if enabled)	PB3(D) & PB4(A)	PB3(D) & PB4(A)		PD3(D) & PD4(A)	PC6(D) & PC7(A)	PC4(D) & PC5(A)	PC6(D) & PC7(A)	PC4(D) & PC5(A)
NRF24L01 CE						PB2	PB0	PB4	PB4	PB2	PB4
1-WIRE		PA4	PB3			PD6	PD3	PB5	PB5	PD6	PB5
Ana. Comp.	PB[1:0]	PA2/ADC[1:0]	/ADC[1:0]	PB[3:2]	PB[3:2]	PB3/ADC[1:0]	PD7/ADC[1:0]	ADC[1:0]	PE7/ADC[1:0]	PB3/ADC[1:0]	PE3/ADC[1:0]
Serial I/O	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	PB0 <sub>(RX)</sub> & PB1 <sub>(TX)</sub>	PB0 <sub>(RX)</sub> & PB1 <sub>(TX)</sub>	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	USB or PD2 <sub>(RX)</sub> & PD3 <sub>(TX)</sub>	USB or PD2 <sub>(RX)</sub> & PD3 <sub>(TX)</sub>	PD0 <sub>(RX)</sub> & PD1 <sub>(TX)</sub>	PE0 <sub>(RX)</sub> & PE1 <sub>(TX)</sub>

# AttoBASIC Application Notes

Version 2.34+

## Language Extension Modules enabled for default HEX file builds

Function	AT-90S2313 / AT-tiny2313	ATtiny84	ATtiny84	ATtiny85	ATtiny85	AT90S8515	AT-mega88	ATmega168	ATmega163	AT-mega16 / AT-mega32	ATmega644 / 1284	ATmega328	ATmega32U4	AT90-USB1286	ATmega2560
DEBUG								•		•	•	•	•	•	•
PORT I/O	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
USART SIO	•					•	•	•	•	•	•	•	•	•	•
USB SIO													•	•	
SWUART SIO (½ Duplex)		•	•	•	•										
DDS			•		•			•		•	•	•	•	•	•
DS						•			•		•	•	•	•	•
PWM	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
SPI							•	•		•	•	•	•	•	•
DATA FILE								•		•	•	•	•	•	•
TWI							•	•		•	•	•	•	•	•
1-WIRE		•	•	•	•			•		•	•	•	•	•	•
A/D Conv.		•	•	•	•		•	•	•	•	•	•	•	•	•
INT REG's								•			•	•	•	•	•
ANALOG COMPARATOR	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
SELF-START	• (Pin)	•	•	•	•	• (Pin)	•	•	• (Pin)	•	•	•	•	•	•
ICP								•		•	•	•	•	•	•
EFS								•		•	•	•	•	•	•
SYS CLK		•	•	•	•	•		•		•	•	•	•	•	•
LOW POWER		•	•	•	•		•	•		•	•	•	•	•	•
NRF24L01								•		•	•	•	•	•	•
DHT								•		•	•	•	•	•	•
RTC								•		•	•	•	•	•	•
NV CONFIG		•	•	•	•		•	•		•	•	•	•	•	•
Xtra Ops		•		•	•		•	•	•	•	•	•	•	•	•
CRC16				•	•			•		•	•	•	•	•	•
HELP				•	•		•	•		•	•	•	•	•	•
RAM / EEPROM (bytes)	128/128	512/512	512/512	512/512	512/512	512/512	1024/512	1024/512	1024/512 or 2K/1K	1K/512 or 2K/1K	16K/4K or 8K/2K	2K/1K	2.5K/1K	8K/4K	8K/4K
Variables Available	A-D	A-M	A-M	A-M	A-M	A-Z	A-Z	A-Z	A-Z	A-Z	A-Z	A-Z	A-Z	A-Z	A-Z
Data Stack Depth	4	9	9	9	9	12	17	34	12	17 or 34	17 or 34	17 or 34	17 or 34	17 or 34	17 or 34
FOR-NEXT & GOSUB depth	1	2	2	2	2	1	4	4	1	4	4	4	4	4	4
DATA Statement Size (bytes)	N/A	8	8	8	8	N/A	8	16	N/A	32	32	32	32	32	32
Nonvolatile Data Storage (bytes)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
System Clock (MHz)	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20	4/8/16 /20
Boot-Loader	[none]	OptiBoot85	[none]	OptiBoot85	[none]	[none]	[none]	OptiBoot	[none]	OptiBoot	OptiBoot	OptiBoot	LUFA DFU or CDC	LUFA DFU or CDC	STK500V2

# AttoBASIC Application Notes

Version 2.34+

## The traditional “Hello World” program

A "Hello world" program is a computer program that outputs the string "Hello World!" on a display device. Since printing output is typically one of the simplest programs possible in most programming languages, it is often traditionally used to illustrate to beginners the most basic syntax of a programming language. It is mostly used to verify that a language is compiling and/or operating correctly. The following example program illustrates this concept as both a program and on the command line in direct mode.

```
10 PRINT "Hello World!~"      # A program to print "Hello
                              # World!" followed by a new
                              # line

PRINT "Hello World!~"        # Direct command to print
                              # "Hello World!" followed
                              # by a new line
```

On a device that does not display text, a simple program to produce an indicator, such as turning on/off an LED, is often substituted for "Hello world" as the introductory program. The following program illustrates this concept with a combined “blink” and “Hello World!” program written to run on an ARDUINIO Duemilanove.

```
10 SDB5                      # Set Port pin B5 to an output
20 XBB5 : DELAY 25            # Toggle the LED state then
                              # delay 0.25 seconds
30 PRINT "Hello World!~"      # Print "Hello World!"
40 GOTO 10                    # Loop forever thus blinking
                              # the LED and printing "Hello
                              # World!"
```

```
=====
AttoBASIC for ATmega328P @16MHz
Ver: 2.32 Build code: 20131022.1747
-----
Original code (C)2002 by Richard Cappels
http://projects.cappels.org projects@cappels.org
-----
Enhancements by K. Scott Vitale (ksv_prj@gmx.com)
ATmega88/168/328 port (C)2011
1-Wire(R) support (C)2013
nRF24L01(+) RF Transceiver support (C)2013
DHT Humidity and Temperature support (C)2013
EEP File System support (C)2013
Sleep, RTC and Data-file support (C)2012
TWI/SPI/DDoS & Input Capture (C)2011
-----
OptiBoot Bootloader based on stk500boot.c by
Jason Kyle, Peter Knight, Bill Westfield & Ladyada
-----
All rights and remedies expressly reserved
Not for commercial use
=====
>PRINT "Hello World!~"
Hello World!
>|
```

## ATtiny84 and ATtiny85 Support (Version 2.34+)

### Overview:

As of version 2.34, AttoBASIC supports the ATtiny84 and ATtiny85. Like the ATmega88, this MCU has only 8KB of FLASH and 512 bytes of RAM and EEPROM. Unlike the ATmega88, the ATtiny85 has merely five (5) I/O pins or six (6) if the RSTDISBL fuse is programmed, while the ATtiny84 has eleven (11) I/O pins or twelve (12) if the RSTDISBL fuse is programmed. The ATtiny84 and ATtiny85 are somewhat limited in their suite of on-chip hardware peripherals. In Particular is their lack of a dedicated UART. Instead, the MCU uses a “Universal Serial Interface” (USI), which directly supports an SPI or TWI interface ... but not both at the same time. With some software assistance, it can emulate a “hardware” UART, which is NOT what AttoBASIC has implemented. Instead, an interrupt-driven software UART has been implemented, which allows the use of the *sleep* instruction to save power. As with a hardware-based UART, two (2) I/O pins are dedicated to UART communications, leaving three (3) (or four (4)) additional pins available for I/O.

### Software UART:

The UART implemented is what is termed a “half-duplex UART”, which means that it can only transmit OR receive a single character at a time. In AttoBASIC, this is not normally an issue. However, when running a program that is constantly printing to the console, the control-S or control-C keys may have to be pressed more than once for AttoBASIC to see the keystroke and react to it. Also, to give AttoBASIC’s UART software time to “see” a received character, a small delay of 1 millisecond is inserted between each character transmitted.

### Bootloader:

Unlike their cousins, the ATtiny84 and ATtiny85 do not contain special hardware to support a boot-loader program. However, clever software engineers have devised a method of implementing one. In HEX file builds indicating such, AttoBASIC uses the Tiny85 version of the OptiBoot boot-loader. Refer to the writing entitled *Device Programming Instructions* for details on using the OptiBoot85 boot-loader.

# AttoBASIC Application Notes

Version 2.34+

## ATtiny2313 Support (Version 2.34+)

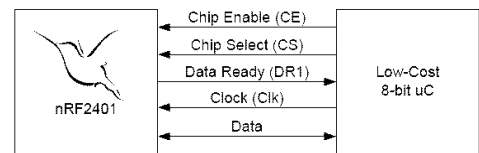
As of version 2.34, AttoBASIC supports the addition of pre-built HEX files for the ATtiny2313. AttoBASIC for the 2313 is very limited as the ATtiny2313 MCU has only 2KB of FLASH, 128 bytes of RAM and 128 bytes of EEPROM. The program space is limited to 71 bytes. The source code for the 2313 series has not been merged into the current AttoBASIC source tree and will never be. The addition of the AttoBASIC HEX files for the 2313 are for posterity reasons only.

## AT90S8515 and ATmega163 Support (Version 2.34+)

As of version 2.34, AttoBASIC supports the addition of pre-built HEX files for the AT90S8515 and ATmega163. The source code for each of these builds has not been merged into the current AttoBASIC source tree and will likely never be. The addition of the AttoBASIC HEX files for these MCU's is for posterity reasons only as the current AttoBASIC source code was derived from this root.

## 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*<sup>TM</sup> technology is inherent in the nRF24L01(+) devices. As a power-up default, *ShockBurst*<sup>TM</sup> 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.



# AttoBASIC Application Notes

Version 2.34+

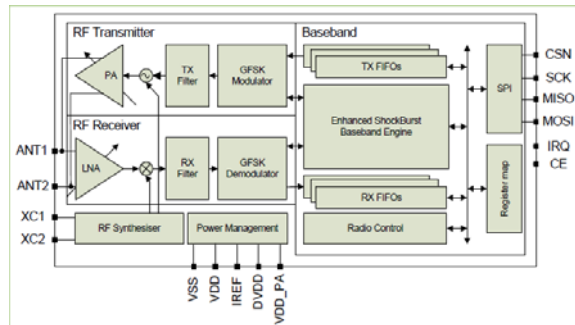
The AttoBASIC command set does not directly support the “dynamic payload” and “Payload with ACK” features of *Enhanced ShockBurst™*, however, with the use of the **RFW** and **SPW** commands, one may make use of them.

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

# AttoBASIC Application Notes

Version 2.34+

```
75 PRINT "D:" ; PRX N
80 PRINT "D:" ; PRX 0
85 PRINT "D:" ; PRX P
90 RFT M N 0 P
95 RET
```

# SEND THE RTC DATA  
# 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
15 RFI 2 0 1
20 RFB 0 4
25 RFF ; RFE 1
30 A:= RFR; A:= A AND 1
35 PRI "." ; SLP 3
40 IF !A THEN GOTO 30
45 PRINT "~PAYLOAD # "; PRINT C
50 C:= C+1
55 P:= RFX

60 FOR N= 1 4
65 PRI "D:" ; PRX READ N
70 NEXT
75 PRI "-----"
80 GOTO 25
```

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

60 FOR N= 1 4
65 PRI "D:" ; PRX READ N
70 NEXT
75 PRI "-----"
80 GOTO 25

5 REM nRF24L01 MULTICEIVER-TRANSMITER ON PIPE 3'S ADDRESS
10 C:=0
15 RFI 1 0 1
20 RFA 0 #C2 #C2 #C2 #C2 #C3
25 RFA 7 #C2 #C2 #C2 #C2 #C3
30 RFW 4 #13
35 RFF
```

# INIT PAYLOAD COUNTER  
# POWER UP IN RX MODE, CHAN 0, 2 MBPS  
# **4 bytes of data for RX3**  
# FLUSH RX AND TX FIFO AND ENABLE RX MODE  
# FETCH THE STATUS OF RX FIFO  
# PRINT A STATUS CHARACTER AND DELAY  
# CHECK FIFO  
# PRINT SOME INFO  
# INCREMENT THE PAYLOAD COUNTER  
# TRANSFER THE RX DATA TO THE DATA BUFFER AND PRINT  
# THE PIPE NUMBER THAT RECEIVED THE DATA  
# SET UP THE LOOP COUNTER FOR 4 BYTES  
# PRINT THE DATA RECEIVED  
# LOOP SOME MORE  
# PRINT A PERFORATION  
# LOOP

#INIT PAYLOAD COUNTER  
# POWER UP IN TX MODE, CHANNEL 0, 2 MBPS  
# **SET PIPE 0 TO MAC FOR PIPE 3 (for ShockBurst)**  
# **SET TX TO MAC FOR PIPE 3**  
# **ADJUST AUTO-REXMIT DELAY TO 500US**  
# FLUSH RX AND TX BUFFERS

# AttoBASIC Application Notes

## Version 2.34+

```
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.

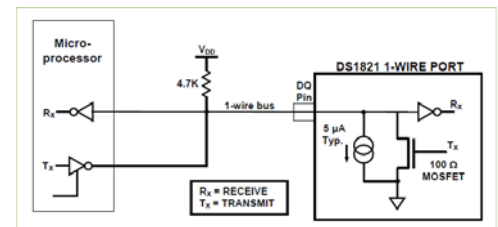
In particular is during transmission of a packet. If a receiver did not receive a packet and the MAX\_RT flag is set in the "STATUS" register, then that last packet is "stuck" in the FIFO buffer, effectively blocking any packets scheduled for transmission after it. There are two (2) ways to resolve this 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 adds language extensions to support the DALLAS/MAXIM 1-Wire® protocol using three (3) primitive commands; The **OWI** command resets the 1-Wire® bus and detects the presence of devices, the **OWW** command writes data to a device on the 1-Wire® bus while the **OWR** 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 encountered data corruption problems using only the port pin pull-up but design guidelines suggest using a separate pull-up resistor (in the range of 2.2KΩ to 4.7 KΩ) in addition to the port pin pull-up.

The use of the **OWI**, **OWW** and **OWR** 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 OWI = 0 THEN GOTO 40      # initialize bus and test for device present
15 OWW $EE                     # start a temperature conversion
20 OWW $AA                     # initiate a read
25 PRINT "Temperature: " ; PRINT OWR # print the temperature read
30 SLP 9 ; GOTO 10             # sleep for 9 seconds and loop forever
35 END
40 PRINT "NO DEVICES!~" ; END   # inform user of error and end
```

# AttoBASIC Application Notes

## Version 2.34+

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 EEPROM.

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

A variation of the **\$WR** 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 $WI = 0 THEN GOTO 45 # initialize bus and test for device present
15 $WW $33 ; $WR 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 to life, the universe and everything 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
```

# AttoBASIC Application Notes

## Version 2.34+

```
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
```

### EEPROM – Non-volatile configuration (Version 2.34+)

As of version 2.34, AttoBASIC adds support for saving configuration information to non-volatile memory (EEPROM). Currently, the only non-volatile option supported is enabling or disabling of the “self-start” feature, which saves an I/O pin over previous versions of AttoBASIC. Access to the non-volatile options is via the **CFG** command. There are also four (4) user application bits available. The bit positions are defined as follows:

Bit Position	Description
0	SELF-START; “0” = disable, “1” = enabled
1	Reserved
2	Reserved
3	Reserved
4	User application use 0
5	User application use 1
6	User application use 2
7	User application use 3

The following sample program illustrates using the **CFG** command as a reverse comparison test condition in an **IF-THEN** statement:

```
5 REM TEST CFG FUNCTIONALITY
10 CFG 8 0: GOSUB 45 # CLEAR THE CONFIG REGISTER AND PRINT IT
15 IF 1 = CFG 7 THEN GOSUB 55: GOTO 25 # TEST FOR BIT 7 SET AND PRINT IF TRUE THEN JUMP
20 GOSUB 50 # PRINT IF FALSE
25 CFG 7 1: GOSUB 45 # SET BIT 7 OF THE CONFIG REGISTER AND PRINT
30 IF 1 = CFG 7 THEN GOSUB 55: GOTO 40 # TEST FOR BIT 7 SET AND PRINT IF TRUE THEN JUMP
35 GOSUB 50 # PRINT IF FALSE
40 END # FINISHED!
45 PRINT "CONFIG[7]: "; PRINT CFG 7: RET # PRINT THE CURRENT VALUE OF THE CONFIG REGISTER
50 PRINT "CONFIG BIT CLEAR~": RET # PRINT THE MESSAGE
55 PRINT "CONFIG BIT SET~": RET # PRINT THE MESSAGE
```

### EEPROM – Reserved Application Space (Version 2.34+)

As of version 2.34, AttoBASIC adds support for reservation of EEPROM space for application use. The reserved addresses are 0x08 to 0x17 (or 8 to 23). The EER and EEW commands may be used to read and write to those locations specifically. Insure that the application does not overwrite vital areas of the EEPROM that are reserved for AttoBASIC’s internal use, 0x00 to 0x07 (and 0x18 to 0x1F if the EFS is enabled).

### Using DATA, READ and RESTore (Version 2.30+)

As of version 2.30, AttoBASIC supports storing up to sixteen (16) [or thirty-two (32)] 8-bit values in RAM using the **DATA**, **READ** and **RESTore** 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 overwritten by another **DATA** statement or if the nRF24L01, SPI, TWI or 1-Wire<sup>®</sup> routines are enabled, execution of the **RFX**, **SPR**, **OWR** or **TWR** commands, whether executed within a program or in immediate mode. Note that if the nRF24L01, SPI, TWI or 1-Wire<sup>®</sup> routines are enabled then the DATA size is thirty-two (32) bytes.

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
```



# AttoBASIC Application Notes

## Version 2.34+

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 1<sup>st</sup> data element is at position “0” and the 8<sup>th</sup> 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 ; GOTO 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 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

# AttoBASIC Application Notes

## Version 2.34+

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 1<sup>st</sup> byte of a data storage block specifies the type of block, either 0x01 for data or 0xFF for unused. The 2<sup>nd</sup> through 31<sup>st</sup> byte contains the program data and the 32<sup>nd</sup> 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 byte 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 1<sup>st</sup> program line number, it may be helpful to optionally make the 1<sup>st</sup> 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 and Tiny85 due to code size but it is available on the Mega168 even though there is only 512 bytes of EEPROM available.

### System Clock Prescaler and internal RC oscillator adjustment (Version 2.34+)

As of version 2.34, AttoBASIC adds language extensions to support reading and writing the System Clock Prescaler (CLKPR) and the internal RC oscillator's calibration register. The main purpose for reducing the system clock is to reduce power consumption during long periods of idle.

The following program illustrates using the **OSC** and **CLK** command in a program. When run, it will set the system clock prescaler to divide the system clock by 1, 2, 4, 8 and 16, printing a message and blinking an LED on PORTB0 10 times with an initial on/off duration of 10mS. The on/off duration of the LED doubles as the system clock is reduced by each step in the loop. If the particular MCU build uses a Software UART, the baud rate at which messages are printed will divide by a factor of 2 with each step of the loop.

```
5 SDB0; CB00          # Set the LED's I/O port and state to low
10 OSC 122            # Set the calibration of the internal oscillator
15 FOR A=0 4          # loop 0 to 4 (div by 1, 2, 4, 8 & 16)
20 CLK A; GOSUB 45     # Set SysClk and print current value
25 GOSUB 50            # Blink the LED
30 NEXT               # Loop
35 CLK 0; GOSUB 45     # Return to 1x divider
40 END                # All done now
45 PRI "CLK="; PRI CLK; RET # Inform user of current SysClk
50 FOR B=1 10         # Set up loop
55 CB00; DEL 1; SB00; DEL 1 # Blink the LED 10mS on/10mS off at 1x divider
60 NEXT               # Loop
65 RET                # Return
```

Caution should be exercised when using the **CLK** command as setting the system clock to a lower frequency will reduce run current but also reduce the clock to all the timers and the ADC. If the user's program does

# AttoBASIC Application Notes

## Version 2.34+

not reset the system clock prescaler back to "0" then one may lose the ability to communicate with AttoBASIC until a hardware reset is activated.

When adjusting the oscillator calibration value with the **OSC** command, only small incremental adjustments should be made as using too large a value can cause the internal oscillator to lock up. See the particular MCU's datasheet for details.

### 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 ( $\geq 3.3V$ ), 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 DHT 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.



# AttoBASIC Application Notes

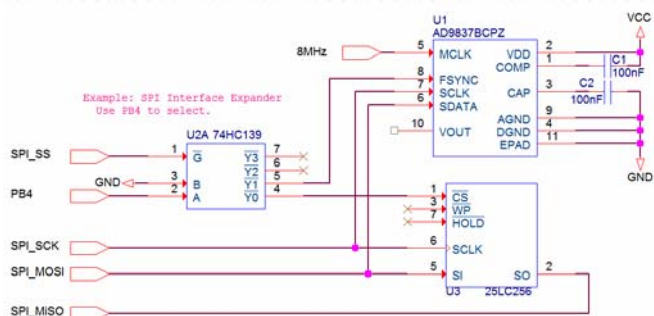
## Version 2.34+

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 1          # sleep for 1 second
30 PRI ADC 0      # print the value on ADC channel 0
40 GOTO 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 this way, AttoBASIC can be used to easily capture analog or digital values at pre-determined intervals and record them in the data file for retrieval at a later time.



The EEPROM is connected to the SPI pins of the AVR. A device with up to 64K bytes of storage is directly supported. If an external EEPROM is connected to the SPI pins, then the ability to use another SPI device is disallowed unless additional support hardware 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@RTCL   # store lower 16-bits of RTC (time-stamp)
23 DFL PEEK VPG@RTCD
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 GOTO 20            # loop forever
```

### DDS - Direct Digital Synthesis (Version 2.00+)

As of version 2.00, AttoBASIC adds a language extension to support the generation of a variable frequency square wave using Direct Digital Synthesis (DDS). Since the software DDS method employed uses a 24-bit accumulator, the frequency must be set using three 8-bit variables; X, Y and Z. For simplicity, each variable is read by the DDS command as a series of six (6) 2-digit BCD values, where each digit contains the value of 0 to 9 (inclusive). Example: X:Y:Z, where X contains the value signifying the 100K and 10K hertz positions, Y contains the value signifying the 1K and 100 hertz positions and Z contains the value signifying the 10 and 1 hertz positions. Once X:Y:Z is set, the "DDS 1" command is issued to enable the output. If the frequency is changed after the initial "DDS 1" command, a subsequent "DDS 1" command must be reissued to update the DDS registers. *Note that when using the Input Capture command, the DDS function is temporarily inhibited.*

The sampling frequency for the internal DDS routines is derived from the MCU's master clock and is calculated as "**F\_CLK/128**". Therefore, the maximum attainable frequency is based on the *Nyquist-Shannon Sampling Theorem* and the *Nyquist frequency* for any given MCU clock speed can be calculated as

# AttoBASIC Application Notes

Version 2.34+

“F\_CLK/256” rounded down to the nearest 1KHz.

The following program illustrates using the DDS command:

```
5 REM DDS functional test
10 X:=0: Z:=0                # start at "00:00:00" Hz
15 FOR N= 0 TO 9             # sweep from 1KHz to 9KHz in 1KHz steps
20 Y:= SWAP N                 # swap the 1K and 100 Hz positions
25 DDS 1                      # enable the output
30 PRI "Freq in KHz: "; PRI N # update the display
35 DEL 200                    # delay 2 seconds
40 NEXT                       # increment loop
45 END                        # all done
```

## 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 TO 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 2.00+)

As of version 2.00, AttoBASIC supports the AVR's SPI interface in Master mode at speeds of up to (CPU\_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 SPS 0                        # Select the slave by dropping the SS pin
20 SPW #21: SPW #58             # Write #21 and #58 to the SPI device
25 PRI SPR                      # Print the data read from the SPI device
30 SPS                          # Deselect the slave by raising the SS pin
```

As of version 2.34, AttoBASIC no longer controls the SS pin. It is now the user's responsibility to do so using the *SPS* command.

As of version 2.32, the SPW and SPR commands offer multiple byte transfers. The following example program illustrates the use of multi-byte data transfers using the SPW and SPR commands, which are emphasized in bold text.

# AttoBASIC Application Notes

## Version 2.34+

```
5 REM SPI Test w/ nRF24L01
10 SPM 0: SPS 0 # Initialize SPI interface to Mode 2 and drop SS pin
15 SPW #2A #AA #AA #AA #AA #01 # Set the nRF24L01 RX pipe Address
20 SPS # Deselect the slave by raising the SS pin
25 SPW #0A: SPR 5: SPS # Read the address into DATA buffer
30 FOR N=1 5: PRX READ: NEXT # Print out the RX pipe Address
```

## TWI (I<sup>2</sup>C) interface (Version 2.00+)

As of version 2.00, AttoBASIC adds language extensions to support the AVR's TWI (I<sup>2</sup>C) 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 a simple bus error check illustrates:

```
5 REM Write to a TWI Device
10 TWI 0: TWS # init the TWI interface @400kbps and assert a
# START condition
15 IF #40 <> TWW #5B # address the slave (write bit set) and test for ACK
20 PRINT "No Device~": END # inform the user no device responding at address #5B
25 TWW #00: TWW #FE: TWP: END # write #00 and #FE to the slave then assert STOP
# condition to signal end of transmission
```

As of version 2.32, the bulk of AttoBASIC's TWI (I<sup>2</sup>C) commands return the status of the TWI bus after the low-level hardware commands have finished executing. No longer does AttoBASIC abort a user's program and complain about a bus error. All bus errors are now to be handled by the user's program. The following example program shows how the TWI bus can be scanned at all device addresses looking for a device to respond with an ACK(nowledgement). In particular, it demonstrates two programming concepts (emphasized in bold); 1) nesting of FOR-NEXT loops, one for each nibble of the device address and 2) the use of reverse-conditional testing with the **IF** command:

```
5 REM TWI Bus Device Scan
10 TWI # 400K bus speed
15 FOR H= #0 #F # Device address loop MSN
20 FOR L=0 #F # Device address loop LSN
25 A:= L + SWAP H # swap H to high nibble and add the value of L
# to calculate the slave address
30 TWS: GOSUB 45: TWP # address slave if there
35 NEXT
40 NEXT: END # end of loops
45 IF #40 = TWW A # Was ACK on address?
50 PRINT "ACK at ": PRX A: RET # Print "ACK" if a device responds then return
54 RET # Return if no device responds
```

Also as of version 2.32, AttoBASIC's TWI (I<sup>2</sup>C) support transfers bulk data to the DATA statement's buffer by the TWR command for subsequent reading by the READ command. The following example program combines a bus scan for a MICROCHIP MCP3221 12-bit A/D converter and returns the address and the reading (2 bytes) if the device is found on the TWI bus, otherwise, error-checking is employed.

```
5 REM MCP3221 TEST
10 R:= 0: A:= #91 # Clear a variable and set device start address
15 TWI # 400K bus speed
20 FOR N=0 7 # device 0 to 7 sub-address loop
25 M:= A + LSL N # calculate slave address
30 TWS: GOSUB 75: TWP # address slave if there
35 NEXT # end of loop
40 IF R = 0 THEN PRINT "No Device!~": END # Error if no device
45 TWS: TWW R: TWR 2: TWP # Start, address slave, read two bytes
50 IF 0 = RES THEN PRINT "No Data Read!~": END
55 PRINT " MCP3221 at Address ": PRX R # inform user
60 PRINT "H: ": PRX READ # Print high byte
65 PRINT "L: ": PRX READ # Print low byte
70 END # end of program
75 IF #40 = TWW M THEN R:= M: RET # Test for slave ACK, assign R to address, return
80 RET # No device detected, return
```

As of version 2.34, AttoBASIC's TWI (I<sup>2</sup>C) support allows the TWW command to transfer character strings to a slave on the bus as the following example illustrates.

# AttoBASIC Application Notes

## Version 2.34+

```
5 REM ST7032 LCD Test V2.34
10 twi : tws                                     # Initialize the TWI bus
15 tww $7c $30 $30 $39 $14 $70 $57 $bB         # Initialize the ST7032 LCD
20 del 20: tww $38 $0c $0b $01 : twp           # wait 200mS then set the modes
25 del 10: tws : tww $7c $80 $80 $40           # wait 100mS then set the start address
30 tww "ABCDEFGHJKLMNOP" : twp                  # Send the string
35 tws : tww $7c $80 $C0 $40                   # Set the next address
40 tww "1234567890123456" : twp                # Send the string
```

Another example of interfacing with a device on a TWI bus follows. In this case, a TEXAS INSTRUMENTS BQ32000 Real-Time Clock (RTC) IC is used as the example device. For clarity, each functional section of the program is separated by a blank line. A dummy date and time are written to the RTC using the DATA and READ commands. A four (4) second delay is executed before reading the RTC registers. The delay insures that the RTC is running as register 0 shows "4" seconds have elapsed since the RTC was written. In this program, no error checking is employed as all reads and writes to the device are assumed to be have occurred without error.

```
5 REM BQ32000 RTC Test
10 TWI 1                                           # init TWI @100kbps
15 A:= $D0 : B:= $D1                             # write and read addresses for BQ32000 RTC
20 TWS : TWW A 0 $80 : TWP                       # set the RTC's STOP bit (must toggle to start RTC)
25 DATA $40 $20 $45                             # setup data for configuring RTC control registers
30 FOR R=7 9                                     # set control registers 7 to 9
35 D:= READ : GOSUB 115                         # read each data and write to device
40 NEXT                                           # end of loop

45 DATA $13 $17 $10 $00 $05 $30 $00           # setup RTC's date (year to seconds)
50 FOR Z=249 255                                 # set data registers 6 to 0 (count down)
55 R:= COM Z : D:= READ                         # set register and data
60 GOSUB 115                                     # write it
65 NEXT                                           # end of loop

70 PRINT "Wait 4 sec~" : SLP 4                   # wait 4 seconds to see if the clock is running
75 R:=0 : GOSUB 110                             # select register 0 and write it
80 TWS : TWW B : TWR 7 : TWP                   # slave to read and retrieve 7 bytes
85 FOR R=0 6                                     # Loop from register 0 to 6
90 PRINT "R:" : PRI R                           # print the register
95 PRINT "D=" : PRX READ                       # print the data in the register
100 NEXT                                         # end of loop
105 END                                           # end of program

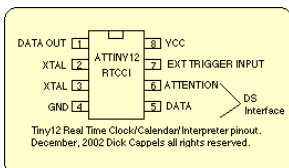
110 TWS : TWW A : TWW R : TWP : RET             #Select a register
115 TWS : TWW A R D : TWP : RET                # Select device and write data to a register
```

### 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.

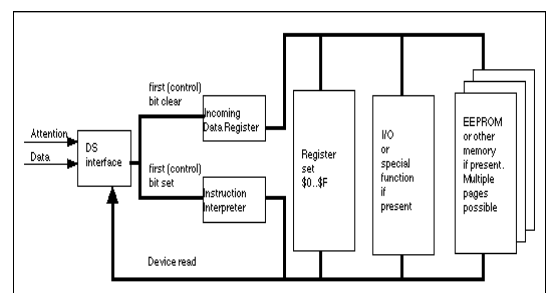
The DS protocol was designed to provide firmware-based bidirectional host-to-slave inter-processor communications for situations in which no hardware solution is available (TWI, SPI, CAN, 1-WIRE®, etc.)

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



# AttoBASIC Application Notes

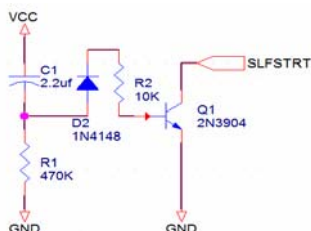
Version 2.34+

web site at: <http://www.cappels.org/dproj/dspage/ds.htm>

Using AttoBASIC with the DS protocol requires one first send the DATA then the COMMAND. Also, unless the DS slave uses internal pull-up resistors on the DATA and ATTN lines, the user must provide them. A good value to use is 10KΩ.

## Self-Start Feature (Version 1.0 to 2.33)

AttoBASIC supports a "self-start" feature wherein if the designated SLFSTRT pin is held "low" for at least 250mS during power-up, the program stored in 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. Note that the hardware version of self-start has been depreciated and replaced with the "soft" self-start, which is accessible through the **CFG** command.

## Self-Start Feature (Version 2.34+)

As of version 2.34, AttoBASIC's "self-start" feature has been enhanced, whereas the use of an I/O pin to signal the user's desire to self-start a program is no longer needed. This feature is now stored in the non-volatile memory of AttoBASIC, which is read upon power-up or the application of an external reset. Note that a program can set and clear configurations. Example:

```
CFG 0 1      # Enable self-start
CFG 0 0      # Disable self-start
```

### Notes:

- 1) As of version 2.34, the pre-built HEX files no longer support the hardware-based "self-start" method. This completely frees the I/O port for other uses. However, the program routines are still available within AttoBASIC's source code if one wishes to be adventurous and enable them.
- 2) On USB-Serial I/O builds, AttoBASIC will wait up to five (5) minutes for a host to connect before it gives up and enters a standby state. If the "self-start" option is enabled, AttoBASIC will wait five (5) for the host to connect then execute the "self-start" program. This provides a shorter startup delay, which affords the user a means to connect and interact with AttoBASIC in the event the "self-start" program must be terminated manually. (i.e. for debugging)
- 3) If the "self-start" program ends and does not continuously execute a repetitive loop, then AttoBASIC will wait two (2) minutes for a host to connect before giving up and entering a standby state.
- 4) In using a "new" AVR with the EEPROM completely unprogrammed, all bytes will be set to 0xFF, which means that AttoBASIC will see the configuration bit for "self-start" as enabled. This is not a problem, as the solution is to write a "0" to bit "0" of the configuration byte. Ex: **CFG 0 0**

## Saving program space by compressing characters

As AttoBASIC is implemented in the ATMEL AVR 8-bit series of microcontrollers, there is a limited amount of memory available. The AttoBASIC operating code is stored in FLASH memory while user programs are stored in RAM. If the EFS language extension is enabled, then "saved" user programs are stored in EEPROM. In some cases, a user's program may be large and may not fit into the available RAM. Consider that each character in a user's program takes 1 byte to store, which includes spaces and the End-Of-Line (EOL) terminator code, 0x0D. There are a few tricks that can be played to reduce the number of characters a program uses as illustrated below.



# AttoBASIC Application Notes

## Version 2.34+

This program calculates the value of I squared and prints it. The program takes 90 bytes of RAM.

```
10 FOR I = 0 TO 15
20 PRINT "I= "
30 PRINT I
40 PRINT "I*I= "
50 PRINT I*I
60 NEXT
70 END
```

By shortening commands to three (3) characters, removing spaces and combining statements on a single line using the “;” (semicolon) keyword, the program can be reduced to using 55 bytes of RAM, a 38% savings of space. Its not very pretty but sometimes availability of program space overrides how pretty a program will look.

```
10FOR I=0 15;PRI"I= ";PRI I;PRI"I*I= ";PRI I*I;NEX;END
```

Note that when removing spaces between commands and their parameters, using say “PRINT” with a number (**PRI42**) is legal while using “PRINT” with a variable (**PRIA**) will cause an error. This is because AttoBASIC searches for keyword commands within a four (4) byte text string in a dictionary lookup. It can readily distinguish between an alphabet character command and a number directly following the command but an alphabet character command directly followed by a variable, another alphabet character, is seen as a single four (4) alphabet character command and will not be found within the dictionary.

### 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. Program line number zero (0) is valid under AttoBASIC.
2. 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.
3. 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

# AttoBASIC Application Notes

## Version 2.34+

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 ; GOTO 100 [true statement]  # execute if A = B
30 PRINT "Not Equal~" ; GOTO 200 [not true statement] # execute if A <> B
```

- Note that execution of the TRUE statement does not skip the FALSE statement.

4. 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, if there was one. Variables are handled differently in AttoBASIC and thus can exist peacefully on the left side of a relational operator command. 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 to B
```

In the following more complex example, the compound bitwise result of \$55 OR’ed with \$FF is then AND’ed with \$AA and assigned to the variable **B**. Keep in mind that the order of operations is from right to left so that complex functions can be seen as **(EQ1 (EQ2 (EQ3 (EQ4) ) ) )**.

```
B:= COM $AA AND $55 OR $FF      # assign the inverted value of $AA & ($55 | $ff)
```

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

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
```

The following example illustrates a reverse comparison using the above example for a reference. Once again, the bitwise result of \$55 AND’ed with \$AA is tested to be equal to zero (0) but in reverse comparison order, which evaluates from right to left.

# AttoBASIC Application Notes

## Version 2.34+

```
10 IF 0 = $AA AND $55          # IS ($AA & $55) = 0?
20 PRINT "Zero~" ; GOTO 10      # print "Zero" if so, loop
30 PRINT "Not Zero~" ; GOTO 10  # print "Not Zero" not so, loop
```

5. Another method of dealing with the results 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
```

6. The "\$" is an operator and must be formally delimited from other operators such as +, -, =, <, !, etc.
7. 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")
8. 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

A := 255

FOR A =

A:= \$3A

PRINT A

PRINT A, B, 4, 5

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

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

GOTO A

### Erroneous Usage

A := 256

FORA =

A:=\$3A

### Reason for error

Exceeds the value of an 8-bit number.

AttoBASIC sees this as a command named "FORA", which is undefined.

AttoBASIC requires a space between the "=" and "\$" characters.

AttoBASIC will print the values on a line from right to left.

No error.

Valid syntax but not necessary



# AttoBASIC Application Notes

## Version 2.34+

GOTO A+B  
GOTO 2 + 2 + 5

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

Starting with AttoBASIC version 2.00, 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<sup>®</sup> 8-bit AVR<sup>®</sup> 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
_USB_Drivers	; contains the WINDOWS <sup>®</sup> USB drivers for LUFA and USB SIO
_Serial_IO	; contains the source code for the various USB and Serial I/O modules
_Bootloaders	; contains the source code and HEX files for the various boot-loaders
_Docs	; contains the various support documentation

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 two files from the "*Include*" subdirectory named "*Defs\_MCU-Options.inc*" and "*Defs\_Constants.inc*". "*Defs\_MCU-Options.inc*" is the file where the specific options relating to MCU, clock speed and "*Defs\_Constants.inc*" is the file where specific module ex/inclusion is performed.

In certain MCU's, such as the ATmega88 and ATtiny85, which contain limited FLASH, only select modules are enabled in the default HEX file builds. Please consult the file named "*Defs\_Constants.inc*" for specifics.

#### **Within the "*Defs\_MCU-Options.inc*" file:**

##### 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:

# AttoBASIC Application Notes

## Version 2.34+

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 major version, [Mn] is the minor version and [Sb] is the subversion of the build. The following line in “*AttoBASICV234.asm*” 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 = 234 ;version 2.34 (100x so its an integer)
```

### Within the “*Defs\_Constants.inc*” file:

#### Enabling and disabling support modules:

For all MCU’s with FLASH memory of 32KB or larger, all software support modules are enabled. Due to FLASH size constraints on the 8KB and 16KB ATtiny85, 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 ATtiny85, 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:

# AttoBASIC Application Notes

## Version 2.34+

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*".

### **Interfacing with AttoBASIC on a Linux platform (Version 2.00+)**

Since his involvement with AttoBASIC starting with version 2.00, the author chose to actively develop AttoBASIC on a mixed WINDOWS<sup>®</sup> and Linux platform. As the ATMEL development tools are designed to run on a WINDOWS<sup>®</sup> platform, the author's development environment needed to support WINDOWS<sup>®</sup> but Linux possesses a multitude of free utilities for performing a wide variety of tasks, which makes Linux a "Swiss Army knife" for a software and hardware developer.

Originally, a dedicated WINDOWS<sup>®</sup> XP SP3 platform was used and the development files were stored on a shared network drive hosted by a dedicated Linux server. An SSH shell was used to log into the Linux server and a slew of utilities were used and embedded into custom BASH script files to process the various build flavors of AttoBASIC.

At one point, the author needed to "go mobile" and did not always have access to the dedicated Linux server. So the development files were transferred to his laptop. However, WINDOWS<sup>®</sup> lacked the Linux tools previously used. Eventually, the author's development environment evolved into a Linux Mint 13 system running a (licensed) copy of WINDOWS<sup>®</sup> XP SP3 in an ORACLE Virtualbox virtual machine.

What follows is a description of the Linux script files written by the author. Anyone wishing to further develop AttoBASIC may find them of use. These script files are written in BASH, some do not function properly with DASH (default for UBUNTU and MINT). Most have been coded to directly call *bash* from the */bin* directory. As a general rule, these script files need the following Linux software packages installed (dependencies as well); *bash*, *srecord* utilities, *wine*, *zip/7z*, *sed*, *grep*, *dos2unix*, *picocom*, *ascii-xfr* (part of *minicom*), *udev* tools, *avrdude*, *make* utilities and the *find* utilities.

<code>terminal.sh</code>	Used to invoke picocom with USB-to-serial converters. Will display a list of <i>/dev/ttyACMx</i> and <i>/dev/ttyUSBx</i> devices to choose from. Sets the baud rate to 38.4k and defines the proper character and line delays for using <i>ascii-xfr</i> to upload user programs.
<code>WineBuild.sh</code>	Single-pass call to the AVRASM2.EXE program, which runs under a <i>wine</i> console. Invoking this script will assemble the currently selected features of AttoBASIC.
<code>mk_buildflavors.sh</code>	This is the do-it-all script used to perform the mass build of AttoBASIC flavors. It manipulates the <i>Defs_MCU-Options.inc</i> file and calls the <i>Wine-Build.sh</i> script. It also facilitates (optionally) compiling the various bootloaders and USB Serial I/O modules. The <i>Makefile</i> in each of the various bootloader and USB Serial I/O directories actually perform the compilation for each target MCU and clock frequency then create a file in the proper file format (using <i>.db</i> statements) to be included in the overall build process.
<code>mk_avrfreaks.sh</code>	The AVRFREAKS web site imposes (undisclosed) project file size limita-

# AttoBASIC Application Notes

## Version 2.34+

tions (of about 1.8MB). This script creates three (3) individual ZIP files from the various project files and directories and names them appropriately.

`mk_dr_def_prg.sh`

This script supports the AVR Data Recorder project. It pulls the default data recording program from the AVR Data Recorder's home project directory and creates a file in the proper file format (using *.db* statements) to be included in the overall build process. It is called from the `mk_buildflavors.sh` script

`mk_prog-avr.sh`

This script allows easy selection of the various AttoBASIC HEX files and subsequent programming of the target MCU using the target platforms bootloader. Multiple programming methods are supported, including; AVR DRAGON (JTAG, ISP, HVSP and HVPP), AVRISP and USBTINY. The STK500v2 and AVR109 bootloader protocols are supported using the USART and LUFA CDC and DFU bootloaders are supported using the USB interface on certain builds. Fuses may also be programmed if the particular bootloader supports it. At invocation, a list of available MCU's is displayed for the user to choose from, then the applicable HEX file is chosen, then a list of `/dev/ttyACMx` and `/dev/ttyUSBx` devices is displayed for the user to choose from. After the target's serial interface is selected, *avrdude* is used to determine the target platform's correct baud rate and the file is uploaded. If errors are detected, the user is notified of such.