AVR Programming Project

Documentation of AVRprog software development

Project team: Michael Bäuerle
Postweg 2
70736 Fellbach−Schmiden

Markus Bernauer
Seminarstraße 2
73230 Kirchheim

Last modification: 2006−02−04 by Michael Bäuerle

Content

Overview

History

This project was started to develop a complete solution to program Atmel AVR Microcontrollers in a UNIX environment. Porting it to other OS's was kept in mind.

At the beginning we have no information about the 'Micro ISP' and 'Atmel STK300' programmers we used with Windows software before this project was started.

To solve this problem Markus developed the 'Serial 1' programmer that uses the RS−232 to transfer data between PC and programmer (Note: The term "serial" means programming the AVR in the low voltage serial (aka ISP) mode rather than the high voltage parallel mode. It means NOT the serial RS−232 interface).

This programmer was the first one supported by the UNIX software AVRprog that we developed later.

Because we like to do things right or not at all, we create this docu to fix project related information.

AVRprog features:

- Frontend with GUI
  Text based frontends are also possible but do not exist at the moment.
- Support for different programmer types
  Serial1, STK200, STK300, AVR910, ATK500 and AE−AVR PRG programmers as well as AVR109 bootloaders are supported at the time of writing.
- Support for different interfaces
  RS−232 and IEEE−1284 (aka Centronics) at the moment.
- Supports network connection to use a programmer on a different machine
  The network support makes it possible to port only designated components to other platforms.
- Modular construction
  You have to change only one component to support a new programmer, change the user interface or add network transparent programmer access.
- Well defined interface between components
- Integrated uniform error naming and reporting scheme
  All functions use the same error numbers (sense codes) that always have the same meaning.

The software is intended for easy use − with less menues and self explaining GUI. As much as possible should be handled by the software − only the things that are really needed should be given by the user.

General

The software consist of several components that are stacked.

The main components are:

- Frontend: User interface. Communicate with the backend via a unified API
- [Netlayer: Provides a transparent connection between frontend and backend using the network. Can be omitted]
- Backend: Communicate with the programmer hardware via a driver library. Provide a unified API for different programmer hardware to the frontend
- Drivers: Used by the backend to communicate with different programmer hardware (The programmer drivers are treated as part of the backend for this implementation. The driver API documentation is shipped with the backend)

To give an overview here is the layout of the construction:

(In braces the names of the files that do the job for the Serial1 programmer in UNIX are specified)

In the local configuration the construction looks like this:

Frontend (frontend)

Backend (libbackend.so)

Driver (libserial1.so)

Serial1 Programmer

With network support like this:

Frontend (frontend)

Netlayer client (libnetclient.so)

−−−−−−−−−−−−−−−−−−−−−−−−−−−− [network hop]

Netlayer server (AVRprogd)

Backend (libbackend.so)

Driver (libserial1.so)

Serial1 Programmer

In an outside view all components above the line can be called "Client" and all modules below the line can be called "Server".

Platform and OS support

AVRprog was designed to be platform independent. It should be possible to port it to nearly all sort of systems. Being maximum user friendly combined with avoiding of bloat was the major design goal (apart from the fun ;−).

The UNIX version is an implementation based on shared libraries (all modern systems support them), but shared libraries are not a requirement. The DOS backend and netlayer server demonstrate an implementation using static libraries and an integrated TCP/IP network protocol stack.
Finally it should also be possible to completely integrate an AVRprog server into a programmer that has a network interface build in ...

Note: Not all components are available for all platforms!

The following operating systems on the listed platforms have been tested to work:

UNIX:

- GNU/Linux 2.2
- GNU/Linux 2.4
- GNU/Linux 2.6
- NetBSD 2.0

- GNU/Linux 2.2
- SunOS 5.7 (aka "Solaris 7")

- HP−UX 11.11

- AIX 4.3.3

DOS:

- Microsoft DOS 6.2.0

Don’t ask why there is no Windows version ... nobody has written one yet and we don’t like Window (Nevertheless it should be no problem to write one).

Please report success or failure if you have tested AVRprog on a platform/OS combination not listed above.

Backend API V0.2 (used by frontend)

Function prototypes

[To use them: Include ’backend.h’]

set_device_file(A5)	Interface where programmer is connected to
programmer_enable(A0)	Establish connection to programmer
programmer_disable(A0)	Shutdown connection to programmer
backend_info(A1)	Get backend information (version)
driver_info(A1)	Get driver information (driver, version)
request_sense(A6)	Get additional information on last error
progmode_enable(A0)	Set MCU in programming mode
progmode_disable(A0)	Set MCU in running mode
read_devcode(A1)	Get MCU device code
chip_erase(A0)	Erase flash and EEPROM on MCU
read_flash(A3)	Read from MCU flash memory
write_flash(A3)	Write to MCU flash memory
read_eeprom(A4)	Read from MCU EEPROM memory
write_eeprom(A4)	Write to MCU EEPROM memory
write_lock(A2)	Write MCU lock bits

Return values

[Type: (unsigned char)]

0 : OK

1 : Error

2 : Not supported by programmer driver library

Function parameter

[All strings (char*): 256 characters]

A0 : (void)

A1 : (unsigned char * string)

A2 : (unsigned char lockbyte)

A3 : (unsigned short int address, unsigned short int count, unsigned short int* data)

A4 : (unsigned short int address, unsigned short int count, unsigned char* data)

A5 : (char* device_file, unsigned char if_type)

A6 : (unsigned char* sense_code)

Interface type (A5: if_type)

[Type: (unsigned char)]

0 : RS−232

Sense codes (A6: sense_code)

[Type: (unsigned char)]

0 : No sense

1 : Interface type not supported

2 : Device file not exist or have wrong type

3 : Cannot open device file

4 : No programmer found

5 : Programmer not enabled

6 : Cannot close device file

7 : Communication error

8 : Communication timeout

9 : Cannot configure device file

Backend API V0.3 (used by frontend and netlayer client)

Function prototypes

[To use them: Include 'backend.h' (for single application) or 'netclient.h' (for client/server)]

Group 1 contain functions to establish and shutdown a connection to the programmer, get version numbers of programs and interfaces and for error recovery.

Group 2 contains functions for controlling the programmer.

Group 1:

set_network_address(A1)	Set network address of machine where p rogrammer is attached to
set_device_file(A5)	Interface where programmer is connected to
programmer_enable(A0)	Establish connection to programmer
programmer_disable(A0)	Shutdown connection to programmer
backend_info(A1)	Get backend information (version)
driver_info(A1)	Get driver information (driver, version)
network_info(A7)	Get network layer information (versions)
request_sense(A6)	Get additional information on last error

Group 2:

progmode_enable(A0)	Set MCU in programming mode
progmode_disable(A0)	Set MCU in running mode
read_devcode(A8)	Get MCU device code
chip_erase(A0)	Erase flash and EEPROM on MCU
read_flash(A3)	Read from MCU flash memory
write_flash(A3)	Write to MCU flash memory
read_eeprom(A4)	Read from MCU EEPROM memory
write_eeprom(A4)	Write to MCU EEPROM memory
write_lock(A2)	Write MCU lock bits

Return values

[Type: (unsigned char)]

0 : OK

1 : Error

2 : Not supported

Function parameter

[Data arrays: max. 64K − 1 elements Strings: max. 255 chars (including trailing zero)]

A0 : (void)

A1 : (char * string)

A2 : (unsigned char lockbyte)

A3 : (unsigned short int address, unsigned short int wordcount, unsigned short int* data)

A4 : (unsigned short int address, unsigned short int bytecount, unsigned char* data)

A5 : (char* device_file, unsigned char if_type)

A6 : (unsigned char* sense_code)

A7 : (char * string, char * string)

A8 : (unsigned char* device_code) [0x1E at low address]

Interface type (A5: if_type)

[Type: (unsigned char)]

0 : RS−232

1 : IEEE−1284

Sense codes (A6: sense_code)

[Type: (unsigned char)]

0 : No sense

1 : Interface type not supported

2 : Device file not exist or have wrong type

3 : Cannot open device file

4 : No programmer found

5 : Programmer not enabled

6 : Cannot close device file

7 : Communication error (Programmer driver)

8 : Communication timeout (Programmer driver)

9 : Cannot configure device file

10 : Cannot open network connection

11 : Communication error (Network)

12 : No open network connection

13 : MCU not supported

14 : MCU locked

Backend API V0.4 (used by frontend and netlayer client)

Function prototypes

[Include "backend.h" (for single applicatiom) or "netclient.h" (for client/server)]
Mandatory functions are printed **bold** and are not allowed to return 2 (not supported).

Group 1 Functions that change state of AVRprog
Group 2 Functions that do not change state of AVRprog

Group 1:

programmer_enable(P0)	Establish connection to programmer
programmer_disable(P0)	Shutdown connection to programmer
progmode_enable(P0)	Set device into programming mode
progmode_disable(P0)	Set device into running mode

Group 2:

request_sense(P6)	Get additional information on last error
set_network_address(P1)	Set network address of server machine
set_device_file(P5)	Set Interface to which the programmer is connected
set_prg_method(P2)	Select programming method
network_info(P7)	Get network layer information (versions)
backend_info(P1)	Get backend information (version)
driver_info(P1)	Get driver information (driver, version)
read_devcode(P8)	Get MCU device code
chip_erase(P0)	Erase and unlock device
read_flash(P3)	Read from Flash−EPROM memory
write_flash(P3)	Write to Flash−EPROM memory
read_eeprom(P4)	Read from EEPROM memory
write_eeprom(P4)	Write to EEPROM memory
read_lock(P9)	Read lock bits
write_lock(P2)	Write lock bits
read_fuses(P10)	Read fuse bits
write_fuses(P11)	Write fuse bits
read_osc_calib(P12)	Read oscillator calibration byte(s)

Return values

[Type: (unsigned char)]

0 : OK

1 : Error

2 : Not supported

Function parameter

[Data arrays if not specified: max. 64K elements; Strings if not specified: max. 255 chars]

P0 : (void)

P1 : (char* string)

P2 : (unsigned char value)

P3 : (unsigned short int address, unsigned short int wordcount,
unsigned short int* data)

P4 : (unsigned short int address, unsigned short int bytecount,
unsigned char* data)

P5 : (char* device_file, unsigned char if_type)

P6 : (unsigned char* sense_code)

P7 : (char* string, char* string)

P8 : (unsigned char device_code[3]) [0x1E at low address]

P9 : (unsigned char* value)

P10 : (unsigned char mask, unsigned char* low, unsigned char* high,
unsigned char* extended) ['mask': Set bits indicate valid fuse bytes]

P11 : (unsigned char mask, unsigned char low, unsigned char high,
unsigned char extended) ['mask': Set bits indicate valid fuse bytes]

P12 : (unsigned char count, unsigned char* buffer)

Interface type (P5: 'if_type')

[Type: (unsigned char)]

0 : RS−232

1 : IEEE−1284

Programming method (P2: 'value' for 'set_prg_method()')

[Type: (unsigned char)]

0 : LVS ('Low Voltage Serial' aka 'ISP') [Default if nothing is set]

1 : HVP ('High Voltage Parallel')

2 : HVS ('High Voltage Serial')

3 : JTAG

Sense codes (P6: sense_code)

[Type: (unsigned char)]

0 : No sense

1 : Interface type not supported

2 : Device file not exist or have wrong type

3 : Cannot open device file

4 : No programmer found

5 : Programmer not enabled

6 : Cannot close device file (obsolete, for compatibility only)

7 : Communication error (Programmer driver)

8 : Communication timeout (Programmer driver)

9 : Cannot configure device file

10 : Cannot open network connection

11 : Communication error (Network)

12 : No open network connection

13 : MCU not supported

14 : MCU locked

15 : Programming mode not enabled

16 : Programming method not supported

Backend

General

The backend is used to provide a unified API for different programmer hardware.

It uses autodetection to find out the programmer type. The frontend don't have to care for different programmers.

Multiple programmers can be connected to different interfaces of the (server) machine. They are selected with the 'set_device_file()' call. Since API V0.4 every programmer can support multiple programming techniques. They are selected with the 'set_prg_method()' call.

All functions for the used version of the Backend API must be implemented, but it is legal to always return 2 (not supported) − Exept the functions printed **bold**, they are mandatory and are not allowed to return 2 (not supported).

All components should use the same interface version but all APIs with the same major number are backward compatible. This means that old frontends or netlayer servers can use new backends (but old backends or netlayer clients do not work with new frontends).

Function call order

API V0.2:

- Error recovery: 'request_sense()' after return code '1' at every stage
- Open connection: ['backend_info()'] −> '**set_device_file()**' −> '**programmer_enable()**' [−> 'driver_info()']
- Use connection: '**progmode_enable()**' [−> all others] −> '**progmode_disable()**'
- Close connection: '**programmer_disable()**'

API V0.3:

- Error recovery: 'request_sense()'
  Allowed after return code '1' at every stage. Not allowed after other return codes.
- Open connection: 'set_network_address()' [−> 'network_info()'] −> ['backend_info()'] −> '**set_device_file()**' −> '**programmer_enable()**' [−> 'driver_info()']
  The optional '*_info()' calls are allowed at every position. The specified position shows the earliest time they return valid information.
- Use connection: '**progmode_enable()**' [−> all others] −> '**progmode_disable()**'
  You never have to disable programming mode except you want to do so even if the datasheet of the MCU specify this (for chip erase or such things) − the backend will do this automatically if it is needed.
- Close connection: '**programmer_disable()**' −> 'set_network_address("disconnect")

If there is no netlayer present, 'set_network_address()' returns 2 (not supported) and the following 'network_info()' as well as the final disconnect will also return 2.

Always implement the specified call order to ensure compatibility.

API V0.4:

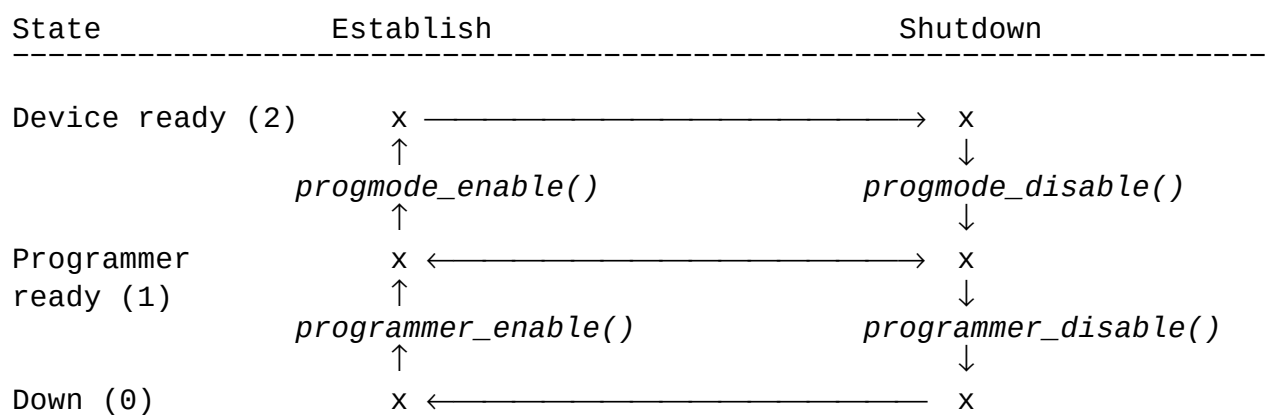
This API introduces a state machine. AVRprog can be in one of 3 major states:

1) Down

2) Programmer ready

3) Device ready

Only the 4 functions in the following diagram change the state when returning 0 (OK):



If one of these functions return 1 (Error), the state is unchanged.

Remember:
The backend never do implicit state changes − even if an illegal call order is used!

The following list specify which functions are allowed in which state (Mandatory functions are printed **bold**, state changing functions are printed *italic*):

Function calls allowed in State 0 (’Down’):

- ’set_network_address()’
- ’**set_device_file()**’
- ’***programmer_enable()***’
- ’get_network_info()’
- ’get_backend_info()’
- ’**request_sense()**’

Order:

[’set_network_address()’ => ’get_network_info()’] => ’**set_device_file()**’
=> ’***programmer_enable()***’

’**request_sense()**’ is valid if a function have returned 1 (Error)

’get_backend_info()’ is independent

Function calls allowed in State 1 (’Programmer ready’):

- ’***programmer_disable()***’
- ’set_prg_method()’
- ’***progmode_enable()***’
- ’get_network_info()’
- ’get_backend_info()’
- ’get_driver_info()’
- ’**request_sense()**’

Order:

[’set_prg_method()’] => ’***progmode_enable()***’

’**request_sense()**’ is valid if a function have returned 1 (Error)

All other are independent

Function calls allowed in State 2 (’Device ready’):

- ’***progmode_disable()***’
- ’get_network_info()’
- ’get_backend_info()’
- ’get_driver_info()’
- ’chip_erase()’
- ’read_lock()’
- ’write_lock()’
- ’read_fuse()’
- ’write_fuse()’
- ’read_osc_calib()’
- ’write_osc_calib()’
- ’read_flash()’
- ’write_flash()’
- ’read_eeprom()’
- ’write_eeprom()’
- ’**request_sense()**’

Order:

’**request_sense()**’ is valid if a function have returned 1 (Error)

All other are independent

Note:
If the user request to quit, the frontend must go back to ’Down’ state before terminating.

Error recovery

Use the ’request_sense()’ function to get information what goes wrong. If ’request_sense()’ returns 0 (OK) with sense code 0 (No sense), there is no information about the error.

Frontend design notes:
- The backend stay in the current state when 1 (Error) is returned by one of the state changing functions ... This means you have to retry or proceed with the corresponding function in the diagram (see previous section) to ensure the correct function call order.
- Do not restart from the beginning until you are in the ’Down’ state again!

Backend design notes:
- The backend should never do implicit state changes, even if an illegal function call order is used by the frontend.
- To avoid deadlocks, the ’x_disable()’ functions should return 1 (Error) only on state errors (if the function call is not allowed in the current state). If another error occur, it should not be reported to the frontend − Instead do the best possible error recovery, execute the state change and report 0 (OK).

How to write/port a backend

Creating a library that exports the mandatory functions specified in the backend API should be enough to be compatible with AVRprog. All optional functions must be present but are allowed to do nothing returning 2 (not supported).

Always implement the newest available API. Backward compatibility is maintained within the same major number.

How to write programmer drivers for the UNIX backend

UNIX backend since V0.12 contain a driver ’skeleton’ that is fully functional integrated into the backend (This means that a driver library is compiled for it and the autodetect mechanism ask it whether a programmer is found). It is disabled by always returning "no programmer found", removing a single line activates the driver.

The ’skeleton’ driver lacks any communication code but contain function stubs with comments what to implement there. Together with the code of the existing drivers as reference it should be really easy to implement support for a given programmer into the ’skeleton’ driver.

Netlayer

General

The network layer ("netlayer") is a nearly transparent data link between the frontend and the backend.

The backend with the network layer on top looks like the bare backend again. Therefore no (in reality a little bit of work is neccessary) changes are required in the frontend.

The exception is the network address of the target machine − the frontend must provide a possibility to specify it.

The network layer splits the application in a client and a server part. Both parts communicate via the network.

The API version of the network components should match to the version used in the backend and frontend. The same compatibility rules as for the backend apply (newer API versions on the lower level components are allowed).

Implementation

To use the netlayer, the frontend must be linked against the netlayer client library instead of the backend library.

On the target machine, the netlayer server daemon must be running. This daemon is linked against the backend library.

If the netlayer construction is used and a local programmer should be accessed, the loopback device can be used. Therefore the own network address must be specified as the target machine (and a netlayer server daemon must be running on the local machine).

At the moment only the IP protocol (V4) is supported by the netlayer.

TCP is used on OSI Layer 4.

Network protocol V0.0 (OSI Layer 5)

General:

This protocol is used to transport commands, status and the programming data between server and client via a network connection.

This protocol uses packets of different size, but similar structure to transmit the data.

Note: This protocol is suited to transport API V0.3 calls only!

General packet layout:

Header:

Byte 0: Magic number (0xAA)

Byte 1: Size of the data filed (high byte)

Byte 2: Size of the data field (low byte)

Byte 3: Type of the data (command, status or programming data)

Data field:

Byte 4: ID / Byte 0 of the data

Byte 5: Byte 1 of the data

Byte 6: Byte 2 of the data

...

Byte n < 517: Last Byte of the data (**max. 512 data bytes** −> **max. 516 bytes packetsize**)

Type numbers (Byte 3):

0: Command (can contain additional data)

1: Status (contains only status byte)

2: Data (contains only data)

Commands, Status:

Command and status packets contain a ID number in Byte 4 (first data byte) to identify the command or the status.

Command packets can contain a data field. This data filed starts at byte 5 in this case.

Status packets never contain a data field!

Command ID’s:

0: Check my protocol version
1: Listen to reply
2: Get server version
3: Disconnect (This aborts the OSI Layer 4 and 5 connections)
4: Set device file
5: Programmer enable
6: Programmer disable
7: Backend info
8: Driver info
9: Request sense
10: Programming mode enable
11: Programming mode disable
12: Read device code
13: Chip erase
14: Read flash memory
15: Write flash memory
16: Read EEPROM memory
17: Write EEPROM memory
18: Write lock bits

Status ID’s:

0: OK / Accepted
1: Error / Rejected
2: Not supported
3: End of data
4: Abort

Communication:

In the following definition the term ’transfer’ means one complete operation (see below).

Transmission:

The term ’transmission’ means the complete unidirectional data traffic until one must wait for the other side (or no more data is left at the end of a transfer):

- A transmission can consist of several packets.
- A transmission can use one of the following transfer types:
  a) Command packet, Statuspacket
  b) Commandpacket, n Datapackets, Statuspacket
  c) Statuspacket
- **Each transmission must end with a status packet!!!**
- Type a) or b) transmissions always end with status 3 (End of data) on success, or status 4 (Abort) if something goes wrong.

Transfer:

A transfer have the following phases:

1) Command and data transmission (Type a) or b)) from client to server
2) Statustransmission (Type c)) from server to client
3) [Data transmission (Type a) or b)) from server to client if the client requested it (because a transmission cannot start with a data packet, it starts with command 1 (Listen to reply))]

Therefor each transfer consists of 3 packets minimum (2 for request, 1 for reply)

A transfer must be initiated by the client. The server is passive and do not send something without request.

Connection :

Before normal commands and data can be transferred, a connection at OSI Layer 4 and a session at OSI Layer 5 must be established.

This is only possible, if the Layer 5 protocol versions of server and client are compatible.

Only the client can establish a session. This works as follows:

- The client establish a Layer 4 connection
- The client sends command 1 (Check my protocol version) to the server. In the data field the client Layer 5 protocol version is transferred
- The client terminates the transmission with status 3 (End of data)
- The server is checking the supplied version and send status 0 (Accept) or status 1 (Reject)

If the server send status 0 (Accept) the session is established.

If the server send status 1 (Reject) the Layer 5 protocol versions are not compatible and the Layer 4 connection is closed by the server.

Command description (client view):

[C: Command packet D: Data packet S: Status packet]

Command and status bytes are in 'unsigned char' format.

0: Check my protocol version (Only valid as the first command)

Send:

C(0)

D(Major protocol version number <unsigned char>,

Minor protocol version number <unsigned char>)

S(3)

Expect:

S(x)

x=0: Protocol version accepted, Layer 5 Session established

x=1: Protocol version not supported

1: Listen to reply

This is the only command that is used by the server (it is forbidden for the client)

Expect:

D(Reply)

S(3)

2: Get server version

Send:

C(2)

S(3)

Expect:

S

[C(1)

D(Major version number <unsigned char>, Minor version number

<unsigned char>, Major interface version number <unsigned char>, Minor interface version number <unsigned char>)

S(3)]

3: Disconnect (This aborts the OSI Layer 4 and 5 connections)

Send:

C(3)

S(3)

Expect:

S

4: Set device file

Send:

C(4)

D (Type <unsigned char>, String <char[]>)

S(3)

Expect:

S

5: Programmer enable

Send:

C(5)

S(3)

Expect:

S

6: Programmer disable

Send:

C(6)

S(3)

Expect:

S

7: Backend info

Send:

C(7)

S(3)

Expect:

S

[C(1)

D (String <char[]>)

S(3)]

8: Driver info

Send:

C(8)

S(3)

Expect:

S

[C(1)

D (String <char[]>)

S(3)]

9: Request sense

Send:

C(9)

S(3)

Expect:

S

[C(1)

D(Sense code <unsigned char>)

S(3)]

10: Programming mode enable

Send:

C(10)

S(3)

Expect:

S

11: Programming mode disable

Send:

C(11)

S(3)

Expect:

S

12: Read device code

Send:

C(12)

S(3)

Expect:

S

[C(1)

D(Device code <char[3]>)

S(3)]

13: Chip erase

Send:

C(13)

S(3)

Expect:

S

14: Read flash memory

Send:

C(14)

D (Address (high byte first) <unsigned short int>,

Word count (high byte first) <unsigned short int>)

S(3)

Expect:

S

[C(1)

D (Data (high byte first) <unsigned short int[]>)

S(3)]

15: Write flash memory

Send:

C(15)

D (Address (high byte first) <unsigned short int>,

Word count (high byte first) <unsigned short int>,

Data (high byte first) <unsigned short int[]>)

S(3)

Expect:

S

16: Read EEPROM memory

Send:

C(16)

D (Address (high byte first) <unsigned short int>,

Byte count (high byte first) <unsigned short int>)

S(3)

Expect:

S

[C(1)

D (Data <unsigned char[]>)

S(3)]

17: Write EEPROM memory

Send:

C(17)

D (Address (high byte first) <unsigned short int>,

Byte count (high byte first) <unsigned short int>,

Data <unsigned char[]>)

S(3)

Expect:

S

18: Write lock bits

Send:

C(18)

D (Lockbyte <unsigned char>)

S(3)

Expect:

S

How to write/port a netlayer

The main job of the netlayer is to be invisible ;−) You have to hide all the network stuff inside the netlayer and present a backend API on the top side. On the bottom side you have to do valid calls for the same backend API to get the functionality from the backend.

It is like a tunnel for the API calls from the top side to the bottom side. The status must be transferred in the opposite direction.

Using our network protocol makes it compatible with other netlayer components.

Frontend

General

The frontend implements the user interface. This can be a simple text based command interpreter, an automated script or a nice graphical window with menus and buttons. The frontend only have to deal with the user − it need to know nothing about the used programming hardware, only to which machine it is connected to.

The frontend must implement and maintain the major AVRprog states and the resulting call order specified in the backend section. It must also deal with data file formats. The rest is pure presentation.

How to write/port a frontend

The minimum implementation must at least implement the mandatory backend API function calls to create the state machine and error recovery system. Additionally one or more of the optional commands must be implemented to be able to do something useful.

EOF