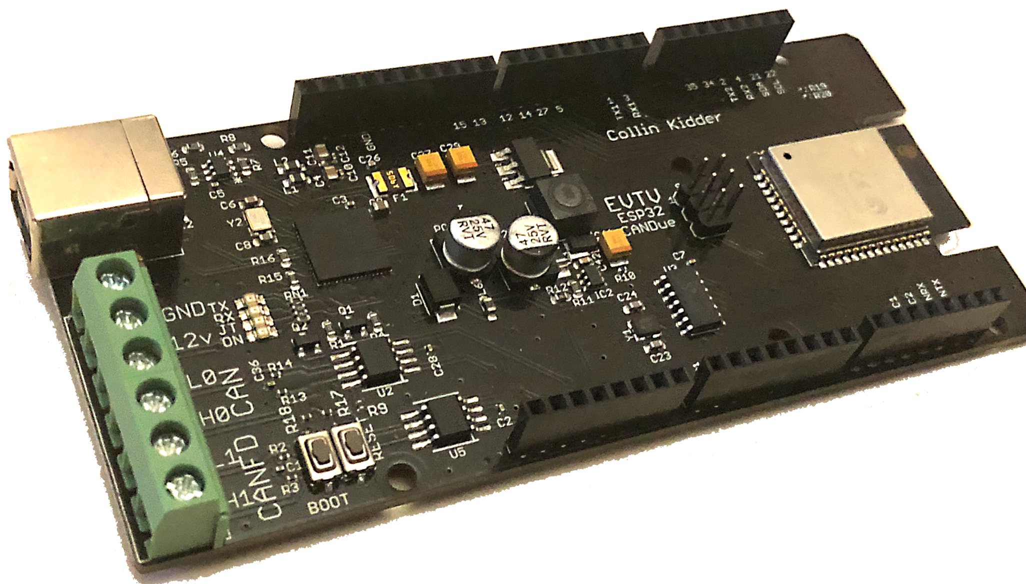
EVTV ESP32 CANDue



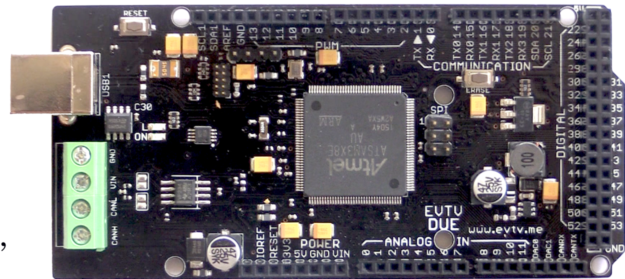
Tensilica Xtensa LX6
ESP32 WROOM Microcontroller
With Dual CAN Bus Ports
WiFi and Bluetooth BLE

Copyright 2018 EVTV LLC

INTRODUCTION

This document describes the EVTV ESP32 CANdue Microcontroller single-board computer.

Since 2015, EVTV has very successfully produced our own version of the popular Arduino Due microcontroller board. Using the Arduino R3 form factor, this EVTV product added a couple of key elements for our needs, specifically onboard EEPROM memory and a Controller Area Network (CAN) port. Almost everything involved with electric vehicles requires communication and CAN has emerged as the common protocol not only for electric vehicles, but really all modern automobiles.



With its integrated CAN port and a much more durable and robust mini-B printer port style USB connector, this has become the go-to device for any sort of inexpensive controller application requiring CAN. The 84MHz clockspeed and 32-bit architecture has made this a very quick and capable device for most applications of any sort.

Further, it is fully compatible with the Arduino Integrated Design Environment (IDE). This enormously popular free software has allowed millions to learn to program microprocessors. It's clean, simple, installs easily and it is free.

But the pace of development in technology continues. And today, features such as WiFi and Bluetooth Low Energy, unheard of on such a small single board computer, have actually now been subsumed into individual chips.

Espressif has introduced there Tensilica Xtensa line of microcontrollers and the ESP32 version features a 240Mhz dual-core 32-bit controller with Wifi and Bluetooth communications built right into the chip.

ESP32 integrates Wi-Fi (2.4 GHz band) and Bluetooth 4.2 wireless radio solutions on a single chip, along with dual high performance cores, Ultra Low Power co-processor and several peripherals. Powered by 40 nm technology, ESP32 provides a robust, highly integrated platform to meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.

Copyright 2018 EVTV LLC

This represents a three-fold increase in speed with some very key communications features. Bluetooth Low Energy (BLE) version 4.2 is particularly interesting in that it allows easy integration of controllers and smart phones. The ESP32 chip has become a darling of the Internet of Things (ioT) community.

And so EVTV introduces the **EVTV ESP32 CANDue**. This is physically very nearly the same board form factor, with the ESP32 Wifi/Bluetooth chip at 240MHz, and TWO CAN ports. Screw terminals allow easy access to wire to CAN busses and automotive 12v power.

It runs easily on automotive 12v and has onboard power supply to drop that down to the 3.3v levels used on these microcontrollers.

While the EVTV CANdue remains a staple, the **EVTV ESP32CANDue** represents the next generation of this popular little board and provides an additional CAN port, much higher USB port speeds, and WiFi and Bluetooth wireless communications – all on a 4x2 inch board.

For us, one of the most valuable features of the ESP32 chip is an included internal Controller Area Network (CAN) controller. But to use this feature, you must provide a CAN transceiver chip to actually interface with a CAN network.

Introduced in 1988 by Bosch Gmbh, CAN has become the defacto communications protocol used on virtually all modern automobiles but particularly all modern electric vehicles.

The Texa Instruments **SN65HVD234** chip is used in applications employing the controller area network (CAN) serial communication physical layer in accordance with the ISO 11898 standard. As a CAN transceiver, each provides transmit and receive capability between the differential CAN bus and a CAN controller, with signaling rates up to 1 Mbps.

Designed for operation in especially harsh environments, the device features cross-wire protection, overvoltage protection up to ±36 V, loss of ground protection, overtemperature (thermal shutdown) protection, and common-mode transient protection of ±100 V. These devices operate over a wide –7 V to 12 V common-mode range. This transceiver is the interface between the host CAN controller on the microprocessor and the differential CAN bus used in industrial, building automation, transportation, and automotive applications.

Collin Kidder wrote the **due_can** library available for all Arduino CAN applications. This library provides advanced interrupt features, filters, and masks far beyond the earlier libraries available for CAN adapters on the 8-bit Arduino boards making CAN communications an EASY software task instead of a chore and with performance just not attainable with the earlier chips and libraries.

Copyright 2018 EVTV LLC

In automotive applications in the electric vehicle field, this can be the difference between success and failure. We've learned the hard way that the CAN frame rates of the Tesla Model S, for example, can easily exceed 1500 frames per second, and the existing hardware and software solutions for Arduino just cannot deal with this without dropping frames.

The CAN interface uses four screw terminals on the edge of the board that are named CAN0 and CAN1 in software by convention and include CANHI and CANLO connections for the differential CAN bus.

With this generation, we have added a second CAN channel CAN1. This uses a new Microchip **MCP2517FD** CAN controller offering a new "flexible data rate" concept and frame lengths as high as 64-bytes. Automotive OEM's are rapidly adopting this new CAN protocol because it makes it much faster and easier to transport largish firmware upgrades to other devices on the CAN bus.

CAN1 CAN be operated as a normal CAN bus participant as long as the FD features are not used. But you cannot mix and match devices on the same bus that use both CAN2.0 and CAN FD.

12V POWER

Software development on Arduino boards commonly involves powering the board at +5vdc by USB connection that is also used to upload the software. Additionally, a barrel connector is provided for 9-16v to power the board without USB.

In automotive applications, this barrel connector is again not sufficiently robust. It can easily vibrate loose, leaving your board disconnected from the circuit.

We have provided two screw terminal connections for 12v power and the 12v return – usually frame ground. This allows you to easily and securely connect your **EVTV ESP32 CANdue** microcontroller to vehicle power. Typically, you would turn on the board which would automatically initialize and begin running the last program loaded, when powered up by 12v.

In this way, you can easily connect by USB to upload programs or make configuration changes to existing programs. But then you can disconnect the laptop and put it away. The board will faithfully come up and run the new software or configuration whenever powered by 12v.

Copyright 2018 EVTV LLC

TENSILICA XTENSA ESP32

We first encountered descriptions of this new Chinese multicontroller chip by Espressif in December 2015. These 32-bit chips were not widely available until the spring of 2016 and we really wanted to do something with one then. But documentation was meager and there was talk of developing an Arduino IDE interface for it. So we adopted a wait-and-see watch on this device.

The exciting part was the inclusion of both WiFi and Bluetooth wireless communications IN the microprocessor and the dual core capability.

We were VERY interested in Blue Tooth Low Energy BLE as a medium for developing Smart Phone/Tablet displays for all of our control products.

But early Bluetooth devices were quite limited. The Adafruit nRF module we first tried was limited to a total of 30 "characteristics" or variables that could be interfaced and suffered badly from blocking issues. Most of our controllers have CAN bus timing requirements and cannot simply wait around for Bluetooth transmissions to be completed.

At the core of this module is the ESP32-D0WDQ6 chip*. The chip embedded is designed to be scalable and adaptive. There are two CPU cores that can be individually controlled, and the clock frequency is adjustable from 80 MHz to 240 MHz. The user may also power off the CPU and make use of the low-power co-processor to constantly monitor the peripherals for changes or crossing of thresholds. It includes the freeRTOS operating system and so the BLE and radio functions can run separately on a different core from your main program, eliminating many of the blocking issues and delays.

Using Wi-Fi allows a large physical range and direct connection to the internet through a Wi-Fi router, while using Bluetooth allows the user to conveniently connect to the phone or broadcast low energy beacons for its detection.

ESP32 supports a data rate of up to 150 Mbps, and 20.5 dBm output power at the antenna to ensure the widest physical range. As such the chip does offer industry-leading specifications and the best performance for electronic integration, range, power consumption, and connectivity.

The operating system chosen for ESP32 is freeRTOS with LwIP; TLS 1.2 with hardware acceleration is built in as well. Secure (encrypted) over the air (OTA) upgrade is also

Copyright 2018 EVTV LLC

supported, so that developers can continually upgrade their products even after their release.

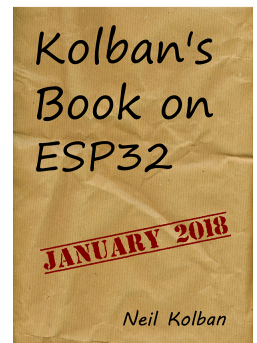
This means communications can be handled by ONE processor while the normal operations are handled by the other, and BOTH run at 240MHz, about three times the speed of our existing SAM3X processor. It's a twofer-one deal.

Further, it featured an unlimited number of services and characteristics (think variables) that can be advertised and updated via BLE. AND it can act not only as a peripheral (server) but also a client (the normal Smart Phone) role.

This opens the door to our ESP32 controller in turn monitoring and controlling OTHER BLE sensors for items such as voltage, temperature, current, etc. and bringing all that together for a single interface to the Smart Phone. That's pretty exciting.

The good news is that the Arduino IDE interface DID come about. The bad news is that the BLE and WiFi support within it ranged from non-existent to a horror.

But the community continued to work on this over time and today, WiFi, BLE, and multitasking appear to be readily available via the Arduino IDE interface. And improving as we go along.



Better, a true open source hero has emerged in the form of Neil Kolban who authored a detailed document covering the ESP32. Better yet, he has continued to update it almost continuously and at this writing, the latest edition was issued in January 2018. https://leanpub.com/kolban-ESP32/ It's $4.99 and worth about 8x that amount. He's consummately qualified to write it and has done an excellent service documenting the ESP32 which of course as a Chinese chip had some serious weakness in the area of documentation. It's now over 1000 pages.

He also does a series of of excellent video tutorials specifically on the ESP32 and BLE. https://www.youtube.com/watch?v=2_vlF_02VXk

Kolban is a senior engineer at Salient Process of Sacramento California, working on Business Process Management and Decision Management software for IBM mainframe environments. He actually resides in North Richland Hills Texas. He has a Masters degree in Computer Science from the University of Glasgow, Scotland.

He's actually gone quite beyond this and developed some C++ classes specifically for the ESP32 BLE function. https://github.com/nkolban

Copyright 2018 EVTV LLC

From a historical perspective, I may have lived too long. In October of 1977, Digital Equipment introduced the VAX 11/780 mini-computer, a head to head competitor to the venerated IBM 360.

In 1984 Reinhold P Weicker wrote a benchmark program that would perform roughly one million integer instructions per second on either minicomputer. He called it the Dhrystone benchmark as a bit of a pun on the floating point operation benchmark popular at the time, the Whetstone. In any event, the Dhrystone Million Instructions Per Second or DMIPS has served as a handy benchmark of computer processing power since.

The Teslilica ESP 32 is rated at 600 DMIPS. That is, this little chip, widely available for less than $3.50, is about 600 times more powerful than the two leading minicomputers of 1977, the DEC VAX 11/780 and the IBM360.

A more modern comparison would be to the original Raspberry Pi running at 700 MHz, almost three times the clock speed of the ESP32, with 847 DMIPS.

With Bluetooth Low Energy and WiFi built in, you can see why we are excited about this little chip.

ESP32-D0WDQ6 contains two low-power Xtensa® 32-bit LX6 microprocessors. The internal memory includes:

- 448 kB of ROM for booting and core functions.

- 520 kB (8 kB RTC FAST Memory included) of on-chip SRAM for data and instruction. – 8 kB of SRAM in RTC, which is called RTC FAST Memory and can be used for data storage; it is accessed by the main CPU during RTC Boot from the Deep-sleep mode.

- 8 kB of SRAM in RTC, which is called RTC SLOW Memory and can be accessed by the co-processor during the Deep-sleep mode.

- 1 kbit of eFuse, of which 256 bits are used for the system (MAC address and chip configuration) and the remaining 768 bits are reserved for customer applications, including Flash-Encryption and Chip-ID.

Features of the ESP32 include the following:[3]

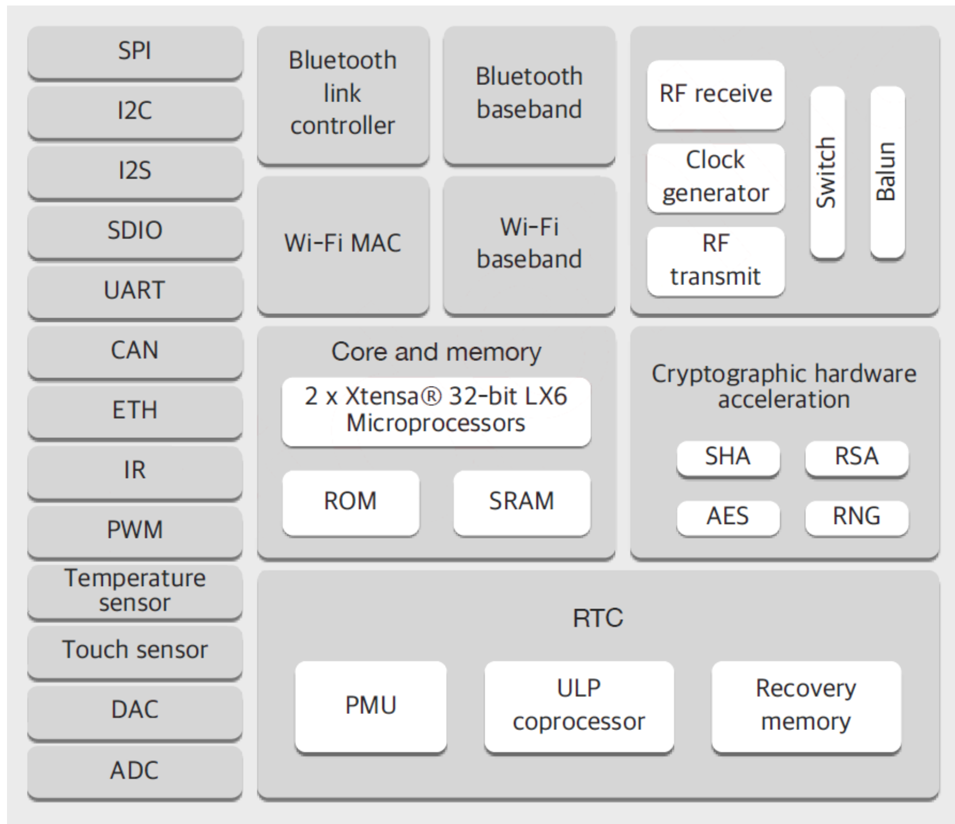
- Processors:
  - CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 160 or 240 MHz and performing at up to 600 DMIPS
  - Ultra low power (ULP) co-processor
- Memory: 520 KB SRAM
- Wireless connectivity:

Copyright 2018 EVTV LLC

- Wi-Fi: 802.11 b/g/n
- Bluetooth: v4.2 BR/EDR and BLE
- Peripheral interfaces:
  - 12-bit SAR ADC up to 18 channels
  - 2 × 8-bit DACs
  - 10 × touch sensors (capacitive sensing GPIOs)
  - Temperature sensor
  - 4 × SPI
  - 2 × I²S interfaces
  - 2 × I²C interfaces
  - 3 × UART
  - SD/SDIO/CE-ATA/MMC/eMMC host controller
  - SDIO/SPI slave controller
  - Ethernet MAC interface with dedicated DMA and IEEE 1588 Precision Time Protocol support
  - CAN bus 2.0
  - Infrared remote controller (TX/RX, up to 8 channels)
  - Motor PWM
  - LED PWM (up to 16 channels)
  - Hall effect sensor
  - Ultra low power analog pre-amplifier
- Security:
  - IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
  - Secure boot
  - Flash encryption
  - 1024-bit OTP, up to 768-bit for customers
  - Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)
- Power management:
  - Internal low-dropout regulator
  - Individual power domain for RTC
  - 5uA deep sleep current
  - Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt

Copyright 2018 EVTV LLC

ESP32 BLOCK DIAGRAM



ESP32 MODULE PINOUT

The EVTV ESP32 CANDue uses a 38-pin Espressif ESP32-WROOM module. This is a bit different from the Atmel chips you may be accustomed to in that it is much more limited in the number of pins available. But who ever used all those anyway? And the ESP32 allows general purpose input and output (GPIO) on almost ALL pins and so you can define them as ADC, DAC, digital in, digital out etc. with great flexibility.

Copyright 2018 EVTV LLC

ESP32 PINOUT

Power
GND
Serial Pin
Analog Pin
Control
Physical Pin
Port Pin
Touch Pin
DAC Pin
PWM Pin

On board Antenna

GND 1
VCC 2
As know as CHPD EN 3
ADC_H ADC1_0 SensVP GPIO36 4
ADC_H ADC1_3 SensVN GPIO39 5
ADC1_6 GPIO34 6
ADC1_7 GPIO35 7
XTAL32 ADC1_4 Touch9 GPIO32 8
XTAL32 ADC1_5 Touch8 GPIO33 9
ADC2_8 DAC_1 GPIO25 10
ADC2_9 DAC_2 GPIO26 11
ADC2_7 Touch7 GPIO27 12
HSPICLK MTMS ADC2_6 Touch6 GPIO14 13
HSPIQ MTDI ADC2_5 Touch5 GPIO12 14
GND 15
HSPIID MTCK ADC2_4 Touch4 GPIO13 16
SPIHD HS1D2 SDD2 U1RXD GPIO9 17
SPIWP HS1D3 SDD3 U1TXD GPIO10 18
SPICS0 HS1CMD SDCMD U1RTS GPIO11 19

38 GND
37 GPIO23
36 GPIO22 HS2CLK VSPIWP
35 GPIO1 HS2D1 U0TXD CLK03
34 GPIO3 HS2D0 U0RXD CLK02
33 GPIO21 HS2CMD VSPIH2
32 GPIO20 HS2D3 VSPIID
31 GPIO19 HS2D2 VSPIQ
30 GPIO18 HS1D7 VSPICLK
29 GPIO5 HS1D6 VSPICS0
28 GPIO17 HS1D5
27 GPIO16 HS1D4
26 GPIO4 Touch0 ADC2_0 HSPIHD
25 GPIO0 Touch1 ADC2_1 CLK01
24 GPIO2 Touch2 ADC2_2 HSPIWP
23 GPIO15 Touch3 ADC2_3 MTD0 HSPICS0 U0CTS
22 GPIO8 SDD1 HS1D1 SPI
21 GPIO7 SDD0 HS1D0 SPIQ
20 GPIO6 U1CTS SDCLK HS1CLK SPICLK

http://esp32.com/

BY SA
07 DEC 2015
ver 1 rev 0

There are of course a few provisos and special purpose pins, such as the CAN output pins that could be used for GPIO but if you want CAN you must use the pins dedicated to that. The diagram shows the variety of purposes each pin can be used for.

We have tried to map the pins to our Arduino form factor board as artfully as possible allowing minimum confusion between the Arduino interface, the ESP IDF programming interface, and the pin numbers themselves.

Copyright 2018 EVTV LLC

SPECIFICATIONS

ESP-WROOM-32 Specifications

Categories	Items	Specifications
Wi-Fi	RF certification	FCC/CE/IC/TELEC/KCC/SRRC/NCC
	Protocols	802.11 b/g/n/e/i (802.11n up to 150 Mbps)
		A-MPDU and A-MSDU aggregation and 0.4 μs guard interval support
	Frequency range	2.4 ~ 2.5 GHz
Bluetooth	Protocols	Bluetooth v4.2 BR/EDR and BLE specification
	Radio	NZIF receiver with -97 dBm sensitivity
		Class-1, class-2 and class-3 transmitter
		AFH
	Audio	CVSD and SBC
Hardware	Module interface	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR
		GPIO, capacitive touch sensor, ADC, DAC, LNA pre-amplifier
	On-chip sensor	Hall sensor, temperature sensor
	On-board clock	40 MHz crystal
	Operating voltage/Power supply	2.7 ~ 3.6V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Operating temperature range	-40°C ~ +85°C
	Ambient temperature range	Normal temperature
	Package size	18±0.2 mm x 25.5±0.2 mm x 3.1±0.15 mm
Software	Wi-Fi mode	Station/SoftAP/SoftAP+Station/P2P
	Wi-Fi Security	WPA/WPA2/WPA2-Enterprise/WPS
	Encryption	AES/RSA/ECC/SHA
	Firmware upgrade	UART Download / OTA (download and write firmware via network or host)
	Software development	Supports Cloud Server Development / SDK for custom firmware development
	Network protocols	IPv4, IPv6, SSL, TCP/UDP/HTTP/FTP/MQTT
	User configuration	AT instruction set, cloud server, Android/iOS app

Copyright 2018 EVTV LLC

Absolute Maximum Ratings

Parameter	Symbol	Min	Typ	Max	Unit
Power supply	VDD	2.7	3.3	3.6	V
Minimum current delivered by power supply	I_{VDD}	0.5	-	-	A
Input low voltage	V_{IL}	-0.3	-	$0.25 \times V_{IO}$[1]	V
Input high voltage	V_{IH}	$0.75 \times V_{IO}$[1]	-	V_{IO}[1]+0.3	V
Input leakage current	I_{IL}	-	-	50	nA
Input pin capacitance	C_{pad}	-	-	2	pF
Output low voltage	V_{OL}	-	-	$0.1 \times V_{IO}$[1]	V
Output high voltage	V_{OH}	$0.8 \times V_{IO}$[1]	-	-	V
Maximum output drive capability	I_{MAX}	-	-	40	mA
Storage temperature range	T_{STR}	-40	-	85	°C
Operating temperature range	T_{OPR}	-40	-	85	°C

Wi-Fi Radio Characteristics

Description	Min	Typical	Max	Unit
Input frequency	2412	-	2484	MHz
Input reflection	-	-	-10	dB
Tx power				
Output power of PA for 72.2 Mbps	13	14	15	dBm
Output power of PA for 11b mode	19.5	20	20.5	dBm
Sensitivity				
DSSS, 1 Mbps	-	-98	-	dBm
CCK, 11 Mbps	-	-91	-	dBm
OFDM, 6 Mbps	-	-93	-	dBm
OFDM, 54 Mbps	-	-75	-	dBm
HT20, MCS0	-	-93	-	dBm
HT20, MCS7	-	-73	-	dBm
HT40, MCS0	-	-90	-	dBm
HT40, MCS7	-	-70	-	dBm
MCS32	-	-89	-	dBm
Adjacent channel rejection				
OFDM, 6 Mbps	-	37	-	dB
OFDM, 54 Mbps	-	21	-	dB
HT20, MCS0	-	37	-	dB
HT20, MCS7	-	20	-	dB

Copyright 2018 EVTV LLC

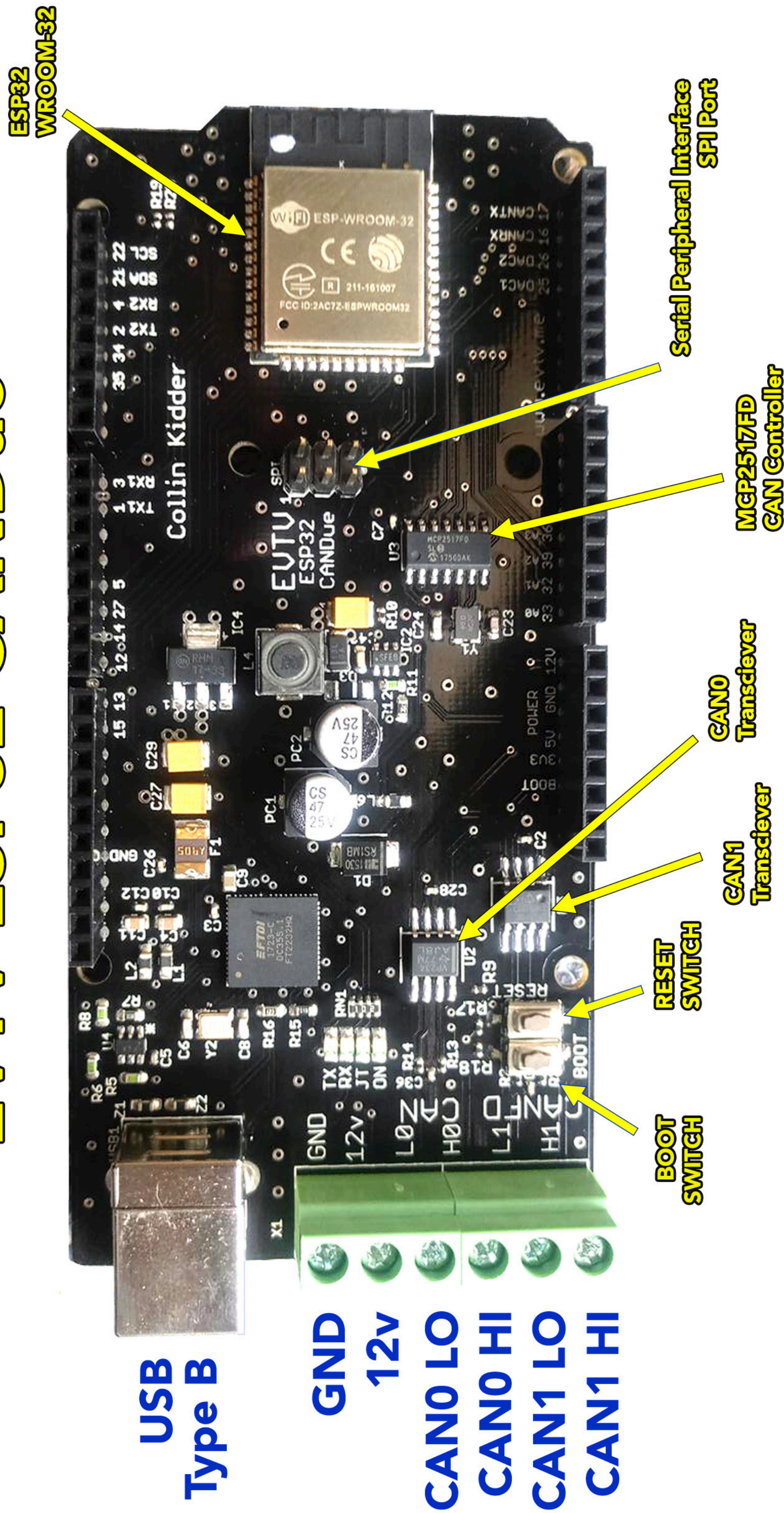
ESP32 Bluetooth Low Energy Transmitter Specification

Parameter	Conditions	Min	Typ	Max	Unit
Sensitivity @30.8% PER	-	-	-97	-	dBm
Maximum received signal @30.8% PER	-	0	-	-	dBm
Co-channel C/I	-	-	+10	-	dB
Adjacent channel selectivity C/I	F = F0 + 1 MHz	-	-5	-	dB
	F = F0 - 1 MHz	-	-5	-	dB
	F = F0 + 2 MHz	-	-25	-	dB
	F = F0 - 2 MHz	-	-35	-	dB
	F = F0 + 3 MHz	-	-25	-	dB
	F = F0 - 3 MHz	-	-45	-	dB
Out-of-band blocking performance	30 MHz ~ 2000 MHz	-10	-	-	dBm
	2000 MHz ~ 2400 MHz	-27	-	-	dBm
	2500 MHz ~ 3000 MHz	-27	-	-	dBm
	3000 MHz ~ 12.5 GHz	-10	-	-	dBm
Intermodulation	-	-36	-	-	dBm

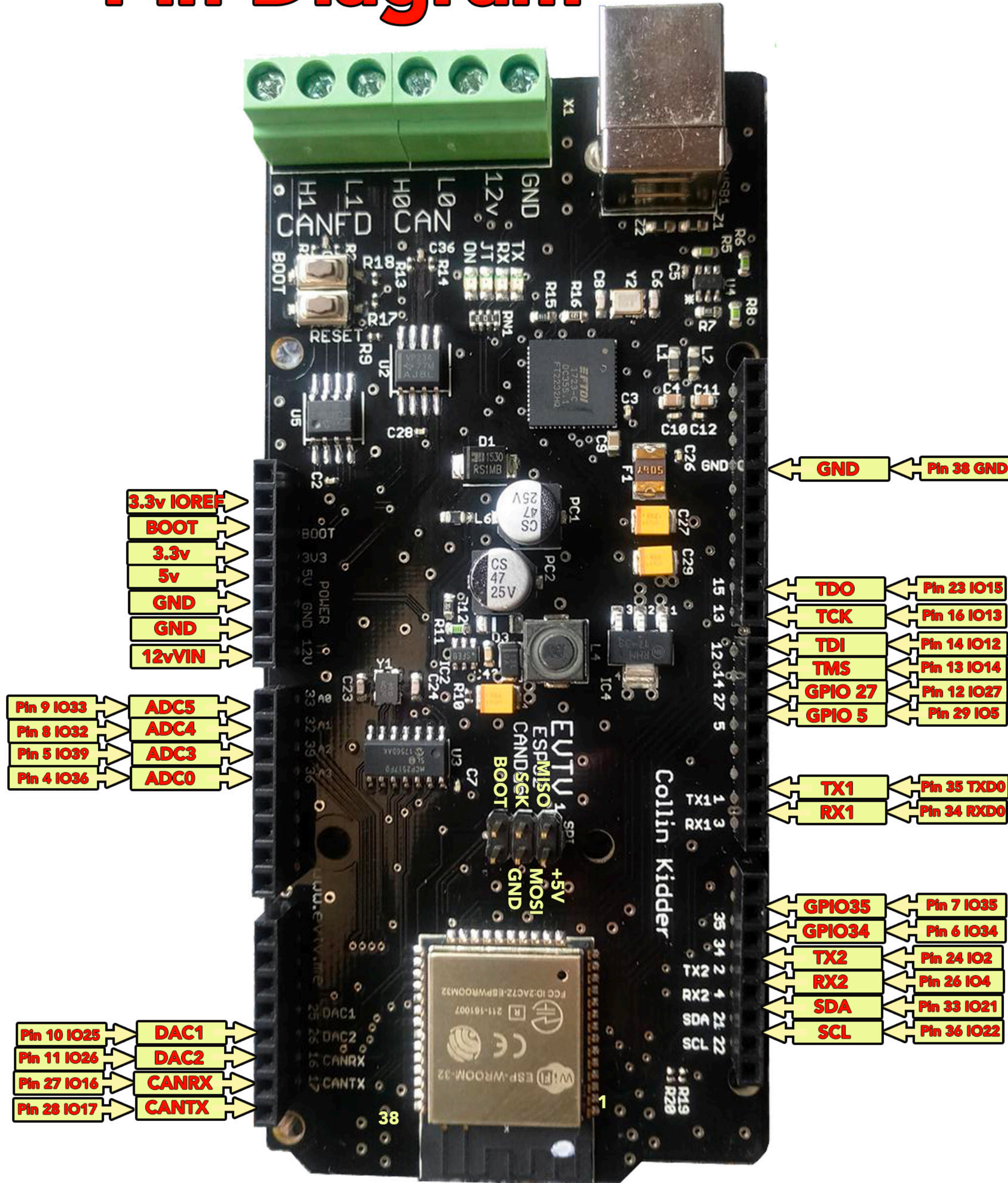
ESP32 Bluetooth Low Energy Receiver Specification

Parameter	Conditions	Min	Typ	Max	Unit
Sensitivity @30.8% PER	-	-	-97	-	dBm
Maximum received signal @30.8% PER	-	0	-	-	dBm
Co-channel C/I	-	-	+10	-	dB
Adjacent channel selectivity C/I	F = F0 + 1 MHz	-	-5	-	dB
	F = F0 - 1 MHz	-	-5	-	dB
	F = F0 + 2 MHz	-	-25	-	dB
	F = F0 - 2 MHz	-	-35	-	dB
	F = F0 + 3 MHz	-	-25	-	dB
	F = F0 - 3 MHz	-	-45	-	dB
Out-of-band blocking performance	30 MHz ~ 2000 MHz	-10	-	-	dBm
	2000 MHz ~ 2400 MHz	-27	-	-	dBm
	2500 MHz ~ 3000 MHz	-27	-	-	dBm
	3000 MHz ~ 12.5 GHz	-10	-	-	dBm
Intermodulation	-	-36	-	-	dBm

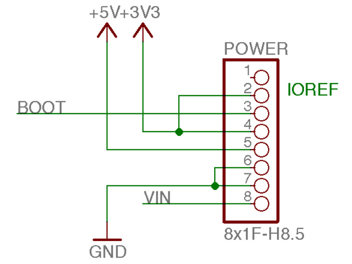
Copyright 2018 EVTV LLC

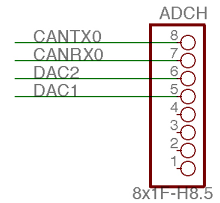

EVTV ESP32 CANDue
ESP32 WROOM-32
Serial Peripheral Interface SPI Port
MCP2517FD CAN Controller
CAN0 Transciever
CAN1 Transciever
RESET SWITCH
BOOT SWITCH
USB Type B
GND
12v
CAN0 LO
CAN0 HI
CAN1 LO
CAN1 HI

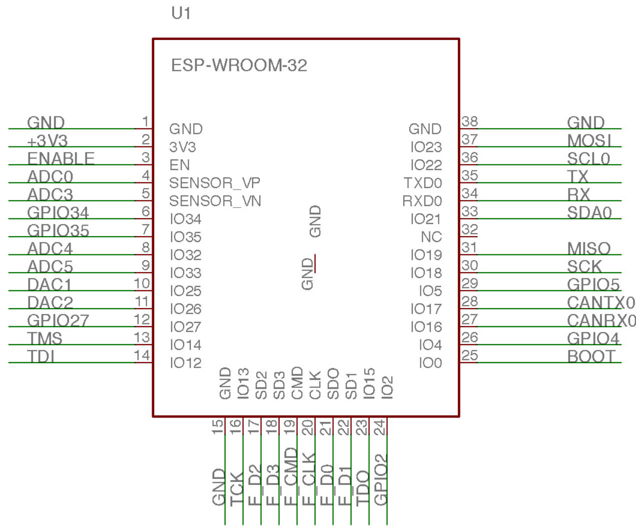
EVTV ESP32 CANDue Pin Diagram

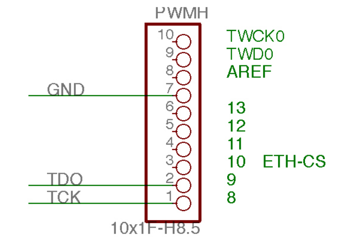


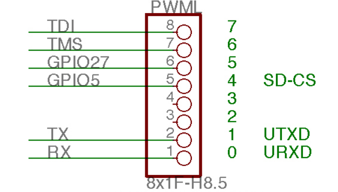
EVTV ESP32 CANDue Processor

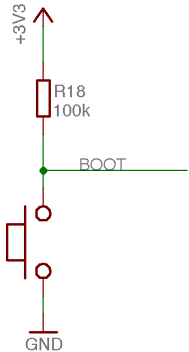


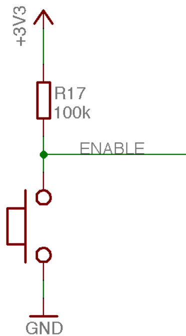


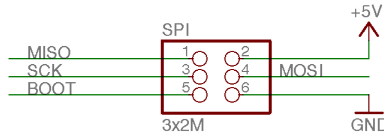


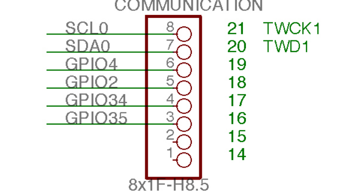


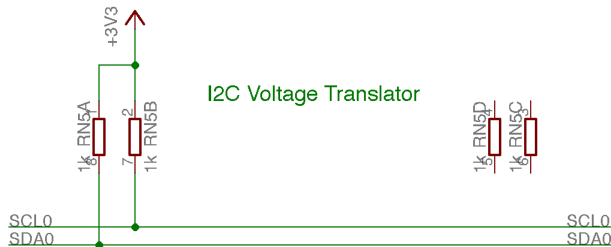


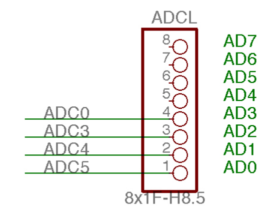










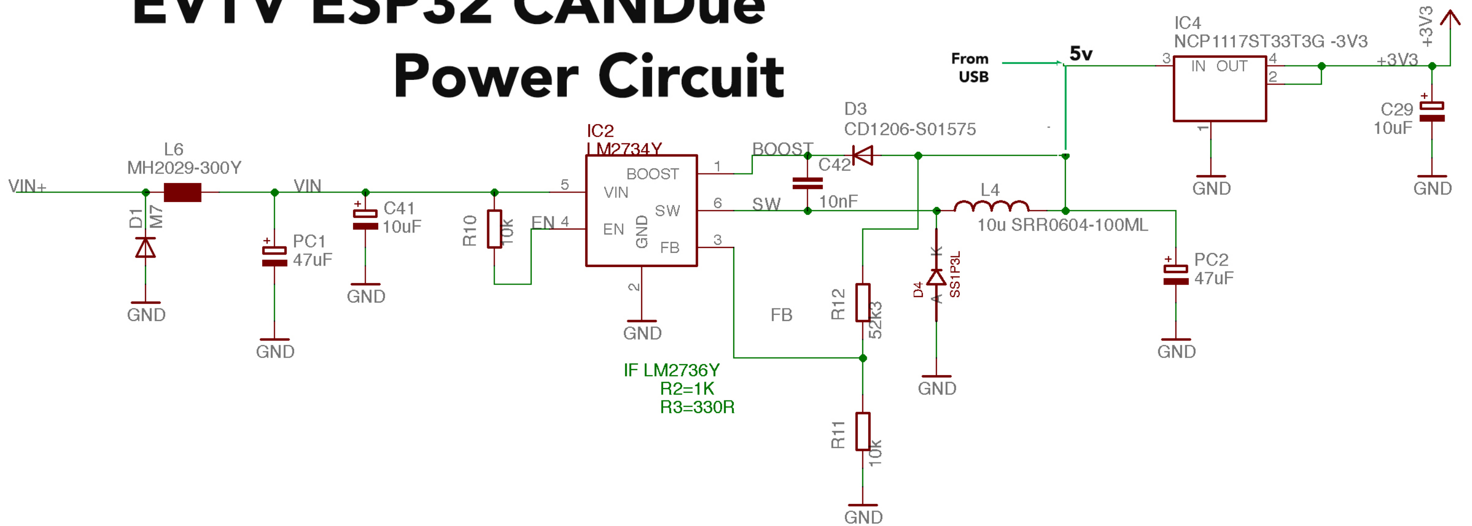


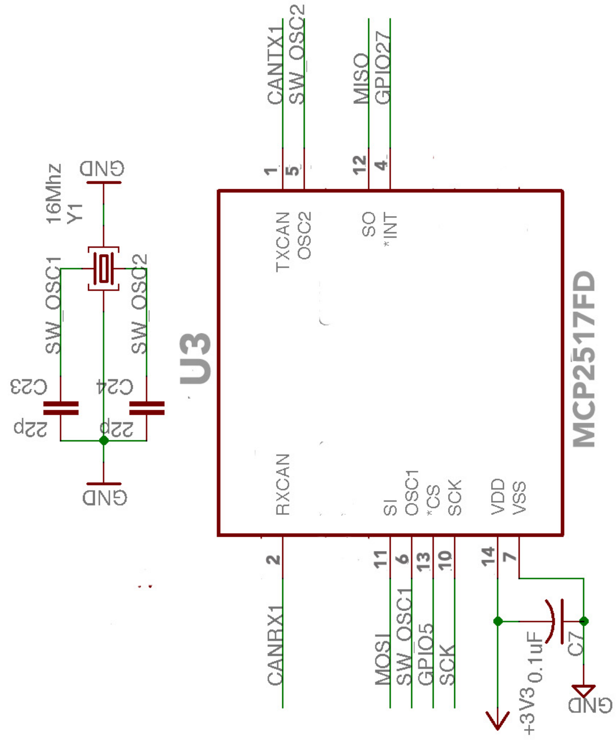
EVTV ESP32 CANDue USB Communications

USB AB

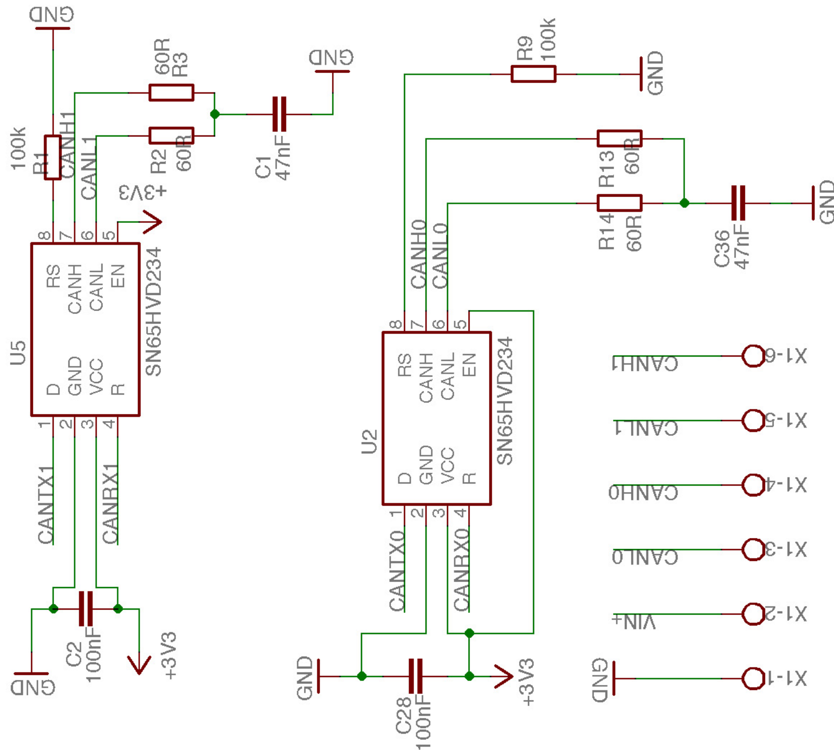
U2 FT2232H

EVTV ESP32 CANDue Power Circuit





EVTV ESP32 CANDue CAN Circuit



Copyright 2018 EVTV LLC

USB COMMUNICATIONS CIRCUIT

The ESP32 module by convention uses two general purpose input/output (GPIO) pins, GPIO 5 (pin 34) and GPIO 18 (35) as a Universal Asynchronous Receiver Transmitter (UART) serial interface. Pin 35 is used as the TX or transmit line while pin 34 is used to receive (RX).

A Future Technology Devices Internal FT2232H Dual High Speed USB chip is used to translate this UART port to the more commonly available and usable Universal Serial Bus or USB port. The EVTV ESP32 CANDue provides this on a sturdy USB mini-B printer port style connector we simply find more physically robust than the tiny USB ports that have become popular.

The FT2232H is FTDI's 5th generation USB chip and supports USB 2.0 High Speed at up to 480 Mbps.

CAN CIRCUIT

The ESP32 chip features an integral CAN controller. This connection appears on pin 27 (**CANRX0**) and pin 28 **CANTX0** of the module.

The module does not feature the actual CAN transceiver necessary to transmit and receive CAN level signals. A Texas Instruments SN65HVD234 transceiver chip U2 translates **CANTX0** and **CANRX0** to the appropriate differential **CAN0 HI** and **CAN0 LO** signal outputs available at the screw terminal connector X1 as **H0** and **L0**. These outputs are 120ohm terminated and filtered to ground. The transceiver **ENABLE** signal is tied directly to +3.3v holding it enabled at all times.

To provide a second CAN channel, a Microchip MCP2517FD CAN controller chip is connected to the processor via the Serial Peripheral Interface bus (SPI) using the usual MOSI (processor pin 37), SCK (30) and MISO(31) connections available on the processor module. Note that this SPI bus is also made available for other addressable devices via a six pin connector.

This controller produces the **CANRX1** and **CANTX1** logic outputs that are used by a MCP2562FD transceiver chip, U5, to produce the second **CAN1 HI** and **CAN1 LO** signals provided as **H1** and **L1** at the X1 connector identically to the **CAN0** bus.

This MCP2517FD chip includes the latest Flexible Data Rate (FD) protocol in addition to the normal CAN2.0B. This allows higher data rates on the CAN bus and most automobile manufacturers are expected to adopt FD in the next two years.

Better, this chip provides for up to 32 filters and masks on incoming CAN messages, which can make programming CAN much easier and more flexible.

Copyright 2018 EVTV LLC

Finally, the MCP2517FD allows CAN frames to have up to 64 bytes of data payload, eight times the data in a normal CAN2.0B frame.

The built-in CAN module (CAN0) in the ESP32 is not CAN-FD compatible so only CAN1 should be connected to such busses.

CAN PROGRAMMING

The EVTV CANdue Microcontroller is in all respects program compatible with the Arduino Integrated Design Environment.

To select the board, in the Arduino IDE simply select

TOOLS

BOARD

ESP32 Dev Module

In your program, you will want to include the ESP32_CAN library using the statement:

```
#include "esp32_can.h”
```

See the examples included with the **esp32_can** library for further instructions on use.

ESP32_CAN

esp32__can is a board specific library written by Collin Kidder of Sparta Michigan. The latest version is always available at http://github.com/collin80. It provides functions and methods to easily deal with the very powerful CAN bus transceiver functions available on the EVTV ESP32 CANDue Microcontroller, including the improved capabilities of the MicroChip MCP2517FD chip.

The Arduino IDE (Integrated Design Environment) provides a basic C++ programming language syntax with some curious “extensions” to deal with hardware easily and directly.

Copyright 2018 EVTV LLC

These extensions are essentially hardware specific and deal with things like digital input and output pins, analog to digital conversion pins, and pulse width modulated output pins.

And so, both the Arduino hardware and the Arduino IDE "language" are curiously adapted to dealing with hardware and sensors and the outside world – lights, switches, potentiometers, temperature sensors, etc.

The hardware of the Arduino is endlessly extensible by the addition of "shields". Shields are printed circuit boards with additional hardware that can basically "plug in" to the headers on the main Arduino Due board and so connect to it.

In theory, the manipulation of pins and data on them will of course operate any hardware provided on shields.

As Yogi Berra says, in theory, theory and practice are the same, but in practice, they aren't. Much hardware has very involved data schemes to either send data to it or retrieve data from it.

And so we find the language of Arduino is ALSO extensible. We do this with "libraries" that basically provide new C++ CLASS structures and methods that act to extend the language. And these hide most of the detail of dealing with the hardware device, reducing it to an OBJECT you can easily command with much simpler instructions.

To add a library, you usually simply add it to your users/Arduino/Libraries directory. But in each program where you use that library, you must also add an "include" statement. This causes the library to be included when the program is compiled, and calls to functions and methods in that library are then linked into the resulting program.

```
#include<esp32_can>; //This is an include statement
```

And so you may have MANY libraries in your users/Arduino/libraries directory, but only those specified with include statements will be compiled into any particular program.

CAN PORTS

The EVTV ESP32 CANDue Microcontroller offers two CAN ports designated **CAN0** and **CAN1.**

INITIALIZATIONS AND BEGINNINGS

So our first requirement is an include statement.

Copyright 2018 EVTV LLC

```
#include<esp32_can>; //This is an include statement
```

Using this include file will automatically create **CAN0** and **CAN1** objects for you, ready to setup for CAN bus access. Our second is to INITIALIZE the CAN port we want to use with a **begin** statement. We can do either or both.

```
if (CAN0.begin(500000))
  {
  Serial.println("Using CAN0 – initialization completed.\n");
  }
 else Serial.println("CAN0 initialization (sync) ERROR\n");
```

The basic initialization is handled by

```
CAN0.begin(500000)
```

Note that `500000` is the data rate 500kbps

If Can0 initialization is successful, the `CAN0.begin` routine will return a value of **1** which is Boolean **TRUE**.

As you can see above, we used this feature to determine whether CAN initialization was successful and send a message out the Serial port to display on screen.

Finally, we need to establish some very specific variables to handle CAN frame data that we want to send and CAN frame data that we will be receiving. These variables are of type **CAN_FRAME**.

```
CAN_FRAME outFrame,  inFrame;
```

CAN_FRAME DATA TYPE STRUCTURE

In this example, we have set two variables, **outFrame** and **inFrame** of the **CAN_FRAME** type.

CAN_FRAME is actually a STRUCTURE – a variable form that contains a number of other variables within its structural envelope. To send CAN data, we have to populate one of these frames with our data we want to send (**outFrame**). And to receive CAN data, we have to have this variable structure available to store the received data (**inFrame**).

The CAN protocol has a number of options but is basically pretty simple. Some housekeeping data up front including the message ID, how long the data payload is, and some other incidentals, and then a data payload of 0 to 8 bytes. The **CAN1** object is CAN-FD compatible. CAN-FD changes the way CAN works a bit. Instead of 0-8 data bytes it is possible to send 0-64 bytes. This will be covered later on.

The easiest way to deal with that is set up a structure with ALL of the elements possible in a CAN message frame. You fill in the ones you need and once you have everything you want accounted for - then send the frame with one call. It's kind of like filling out a message form before handing it to the telegrapher to send. Here are the raw structures used:

```
typedef union {
    uint64_t int64;
    uint32_t int32[2];
    uint16_t int16[4];
    uint8_t int8[8];
} BytesUnion;

typedef struct
{
    uint32_t id; // 29 bit if ide set, 11 bit otherwise
    uint32_t fid; // family ID - used internally to library
    uint8_t rtr; // Remote Transmission Request (1 = RTR, 0 = data frame)
    uint8_t priority; // Priority but only for TX frames and optional (0-31)
    uint8_t extended; // Extended ID flag
    uint32_t time; // CAN timer value when mailbox message was received.
    uint8_t length; // Number of data bytes
    BytesUnion data; // 64 bytes - lots of ways to access it.
} CAN_FRAME;
```

And so we find that **outFrame** actually has a number of structural elements.

outFrame.id - 32 bit variable containing the CAN message ID. This can be either 11 bits or 29 bits long. With two important uses. The ID is used to create a transmit priority on the CAN bus. When multiple devices on the bus try to send at once the lowest ID wins automatically. This allows frames with low IDs to break through congestion and be delivered anyway.

Copyright 2018 EVTV LLC

`The second important usage is that the ID is used by receiving devices to determine the type of data in the frame. For instance, a motor controller might know that messages with ID 0x232 are meant to tell it the requested torque and RPM.`

`outFrame.fid` –32-bit variable for family ID. Used internally by the CAN library so it is best not to put anything into this field. The value in the field is not of much use outside the internal code of the library.

`outFrame.rtr` – 8-bit variable – Remote Transmission Request. RTR has been deprecated for some time but there are still some rare devices that use it. The purpose of RTR is to signal another device on the bus that we'd like to receive some piece of information. We might send an RTR frame with ID 0x123 to some other device. When the device sees this RTR frame it will then send back a certain group of data. But, it is not commonly used and should be set to 0 unless you're absolutely sure you want to use it. Setting RTR will zero out all of your data bytes when sending no matter what you set them to.

`outFrame.priority` – (1 byte) Transmit priority from 0 to 31. It is not commonly used and should not be set unless there are specific requirements. Messages with a higher transmit priority will be sent before messages with a lower priority. This can somewhat short circuit the usual ID based priority scheme on the bus. In most all cases this field can be left at its default value.

`outFrame.extended` – (1 byte) Extended Addressing (29-bit) 0 = 11-bit/1=29bit

`outFrame.time` – (4 bytes) – Time stamp set by hardware when this frame came in (received frames)

`outFrame.length` - (1 byte) The number of data bytes in this frame. Ranges from 0 – 8. You can send frames with no data bytes if desired. Setting this field any higher than 8 will cause the value to be set to 8 when sending.

`outFrame.data` – (8 bytes) A union that is 8 bytes long. Allows for retrieval of data bytes from received frames and setting bytes to send for transmitted frames. A union is a way to access the same bytes in different ways. This particular union allows for setting and retrieving the 8 bytes individually, in groups of 2, 4 or all 8 at once. It should be noted that not all frames are sent or received with 8 data bytes. Setting bytes past the length you've specified when sending will just be ignored. Any bytes past the length in a received frame will be 0. All multi-byte values are stored little endian. There are two ways to store a value that takes multiple bytes to store. One can store the value such that the first byte is the lowest part of the value or such that the first byte is the highest part of the value. For instance, take the value 0x1234. This value takes 2 bytes to store: 0x12 and 0x34. We can store it in memory as 0x12 followed by 0x34 or we can store it as 0x34 followed by 0x12. The first method is called big endian while the second method is called little endian and is the way this library and the ESP32 works.

Copyright 2018 EVTV LLC

outFrame.data.int64[0] – Allows for getting or setting all 8 possible bytes at once. This can be used to save or load all the bytes in a single line. It is called uint64 because it is an unsigned integer 64 bits (8 bytes) wide.

Example: **outFrame.data.int64[0] = 0x1122334455667788**;

It might be counter intuitive but because the value is stored little endian you will find that the above line sets the first byte to 0x88, the second to 0x77, and so on until the 8th byte is set to 0x11.

outFrame.data.int32[2] – Allows one to get or set the 8 bytes in groups of 4 bytes. That is, **outFrame.data.int32[0]** is bytes 0, 1, 2, 3 and **outFrame.data.int32[1]** is bytes 4, 5, 6, 7.

outFrame.data.int16[4] – Allows one to get or set the 8 bytes in groups of 2 bytes per entry. **outFrame.data.int16[0]** is bytes 0 and 1, **outFrame.data.int16[3]** is bytes 6 and 7.

outFrame.data.int8[8] – Allows for direct access to all of the bytes individually.

inFrame has exactly the same structure, but of course in a different location in memory.

Note that the CAN message structure simply contains the bytes. The software on either end of the transmission determines whether these bytes are signed or unsigned, and whether they use Least Significant Bit first or Most Significant Bit first and actually what to do with them.

One can cast the unsigned values from **CAN_FRAME** to signed types if necessary:

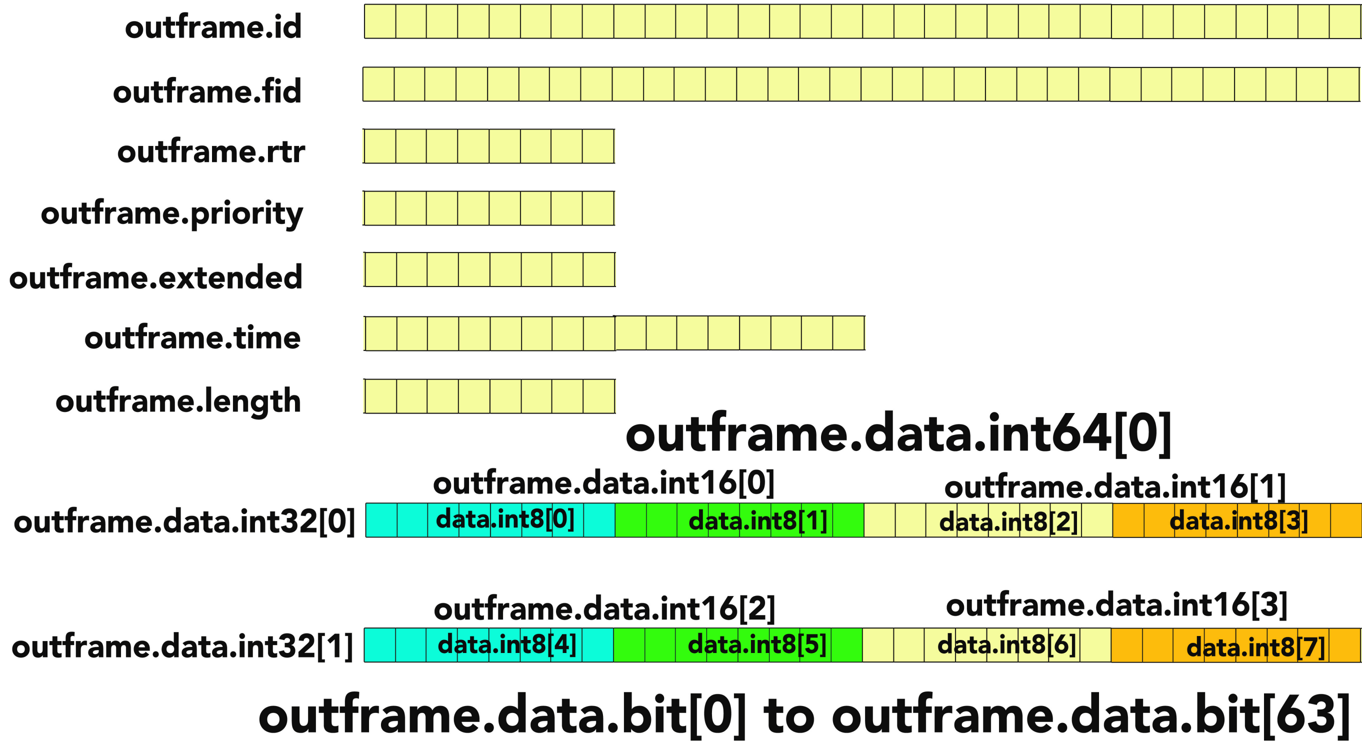
```
int myVariable = (int16_t)outFrame.data.int16[0];
```

When we define a variable as type **CAN_FRAME**, it simply reserves bytes in memory under that name (**inFrame**)to hold the frame and allows access by use of the frame subunits (**inFrame.id, inFrame.data.int8[5]**). You can define as many **CAN_FRAME** variables as you like.

Copyright 2018 EVTV LLC

ESP32_CAN DATA STRUCTURE

CAN_FRAME outframe



SENDING CAN FRAMES

Sending CAN frames is actually very simple. As described, we load the CAN frame we want to send, and then send it with a single call.

Copyright 2018 EVTV LLC

```
CAN_FRAME myFrame;
myFrame.id = 0x25A;
myFrame.length = 1;
myFrame.data.int8[0] = 128;
CAN0.sendFrame(myFrame);
```

Here we define variable **myFrame** as type **CAN_FRAME**.

We then set the id to **0x25A**. By convention, and you'll rarely see this otherwise, CAN message IDs are almost always referred to in the hexadecimal numbering system. We note this by prepending **0x** to the number.

We also set **myFrame.length** to 1 indicating that we will only be transmitting a single data byte.

We define that data byte as containing the decimal value 128.

And finally, we send the frame with the statement **CAN0.sendFrame(myFrame);**

We're basically telling the **CAN0** hardware port to make up a CAN message from the data in the **myFrame** structured variable, and send it out on the bus.

Note that there are a number of elements defined in the structure **myFrame** that we didn't set at all. The defaults, usually zero, will be fine. In the majority of cases, all you need is an ID, a data length and your data.

In fact, in this case, to send another frame with new data, I know that the id and length are already set.

```
myFrame.data.int8[0] = 129;
CAN0.sendFrame(myFrame);
```

And so we see that we have sent an entire new frame with just the data byte changing.

Copyright 2018 EVTV LLC

RECEIVING BUFFERED CAN FRAMES

We can receive CAN frames in two ways. The CAN library actually buffers incoming frames which we can check for or poll using the available command.

We can also set up a callback such that any time a frame is received, it is routed to a method in our program that is designed to handle the incoming frame. This will be described later.

```
CAN_FRAME inFrame;

if (CAN0.available())

{

    CAN0.read(inFrame);

}
```

We would place this call somewhere in our LOOP portion of the Arduino program. Periodically, it would poll for `CAN0.available()` which would return `1` if true and `0` otherwise.

If there is a frame available, the `CAN0.read` command retrieves it and loads all the data into our already defined `inFrame` structure. And so `CAN0.read` actually passes the address of the `inFrame` structure to the object which then uses that address to copy data out of the buffer and into the `inFrame` structured variable.

Once we have received the data, we can then go examine it in the `inFrame` structure.

CAN FRAME FILTERS

CAN can support a number of devices in theory. In practice, above about 30 devices and the bus becomes quite busy. But the central tenet of CAN is that there really is no intelligence in the protocol. The messages aren't even addressed to any specific device. Each device simply broadcasts their messages to everyone on the bus. No checking to see if it was received. No handshakes. Nothing.

The intelligence is supposed to be in the devices themselves. The DMOC645 controller for example, knows that a torque command will be received under message ID `0x232` and that the first two bytes will contain the command as a 16-bit unsigned integer that is offset by 30000. So it receives the value, subtracts 30000, and takes the result as a torque command.

Copyright 2018 EVTV LLC

The Vehicle Control Unit broadcasts this torque command and sets the message ID to **0x232**. It doesn't know if there are any DMOC645's on the bus, doesn't know how many, and doesn't know or care what it does with it. It simply translates throttle inputs to a torque command, adds 30000 to it, and puts it in the first two bytes of the payload.

So what does the DMOC645 device do when it receives a CAN message with ID **0x332**? Nothing. It has no knowledge of **0x332** messages so it simply discards them. Actually, it is worse than that. Since it has no knowledge of **0x332** messages, it actually sets a filter in the CAN transceiver chip to not even interrupt it for **0x332** messages. As a result, it never receives them at all. The internal CAN transceiver receives it, and simply ignores it – dramatically reducing the computational overhead for the DMOC645 multicontroller.

So picture the DMOC645 as actually LOOKING for **0x232** messages, and totally ignoring, in fact filtering out, all **0x332** messages.

In this way, each device on the bus has a list of messages it sends, and the data it wants to send in them. And it also has a list of messages it will receive, and how to deal with data in those. And typically any specific device might have 3-5 messages it sends, and another 3 or 4 it responds to. ALL OTHER TRAFFIC IS TOTALLY IGNORED.

Some devices broadcast a single message id with specific information in it and don't listen for ANY messages incoming. This would be like a temperature sensor. It only does one thing – measure temperature. And it reports it on the CAN bus for any who care. But it doesn't DO anything else, and doesn't need information from any other device at all.

esp32_can features some powerful filtering options. If you want to monitor all the traffic on a CAN bus, obviously you don't want to filter out anything. But for most applications, you are looking for a relative handful of messages, and it is an enormous reduction in processor overhead to set filters to ignore everything else. In fact, both **Can0** and **Can1** default to accepting NO frames. You must specify the frames you are interested in before you will receive any. If you do not want to use filtering you can use the **watchFor();** method to accept all frames like so:

```
CAN0.watchFor(); //Accept any frame
```

In order to accept only specific sets of frames use the **setRXFilter** command.

```
CAN0.setRXFilter(msgid, mask, extended);
```

This command takes three arguments (with an optional fourth). The first is the message id of the messages you WANT to receive. The second element is the MASK and the third indicates whether this is for standard 11-bit message ids (false) or extended 29-bit message ids (true).

Copyright 2018 EVTV LLC

CAN0.setRXFilter(0x232, 0x7FF, false);

In this example, the message ID included is **0x232**. The third element is **false** indicating standard 11-bit addresses are used.

The MASK in this case indicates that we want to receive ONLY messages that exactly match the **0x232** message ID

Our mask, **0x7FF** would be represented in binary as **0111 1111 1111**

Note that 11 of 12 bits are set. Our standard message IDs are limited to 11 bits and so all messages must be numbered in the range **000** to **7FF**. Think of the mask as defining the specific bits that MUST MATCH. Anywhere there is a '1' in the bits of the mask the corresponding bit in the filter ID must match the corresponding bit in the incoming frame ID. And so this mask indicates that all 11 bits must match for a message to be valid.

If we set a mask of **0x7F0** **0111 1111 0000**
And our base message is **0x230** **0010 0011 0000**

We logically AND those two to get the result **0010 0011 0000**

If we receive a message of **0x23A** **0010 0011 1010**
And we logically AND that with the mask to get the result **0010 0011 0000**

We see that the results of the FIRST AND and the results of the SECOND AND are equal and we accept the message.

Our mask indicates that the last four bits do NOT have to match but the first 7 DO. This would accept any number from **0x230** to **0x23F**.

If we set the mask to **0x700** **0111 0000 0000**

Our mask indicates that only the first three bits need match. We can accept any message ID from **0x200** to **0x2FF.**

And of course a mask of **0x000** would accept all messages. It would be pointless to set such a filter.

CAN1.setRXFilter(0x18FF50e5, 0x1FFFFFFF, true);

In this filter example, we use 29-bit extended addresses. The mask indicates that we must have an exact match on all bits for messages with **ID 0x18FF50e5**

Note that you can set up to 32 different filters covering 32 different message ranges and any can be either 11-bit or 29-bit.

EASY CAN FILTERS

CAN filters can be somewhat easier to employ using the **watchFor** functions.

CAN0.watchFor();

As previously mentioned, this command allows ALL messages to be accepted. But better, it actually sets up one mailbox for standard frames, and a second mailbox for extended frames. All messages of either standard or extended frame are then accepted. It is valid to call this command after having already set up other filters. In that case those other filters will selectively accept some frames and this call will cause every other frame to get accepted in one giant group. The reasons to do this will become more clear when we cover callbacks.

CAN0.watchFor(0x740);

This command will cause CAN0 to watch for a specific address, in this case, **0x740.** Whether it is extended or standard addressing is set automatically depending on the address provided. This will use one of the 32 filters possible.

CAN0.watchFor(0x620, 0x7F0);

This command specifies a message **0x620** and a mask **0x7F0**. It applies the mask just as described earlier to accept all messages from **0x620** through **0x62F.**

CAN0.watchForRange(0x620, 0x64F);

This command accepts messages by message ID in the range from the first message ID given to the second. It will attempt to create a mask and ID suitable for this. This process may not be exact as the mask can only work in binary. The resultant filter might allow a larger range through than you specified. For exact filtering you should use one of the methods that allow for explicitly specifying the mask and ID. But, this form of the method can be useful when getting started in order to not have to deal with determining masks and filter IDs yourself.

Copyright 2018 EVTV LLC

CAN CALLBACKS

An earlier description provided the details of buffered CAN frames and how to retrieve them.

There is a second way to receive CAN frames that many find more efficient. This is through CAN callbacks. They are also called interrupts as they are executed based on hardware interrupts and they can interrupt the execution of your program. On the ESP32 they are not truly called from a hardware interrupt but they will still cause the ESP32 to immediately execute your callback function. This will temporarily interrupt the rest of your program. Its current state will be automatically saved and restored for you as this happens.

CAN callbacks are simply a means of calling a processing method in your program, when and only when a CAN frame is received. In this way, your program can attend to other duties without the overhead of checking to see if a CAN frame has come in.

When a valid message DOES arrive and qualifies through the filters set, the CAN object calls the defined method in your program and passes it the CAN frame. It can then process the CAN frame and return.

As the normal program loop can cycle hundreds of thousands of times per second, this vastly reduces the overhead of CAN messages, which might only be received 30 times per second.

Better, you can have different CAN handler methods for different received message IDs.

```
CAN0.setRXFilter(1, 0x18FF50e5, 0x1FFFFFFF, true);

CAN0.setCallback(1, convertIncoming);
```

In the first line above, we see our familiar filter statement. But we have a new element, the first, set to 1. When you set a filter, the library normally picks the first free mailbox for you. There are 32 "mailboxes" provisioned as a function of the chip design and this is why you can have up to 32 filters.

But you can optionally designate a specific mailbox to use 0-31. We can use this to set our filter, and then tie the output to a given routine in our program.

The second line introduces a new function, `setCallback.` This function then lets us tie any valid message filtered through mailbox 1 to be routed to our routine in our program that handles this.

Copyright 2018 EVTV LLC

In our Arduino program structure, you always have a setup routine and a loop routine.

```
void setup(){

Some setup statements.…

}


void loop () {

     Some statements we execute over and over without end.

}
```

We want to add a method to our program to handle this CAN message, but we do NOT want it to be part of the main program loop or the setup.

```
byte fromCharger[15];

void convertIncoming(CAN_FRAME *frame){

    fromCharger[3]=(uint8_t) (frame->id>>24);

    fromCharger[2]=(uint8_t) (frame->id>>16);

    fromCharger[1]=(uint8_t) (frame->id>>8);

    fromCharger[0]=(uint8_t) (frame->id>>0);

    for(int i=4; i<12; i++){

        fromCharger[i]=frame->data.int8[i-4];

    }

    calculateCharger();

}
```

The library passes the CAN message data structure to the `convertIncoming` method as `frame`. In this method, we are extracting information from the frame and arranging it in a 15 byte array titled `fromCharger` and then calling ANOTHER method, `calculateCharger`, which has access to `fromCharger` as well.

We can set up to 32 callbacks and each can be to the same method, or other entirely different methods in our program, all based on their incoming addresses.

Copyright 2018 EVTV LLC

Finally, we can set a GENERAL callback to handle all mailboxes that do not otherwise have a callback associated with them.

```
CAN0.setGeneralCallback(someOtherMethod);
```

It will be applied to any mailboxes that do not have an interrupt attached. This might be useful in cases where you've set up special callbacks for important frames but you want to accept everything else on the bus and peek at all the traffic.

This combination is actually quite powerful. For example, we could set filters and interrupts for two named mailboxes, and then set six more filters that don't specify a mailbox. Then we can set one interrupt for the first mailbox, another interrupt for the second mailbox, and then a general interrupt to handle the remaining six.

In this way, we can route messages to specific methods based on their message ID. The methods are ONLY called when a specific message is received. Once processed, control is returned to the overall general Arduino program loop.

CAN interrupts can be removed with the command

```
CAN0.removeCallback(0);
```

This would remove the interrupt attached to mailbox 0. If the mailbox is omitted,

`CAN0.removeCallback();` will remove ALL `Can0` interrupts.

`CAN0.removeGeneralCallback();` **`will remove the general callback.`**

This section provides descriptions of the basic functions of due_can and these are certainly sufficient to write powerful CAN programs. But there are many more functions in the library. Refer to the library source code and example programs for more detailed information.

PUTTING IT ALL TOGETHER – A CAN EXAMPLE

Let's put all this new CAN knowledge together in a simple but tricky example. We have two Arduino Due's that we start up at two different times. The millis() method will give us the number of milliseconds that have occurred since we started the machine.

In this case, we want to display the time in hours minutes and seconds on BOTH Arduino Due's Basically we want to synchronize our watches via CAN.

On the first Arduino Due:

```
#include <esp32_can>
#define Serial SerialUSB
CAN_FRAME outFrame;
uint8_t correction; //8bit integer holding a correction

void setup(){  //Our setup function
    Serial.begin(115000);
    If (CAN0.begin(500000);){
        Serial.println("Can0 initialized…");
    }
    else Serial.println("Can0 failed…");

    OutFrame.id = 0x05B; //Let's use message ID 0x05B
    correction=5; //5 milliseconds for propogation
}
```

Copyright 2018 EVTV LLC

```
void loop(){
    outFrame.data.int32[0] = millis() + correction;
    CAN0.send(outFrame);
    int seconds = (int)(millis()/1000)%60;
    int minutes = (int)((millis()/(1000*60))%60);
    int hours = (int)((millis()/(1000*60*60))%24);
    char buffer[9];

sprintf(buffer,"%02d:%02d:%02d.%03d",hours,minutes,seconds,milli
seconds);
    Serial.println(buffer);
}
```

This program sets up **CAN0** as our output port at a data rate of 500KBPS.
In the main loop, it prints the current time since startup in hours, minutes and seconds since startup out the serial port and transmits a four byte value representing current milliseconds plus a correction value of 5 milliseconds to account for the propagation delay in sending our time over the bus. And it does this in the standard Arduino Due little endian format – storing this value in the first four bytes using message address 0x05B.

On the second Arduino Due device:

Copyright 2018 EVTV LLC

```
#include<esp32_can>;


void setup(){  //Our setup function
    Serial.begin(115000);
    If (CAN0.begin(500000){
        Serial.println("Can0 initialized…");
    }
    else Serial.println("Can0 failed…");
    CAN0.setRXFilter(1, 0x05B, 0x7FF, false);
    CAN0.setCallback(1, getOurTime);
}
void loop(){
}
void getOurTime (CAN_FRAME *timeFrame){
    long remillis=timeFrame.data.int32[0];
    int milliseconds = (int(remillis/1)%1000 ;
    int seconds = (int)(remillis/1000)%60;
    int minutes = (int)((remillis/(1000*60))%60);
    int hours = (int)((remillis/(1000*60*60))%24);
    char buffer[9];

sprintf(buffer,"%02d:%02d:%02d.%03d",hours,minutes,seconds,milli
seconds);
    Serial.println(buffer);
}
```

This program is a little bit different. First, we are going to set a filter on mailbox 1 that ONLY responds to `0x05B` message IDs.

Copyright 2018 EVTV LLC

The CAN bus has little intelligence, but it DOES work out bus contention on different devices. Lower ID numbers correspond to higher priority. If two devices on the bus want to both send at the same time the one sending a frame with the lower ID number will win. We are only going to transmit this time mark once per second, and we are going to correct it with a calibration factor accounting for propagation delay. But we DO want that message to get through on time. And so by setting a LOW message address ID number, we give it a higher priority on the bus. In this way, as we add devices on the bus with higher addresses, our delay should not change much even though the traffic on the bus grows enormously.

Second, we set a callback for Can0 that passes received frames to the function **`getOurTime`**. This function receives the frame from the library and note that we do not really explicitly declare the **`timeFrame`** variable elsewhere.

`getOurTime` sets a local **`remillis`** variable to **`timeFrame.data.int32[0]`**. Recall that **`.int32[0]`** is an alternate data structure representing the first four bytes in a data structure of up to 8 bytes. Since millis() returns a long integer of four bytes, in little endian format, this is perfect. We can simply copy this value to the **`remillis`** variable on the second machine.

Finally, our interrupt function prints the new formatted hours, minutes and seconds out the serial port.

If we set up these two ESP32s connected to two laptops displaying the USB output on respective terminal programs, the objective is for the printed times to match. If they do not, we can correct by going back to the first program and changing the **`correct`** variable value from 5 to something else and in this way synchronizing the two systems.

Note that in this case, the standard Arduino loop function contains no code. In fact, you can place 12000 lines of code in this loop and have the program do whatever you like. It will have no effect on our time function at all. This is because that loop program iteration will be interrupted on receipt of a **`0x05B`** time message and the time printed again. Once that very brief operation has concluded, the program code in loop will continue from exactly where it left off when interrupted.

CAN FLEXIBLE DATA RATE

Up to this point **`Can0`** and **`Can1`** have been interchangeable. Any of the previous examples using **`Can0`** could have just as easily used **`Can1`** and they'd have worked the exact same way. However, Can1 has a powerful secret – it also is able to go into CAN-FD mode. So, what is CAN-FD and how is it different from regular CAN? The biggest difference is that CAN-FD allows for up to 64 data bytes instead of 8. This can drastically cut down on the number of frames necessary to send large amounts of data over the CAN bus. One very compelling application is firmware updates over CAN. CAN-FD does this by living up to the

Copyright 2018 EVTV LLC

extra 2 letters on the end. FD stands for "flexible datarate." CAN-FD can send the data bytes at a different speed than the rest of the frame (the ID, length, etc). The CAN bus might be running at a speed of 500k but a CAN-FD device could send the data bytes at up to 8,000,000 bits per second instead. It will still always send the rest of the frame at the original speed, only the data bytes are sped up. You might wonder, what happens to regular CAN devices if a CAN-FD device starts to transmit at 8M bits per second? Well, they fault. You cannot send any CAN-FD frames on a CAN bus that has any devices that are not CAN-FD compatible. Even one incompatible device will cause the bus to go into a fault state and not operate properly just as soon as a CAN-FD frame is sent. But, CAN-FD compatible devices can exist on a bus full of regular CAN devices – it just cannot send any CAN-FD frames while on the bus. A CAN-FD device can still send regular CAN frames as often as necessary even on a bus with devices that are not CAN-FD compatible. The two types of frames are exclusive and a CAN-FD device can send and receive either type.

At the moment CAN-FD is very new and very rare. However, projections are that CAN-FD will be in all major manufacturer's cars by 2020. So, the future of CAN-FD is not very far off. The EVTV ESP32 CANDue board is compatible with this upcoming bus type and is ready when CAN-FD compatible vehicles begin to appear.

In order to support the larger frames possible with CAN-FD the ESP32 CAN library for Can1 has a second set of structures for CAN-FD:

```
typedef union {
    uint64_t int64[8];
    uint32_t int32[16];
    uint16_t int16[32];
    uint8_t int8[64];
} BytesUnion_FD;

typedef struct
{
    uint32_t id; // 29 bit if ide set, 11 bit otherwise
    uint32_t fid; // family ID - used internally to library
    uint8_t rrs; // RRS for CAN-FD (optional 12th standard ID bit)
    uint8_t priority; // Priority but only for TX frames and optional (0-
31)
    uint8_t extended; // Extended ID flag
    uint8_t fdMode; // 0 = normal CAN frame, 1 = CAN-FD frame
    uint32_t time; // CAN timer value when mailbox message was received.
    uint8_t length; // Number of data bytes
    BytesUnion_FD data; // 64 bytes - lots of ways to access it.
} CAN_FRAME_FD;
```

These are all mostly like the items previously covered for CAN frames but now the data structures are all eight times larger. There are two other differences:

canFrame.rrs – (1 byte) This is really just a single bit. If set this bit can be added to the top side of an 11 bit ID to make it 12 bits instead. There are NO RTR frames on CAN-FD. You simply cannot send a CAN-FD frame as RTR. You can still send regular CAN frames as RTR even with a CAN-FD device like Can1.

canFrame.fdMode – (1 byte) This bit sets whether the frame is FD mode or not. A CAN-FD device can send either type and you set the type you want to send here. This field is also set when receiving frames to specify whether the frame was FD or not. But, you might figure that out anyway if you see that the frame has specified it has 32 bytes of data.

ANALOG TO DIGITAL INPUTS

The ESP32 integrates two 12-bit SAR (Successive Approximation Register) ADCs supporting a total of 18 measurement channels (analog enabled pins).

The ADC driver API supports ADC1 (8 channels, attached to GPIOs 32 - 39), and ADC2 (10 channels, attached to GPIOs 0, 2, 4, 12 - 15 and 25 - 27). However, the usage of ADC2 has some restrictions for the application. ADC2 is used by the Wi-Fi driver. Therefore the application can only use ADC2 when the Wi-Fi driver has not started.

The EVTV ESP32 CANDue is designed to provide ready access to six of the ADC` analog/digital inputs of the ESP 32 chips:

- AO ADC1_CH5 GPIO 33 PIN 9
- A1 ADC1_CH4 GPIO 32 PIN 8
- A2 ADC1_CH3 GPIO 39 PIN 5
- A3 ADC1_CH0 GPIO 36 PIN 4
- 34 ADC1_CH6 GPIO 34 PIN 6
- 35 ADC1_CH2 GPIO 35 PIN 7

These inputs are successive approximation with a maximum resolution of 12-bit with a maximum sampling rate of 46kHz. They can produce a digital result between 0 and 4096 representing voltages up to 3.3vdc or 0.000805 (805 microvolts) per digit. This range can be expanded to 1.1v full scale for a resolution of 0.000268 (268 microvolts) per digit.

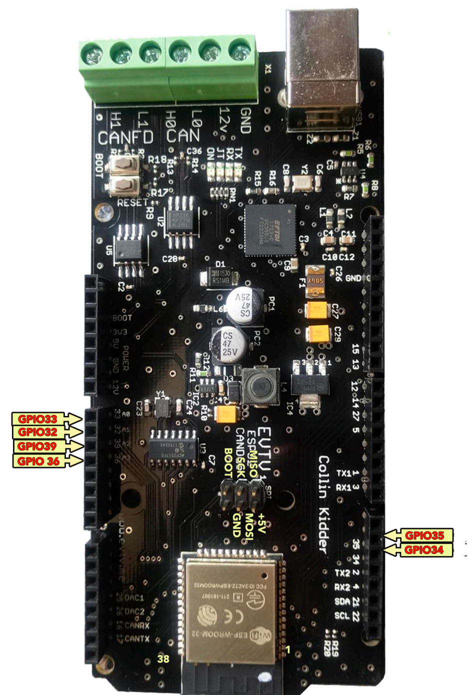
And so for example, if you used external circuitry to isolate and scale 400vdc to 3.3v, you could read that voltage to an accuracy of 100 millivolts.

Copyright 2018 EVTV LLC

Note that most pins on the ESP32 are General Purpose Input and Output (GPIO) and can be configured as inputs or outputs and generally as analog or digital quite flexibly. But while that is the intent, actual practice varies somewhat and these particular pins are configurable for input ONLY. But note that they can be configured for DIGITAL input as easily as analog.

Before you examine the listing above of the six channels, note that you will be tempted to make sense of it and find some rational pattern to it. STOP. DON'T DO THAT. There is nothing rational about it.

EVTV ESP32CANDue Analog Inputs



First, chip makers often offer the same basic chip die and functions on a number of different packages, in different physical sizes and different pin terminations for a variety of reasons including miniaturization and mechanical and robotic assembly. So by all rights a General Puropose Input Output Pin would of course be PIN 33 and we would call it pin 33. But if we offered a smaller package with a reduced number of pins, of course that might be PHYSICAL pin 9 on the chip.

And that is before the software/firmware people ever get ahold of SOME version of the hardware. In this case breaking analog to digital inputs into two banks (ADC1 and ADC2) and seven channels (0-6 which also doesn't make sense but it IS how programmers think).

And THEN you have the case of someone USING the chip on an electronic circuit board and wanting to tie THAT somehow to a pin or interface for the board (in this case designated A0.

And then of course you have the application development software interface, herein where we use `ADC1_CHANNEL_5.` And it is our observation that EVERY attempt to rationalize and simplify this system tends to MAKE IT WORSE.

```
#include <driver/adc.h>
```

This include statement provides the drivers necessary for ADC operations.

```
adc1_config_width(ADC_WIDTH_12Bit);
```

Copyright 2018 EVTV LLC

This statement configures the resolution for ALL 7 channels of ADC1 as 12 bit or 4096 resolution. Note that there is also an ADC2 series of channels, but they are almost never used as they are shared with the WiFi function of the ESP32.

```
adc1_config_channel_atten(ADC1_CHANNEL_5, ADC_ATTEN_11db);
```

When VDD_A is 3.3V:

- 0dB attenuaton (ADC_ATTEN_DB_0) gives full-scale voltage 1.1V
- 2.5dB attenuation (ADC_ATTEN_DB_2_5) gives full-scale voltage 1.5V
- 6dB attenuation (ADC_ATTEN_DB_6) gives full-scale voltage 2.2V
- 11dB attenuation (ADC_ATTEN_DB_11) gives full-scale voltage 3.9V At 11dB attenuation the maximum voltage is limited by VDD_A, not the full scale voltage. So 3.3v

This has the effect of setting the scale or attenuation of a specific channel. With attenuation of 11db you will be able to handle signals of up to 3.3v, the operating voltage of the chip and it will be read in 4096 increments in the 12 bit resolution. Similarly, the least attenuation of 0db would mean that a full scale indication of 4096 would be provided in response to a 1.1v input. And so finer voltage levels can be discerned.

You can of course combine this with external resistive voltage dividers to conceivably measure values of thousands of volts by dividing the whole down to something less than 3.3v and measuring that.

```
adc1_get_raw(ADC1_CHANNEL_5);

int val = adc1_get_raw(ADC1_CHANNEL_0);
```

Read ADC1 channel 0 and put digital value into the integer val.

We can also do this with the more familiar Arduino Commands but with less functionality. If you accept the 12 bit and 11dB defaults, we can use the much more familiar Arduino analog read syntax:

```
pinMode(34,INPUT);

analogRead(34);

analogSetWidth(12); 9-12

analogSetAttenuation(ADC_6db); // all ADC pins

analogSetPinAttenuation(ADC_11db); // specific pin
```

Copyright 2018 EVTV LLC

```
ADC_0db:
sets no attenuation (1V input = ADC reading of 3959).

 ADC_2_5db:
sets an attenuation of 1.34  (1V input = ADC reading of 2975).

 ADC_6db:
sets an attenuation of 1.5  (1V input = ADC reading of 2086).


ADC_11db:
 sets an attenuation of 3.6 .(1V input = ADC reading of 1088)
```

```
analogSetCycles(8):
set the number of time cycles per sample. Default 8. Range: 1 to
255.

analogSetSamples(1); //Default 1 – number of samples averaged
per read call
```

ADC depends on an internal reverence voltage of nominally 1.1vdc. But this voltage can vary. You can read the value of this reference voltage.

```
adc2_vref_to_gpio(25);
analogRead(25);
```

The values returned by any ADC reading are also not precisely linear across the entire range. Correcting for this is extremely ugly and requires the use of a polynomial equation.

```
  const double f1 = 1.711136146048750le+001;
  const double f2 = 4.2319467860421662e+000;
  const double f3 = -1.9077375643188468e-002;
  const double f4 = 5.4338055402459246e-005;
  const double f5 = -8.7712931081088873e-008;
  const double f6 = 8.7526709101221588e-011;
  const double f7 = -5.6536248553232152e-014;
  const double f8 = 2.4073049082147032e-017;
  const double f9 = -6.7106284580950781e-021;
  const double f10 = 1.1781963823253708e-024;
  const double f11 = -1.1818752813719799e-028;
  const double f12 = 5.1642864552256602e-033;
```

Copyright 2018 EVTV LLC

```
double rd = analogRead(adcpin);

double correctedvalue=f1+f2*pow(rd,1)+f3*pow(rd,2)+f4*pow(rd,3)+f5*pow(rd,4)
+f6*pow(rd,5)+f7*pow(rd,6)+f8*pow(rd,7)+f9*pow(rd,8)+f10*pow(rd,9)+f11*pow(rd,10)+f12*pow(rd,11)
;
```

You would only go to this length if you really needed better accuracy across the range.

ANALOG OUTPUTS

The ESP32 features two true Digital to Analog converters on **GPIO 25** and **26**. The syntax is unique to the ESP32 boards and is somewhat low resolution at 8-bits.

But it will produce an analog output voltage from about 0.080 to 3.300 vdc in 255 increments

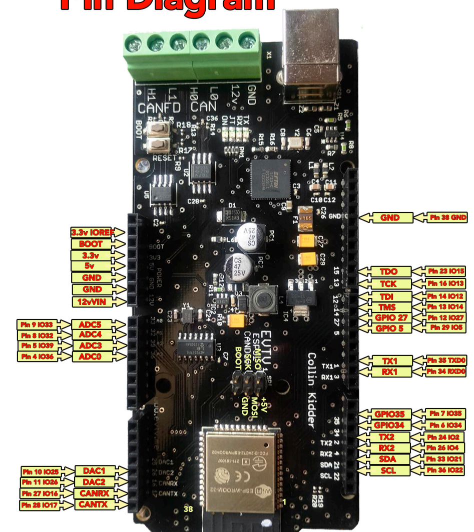
```
dacWrite(25,100);
```

This command would produce the output 100/256*3.3vdc or 1.289vdc. This is a very low current output and would need amplification to be useful. But it sufficiently responsive to produce audio output.

DIGITAL I/O

All the above mentioned analog input pins AND the DAC analog output pins can also be used as digital INPUT pins. In general they cannot be used as digital output pins.

EVTV ESP32 CANDue Pin Diagram




```
pinMode(39,INPUT);

digitalRead(39);
```

The pinmode statement only has to be performed once. Thereafter, the `digitalRead(39)` statement will return a `1` if a voltage of 3.3v appears on the input and a `0` if it does not.

These pins CANNOT be used for digital output however.

Almost all other ESP32 pins can be used for general purpose digital input and output pins in theory. In practice, our board

design reserves a number of those pins for duties such as USB serial port communications, additional serial ports, two CAN ports, and Serial Peripheral Interface pins. The following pins are then provided for general digital input/output duties.

- **GPIO 2 TX2**
- **GPIO 4 RX2**
- **GPIO 5**
- **GPIO 12**
- **GPIO 13**
- **GPIO 14**
- **GPIO 15**
- **GPIO 21 SDA**
- **GPIO 22 SCL**
- **GPIO 27**

To use as digital output:

```
pinMode(15,OUTPUT);

digitalWrite(15,HIGH);

digitalWrite(15,LOW);
```

Writing HIGH will cause the output pin to go to the reference voltage of 3.3 volts. Writing LOW will cause the output to go to 0v. Output current is limited to 40 milliamperes. **GPIO 21** and **22** are nominally used for **I^{2}C** serial communications. They are connected to 3.3vdc via 4.7k pullup resistors on the board and so they exhibit the behavior that they will read high with a digitalRead statement all the time with nothing connected. Any external connection will overcome this of course with a high or low value and similarly any statement to digitalWrite output high or low will work correctly as well. But their initial power-on state would read as a HIGH.

DIGITAL I/O INTERRUPTS

It can be very useful to use the digital input capability of the ESP32CANDue to monitor external events and act on them. I very useful feature is the concept of digital input interrupts.

```
pinMode(34,INPUT);
attachInterrupt(digitalPinToInterrupt(34), MyFunction, FALLING);

void MyFunction()
{
     Serial.println("External event occurred");
     Return;
```

Copyright 2018 EVTV LLC

```
)
```

In this example, GPIO 34 is monitored and in the event it changes from a HIGH state to a LOW state, the program loop will be halted and the user function "MyFunction" will be executed. As soon as that function completes, the program resumes operation where it was in the main loop when it was interrupted.

The trigger for this interrupt can be specified as RISING, FALLING, or CHANGE. Obviously RISING would then trigger when the pin changes from a LOW state to a HIGH state. CHANGE triggers with EITHER a rising or falling state.

Any pin that can be configured for digital INPUT can be used for interrupt routines. The pin mode and interrupt definition would normally be configured ONCE in SETUP while the actual function would appear elsewhere in the source code. Thereafter, the change in pin state would cause execution of the specified function.

PULSE WIDTH MODULATION

We have already noted that the ESP32CANDue board features two analog outputs on GPIO 25 and 26 with the very simple **`dacWrite`** command. We didn't spend much space describing it because they really aren't very useful in the grand scheme of things.

We can achieve essentially the same thing on ANY GPIO pin that can be used as output using **Pulse Width Modulation**.

Pulse Width Modulation is a very common technique where we can vary the AVERAGE output voltage on any digital pin by turning it to ON (digital 1 or 3.3v) and OFF (digital 0, 0v) at a specific rate. That rate is comprised of two elements, the frequency and duty cycle.

If we turn the output ON for a period and then off for a second period and we rinse and repeat, the time duration sum of the ON and OFF periods is the FREQUENCY for example.

For example, if we turned it ON for 8.33 milliseconds and OFF for 8.33 millisconds, the total DURATION of our waveform would be 16.66 milliseconds. We can also express this as the number of cycles of this waveform – 60 complete cycles per second or 60 Hz. So 60 would be our FREQUENCY.

Copyright 2018 EVTV LLC

And because we had the output ON for half the waveform and OFF for the other half, we would say we had a 50% DUTY CYCLE and our average output voltage would be around 3.3v/2 or 1.65v.

Of course if we turned it on for 3.332 milliseconds and off for 13.382 milliseconds, we would STILL have a 60Hz waveform but our DUTY CYCLE would be 25% and our output voltage would average 3.3/4 or 0.825 volts.

At somewhat higher frequencies, this average output voltage consists of so many pulses, that it is very difficult to distinguish from a true analog voltage for all practical purposes.

The ESP32 chip allows us to do this on ANY output GPIO at an astonishing variety of frequencies and resolutions – up to 40 MHz in frequency and with our percentage of duty cycle specified not to 0-100% but from 0 to 32768 discrete steps. But these limits are somewhat interactive as we will see.

This gives us an equally astonishing control over precisely what the average output voltage is and of course how quickly we can change that.

The Arduino had a very similar function termed **`analogWrite(pin, value)`** where **`pin`** specified the output pin, usually from a very few special pins, and **`value`** was the PWM duty cycle from 0 to 255 – eight bit precision.

There IS no **`analogWrite`** function available for the ESP32 chip even in the Arduino IDE and for very good reason. The simplistic **`analogWrite`** is not sufficiently complex syntax to take advantage of the flexibility of the ESP32 chip. The 8-bit precision was a bit limiting and there is no provisions in the function to change the frequency at all. As different Arduino chips indeed use different clocks, this turned into a very complicated situation with regards to frequency.

It has been replaced in the ESP32 by the **`ledcWrite(channel, duty cycle)`** function. But it is somewhat more complex to set up and use to take advantage of the greater flexibility and power of this.

The basic format to achieve PWM on the EVTV ESP32CANdue board:

```
pinMode(13,OUTPUT);
ledcSetup(channel, frequency, resolution);
ledcAttachPin(GPIO pin, channel);
ledcWrite(channel, dutycycle);
```

CHANNEL. The ESP32 has four clocks and using different divisors we can select 16 channels numbered 0-15.

FREQUENCY is the desired frequency in Hz.

RESOLUTION is the number of bits we want to use to set the duty cycle and must be

Copyright 2018 EVTV LLC

between 1 and 15.

As mentioned, the maximum frequency and bits are interactive and based on the 80 MHz base clock of the chip. **MaxFreq=80,000,000/2^{bits}**

And so we can work this out for you.

RESOLUTION	INCREMENTS	MAX FREQUENCY
1	2	40000000
2	4	20000000
3	8	10000000
4	16	5000000
5	32	2500000
6	64	1250000
7	128	625000
8	256	312500
9	512	156250
10	1024	78125
11	2048	39062.50
12	4096	19531.25
13	8192	9765.62
14	16383	4882.81
15	32767	2441.41

And so we can see from this chart that we could set the resolution to 12-bit and have a luxurious 4096 discrete increments to specify our duty cycle and at ANY frequency up to 19531 Hz maximum.

```
pinMode(13,OUTPUT);
ledcSetup(0,17500,12);
ledcAttachPin(13,0);
ledcWrite(0,2048);
```

This will set GPIO pin 13 up to exhibit a 17.5kHz waveform and a 50% duty cycle for an analog output of 1.65volts. This will be constant and continuous until you change it.

Note that the first two statements only need to be executed ONCE in setup. Subsequently, any iterations of `ledcWrite(0,dutycycle)` will set the PWM duty cycle value for that channel 0 and the already associated pin 13. And so you can easily vary the duty cycle anywhere in your program.

There is another potentially useful function `ledcDetachPin(13)` that will disassociate pin 13 from channel 0, allowing it to be reassigned for other duties.

Copyright 2018 EVTV LLC

A couple of useful functions allow you to READ the state of the PWM pin output.

`ledcRead(channel);` returns the current PWM value actually on the pin.

`ledcReadFreq(channel);` returns the current frequency of the waveform on the pin.

SERIAL PERIPHERAL INTERFACE (SPI) PORT

We previously noted that the CANFD output on CAN1 is provided using the MicroChip MCP2517FD chip. This communicates with the ESP32 microprocessor using the Serial Peripheral Interface at a relatively high 20.0 Mbps.

The SPI bus allows the ESP32 to act as a MASTER on this serial bus with any number of SLAVE devices. A SLAVE is turned on using a chip select signal on a separate output. In the case of CANFD, GPIO5 is used as the CS output for the MCP2517FD chip.

As one of the original features of the Arduino model, an In Circuit Serial Programming (ICSP) port was provided to allow direct programming of the microcontroller. As the Arduino is normally programmed by uploading binaries through the more familiar USB port, it was almost never used for this purpose.

Copyright 2018 EVTV LLC

Rather, and by convention, the pins on this ICSP port were used as a Serial Programming Interface (SPI) port. And it became the standard way of connecting Arduino shields that featured SPI devices.

We have retained this convention on the EVTV ESP32 CANDue board. The diagram below shows the pin identifications for the SPI port.

Pin 1 is the Master In/Slave Out (MISO) signal connected to ESP32 pin 31.

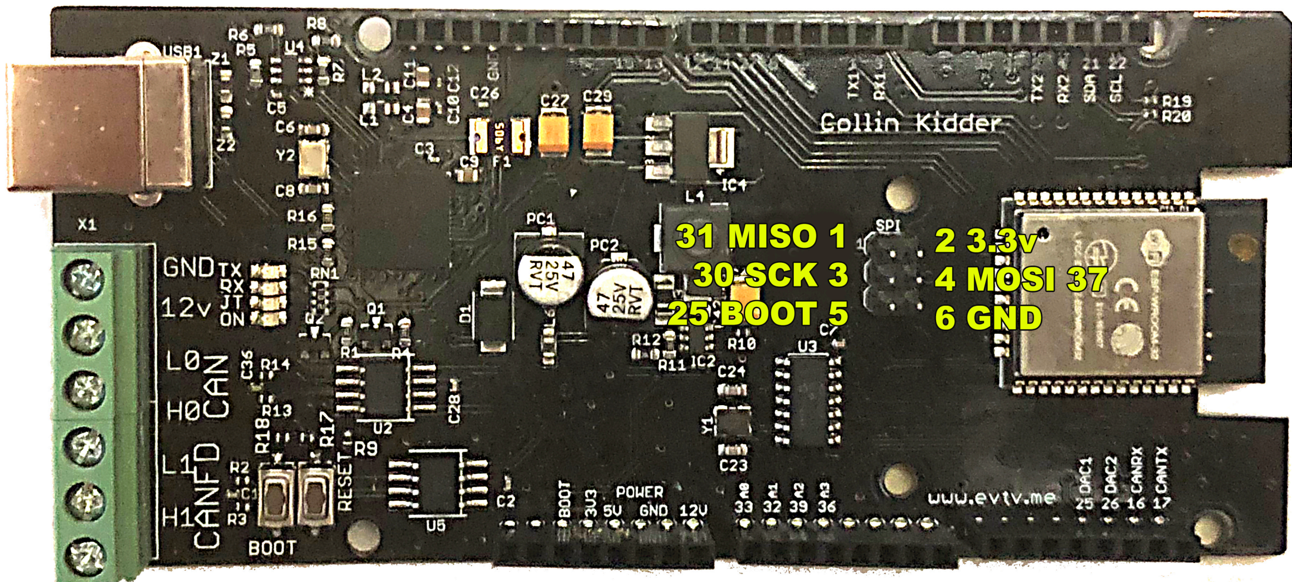
Pin 2 provides 3.3vdc power.

Pin 3 provides the Serial Clock signal (SCK) available on ESP32 pin 30.

Pin 4 provides the Master Out/Slave In (MOSI) signal from ESP32 pin 37.

Pin 5 is conventionally the RESET pin used for ICSP applications. We've connected it to the BOOT switched ground from the BOOT switch. It CAN be used as Chip Select.

Pin 6 is the ground return for the 3.3v supply.



In this way, you can make connections to any external SPI device and use the standard Arduino/ESP32 SPI library to communicate with it.

Any unused pin from the available digital outputs can be used as a chip select (CS) output to your SPI device. Set it to LOW to select your device.

Indeed, you can connect multiple SPI devices to this port and operate them in turn by using their specific CS output pins.

The BOOT pin GPIO-0 would normally be used as Chip Select.

```
pinMode(0,OUTPUT);

digitalWrite(0,LOW);  Chip Selected and device activated.

digitalWrite(0,HIGH);  Chip De-selected and device inactivated.
```

Copyright 2018 EVTV LLC