ZBasic

Application Note

AN-103 Preparing ZBasic Generic Target Devices

Introduction

The various AVR devices that can be used as ZBasic generic target devices are supplied by Atmel with a certain default configuration that may or may not be appropriate for your application. For example, almost all such devices come from the factory running on an internal R-C oscillator (typically 8MHz). If the device has an internal divide-by-8 prescaler for the clock, it is usually enabled by default yielding a 1MHz operational frequency. Before "flashing" the device with your application, you will need to set the "fuses" of the AVR to configure it appropriately for your application. This application note describes some of the issues involved relating to selecting the proper fuse settings, installing the bootloader (if desired) and installing the application code.

AVR Device Programmers

In order to program the AVR fuses and/or program the Flash memory, a special device called a "programmer" is needed. These are sometimes referred to collectively as "ISP" (In System Programming) devices. The Atmel AVRISP2 product is highly recommended for this purpose. It is available from a variety of sources (i.e. Digi-Key's part number is ATAVRISP2-ND) for about $37. This device has a USB interface and can program all ATxmega, all ATmega and most ATtiny devices. There are also other AVR programmers available for a lower cost such as the various renditions of the open source USBasp originally developed by Thomas Fischl. These can often be found for under $10. There is also a variation of the USBasp called the USBtiny - one example of this is the AVR Pocket Programmer available from SparkFun for about $15.

One important thing to keep in mind is that Atmel devices use several different programming protocols. Some devices can only be programmed with one particular protocol while others may be programmable via several different protocols. Further, not all programming devices support all of the protocols. The table below summarizes the programming protocols used for various AVR devices.

Atmel Programming Protocols

Protocol	Description	Supported Devices
SPI	In-System Programming using the Serial Programming Interface interface. Requires four signal lines plus power/ground. For most AVR devices, the signal lines are the device's SPI interface: MOSI, MISO, SCK plus RESET.	Many ATtiny, most ATmega.
PDI	The Programming and Debug Interface is only used on xmega devices. Requires two signal lines plus power/ground.	All ATxmega.
HVPP	High Voltage Parallel Programming requires many signal lines and is not often used.	Many ATmega.
HVSP	High Voltage Serial Programming is similar in some ways to HVPP but requires many fewer signal lines.	Some ATtiny.
JTAG	This is actually a debugging protocol but it also provides programming capability. Requires four signal lines plus power/ground.	Most ATmega, all ATxmega.
DW	Debug Wire is actually a debugging protocol but it provides some programming capability (fuses cannot be set). Requires one signal line plus power/ground.	Many ATtiny.
TPI	The Tiny Programming Interface is similar in some respects to Debug Wire and is used only on ATtiny devices with a small number of pins.	Some (smaller) ATtiny.

Published January 2013

It is important to note that a particular programming device will likely be capable of programming using only a subset of the protocols listed above. For example, the AVRISP2 device only supports SPI, PDI and TPI. The USBasp and USBtiny devices typically support only SPI and PDI.

AVR Programmers Applications

Once you have a programming device, you'll also need some application software to use it to actually program AVR chips. The open source AVRdude application supports a wide variety of programming devices including both the AVRISP2 and USBasp. AVRdude is a command-line application that has a very capable (and somewhat complex) command set but there are tutorials and other information available on the Internet describing how to use it.

For the AVRISP2, Atmel's AVR Studio 4 application provides a GUI interface to the programming hardware but it also provide a command line application called STK500.exe that can be used to control the programmer that is somewhat easier to use than AVRdude. The corresponding program provided with AVR Studio 6 is called ATPROGRAM.exe.

In this application note, we will show AVRdude commands to use in conjunction with the SparkFun AVR Pocket Programmer for various operations.

Connecting the Programmer to the Target Device

Atmel has specified the layout of a six-pin, dual row (i.e., 2x3) connector for programming devices. The pins are numbered in the usual order when viewed from the top: (lower row, left to right) 1, 3, 5; (upper row, left to right) 2, 4, 6. The pin assignments for the SPI and PDI protocols are shown in the table below.

Atmel Programming Connector

Pin	SPI	PDI
1	MISO	PDI Data
2	Vcc	Vcc
3	SCK	
4	MOSI	
5	Reset	PDI Clock
6	Ground	Ground

In most cases, the target board must provide its own power and the presence of Vcc on the ISP connector is to allow the programmer to confirm that the target is powered. Some programmers are designed to optionally power the target via the Vcc ISP pin. Consult the documentation for your programmer to confirm its capabilities and requirements.

When designing or breadboarding your circuit, you must be sure not to have other circuits driving the signal pins needed for programming. Further, for xmega devices you must avoid having too much capacitance on the PDI Clock pin (same as the device RESET pin). Atmel has recommendations regarding connections to the programming pins in their application note AVR402: AVR Hardware Design Considerations (available from the Atmel website) along with other useful suggestions.

Choosing the Fuse Settings for your AVR

The "fuses" on AVR devices are simply another form of programmable memory that serves a particular purpose. Most AVR devices have a "low fuse byte" and a "high fuse byte"; more recent AVR devices also have an "extended fuse byte". The functions of each bit of each of the fuse bytes varies from one AVR device to another and often, some of the bits of one or more of the fuse bytes are not used. Because the fuse bit functions vary, we will use the ATmega644P as an example for discussion. In any event, you should read the section of the data sheet for your device that discusses the function of the fuse bits.

Published January 2013

One point of confusion regarding the fuse bits is that the documentation may refer to particular fuse bits as being "unprogrammed" or "programmed"; these refer to the logic states one and zero, respectively. The function of each of the fuse bits for the ATmega644P is shown in the table below. Discussion of individual fuse bits and groups of fuse bits follows. You may find it useful to use one of the "AVR Fuse Calculator" applications available online. One example of this can be found at http://www.engbedded.com/fusecalc.

ATmega644P Fuse Bytes

Bit	Extended	High	Low
7	*not used*	OCDEN	CKDIV8
6	*not used*	JTAGEN	CKOUT
5	*not used*	SPIEN	SUT1
4	*not used*	WDTON	SUT0
3	*not used*	EESAVE	CKSEL3
2	BODLEVEL2	BOOTSZ1	CKSEL2
1	BODLEVEL1	BOOTSZ0	CKSEL1
0	BODLEVEL0	BOOTRST	CKSEL0

As you may surmise from studying the table above, there are some individual fuses that control certain aspects of the AVR configuration while in other cases, groups of fuses control the configuration. For example, in the Extended fuse byte the three BODLEVELn fuses taken together control the configuration of the AVR's BrownOut Detector. By default, the BrownOut Detector is disabled (bit value 111) but three other possible bit combinations provide for a 1.8V, 2.7V and 4.3V threshold for the BrownOut Detector.

In the High fuse byte, the OCDEN and JTAGEN fuses control debugging capabilities. By default, the On Chip Debug capabilities are disabled and the JTAG interface is enabled. Generally speaking, you'll want to set both of these fuses to 1. The SPIEN fuse controls the ability to program the device using the SPI protocol. Generally speaking, you'll want to set this fuse to 0 to enable SPI programming. The WTDON fuse controls whether the WatchDog Timer is always on or not; the desired setting depends on the needs of your application. The EESAVE fuse controls whether the onboard EEPROM will be erased when a "erase chip" operation is performed; the desired setting depends on the needs of your application. The last three fuses of the High fuse byte are related to the bootloader. The two BOOTSZn fuse bits control the size of the boot section - for the ATmega644P the choices are 256 bytes, 512 bytes, 1024 bytes and 2048 bytes (N.B.: the datasheet and some fuse calculation applications discuss the sizes in terms of *words*). Lastly, the BOOTRST fuse controls whether the device begins executing code after a reset at address 0 (BOOTRST=0) or at the first byte of the boot section (BOOTRST=1); if you're using a bootloader you'll want the latter setting.

In the Low fuse byte, the CKDIV8 fuse controls whether the divide-by-8 prescaler for the system clock is enabled. When the prescaler is enabled, the frequency of the selected clock source is divided by 8 before being used as the main clock source. The CKOUT fuse determines whether the main clock signal is output on a specific pin; this may be useful in some applications but it is probably not enabled in most. The two SUTn fuses control how long the StartUp Time will be, specified as a combination of time and/or clock cycles. Generally speaking, you'll want to use a longer startup time to allow voltages and clock sources to stabilize before execution begins. Lastly, the four CKSELn fuses control the actual clock source to be used and you'll need to study carefully and understand well the effects of the various available values. The value that you select for these bits is probably the most critical of all the fuse values because an incorrect value can result in there being no functional clock thus disallowing further programming and setting of fuses - a condition commonly referred to as being "bricked". Fortunately, however, such devices are seldom truly "bricked" because there is a way to "rescue" them (discussed later).

One important aspect of programming a device using the SPI protocol is that the supplied SCK signal must be no more than one quarter of the operating frequency of the AVR. For an ATmega644P, the default fuse settings on virgin devices (internal 8MHz oscillator with divide-by-8 prescaler) result in a 1MHz operating frequency. This means that the SCK signal must not be faster than 250KHz. On the other hand, if you have a 16MHz crystal with the correct load capacitors connected, after you set the fuses to use the crystal and disable the divide-by-8 prescaler you could use an SCK signal as fast as 4MHz. The advantage of using a higher frequency is that the Flash programming operation will take less time overall at higher frequencies.

For a typical application having an ATmega644P with an 8MHz to 20MHz crystal and a 2K byte bootloader, the recommended fuse settings are: EXT=fd, HI=D4, LO=d7.

Published January 2013

Setting the Fuses

After making the connections of the programmer to the device to be programmed and applying power, it is highly recommended to first check that you can successfully read the fuses or the signature bytes of the device. Doing so will prove that the programmer is connected properly and can communicate successfully with the chip. The AVRdude command shown below reads the three fuse values and outputs them to the console. The command assumes use of the USBtiny programmer with an ATmega644P running at the factory default frequency.

```
avrdude -qq -cusbtiny -B8 -pm644p -Uefuse:r:con:h -Uhfuse:r:con:h -Ulfuse:r:con:h
```

For a virgin ATmega644P, the output from the command above should be:
```
0xff
0x99
0x62
```

You'll note from this output that the unassigned fuse bits (e.g. bits 7-3 of Ext fuses) read back as "unprogrammed" or logic 1. If you get an error message instead of the output shown above, chances are that the SPI connection is wired incorrectly or that the power is off on the target.

You can also read the device signature using the command shown below. This is a useful test of the programmer connection, bit rate, etc. Note that the AVRdude software (usually) reads the signature of a device and compares the result to the expected values; it will emit an error message if the signature does not match. The signature of an AVR device can be found in the datasheet for the device.

```
avrdude -qq -cusbtiny -B8 -pm644p -Usignature:r:con:h
```

For an ATmega644P, the output from the command above should be:
```
0x1e,0x96,0xa
```

The `-qq` option in the AVRdude commands above is not strictly necessary; it serves to reduce the amount of output generated by the program. You may want to experiment with the above commands without this option and with just `-q`.

The next step is to write the new fuse byte values using the AVRdude command below.

```
avrdude -qq -cusbtiny -B8 -pm644p -Uefuse:w:0xfd:m -Uhfuse:w:0xd4:m -Ulfuse:w:0xd7:m
```

After the fuses are successfully set to use an external crystal, you can again read out the fuse settings using a faster bit rate. The command below assumes a 16MHz crystal which allows a bit clock as fast as a 0.25uS period. The output should match the fuse values set in the preceding command.

```
avrdude -qq -cusbtiny -B0.25 -pm644p -Uefuse:r:con:h -Uhfuse:r:con:h -Ulfuse:r:con:h
```

Flashing the Bootloader

The Flash memory of the AVR can be programmed using a command like the one below. The file `atmega644p_boot.hex` is the ZBasic bootloader provided with recent ZBasic installations. Note, particularly, the file type indicator (`:i`) that specifies and Intel Hex File format.

```
avrdude -cusbtiny -B0.25 -pm644p -Uflash:w:atmega644p_boot.hex:i
```

With the USBtiny programmer and the bit rate set to 0.25uS, the operation above takes about 40 seconds to complete the writing process. By default, AVRDude also performs a verification pass to confirm that the Flash content matches the source file. Note that if the -B option were omitted the default bit rate of 1uS would be used causing the writing process to take about 160 seconds.

Published January 2013

For purposes of comparison, the Atmel AVRISP2 programmer can complete the writing operation of the same bootloader in less than a second.

Flashing the Bootloader And Application Together

Once the bootloader is installed, you can download applications to the device via the ZBasic IDE or using the zload.exe command line application. For production purposes, it is often convenient to flash a device with both the bootloader and the application code at the same time, thus avoiding the extra step of downloading the application. To accomplish this, you must produce a single .hex file containing both the bootloader and the application code. The command line application `srec_cat.exe` that is installed along with ZBasic is useful for this purpose. (You'll find `srec_cat.exe` in the `WinAvr/bin` and `avr-gcc/bin` subdirectories of the ZBasic installation directory.)

In order to generate an Intel Hex File for the ZBasic application, you must compile it with the `DeviceParameter ZBasicBootloader` set to zero (otherwise, the ZBasic IDE will generate a .zxb file). Assuming that you have an application file named `sample.hex`, the following command will produce a file named flash.hex containing the bootloader and application code.

```
srec_cat -output flash.hex -intel sample.hex -intel atmega644p_boot.hex -intel
```

Writing EEPROM

If your ZBasic application is compiled with `DeviceParameter ZBasicBootloader` set to zero, the ZBasic compiler will generate an Intel Hex file with the EEPROM contents - the filename will be the same as the project name but with a .eep extension. The AVRdude command below will write the `sample.eep` file to the device.

```
avrdude -cusbtiny -B0.25 -pm644p -Ueeprom:w:sample.eep:h
```

Setting Lock Bits

Once you've flashed your device with the bootloader and/or application code, you may want to set lock bits. There are three pairs of lock bits: one pair for the bootloader section, one pair for the application section, and one pair protecting the device from further programming and fuse bit changes. For most applications, it will be sufficient to set the lock bits to prevent the application from inadvertently writing in the bootloader section but please read the AVR datasheet for a full description of the lock bits to decide what is appropriate for your application.

The following command will set the lock bits to prevent writes to and reads from the bootloader section.

```
avrdude -cusbtiny -pm644p -Ulock:w:0x0f:m
```

Rescuing a "Bricked" Device

On occasion, you may inadvertently mis-program the fuses of a device causing it to no longer respond to the programmer. If the reason for the device not responding is that it no longer has a functioning clock (e.g. fuses set for a crystal but no crystal is present), more likely than not you can recover by supplying an external clock signal. The frequency of the signal isn't particularly important (but recall that the ISP clock frequency must be no more than one fourth of the effective clock frequency) but it should be a square wave with an excursion of zero to near Vcc. For most AVR devices that support an external clock signal, the pin to which the clock should be applied is labeled XTAL1, but check the device datasheet to be certain.

If you don't have a signal generator capable of producing the required square wave, you can often use another AVR chip to produce the signal. For example, an AVR that has the ability to output its clock (enabled by the

CKOUT fuse) is a good candidate. If all else fails, you can write a simple program that sits in a loop and toggles an output pin.

With the external clock signal applied and the programmer's ISP clock slowed appropriately, you should then be able to read the device signature and fuses. Once you have verified the basic operation you can then proceed to program the correct fuse values.

Author

Don Kinzer is the founder and CEO of Elba Corporation. He has extensive experience in both hardware and software aspects of microprocessors, microcontrollers and general-purpose computers. Don can be contacted via email at dkinzer@zbasic.net.