AVR911: AVR Open Source Programmer

Features

- **Open source C++ code**
- **Modular design**
- **Reads device information from AVR Studio's XML files**
- **Supports the Bootloader in AVR109**
- **Supports the In-System Programmer in AVR910**
- **Command-line equivalent to AVR Studio command-line tools**
- **Expandable to other programmer types**
- **Expandable to other communication channels, e.g. USB**

Introduction

The AVR Open Source Programmer (AVROSP) is an AVR programmer application equivalent to the AVRProg tool included in AVR Studio. It is a command-line tool, using the same syntax as the other command-line tools in AVR Studio.

The open source code and its modular design make it easy to port the application to other platforms and to add support for other programmer types and communication channels. Currently, AVROSP supports the programmers described in AVR109 and AVR910 through the standard PC serial port. The application note describes how to add more support.

AVROSP reads and writes Intel HEX files, and can use an existing AVR Studio installation to get required device parameters. This means that AVROSP automatically supports all devices supported by AVR Studio. No update is required for future AVR devices other than keeping your AVR Studio installation up to date.

8-bit AVR® Microcontrollers

Application Note

Rev. 2568A-AVR-07/04

Background and theory

From a user's point of view, programming an AVR device basically consists of the following steps: Wiring up the programmer to the device, preparing the binary files to be programmed and finally launching an application for the specific programmer in use. These steps are basically the same, regardless of programmer type. The AVROSP application tries to generalize this procedure into one application, thereby eliminating the need for different applications with different syntax and usage for each programmer type. Whether you are using a bootloader, an in-system programmer or a third-party programmer, the procedure is basically the same. AVROSP gives a consistent interface to the programming operation.

Application Note AVR109 and AVR910 describe Bootloader and an In-System Programmer (ISP). They both support the same operations - the Bootloader through the on-chip UART, and the ISP through a AT90S1200-based programmer. The Bootloader is compiled for a specific device, and naturally supports programming it. The ISP is not updated to directly support devices after ATmega163, but has a "Universal Command" that can be used to program all devices that support in-system programming. AVROSP supports both programmers. It reads the programmer's signature and decides which commands to use to communicate with the programmer. The user therefore does not need to specify the programmer type, provided that the programmer responds correctly to the "Read programmer ID"-command. Please refer to AVR109 and AVR910 for more information on the protocols.

A minimal set of device information required for programming (memory sizes, lock and fuse bits etc.) is available in the XML-type Part Description Files in an AVR Studio installation. AVROSP reads the information it needs from these files. If AVR Studio is not installed, e.g. if AVROSP is used in production programming or ported to other platforms (e.g. Linux), the Part Description Files could still be used. The application first searches the current directory, the AVROSP home directory, the directories specified in the PATH environment variable, and finally the AVR Studio installation. Therefore the Part Description Files could be copied to a directory in the PATH, and there is no need for installing AVR Studio.

The original Part Description Files are very large, and it takes some time to parse them. Therefore AVROSP creates a small XML file containing only the parameters of interest, and stores it in the AVROSP home directory. If updates are made to the original files, e.g. by an AVR Studio upgrade, the cached XML files should be deleted to tell AVROSP to regenerate them.

Currently, the Part Description Files do not contain any information on the specifics of each device's programming algorithm. The ISP module of AVROSP therefore implements the algorithm used in the most recent AVR devices. Due to this, in-system programming of some devices that use slightly different algorithms are not supported. This applies to the following devices: ATtiny12, ATtiny15, ATtiny26, ATtiny2313, AT90S1200, AT90S2313, AT90S2323/2343, AT90S4433, AT90S8515 and AT90S8535.

The bootloader module of AVROSP supports all devices with bootloader capabilities.

Note: ATtiny11 and ATtiny28 do not support ISP or bootloader programming, and are not supported by AVROSP. The user could customize the code to support other programmers, e.g. a serial high-voltage programmer.

Quickstart information

This section describes the necessary steps to get up-and-running quickly, if you have no need to modify or customize the application code

The executable file `avrosp.exe` is the only file required to use AVROSP. It is contained in the `avr911.zip` file that comes with this application note. The ZIP-file also contains the complete source code and a copy of the Part Description Files from an AVR Studio installation.

Copy the executable to a new directory and add the directory name to the PATH environment variable. If you do not want to install AVR Studio, copy the XML files to a subdirectory (to prevent the cached files from overwriting the original files), and add that too to the PATH. Now everything should be ready to use.

Note: The communication port settings (baud rate, parity control etc.) must be set manually before using AVROSP. For example, to use AVROSP with a bootloader communicating through COM1 with 115200 bps, no parity control and 8 data bits, run the following DOS command:

```
mode com1 baud=115200 parity=n data=8
```

Command-line syntax

All parameters must start with a minus, one or more characters and a number of optional values. There can be no spaces between the minus, the characters or the optional values. The order of the parameters is not important. In case of conflicting parameters, e.g. selecting both COM1 and COM2 for communication, the last parameter always counts. The supported command-line parameters are listed in Table 1.

Table 1. Command-line parameters

Parameter	Description
-d<*name*>	Device name. Must be applied when programming the device.
-if<*infile*>	Name of Flash input file. Required for programming or verification of the Flash memory. The file format is Intel Extended HEX.
-ie<*infile*>	Name of EEPROM input file. Required for programming or verification of the EEPROM memory. The file format is Intel Extended HEX.
-of<*outfile*>	Name of Flash output file. Required for readout of the Flash memory. The file format is Intel Extended HEX.
-oe<*outfile*>	Name of EEPROM output file. Required for readout of the EEPROM memory. The file format is Intel Extended HEX.
-s	Read signature bytes.
-O<*addr*>	Read oscillator calibration byte from device. *addr* is optional.
-O#<*value*>	User-defined oscillator calibration value. Use this to provide a custom calibration value instead of reading it from the device with –O<*addr*>.
-Sf<*addr*>	Write oscillator calibration byte to Flash memory. *addr* is byte address.
-Se<*addr*>	Write oscillator calibration byte to EEPROM memory. *addr* is byte address.

-e	Erase device. The device will be erased before any other programming takes place.
-p<*t*>	Program device. Set *t* to f for Flash, e for EEPROM or b for both. Corresponding input files are required.
-r<*t*>	Read out device. Set *t* to f for Flash, e for EEPROM or b for both. Corresponding output files are required.
-v<*t*>	Verify device. Set *t* to f for Flash, e for EEPROM or b for both. Can be used with –p<*t*> or alone. Corresponding input files are required.
-l<*value*>	Set lock byte. *value* is an 8-bit hex value.
-L<*value*>	Verify lock byte. *value* is an 8-bit hex value to verify against.
-y	Read back lock byte.
-f<*value*>	Set fuse bytes. *value* is a 16-bit hex value describing the settings for the upper and lower fuse bytes.
-E<*value*>	Set extended fuse byte. *value* is an 8-bit hex value describing the extend fuse settings.
-F<*value*>	Verify fuse bytes. *value* is a 16-bit hex value to verify against.
-G<*value*>	Verify extended fuse byte. *value* is an 8-bit hex value to verify against.
-q	Read back fuse bytes.
-x<*value*>	Fill unspecified locations with a value (00-FF). The default is to not program locations not specified in the input files.
-af<*start*>,<*stop*>	Flash address range. Specifies the address range of operations. The default is the entire Flash. Byte addresses in hex.
-ae<*start*>,<*stop*>	EEPROM address range. Specifies the address range of operations. The default is the entire EEPROM. Byte addresses in hex.
-c<*port*>	Select communication port, COM1 to COM8. If this parameter is omitted the program will scan the COM ports for a programmer.
-b<*t*>	Get attached programmer's revisions. Set *t* to h for hardware revision or s for software revision.
-g	Silent operation. No output to screen.
-z	No progress indicator. E.g. if piping to a file for log purposes, use this option to avoid the characters used for the indicator.
-h	Help information (overrides all other settings).
-?	Same as –h

Some examples follows:

```
avrosp –dATmega128 –pf –vf –ifprogram.hex –e
```

The above example will first erase the entire memory contents and then program and verify the data contained in `program.hex` to an attached Atmega128 device.

```
avrosp –dATmega32 –re –oedump.hex –ae0,ff –cCOM2
```

The above example will read the first 256 bytes of the Atmega32's EEPROM memory to the file `dump.hex`. Only COM2 will be used.

```
avrosp –dATmega64 –O#a0 –Se0 –lc0
```

The above example will write the custom oscillator calibration value $A0_{hex}$ into EEPROM address 0_{hex} and then protect the Atmega64's memory by writing the lock byte to $C0_{hex}$.

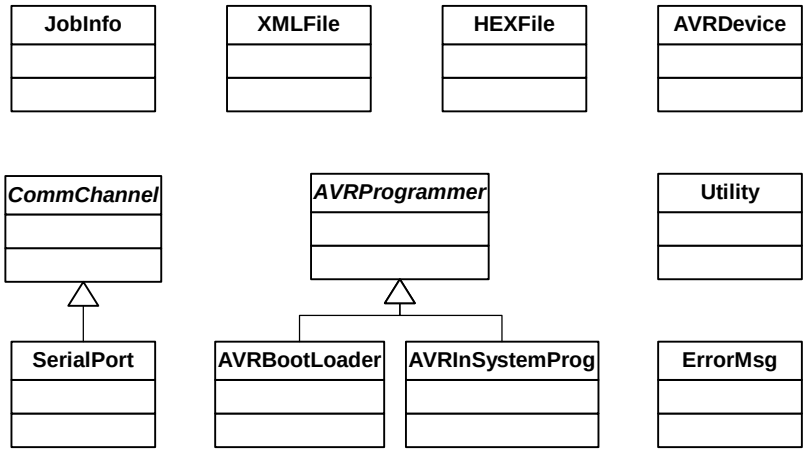
Note: When programming fuse bits, the bit pattern is not checked to be a valid fuse setting for the device. Care should be taken not to program invalid fuse settings, as this could render the device inoperable. High-voltage programming could be the only way to recover from such a situation.

Implementation

This section assumes that the reader has some knowledge of object-oriented programming concepts, and the C++ programming language in particular.

The source code is free in all ways, meaning that users can modify and enhance the application and redistribute it as they wish. More information on free software is available at the following URL: http://www.gnu.org/philosophy/free-sw.html.

Figure 1. AVROSP Class Diagram



Most of the top-level work is encapsulated in the `JobInfo` class. It uses objects of class `XMLFile`, `HEXFile` and `AVRDevice` to read and write XML and HEX files and to extract device information from the Part Description Files.

The two helper classes `Utility` and `ErrorMsg` are used throughout the application.

The part of `JobInfo` that communicates with the programmer does not need to know what kind of communication channel to use. It decodes the command line and creates an instance of the required derived class, e.g. the `SerialPort` class. The rest of the code just works through the generalized `CommChannel` parent class. Currently, only a class for the PC COM port is implemented, but to use e.g. USB or TCP/IP communication, you could derive a specialized class from the `CommChannel` base class, and add a check for this channel type in the command line parser.

The same method is used for the programmer type. The code that operates on the programmer does not need to know which type of programmer is attached. The `JobInfo` class retrieves the programmer ID string and creates an appropriate object for the specific programmer. The rest of the code operates through the generalized `AVRProgrammer` interface. Currently, only classes for the Bootloader described in Application Note AVR109 and the In-System Programmer described in Application Note AVR910 are implemented. However, you could derive your own specialized

programmer from the AVRProgrammer base class, and add a check for it in the ID string decoding part of JobInfo.

This design makes the application very flexible. Future extension with other communication channels and programmer types is an easy task.

Class Descriptions

Only the public interface methods are described here. The reader should refer to the commented code for detailed information on the inner workings of the various classes. Each class is described with a brief introduction to the class' purpose and then each of its public methods is described with return type, parameters and purpose.

AVRDevice

This class provides a container for relevant parameters for the current programmed device, such as memory sizes and signature bytes. The class also contains functionality for retrieving device information from an AVR Studio installation.

AVRDevice

This is the constructor for the class. It takes one string parameter, the name of the device to retrieve information for. The parameters are not retrieved automatically. The class provides a method for reading the information for AVR Studio, but derived classes could implement other means of getting this information.

~AVRDevice

This is the destructor for the class. It currently has no function, just a placeholder for future extensions.

readParametersFrom AVRStudio

This searches the current directory, the application home directory, directories in the PATH environment variable and, if available, the AVR Studio installation for the required XML file. To get the path for the AVR Studio installation, the Windows Registry Database is queried for the key named "HKEY_LOCAL_MACHINE\Software\Atmel\AVRTools\AVRToolsPath". The method then parses the XML file and retrieves the necessary information. An exception is thrown if the file is not found or if the file contains errors. The method takes a list of search paths as parameter, and returns no value.

getFlashSize

This is an access method for the Flash memory size parameter. It takes no parameters, and returns a long value indicating the number of Flash bytes.

getEEPROMSize

This is an access method for the EEPROM memory size parameter. It takes no parameters, and returns a long value indicating the number of EEPROM bytes.

getPageSize

This is an access method for the Flash page size parameter. It is only valid for AVR parts with Flash pages. Other parts return the value –1. The method takes no parameters, and returns a long value indicating the number of bytes in each Flash page.

getFuseStatus

This is an access method for checking if the device has fuse bits. It takes no parameters, and returns true if there is a "FUSE" section in the XML file, false otherwise.

getXFuseStatus

This is an access method for checking if the device has extended fuse bits. It takes no parameters, and returns true if the device has extended fuses, false otherwise.

getSignature

This is an access method for retrieving the signature bytes for the device. Note that this is not the signature bytes read from the actual device, but the signature read from the XML files. The method takes three long pointers as parameters, and returns no value. The signature bytes are copied to the variables pointed to by the parameters. An exception is thrown if any of the pointers are null pointers.

AVRProgrammer

This class is an abstract class, providing a framework for implementing an interface for a specific AVR programmer, e.g. a boot loader or an in-system programmer. Almost all methods are virtual and empty and must be overloaded by derived classes.

	The programmer works through a generalized byte-oriented communication channel, which must be provided when creating an instance of this class.
AVRProgrammer	This is the constructor for the class. It takes one parameter, a pointer to a `CommChannel` object. This is the channel used for later communication with the actual programmer. An exception is thrown if a `null` pointer is provided for the communication channel.
~AVRProgrammer	This is the destructor for the class. It currently has no function, just a placeholder for future extensions.
readProgrammerID	This is a static method for reading a connected programmer's ID-string. This method sends a 'S' to the programmer through the communication channel supplied in the method's only parameter, a pointer to a `CommChannel` object. Use this method to decide which derived programmer object to create. The method returns the seven-character ID-string of the programmer. An exception is thrown if a `null` pointer is provided for the communication channel, or if an error occurs during communication.
setPagesize	This is an access method for the programmer's Flash page size information. This must be used prior to any Flash operations. The method takes one `long` parameter, the Flash page size in bytes, and returns no value.
enterProgrammingMode	This method is virtual and not implemented in this abstract base class. It should tell the attached programmer to enter programming mode. The method takes no parameters, and returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.
leaveProgrammingMode	This method is virtual and not implemented in this abstract base class. It should tell the attached programmer to leave programming mode. The method takes no parameters, and returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.
chipErase	This method is virtual and not implemented in this abstract base class. It should tell the attached programmer to erase the attached device's contents. The method takes no parameters, and returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.
readOSCCAL	This method is virtual and not implemented in this abstract base class. It should read one of the oscillator calibration bytes from the attached device. The method takes two parameters, a `long` indicating which OSCCAL value to read and a `long` pointer to a variable to which the value should be copied. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.
readSignature	This method is virtual and not implemented in this abstract base class. It should read the signature bytes from the attached device. The method takes three parameters, three `long` pointers to the variables to which the values should be copied. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.
checkSignature	This method is virtual and not implemented in this abstract base class. It should check the supplied signature bytes against the attached device. The method takes three parameters, three `long` parameters containing the signature to be checked. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or if the supplied signature does not match the attached device's signature.

writeFlashByte — This method is virtual and not implemented in this abstract base class. It should write a byte to the attached device's Flash memory. The method takes two parameters, a long indicating the Flash byte address and a long containing the byte value to write. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.

writeEEPROMByte — This method is virtual and not implemented in this abstract base class. It should write a byte to the attached device's EEPROM memory. The method takes two parameters, a long indicating the EEPROM byte address and a long containing the byte value to write. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.

writeFlash — This method is virtual and not implemented in this abstract base class. It should write the contents of the supplied HEX file object to the attached device's Flash memory. The method takes one parameter, a pointer to the required `HEXFile` object. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

readFlash — This method is virtual and not implemented in this abstract base class. It should read data from the attached device's Flash memory into the supplied HEX file object. The method takes one parameter, a pointer to the required `HEXFile` object. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

writeEEPROM — This method is virtual and not implemented in this abstract base class. It should write the contents of the supplied HEX file object to the attached device's EEPROM memory. The method takes one parameter, a pointer to the required `HEXFile` object. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

readEEPROM — This method is virtual and not implemented in this abstract base class. It should read data from the attached device's EEPROM memory into the supplied HEX file object. The method takes one parameter, a pointer to the required `HEXFile` object. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

writeLockBits — This method is virtual and not implemented in this abstract base class. It should set the attached device's lock bits to the supplied value. The method takes one parameter, a `long` containing the desired lock bits. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.

readLockBits — This method is virtual and not implemented in this abstract base class. It should read the attached device's lock bits. The method takes one parameter, a pointer to a `long` to which the lock bits should be copied. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

writeFuseBits — This method is virtual and not implemented in this abstract base class. It should set the attached device's fuse bits (both low and high byte) to the supplied value. The method takes one parameter, a `long` containing the desired fuse bits. The method

returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.

readFuseBits — This method is virtual and not implemented in this abstract base class. It should read the attached device's fuse bits (both low and high byte). The method takes one parameter, a pointer to a `long` to which the fuse bits should be copied. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

writeExtendedFuseBits — This method is virtual and not implemented in this abstract base class. It should set the attached device's extended fuse bits to the supplied value. The method takes one parameter, a `long` containing the desired extended fuse bits. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur.

readExtendedFuseBits — This method is virtual and not implemented in this abstract base class. It should read the attached device's extended fuse bits. The method takes one parameter, a pointer to a `long` to which the extended fuse bits should be copied. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

programmerSoftwareVersion — This method is virtual and not implemented in this abstract base class. It should read the programmer's software version. The method takes two parameters, two `long` pointers to variables to which the major and minor version number should be copied, respectively. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

programmerHardwareVersion — This method is virtual and not implemented in this abstract base class. It should read the programmer's hardware version. The method takes two parameters, two `long` pointers to variables to which the major and minor version number should be copied, respectively. The method returns `true` if this operation is supported by the programmer, `false` otherwise. An exception should be thrown if any communication errors occur, or a `null` pointer is provided.

AVRBootloader — This is a class derived from the `AVRProgrammer` base class. It provides a generic interface for the Bootloader application described in Application Note AVR109. It overrides all the virtual methods from its base class, and provides a constructor with equivalent parameters.

AVRInSystemProg — This is a class derived from the `AVRProgrammer` base class. It provides a generic interface for the In-System Programmer application described in Application Note AVR910. It overrides all the virtual methods from its base class, and provides a constructor with equivalent parameters.

CommChannel — This class is an abstract class, providing a framework for implementing a byte-oriented communication channel. All methods are virtual and empty and must be overloaded by derived classes.

~CommChannel — This is the destructor for the class. It currently has no function, just a placeholder for future extensions.

openChannel — This method should perform the necessary operation to open the communication channel. This should always be called prior to any communication. The method takes no parameters, and returns no value. An exception should be thrown if any errors occur.

closeChannel — This method should perform the necessary operation to close the communication channel. This should always be called when all communication is finished. The method takes no parameters, and returns no value. An exception should be thrown if any errors occur.

sendByte — This method should send a byte to the communication channel. The method takes one parameter, a `long` containing the byte to be sent. The method returns no value. An exception should be thrown if any communication errors occur, or the channel has not been opened.

getByte — This method should wait for and receive a byte from the communication channel. The method takes no parameters, and returns a `long` containing the received byte. An exception should be thrown if any communication errors occur, or the channel has not been opened.

flushTX — This method should flush the transmit buffer. The method takes no parameters, and returns no value. An exception should be thrown of any communication errors occur, or the channel has not been opened.

flushRX — This method should flush the receive buffer. The method takes no parameters, and returns no value. An exception should be thrown of any communication errors occur, or the channel has not been opened.

sendMultiple — This method should send an array of bytes to the communication channel. The method takes two parameters, a pointer to the first `unsigned char` in the array and a `long` indicating the array size in bytes. The method returns no value. An exception should be thrown if any communication errors occur, or the channel has not been opened.

SerialPort — This is a class derived from the `CommChannel` base class. It provides a generic communication channel interface for the standard PC COM port. It overrides all the virtual methods from it's base class, and has it's own specific constructor.

SerialPort — This is the constructor for the class. It initializes the port, but does not open the communication channel. The constructor takes two parameters, a `long` containing the COM port number (1 to 8) and a `long` containing the communication timeout limit in seconds. An exception is thrown is an invalid port number or timeout value is provided.

HEXFile — This is a class providing basic functionality for reading and writing Intel Extended HEX files. It also has methods for defining the memory range to be used. This is useful for reading or writing only parts of the AVR memories.

HEXFile — This is the constructor for the class. It takes two parameters, a `long` indicating the required maximum data buffer size and a `long` containing the default byte value to be used when initializing the buffer. An exception is thrown if not enough memory is available.

~HEXFile — This is the destructor for the class. It deallocates all previously allocated memory.

readFile — This method reads data from a HEX file. The method takes one parameter, the HEX file name, and returns no value. An exception is thrown if any file access errors occur, or the file format is invalid.

writeFile — This method writes data to a HEX file. The method takes one parameter, the HEX file name, and returns no value. An exception is thrown if any file access errors occur.

setUsedRange — This method overrides the memory range indicators. This can be used to limits the range for read and write operations. The method takes two parameters, two `long`

	variables containing the new start and end limits, respectively. The method returns no values. An exception is thrown if the provided range is invalid.
clearAll	This method sets the entire data buffer to the desired byte value. The method takes one parameter, a long containing the desired byte value, and returns no value.
getRangeStart	This is an access method for the start address of the current range. The method takes no parameters, and returns the start address.
getRangeEnd	This is an access method for the end address of the current range. The method takes no parameters, and returns the end address.
getData	This is an access method for the data in the buffer. It takes one parameter, a `long` containing the byte address, and returns a `long` containing the byte value. An exception is thrown if the address is outside legal ranges.
setData	This is an access method for setting the data in the buffer. It takes two parameters, a long containing the byte address and a `long` containing the byte value. An exception is thrown if the address is outside legal ranges.
getSize	This is an access method for retrieving the buffer size. The method takes no parameters, and returns a `long` containing the buffer size in bytes.
XMLFile	This class provides a simple XML parser for reading the AVR Part description files that come with AVR Studio. It can also be used in other projects for general XML parsing. Note that the class does not support attributes inside tags, although no errors are generated if such tags are encountered. The attributes are simply ignored. The class reads the entire XML file and builds a memory resident tree from the contained information.
XMLFile	This is the constructor for the class. It takes one string parameter, the name of the XML file to be parsed. The constructor reads the file immediately, so if no exceptions are thrown, the XML tree is built and ready when the constructor finishes.
~XMLFile	This is the destructor for the class. It deallocates all memory previously allocated for the memory resident XML tree.
exists	This method checks if a node exists at a given path. The method takes one string parameter, the full path including the node name, and returns `true` if the node exists, `false` otherwise.
getValue	This method retrieves a node's value from the XML tree. The method takes one string parameter, the full path including the node name, and returns the `string` value of the node. An exception is thrown if the node is not found. Use the `exists` method to ensure that the node exists.
print	This method prints the entire XML tree contents in a short format. The method was originally implemented for debugging purposes. It takes no parameters, and returns no value.
JobInfo	This is a class holding all information extracted for the command line parameters. The class also contains the functionality for performing the necessary operations.
JobInfo	This is the constructor for the class. It initializes all information to default values.
parseCommandline	This method parses the command line parameters. It takes two parameters, the familiar `int argc` and `char *argv[]` from the `main()` function. The method returns no parameters. An exception is thrown if any invalid parameters are encountered.

doJob	This method performs all the work necessary to fulfill all tasks extracted from the command line parameters. It also takes care of creating the required communication channel and programmer objects. The method takes no parameters, and returns no value. An exception is thrown if any errors occur.
Utility	This class serves as a container and namespace for often used functions. It is instantiated in the source file, and an external reference to an `Util` object is provided in the header file. It is especially used for log and progress messages, and for enabling silent operation.
Utility	This is the constructor for the class. It takes no parameters. The constructor initializes the internal log and progress status to enable both log and progress messages.
~Utility	This is the destructor for the class. It currently has no function, just a placeholder for future extensions.
muteLog	This method prevents all further log messages from being display on screen. It takes no parameters, and returns no value.
muteProgress	This method prevents all further progress messages from being display on screen. It takes no parameters, and returns no value.
log	This method prints log-type messages to the screen, if not muted. The method takes one parameter, the message string. The method returns no value.
progress	This method prints progress-type messages to the screen, if not muted. The method takes one parameter, the message string. The method returns no value.
convertHex	This method converts a hexadecimal string to a number. The method takes one parameters, the string, and returns a `long` containing the converted number. An exception is thrown if any conversion errors occur.
convertLong	This method converts a number to a string, using a specified radix. The method takes two parameters, a long containing the number to be converted and a long containing the desired radix to be used.
getRegistryValue	This method retrieves a value from the Windows registry database. The method takes two parameters, a string containing the registry key path and a string containing the key name. The method returns a string containing the retrieved value. An exception is thrown if any errors occur during the database operations.
ErrorMsg	This class serves as a container for error messages to be thrown as exceptions.
ErrorMsg	This is the constructor for the class. It takes one parameter, the error message string.
~ErrorMsg	This is the destructor for the class. It currently has no function, just a placeholder for future extensions.
What	This is an access method for the error message string. It takes no parameters, and returns a copy of the error message.

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
Tel: (41) 26-426-5555
Fax: (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
Tel: (33) 2-40-18-18-18
Fax: (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
Tel: (33) 4-42-53-60-00
Fax: (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
Tel: (44) 1355-803-000
Fax: (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
Tel: (49) 71-31-67-0
Fax: (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-30-00
Fax: (33) 4-76-58-34-80

Literature Requests
www.atmel.com/literature

2568A-AVR-07/04