AR488 GPIB Controller
User Manual

Version:
0.48.08

Dated:
31th January 2020

Twilight Logic

01001010 01000011 01001000

The 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, Nano, Mega 2560 and Micro 32U4 boards and provides a low cost alternative to other commercial interfaces.

To build an interface, at least one of the aforementioned Arduino boards 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. Alternatively, a PCB board can be designed to incorporate a directly mounted IEEE488 connector.

The interface firmware can optionally support the SN75160 and SN75161 GPIB transceiver integrated circuits. Details of construction and the mapping of Arduino pins to GPIB control signals and the data bus are explained in the *Building an AR488 GPIB Interface* section.

The interface firmware supports standard Prologix commands (with the exception of ++help) and adheres closely 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, however an AVR programmer can also be used to upload the firmware to the Arduino microcontroller. 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 (http://www.ulrich-bangert.de/html/downloads.html)
- KE5FX GPIB Configurator (http://www.ke5fx.com/)
- Luke Mester's HP3478 Control (https://mesterhome.com/gpibsw/hp3478a/)
- Python scripts
- Anything else that can use the Prologix syntax!

Terminal clients connect via a virtual COM port and should be set to 115200 baud, no parity, 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 KE5FX 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 an 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. All device mode commands 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.

Listen-only and talk-only (lon and ton) modes

In device mode, the interface supports both “listen-only” and “talk-only” modes (for more details see the *++lon* and *++ton* commands. These modes are not addressed modes and do not require a GPIB address to be set. Therefore if any GPIB address is already set, it is simply ignored. Moreover, when in either of these modes, devices are not controlled by the CIC. Data characters are sent using standard GPIB handshaking , but GPIB commands are ignored. The bus acts as a simple one to many transmission medium. In *lon* mode, the device will receive any data placed on the bus by any talker, including any other addressed device or controller. Since only ONE talker can exist on the bus at a time, there can only be one device in "talk-only" mode on the bus, however multiple "listen-only" devices can be present and all will receive the data sent by the talker.

Wireless communication

The AR488 interface can communicate using a Bluetooth module (HC05 or HC06). The firmware sketch supports auto-configuration of the Bluetooth HC05 module, the details of which can be found in the Configuration section and the AR Bluetooth Support supplement. Automatic configuration is not possible with a HC06 module so although this can be used to provide Bluetooth communication, it has to be configured manually.

Configuration

Configuration of the AR488 is achieved be editing the *AR488_Config.h* file. This is a C++ style header file containing various definition statements, also known as 'define macros' , starting with keyword *'#define'*, that can be used to configure the firmware. The *AR488_config.h* file must be included in the main AR488 sketch as well as any other module header file (e.g. *AR488_Layouts.cpp* and *AR488_Layouts.h*) with an include statement:

```
#include "AR488_Config.h
```

A number of these definition statements are contained within an *#ifdef* .. *#endif* construct, some of which may contain additional *#else* or *#elif* elements. The presence of these constructs is necessary and they should not be changed or removed. Only the definitions within them should be changed as required. Nothing should need to be changed in any other file.

Firmware version

This is in the format:

```
#define FWVER "AR488 GPIB controller, ver. 0.48.08, 27/01/2020"
```

This entry should not exceed 47 characters and should not be changed.

Board Selection and serial port configuration

The AR488 supports a number of Arduino AVR boards and also a custom GPIO pin layout which can be defined by the user in the *Custom Board Layout* section. If a custom GPIO pin layout is to be used, then following entry must have the comment characters (preceding '//') removed:

```
//#define AR488_CUSTOM
```

Otherwise, the comment characters should remain in place which has the effect of disabling the definition by designating it as a comment. The compiler ignores comment statements. Following this is an *#ifdef* statement containing several sections preceded by an *#elif* keyword. Each of these is followed by a token that corresponds to known Arduino definitions for microprocessor types. The structure looks like this:

```
/*
 * Configure the appropriate board/layout section
 * below as required
 */
#ifdef AR488_CUSTOM
…
#elif __AVR_Atmega328P__
…
#elif __AVR_Atmega32U4__
...
#elif __AVR_Atmega2560__
…
#endif  // Board/layout selection
```

When the custom layout is selected, all other layouts are ignored. If the custom layout is not selected, then the section corresponding to the automatically detected Arduino microprocessor will apply. Each section contains a definition referencing one or more pre-defined board layouts as well as serial port definitions corresponding to the features of specific boards. For example here are the definitions for boards based on the 328p microprocessor which are found within the *__AVR_Atmega328P__* section of the *#ifdef* statement:

```
/* Board/layout selection */
 #define AR488_UNO
 //#define AR488_NANO
 /*** Serial ports ***/
 //Select HardwareSerial or SoftwareSerial (default = HardwareSerial) ***/
 // The UNO/NANO default hardware port is 'Serial'
 // (Comment out #define AR_HW_SERIAL if using SoftwareSerial)
 #define AR_HW_SERIAL
 #ifdef AR_HW_SERIAL
   #define AR_SERIAL_PORT Serial
   #define USE_SERIALEVENT
 #else
   // Select software serial port
   #define AR_SW_SERIAL
 #endif
```

The section contains definitions for two boards, namely the Uno and the Nano. Only ONE of these should be selected by removing the preceding comment characters:

```
#define AR488_UNO
//#define AR488_NANO
```

The default entry is *AR488_UNO*, which selects the pre-defined template for the Arduino UNO board in AR488_Hardware.h. Selecting AR488_NANO will select the pre-defined template for the Nano board. In order to compile the sketch for the selected board, in addition to selecting the template in Config.h, the correct board must be selected in the Board Manager within the Arduino IDE (see Tools | Board:).

Following this are definitions for the serial port:

```
 #define AR_HW_SERIAL
 #ifdef AR_HW_SERIAL
   #define AR_SERIAL_PORT Serial
   #define USE_SERIALEVENT
 #else
   // Select software serial port
   #define AR_SW_SERIAL
 #endif
```

By default, the most commonly used serial port for a particular board will be enabled. In the example above, the hardware port named *Serial* is selected. To switch between the default hardware port and a *SoftwareSerial* port, it is necessary only to comment out *#define AR_HW_SERIAL* by preceding the line with '//'.

The section for the 32u4 (Micro/Leonardo) is similar, except by default *AR_CDC_SERIAL* is enabled and switching is between the USB CDC port and the hardware port *Serial1*.

The Mega 2560 has 4 hardware serial ports so either '*Serial*', '*Serial1*', '*Serial2*' or '*Serial3*' must be selected. Most likely the default port named *Serial* will be used although other options are

possible if required. However, please note that the default GPIO pin layout for the Mega 2560 board (AR488_MEGA2560_D) uses the pins assigned to *Serial2* for other purposes, so this cannot be used as a serial port with that particular layout definition. However, it can be used with the *E1* and *E2* definitions.

For any board, adding the line *#define AR_SW_SERIAL* and commenting out the *AR_HW_SERIAL* and/or *AR_CDC_SERIAL* definitions will invoke the *SoftwareSerial* library. When a SoftwareSerial port is required, then the GPIO pins as well as the baud rate to be used will need to be configured in the following *Software Serial Support* section.

Where a board has more than one hardware or CDC serial port available, it will be necessary to correctly specify the Arduino name of the serial port to be used. For the Uno and Nano this will always be '*Serial'* because those boards have only one UART and therefore only one hardware serial port. On the other hand, some boards have more than one serial port available. For example, on the Pro Micro , enabling the hardware port rather than the CDC port will automatically select *Serial1* instead of *Serial*. For the Mega 2560 there are four choices and in addition to the correct port name, the corresponding serial interrupt must be chosen. It is therefore necessary to uncomment both of a pair of lines for the selected port, e.g:

```
#define AR_SERIAL_PORT Serial
#define USE_SERIALEVENT
```

Each port definition will have a corresponding *USE_SERIALEVENT* definition which must be enabled along with the port definition. The remaining choices must then be commented out by preceding them with '//'.

It should be noted that for any board, only ONE serial port can be used and therefore only one port should be enabled.

It is important to make sure that the correct board is selected in the Arduino IDE Boards Manager (*Tools => Board*) otherwise the sketch will not compile correctly.

Software Serial port configuration

The *SoftwareSerial* library can be used with any board provided that at least two pins are available. One of these must be a PWM enabled GPIO pin which is required to emulate the transmit (Tx) output for the serial two wire connection. The receive (Rx) pin can be assigned any available GPIO pin.

Enabling *SoftwareSerial* can be done by removing the comment characters ('//') preceding the *#define AR_SW_SERIAL* entry in the relevant board selection section. In addition, the pins to be used as well as the board rate will need to be configured in the *Software Serial Support* section as follows:

```
#ifdef AR_SW_SERIAL
  #define AR_SW_SERIAL_RX 53
  #define AR_SW_SERIAL_TX 51
  #define AR_SERIAL_BAUD 57600
#else
  #define AR_SERIAL_BAUD 115200
#endif
```

The appropriate GPIO pin numbers should be specified after the *#define AR_SW_SERIAL_RX* and the *#define AR_SW_SERIAL_TX* statements within the *#ifdef AR_SW_SERIAL* clause. The baud rate should be specified here as well after *#define AR_SERIAL_BAUD*. Please note that, when using *SoftwareSerial*, the maximum baud rate that can be achieved reliably is 57600 baud.

The hardware/CDC serial port baud rate is specified after the *#else* statement. The default hardware baud rate is 115200, but any valid baud rate can be specified.

Please note also, that when using USB CDC ports, the Arduino board will NOT be reset when a serial connection is made over USB as is the case with Uno and Nano boards. The reset button must be pressed in order to reset the board. The Arduino IDE seems to take care of programming the board automatically but when using the Arduino IDE on Linux, the *modemmanager* service will need to be disabled as it interferes with serial ports and disrupts the normal operation of the programming process, causing boards to end up in a state where they can no longer be programmed over USB. Boards that have been disabled in this way can be recovered by uploading a bootloader to them using an AVR programmer. Linux Mint (and probably Ubuntu) will have this service enabled and running in the background by default. This service is not required but may be useful in the event that a serial modem is connected to the PC.

Serial Interrupt Handling

All AVR boards support *SerialEvent*, which is an interrupt handling function that is triggered when a character is received into the serial input buffer. This interrupt function can be used to immediately accept and process the character while the interface is performing other tasks such as sending data over the GPIB bus. This avoids a delay which would otherwise be inevitable while waiting for the the exiting process to complete and return to the next iteration of the *void loop()* function. Non-Arduino boards may not support this interrupt event and neither does *SoftwareSerial*, so for this reason, the *SerialEvent* handler is used only when compiling for AVR boards that support this feature, for example the Uno, Nano and Mega2560.

When working with programs and scripts (e.g. Python) and where the *SerialEvent* handler is not supported, e.g. 32u4 and non-AVR boards, then in order to compensate for the additional delays that may result, a short delay may need to be added after any script statement that sends a command to the interface or data to the instrument to allow time for the instrument to respond and the processing of the response to complete. The amount of delay will be a matter of trial and error as it will depend on factors such as the interface hardware being used as well as the speed of the instrument being addressed.

In most cases enabling *SerialEvent* will be handled within the board layout definitions. Where *USE_SERIALEVENT* is defined, then the serial event handler will be used, otherwise the capture of incoming serial data will be handled within the *void loop()* function.

Detection of SRQ and ATN pin states

Arduino AVR boards support interrupts to detect a change in pin states and this has been implemented for the UNO, NANO and MEGA boards to improve response. When a supported

board template has been selected, (see the *Board Selection and Serial Port Configuration* section), then *USE_INTERRUPTS* will be activated by default.

Other boards may not support interrupts and interrupts cannot be used with the custom GPIO pin layout, so instead, pin states are detected during each interation of the *void loop()* function. When *AR488_CUSTOM* is enabled, or a non-AVR board is used, or an unsupported board is selected as the compilation target, then the *USE_PINHOOKS* option will be automatically selected instead.

The section looks as follows:

```
#ifdef __AVR__
  // For supported boards use interrupt handlers
  #if defined (AR488_UNO) || defined (AR488_NANO) || defined (AR488_MEGA2560) ||
defined (AR488_MEGA32U4)
    #define USE_INTERRUPTS
  #else
    // For other boards use in-loop checking
    #define USE_PINHOOKS
  #endif
#else
  #define USE_PINHOOKS
#endif
```

For manual configuration, only ONE of the following options should be enabled:

```
#define USE_PINHOOKS

#define USE_INTERRUPTS
```

The other entry should be preceded by '//' to indicate that it has been commented out. The default setting for supported AVR boards is *#define USE_INTERRUPTS*. Interrupts are used by default because they usually respond faster than in-loop checking.

Macro support

Macros in this context are short sequences of commands that can be used to accomplish a particular task. Controlling an instrument usually requires sequences of commands to be sent to the device to configure it, or to perform a particular task. Sometimes such sequences are performed frequently or repetitively. In those circumstances, it may be more efficient to pre-program the required sequence and then execute it when required using a single command.

The AR488 supports a macro feature which allows user programmed command sequences to be run when the interface starts up, as well as up to 9 user defined command sequences to be executed at runtime.

Macros must be programmed before the sketch is compiled and uploaded to the interface. Macros can be added to the designated *AR488 MACROS SECTION* in the *AR488_Config.h* file. Both interface ++ commands and direct instrument commands can be included in macros. Programming specific instruments is beyond the scope of this manual as commands will be specific to each instrument or implemented according to the manufacturers choice of programming language or protocol. However, in general, in order to create macros, a few simple rules will need to be followed.

Firstly, macros need to be enabled. In the AR488_Config.h file there are two definitions under the heading 'Enable Macros':

```
#define USE_MACROS    // Enable the macro feature
#define RUN_STARTUP   // Run MACRO_0 (the startup macro)
```

The *#define USE_MACROS* construct enables or disables the macro feature. When this line is commented out by preceding it with '//' then macros are disabled. Removing the preceding '//' will enable the macro feature.

The *#define RUN_STARTUP* statement controls whether the start-up macro will run when the interface is powered up or re-started. The start-up macro is designated MACRO_0 and if *#define RUN_STARTUP* is enabled, this macro will run when the interface is powered on or reset.

When *#define USE_MACROS* is disabled, then the start-up macro will not be activated when the interface is powered up or reset and none of the user macros (1-9) will be available at runtime.

When enabled, MACRO_0 will run when the interface is powered up or reset but only if *#define RUN_STARTUP* is also enabled. The user macros (1-9) will always be available and can be executed by the user at runtime by using the *++macro* command. For more information please see the *++macro* command heading in the Command Reference.

The start-up macro can be used in addition to the interface settings that can be saved using the *++savecfg* command, to not only to set up the interface, but also to initialise and configure the instrument for a specific function. In this way, instrument commands that select function, range and other control features can be sent automatically as the interface starts up.

Unless steps have been taken to disable the automatic reset that occurs when a USB serial connection is opened to the interface, the start-up macro will run every time that a serial connection is initiated to the interface. On the other hand, disabling reset prevents the Arduino from being programmed via USB, so is not advised unless the intention is to program the Arduino using a suitable AVR programmer.

In the AR488_Config.h file, sketch, below the help information there is a section that starts:

```
/*******************************/
/***** AR488 MACROS SECTION *****/
/***** vvvvvvvvvvvvvvvvvvv *****/
#ifdef USE_MACROS
```

Macros are defined here. The first macro is the startup macro, an example of which might be defined as follows:

```
#define MACRO_0 "\
++addr 9\n\
++auto 2\n\
*RST\n\
:func 'volt:ac'
"
/* End of MACRO_0 (Startup macro)*/
```

Note that the macro code itself, is shown in **bold,** and has been inserted immediately after the *#define MACRO_0* line and before the ending comment:

```
#define MACRO_0 "\
macro
"
/*<-End of startup macro*/.
```

All macro commands comprising the macro must be placed after the '\' on the first line and before the final quote on the line before the ending comment. Nothing outside of these lines, including the quote marks and the '\' and after the macro name should be modified. The final quote mark can be appended to the last command in the sequence if preferred. It is shown here on a separate line for clarity. Everything between the two quote marks is a string of characters and must be delimited. The '\' character indicate to the pre-processor that the string continues on the next line. Each command ends with '\n' which is the newline terminator and serves to delimit each command. The actual sequence shown above is therefore comprised of 4 commands, each command ending with '\n' and then a '\' to indicate that the next command is to follow on the next line. Try to avoid leaving or including any unnecessary spaces.

Each of these commands is either a standard AR488 interface command found in the command reference, or an instrument specific command. All AR488 interface Prologix style commands begin with `++` so the first two commands set the GPIB address to *7* and *auto* to *1*. The next two commands are direct instrument commands using the SCPI protocol, the first of which resets the instrument and the second selects the instrument AC voltage function.

As shown, each command must be terminated with a '\n' (newline) or '\r' (carriage return) delimiter character.

User defined macros that can be run using the *++macro* command follow next, and have a similar format, e.g:

```
#define MACRO_2 "\

"
/*<-End of macro 2*/
```

Once again, the required command sequence must be placed between the two quotes and after the first '/' and be terminated with a '\n' or '\r' delimiter. Each line must be wrapped with '\'.

There is a slightly shorter method of defining a macro by placing all commands on a single line. For example this:

```
#define MACRO_1 "++addr 7\n++auto 1\n*RST\n:func 'volt:ac'"
```

Is exactly the same as this:

```
#define MACRO_1 "\
++addr 7\n\
++auto 1\n\
*RST\n\
:func 'volt:ac'\
"
```

The first definition is more condensed and requires no line wrap characters, but it is perhaps easier to see what is going on in the latter example. Either will function just the same and take up the same amount of memory.

The macro definition area provided in the sketch ends with:

```
#endif
/***** ^^^^^^^^^^^^^^^^^^^^ *****/
/***** AR488 MACROS SECTION *****/
/*******************************/
```

Anything outside of this section does not relate to macros.

Provided that the commands have been specified correctly and the syntax is correct, the sketch should compile and can be uploaded to the Arduino. The start-up macro will run as soon as the upload is completed so the instrument should respond immediately. Please be aware that, unless serial reset has been disabled, it will run again when a USB serial connection is made to the interface. The instrument will probably respond and reconfigure itself again.

Please note that, although AR488 interface ++ commands are verified by the interface, and will respond accordingly, there is no sanity checking by the interface of any direct instrument commands. These command sequences are sent directly to the instrument, which should respond as though the command sequence were typed directly into the terminal or sent from a suitable instrument control program. Please consult the instrument user manual for information about the behaviour expected in response to instrument commands.

Macro sequences can include any number delimiter separated of commands, but any individual command sequence should not exceed 126 characters. This may be particularly relevant to SCPI commands which can be composed of multiple instructions separated by colons.

SN7516x GPIB transceiver support

Support for the SN75160 and SN75161 GPIB transceiver integrated circuits can be enabled by uncommenting the following line:

```
//#define SN7516X
```

The pins used to control the ICs are defined in the section that follows:

```
#ifdef SN7516X
  #define SN7516X_TE 6
//  #define SN75161_DC 13
#endif
```

Specify the pin to be used for the SN7516X_TE signal. The above example shows pin 6 being used and this is connected to the talk-enable (TE) pin on **both** ICs. The SN75161 handles the GPIB control signals and in addition to the TE pin, also has a direction-control (DC) pin. This is used to determine controller or device mode operation. A GPIO pin can be assigned to drive this pin, in which case the SN75151_DC definition shown above should be uncommented and an appropriate GPIO pin number assigned.

Alternatively, since the REN signal is asserted in controller mode and un-asserted in device mode, this signal can be used to drive the DC pin of the SN75161. In this case, the SN75161_DC definition should remain commented out and the GPIO pin assigned to the REN signal should be connected to both DC and REN on the SN75161 IC. There is one small caveat when using this configuration. The custom *++REN* command, which is used to turn the REN line on and off, cannot be used and will just return:

```
Unavailable.
```

If a separate GPIO pin is used to control DC then the ++REN command will return the status of REN as usual. (See *++REN* in the *Custom Comands* section of the *Command Reference*).

Bluetooth HC05 module Options

This section is used to configure Bluetooth HC05 module options and looks like the below:

```
//#define AR_BT_EN 12             // Bluetooth enable and control pin
#ifdef AR_BT_EN
  #define AR_BT_BAUD 115200     // Bluetooth module preferred baud rate
  #define AR_BT_NAME "AR488-BT" // Bluetooth device name
  #define AR_BT_CODE "488488"   // Bluetooth pairing code
#endif
```

To enable Bluetooth HC05 module auto-configuration, the first line needs to have the preceding comment characters (*//*) removed and a GPIO pin assigned. It is then necessary to set the configuration parameters, including baud rate, the name that the device will be identified with and the pairing code. The *AR_BT_BAUD* parameter must not have double quotes around it.

By default, the name is *AR488-BT* and the pairing code is *488488*. The HC05 module only needs connecting to the RX/TX pins of a serial port and it will be automatically configured with these parameters on interface start-up.

This feature cannot work with the HC06 module as it does not have management mode or an enable pin implemented. Full details of Bluetooth configuration and wiring are included in the separate *AR488 Bluetooth Support* supplement.

Debug options

The AR488 can send certain debug messages to a serial port which can be helpful when trying to diagnose a problem. These should not be required or enabled for normal running of the interface, but if required for debugging, one or more of the following can be enabled by removing the preceding / / comment characters:

```
//#define DEBUG1  // getCmd
//#define DEBUG2  // setGpibControls
//#define DEBUG3  // gpibSendData
//#define DEBUG4  // spoll_h
//#define DEBUG5  // attnRequired
//#define DEBUG6  // EEPROM
//#define DEBUG7  // gpibReceiveData
//#define DEBUG8  // ppoll_h
//#define DEBUG9  // bluetooth
```

By default, debug messages will be sent to the serial port that is used for communication. Where the interface provides additional serial ports or where there are sufficient GPIO pins available to use *SoftwareSerial*, it is possible to send debug messages to an alternative serial port. This has the advantage that debug messages will no longer interfere with normal interface communications. The debug messages can be viewed on the alternative 'debug' port while normal interface operations are in progress on the communications port.

To enable this feature uncomment the following line in the *Debug Options* section in *AR488_Config.h*:

```
//#define DB_SERIAL_PORT Serial1
```

Set the serial port to the port that will receive the debug messages. Configure the baud rate, set the serial port type, and if using SoftwareSerial, the GPIO pins to be used, for example:

```
#define DB_SERIAL_BAUD 57600
#define DB_SW_SERIAL
#ifdef DB_SW_SERIAL
  #define DB_SW_SERIAL_RX 53
  #define DB_SW_SERIAL_TX 51
#endif
```

The above will configure a *SoftwareSerial* port at 57600 baud on GPIO pins 53 and 51. Please note that the maximum advisable speed for a *SoftwareSerial* port is 57600 baud.

Debug messages do not include messages shown when verbose mode is enabled with the *++verbose* command. When the interface is being directly controlled by another program, verbose mode should be turned off otherwise verbose messages may interfere with normal operations.

Custom Board Layout Section

The custom board layout section in the Config.h file can be used to create a custom pin layout for the AR488. This can be helpful for non-Arduino boards and where an adjustment to the layout is required in order to accommodate additional hardware. By default, the definition implements the Uno layout:

```
#define DIO1  A0  /* GPIB 1  */
#define DIO2  A1  /* GPIB 2  */
#define DIO3  A2  /* GPIB 3  */
#define DIO4  A3  /* GPIB 4  */
#define DIO5  A4  /* GPIB 13 */
#define DIO6  A5  /* GPIB 14 */
#define DIO7  4   /* GPIB 15 */
#define DIO8  5   /* GPIB 16 */

#define IFC   8   /* GPIB 9  */
#define NDAC  9   /* GPIB 8  */
#define NRFD  10  /* GPIB 7  */
#define DAV   11  /* GPIB 6  */
#define EOI   12  /* GPIB 5  */

#define SRQ   2   /* GPIB 10 */
#define REN   3   /* GPIB 17 */
#define ATN   7   /* GPIB 11 */
```

To make use of a custom layout, *AR488_CUSTOM* must be selected from the list of boards at the beginning of the Config.h file and the pin numbers/designations in the centre column (shown in bold) should be configured as required.

Please note that on some MCU boards, a number of GPIO pins may not be available as inputs and/or outputs despite a pad or connector being present. Please check the board documentation. Sometimes such information is revealed only in online forum discussions or blogs.

For any board (with the exception of supported boards where the GPIO pins designated for ATN and SRQ have not changed from the pre-defined layout), it will be necessary to disable *#define USE_PCINTS* and enable *#define USE_PINHOOKS* instead. For non-Arduino boards, it may also be necessary to disable *#define USE_SERIALEVENT* to remove dependence on the Serial event interrupt.

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-USB 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 instrument 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|1|2|3]
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 signal is asserted by default unless its status is changed by the ++ren command.

When the ++llo command is issued without a parameter, it behaves the same as it does on the Prologix controller. The LLO signal is sent to the currently addressed instrument and this locks out the LOCAL key on the instrument control panel. 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 display or control panel to indicate that remote control is active and front/rear panel controls will be disabled.

If the ++llo command is issued with the ‘all’ parameter, this will send the LLO signal to the bus, putting every instrument into remote control mode simultaneously. At this point, instruments will not yet show the REM indicator and it may still be possible to operate the front panel controls. On some instruments the LOCAL key may be locked out. However, as soon as an instrument has been addressed and sent a command (assuming that a LOC signal has not been sent yet first), the controller will automatically lock in remote control of that instrument , the REM indicator will be displayed and front/rear 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 place all instuments back into local mode, re-enabling the LOCAL key and panel controls 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 puts all devices on the GPIB bus in local control state. The REM indicator should no longer be visible when the instrument has returned to local control state.

Modes: controller

Syntax: ++loc [all]

++lon

The *++lon* command configures the controller to listen only to traffic on the GPIB bus. In this mode the interface does require to have a GPIB address assigned so the assigned GPIB address is ignored. Traffic is received irrespective of the currently set GPIB address. The interface can receive but not send, so effectively becomes a "listen-only" device. When issued without a parameter, the command returns the current state of "lon" mode.

Modes: device

Syntax: ++lon [0 | 1]
where 0=disabled; 1=enabled

++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 act 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.

When sending data followed by a command, the buffer must first be read by the controller before a subsequent command can be accepted, otherwise the command will be treated as characters to be appended to the existing data in the buffer. Once the buffer has been read, it is automatically cleared and the parser can then detect the ++ command prefix on the next line. Therefore sufficient delay must be allowed for the buffer to be read before sending a subsequent command.

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 32,000 milliseconds (32 seconds).

Modes: controller

Syntax: ++read_tmo_ms <time>
where <time> is a decimal number between 0 and 32000 representing milliseconds

++rst

Perform a reset of the controller.

Please note that the reset may fail and hang the board under certain circumstances. These include:

- the board has an older bootloader. The older bootloader had an problem with not clearing the MCUSR register which triggers another reset while the bootloader is being executed, which causes a perpetual restart cycle. The solution here is to update the bootloader. The newer Optiboot bootloader does not have this problem.

- using a 32u4 board (Micro, Leonardo) programmed with an AVR programmer with no bootloader. There is at present no solution to this problem. When programming with an

AVR programmer, use a recent IDE version to export the binaries and upload the version with the bootloader to the board.

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.

Most, if not all Arduino AVR boards support EEPROM memory, however boards from other vendors may not provide this support. If the command is run on a board that does not support EEPROM, then the following will be returned:

EEPROM not supported.

The *++savecfg* command will save the following current parameter values: 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 multiple 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.

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

auto	*0*
eoi	*0 (disabled)*
eor	*0 (CR+LF)*
eos	*0 (CR+LF)*
eot_enable	*0 (disabled)*
eot_char	*0*
GPIB address - controller	*0*
GPIB address - primary	*1*
GPIB address - secondary	*0*
mode	*controller*
read_tmo_ms	*1200*
status byte	*0*
version string	*default version string*

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. To ensure that settings are saved, after using the *++default* command, configure the interface as required and then use *++savecfg* to save the settings to EEPROM*. The interface can be returned to its default state by using *++default* followed by *++savecfg* without making any further configuration changes.

* this assumes that the board being used supports saving to EEPROM.

Modes: controller, device

Syntax: ++default

++eor

End of receive. While *++eos* (end of send) selects the terminator to add to commands and data being sent *to* the instrument, the *++eor* command selects the expected termination sequence when receiving data *from* the instrument.

The following termination sequences are supported:

Option	*Sequence*	*Hex*
0	CR + LF	0D 0A
1	CR	0D
2	LF	0A
3	None	N/A
4	LF + CR	0A 0D
5	ETX	03
6	CR + LF + ETX	0D 0A 03
7	EOI signal	N/A

The default termination sequence is CR + LF. If the command is specified with one of the above numeric options, then the corresponding termination sequence will be used to detect the end of the data being transmitted from the instrument. If the command is specified without a parameter, then it will return the current setting. If option 7 (EOI) is selected, then *++read eoi* is implied for all *++read* instructions as well as any data being retuned by the instrument in response to direct instrument commands. An EOI is expected to be signalled by the instrument with the last character of any transmission sent. All characters sent over the GPIB bus are passed to the serial port for onward transmission to the host computer.

Modes: controller

Syntax: ++eor[0-9]

++id

This command sets the identification parameters for the interface. Here you can set the instrument name and optional serial number. This command also sets the information that can be used by the interface to respond to a SCPI **idn?* which may be useful where the instrument itself cannot provide such a response. For further information also see the *++idn* command.

The command has one of three invocations and an optional parameter:

++id name
This sets a short name for the interface. The name can be up to 15 characters long and should not include spaces. If the command is specified without a parameter, it will return the current name of the interface. By default, the name is not set and the command will not return a value.

++id serial
This sets an optional serial number for the interface. In the event that there are multiple instances of identical instruments on the bus, each instrument can be given a unique serial number up to 9 digits long. When specified without a parameter, the command returns the currently configured serial number. By default the serial number is not set and the command will return '000000000'.

++id verstr
Sets 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'. When no parameter is given, the command returns the current version string.

Examples:

```
++id name HP3478A
++id serial 347800001
++id verstr GPIB-USB
++id verstr
```

Modes: controller

Syntax: ++id name [name]
++id serial [serialnum]
++id verstr [version string]

++idn

This command is used to enable the facility for the interface to respond to a SCPI *idn? Command. Some older instruments do no respond to a SCPI ID request but this feature will allow the interface to respond on behalf of the instrument using parameters set with the *++id* command. When set to zero, response to the SCPI **idn?* command is disabled and the request is passed to the instrument. When set to 1, the interface responds with the name set using the *++idn name* command. When set to 2, the instrument also appends the serial number using the format *name-99999999*.

Modes: controller

Syntax: ++idn[0-2]

++macro

Instrument control usually requires a sequence of commands to be sent to the instrument to set it up or to perform a particular task. Where such a sequence of commands is performed regularly and repeatedly, it is beneficial to have a means to pre-program the sequence and to be able to run it with a single command.

The AR488 allows up to 9 sequences to be programmed into the Arduino sketch that can be run using the ++macro command. When no parameters have been specified, the macro command will return a list of numbers indicating which macros have been defined and are available to use. When called with a single number between 1 and 9 as a parameter, the command will run the specified macro.

Programming macros is beyond the scope of this manual and will be specific to each instrument or implemented programming language or protocol.

Modes: controller

Syntax: ++macro [1-9]

++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

++ren

In controller mode, this command turns the REN signal on and off. When REN is asserted, the controller can remote-control any device on the BUS. With the REN signal turned off, the

controller can no longer remote-control devices, but can still communicate with them. This is used primarily for diagnostics.

When set to 0, REN is un-asserted. When set to 1, REN is asserted. By default, in controller mode, REN will be asserted.

When REN is used to control the SN75161 GPIB transceiver integrated-circuit, this command is unavailable and will simply return *Unavailable*. (see the *Configuration* and the *Building an AR488 GPIB Interface* sections for more information). When issued without a parameter, the command returns the current status of the REN signal.

Modes: controller

Syntax: ++ren [0|1]

++repeat

Provides a way of repeating the same command multiple times, for example, to request multiple measurements from the instrument.

Between 2 and 255 repetitions can be requested. It is also possible to request a delay between 0 to 10,000 milliseconds (or 10 seconds) between each repetition. The parameter buffer has a maximum capacity of 64 characters, so the command string plus any parameters cannot exceed 64 characters in total. Once started, there is no mechanism to stop the repeat loop once it has begun. The command will run the number of iterations requested and stop only when the request is complete.

Modes: controller

Syntax: ++repeat count delay cmdstring
where:
count is the number of repetitions from 2 to 255
delay is the time to wait between repetitions from 0 to 10,000 milliseconds
cmdstring is the command to execute

++setvstr

This command is DEPRECATED and will be removed in future versions. Please refer to the notes for the *++id verstr* command instead. It sets 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 0 (default), in order to obtain the status byte when SRQ is asserted, a serial poll has to be conducted manually using the *++spoll* command.

When *++srqauto* is set to 1, the interface will automatically detect when the SRQ signal has been asserted by an instrument and will automatically conduct a serial poll, returning 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 subsequent serial poll will be 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.

Without parameters, this command returns the present status of the SRQauto. It returns 0 if a serial poll is not automatically executed (default) and 1 if a serial poll is automatically executed.

Modes: controller

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

++tmbus

The GPIB bus protocol is designed to allow the bus to synchronise to the speed of the slowest device. However, under some circumstances it may be desirable to slow down the bus. The *tmbus* parameter introduces a periodic delay of between 0 to 30,000 microseconds between certain operations on the bus and so slows down the operation of the GPIB bus. The greater the delay, the slower the bus will operate. Under normal running conditions this parameter should be set to zero, which is the default setting.

Modes: controller, device

Syntax: ++tmbus [value]
where [value] is between 0 and 30,000 microseconds

++ton

The *++ton* command configures the controller to send data only on the GPIB bus. When in this mode, the interface does not require to have a GPIB address assigned and the address that is set will be ignored. Data is placed on the GPIB bus as soon as it is received via USB. Only one sender

can exist on the bus, but multiple receivers can listen to and accept the transmitted data. The interface can send, but not receive, so effectively becomes a “talk-only” device. When issued without a parameter, the command returns the current state of “ton” mode.

Modes: device

Syntax: ++ton [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, NANO or MEGA2560 board, 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.

For connection details and wiring diagrams for specific boards, please see:

Appendix A – Uno and Nano
Appendix B – Mega 2560
Appendix C – Micro 32u4

Ideally, in a GPIB cable, ground pins 18, 19, 20, 21, 22, 23 should be connected to a ground wire that forms a twisted pair with the DAV, NRFD, NDAC, IFC, SRQ and ATN control wires, and a shielded twisted pair cable with sufficient multiple pairs would be required. However, if such a cable is not available, then linking them together and connecting them to GND on the Arduino side should suffice, especially if sufficient numbers of conductors are not available.

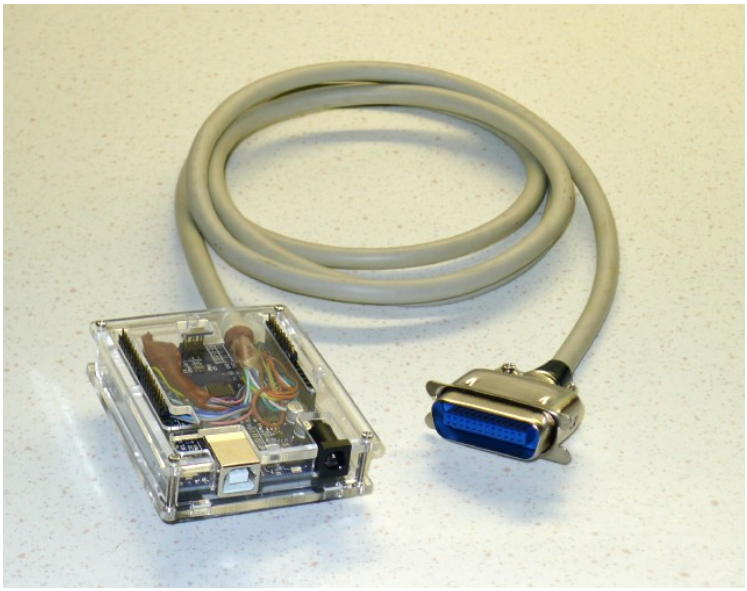
Further information can be found by following the links below:

Additional GPIB pinout information - Link 1
Additional GPIB pinout information - Link 2

Once the cable has been completed, the sketch should then be downloaded to the Arduino board and the interface should be ready to test. In order to provide multi-platform compatibility, the AR488 firmware sketch is modular and comes in several files:

Filename:	*Purpose:*
AR488.ino	This is the main AR488 firmware sketch
AR488_Config.h	This is the configuration file. All configuration options are set here.
AR488_Hardware.h	This is the hardware support C++ header file
AR488_Hardware.cpp	This is the hardware support C++ program file

The firmware is supplied in a ZIP file. Download and unpack all files into a directory called AR488. Load the main sketch, AR488.ino into the Arduino IDE. This should open all files into separate tabs. Edit AR488_Config.h as required and save. Then select the correct board from the list of boards within the Arduino IDE, Tools | Board menu option and compile and upload the sketch.



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.

Multiple Arduinos on the bus and problems with instruments

The AR488 can be used in both controller mode and device mode and only ONE controller can be active at any one time. When there is just one Arduino controller on the bus controlling one or more instruments, this does not present a problem, provided that the Arduino is operating within its current handling limits.

However, it is possible to have one AR488 operating as a controller and another as a device simultaneously on the bus along with other instruments. In this situation and without any additional buffering (see the following section: *SN7516x GPIB transceiver integrated circuits*), problems can arise when two or more Arduinos are connected to the GPIB bus and one of them is powered down. Such problems are manifest by instruments failing to respond to the *++read* or other commands, failing to respond to direct instrument commands, or other erratic bus communication problems.

The reason for this is because when powered down, Arduino control pins do not present with a high impedance. 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 effectively 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 unpredictable manner the result of which may be the aforementioned interference with the proper functioning of other equipment on the GPIB bus. This is a parasitic power phenomenon that is not specific to Arduino microcontrollers and that affects various other devices also. Further information regarding this phenomenon 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 disconnected 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.

SN7516x GPIB transceiver integrated circuits

The AR488 firmware supports SN75160 and SN75161 GPIB transceiver integrated circuits. These ICs provide a buffer between the Arduino and the GPIB bus and allow the full 48mA drive current for a GPIB device. In addition, when powered down, these devices present a high impedance to the GPIB bus so that the connected device does not interfere with the operation of the bus. This solves the 'parasitic power' problem that ocurrs when using Arduinos connected directly without buffering to the the GPIB bus and means that theinterface can be safely powered down without affecting communication on the GPIB bus.

In order to use these GPIB transceiver ICs, at least one SN75160 and one SN75161 will be required and a separate daughterboard will have to be built. The SN75160 provides 3-state outputs for the data bus, whereas the SN75161 provides similar isolation for the GPIB control signals. Connection details can be found in *Appendix A*, which details connections for the Uno board. A similar approach can be used for any other board using available GPIO pins.

Operation of the SN75160 is simple. The Arduino outputs are connected to the 'Terminal I/O ports' side of the IC and the GPIB bus DIO lines to the 'GPIB I/O ports side. The PE pin should be connected to VCC in order to provide 3-state output. The TE (talk-enable) pin is connected to a GPIO pin on the Arduino. The GPIO pin is defined in *Config.h*. For further details see the *Configuration* section.

The operation of the SN75161 is a little more complex as part of the IC is controlled by the TE pin, but also by the DC (direction-control) pin. The TE pin is connected to the same GPIO pin as the 75160 TE pin. The DC pin needs to be driven separately. This can be achieved by connecting DC to a seperate GPIO pin which can also be defined in *Config.h*. Alternatively, it can be controlled by the REN signal. The REN signal is asserted (LOW) in controller mode and un-asserted (HIGH) in device mode which conveniently corresponds to the drive signal required for DC to switch between controller and device mode. When REN is being used to control DC, it cannot be turned off as this would switch the IC into device mode and communication would fail. For this reason, the ++REN command is not available in this configuration (see *++REN* in the *Custom Commands* section for details on the behaviour of this command).

The SN75162 IC differs from the SN75161 in that the REN and IFC signals are independently controlled. The input required is the inverse of the DC signal. Conceivably a separate GPIO pin could be used to drive the SC pin of the SN75162 but this is currently untested and unsupported. Alternatively some means of hardware inversion could be devised and the pin connected to DC, but in this case, experiment at your own risk.

Arduino brownout detection setting

The first three bits of the Arduino extended fuse determine the brownout detection (BOD) setting. BOD will hold the processor in the reset state when the power rail voltage falls below a specific threshold. There are three threshold levels that can be set depending on the bits that is set.

On the boards that were used for development, the default setting of the Extended Fuse seems to be FD, which means that the last three bits will be 101 and therefore that the BOD level is set to BODLEVEL1.

It has been reported that when BOD is disabled (e.g. fuse set to FF) and the Arduino signal pins are connected to power, that under some circumstances the Arduino flash memory can get corrupted and the sketch will have to be downloaded again. It is therefore inadvisable to disable BOD on an Arduino being used as a GPIB interface.

Arduino BOD settings are as follows:

BOD Level:	*Bit setting*	*Threshold*
DISABLED	111	BOD disabled
BODLEVEL0	110	1.7 – 2.0v (avg 1.8v)
BODLEVEL1	101	2.5 – 2.9v (avg 2.7v)
BODLEVEL2	100	4.1 – 4.5v (avg 4.3v)

To check the extended fuse setting, the following AVRDUDE command line can be used:

UNO/NANO:
avrdude -P /dev/ttyACM0 -b 19200 -c usbasp -p m328p -v

MEGA 2560:
avrdude -P /dev/ttyACM0 -b 115200 -c usbasp -p m2560 -v
MEGA 32u4:
avrdude -P /dev/ttyACM0 -b 115200 -c usbasp -p m32u4 -v

The ATMega328p part can be specified as *-p m328p* or *-p atmega328p*. The Mega 2560 and Mega 32u4 can also be specified using either convention. If your Arduino has a 328pb processor IC, then this will have a different signature to the 328p and the -p parameter needs to be specified as *-p m328pb* or *-p atmega328pb*.

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 devise 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.

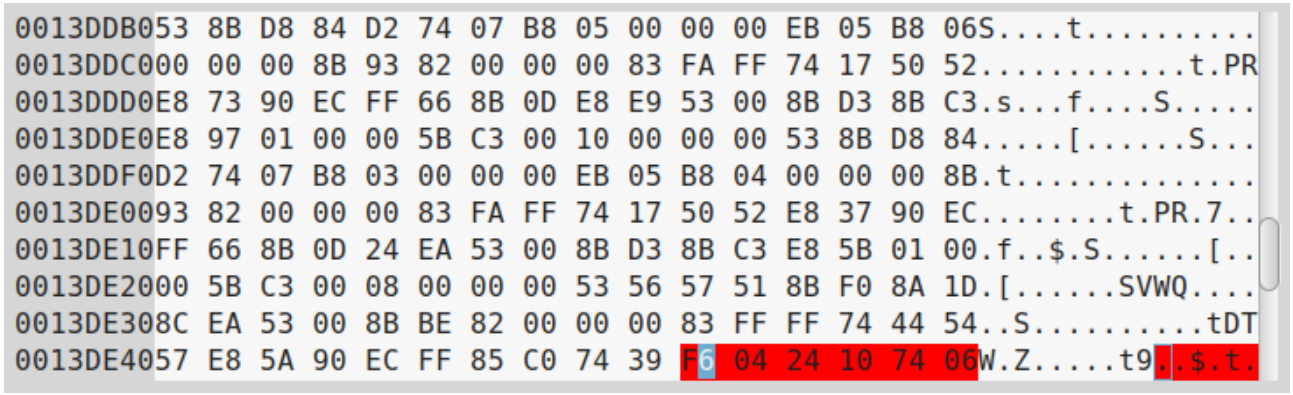
Hacking the Ezgpib binary

If you are familiar with using a hex editor, there is another approach that involves editing the EZGPIB.EXE binary to prevent it looking for an RTS signal being asserted. If the standard Windows USBSER.SYS driver is used, this never happens, so EZGPIB will never find the GPIB adapter. This workaround involves changing a specific byte in the RTS Check routine.

Open up a copy of EXGPIB.EXE version 20121217 in a hex editor. Look for the HEX sequence:

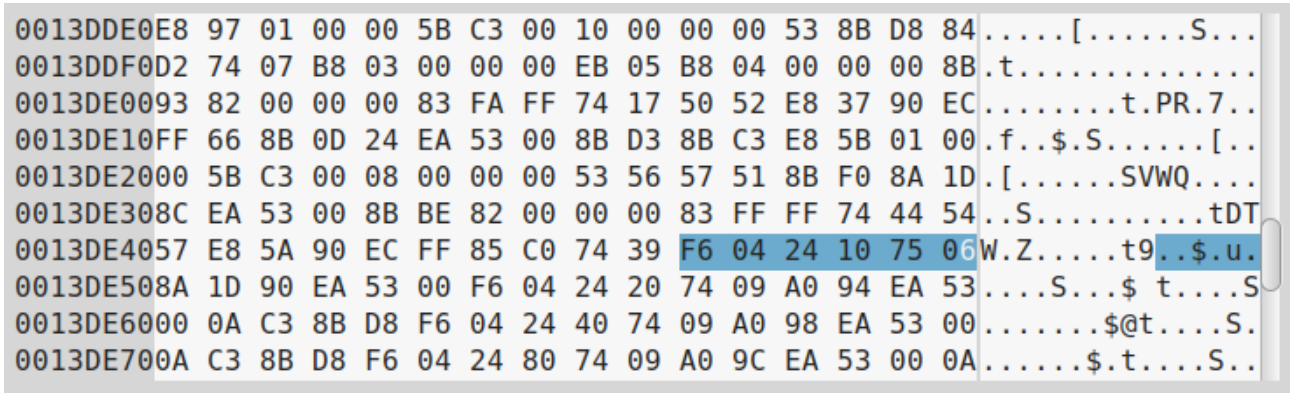
```
F6 04 24 10 74 06
```

Note, that these instructions can also be found on http://www.dalton.ax/gpib/, but show the sequence as `F6 02 24 10 74 06`. I found the sequence to be as above. I'm not sure whether this is an error or because my binary is different from the one that the author was working with. If you can't find the sequence with `04`, check for the one with `02`.



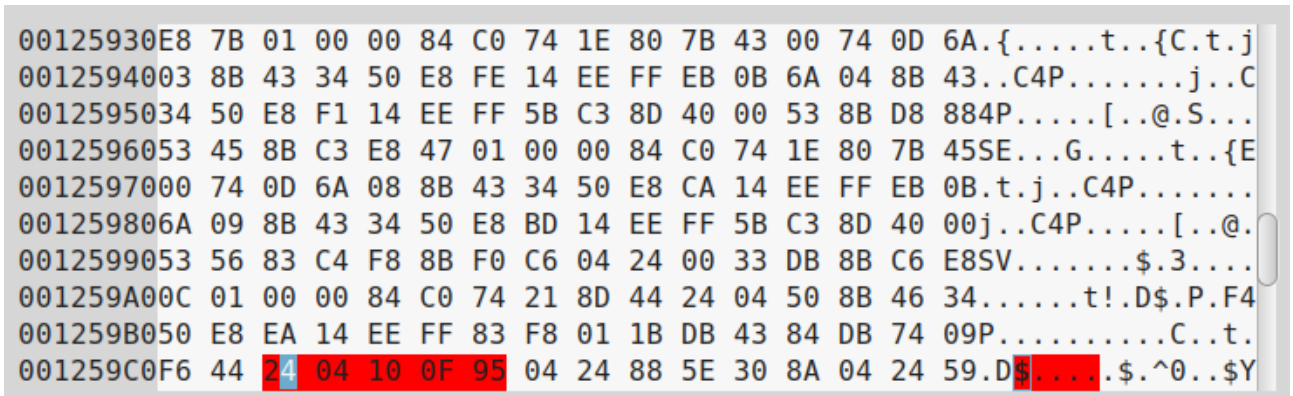
That sequence is the check for RTS. Change the penultimate byte to 75, so that the sequence now reads:

```
F6 04 24 10 75 06
```



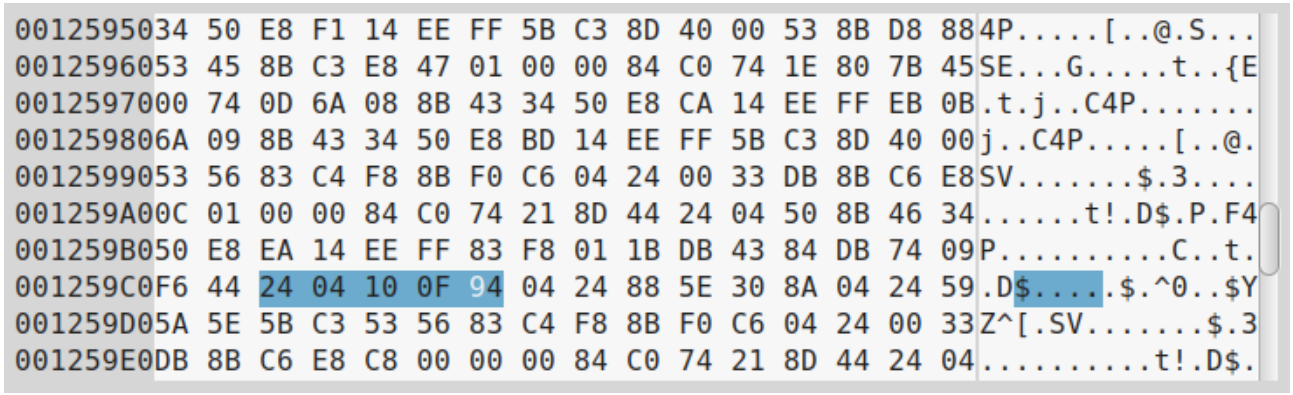
Now look for sequence:

```
24 04 10 0F 95
```



Change the last byte to 94 so that the sequence now reads:

```
24 04 10 0F 94
```



Save the file and close the hex editor. EZGPIB should now find your adapter.

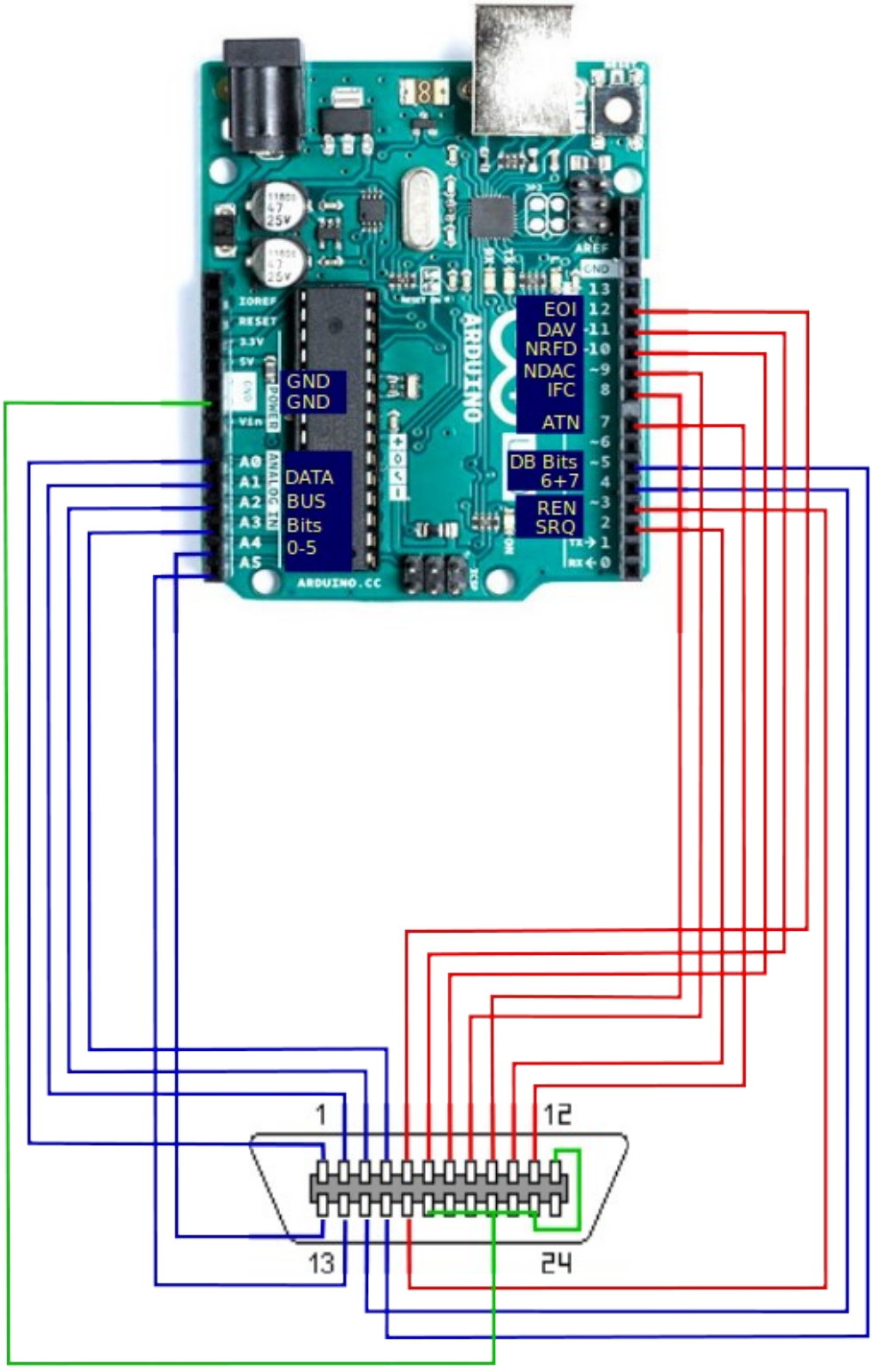
Appendix A – Connection and technical information for Uno and Nano boards

Connection details:

These connections are required between the Arduino UNO/Nano and the IEEE488 connector:

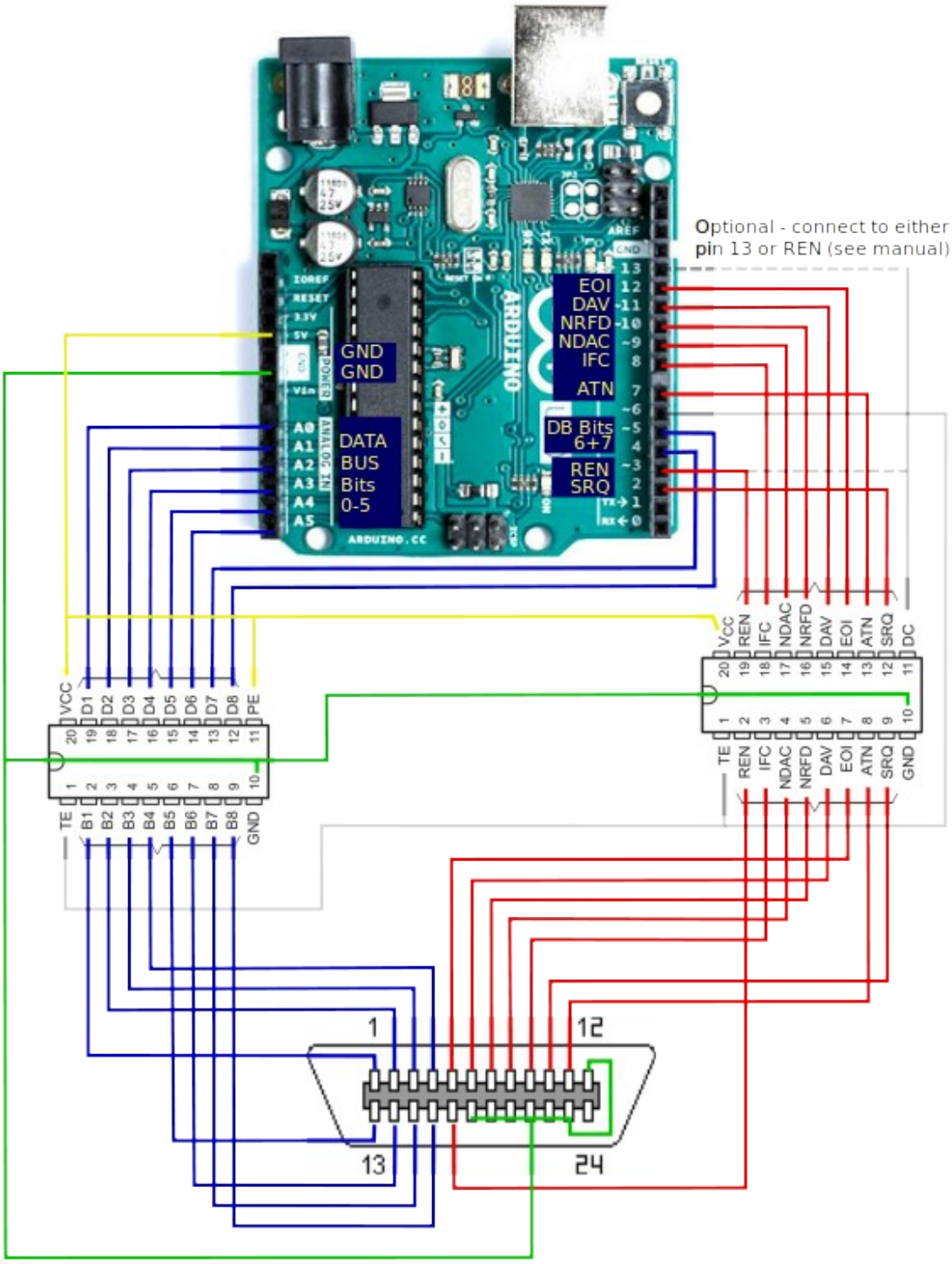
Arduino:	*GPIB connector:*	*Function:*
D2	10	SRQ
D3	17	REN
D7	11	ATN
D8	9	IFC
D9	8	NDAC
D10	7	NRFD
D11	6	DAV
D12	5	EOI
A0	1	DIO1
A1	2	DIO2
A2	3	DIO3
A3	4	DIO4
A4	13	DIO5
A5	14	DIO6
D4	15	DIO7
D5	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

Wiring diagram:



When using SN75160 and SN75161 integrated circuits, the connections involve at least one extra pin to control the talk-enable (TE) pin of the IC. The PE pin on the SN75160 is connected to VCC to maintain a 3-state outputs when TE is high. Connecting PE to ground will allow the outputs to function in pullup-enable mode when TE is high.

On the SN75161, the DC pin can be connected to a separate GPIO pin on the Uno/Nano, or, since ren is always asserted when in controller mode and de-asserted in device mode, to the GPIO pin used for the REN signal.



Appendix B – Connection and technical information for Mega 2560 boards

Connection details:

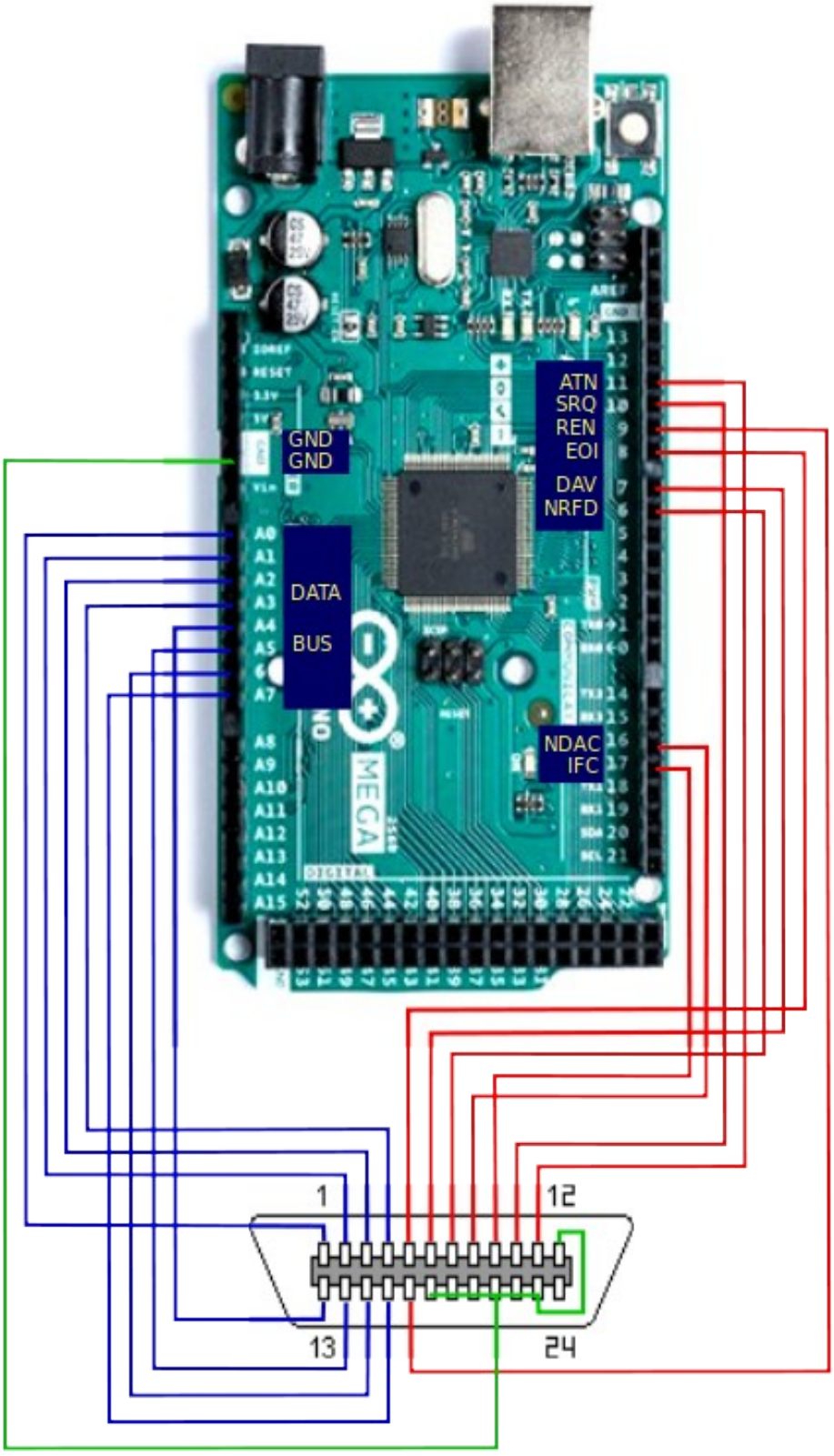
The pinout on the Mega 2560 is as follows:

Arduino:	*GPIB connector:*	*Function:*
D6	7	NRFD
D7	6	DAV
D8	5	EOI
D9	17	REN
D10	10	SRQ
D11	11	ATN
D16	8	NDAC
D17	9	IFC
A0	1	DIO1
A1	2	DIO2
A2	3	DIO3
A3	4	DIO4
A4	13	DIO5
A5	14	DIO6
A6	15	DIO7
A7	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

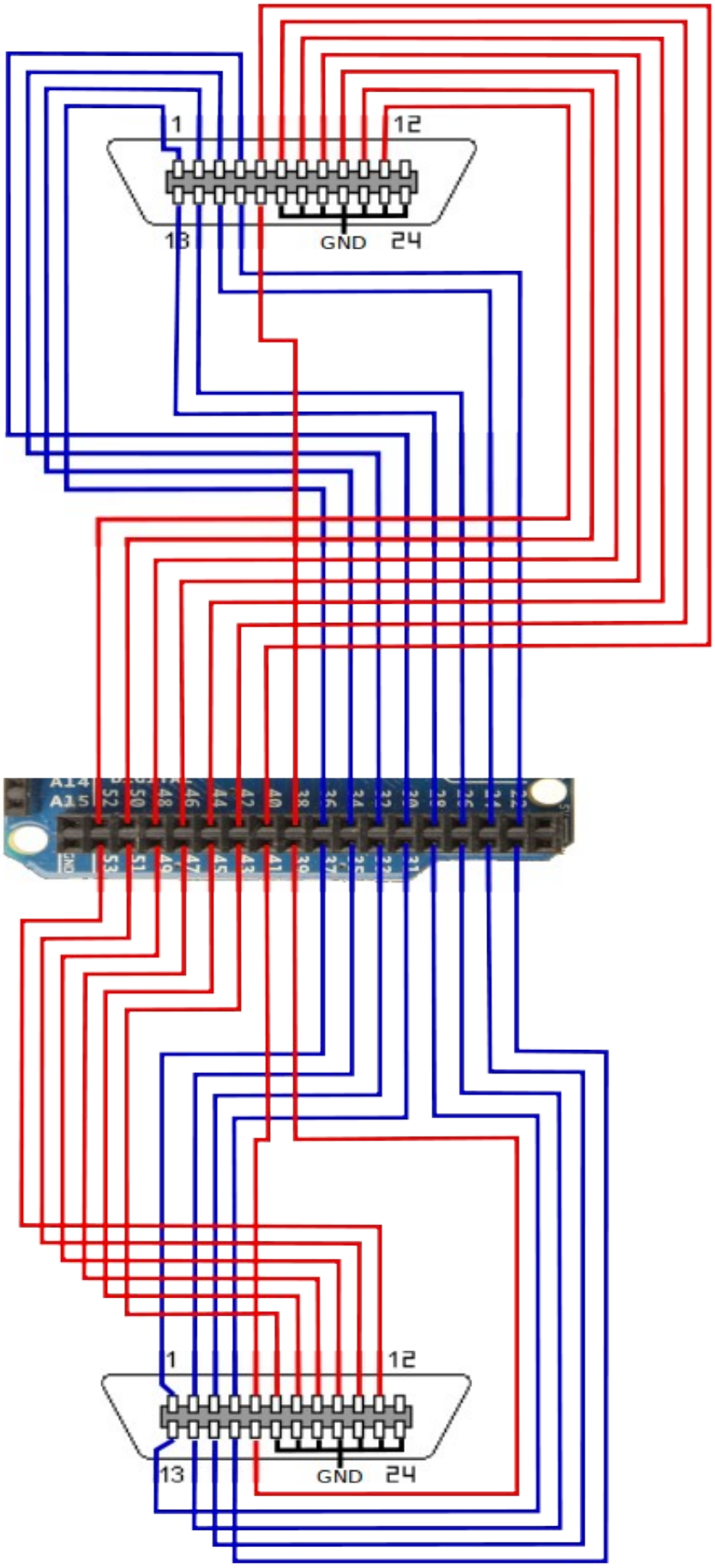
The layout on the Mega was chosen so as to leave pins A8-A15 and the two rows of pins at the top of the board free for expansion including for displays and other peripherals.

Pins 16 and 17 correspond to Serial2. As these have been used for controlling signals on the GPIB bus, they cannot be used for serial communication. If *Serial2.begin* is added to the sketch, these pins will be enabled for serial communication and will no longer function as GPIB control signals. In addition to the default serial port (RX0 and TX0), Serial1 and Serial3 are still available for expansion if required. These two pins were chosen for GPIB signals as they belong to port H along with pins 6 – 9.

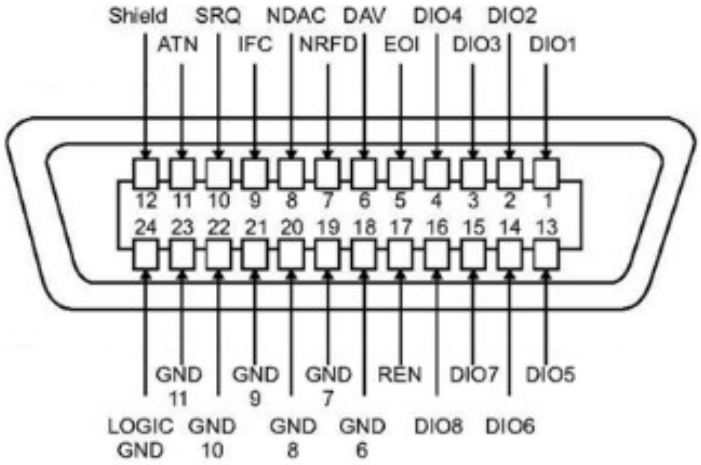
Wiring Diagram (default layout - AR488_MEGA2560_D):



Wiring Diagram (layout E1 and E2):

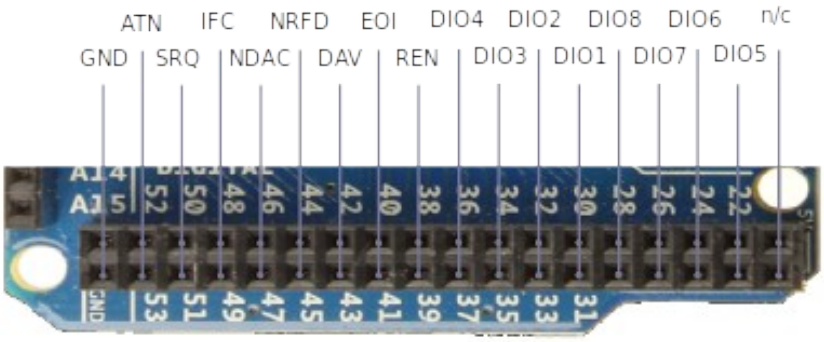


GPIB IEEE488 connector



Male (plug) rear view
Female (socket) front view

Arduino Mega 2560 E1/E2 pinout



Appendix C – Connection and technical information for 32u4 based boards

Micro Connection details:

The pinout on the Micro is as follows:

Arduino:	*GPIB connector:*	*Function:*
A2	7	NRFD
A1	6	DAV
A0	5	EOI
D5	17	REN
D7	10	SRQ
D2	11	ATN
A3	8	NDAC
D4	9	IFC
D3	1	DIO1
D15	2	DIO2
D16	3	DIO3
D14	4	DIO4
D8	13	DIO5
D9	14	DIO6
D10	15	DIO7
D6	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

The Micro is a very small form factor board that can be adapted to fit on the back of an IEEE488 connector. The design was contributed by Artag:

https://www.eevblog.com/forum/projects/ar488-arduino-based-gpib-adapter/msg2718346/#msg2718346

Adapter boards are available from:
https://oshpark.com/shared_projects/HrS1HLSE

Leonardo R3 Connection details:

The pinout on the Leonardo R3 is as follows:

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

The Leonardo R3 has a similar form factor to the Uno. It uses a 32u4 MCU rather than a 328P and has a micro USB port. Instead of a CH340 UART it uses USB CDC emulated serial ports and has one separate hardware serial port available on on RX1 and TX1, whereas the Uno shares these pins with USB. It requires no modification to work with KE5FX tools. The board pin layout is the same as the Uno and the above pinout is identical to the Uno.

Appendix D – XDIAG command notes

The ++xdiag command can be used to test that individual lines are being asserted / unasserted. The command takes two parameters: mode and value. To manipulate control lines use mode 1, and for data lines use mode 0. To assert a line/data bit, simply speciy one of the values in the table below. To assert multiple lines/bits simultaneously, simply add the values. To un-assert a line, subtract its value from 255 The following two tables list the GPIB signals and the command used to assert them.

Command signals:

Assert IFC	`++xdiag 1 1`
Assert NDAC	`++xdiag 1 2`
Assert NRFD	`++xdiag 1 4`
Assert DAV	`++xdiag 1 8`
Assert EOI	`++xdiag 1 16`
Assert REN	`++xdiag 1 32`
Assert SRQ	`++xdiag 1 64`
Assert ATN	`++xdiag 1 128`
Assert ALL	`++xdiag 1 255`
Un-assert ALL	`++xdiag 1 0`

Data bits:

Assert DA01	`++xdiag 0 1`
Assert DA02	`++xdiag 0 2`
Assert DA03	`++xdiag 0 4`
Assert DA04	`++xdiag 0 8`
Assert DA05	`++xdiag 0 16`
Assert DA06	`++xdiag 0 32`
Assert DA07	`++xdiag 0 64`
Assert DA08	`++xdiag 0 128`
Assert ALL	`++xdiag 0 255`
Un-assert ALL	`++xdiag 0 0`