

# **AR488 GPIB Controller**

## ***User Manual***

*Version:*

**0.53.02**

*Dated:*

**8<sup>th</sup> April 2025**

# **Twilight Logic**

01001010 01000011 01001000

# Table of Contents

The AR488 GPIB Controller.....	5
Introduction.....	5
Driver Installation.....	6
Firmware Upgrades.....	6
Wireless Operation.....	7
Bluetooth.....	7
WiFi.....	7
Client Software.....	7
Applications.....	8
GPIB.....	8
A Brief Overview.....	8
GPIB Addressing.....	9
Operating Modes.....	9
Controller Mode.....	9
Device Mode.....	10
Transmission of Commands and Data.....	10
Sending Data and Special Characters.....	11
Receiving Data.....	11
Interrupting Transmission of Data.....	12
Listen-only (LON), Talk-only (TON) and Promiscuous modes.....	12
Configuration.....	13
The Configuration File.....	13
Board Selection.....	13
Serial Port Configuration.....	14
Hardware Serial Ports.....	14
Software Serial Library.....	15
Enabling Bluetooth Support.....	16
Configuring the Debug Serial Port.....	16
Serial Handling in Applications and Scripts.....	18
Macro Support.....	18
SN7516x GPIB Transceiver Support.....	21
Custom Board Layout Section.....	22
Building an AR488 GPIB Interface.....	24
Overview.....	24
Multiple Arduino Interfaces on the Bus.....	26
SN7516x GPIB Transceiver Integrated Circuits.....	26
Arduino Brownout Detection Setting.....	27
USB-to-Serial UART ICs and Handshaking.....	29
CH340G Serial UART.....	30
Workaround 1.....	31
Workaround 2.....	31
FTDI UART (FT232RL).....	31
CP2102 UART.....	31
AR488 Bluetooth Support.....	32
The HC05 and HC06 Modules.....	32
Enabling Bluetooth on the AR488.....	34
Power-up and Detection.....	35
Pairing the AR488 Bluetooth Interface.....	36
Windows 10.....	36

Linux.....	42
Troubleshooting Information for Linux.....	44
Working With Third Party Software.....	47
EZGPIOB and the Arduino Bootloader.....	47
Solution 1 - Capacitor.....	47
Solution 2 – Hacking the EZGPIOB Binary.....	48
The KE5FX Toolkit.....	50
Command Reference.....	51
+addr.....	51
++allspoll.....	52
++auto.....	53
++clr.....	54
++dcl.....	54
++default.....	55
++eoi.....	56
++eor.....	57
++eos.....	58
++eot_enable.....	58
++eot_char.....	59
++eot_char.....	60
++flag.....	61
++fndl.....	62
++help.....	63
++id.....	64
++idn.....	65
++ifc.....	65
++llo.....	66
++loc.....	67
++lon.....	68
++macro.....	69
++mode.....	70
++ppoll.....	71
++prom.....	72
++read.....	73
++read_tmo_ms.....	74
++ren.....	74
++repeat.....	75
++rst.....	76
++savecfg.....	77
++send.....	78
++spoll.....	79
++srq.....	79
++srq.....	81
++srqauto.....	82
++status.....	83
++tct.....	84
++ton.....	85
++trg.....	86
++ver.....	86
++verbose.....	87
++xdiag.....	88
Appendices.....	90

A – ATmega328P Boards.....	91
Connection details for UNO and Nano.....	91
Wiring diagram.....	92
B - ATmega2560 boards.....	94
Connection details for layout AR488_MEGA2560_D.....	94
Connection details for layout AR488_MEGA2560_E1.....	96
Connection details for layout AR488_MEGA2560_E2.....	97
Wiring Diagram for layout - AR488_MEGA2560_D.....	98
Wiring Diagram - layout E1 and E2.....	99
E1/E2 layout end header pinout details.....	100
C – ATmega32u4 Boards.....	101
Connection details for the Arduino Micro.....	101
Connection details for the Leonardo R3.....	103
D – Raspberry RP2040/RP2350 Boards.....	104
Connection details for Raspberry Pico/W layout RAS_PICO_L1.....	104
Connection details for Raspberry Pico/W layout RAS_PICO_L2.....	106
E – Espressif ESP32 Boards.....	108
Connection details for layout ESP32_DEVKIT1_WROOM_32.....	108
Connection details for layout ESP32_TTGO_T8_161.....	109
Connection details for layout ESP32_LOLIN32_161.....	110
Connection details for layout ESP32_ESP32DEV.....	111
F – MightyCore ATmega644/1284 Boards.....	112
Connection Details for layout AR488_MEGA644P_MCGRAW.....	112
G - IEEE Connector Pinout.....	113

# The AR488 GPIB Controller

## Introduction

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 328p boards, the Mega 2560 board, Micro and Leonardo 32U4 boards and MicroCore ATmega644 and ATmega1024 boards. It provides a low cost alternative to other commercial interfaces. There is also a forked version for the ESP32 which can be found here:

<https://github.com/douardda/AR488/tree/esp32>

A list of other related projects and utilities built by others can be found here:

<https://www.eevblog.com/forum/projects/ar488-arduino-based-gpib-adapter/msg2184323/#msg2184323>

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 project can be expanded by optionally adding the SN75160 and SN75161 GPIB transceiver integrated circuits. These devices provide proper tri-state IO and buffering between the Arduino and the GPIB bus. Details of construction as well as mapping of the 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. A number of additional custom commands have also been implemented. Details of all commands can be found in the *Command Reference* section.

The Table of Contents section of this document is hyperlinked. Use Ctrl + Right-Click on the mouse to go directly to the relevant section of the document.

## Driver Installation

In order to be able to connect to a computer via USB, the above mentioned boards will usually have an on-board serial to USB converter chip. Windows 7, Windows 10 and Windows 11 will automatically recognize the 16u2 chip used on genuine Arduino boards as well as the CH340G chips used on alternative boards. It will also recognize the FT232RL adapter board as well as the CP2102 used on the ESP32. Windows should automatically install the appropriate driver from Windows Update and then present a COM port.

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](http://www.wch.cn/download/CH341SER_EXE.html)

The official CVP2102 driver can be found here:

<https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers?tab=downloads>

There seems to be no official driver source for the 16u2 chip.

Modern versions of Linux such as Mint, MX and Ubuntu contain these driver modules in the kernel and so will automatically recognize both chipsets.

There is also an interesting alternative Nano board made by Logic Green. This has an LGT8F328P (328p) clone MCU capable of twice the speed of the Nano. The UART is a Holtek HT42B534-2 IC. This UART chip does not appear to be officially supported in Linux seem to be recognized by the Kernel. There seems to be no official driver source for the Holtek chip but it should be recognized by Windows.

## Firmware Upgrades

The firmware is updated by compiling the source in the Arduino IDE and then uploading the compiled code over USB, but an AVR programmer can also be used to upload the compiled firmware binary to the Arduino microcontroller.

Updates are available from <https://github.com/Twilight-Logic/AR488>

# Wireless Operation

## Bluetooth

The AR488 interface can communicate using a Bluetooth HC05 master module. The HC05 module supports command mode and can be auto-configured by the AR488 sketch using serial communication. Details of how to enable and configure this feature can be found in the *Configuration* and *Bluetooth* sections. While it is possible to use the HC06 slave module to provide Bluetooth communication, the slave module does not support command mode so cannot be configured remotely by the AR488 sketch. It has to be configured manually. For this reason, the master module is recommended.

## WiFi

David Douard has produced fork of the AR488 project running on an ESP32. His project can be found here:

<https://github.com/douardda/AR488-ESP32/tree/esp32>

## Client Software

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

Serial Terminal software:

<https://www.putty.org/>

EZGPIB:

<http://www.ulrich-bangert.de/html/downloads.html>

KE5FX GPIB Toolkit (GPIB Configurator):

<http://www.ke5fx.com/gpib/readme.htm>

Python scripts and almost anything that can access the Prologix adapter

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

# Applications

There are a number of compatible applications that can be used with the AR488:

Luke Mester's Instrument Control Software (HP3478 and WaveTek 178) :

<https://mesterhome.com/gpibsw/>

Data logging software by Nirav Patel:

<https://www.eevblog.com/forum/projects/ar488-arduino-based-gpib-adapter/msg3586488/#msg3586488>

PyMeasure:

<https://www.eevblog.com/forum/projects/ar488-arduino-based-gpib-adapter/msg4604662/#msg4604662>  
<https://pymessage.readthedocs.io/en/latest/>

## GPIB

### A Brief Overview

GPIB is an early technology designed for interconnecting computers, peripherals (displays, disks and printers) and instruments originally designed by HP and known as the Hewlett-Packard Interface Bus (HPIB). This was later published by the Institute of Electrical and Electronics Engineers as document IEEE488/1975. It became adopted by other companies and became known as the General Purpose Interface Bus (GPIB). The standard has been updated and released as IEEE488.1 and later IEEE488.2.

The GPIB bus is a parallel bus consisting of 8 data signals to transmit characters plus 8 control signals. It uses relatively heavy 24 way cables terminated with a 24-way Centronics connector. Modern implementations can run over Ethernet cables.

There have been further enhancements to the Standards for Programmable Instrumentation (SCPI) and National Instruments "high speed 488". Some interesting historical and technical information can be found here:

<https://www.hp9845.net/9845/tutorials/hpib/>

Full details of the standard can be found here:

[https://webuser.unicas.it/misure/MAQ\\_OLD%20\(VO\)/Dispense/DISP\\_7STANDARD%20IEEE%20488\\_2%201992.pdf](https://webuser.unicas.it/misure/MAQ_OLD%20(VO)/Dispense/DISP_7STANDARD%20IEEE%20488_2%201992.pdf)



## GPIB Addressing

In order to identify addresses on the bus, each device has to have a GPIB address. GPIB addresses range from 0 to 31, there being 32 available addresses in all. Address 0 is often used by the controller. These are known as primary addresses.

In addition to primary addresses, there is also an expanded addressing scheme known as secondary addressing. This adds a second address byte to the primary address. Addresses range from 0 to 31 as for primary addressing but the combination of two address bytes allows a larger number of devices to be addressed.

In order to remote-control an instrument, the controller may send control bytes which also all have values below 32. So how does the instrument differentiate these values?

When GPIB sends an address over the bus, the value of the byte that it actually sends will depend on whether it's a primary address or secondary address. The GPIB bus is not a duplex bus, in other words characters can only be sent in one direction at a time. Therefore, the controller addresses an instrument to either listen or to talk. This is also done by changing the value of the primary address. To address a device to listen, the controller will add 32 (0x20) to the primary address value. To address a device to talk, the controller will add 64 (0x40) to the address value. What this is actually doing is setting bits 5 and 6 of the address byte. When the controller wants to send a secondary address, it adds 96 (0x60) to the address which sets both bits.

In this way the instrument can determine whether it has been addressed to listen, talk or has received a secondary address. This is also why you might see secondary addresses referenced as values between 0-31 or between 96-126. If neither bit is set, then the value is actually below 32 and the instrument interprets this as a control byte.

The AR488 supports both primary and secondary addressing using the `++addr` command. It also provides two additional commands (`++secread` and `++secsend`) to simplify sending and receiving data when using a secondary address.

## 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 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 Commands and Data

All data from the computer is sent to the interface over the USB to Serial. Commands sent to the interface are prefixed with a double plus sign *++* . Anything sent with this prefix is processed by the command parser. Data to be sent directly to the instrument is not prefixed and is sent directly over GPIB to the instrument. Character sequences are terminated with a line ending character, which is usually CR, LF or CRLF depending on the operating system in use and the terminal software settings. The terminators are stripped by the interface before the data is processed. A terminator corresponding to the configuration of the AR488 interface (*see ++eos and ++eoi* commands) is then appended before onward transmission over GPIB to the instrument.

Data returned from the instrument is not processed or parsed with one exception. If EOI only is being used as a terminator, a character may be substituted to indicate a terminator depending on the configuration of the *++eot* and *++eot\_char* commands.

## 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 the sequence '++' precedes a command token. Special care needs to be taken when sending binary data to an instrument, because we do not want control characters to be interpreted as some kind of action. Rather, they need to be treated as ordinary binary data and transmitted as part of the binary data stream.

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 will also likely be necessary to prevent the interface from terminating the binary data sequence with the usual carriage return, newline or both (0D 0A) as this will confuse most instruments and rely on the EOI signal instead. The command *++eos 3* can be used to turn off the transmission of 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.

## Interrupting Transmission of Data

In normal operation, the user issues the ++ command, the instrument responds and data is returned from the instrument over serial. Once the transmission is complete, the next ++ command can be sent. However, under certain conditions when the instrument is addressed to talk, data transmission may continue indefinitely, for example:

- when eos is set to 3 [no terminator character] and the expected termination character is not received from the instrument
- 'read with eoi' has been configured on the interface but the instrument is not configured to assert eoi
- auto mode is enabled

If this happens, the interface should still respond to the a double plus followed by an exclamation mark ++! or a command (e.g. ++auto 0 or ++rst). There may be a slight delay before response but data transmission should stop allowing the interface or instrument configuration to be adjusted.

## Listen-only (LON), Talk-only (TON) and Promiscuous modes

In device mode, the interface supports "listen-only", "talk-only" and promiscuous modes (for further details see the ++lon, ++ton and ++prom commands. Talk-only and listen-only modes are non-addressed modes and do not require a GPIB address to be set. Any GPIB address that has already been set is ignored. 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 way transmission medium. In LON mode, the device is in listen-only mode and will receive any data placed on the bus by a talker in *Talk-Only* Mode. In TON mode the device is in *Talk-Only* mode. It can send data only and cannot receive data from other devices on the bus. Only one talker can exist on the bus, but it is possible to have many listeners.

In Promiscuous mode the interface will ignore addressing and receive any data being sent across the GPIB bus and can be used when there is a controller on the bus. It will ignore command bytes and forward only the received data to the serial port.

LON and TON are standard GPIB modes. Promiscuous mode is a custom option provided by the AR488 interface.

# Configuration

## The Configuration File

Configuration of the AR488 is achieved by 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.

The firmware version is defined at the top of the `AR488_Config.h` file in the format:

```
#define FWVER "AR488 GPIB controller, ver. 0.51.09, 20/06/2022"
```

The `#define` command is a compiler directive instructing it to store the version string, which is contained within the double quotes, in a variable called `FWVER`. This variable is used to present the version information within the program. It should not exceed 47 characters in length and ideally should also not be changed. If required, the string can be overridden using the `++id` command (see the *Command Reference*).

## Board Selection

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:

```

/** UNO and NANO boards */
#elif __AVR_ATmega328P__
/* Board/layout selection */
#define AR488_UNO
//#define AR488_NANO

```

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: ).

**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.**

## Serial Port Configuration

### Hardware Serial Ports

The firmware implements separate ports for data flow and debug messages. These can be enabled activating or de-activating the `DATAPORT_ENABLE` and `DEBUG_ENABLE` flags by adding or removing the preceding comment `/*` character sequence. In order to receive data from instruments and responses to commands, `DATAPORT_ENABLE` must be defined, otherwise the port will be disabled and there will be no output. It is not recommended to comment this flag out.

Here is an example of the serial port configuration section for the data port:

```
/***** Communication port *****/
#define DATAPORT_ENABLE
#ifndef DATAPORT_ENABLE
  // Serial port device
  #define AR_SERIAL_PORT Serial
  // #define AR_SERIAL_SWPORT
  // Set port operating speed
  #define AR_SERIAL_SPEED 115200
  // Enable Bluetooth (HC05) module?
  // #define AR_SERIAL_BT_ENABLE 12 // HC05 enable pin
  // #define AR_SERIAL_BT_NAME "AR488-BT" // Bluetooth device name
  // #define AR_SERIAL_BT_CODE "488488" // Bluetooth pairing code
#endif
```

AR\_SERIAL\_PORT specifies the port name to use.

AR\_SERIAL\_SPEED specifies the baud rate to use.

On most interfaces the primary serial interface is called *Serial*, however interfaces come in various flavours. Boards such as the UNO, Nano and Mega2560 implement a hardware UART chip. The primary port on boards that are based on the 32u4 microcontroller, such as Micro Pro and Leonardo, is implemented in firmware and called a CDC port. In addition, these boards have a hardware port called *Serial1* implemented on the Rx/Tx pins.

The Mega 2560 has 4 hardware serial ports so either of '*Serial*', '*Serial1*', '*Serial2*' or '*Serial3*' should 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 when that particular layout definition is in use. However, it can be used with the *E1* and *E2* definitions.

Since the format for serial port names is standard for both hardware and CDC ports, the Arduino handles this automatically behind the scenes.

## Software Serial Library

A serial port can also be implemented using a library called *SoftwareSerial* which allows a pair of arbitrary GPIO pins to be used to run a serial port. The AR488 supports this option, although ports implemented using this library have a practical speed limitation of 57600 baud. Ports implemented using the *SoftwareSerial* library need additional configuration options to be specified.

To define a SoftwareSerial port, the entry:

```
#define AR_SERIAL_PORT Serial
```

Needs to be commented out. Instead the line:

```
// #define AR_SERIAL_SWPORT
```

Needs to be uncommented by removing the preceding '//'.  
The *SoftwareSerial* port section then needs to be configured by specifying the RX and TX pins as shown in the example below:

```
#if defined(AR_SERIAL_TYPE_SW) || defined(DB_SERIAL_TYPE_SW)
    #define SW_SERIAL_RX_PIN 11
    #define SW_SERIAL_TX_PIN 12
#endif
```

It should be noted only ONE *SoftwareSerial* port can be set per board, and only ONE serial port can be enabled in within the data port section.

## Enabling Bluetooth Support

The AR488 firmware supports Bluetooth using the HC05 module. Since communication between the Arduino and the HC05 module is via serial, it is important that a serial port is configured as indicated above first.

Once that is done, Bluetooth can be enabled by un-commenting and configuring the following 3 lines:

```
//#define AR_SERIAL_BT_ENABLE 12          // HC05 enable pin
//#define AR_SERIAL_BT_NAME "AR488-BT"    // Bluetooth device name
//#define AR_SERIAL_BT_CODE "488488"      // Bluetooth pairing code
```

AR\_SERIAL\_BT\_ENABLE sets the GPIO pin that is connected to the EN(able) pin on the HC05.

AR\_SERIAL\_BT\_NAME sets the device name that is broadcast by the HC05.

AR\_SERIAL\_BT\_CODE sets the Bluetooth pairing code.

More detailed information on pairing and setting up in Windows and Linux can be found in the Bluetooth section.

## Configuring the Debug Serial Port

The Debug port is configured in a similar way to the data port. It can be set to the same port as the data port, or to a separate one, allowing debug messages to be sent to a separate interface. Here is an example of the configuration:

```
//#define DEBUG_ENABLE
#ifdef DEBUG_ENABLE
    // Serial port device
    // #define DB_SERIAL_PORT Serial
    #define DB_SERIAL_SWPORT
    // Set port operating speed
    #define DB_SERIAL_SPEED 57600
#endif
```



DEBUG\_ENABLE enabled or disables the debug port.

DB\_SERIAL\_PORT sets the port name.

DB\_SERIAL\_SWPORT alternatively sets a *SoftwareSerial* port. The Tx and Rx pins need to be set in the *SoftwareSerial* port configuration section. Only ONE *SoftwareSerial* port can be set per board.

DB\_SERIAL\_SPEED sets the baud rate for the port.

To enable this feature, first uncomment *#define DEBUG\_ENABLE*. Set the serial port to the port that will receive the debug messages and configure the baud rate. If using *SoftwareSerial*, comment out DB\_SERIAL\_PORT and un-comment DB\_SERIAL\_SWPORT instead. Next, configure the GPIO pins to be used, for example:

```
#if defined(AR_SERIAL_SWPORT) || defined(DB_SERIAL_SWPORT)
  #define SW_SERIAL_RX_PIN 50
  #define SW_SERIAL_TX_PIN 51
#endif
```

The above example will configure a *SoftwareSerial* port at 57600 baud on GPIO pins 50 and 51. Please note that the maximum advisable speed for a *SoftwareSerial* port is 57600 baud.

The next step is to select the debug message you require to view. Selectively enabling debug levels can be helpful when trying to diagnose a problem. They should not be required or left enabled for normal running of the interface, but only enabled when required for debugging. One or more of the following can be enabled by removing the preceding // comment characters:

```
// Main module
//#define DEBUG_SERIAL_INPUT      // serialIn_h(), parseInput_h()
//#define DEBUG_CMD_PARSER       // getCmd()
//#define DEBUG_SEND_TO_INSTR    // sendToInstrument();
//#define DEBUG_SPOLL            // spoll_h()
//#define DEBUG_DEVICE_ATN       // attnRequired()
//#define DEBUG_IDFUNC           // ID command

// AR488_GPIBbus module
//#define DEBUG_GPIBbus_RECEIVE   // GPIBbus::receiveData(), GPIBbus::readByte()
//#define DEBUG_GPIBbus_SEND      // GPIBbus::sendData()
//#define DEBUG_GPIBbus_CONTROL  // GPIBbus::setControls()
//#define DEBUG_GPIB_COMMANDS    // GPIBbus::sendCDC(), GPIBbus::sendLLO(),
GPIBbus::sendLOC(), GPIBbus::sendGTL(), GPIBbus::sendMSA()
//#define DEBUG_GPIB_ADDRESSING  // GPIBbus::sendMA(), GPIBbus::sendMLA(),
GPIBbus::sendUNT(), GPIBbus::sendUNL()
//#define DEBUG_GPIB_DEVICE      // GPIBbus::unAddressDevice(),
GPIBbus::addressDevice

// GPIB layout module
//#define DEBUG_LAYOUTS

// EEPROM module
//#define DEBUG_EEPROM           // EEPROM

// AR488 Bluetooth module
//#define DEBUG_BLUETOOTH        // bluetooth
```

Provided that `ENABLE_DEBUG` has been set and one of the above options has been selected, debug messages will be sent to the configured debug serial port. This can be the same port as the data port or, where the board provides additional serial ports or where sufficient GPIO pins are available to use *SoftwareSerial*, this can be an alternative serial port. The advantage of sending debug messages to another port is that they will not interfere with normal interface communications. Debug messages can be viewed on the alternative 'debug' port while normal interface operations are in progress on the communications port.

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.

## Serial Handling in Applications and Scripts

When working with programs and scripts (e.g. Python), it should be borne in mind that the Arduino is only 64 bytes in size. Due to the memory constraints of the Arduino, the additional processing buffer provided by the AR488 program is also limited to only 128 bytes. The UART ICs on some boards provide no handshaking between the PC and the Arduino serial port. Although the Arduino can keep up pretty well, the serial input buffer can easily overflow with loss of characters if too much data is passed too quickly. This means that a bit of trial and error may be required when working with scripts to establish whether and how much delay is required between commands. A short delay may sometimes be needed to avoid a buffer overflow. The amount of delay will depend on factors such as the interface hardware being used, the time taken for the instrument to respond, as well as the GPIB speed of the instrument being addressed.

## 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 *****/
/***** vvvvvvvvvvvvvvvvvvvvvvvvvvvvv *****/
#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 "\n\
++addr 9\n\
++auto 2\n\
*RST\n\
:func 'volt:ac'\n\
"\n\
/* 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 "\n\nmacro\n\n/*<-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 "\n\n/*<-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 "\n\n++addr 7\n\n++auto 1\n\n*RST\n\n:func 'volt:ac'\n\n"
```

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*).

## 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.

When `AR488_CUSTOM` is defined, interrupts cannot be used to detect pin states and therefore `USE_INTERRUPTS` will not be defined and interrupts will not be activated. Pin states will be checked on every iteration of `void loop()` instead.

# Building an AR488 GPIB Interface

## Overview

Construction of an Arduino GPIB interface is relatively straightforward and requires a supported single Arduino 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 (also Logic Green LGT8F328P)

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](#)

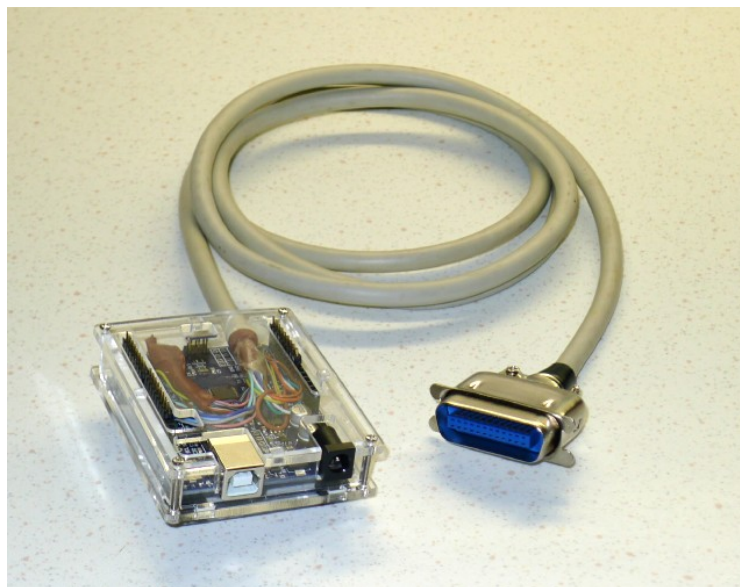
Once the cable has been completed, the sketch should then be downloaded to the Arduino board and the interface should be ready to test.

In order to provide multi-platform compatibility, the AR488 firmware sketch is modular and comes in several files:



<i>Filename:</i>	<i>Purpose:</i>
<b>AR488.ino</b>	This is the main AR488 firmware sketch
<b>AR488_Config.h</b>	This is the configuration file. All configuration options are set here.
<b>AR488_ComPorts.h</b>	Communication ports header file
<b>AR488_ComPorts.cpp</b>	Communication ports implementation
<b>AR488_Eeprom.h</b>	EEPROM functions header file
<b>AR488_Eeprom.cpp</b>	EEPROM functions implementation
<b>AR488_GPIBbus.h</b>	GPIB functions header file
<b>AR488_GPIBbus.cpp</b>	GPIB functions implementation
<b>AR488_Layouts.h</b>	Board layout header file
<b>AR488_Layouts.cpp</b>	Board layout functions implementation

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. All the other files should be automatically loaded by the IDE into separate tabs. Edit the *AR488\_Config.h* file 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 Arduino Interfaces on the Bus

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. However, without any additional buffering (see the following section: *SN7516x GPIB transceiver integrated circuits*), problems can arise when two or more Arduino interfaces 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, may do so in an unpredictable manner and the result of this may be that the interface does not function correctly with other equipment on the GPIB bus. This is a parasitic power phenomenon that is not specific to Arduino microcontrollers only, but can affect 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!/?](https://www.eevblog.com/forum/blog/eevblog-831-power-a-micro-with-no-power-pin!/)

Consequently, unpowered Arduino devices may adversely affect other devices on the GPIB bus. It is therefore essential to either keep Arduino devices powered on, or else 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. Nevertheless, other than when a sole 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 occurs when using Arduinos connected directly without buffering to the the GPIB bus and means that the interface 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 separate 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:

<i>BOD Level:</i>	<i>Bit setting</i>	<i>Threshold</i>
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*.

# USB-to-Serial UART ICs and Handshaking

Most Arduino boards have a serial ports. Some, like the Mega 2560 even have multiple serial ports. In order to communicate over USB with a PC, most boards will also have a Universal Asynchronous Receiver Transmitter (UART) chip to implement the Serial to USB connection. The chip manages the serial transmission of data as well as handshaking between device.

Handshaking is a process for signalling between devices to indicate when they are ready to receive or send. It prevents data from being lost by one device still trying to send while the other is not yet ready to receive. It also provides a means to ensure that buffers are managed so that they are not overrun. Most commonly when computer programs want to communicate over a serial port, they will make use of two signals – Clear-to-Send (CLS) and Ready-to-Send (RTS). Programs will assert RTS to indicate that they are ready to send and then check the CTS response to confirm that the interface is ready to accept data before sending anything. As soon as CTS is de-asserted, the program will stop sending.

Not all UART chips and boards are implemented equally and some do not provide handshaking at all. While handshaking may not be a problem when only a few bytes of data are being sent, it can become an issue where larger volumes of data need to be sent to or from the Arduino over the USB-to-serial connection. The serial buffer on most MCU boards is very small. On the Uno, Nano and Leonardo, it is only 64 bytes in size. As can be appreciated, without any mechanism to control the data throughput, it is likely that this buffer will overflow very quickly. The result is likely to be data loss. The transmission of data then becomes incomplete and garbled.

From the perspective of a desktop computer or laptop operating system it may look like the connection is being made to a serial port, in actuality this device is implemented as a virtual serial port running over a USB connection, with the UART chip on the Arduino handling the incoming connection. By default, many clone Arduino boards do not implement a method to control data flow. Because of this, when sending a larger volume of data such as a file, the computer will keep on sending data to the Arduino regardless of whether the MCU can accept it or not. If the MCU cannot accept the data, it is simply lost. This problem can be further compounded when data has to be then sent onward from the MCU. For example, if the Arduino were sending to a plotter, the MCU may need to wait for the plotter to process the information already in its buffer before sending it further data. However, it has no way of signalling the serial port to halt transmission until the plotter is ready. Once again, data loss results because the computer will continue sending regardless.

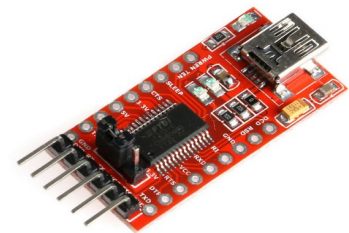
A method to control the flow of data at the serial port is therefore essential. This can be implemented in hardware or in software. One common hardware approach to serial flow control is to use a pair of additional wires or signals to transmit Request to Send (RTS) and Clear to Send (CTS) signals which are defined in the RS232 protocol. UART chips often implement these signals and present them to the PC virtually as part of the virtual serial port implementation.

Alternatively, a software method known as XON/XOFF can be used. XON/XOFF flow control. XON/XOFF does not require additional wires but uses ASCII characters DC1 (decimal 17, hex 0x11) to resume transmission and DC3 (decimal 19, hex 0x13) to pause transmission. This can work fine for text data which does not include those special characters, but might be a problem when transmitting binary data. This is because binary data can include any byte of a value between 0

and 255, which would include the characters just mentioned. For this reason, XON/XOFF can only reliably be used when sending text data. The advantage of the hardware RTS/CTS method is that it can be used with both text and binary data. Whichever method is used, each party on the serial line can exchange signals to pause and restart the transmission. This is known as a handshaking protocol. For this reason, XON/XOFF is very rarely used nowadays.

When serial is implemented over USB, no additional wires are used as the serial port is implemented virtually over the USB protocol, however in order to use RTS/CTS signals, these signals must be accessible somewhere. Unfortunately, by default, many alternative Arduino boards do not support serial handshaking using RTS/CTS signalling by default. Commonly used clone boards have on board a serial UART IC such as the CH340 or the Holtek HT42B534 which exposes the RTS and CTS pins, but these pins are usually not connected anywhere. It is, of course possible that the CH340G could be connected to GPIO pins in order to list16 pten to the RTS signal from the remote computer and send it a CTS signal. On the Holtek chip, the CTS pin does not appear to be functional and remains asserted continuously.

It is also possible to obtain a USB-to-Serial breakout boards, for example, the most common one of these is the FT232RL USB2.0 to TTL Serial Adapter breakout board, but there are others such as one based on the CP2102 or Prolific UART chips.



While serial communications are problematic when using the CH340 or the Holtek HT42B534 chips, they generally work well with FTDI, CP2102 and Prolific chips.

Thankfully Holtek chips are not found very often, but they might be present on some Logic Green LGT8F328P boards. While the GPIB interface will run on the LGTF328P board, serial handshaking is not possible with the Holtek UART chip.

## CH340G Serial UART

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.

## **FTDI UART (FT232RL)**

Arduino boards that contain the ATMEGA MEGA 16u2 chip are recognized as FTDI serial devices. The functionality provided by the FTDI chip is embedded within this 16u2 chip. This chip does not expose the RTS/CTS signals so the CH340 style modification is not possible nor is actually required in order for it to be able 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.

For some reason, Arduino boards running the 16u2 chip are not recognized by the EZGPIB program.

## **CP2102 UART**

Boards such as the ESP32 that have the CP2102 chip on board should work fine over USB, although, of course, it would no doubt be desirable to operate over WiFi.



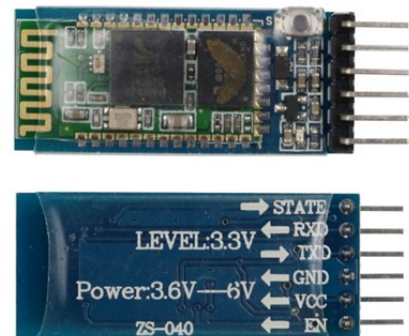
# AR488 Bluetooth Support

## The HC05 and HC06 Modules



Bluetooth is a common short range wireless connection technology for connecting devices and sending data. Bluetooth devices can typically communicate over distances not exceeding 10 metres, although some devices can communicate over greater distances. In order to communicate with each other, Bluetooth devices must be paired. Once paired, the device facilities can be discovered and a connection established.

Bluetooth transceiver modules such as the HC05 and HC06 can be connected to an Arduino quite easily and manually configured to provide a wireless connection in place of the USB cable. These boards come as either the transceiver board itself with soldered edge connectors only and requiring a DC 3.3V supply, or, mounted on an “adapter” or “breakout” board (as shown opposite) with external pin connectors at one end and which can be supplied directly with 5V.

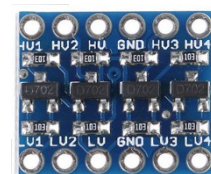


The AR488 supports auto-configuration of the HC05 Bluetooth breakout board for Arduino. This board can be just connected to the serial port of the Arduino and the configuration set in AR488\_Config.h file. The HC05 board will be then configured automatically on start-up. Ideally the board would be connected to the RX and TX pins of a second serial port, e.g. Serial1 on the 32u4 or Serial1/Serial3 on the Mega 2560. The HC06 module does not support master mode, so auto-configuration is not possible with the HC06. However, either the HC05 and HC06 breakout boards or modules can be configured manually if required and used with the AR488.

The HC05 and HC06 modules have a very similar appearance, but the HC05 can be identified by the fact that it has six pins, while the HC06 has only four. There is no EN or STATUS pin on the HC06.

Unlike most Arduino boards, which are supplied with 5V DC as is common to all USB devices, the HC05 module requires a 3.3V supply. “adapter” or “breakout” board provides a 3.3V regulator for this purpose so it can be supplied with 5V, however, the serial input and as well as the enable pin still operate at a 3.3V signalling level as

indicated by the ‘LEVEL:3.3V’ marked on the board. Applying 5V to these pins can damage the module and while some tutorials do show these connected directly to the RX/TX pins on the Arduino, this is not recommended and ideally, a level shifter should be used. This can be made quite easily but ready made boards such as the one shown opposite can also be purchased. Such boards usually have 4 “channels” (HV1-HV4 and corresponding LV1-LV4) which is sufficient for our purposes. There are 4 transistors across the centre and 6 soldered connectors along each edge. They board is usually supplied with a row of pins which can be soldered to the connectors if required or wires can be soldered to the board directly. The HV side is connected to the Arduino and the LV side to the HC05.



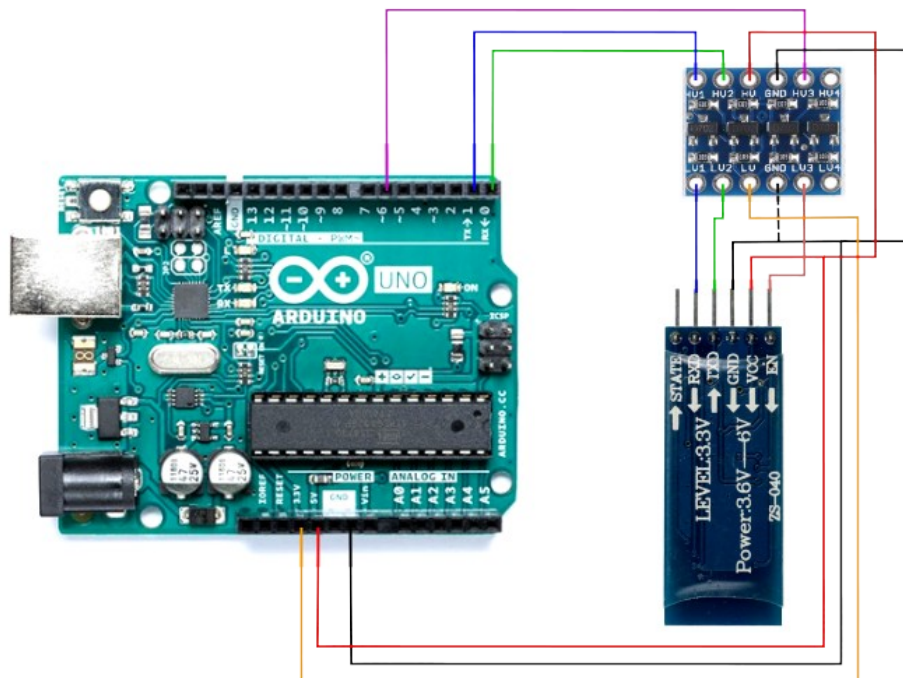
The  
output



The level shifter board is easily supplied from the 5V and 3.3V pins on the Arduino with the 5V pin connected to HV and 3.3V pin connected to LV. The GND connector must be connected to the Arduino GND pin as well as the HC05 GND pin. Either one or both GND connectors on the level shifter can be used as they are linked together. The Arduino TX, RX and enable pin connections are made to one of the HV side channels and connections to the HC05 Bluetooth board are made to the corresponding LV side channel. It does not matter which actual channel numbers are used for which function so long as TX on the Arduino connects with RX on the HC05, RX on the Arduino to the TX on the HC05 and EN on the HC05 connects to the GPIO pin configured for this purpose on the Arduino, all connections being made via correspondingly numbered HV/LV channel on the level shifter.

The HC05 breakout board requires 5 connections in total including 5V DC power to VCC, GND, TXD, RXD and EN or "Enable" that enables the Arduino master mode required for configuration. In the example shown below, the EN pin is connected via the level shifter to pin 6 on the Uno, but any spare Arduino GPIO pin can be used. The pin marked STATE is left disconnected. Factory set serial communication speeds also seem to vary and while some modules operate at 38400 baud, others seem to require a 9600 baud connection. The AR488 Bluetooth module will automatically detect the default speed and set it to the configured speed.

The Bluetooth HC05 transceiver breakout board requires three channels. The diagram below is an example of how an Arduino Uno might be wired to the HC05 module:



*Diagram of the Arduino to HC05 connections*

Clones of the Pro Micro board do not have a 3.3V regulator on board, so a 3.3V supply will have to be provided separately.

## Enabling Bluetooth on the AR488

The Bluetooth feature is disabled by default, but can be enabled by removing the comment characters `//` preceding the definition for `AR_SERIAL_BT_ENABLE` within the data port configuration section within the `AR488_Config.h` file:

```
//#define AR_SERIAL_BT_ENABLE 12          // HC05 enable pin
//#define AR_SERIAL_BT_NAME "AR488-BT"    // Bluetooth device name
//#define AR_SERIAL_BT_CODE "488488"      // Bluetooth pairing code
```

so that it becomes:

```
#define AR_SERIAL_BT_ENABLE 12          // HC05 enable pin
#define AR_SERIAL_BT_NAME "AR488-BT"    // Bluetooth device name
#define AR_SERIAL_BT_CODE "488488"      // Bluetooth pairing code
```

This will activate the Bluetooth auto-configuration. Set the pin number to the number of the GPIO pin that will be used to control the EN pin on the HC05.

The default baud rate can be set to anything that the Arduino and the HC05 board will support. Please note that there are no double quotes around this parameter.

The default pairing device name is *AR488-BT*, but this can be changed to anything desired by the following line:

```
#define AR_SERIAL_BT_NAME "AR488-BT"
```

where *AR488-BT* is replaced by whatever string is desired.

The default pairing code is *488488*, but this can be changed by modifying the number in quotes after the `AR_BT_CODE` keyword:

```
#define AR_SERIAL_BT_CODE "488488"
```

The code must be enclosed in double quotes and must be at least 6 digits long.

The diagram above shows a Uno wired up to a HC05 connector. It should be noted that the Uno and Nano have only one UART and the 328p has only one set of Tx/Rx pins. Since the serial protocol was designed for 1:1 connections, on these Arduino boards, USB and Bluetooth should not be used at the same time. The sketch must be uploaded to the Arduino prior to the HC05 being connected to the Tx/Rx pins, otherwise the upload will fail. Once the sketch has been uploaded, the module can then be connected and both Arduino and HC05 module powered up. This is not a problem where an Arduino has additional serial ports available such as the 32u4 and the Mega 2560. The HC05 can be connected to a secondary port and the USB port used to program as normal.

## Power-up and Detection

Once the sketch has been configured and uploaded to the board and the HC05 Bluetooth module connected up, the AR488 can be powered up. On start-up it will automatically detect the baud rate of the HC05 module and configure it. By default, the AR488 will appear as a Bluetooth device called *AR488-BT*, but otherwise should appear with the configured name. On Windows, it may appear as "Other device" until the screen is refreshed.

The first time that the HC05 board is used, the LED on the HC05 board will initially blink slowly for a few seconds. The internal LED on the Arduino should then blink twice to indicate that the baud rate has been successfully detected, and following that, 3 more times to indicate that configuration has been successful. A few moments after this, the LED on the HC05 should begin flashing rapidly to indicate that it has switched to slave mode and is ready for pairing.

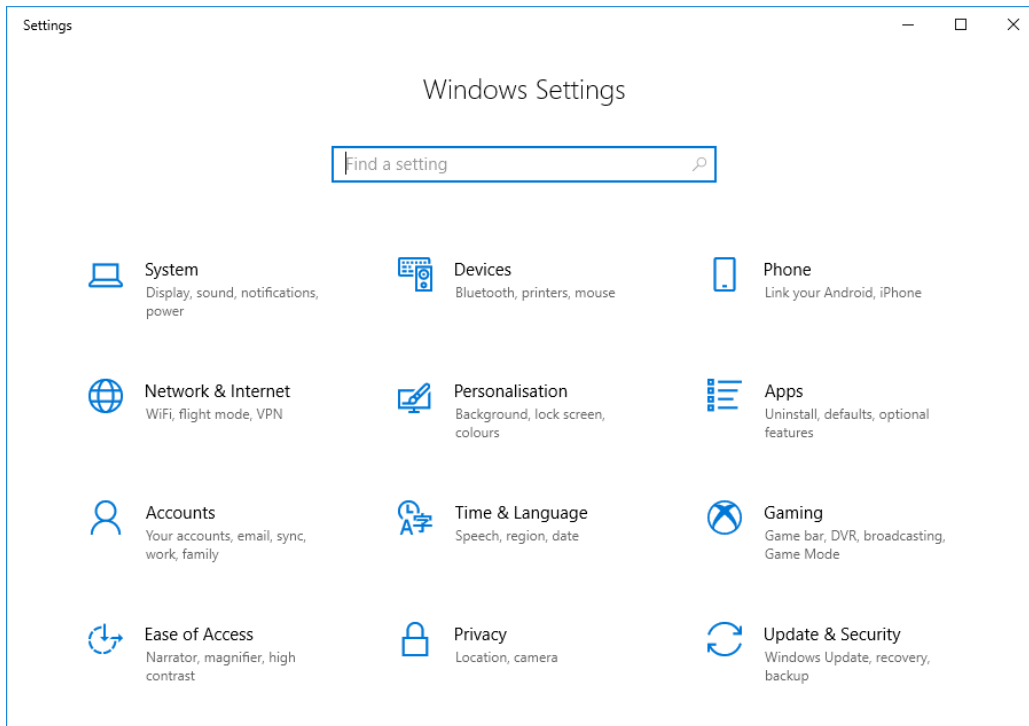
On subsequent power-up, the LED on the HC05 board will initially blink slowly for a few seconds. The Arduino internal LED will blink twice to indicate successful auto-detection of the baud rate, and then once to indicate that the configuration has not changed. The LED on the HC05 will then blink rapidly to indicate that it has switched to slave mode and the module is ready for pairing. If any of the Bluetooth configuration parameters are changed in the sketch, the AR488 will perform the auto-configuration process again. If nothing has changed, the auto-configuration process is skipped and the HC05 board just goes into slave mode.

Once paired, the LED on the HC05 board will blink twice every few seconds.

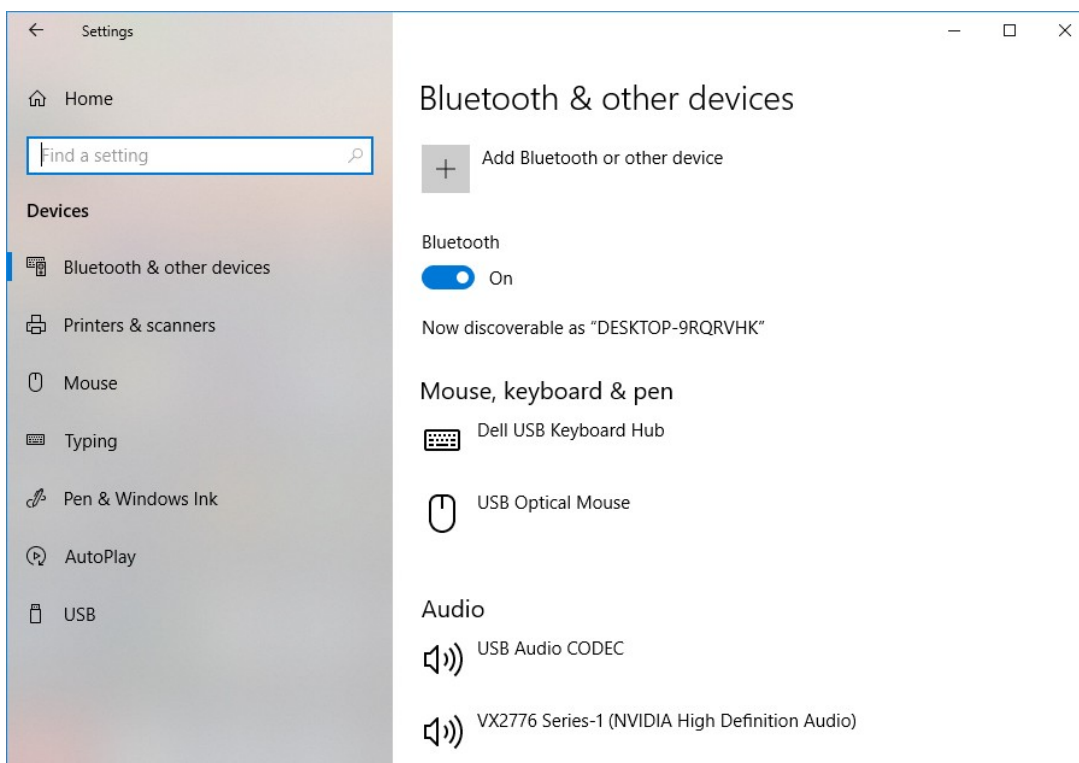
# Pairing the AR488 Bluetooth Interface

## Windows 10

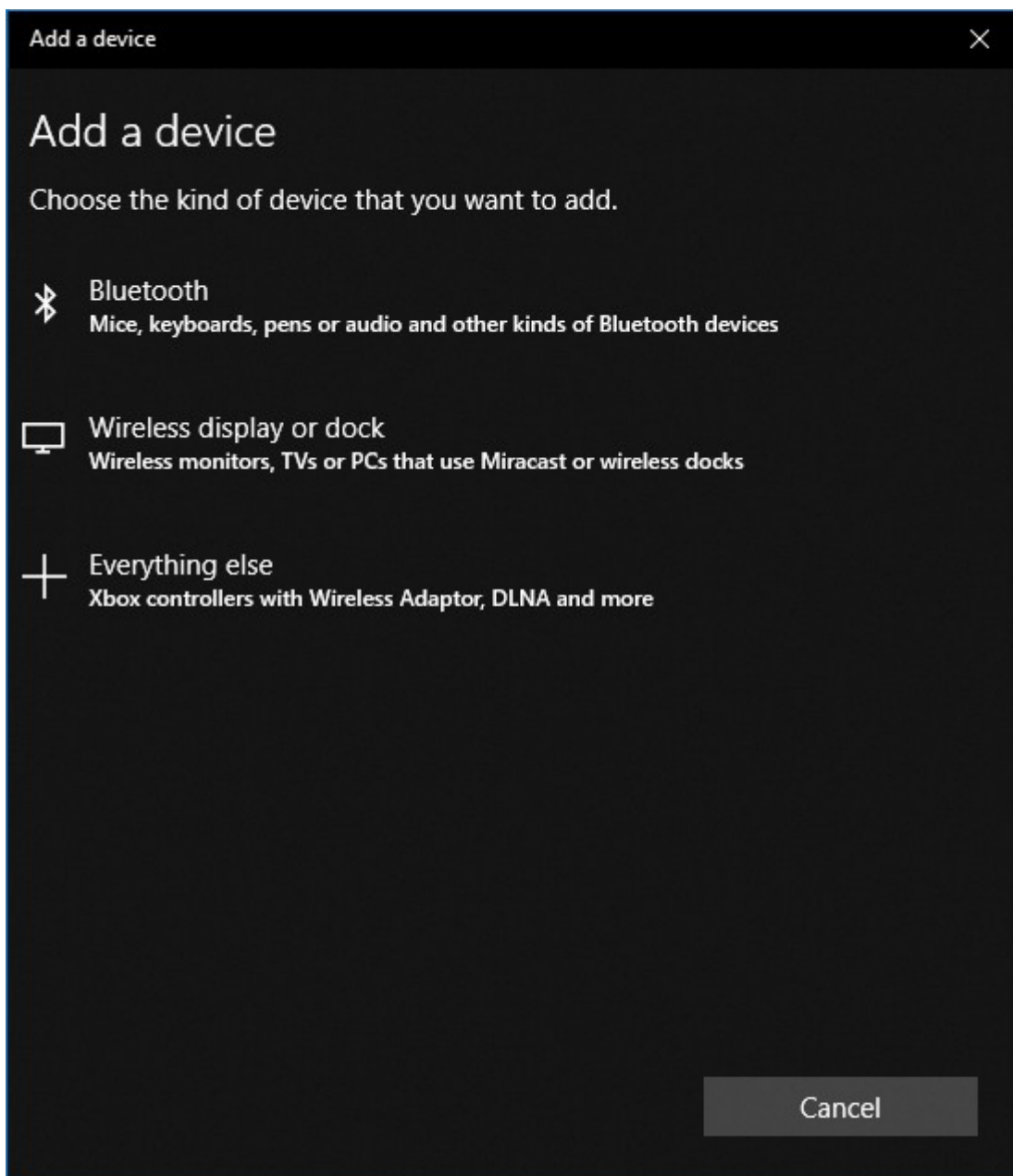
Go to *Windows Settings* (the cog icon on the left of the windows menu) and select *Devices*:



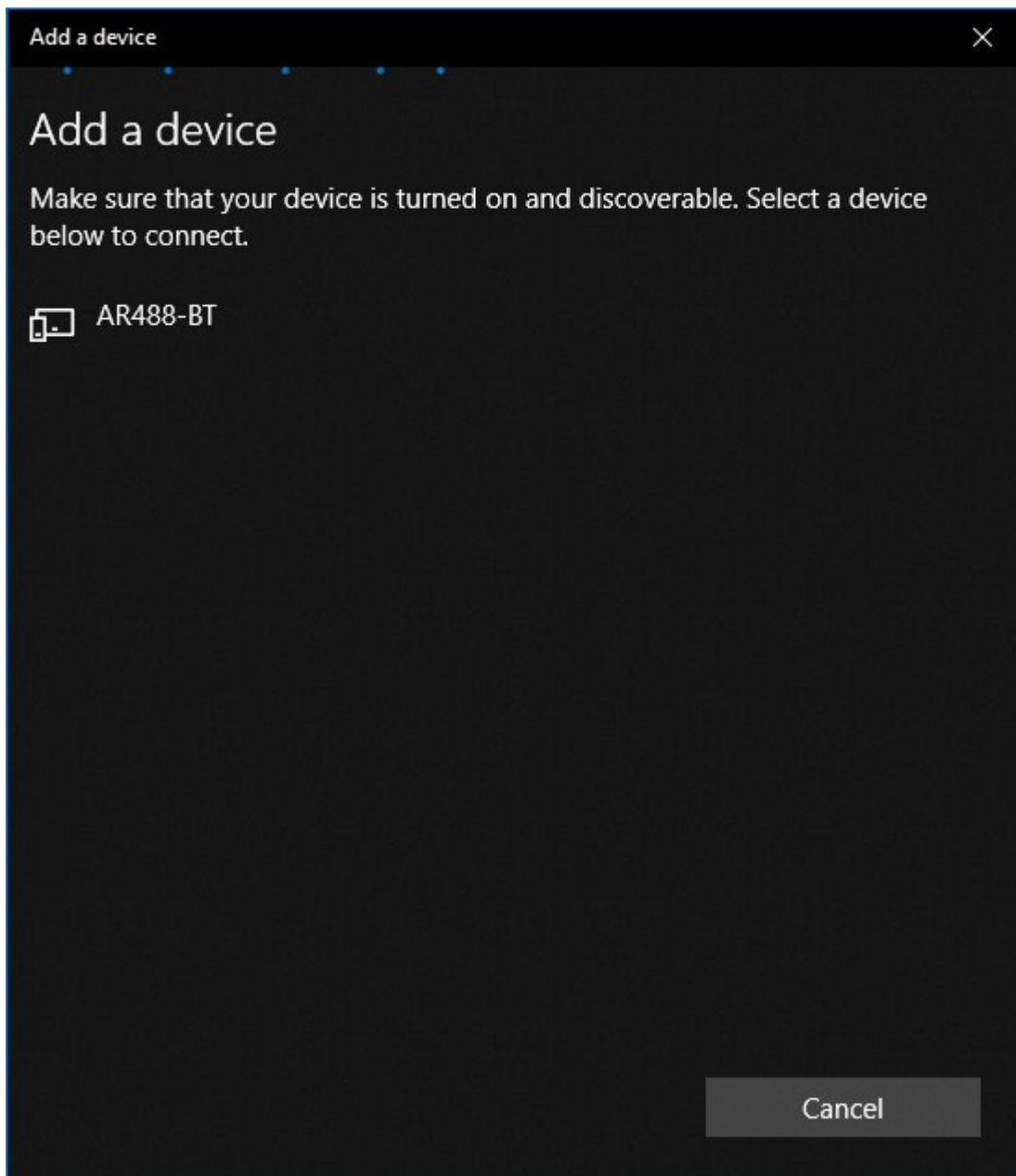
This opens the *Bluetooth & other devices* dialogue:



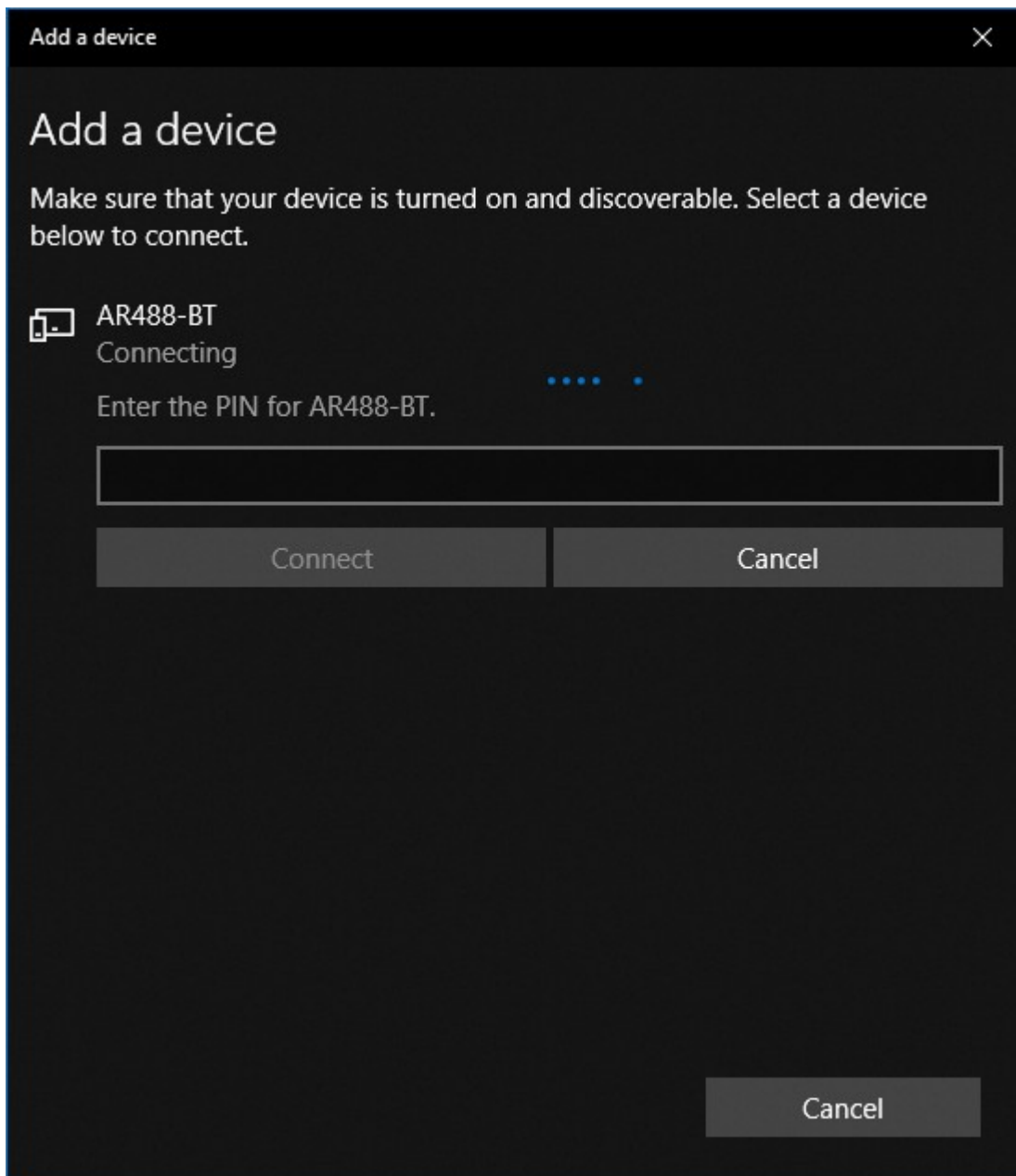
Make sure that Bluetooth is turned on. Click *Add Bluetooth or other device*. This will open another Window:



Click *Bluetooth*. Windows should now look for devices. It should momentarily show the AR488-BT device as *unknown device*, but this should quickly change to *AR488-BT*.



Click the AR488-BT device. After a few moments prompt will appear requesting the pin.



Enter the pin and click *Connect*. If it times out, the dialogue may show *“Try connecting your device again”*. Click on the device again to try once more. Once successful, this will be clearly confirmed:

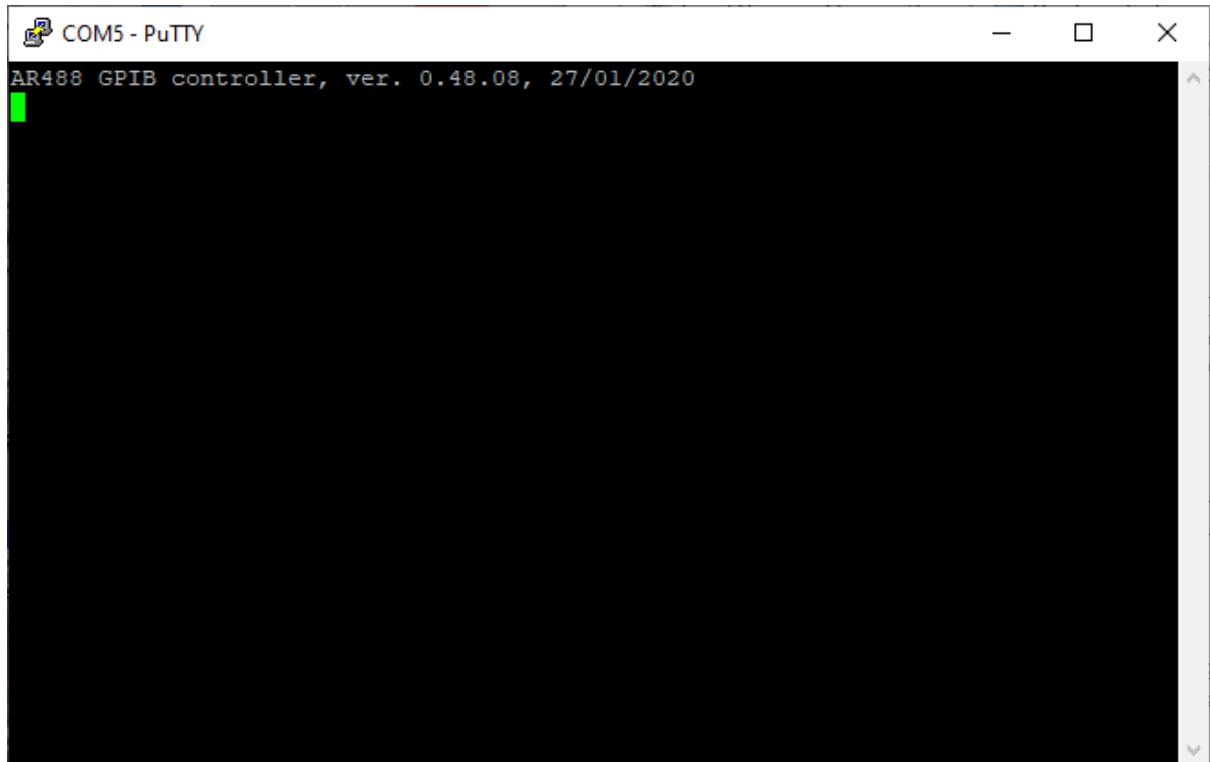
The device status should now be shown as *Paired*. Click the *Done* button to close the device configuration window.

Back on the *Bluetooth & other devices* window, scroll down. The AR488-BT device should be listed under *Other Devices*. The window can now be closed.

Right-click *This PC* and select *Manage*. Click on *Device Manager* and expand the *Port (COM & LPT)* section. The device should be shown as *Standard Serial over Bluetooth link (COMx)* where COMx will be the assigned COM port.

Open a terminal session to the assigned COM port and test communication with the device:





The interface should respond as normal.

## Linux

These instructions should work on most Linux distributions. Depending on your distro of Linux, the *bluez* or *bluez5* Bluetooth tools package may already be installed by default, or you may be able to download and install it from the distribution repository using apt or other package manager. Otherwise it will need to be compiled from source.

First, make sure that your Bluetooth dongle or built-in device is working correctly on your computer or laptop.

Make sure that Bluetooth is turned on and your system can identify your Bluetooth hardware. Once you have confirmed that your Bluetooth hardware is working, open a terminal and at the command prompt type:

```
% bluetoothctl
```

This should list any known Bluetooth devices, show *Agent registered* at the end of the list, and a [bluetooth]# prompt:

```
$ bluetoothctl
[NEW] Controller 00:80:98:94:AB:7E agabus [default]
[NEW] Device 78:3A:84:93:BC:B9 iPad
[NEW] Device 10:2F:6B:BD:49:F1 N930 phone
Agent registered
[bluetooth]#
```

If your device is not listed, then at the prompt type:

```
scan on
```

This should initiate a scan for new devices. In a few seconds any new devices should be listed:

```
[bluetooth]# scan on
Discovery started
[CHG] Controller 00:80:98:94:AB:7E Discovering: yes
[NEW] Device 98:D3:31:F9:4E:6D AR488-BT
```

The AR488-BT device should be detected and its mac address listed. To pair the device type:

```
[bluetooth]# pair 98:D3:31:F9:4E:6D
```

where the MAC address is the address of YOUR Bluetooth device. The *bluetoothctl* utility should respond with something like:

```
[bluetooth]# pair 98:D3:31:F9:4E:6D
Attempting to pair with 98:D3:31:F9:4E:6D
[CHG] Device 98:D3:31:F9:4E:6D Connected: yes
Request PIN code
[AR488lm[agent] Enter PIN code:
```

Enter the pairing code configured on the HC05 device. The default pairing code is 488488, but if a custom six digit code has been configured then that should be provided instead. The *bluetoothctl* utility will now attempt to pair with the device. If successful, the output should be something like:

```
[AR481m[agent] Enter PIN code: 488488
[CHG] Device 98:D3:31:F9:4E:6D UUIDs: 00001101-0000-1000-8000-00805f9b34fb
[CHG] Device 98:D3:31:F9:4E:6D ServicesResolved: yes
[CHG] Device 98:D3:31:F9:4E:6D Paired: yes
Pairing successful
[AR488-BT]#
```

At this point the device is paired, but there is no serial port associated with it. Another tool called *rfcomm* can be used to associate a serial port with the Bluetooth device. First exit the *bluetoothctl* utility by typing:

```
[bluetooth]# exit
```

The utility will respond with the following and return to the system prompt:

```
Agent unregistered
[DEL] Controller 00:80:98:94:AB:7E agabus [default]
$
```

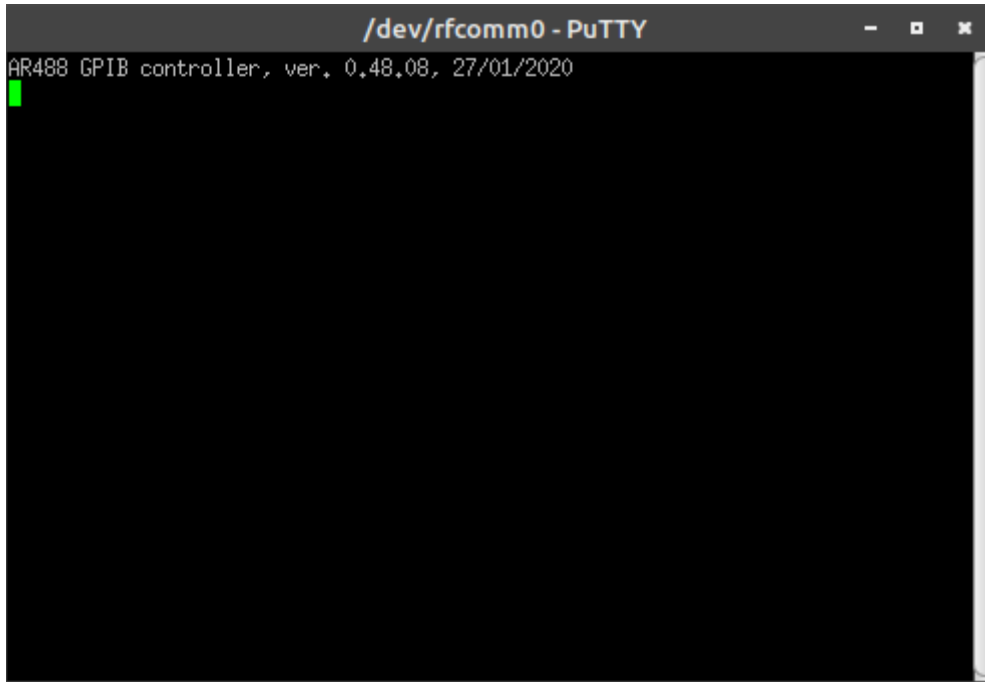
Type the following to associate a serial port with the Bluetooth device:

```
$ sudo rfcomm bind 0 98:D3:31:F9:4E:6D 1
```

You may be prompted for the sudo password. After this has command has completed, a serial port called */dev/rfcomm0* will be created. To confirm that the port binding has been established can be done with the following command:

```
$ ls /dev/rfcomm*
/dev/rfcomm0
```

The repose confirms that the port is now available, so we can open a TTY terminal such as PuTTY and try to establish a connection to it:



The terminal should connect and the device should respond in the usual manner.

To unbind the /dev/rfcomm0 port at the end of the session, from the system \$ prompt type:

```
$ sudo rfcomm unbind 0 98:D3:31:F9:4E:6D 1
```

## Troubleshooting Information for Linux

A noteworthy point is that *bluetoothctl* is launched while a connected session is in progress, then the prompt will show the name of the currently connected device:

```
[AR488-BT] #
```

To get further information about a device in *bluetoothctl*, when at the [bluetooth]# prompt type:

```
info 98:D3:31:F9:4E:6D
```

where the MAC address is the address of the device you would like further information on. The program should respond with something like this:

```
[bluetooth]# info 98:D3:31:F9:4E:6D
Device 98:D3:31:F9:4E:6D (public)
    Name: AR488-BT
    Alias: AR488-BT
    Class: 0x00001f00
    Paired: yes
    Trusted: no
    Blocked: no
    Connected: yes
    LegacyPairing: yes
    UUID: Serial Port (00001101-0000-1000-8000-00805f9b34fb)
[bluetooth]#
```

You can also list all the devices that have been paired with:

```
[bluetooth]# paired-devices
Device 98:D3:31:F9:4E:6D AR488-BT
Device 10:2F:6B:BD:49:F1 N930 phone
[bluetooth]#
```

To remove a paired device type:

```
[bluetooth]# remove 98:D3:31:F9:4E:6D
[DEL] Device 98:D3:31:F9:4E:6D AR488-BT
Device has been removed
[bluetooth]#
```

If *bluetoothctl* responds with:

```
Failed to start discovery: org.bluez.Error.NotReady
```

This may indicate that the device is not powered on or blocked. Try this first:

```
[bluetooth]# power on
```

If this reports:

```
Failed to set power on: org.bluez.Error.Blocked
```

Then exit *bluetoothctl* and run this command to check for blocked devices:

```
$ rfkill list
1: hci0: Bluetooth
    Soft blocked: yes
    Hard blocked: no
```

If, as shown above, Bluetooth is 'Soft blocked' then it might be possible to unblock it with:

```
rfkill unblock all
```

Now run *bluetoothctl* again.

Otherwise check that device drivers were properly loaded.

If a device is 'Hard blocked' then there is a hardware problem, e.g. a toggle switch may be in the wrong position, faulty cable, power off etc. or the device may be disabled in BIOS.

# Working With Third Party Software

## EZGPIB and the Arduino Bootloader

EZGPIB is a tool for easy GPIB, TCP or serial based data acquisition in conjunction with a Prologix GPIB to USB interface. It works with DLL based plugin cards & USB controllers as well as the latest Prologix LAN GPIB interface. It provides an IDE programming environment that can be used to work with GPIB devices and supports the Prologix protocol.

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. There are a couple of possible solutions:

### Solution 1 - Capacitor

One option 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 10 $\mu$ F 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.

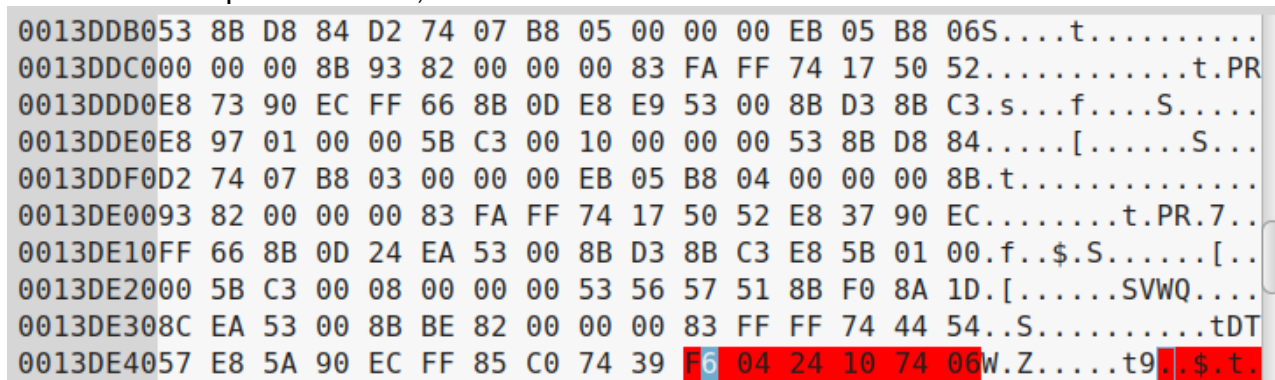
## Solution 2 – Hacking the EZGPIB Binary

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

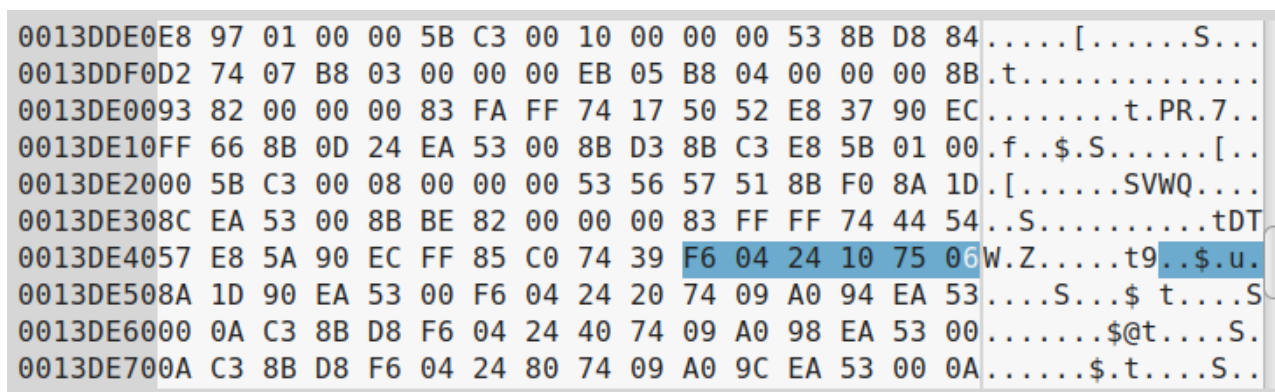
Note, that these instructions can also be found on <http://www.dalton.ax/gpib/>, but show the sequence as F6 02 24 10 74 06. I found the sequence to be as above. I'm not sure whether this is an error or because my binary is different from the one that the author was working with. If you can't find the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



0013DDB053 8B D8 84 D2 74 07 B8 05 00 00 00 EB 05 B8 06S....t.....  
0013DDC000 00 00 8B 93 82 00 00 00 83 FA FF 74 17 50 52.....t.PR  
0013DDD0E8 73 90 EC FF 66 8B 0D E8 E9 53 00 8B D3 8B C3.s...f....S....  
0013DDE0E8 97 01 00 00 5B C3 00 10 00 00 00 53 8B D8 84.....[.....S...  
0013DDF0D2 74 07 B8 03 00 00 00 EB 05 B8 04 00 00 00 8B.t.....  
0013DE0093 82 00 00 00 83 FA FF 74 17 50 52 E8 37 90 EC.....t.PR.7..  
0013DE10FF 66 8B 0D 24 EA 53 00 8B D3 8B C3 E8 5B 01 00.f..\$.S.....[..  
0013DE2000 5B C3 00 08 00 00 00 53 56 57 51 8B F0 8A 1D.[.....SVWQ....  
0013DE308C EA 53 00 8B BE 82 00 00 00 83 FF FF 74 44 54..S.....tDT  
0013DE4057 E8 5A 90 EC FF 85 C0 74 39 F6 04 24 10 74 06W.Z....t9..\$.t.

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



0013DDE0E8 97 01 00 00 5B C3 00 10 00 00 00 53 8B D8 84.....[.....S...  
0013DDF0D2 74 07 B8 03 00 00 00 EB 05 B8 04 00 00 00 8B.t.....  
0013DE0093 82 00 00 00 83 FA FF 74 17 50 52 E8 37 90 EC.....t.PR.7..  
0013DE10FF 66 8B 0D 24 EA 53 00 8B D3 8B C3 E8 5B 01 00.f..\$.S.....[..  
0013DE2000 5B C3 00 08 00 00 00 53 56 57 51 8B F0 8A 1D.[.....SVWQ....  
0013DE308C EA 53 00 8B BE 82 00 00 00 83 FF FF 74 44 54..S.....tDT  
0013DE4057 E8 5A 90 EC FF 85 C0 74 39 F6 04 24 10 75 06W.Z....t9..\$.u.  
0013DE508A 1D 90 EA 53 00 F6 04 24 20 74 09 A0 94 EA 53....S...\$ t....S  
0013DE6000 0A C3 8B D8 F6 04 24 40 74 09 A0 98 EA 53 00.....\$@t....S.  
0013DE700A C3 8B D8 F6 04 24 80 74 09 A0 9C EA 53 00 0A.....\$.t....S..

Now look for sequence:

24 04 10 0F 95



```

00125930E8 7B 01 00 00 84 C0 74 1E 80 7B 43 00 74 0D 6A.{.....t..{C.t.j
0012594003 8B 43 34 50 E8 FE 14 EE FF EB 0B 6A 04 8B 43..C4P.....j..C
0012595034 50 E8 F1 14 EE FF 5B C3 8D 40 00 53 8B D8 884P.....[..@.S...
0012596053 45 8B C3 E8 47 01 00 00 84 C0 74 1E 80 7B 45SE...G.....t..{E
0012597000 74 0D 6A 08 8B 43 34 50 E8 CA 14 EE FF EB 0B.t.j..C4P.....
001259806A 09 8B 43 34 50 E8 BD 14 EE FF 5B C3 8D 40 00j..C4P.....[..@.
0012599053 56 83 C4 F8 8B F0 C6 04 24 00 33 DB 8B C6 E8SV.....$.3....
001259A00C 01 00 00 84 C0 74 21 8D 44 24 04 50 8B 46 34.....t!.D$.P.F4
001259B050 E8 EA 14 EE FF 83 F8 01 1B DB 43 84 DB 74 09P.....C..t.
001259C0F6 44 24 04 10 0F 95 04 24 88 5E 30 8A 04 24 59.D$.....$.^0..$Y

```

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

24 04 10 0F **94**

```

0012595034 50 E8 F1 14 EE FF 5B C3 8D 40 00 53 8B D8 884P.....[..@.S...
0012596053 45 8B C3 E8 47 01 00 00 84 C0 74 1E 80 7B 45SE...G.....t..{E
0012597000 74 0D 6A 08 8B 43 34 50 E8 CA 14 EE FF EB 0B.t.j..C4P.....
001259806A 09 8B 43 34 50 E8 BD 14 EE FF 5B C3 8D 40 00j..C4P.....[..@.
0012599053 56 83 C4 F8 8B F0 C6 04 24 00 33 DB 8B C6 E8SV.....$.3....
001259A00C 01 00 00 84 C0 74 21 8D 44 24 04 50 8B 46 34.....t!.D$.P.F4
001259B050 E8 EA 14 EE FF 83 F8 01 1B DB 43 84 DB 74 09P.....C..t.
001259C0F6 44 24 04 10 0F 94 04 24 88 5E 30 8A 04 24 59.D$.....$.^0..$Y
001259D05A 5E 5B C3 53 56 83 C4 F8 8B F0 C6 04 24 00 33Z^[.SV.....$.3
001259E0DB 8B C6 E8 C8 00 00 00 84 C0 74 21 8D 44 24 04.....t!.D$.

```

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

## The KE5FX Toolkit

KE5FX by John Miles provides testing tools including a plotter emulator and Prologix configuration utility and can be used with various instruments that support GPIB.

The Prologix GPIB Configurator program that is part of the KE5FX GPIB tools package looks for a specific character sequence in the string returned by the `++ver` command in order to identify a Prologix interface. The response to the `++ver` command must contain the sequence `'version 6'` somewhere in the returned version string.

On the AR488, the version string displayed with the `++ver` command can be set to anything convenient by using the following example commands:

```
++id verstr AR488 GPIB-USB version 6
++savecfg
```

This will set the version string to `'AR488 GPIB-USB version 6'` and should be enough to get the interface recognized by the Prologix GPIB Configurator program. The version number does not actually require a `'.1'` suffix, but:

```
++id verstr AR488 GPIB-USB version 6.1
```

would work just as well. The `++savecfg` command saves the setting to EEPROM. This is important since the value of the version string that has been set needs to survive a reset. That the string has been saved can be confirmed by resetting or power cycling the adapter and using the command:

```
++id verstr
```

This should display the version string entered using the previous command.

Alternatively, the version string could be set by changing the value of the `FWVER` variable set using a `#define` statement near the beginning of the `AR488_Config.h` file. This will set the default version string in the compiled program which will be stored along with the code in program memory and will not require EEPROM storage.

# Command Reference

The controller identifies commands by prefixing them with a double plus ++ character sequence. The following sequence of characters is an interface command. When the ++ is omitted, the sequence of accepted characters is assumed to be destined for and sent directly to the connected instrument.

The aim of the project was to support the Prologix command set and operate in a manner that is fully compatible with it. Therefore, with the exception of the ++savecfg command, where command keywords correspond to a Prologix equivalent, they will operate in the same way and be fully compatible with the Prologix GPIB-USB controller. However, some commands may have been enhanced with additional functionality.

In addition, in order to enhance its features and operation, the AR488 controller also implements a number of custom commands. The command reference that follows will clearly indicate which commands are standard and which are additional custom commands as well as any enhancements to the original Prologix command set.

## +addr

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller, device

*Description:* Used to set or query the currently set GPIB address.  
In controller mode, the address refers to the primary GPIB address and optionally the secondary address of the instrument that the operator wishes to communicate with.  
In device mode, the address represents the address assigned to this interface which is now acting as a device.  
By default, the address of the controller is 0.  
When issued without an address, the command returns the currently set primary and optional secondary address.

*Syntax:* ++addr [ pri [,sec] ]

*Comments:* pri is a GPIB primary address between 1 (0x01) and 30 (0x1E)  
sec is a GPIB secondary address between 96 (0x60) and 126 (0x7E)  
sec can also be a number between 0 (0x00) and 31 (0x1F) in which case it will automatically get converted to a range between 0x60 and 0x7F by the addition of 0x60 to the value.

## **++allspoll**

<i>Prologix compatible:</i>	no
<i>Enhanced:</i>	n/a
<i>Modes:</i>	controller
<i>Description:</i>	Alias equivalent to ++spoll all. See ++spoll for further details.
<i>Syntax:</i>	++allspoll
<i>Comments:</i>	

## **++auto**

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller

*Description:* 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.

When set to zero, auto is disabled.

When set to 1, the controller will automatically attempt to read a response from the instrument after any instrument command or, in fact, when 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 *++* has been sent to the instrument, but only if that sequence is a query command that ends with the ? character, such as, for example *\*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*.

When called without a parameter, the command returns the value of the current setting for auto.

*Syntax:* ++auto [0|1|2|3]

*Comments:* 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**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller
<i>Description:</i>	This command sends a Selected Device Clear (SDC) command to the currently addressed instrument. Details of how the instrument should respond may be found in the instrument manual. Typically the instrument may perform a reset, but other behaviours are possible.
<i>Syntax:</i>	++clr
<i>Comments:</i>	

## **++dcl**

<i>Prologix compatible:</i>	no
<i>Enhanced:</i>	n/a
<i>Modes:</i>	controller
<i>Description:</i>	Send Device Clear (DCL) to all devices on the GPIB bus.
<i>Syntax:</i>	++dcl
<i>Comments:</i>	

## ++default

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller, device

*Description:* 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

*Syntax:* ++default

*Comments:* After using the *++default* command and configuring new settings, the *++savecfg* command should be used to store the new settings in EEPROM\*. Otherwise, the previously stored configuration will be re-loaded from non-volatile memory the next time that the interface is powered up. 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.

## **++eoi**

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller, device

*Description:* 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 signal is also used in conjunction with ATN to initiate a parallel poll, however, the ++eoi command has no bearing on that activity. When issued without a parameter, a value corresponding to the current status of the EOI feature will be returned.

*Syntax:* ++eoi [0|1]

*Comments:* 0 disables and 1 enables asserting of the EOI signal to indicate the last character sent.



## **++eor**

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* 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:

<i>Option</i>	<i>Sequence</i>	<i>Hex</i>
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 returned 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.

*Syntax:* ++eor[0-9]

*Comments:*

## **++eos**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller, device
<i>Description:</i>	<p>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 <i>from</i> the instrument.</p> <p>When issued without a parameter, the command will return the current configuration</p>
<i>Syntax:</i>	++eos [0 1 2 3]
<i>Comments:</i>	0=CR+LF, 1=CR, 2=LF, 3=none

## **++eot\_enable**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller, device
<i>Description:</i>	<p>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.</p> <p>When issued without a parameter, the command will return the current configuration.</p>
<i>Syntax:</i>	++eot_enable [0 1]
<i>Comment:</i>	0 disables and 1 enables sending the EOT character to the USB output

## **++eot\_char**

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller, device

*Description:* 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.

*Syntax:* ++eot\_char [<char>]

*Comment:* <char> is a decimal number that is less than 256

## **++eot\_char**

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller, device

*Description:* 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.

*Syntax:* ++eot\_char [<char>]

*Comment:* <char> is a decimal number that is less than 256

## ++flag

*Prologix compatible:* no

*Enhanced:* no

*Modes:* controller

*Description:* This command enables flags that perform a rudimentary handshaking when sending a large volume of data. These are not serial handshaking flags as such, but provide 3 indicators: interface ready, receive complete, send complete. When enabled, each of these actions will generate an output at the serial port of the following character sequences:

<i>Flag</i>	<i>Bit</i>	<i>Value</i>
AR488-RDY	1	0x01
Read^OK	2	0x02
Sent^OK	3	0x04

To turn on an individual flag, specify a value corresponding to its bit. To turn on multiple flags, the corresponding values are added. For example to turn on Read^OK and Sent^OK only, specify ++flags 6. To turn on all flags, use ++flags 7. When turning on the AR488-RDY flag, it is necessary to save the setting using the ++savecfg command otherwise it will be lost on restart.

*Syntax:* ++flag value

*Comment:* Bit values are OR'ed (added) to enable only the bits required.

## **++fndl**

*Prologix compatible:* no

*Enhanced:* no

*Modes:* controller

*Description:* Find listeners: searches the GPIB bus and finds all of the listeners within the given range. Addresses can be specified individually, separated by a space, comma or tab. They can also be specified as a range with a start address and end address separated by a hyphen. There must be no spaces between the hyphen and address values. The whole address range from 0 to 31 can be searched by using the keyword *all*.  
The command will return a comma separated list of listener addresses. Listeners on secondary addresses will be presented as an extended address in the format pri:sec.

*Syntax:* ++fndl addr, addr, addr....  
++fndl startaddr-endaddr  
++fndl all

*Comment:* Example output: 3,5:1,5:2,5:3,11,22

## **++help**

*Prologix compatible:*      yes

*Enhanced:*                      no

*Modes:*                          controller, device

*Description:*                  Returns a description of the command. The returned information is on the format:

`++cmd: [P] Short description of command`

where ++cmd is the command for which information was requested, [P] or [C] indicate whether this is a Prologix command or an AR488 custom command, and the remaining text is the command description.

Entering an invalid command will simply return:

`Unrecognized command`

*Syntax:*                          ++help <command>

*Comment:*                      <command> is a name of a valid command without the ++ prefix

## ++id

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* 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.

*Syntax:*

- ++id fwver
- ++id name [name]
- ++id serial [serialnum]
- ++id verstr [version string]

*Comment:*

*++id name*  
Displays or sets a short descriptive 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. If specified with a character string, the command will set the interface name to the provided string. By default, the name is not set and the command will not return a value.

*++id serial*  
Displays or sets an optional 9-digit 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. When specified without a parameter, the command returns the currently configured serial number. By default no serial number is set so the command will return '000000000'.

When specified with a parameter, the command sets the serial number to the provided alphanumeric numeric string.

*++id verstr*  
Displays or sets the user-defined version string that the controller responds with on startup 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. When provided with a character string of up to 47 characters, the command will set the version string to the supplied character string.

*++id fwver*



Displays the actual version of the controller firmware.

Examples:

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

## ++idn

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* This command is used to enable the facility for the interface to respond to a SCPI *\*idn?* command. Older instruments that do not respond to SCPI commands will not return anything in response to *\*idn?*. This feature will allow the AR488 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-999999999*.

*Syntax:* ++idn[0-2]

*Comment:*

## ++ifc

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller

*Description:* Asserts the GPIB IFC signal for 150 microseconds, making the AR488 the Controller-in-Charge on the GPIB bus.

*Syntax:* ++ifc

*Comment:*

## **++llo**

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller

*Description:* Disables 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 or illuminate an indicator on their display or control panel to show that remote control is active and that front/rear panel controls are now 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 subsequently addressed and sent a command (assuming that a LOC signal has not been sent 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.

*Syntax:* *++llo* [all]

*Comment:*

## **++loc**

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller

*Description:* Relinquishes remote control and re-enables front panel operation of the currently addressed instrument. Remote control of the instrument is relinquished 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 GPIB specification, this command would place all instruments 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 GPIB 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.

*Syntax:* ++loc [all]

*Comment:*

## ++lon

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* device

*Description:* Configures the controller to listen only to traffic on the GPIB bus. Since in *LON* mode the interface does not need to have a GPIB address assigned, the assigned GPIB address is ignored. The talker must be in Talk-Only mode and traffic is received over the GPIB bus irrespective of the currently set GPIB address. The interface can receive data in *LON* mode, but cannot send data, so effectively becomes a “listen-only” device. When issued without a parameter, the command returns the current state of *LON* mode. When *LON* mode is enabled, *TON* mode is automatically disabled.

*Syntax:* ++lon [0 | 1]

*Comment:* 0=disabled; 1=enabled  
See also ++ton and ++prom commands.

## **++macro**

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* 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 its programming language or protocol.

*Syntax:* ++macro [1-9]

*Comment:*

## ++mode

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller, device

*Description:* 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.

*Syntax:* ++mode [0|1]

*Comment:*

## ++ppoll

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* 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 a number of 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 be assigned 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.

*Syntax:* ++ppoll

*Comment:*

## ++prom

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* device

*Description:* Promiscuous mode allows data sent between a controller and device on the bus to be monitored. It is similar to LON mode, except that LON mode is a non-addressed mode that receives data from a *Talk-Only* node and where no controller is present on the GPIB bus. On the other hand *prom* mode can receive when a controller is present and addressing is used. Command bytes are ignored but the interface will receive data sent across the bus between the controller and any other device. When *prom* mode is enabled, LON and TON modes are automatically disabled.

*Syntax:* ++prom [0|1]

*Comment:*



## **++read**

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller

*Description:* This command can be used to read data from the instrument currently addressed using the *++addr* command. When used in this way, the command is Prologix compatible.

When provided with a primary address or a combination of primary and secondary address, it can read data directly from the instrument at the specified address without affecting the currently addressed instrument. This is an enhanced feature that is not available on the Prologix interface.

Depending on the parameters passed, data can be read until:

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

Timeout can be set using the *++read\_tmo\_ms* command which specifies the maximum permitted delay for a single character to be read. This value does not relate to the time taken overall to read all of the transmitted data. For details see the description of the *++read\_tmo\_ms* command.

*Syntax:* `++read [eoi|<char>]`  
`++read pri [sec] [eoi|<char>]`

*Comment:* *eoi* specifies that data is read until and EOI signal is detected  
<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

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller
<i>Description:</i>	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).
<i>Syntax:</i>	++read_tmo_ms <time>
<i>Comment:</i>	<time> is a decimal number between 0 and 32000 representing milliseconds

## ++ren

<i>Prologix compatible:</i>	no
<i>Enhanced:</i>	n/a
<i>Modes:</i>	controller
<i>Description:</i>	<p>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.</p> <p>When REN is being used to control the SN75161 GPIB transceiver integrated-circuit, this command is unavailable and will simply return <i>Unavailable</i>. (see the <i>Configuration</i> and the <i>Building an AR488 GPIB Interface</i> sections for more information). When issued without a parameter, the command returns the current status of the REN signal.</p>
<i>Syntax:</i>	++ren [0 1]
<i>Comment:</i>	0 = REN un-asserted 1 = REN asserted

## **++repeat**

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* Provides a way of repeating the same command multiple times, for example, to request multiple measurements from an instrument. Between 2 and 255 repetitions can be requested. It is also possible to request a delay of 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.

*Syntax:* ++repeat count delay cmdstring

*Comment:* *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

## **++rst**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller, device
<i>Description:</i>	Performs a reset of the controller.
<i>Syntax:</i>	++rst
<i>Comment:</i>	<p>Reset may fail and hang the board under certain circumstances. These include:</p> <ul style="list-style-type: none"><li>• 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.</li><li>• 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.</li></ul>

## **++savecfg**

*Prologix compatible:* no

*Enhanced:* no

*Modes:* controller, device

*Description:* 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 write configuration parameters to EEPROM “on the fly” every time they are changed. Instead, issuing the *++savecfg* command saves a snapshot of the complete and current configuration.

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.

*Syntax:* ++savecfg

*Comment:*

## **++send**

*Prologix compatible:* no

*Enhanced:* no

*Modes:* controller

*Description:* Sends data to the specified primary or combined primary and secondary (extended) address. Using this command does not affect the currently addressed instrument.

Ordinarily one might send:

[++unt]

[++unl]

++addr pri,sec

some\_data

This command combines that sequence into a single command. In addition ++auto is set to 1, then a read will be automatically performed after the data is sent to retrieve the response. In this way, all three actions (address, send, read) can be performed using just one command.

*Syntax:* ++send pri [sec] data

*Comment:* Addresses and data can be separated by a comma, space or tab

## ++spoll

*Prologix compatible:* yes

*Enhanced:* yes

*Modes:* controller

*Description:* 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 p

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* controller

*Description:* Returns the present status of the SRQ signal line. It returns 0 if SRQ is not asserted and 1 if SRQ is asserted.

*Syntax:* ++srq

*Comment:*

oll 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, 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.

*Syntax:* ++spoll [<PAD>|all|<PAD1> <PAD2> <PAD3>...]

*Comment:* <PAD> and <PADx> are the primary GPIB addresses.  
*all* specifies that all instruments should be polled.  
See also the ++allspoll and ++srqauto commands.



## **++srq**

*Prologix compatible:*    yes

*Enhanced:*                no

*Modes:*                    controller

*Description:*            Returns the present status of the SRQ signal line. It returns 0 if SRQ is not asserted and 1 if SRQ is asserted.

*Syntax:*                  ++srq

*Comment:*

## **++srqauto**

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* 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.

*Syntax:* ++srqauto [0|1]

*Comment:* 0=disabled, 1=enabled

## ++status

*Prologix compatible:* yes

*Enhanced:* no

*Modes:* device

*Description:* Sets or displays 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.

Bit:	Value:	Meaning:
7	128	Always zero
6	64	<b>RQS</b>
5	32	Calibration enabled or error
4	16	Output available. Front/rear switch.
3	8	Remote control
2	4	Auto-zero
1	2	Autorange enabled. Front/rear switch.
0	1	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.

*Syntax:* ++status [byte]

*Comment:* [byte] is a decimal number between 0 and 255.

## **++tct**

<i>Prologix compatible:</i>	no
<i>Enhanced:</i>	n/a
<i>Modes:</i>	controller
<i>Description:</i>	The ++tct command sends a command to another device on the GPIB bus to signal it to become the controller and take control of the bus. To send the command, the sender must be in controller mode. On successfully transmitting the command, the interface then relinquishes control and switches into device mode.
<i>Syntax:</i>	++tct ADDR
<i>Comment:</i>	ADDR is the GPIB address of the device being requested to take control

## ++ton

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller

*Description:* The ++ton command configures the interface to send data in non-addressed mode on the GPIB bus. When in this mode, the interface does not require a GPIB address to be assigned, therefore any address that is already set will be ignored. Only one talker can exist on the bus, but multiple receivers in listen-only (LON) mode can listen to and accept the transmitted data. In TON mode, the interface can send, but not receive, so effectively becomes a “talk-only” device.

There are two talk modes:

*Unbuffered* mode (1) – this is the default. In this mode, characters sent to the serial port are immediately transmitted to the GPIB bus. No attempt is made to buffer or filter the data. Once the ++ton 1 command has been entered, a reset will be required (by powering off the interface or pressing the *reset* button) to exit the mode.

*Buffered* mode (2) – in this mode, data sent over USB is buffered in the same way as in controller mode. Special characters such as 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) all need to be escaped with the Escape character hex 0x1B. In this mode, the interface will continue to parse the serial buffer and accept commands so it is possible to turn TON mode off with the ++ton 0 command.

When issued without a parameter, the command returns the current state of the *ton* mode.

*Syntax:* ++ton [0|1|2]

*Comment:* 0=disabled; 1=enabled, unbuffered; 2=enabled, buffered;

## **++trg**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	no
<i>Modes:</i>	controller
<i>Description:</i>	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 <i>++trg</i> , the instrument can be manually triggered and the result read with <i>++read</i> .
<i>Syntax:</i>	<i>++trg</i> [pad1 ... pad15]
<i>Comment:</i>	Padx is an optional primary address. See also <i>++trg</i> and <i>++read</i> .

## **++ver**

<i>Prologix compatible:</i>	yes
<i>Enhanced:</i>	yes
<i>Modes:</i>	controller, device
<i>Description:</i>	<p>Display the controller firmware version. If the version string has been customized with <i>++id verstr</i>, then <i>++ver</i> will display the custom version string. Issuing the command with the parameter 'real' will always display the default AR488 version string. The custom version string is saved when using the <i>++savecfg</i> command.</p> <p>Using this command might be helpful where a program or script requires a specific version string in order to identify the interface.</p>
<i>Syntax:</i>	<i>++ver</i> [real]
<i>Comment:</i>	The [real] keyword refers to the default version string. See also the <i>++id</i> , <i>++ver</i> and <i>++savecfg</i> commands.

## **++verbose**

<i>Prologix compatible:</i>	no
<i>Enhanced:</i>	n/a
<i>Modes:</i>	controller, device
<i>Description:</i>	<p>Toggles verbose mode ON and OFF. Humans usually like to see feedback in response to what they are typing. The first time it is issued, it will turn verbose mode on. When verbose mode is on, additional information will be provided in response to some commands. A command prompt '&gt;' will also appear and provide confirmation that a command has been completed.</p> <p>When issue subsequently, verbose mode will be turned off again.</p>
<i>Syntax:</i>	++verbose
<i>Comment:</i>	Will display the status of verbose mode each time it is issued.

## ++xdiag

*Prologix compatible:* no

*Enhanced:* n/a

*Modes:* controller, device

*Description:* Can be used to show the current state of pins or to test whether individual signals are being asserted or un-asserted. The command takes two parameters: mode and value. To manipulate the GPIB control signals use mode 1. For data signals use mode 0.

To view the current state of pins simply run `++xdiag pins`. This will show whether pins are currently high (1) or low (0).

To test asserting a signal/data bit, simply specify one of the values in the table below. To assert multiple signals/bits simultaneously, simply add the values of the signals to be asserted. To leave a signal un-asserted while asserting all remaining signals, subtract its value from 255. To leave multiple signals un-asserted while asserting all remaining signals, subtract their combined value from 255.

Depending on which mode and value is selected at the time of testing, the corresponding group of signals and pins will be asserted or un-asserted and a table printed showing the result. Signals will automatically revert back to either controller or device mode after 10 seconds.

Please note that GPIB signals are HIGH (1) when inactive (un-asserted), LOW (0) when active (asserted).

*Syntax:* ++xdiag mode [value]  
++xdiag pins

*Comment:* Mode can be either 0 for data signals, or 1 for command signals. value must be specified when mode is set to 0 or 1 and indicates the bits or signals that are to be turned on or off



**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

**Output:**

DIO1:	[14]	1
DIO2:	[15]	1
DIO3:	[16]	1
DIO4:	[17]	1
DIO5:	[18]	1
DIO6:	[19]	1
DIO7:	[4]	1
DIO8:	[5]	1

**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

**Output:**

IFC:	[8]	1
NDAC:	[9]	0
NRFD:	[10]	0
DAV:	[11]	1
EOI:	[12]	1
SRQ:	[2]	1
REN:	[3]	0
ATN:	[7]	1

# Appendices

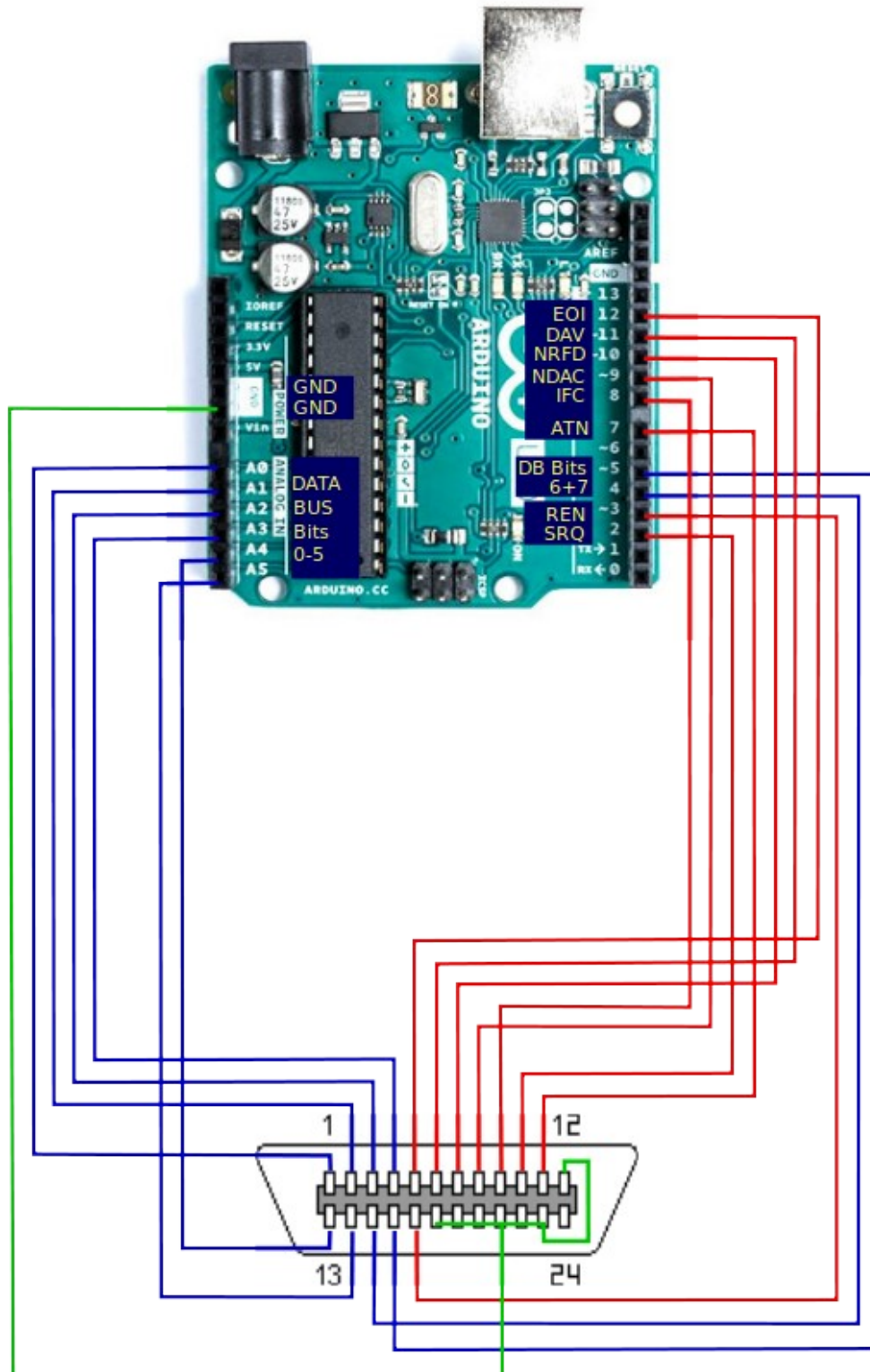
## A – ATmega328P Boards

### Connection details for UNO and Nano

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

<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
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

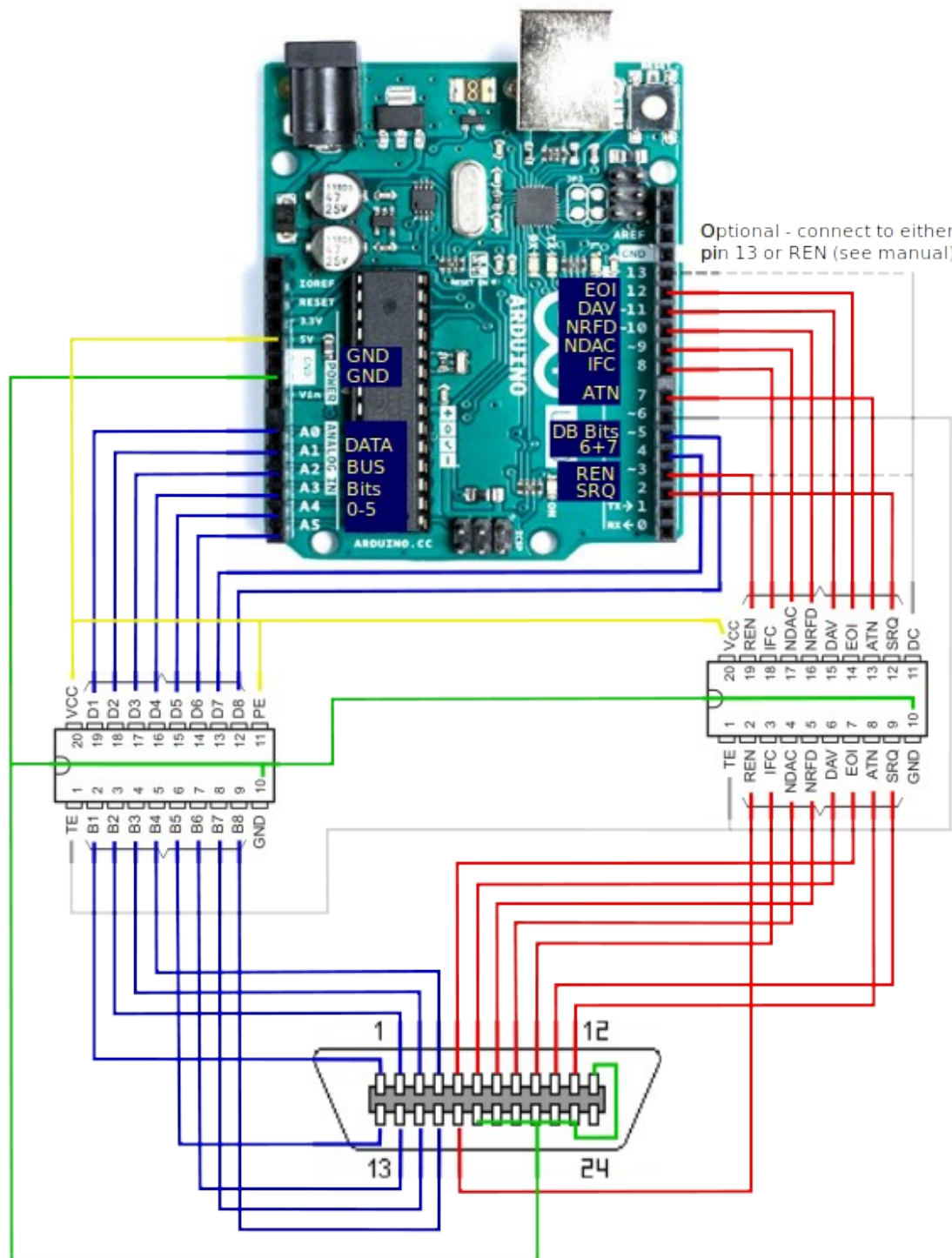
## Wiring diagram



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.



## B - ATmega2560 boards

### Connection details for layout AR488\_MEGA2560\_D

The pinout for the Mega 2560 default layout AR488\_MEGA2560\_D is as follows:

<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
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.

## Connection details for layout AR488\_MEGA2560\_E1

The pinout on the Mega 2560 layout AR488\_MEGA2560\_E1 is as follows:

<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
D44	7	NRFD
D42	6	DAV
D40	5	EOI
D38	17	REN
D50	10	SRQ
D52	11	ATN
D46	8	NDAC
D48	9	IFC
D30	1	DIO1
D32	2	DIO2
D34	3	DIO3
D36	4	DIO4
D22	13	DIO5
D24	14	DIO6
D26	15	DIO7
D28	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

This layout was chosen to use the inside (even numbered) row of the end connector on the Mega2560, leaving remaining pins available for other purposes including shields.



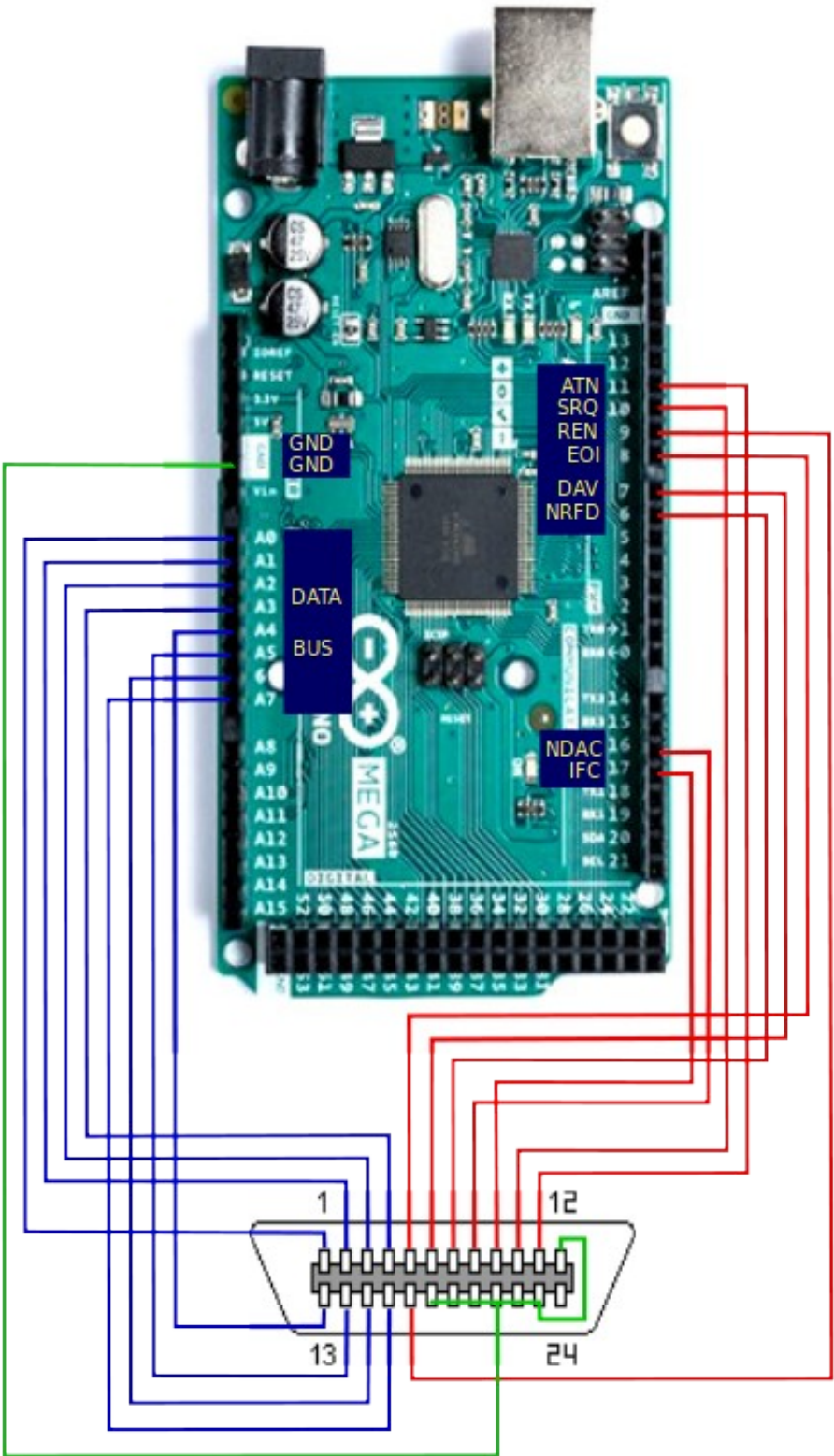
## Connection details for layout AR488\_MEGA2560\_E2

The pinout on the Mega 2560 layout AR488\_MEGA2560\_E2 is as follows:

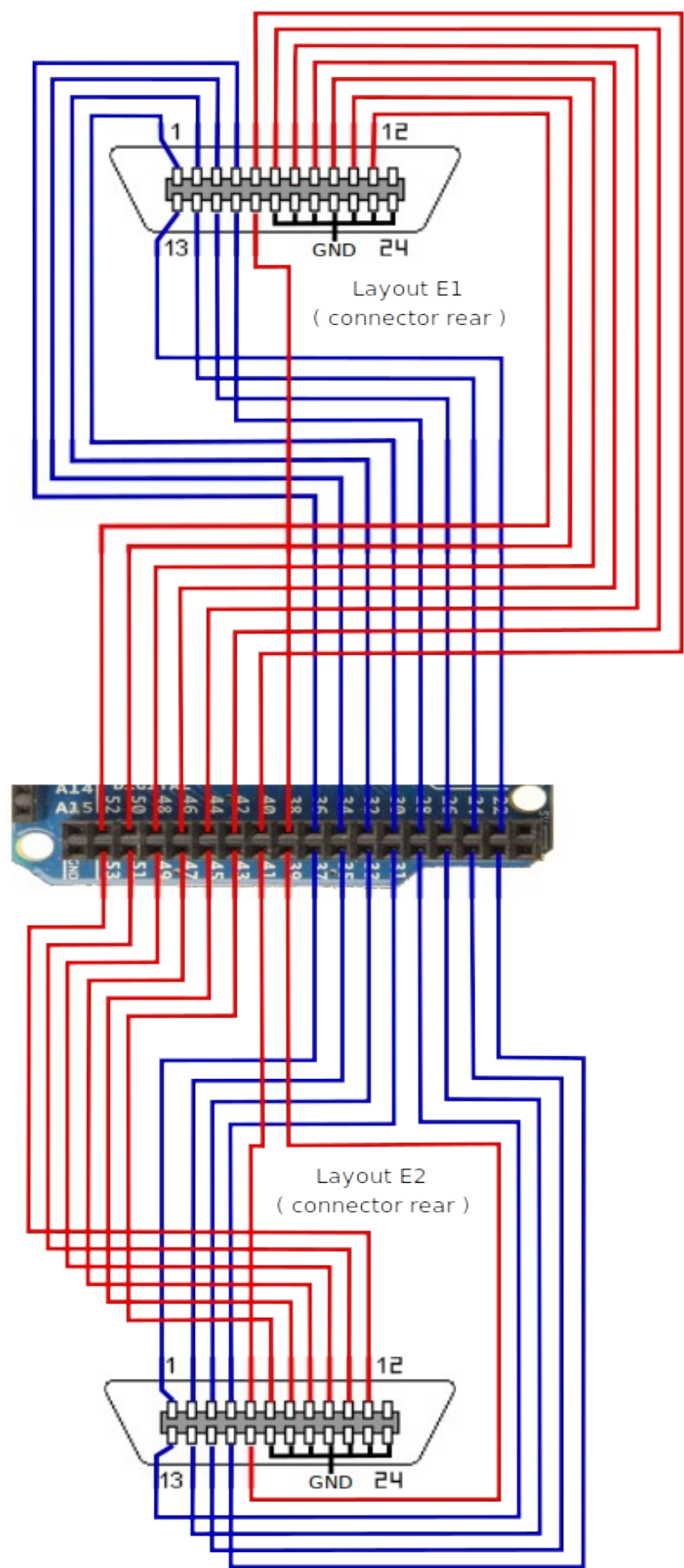
<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
D45	7	NRFD
D43	6	DAV
D41	5	EOI
D39	17	REN
D51	10	SRQ
D53	11	ATN
D47	8	NDAC
D49	9	IFC
D31	1	DIO1
D33	2	DIO2
D35	3	DIO3
D37	4	DIO4
D23	13	DIO5
D25	14	DIO6
D27	15	DIO7
D29	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

This layout was designed to use only the even outer (odd numbered) row of the end connector on the Mega2560, leaving remaining pins available for other purposes including shields.

Wiring Diagram for layout - AR488\_MEGA2560\_D

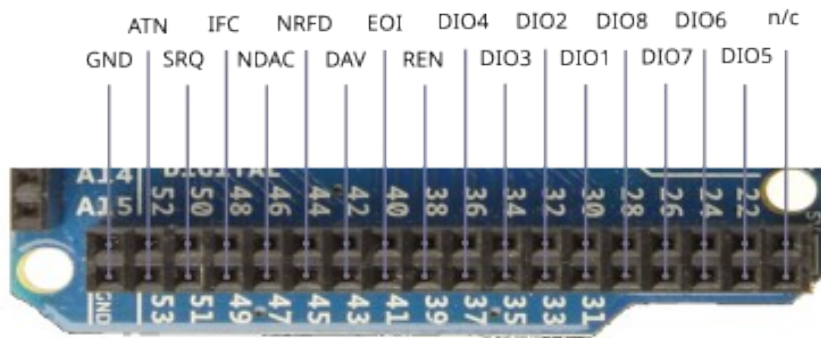


Wiring Diagram - layout E1 and E2



## E1/E2 layout end header pinout details

## Arduino Mega 2560 E1/E2 pinout



## C – ATmega32u4 Boards

### Connection details for the Arduino Micro

The pinout on the Arduino Micro is as follows:

<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
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](https://oshpark.com/shared_projects/HrS1HLSE)

## Connection details for the Leonardo R3

The pinout on the Arduino Leonardo R3 is as follows:

<i>Arduino:</i>	<i>GPIO connector:</i>	<i>Function:</i>
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 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.

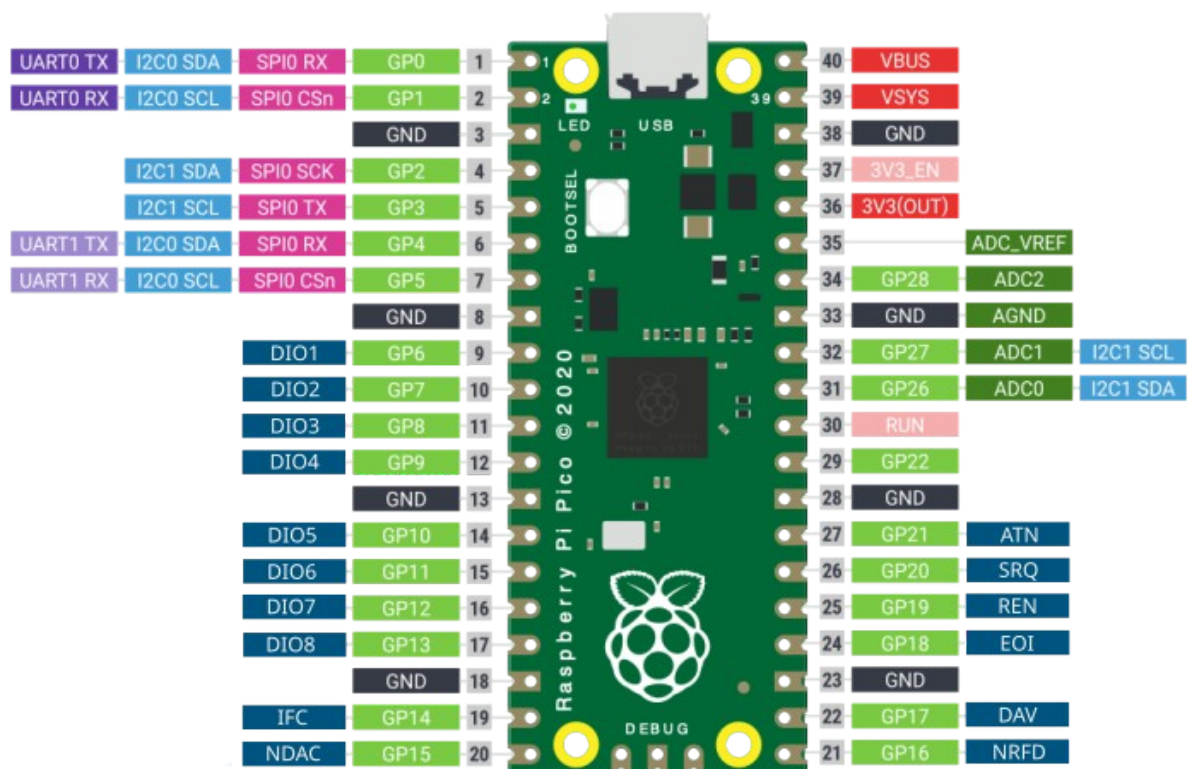
## D – Raspberry RP2040/RP2350 Boards

### Connection details for Raspberry Pico/W layout RAS\_PICO\_L1

The pinout on the Pico layout RAS\_PICO\_L1 is as follows:

<i>Pico:</i>	<i>GPIO connector:</i>	<i>Function:</i>
GP14	9	IFC
GP15	8	NDAC
GP16	7	NRFD
GP17	6	DAV
GP18	5	EOI
GP19	17	REN
GP20	10	SRQ
GP21	11	ATN
GP6	1	DIO1
GP7	2	DIO2
GP8	3	DIO3
GP9	4	DIO4
GP10	13	DIO5
GP11	14	DIO6
GP12	15	DIO7
GP13	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND

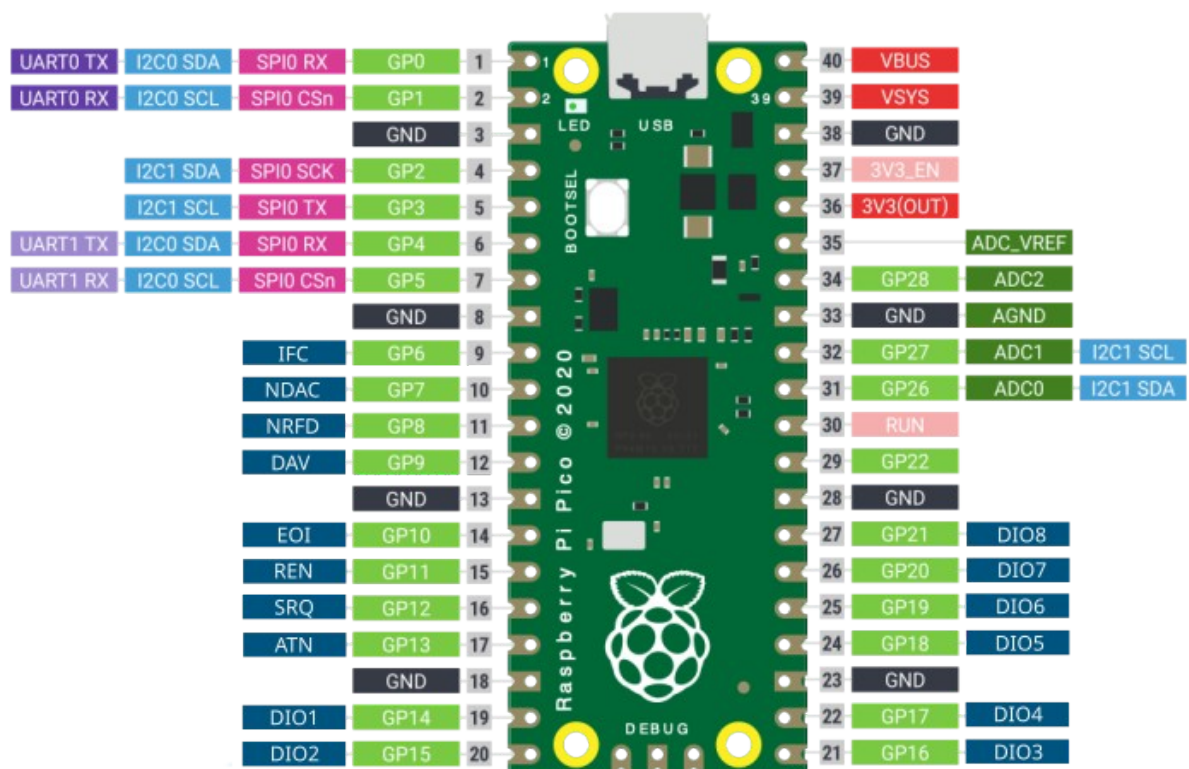




## Connection details for Raspberry Pico/W layout RAS\_PICO\_L2

The pinout on the Pico layout RAS\_PICO\_L2 is as follows:

<i>Pico:</i>	<i>GPIO connector:</i>	<i>Function:</i>
GP6	9	IFC
GP7	8	NDAC
GP8	7	NRFD
GP9	6	DAV
GP10	5	EOI
GP11	17	REN
GP12	10	SRQ
GP13	11	ATN
GP14	1	DIO1
GP15	2	DIO2
GP16	3	DIO3
GP17	4	DIO4
GP18	13	DIO5
GP19	14	DIO6
GP20	15	DIO7
GP21	16	DIO8
GND	12	Shield
GND	18,19,20,21,22,23	GND



## **E – Espressif ESP32 Boards**

### **Connection details for layout ESP32\_DEVKIT1\_WROOM\_32**

This is the connection detail the ESP32 WROOM Devkit 1 board.

## Connection details for layout ESP32\_TTGO\_T8\_161

## Connection details for layout ESP32\_LOLIN32\_161

## Connection details for layout ESP32\_ESP32DEV

## **F – MightyCore ATmega644/1284 Boards**

### **Connection Details for layout AR488\_MEGA644P\_MCGRAW**

This is the connection detail for the MightyCore ATmega644 and ATmega1284 boards used for layout AR488\_MEGA644P\_MCGRAW.



## G - IEEE Connector Pinout

