AR488 GPIB Controller

The AR488 GPIB controller is an Arduino-based controller for interfacing with IEEE488 GPIB devices. The code has been tested on Arduino UNO and NANO boards and provides a low cost alternative to other commercial interfaces.

To build an interface, at least one Arduino board will be required to act as the interface hardware. Connecting to an instrument will require a 16 core cable and a suitable IEEE488 connector. This can be salvaged from an old GPIB cable or purchased from electronics parts suppliers.

Details of construction and the mapping of Arduino pins to GPIB control signals and data bus are explained in the *Building an AR488 GPIB Interface* section.

The interface supports most of the standard Prologix commands with the exception of the ++lon command which is currently unsupported. The commands closely adhere to the Prologix syntax but there are some minor differences. In particular, due to issues with the longevity of the Arduino EEPROM memory, the ++savecfg command has been implemented differently. Details of all commands can be found in the *Command Reference* section.

Installation

Windows 7 and Windows 10 will automatically recognize the FTDI and CH340G chipsets used by Arduino boards and automatically install the drivers from Windows Update.

The official source for FTDI drivers is here:
https://www.ftdichip.com/FTDrivers.htm

The VCP driver provides a virtual COM port for communication while the D2XX (direct) driver allows direct access via a DLL interface.

The official CH340G driver source is here:
http://www.wch.cn/download/CH341SER_EXE.html

Linux Mint appears to automatically recognize both chipsets as well. Since Linux Mint is based on Ubuntu, it is expected that Ubuntu should also automatically recognize both chipsets.

Firmware Upgrades

The firmware is upgradeable via the Arduino IDE in the usual manner. Updates are available from https://github.com/Twilight-Logic/AR488

Client Software

The interface can be accessed via a number of software client programs.

- Terminal software (e.g. PuTTY)
- EZGPIB
- HE5FX GPIB Toolkit – GPIB Configurator (prologix.exe)

Terminal clients connect via a virtual COM port and should be set to 115200 baud, no partity, 8 data bits and 1 stop bit when connecting to the interface. On Linux, the port will be a TTY device such as */dev/ttyUSB0* or */dev/ttyACM0*.

Specific considerations apply when using an Arduino based interface with EZGPIB and the HE5FX toolkit. These are described in the *Working with EZGPIB and KE5FX* section.

Operating Modes

The interface can operate in both controller and device modes.

Controller mode

In this mode the interface can control and read data from various instruments including Digital multimeters (DMMs). oscilloscopes, signal generators and spectrum analyzers. When powered on, the controller and sends out an IFC (Interface Clear) to the GPIB bus to indicate that it is now the Controller-in-Charge (CIC).

All commands are preceded with the ++ sequence and terminated with a carriage return (CR), newline [a.k.a. linefeed] (LF) or both (CRLF). Commands are sent to or affect the currently addressed instrument which can be specified with the *++addr* command (see command help for more information).

By default, the controller is at GPIB address 0.

As with the Prologix interface, the controller has and auto mode that allows data to be read from the instrument without having to repeatedly issue *++read* commands. After *++auto 1* is issued, the controller will continue to perform reading of measurements automatically after the next *++read* command is used and using the parameters that were specified when issuing that command.

Device mode

The interface supports device mode allowing it to be used to send data to GPIB devices such as plotters via a serial USB connection. Device mode commands except ++lon are supported.

Transmission of data

Interrupting transmission of data

While reading of data for the GPIB bus is in progress, the interface will still respond to the ++ sequence that indicates a command. For example, under certain conditions when the instrument is addressed to talk (e.g. when *eos* is set to 3 [no terminator character] and the expected termination character is not received from the instrument, or read with eoi and the instrument is not configured to assert eoi, or auto mode is enabled), data transmission may continue indefinitely. The interface will still respond to the ++ sequence followed by a command (e.g. ++auto 0 or ++rst). Data transmission can be stopped and the configuration can then be adjusted.

Sending Data and Special characters

Carriage return (CR, hex 0D, decimal 13), newline [a.k.a linefeed] (LF, hex 0A, decimal 10), escape (hex 1B, decimal 27) and '+' (hex 2B, decimal 43) are special "control" characters. Carriage return and newline terminate command strings and direct instrument commands, whereas a sequence of two '+'s precedes a command token. Special care needs to be taken when sending binary data to an instrument, because in this case we do not want control characters to prompt some kind of action. Rather, they need to be treated as ordinary and added to the data that is to be transmitted.

When sending binary data, the above mentioned characters must be 'escaped' by preceding them with a single escape (hex 1B, decimal 27) byte. For example, consider sending the following binary data sequence:

54 45 1B 53 2B 0D 54 46

It would be necessary to escape the 3 control characters and send the following:

54 45 **1B** 1B 53 **1B** 2B **1B** 0D 54 46

Without these additional escape character bytes, the special control characters present in the sequence will be interpreted as actions and an incomplete or incorrect data sequence will be sent.

It is also necessary to prevent the interface from terminating the binary data sequence with a carriage return and newline (0D 0A) as this will confuse most instruments. The command *++eos 3* can be used to turn off termination characters. The command *++eos 0* will restore default operation. See the command help that follows for more details.

Receiving data

Binary data received from an instrument is transmitted over GPIB and then via serial over USB to the host computer PC unmodified. Since binary data from instruments is not usually terminated by CR or LF characters (as is usual with ASCII data), the EOI signal can be used to indicate the end of the data transmission. Detection of the EOI signal while reading data can be accomplished with the *++read eoi* command, while an optional character can be added as a delimiter with the

++eot_enable command (see the command help that follows). The instrument must be configured to send the EOI signal. For further information on enabling the sending of EOI see your instrument manual.

Command Reference

The controller implements the standard commands prefixed with two plus (++) character sequence to indicate that the following sequence is an interface command. Commands, with the exception of the *++savecfg* command, should be fully compatible with the Prologix GPIB-ETHERNET controller. However, the interface also implements a number of additional custom commands.

++addr

This is used to set or query the GPIB address. At present, only primary addresses are supported. In controller mode, the address refers to the GPIB address of the instrument that the operator desires to communicate with. The address of the controller is 0. In device mode, the address represents the address of the interface which is now acting as a device.

When issued without a parameter, the command will return the current GPIB address.

Modes: controller, device

Syntax: `++addr [1-29]`
where 1-29 is a decimal number representing the primary GPIB address of the device.

++auto

Configure the instrument to automatically send data back to the controller. When auto is enabled, the user does not have to issue ++read commands repeatedly. This command has additional options when compared with the Prologix version.

When set to zero, auto is disabled.

When set to 1, auto is designed to emulate the Prologix setting. The controller will automatically attempt to read a response from the instrument after any instument command or, in fact, any character sequence that is not a controller command beginning with '++', has been sent.

When set to 2, auto is set to "on-query" mode. The controller will automatically attempt to read the response from the instrument after a character sequence that is not a controller command beginning with '++' is sent to the instrument, but only if that sequence ends in a '?' character, i.e. it is a query command such as '*IDN?'.

When set to 3, auto is set to "continuous" mode. The controller will execute continuous read operations after the first ++read command is issued, returning a continuous stream of data from the instrument. The command can be terminated by turning off auto with ++auto 0 or performing a reset with ++rst.

Modes: controller

Syntax: `++auto [0|1|2|3]`
where 0 disables and 1 enables automatically sending data to the controller

Note:
Some instruments generate a "Query unterminated or "-420" error if they are addressed after sending an instrument command that does not generate a response. This simply means that the instrument has no information to send and this error may be ignored. Alternatively, auto can be turned off (`++auto 0`) and a `++read` command issued following the instrument command to read the instrument response.

++clr

This command sends a Selected Device Clear (SDC) to the currently addressed instrument. Details of how the instrument should respond may be found in the instrument manual.

Modes: controller

Syntax: `++clr`

++eoi

This command enables or disables the assertion of the EOI signal. When a data message is sent in binary format, the CR/LF terminators cannot be differentiated from the binary data bytes. In this circumstance, the EOI signal can be used as a message terminator. When ATN is not asserted and EOI is enabled, the EOI signal will be briefly asserted to indicate the last character sent in a multi-byte sequence. Some instruments require their command strings to be terminated with an EOI signal in order to properly detect the command.

The EOI line is also used in conjunction with ATN to initiate a parallel poll, however, this command has no bearing on that activity.

When issued without a parameter, the command will return the current configuration

Modes: controller, device

Syntax: `++eoi [0|1]`
where 0 disables and 1 enables asserting EOI to signal the last character sent

++eos

Specifies the GPIB termination character. When data from the host (e.g. a command sequence) is received over USB, all non-escaped LF, CR or Esc characters are removed and replaced by the GPIB termination character, which is appended to the data sent to the instrument. This command does not affect data being received *from* the instrument.

When issued without a parameter, the command will return the current configuration

Modes: controller, device

Syntax: `++eos [0|2|3|4]`
where 0=CR+LF, 1=CR, 2=LF, 3=none

++eot_enable

This command enables or disables the appending of a user specified character to the USB output from the interface to the host whenever EOI is detected while reading data from the GPIB port. The character to send is specified using the ++eot_char command.

When issued without a parameter, the command will return the current configuration.

Modes: controller, device

Syntax: `++eot_enable [0|1]`
where 0 disables and 1 enables sending the EOT character to the USB output

++eot_char

This command specifies the character to be appended to the USB output from the interface to the host whenever an EOI signal is detected while reading data from the GPIB bus. The character is a decimal ASCII character value that is less than 256.

When issued without a parameter, the command will return a decimal number corresponding to the ASCII character code of the current character.

Modes: controller, device

Syntax: `++eot_char [<char>]`
where <char> is a decimal number that is less than 256.

++help

Not currently supported

++ifc

Assert the GPIB IFC signal for 150 microseconds, making the AR488 the Controller-in-Charge on the GPIB bus.

Modes: controller

Syntax: `++ifc`

++llo

Disable front panel operation on the currently addressed instrument. In the original HPIB specification, sending the LLO signal to the GPIB bus would lock the LOCAL control on ALL instruments on the bus. In the Prologix specification, this command disables front panel operation of the addressed instrument only, in effect taking control of that instrument. The AR488 follows the Prologix specification, but adds a parameter to allow the simultaneous assertion of remote control over all instruments on the GPIB bus as per the HPIB specification.

This command requires the Remote Enable (REN) line to be asserted otherwise it will be ignored. In controller mode, the REN line is asserted by default unless its status is changed by the ++ren command.

When the ++llo command is issued without a parameter, the LLO signal is sent to the addressed instrument and this locks out the LOCAL key on the instrument control pannel. Because the instrument has been addressed and REN is already asserted, the command automatically takes remote control of the instrument. Most instruments will display REM on their control screen and all front panel controls will be disabled.

If the ++llo command is issued with the 'all' parameter, it will lockout the LOCAL key on every instrument on the GPIB bus. However REM will not be displayed until the instrument is individually addressed. It is still possible to regain local control by pressing LOCAL on the instrument control panel. However, once the instrument is addressed and sent another command, the controller will automatically lock in remote control to that instrument and all front panel controls will be disabled.

Modes: controller

Syntax: `++llo [all]`

++loc

Relinquish remote control and re-enable front panel operation of the currently addressed instrument. This command relinquishes remote control of the instrument by de-asserting REN and sending the GTL signal.

The Remote Enable (REN) line must be asserted and the instrument must already be under remote control otherwise the command has no effect.

In the original HPIB specification, this command would re-enable the LOCAL key on the instrument control panel on ALL instruments currently connected to the GPIB bus. In the Prologix specification, this command relinquishes remote control of the currently addressed instrument only. The AR488 follows the Prologix specification, but adds a parameter to allow the

simultaneous release of remote control over all instruments currently addressed as listeners on the GPIB bus as per the HPIB specification.

If the command is issued without a parameter, it will re-enable the LOCAL key on the control panel on the currently addressed instrument and relinquish remote control of the instrument. If issued with the 'all' parameter, it put all devices on the GPIB bus in local control state.

Modes: controller

Syntax: `++loc [all]`

++lon

Currently unsupported.

++mode

This command configures the AR488 to serve as a controller or a device.

In controller mode the AR488 acts as the Controller-in-Charge (CIC) on the GPIB bus, receiving commands terminated with CRLF over USB and sending them to the currently addressed instrument via the GPIB bus. The controller then passes the received data back over USB to the host.

In device mode, the AR488 can act as another device on the GPIB bus. In this mode, the AR488 can only be set as a GPIB talker or listener and expects to receive commands from another controller (CIC). All data received by the AR488 is passed to the host via USB without buffering. All data from the host via USB is buffered until the AR488 is addressed by the controller to talk. At this point the AR488 sends the buffered data to the controller. Since the memory on the controller is limited, the AR488 can buffer only 120 characters at a time.

If the command is issued without a parameter, the current mode is returned.

Modes: controller, device

Syntax: `++mode [0|1]`
where 0=device, 1=controller

++read

This command can be used to read data from the currently addressed instrument. Data is read until:

- the EOI signal is detected
- a specified character is read

- timeout expires

Timeout is set using the *read_tmo_ms* command and is the maximum permitted delay for a single character to be read. It is not related to the time taken to read all of the data. For details see the description of the *read_tmo_ms* command.

Modes: controller

Syntax: `++read [eoi|<char>]`
where <char> is a decimal number corresponding to the ASCII character to be used as a terminator and must be less than 256.

++read_tmo_ms

This specifies the timeout value, in milliseconds, that is used by the ++read (and ++spoll) commands to wait for a character to be transmitted while reading data from the GPIB bus. The timeout value may be set between 0 and 3000 milliseconds.

Modes: controller

Syntax: `++read_tmo_ms <time>`
where <time> is a decimal number between 0 and 3000 representing milliseconds

++rst

Perform a reset of the controller.

Modes: controller, device

Syntax: `++rst`

++savecfg

This command saves the current interface configuration. On the Prologix interface setting this to 1 would enable the saving of specific parameters whenever they are changed, including addr, auto, eoi, eos, eot_enable, eot_char, mode and read_tmo_ms.

Frequent updates wear out the EEPROM and the Arduino EEPROM has a nominal lifetime of 100,000 writes. In order to minimize writes and preserve the longevity of the EEPROM memory, the AR488 does not, at any time, write configuration parameters “on the fly” every time they are changed. Instead, issuing the *++savecfg* command will update the complete current configuration once. Only values that have changed since the last write will be written.

The configuration written to EEPROM will be automatically re-loaded on power-up. The configuration can be reset to default using the *++default* command and a new configuration can be saved using the *++savecfg* command.

The following parameters will be updated: addr, auto, eoi, eos, eot_enable, eot_char, mode, read_tmo_ms and verstr.

Modes: controller, device

Syntax: `++savecfg`

++spoll

Performs a serial poll. If no parameters are specified, the command will perform a serial poll of the currently addressed instrument. If a GPIB address is specified, then a serial poll of the instrument at the specified address is performed. The command returns a single 8-bit decimal number representing the status byte of the instrument.

The command can also be used to serial poll multiple instruments. Up to 15 addresses can be specified. If the *all* parameter is specified (or the command ++allspoll is used), then a serial poll of all 30 primary instrument addresses is performed.

When polling mulitple addresses, the *++spoll* command will return the address and status byte of the first instrument it encounters that has the RQS bit set in its status byte, indicating that it has requested service. The format of the response is `SRQ:addr,status`, for example: `SRQ:3,88` where 3 is the GPIB address of the instrument and 88 is the status byte. The response provides a means to poll a number of instruments and to identify which instrument raised the service request, all in one command. If SRQ was not asserted then no response will be returned.

When *++srqauto* is set to 1 (for details see the ++srqauto custom command), the interface will automatically conduct a serial poll of all devices on the GPIB bus whenever it detects that SRQ has been asserted and the details of the instrument that raised the request are automatically returned in the format above.

Modes: controller

Syntax: `++spoll [<PAD>|all|<PAD1> <PAD2> <PAD3>...]`
where <PAD> and <PADx> are primary GPIB address and *all* specifies that all instruments should be polled

++srq

This command returns the present status of the SRQ signal line. It returns 0 if SRQ is not asserted and 1 if SRQ is asserted.

Modes: controller

Syntax: `++srq`

++status

Set or display the status byte that will be sent in response to the serial poll command. When bit 6 of the status byte is set, the SRQ signal will be asserted indicating Request For Service (RQS). The table below shows the values assigned to individual bits as well as some example meanings that can be associated with them. Although the meaning of each bit will vary depending on the instrument and the manufacturer, bit 6 is always reserved as the RQS bit. Other bits can be assigned as required.

Bit7	*Bit6*	*Bit5*	*Bit4*	*Bit3*	*Bit2*	*Bit1*	*Bit0*
128	64	32	16	8	4	2	1
Always zero	**RQS**	Calibration enabled or error	Output available. Front/rear switch.	Remote control	Auto-zero	Autorange enabled. Front/rear switch.	Operational error.

The values of the bits to be set can be added together to arrive at the desired status byte value. For example, to assert SRQ, a value of 64 would be sufficient. However if we wanted to use bit 1 to indicate an operational error, then a value of 65 might be used in the event of the error occurring.

Modes: device

Syntax: `++status [byte]`
where byte is a decimal number between 0 and 255.

++trg

Sends a Group Execute Trigger to selected devices. Up to 15 addresses may be specified and must be separated by spaces. If no address is specified, then the command is sent to the currently addressed instrument. The instrument needs to be set to single trigger mode and remotely controlled by the GPIB controller. Using ++trg, the instrument can be manually triggered and the result read with ++read.

Modes: controller

Syntax: `++trg [pad1 … pad15]`

++ver

Display the controller firmware version. If the version string has been changed with ++setvstr, then ++ver will display the new version string. Issuing the command with the parameter ‘real’ will always display the original AR488 version string.

Modes: controller, device

Syntax: `++ver [real]`

Custom commands

This section details ++ commands that are not part of the original Prologix implementation but represent custom commands that have been added and are specific to the AR488 firmware.

++allspoll

Alias equivalent to *++spoll all*. See *++spoll* for further details.

++dcl

Send Device Clear (DCL) to all devices on the GPIB bus.

Modes: controller

Syntax: `++dcl`

++default

This command resets the AR488 to its default configuration.

When powered up, the interface will start with default settings in controller mode. However, if the configuration has been saved to EEPROM using the *savecfg* command, the controller will start with the previously saved settings. This command can be used to reset the controller back to its default configuration.

NOTE: unless the *++savecfg* command is used to overwrite the previously saved configuration, the previous configuration will be re-loaded from non-volatile memory the next time that the interface is powered up. Therefore, after using *++default*, configure the interface as required and use *++savecfg* to ensure that the new settings are applied on power up, or *++default* and then *++savecfg* without any further configuration to return the interface to its default state.

The interface is set to controller mode with the following parameters:

eot_enable	*false*
eot_char	*0*
eoi	*false*
controller mode	*true*
controller address	*0*
primary address	*1*
secondary address	*0*
status byte	*0*
read_tmo_ms	*1200*
version string	*default version string*

Modes: controller, device

Syntax: `++default`

++ppoll

When many devices are involved, Parallel Poll is faster than Serial Poll but is not widely used. With a Parallel Poll, the controller can query up to eight devices quite efficiently using the DIO lines. Since there are 8 DIO lines, up to 8 devices can be queried at once. In order to get an unambiguous response, each device should ideally assign to a separate data line. Devices assigned to the same line are simply OR'ed. Devices respond to the parallel poll by asserting the DIO line they have been assigned.

Response to a Parallel Poll is a data byte corresponding to the status of the DIO lines when the Parallel Poll request is raised. The state of each individual bit of the 8-bit byte corresponds to the state of each individual DIO line. In this way it is possible to determine which instrument raised the request.

Because a single bit can only be 0 or 1, the response to a parallel poll is binary, simply indicating whether or not an instrument has raised the request. In order to get further status information, a Serial Poll needs to be conducted on the instrument in question.

Modes: controller

Syntax: `++ppoll`

++setvstr

Set the version string that the controller responds with on boot-up and in response to the *++ver* command. This may be helpful where software on the computer is expecting a specific string from a known controller, for example 'GPIB-USB'.

The *++ver* command can be used to confirm that the string has been set correctly.

Modes: controller, device

Syntax: `++verstr [string]`
where [string] is the new version string

++srqauto

When conducting a serial poll using a Prologix controller, the procedure requires that the status of the SRQ signal be checked with the *++srq* command. If the response is a 1, indicating that SRQ is asserted, then an *++spoll* command can be issued to determine the status byte of the currently addressed instrument or optionally an instrument at a specific GPIB address.

When polling multiple devices, the AR488 will provide a custom response that includes the address and status byte of the first instrument encountered that has the RQS bit set. Usually, the *++spoll* command has to be issued manually to obtain this information.

When *++srqauto* is set to 1, the interface will automatically detect when the SRQ signal has been asserted by an instrument. The interface will then automatically conduct a serial poll and return the address and status byte of the first instrument encountered that has the RQS bit set in its status byte. If multiple instruments have asserted SRQ, then another serial poll is conducted to determine the next instrument that has requested service. The process continues until all instruments that have requested service have had their status byte read and the SRQ signal has been cleared.

Modes: controller

Syntax: `++srqauto [0|1]`
where 0=disabled, 1=enabled

++verbose

Toggle verbose mode ON and OFF

Modes: controller, device

Syntax: `++verbose`

Building an AR488 GPIB Interface

Construction of an Arduino GPIB interface is relatively straightforward and requires a single Arduino UNO or NANO, a length of cable that is at minimum 16-way and preferably screened, and an IEEE488 connector. An old GPIB cable could be re-purposed by removing one end, or an old parallel printer cable could be used, in which case a separate 24-way IEEE488 connector will need to be purchased.

New GPIB/IEEE488 cables are expensive. Cheaper cables can be found from various sellers on eBay. Connectors can be found by searching for 'Centronics 24' rather than 'IEEE488' or 'GPIB'. In the UK, RS Components sell these as part number 239-1207, for £2.86. They can also be found on eBay. Old parallel printer cables can still be found on charity/thrift shops or on market stalls.

The following connections are required between the Arduino and the IEEE488 connector:

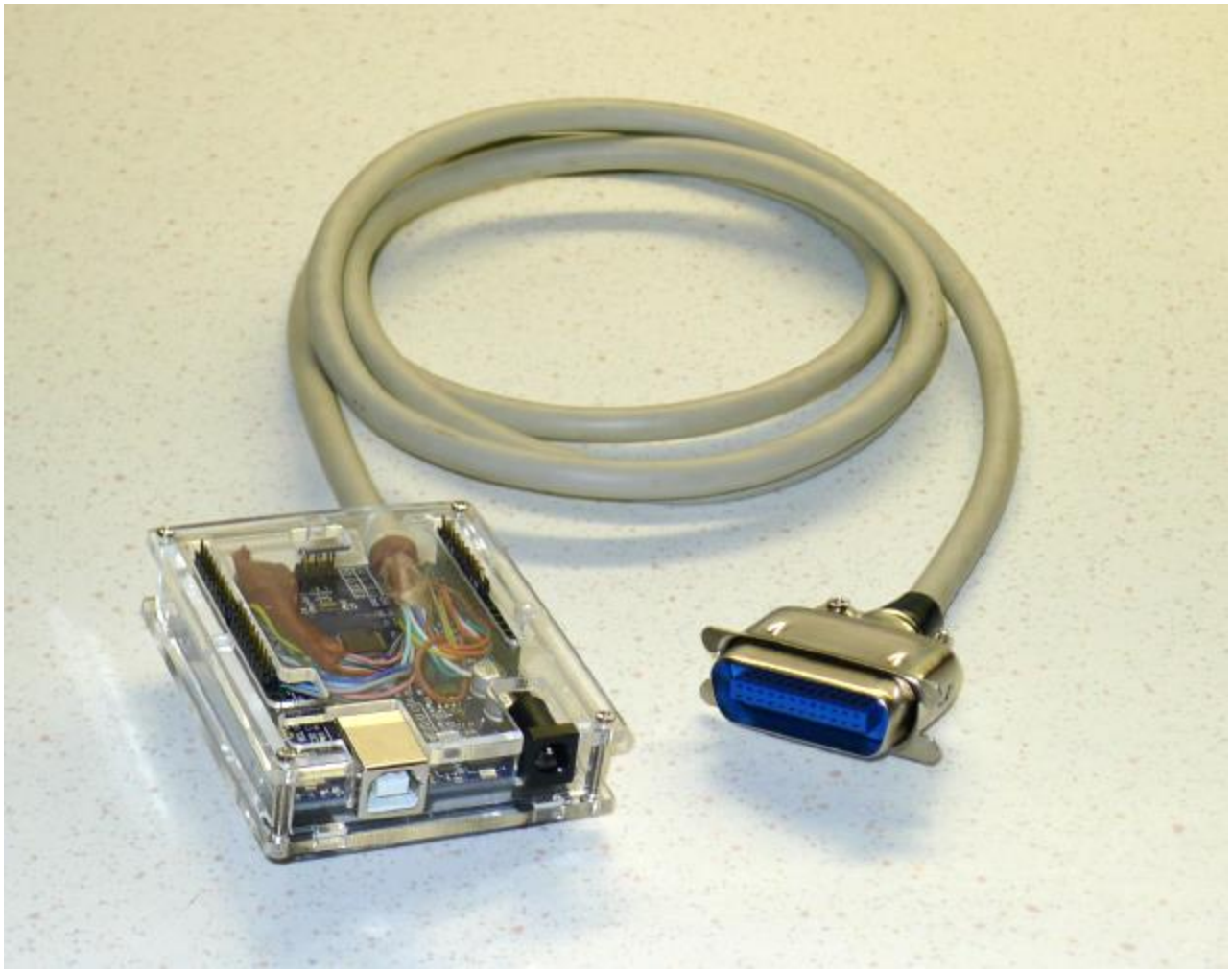
Arduino:	*GPIB connector:*	*Function:*
D2	10	SRQ
D3	17	REN
D7	11	ATN
D8	9	IFC
D9	8	NDAC
D10	7	NRFD
D11	6	DAV
D12	5	EOI
A0	1	DIO1
A1	2	DIO2
A2	3	DIO3
A3	4	DIO4
A4	13	DIO5
A5	14	DIO6
D4	15	DIO7
D5	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

Ideally, the ground pins 18, 19, 20, 21, 22, 23 should be connected to the ground side of a twisted pair with the DAV, NRFD, NDAC, IFC, SRQ and ATN control wires, however linking them together and connecting them to GND on the Arduino side should suffice.

Further GPIB pinout information - Link 1

Further GPIB pinout information - Link 2

Once the cable has been completed, the sketch should then be downloaded to the Arduino and the interface should be ready to test.



An example of a completed Arduino GPIB adapter

The following section details further hardware tweaks that may be required to make the board work correctly with specific GPIB software.

Working with EZGPIB and KE5FX

FTDI serial vs CH340G serial

EZGPIB is an IDE programming environment that can be used to work with GPIB devices. KE5FX provides testing tools that can be used with various instruments that support GPIB. Both programs support the Prologix interface and when communicating with it, both programs assert RTS and expect a CTS response to confirm that the interface is ready to accept data.

The CH340G chipset present in many Arduino compatible boards does not respond with the CTS signal. There appear to be two possible workarounds, one of which requires very good soldering skills. The RTS and CTS signals are exposed via pins 14 and 9 respectively on the CH340G chip. While pin 9 connects to an easily accessible pad for soldering, pin 14 is not connected to anywhere and because it is very small, attaching a wire to it is rather tricky. For this reason, workaround 2 is easier to implement. Disclaimer: please proceed only if you are confident in your soldering skills. I take no responsibility for damaged Arduino boards so if in doubt, ask a qualified or skilled person for assistance.

Workaround 1

The workaround requires that pin 14 be connected to pin 9 on the CH340G chip. When RTS is asserted by the host over USB, the signal is passed to the RTS output on pin 14 of the CH340G. This signal would ordinarily be passed to a serial hardware device which would respond by sending a response to the CTS input on pin 9 of the the CH340G to indicate that it is ready to send. The workaround passes this signal back to the CTS input via the link so that a CTS response will always be echoed back to the host over USB. While this does not provide proper RTS/CTS handshaking, it does allow the interface to respond with a CTS signal and, in turn, the host to be able to accept responses to the commands sent to the interface, even when RTS/CTS handshaking is used.

Workaround 2

Pin 9 of the CH340G needs to be connected to GND. This will keep CTS signal asserted on the Arduino at all times, so again, proper handshaking is not provided. Simply solder a short wire to the pad and connect to a convenient ground point.

A big thanks goes to Hartmut Päsler, who is currently looking after the EZGPIB program, for informing me that the CH340G exposes the RTS/CTS signals via pins and that it might be possible to make use of these pins to device a solution.

Where Arduino boards are recognized as FTDI serial devices, the functionality is embedded within the ATMEGA MEGA 16U2 chip. This chip does not expose the RTS/CTS signals so this modification is not possible nor is it required to work with the KE5FX toolkit. An Arduino board running with the 16U2 chip running AR488 will work fine with the KE5FX GPIB toolkit, but for some reason, it is not recognized by the EZGPIB program.

EZGPIB and the Arduino bootloader

On older Arduino boards it was necessary to press the reset button to program the board. This causes the board to reset and the bootloader to run. The bootloader will expect a particular sequence of bytes within a timeout period and it will then expect a new compiled sketch to be uploaded into memory. On completion of the upload, program control is passed to the newly uploaded code. The timing of the upload is rather tricky and if the timeout period expires or the upload is started too soon, then it will fail and the board will start with the current program code.

Current versions of the board allow code to be uploaded via USB without having to use the reset button. This is accomplished by triggering a reset of the board each time a serial connection is opened. The bootloader is then re-loaded and if the required sequence of bytes is received, and an upload of code proceeds automatically. When this is finished, program execution passes to the new code as before.

The problem with this is that the bootloader is loaded every time that the serial port is opened. This causes a delay of about 1 second before the compiled user program is actually run and the interface is initialised. EZGPIB (and possibly other programs) that do not re-try the connection attempt after waiting a second or so, fails to establish a connection to the interface. Closing the program and immediately trying again usually results in a successful connection.

The solution is to eliminate the delay caused by the board re-starting and the bootloader being re-loaded into memory. This can be done quite easily by placing a 10µF capacitor between the RST and GND pins on the Arduino. This causes the reset pulse, which is generated by activating the serial DTR signal, to be drained to ground without affecting the RESET input on the AtMega328P processor. Since it's a capacitor, there is no direct DC coupling between RESET and GND. When the serial port is now opened, the interface will just respond without the delay caused by re-booting. Assuming the sequence "GPIB-USB" exists in the response to the ++ver command, EZGPIB will now recognize it first time.

The drawback of this approach is that placing a capacitor permanently in this position will prevent the Arduino IDE from being able to program the board. The reset button now has to be used or a switch added to provide an on to run, off to program facility.

Multiple Arduinos on the bus and problems with instruments

The AR488 can be used in both controller mode and device mode. Only ONE controller can be active at any one time and when there is just one Arduino controller on the bus controlling instruments, this does not present a problem. The controller is always powered up and provided that it is operating within its current handling limits, this will work fine.

However, it is possible to have a scenario where one AR488 as operating as a controller and another as a device simultaneously on the bus along with other instruments. However, in this case, problems may sometimes start to arise. Instruments might sometimes fail to respond to the ++read or other commands or to direct instrument commands or both.

The reason for this is because Arduino control pins do not present with a high impedance when the board is powered down. In a powered down state, voltages present on the various signal and

data lines are passed via protection diodes internal to the ATMega processor, to the +VCC rail on the powered down interface. This then causes all pins on the unpowered Arduino to go HIGH. Furthermore, enough power may be present on the +VCC rail to at least partially power the processor, which, even if it does manage to operate, is likely to do so in an unstable manner causing further problems. The result is interference with the proper functioning of other equipment on the GPIB bus. This phenomenon is not specific to Arduino microcontrollers. It affects various other devices too. Further information can be found here:

https://www.eevblog.com/forum/blog/eevblog-831-power-a-micro-with-no-power-pin!/

Consequently, unpowered Arduino devices will adversely affect other devices on the GPIB bus. It is therefore essential to either keep Arduino devices powered on, or physically disconnect them from the bus. This is NOT an issue when there is just ONE Arduino-based GPIB controller remotely controlling instruments on a bus. Therefore, other than when an Arduino is operating as a controller, it is not recommended to leave unpowered Arduino's connected to the bus.