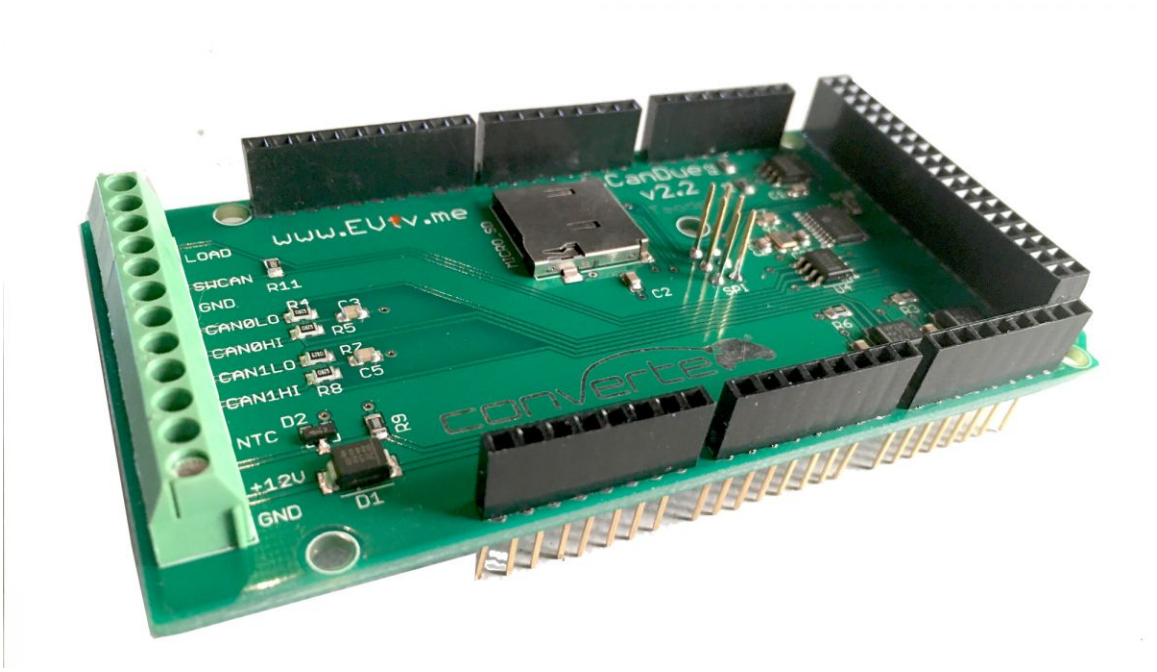
User Manual

CANDue Version 2.20
CAN Bus Shield
For
Arduino Due



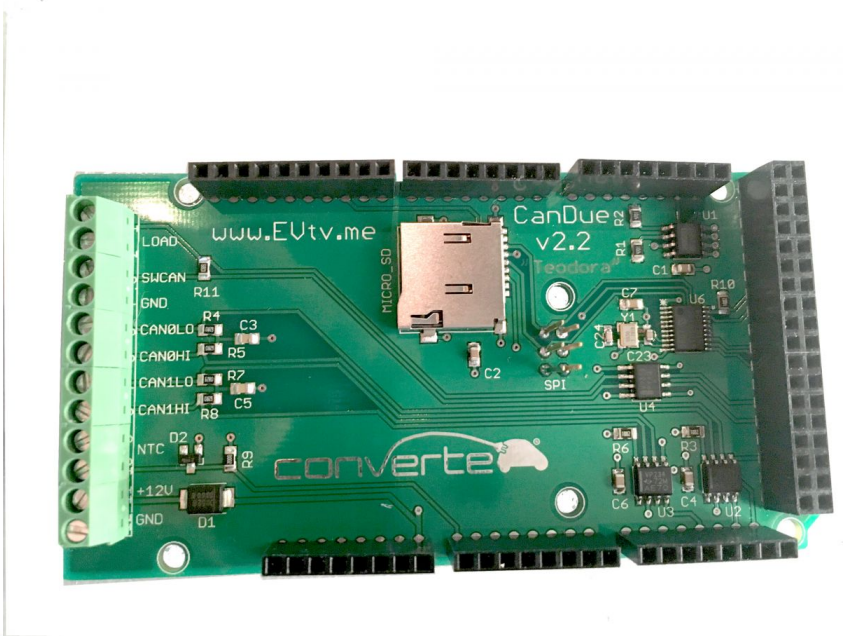
INTRODUCTION

The EVTV CANdue product provides a package of an Arduino Due compatible microcontroller board, CAN bus communications shield, and associated USB cable intended to provide a reasonably complete solution to monitoring CAN bus networks, interacting with them, and providing a means to record large volumes of CAN message traffic. The product includes:

1. Two port CANDue 2.2 CAN bus shield with EEPROM and microSD memory card slot.

2. Generic Arduino Due compatible microcontroller board.

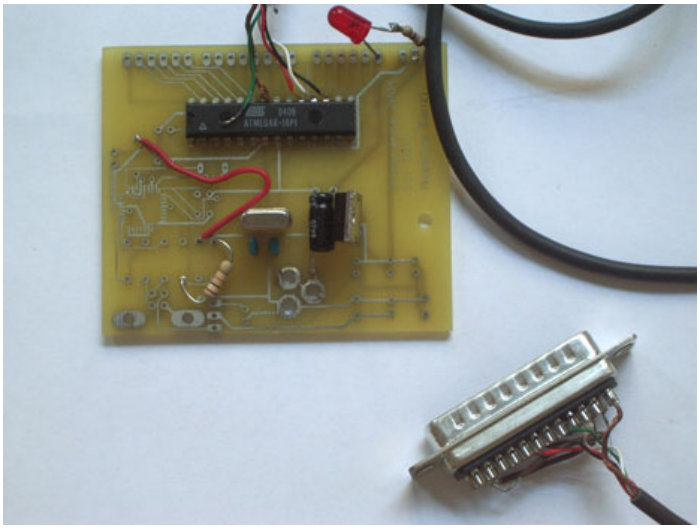
3. USB cable.



ARDUINO DUE

The Arduino microcontroller is an open-source hardware microprocessor controller designed to easily interface with a variety of sensors (to register user inputs), and to drive the responses and behaviors of external components such as LEDs, motors, and speakers (to respond to user inputs).

In 2005, the Arduino team was formed in Ivrea, Italy, consisting of Massimo Banzi, David Cuartielles, Dave Mellis, Gianluca Marino, and Nicholas Zambetti. The Arduino incorporated a programming environment based on Processing language - a programming language conceived by Ben Fry and Casey Reas, the ability to program the board via a standard USB connection, and a low price point (starting at about $35 USD).



The Arduino achieved rapid success even within its first two years of existence, selling more than 50,000 boards. By 2009, it had spawned over 13 different incarnations, each specialized for different applications — for example, the Arduino Lilypad (for wearable technologies projects), the Arduino Mini (miniaturized for use in small interactive objects), and the Arduino BT (with built-in Bluetooth capabilities).

Arduino started as a project for students at the Interaction Design Institute Ivrea in Ivrea, Italy. At that time program students used the Parallax BASIC STAMP controller at a cost of $100, considered expensive for students. Massimo Banzi, one of the founders, taught at Ivrea. The name "Arduino" comes from a bar in Ivrea, where some of the founders of the project used to meet.

The bar itself was named after Arduino, Margrave of Ivrea and King of Italy from 1002 to 1014.

The first prototype board, made in 2005, was a simple design, and it wasn't called Arduino. Massimo Banzi would coin the name later.

Banzi and his collaborators were strong believers in open-source software. Since the purpose was to create a quick and easily accessible platform, they felt they'd be better off opening up the project to as many people as possible rather than keeping it closed. This was kind of a remarkable innovation – open source hardware. The product the team created consisted of cheap parts that could easily be found if users wanted to build their own boards, such as the ATmega328 microcontroller. But a key decision was to ensure that it would be, essentially, plug-and-play: something someone could take out of a box, plug into a computer, and use immediately. Boards such as the BASIC Stamp required that DIYers shell out for half a dozen other items that added to the total cost. But for theirs, a user could just pull out a USB cable from the board and connect it to a computer—Mac or PC—to program the device.

Word of Arduino quickly spread online, with no marketing or advertising. Early on, it attracted the attention of Tom Igoe, professor of physical computing at the Interactive Telecommunications Program at New York University. Igoe too had been teaching courses to nontechnical students using the BASIC Stamp but was impressed by Arduino's features and price.

The **Arduino Due** is Arduino's first ARM-based Arduino development board. This board is based on a powerful 32bit CortexM3 ARM microcontroller made programmable through the familiar Arduino IDE. It increases the computing power available to Arduino users by an order of magnitude.

The Arduino Due has 54 digital input/output pins (of which 12 can be used as PWM outputs), 12 analog inputs, 4 UARTs (hardware serial ports), an 84 MHz clock, a USB-OTG capable connection, 2 DAC (digital to analog), 2 TWI, a power jack, an SPI header, a JTAG header, a reset button and an erase button.

To compile code for the ARM processor, you'll need the latest version of the Arduino IDE: v1.5

Note: Unlike other Arduino boards, the Arduino Due board runs at 3.3V. The maximum voltage that the I/O pins can tolerate is 3.3V. Earlier Arduino boards were based on a 5v power supply. As a result, shields and other hardware for Arduino are generally NOT compatible with the Arduino Due. Providing higher voltages, like 5V to an I/O pin could damage the board.

Features:

- Microcontroller: AT91SAM3X8E
- Operating Voltage: 3.3V
- Recommended Power Input Voltage: 7-12V
- Min-Max Input Voltage: 6-20V
- Digital I/O Pins: 54 (of which 12 provide PWM output)
- Analog Input Pins: 12
- Analog Outputs Pins: 2
- Total DC Output Current on all I/O lines: 130 mA
- DC Current for 3.3V Pin: 800 mA
- DC Current for 5V Pin: 800 mA
- Flash Memory: 512 KB all available for the user applications
- SRAM: 96 KB (two banks: 64KB and 32KB)

Clock Speed: 84 MHz

The Arduino Due provides a very fast, powerful microcontroller that is easily programmed in a C++ syntax vie the Arduino Integrated Design Environment (IDE). You can connect it to a PC or Mac via a USB cable to both power the board and upload compiled software.

CAN BUS

CAN bus (**controller area network**) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer.

CAN bus is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as aerospace, maritime, industrial automation and medical equipment.

Development of the CAN bus started originally in 1983 at Robert Bosch GmbH.[1] The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) congress in Detroit, Michigan. The first CAN controller chips, produced by Intel and Philips, came on the market in 1987.

Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991. This specification has two parts; part A is for the standard format with an 11-bit identifier, and part B is for the extended format with a 29-bit identifier. A CAN device that uses 11-bit identifiers is commonly called CAN 2.0A and a CAN device that uses 29-bit identifiers is commonly called CAN 2.0B. These standards are freely available from Bosch along with other specifications and white papers.

In 1993 the International Organization for Standardization released the CAN standard ISO 11898 which was later restructured into two parts; ISO 11898-1 which covers the data link layer, and ISO 11898-2 which covers the CAN physical layer for high-speed CAN. ISO 11898-3 was released later and covers the CAN physical layer for low-speed, fault-tolerant CAN. The physical layer standards ISO 11898-2 and ISO 11898-3 are not part of the Bosch CAN 2.0 specification.

CAN bus is one of five protocols used in the on-board diagnostics (OBD)-II vehicle diagnostics standard. The OBD-II standard has been mandatory for all cars and light trucks sold in the United States since 1996, and the EOBD standard has been mandatory for all petrol vehicles sold in the European Union since 2001 and all diesel vehicles since 2004.

But in recent years, OBDII has evolved more toward CAN and the other protocols more or less comprise legacy protocols found on older vehicles. CAN won.

ARDUINO DUE AND CAN

The heart of the Arduino Due is a microprocessor controller chip designated **AT91SAM3X8E**. This quite powerful 84 MHz 32-bit chip actually provides internal "controllers" for two CAN channels. Unfortunately, this is just the control logic for the channels. To use CAN, you also need the "power" components that actually transmit a CAN message over the wires. These "transceivers" were not included in the Arduino Due design and no specific CAN commands were included in the Arduino IDE.

And so, to actually provide CAN communications for the Arduino Due, we need a couple of things:

1. Hardware transceivers and a physical connector to wire the system to a CAN bus.

2. A "library" object module to extend the Arduino language to include CAN commands.

EVTV CANDUE CAN BUS SHIELD

The EVTV CANDue Version 2.2 "Teodora" CAN bus shield with EEPROM and Micro SD memory card reader was originally designed by Paulo Almeida and Celso Menaia of Lisbon Portugal and produced by EVTV Motor Werks.

This board features two CAN 2.0 compliant communications channels capable of data rates up to 1 Mbps and a 256KB Electrically Erasable Programmable Memory chip (EEPROM).

The original Arduino featured a 4KB EEPROM in the controller chip itself. This can be very handy to store configuration variables that you want to be able to change, but would like to be persistent from one power cycle to the next.

The AT91SAM3X8E microcontroller used on the Due version of Arduino does not provide this EEPROM. It does feature a persistent "flash" memory of an impressive 512KB, but this memory contains the compiled software you upload to the Arduino. Any time you upload a new version of this software, you would lose all your configuration data.

And so we included a quite large 256KB EEPROM to store variables or log data. The CAN bus transceivers used are a Texas Instruments 3.3v transceiver designated **SN65HVD234.** Each of two chips manages a single CAN channel designated CAN0 and CAN1.

The communications interface of these chips is provided by screw terminals at the edge of the board.

CAN TERMINATION.

The CAN bus specification requires that the two ends of the two wire, twisted wire bus be terminated with a resistive impedance of 120 ohms.

If you are using the CAN bus shield to communicate on an existing vehicle CAN bus, it is pretty much a given that the bus already has two devices on it that serve as "ends" and that they provide proper impedance termination.

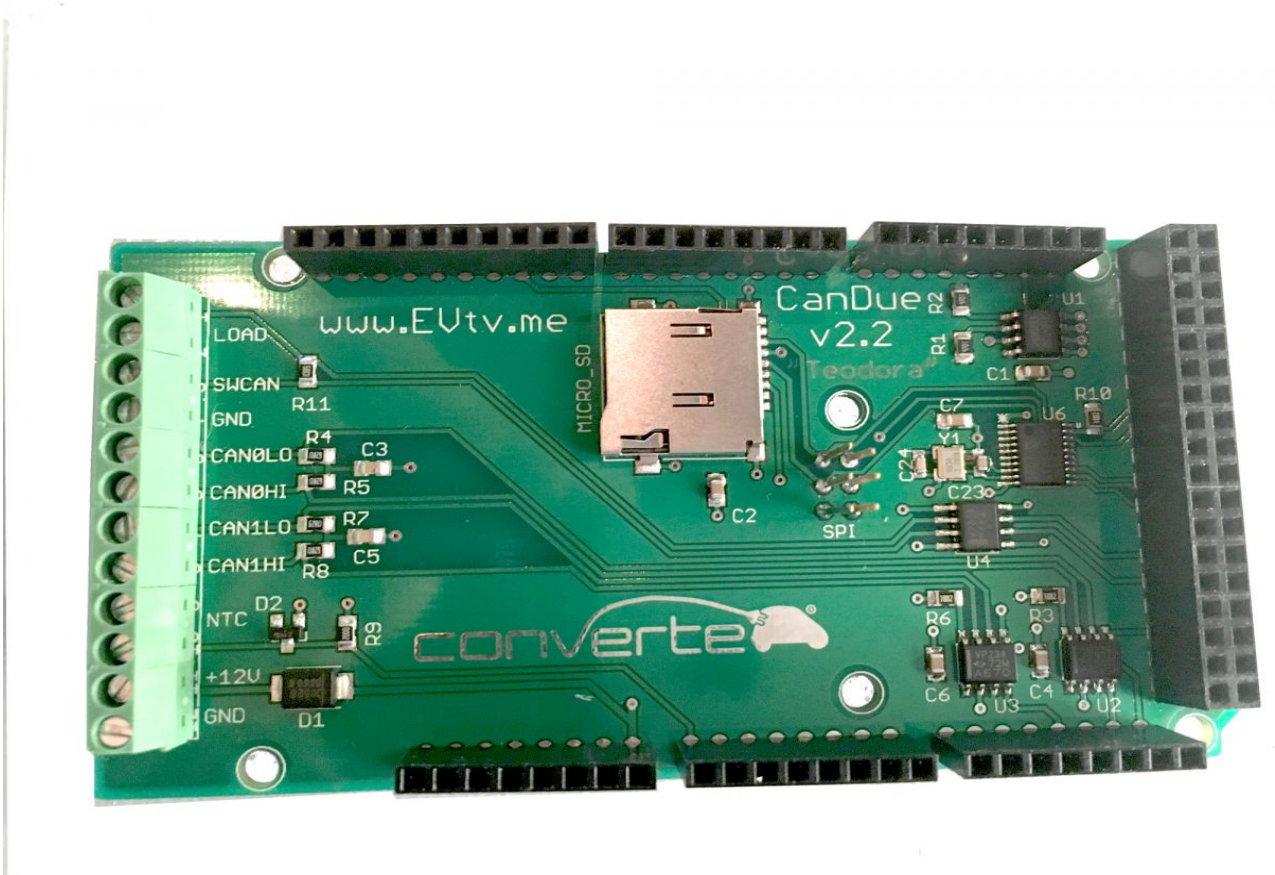
But in some applications, for example communicating with a single charger or battery management component, the Arduino and this shield ARE one of the ends of the bus.

To provide for this, each CAN channel features two 60 ohm resistors, one for each of the two CAN wires, and a capacitor. The capacitor filters noise to ground. The two

60 ohm resistors combine for the requisite 120 ohms while providing a point for the cap filter to connect so that the reactance is equalized across the two wires.

In an upgrade from the switched termination on version 2.0. the 2.2 iteration simply includes a couple of 0 ohm resistors. In our experience, in virtually EVERY case your board must be terminated to have any success on the CAN bus. In the extreme and unlikely event that you might need an unterminated connection, you will have to remove the resistors by desoldering them from the board. We've basically never had this requirement.

.



The CAN0 and CAN1 ports are provided on a screw terminal strip on the side of the shield. This allows you to easily connect CAN1 HI and CAN1 LO to any kind of cable pigtail or connector desired. Two of the screw terminals are also connected to 12 VIN and GND.

CANDue Version 2.2 also supports single wire CAN - a 33.33kbps variant used on the Ebenspacher Heater and the TESLA Charger for example. Data is transferred at a slower speed on a single wire using frame ground as the differential. SWCAN,

GND, and two screw terminals are provided for a load resistor generally used on SW CAN applications.

Finally, two screw terminals are provided for a 10k NTC (Negative Temperature Coefficient) Thermistor. A voltage divider circuit connects the 3.3v reference voltage across the NTC and this voltage divider to provide a readable input to Analog input 0 on the Due. This input is also guarded by a Zener diode to prevent minor overvoltages. In this way, you can use the Due to calculate temperature using most General Motors temperature sensors.

DUE_CAN

due_can is an Arduino Due specific library written by Collin Kidder of Sparta Michigan. It provides functions and methods to easily deal with the very powerful CAN bus transceiver functions available in the CANDue shield.

The Arduino IDE (Integrated Design Environment) provides a basic C++ programming language syntax with some curious "extensions" to deal with hardware easily and directly. 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<due_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 used in any particular program.

CAN PORTS

The CANDue shield provides two hardware ports CAN0 and CAN1.
.

INITIALIZATIONS AND BEGINNINGS

So our first requirement is an include statement.

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

Our second is to INITIALIZE the CAN port we want to use with a `begin` statement. We can do either or both.

```
pinMode(50,OUTPUT);

if (Can0.begin(25000,50)) {

  Serial.println("Using Can0 - initialization
completed.\n");

 }

 else Serial.println("Can0 initialization (sync) ERROR\n");

pinMode(48,OUTPUT);

if (Can1.begin(25000,48)) {

  Serial.println("Using Can1 - initialization
completed.\n");

 }

 else Serial.println("Can1 initialization (sync) ERROR\n");
```

The basic initialization is handled by

```
Can0.begin(25000,50)
```

Note that **250000** is the data rate 250kbps and **50** is the ENABLE pin. This is a hardware function of this particular CANDue shield. We use digital pin 50 to enable the Can0 hardware transceiver. We use digital pin 48 to enable the Can1 transceiver. Due_can is capable of supporting virtually any CAN shield – but you need to know the ENABLE pins used by the particular shield you are using to properly initialize this.

If your CAN board does not use an ENABLE pin, set these values to 255.
If no pin is specified, the default values that will then be used are pin 62 for Can0 and pin 65 for Can1.

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 1 to 8 bytes.

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.

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

```
outFrame.id     //uint32_t – a 32bit (4 byte) variable
                // containing the CAN message ID.

outFrame.fid     //uint32_t – a 32 bit (4 byte) variable for
                // family ID. Not commonly used.

outFrame.rtr    //uint8_t – a one byte variable for
                //Remote Transmission Request. Rarely used.

outFrame.priority //uint8_t a one byte for Priority,
                    //occasionally used for TX frames.
```

```
outFrame.extended  //uint8_t a one byte for Extended
                   //Addressing (29-bit) flag 0= 11-bit

outFrame.length  //uint8_t a one byte for number of data
                 //bytes to follow

outFrame.data.bytes[0] //uint8_t first data byte
outFrame.data.bytes[1] //uint8_t data byte
outFrame.data.bytes[2] //uint8_t data byte
outFrame.data.bytes[3] //uint8_t data byte
outFrame.data.bytes[4] //uint8_t data byte
outFrame.data.bytes[5] //uint8_t data byte
outFrame.data.bytes[6] //uint8_t data byte
outFrame.data.bytes[7] //uint8_t data byte
```

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

For convenience there are two OTHER ways to address the data in the eight data byte payload of the CAN frame bytes[0] through bytes [7].

```
outFrame.s0  //uint16_t bytes[0] and bytes[1] as a two
               //byte unsigned integer.

outFrame.s1  //uint16_t bytes[2] and bytes[3] as a two
               //byte unsigned integer.

outFrame.s2  //uint16_t bytes[4] and bytes[5] as a two
               //byte unsigned integer.

outFrame.s3  //uint16_t bytes[6] and bytes[7] as a two
               //byte unsigned integer.
```

`outFrame.s0` through `outFrame.s3` is actually quite convenient. CAN data is often more easily dealt with as a series of four two-byte integers for presenting data that can often be larger than 255.

The final way to address data in the eight data byte payload is as low and high – two 32-bit unsigned integers.

```
outFrame.low  //uint32_t bytes[0] - bytes[3] as a four
               //byte unsigned integer.

outFrame.high  //uint32_t bytes[4] - bytes[7] as a four
               //byte unsigned integer.
```

This allows us to send and receive 32-bit data via CAN.

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.bytes[5]`). You can define as many `CAN_FRAME` variables as you like.

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.

```
CAN_FRAME myFrame;
myFrame.id = 0x25A;
myFrame.length = 1
myFrame.data.bytes[0] = 128;
Can0.sendFrame(myFrame);
```

Here we define `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 structure, 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.bytes[0] = 129;

Can0.sendFrame(myFrame);
```

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

sendFrame is the most common method to send data over CAN. But there is a simplified alternative command you can use to send CAN frames that does not involve using a `CAN_FRAME` data structure that you may find useful.

```
Can0.setWriteID(0x620);

Can0.write(SomeValue);
```

You first use the `setWriteID` function to establish the message ID you want to use. This method takes a single argument and will use 11-bit or 29-bit extended addressing automatically based on the ID number you provide.

The write function simply stuffs whatever data you provide into the 8 data bytes available in sequence.

This can be useful in certain specialized instances. For example, Arduino Due features 32-bit LONG integers that are actually four bytes.

```
Long myLongInteger=2914145897000;

Can0.setWriteID(0x620);

Can0.write(myLongInteger);
```

These three lines would write the four data bytes containing the value in `myLongInteger` under message ID 0x620. The last four bytes of the block would be padded with zeros.

Note that Arduino Due uses a "little endian" data format where the least significant of the four bytes is stored in the first position while the most significant byte of the long integer is sent in the fourth byte position.

At times, you might wish to reverse this.

```
Can0.setBigEndian(true);
```

This statement sets `Can0.write()` method to send the integer in the reverse order – most significant byte in the first data byte position and the least significant byte in the fourth data byte position.

RECEIVING BUFFERED CAN FRAMES

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

We can also setup an interrupt 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 tennet 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.

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.

due_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.

This is done with 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 or extended 29-bit message ids.

```
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. 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 8 different filters covering 8 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();
```

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.

```
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.

```
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.

CAN INTERRUPTS

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 interrupts.

CAN interrupts are simply a method 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.

This is an INTERRUPT and it does interrupt your normal program operation. It stops it, saves its state, processes the CAN frame, and then returns to the exact point in the program where the interrupt occurred.

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

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

```
Can1.setRXFilter(1, 0x18FF50e5, 0x1FFFFFFF, true);

Can1.attachCANInterrupt(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 just picks a

mailbox for you. There are 8 “mailboxes” provisioned as a function of the chip design and this is why you can have up to eight filters.

But you can optionally designate a specific mailbox to use 0-7. 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, **`attachCANInterrupt`**. 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.

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.bytes[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 8 interrupts and each can be to the same method, or other entirely different methods in our program, all based on their incoming addresses.

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

```
Can1.attachCANInterrupt(someOtherMethod);
```

Note that this is the same call, but does not specify a mailbox. It will be applied to any mailboxes that do not have an interrupt attached.

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.detachCANInterrupt(0);
```

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

`Can0.detachCANInterrupt();` will remove ALL `Can0` interrupts.

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<due_can>; //includes due_can library
uint8_t correction; //8bit integer holding a correction

void setup(){  //Our setup function
     Serial.begin(115000);
     pinMode(50,OUTPUT);
     If (Can0.begin(500000,50);){
          Serial.println("Can0 initialized…");
          }
     else Serial.println("Can0 failed…");
```

```
      Can0.setWriteID(0x05B); //Let's use message ID 0x500
      correction=5; //5 milliseconds for propogation
}

void loop(){
      Can0.write(millis()+correction);
      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,milliseconds);
      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:

```
#include<due_can>; //includes due_can library

void setup(){  //Our setup function
     Serial.begin(115000);
     pinMode(48,OUTPUT);
     If (Can1.begin(500000,48);){
          Serial.println("Can1 initialized…");

          }

     else Serial.println("Can1 failed…");
Can1.setRXFilter(1, 0x05B, 0x7FF, false);
Can1.attachCANInterrupt(1, getOurTime)
}
void loop(){
}
void getOurTime (CAN_FRAME *timeFrame){
     long remillis=timeFrame.low;
     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,milliseconds);
     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 05B message IDs.

The CAN bus has little intelligence, but it DOES work out bus contention on different devices. 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 an interrupt for Can1 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.low`. Recall that `.low` 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 Arduinos 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.

EEPROM USE

The Electrically Erasable Programmable Memory provided on the Dual Channel CAN Shield is a 2 megabit memory chip accessed serially through I2C on the SCA and SCL pins on the Arduino Due board. The 2 megabits of storage works out to 256 kilobytes of storage – 262,144 8-bit byte locations.

The significance of EEPROM storage is that anything you write to this memory is retained even when power is removed. Because of this, it can be used to store data which you desire to be "persistent" and retain its value through any number of power cycles.

While Arduino Flash memory could be purposed for the same thing with a bit of code magic, if you upload a NEW version of your program, you lose any persistent data held in flash memory this way. Not so with EEPROM stored data. It is always retained.

Examples of data where you would want this normally includes configuration variables where you desire to set a variable controlling program operation, but you don't really want to go through this configuration every time you power up. By writing these items to EEPROM, you can place code in your program SETUP to retrieve this data when you first power up and so we say these items are held in "persistent" storage. They are there every time you power up.

Storing data 8-bits at a time is a bit of a problem. BYTE, CHAR, BOOL, and INT_8T and UINT_8T data types can of course be stored directly. But the C++ used in the Arduino Due features a rich set of data types, often 32 bits in length. Storing and retrieving these variables requires a bit of work normally. Fortunately we have eliminated most of it, again through the use of libraries.

The libraries required for optimum use of this EEPROM storage includes:

`#include <due_wire >` This library lets us make the serial communications channel to the EEPROM chip through Arduino **SCA** and **SCL** pins LOOK like a simple serial channel. We can initialize this as:

`#include <Wire_EEPROM.h>` Collin Kidder access the EEPROM chip – revised for Arduino Due and the larger 256KB EEPROM size

`#include <eepromAny.h>` This library greatly simplifies the storage of multibyte variables.

```
We also have to initialize our EEPROM.  We use Arduino Due
Serial3 to communicate with the EEPROM.

EEPROM.setWPPin(19);

Serial3.begin(2400);
```

STORING VARIABLES TO EEPROM

Again, EEPROM is really a series of 8-bit bytes. But the various data types in C++ can be much longer. eepromAny makes this much easier to deal with.

```
EEPROM_write(address, variable);
```

The address can be any unsigned integer address from 0 to 266,143. And the variable can be any named variable of any of the basic types.

For example:

```
float amperehours=121.32;
uint_8t AHaddress = 1200;


EEPROM_write(AHaddress, amperehours);
```

The variable amperehours is actually a four-byte floating point variable containing the value 121.32. We are storing it at location 1200.

You ARE tasked with keeping aware of the length of your variables when assigning locations. The libraries will quite allow you to do the following:

```
EEPROM_write(AHaddress, amperehours);
EEPROM_write(1202, 1034589);
```

Since amperehours requires 4 bytes, you have just written the value 1,034,589 to EEPROM, but you've overwritten the amperehours variable stored there because you wrote it to location 1202, which was already being used by the amperehours value you wrote in the previous line.

So you see, some awareness of memory geometry is required.

RETRIEVING VARIABLES FROM EEPROM

Retrieving variables from EEPROM is actually even easier.

```
EEPROM_read(address, variable);
```

The syntax is essentially the same. The data stored at address will be copied into the variable in your program by that name.

```
EEPROM_read(AHaddress, amperehours);
```

It would be very unusual to need 266,144 bytes of storage for what is usually a handful of configuration variables. But in recent years EEPROM chips have become very inexpensive – less than $4 for the most part. And the difference in price between a 4KB and a 256KB chip is basically nil. We provide 256KB as you may want to have a routine that "logs" a certain amount of data to the chip and perhaps another that prints that log out the serial port to your computer. You could then capture that as a text file, and import to Microsoft Excel.

PAGE MODE

Writing to EEPROM via serial SCL/SCA involves a bit of time. And it is very important to not try to write to it WHILE it is writing to it. There are ways using interrupts to do just that.

Additionally, EEPROMs are life limited to the number of writes you can perform on the chip itself. This particular chip has a rather long write life of 4,000,000 operations. But if you do this in your LOOP section, it doesn't take long to even breach that limit at 84 MHz.

The EEPROM is actually arranged in 1024 individual "pages" of 256 bytes each. It can be an advantage to write an entire page at a time. This actually REDUCES the time you spend writing to EEPROM, which is a bit counterintuitive.

Fortunately, Mr. Kidder has provided us a page mode operation in his `Wire_EEPROM` library.

The first thing we want to do is define a CLASS containing nothing but public variables. These variables can include any kind of existing variables or even structures within the class. This class is defined immediately after our include statements and so the class and its variables behave essentially as GLOBAL variables accessible from anywhere in the program.

```
#include <Arduino.h>
#include <due_can.h>
#include <due_wire.h>
#include "variant.h"
#include <Wire_EEPROM.h>
class EEPROMvariables {
   public:
     uint8_t CANdo
     uint16_t voltage;
     uint16_t current;
     uint16_t terminate;
     float AH;
     float kilowatthours;
};
EEPROMvariables myVars;
```

Immediately after the class, we instantiate an object of that class. Here we are calling the class EEPROMvariables and we have instantiated an object myVars in this class.

From that point, anywhere in our progam we can read and write to any of those variables very quickly simply by using them in the form of `myVars.voltage, myVars.current, myVars.AH, myVars.kilowatthours`, etc.

In our setup, we want to initialize our connection to EEPROM and load our last saved page of these variables.

```
void setup(){
    Wire.begin();
    EEPROM.setWPPin(19);
     EEPROM.read(200, myVars);
}
```

The `EEPROM.read` command names the numeric page (some value between 0 and 1023 and the object. It then loads that specific page of memory into the object using the structure of the variables in the class. The memory requirements of the class can of course be smaller than the 256 byte page, but they CANNOT be longer than 256 bytes. If you need more EEPROM variables, you will have to define more instances of your object and keep them on DIFFERENT pages.

The first time we do SETUP of course the page will be empty and as a result so will all our variables. But if somewhere in our program we do a `EEPROM.write(200, myVars);` then those variables, in whatever state we had them in and containing whatever data we stored there, would be written to that page in EEPROM. The next time we power cycle the device, the stored data would be loaded correctly into our `myVars` object.

In this way, we can use these variables quickly and continuously in our program. But we only actually save them to EEPROM occasionally. How occasionally, it depends on the time nature of your data.

In the example above, we are accumulating ampere hours in the variable `myVars.AH` and we are updating this every 20 milliseconds or so. If we failed to save this for a period of say, 5 seconds, we would potentially lose the last 5 seconds of accumulation worst case. I can live with it. It isn't that much.

But how often you actually write to EEPROM is a function of how time sensitive your data is. Often you are simply storing configuration options for your program and they simply are not time sensitive at all.

```
void loop(){
   if(millis()-lastime >5000)
          {
            lastime=millis();
            EEPROM.write(200, myVars);
          }
}
```

In our loop above, we use the built-in Arduinos function `millis()` to determine the passage of time. Millis will return the number of milliseconds since the last power cycle. If it is greater than 5000 ms or 5 seconds, we are going to execute these lines.

The first thing we do is reset our timer by setting `lastime` to `millis().` Then we write `myVars` to page 200 of our EEPROM.
On the next LOOP pass, and for several hundreds of thousands of others, when we compare `millis()` to `lastime,` it is going to be less than 5000 for a long time.

In this way, we can have ready and immediate access to a number of variables of entirely different types, but still store them to EEPROM so they will be available the next time we power cycle. This, while minimizing the writes to our life limited EEPROM.

MICRO SD CARD SLOT USE

The CANDue board features a microSD card slot capable of holding very small SD cards. In recent years, the data storage capacity of these tiny cards has grown enormously – 256GB or more.

Files can be created, removed, written to, and read from using the familiar DOS FAT 16 or 32 file format using file names in the familiar 8.3 format (up to eight name characters plus three character extension). FAT 16 is good up to 2 GB. For files above 2GB you must use FAT32 as the format.

This feature provides enormous data logging capacity in a removable medium.

The Arduino Due accesses the data card slot through a dedicated Serial Peripheral Interface (SPI) a six pin connector in the center of the board.

Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers.

With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically there are three lines common to all the devices:

MISO (Master In Slave Out) - The Slave line for sending data to the master,

MOSI (Master Out Slave In) - The Master line for sending data to the peripherals,

SCK (Serial Clock) - The clock pulses which synchronize data transmission generated by the masterand one line specific for every device:

SS (Slave Select) - the pin on each device that the master can use to enable and disable specific devices.

When a device's SS pin is low, it communicates with the master. When it's high, it ignores the master. This allows you to have multiple SPI devices sharing the same MISO, MOSI, and CLK lines. The microSD card reader uses **digital output pin 10** as the SS.

By way of trivia you probably do NOT need to program this card, on the Arduino Due version only:

MISO uses digital pin 75

MOSI uses digital pin 74

SCK uses digital pin 76

Very little of this is necessary to successfully use the microSD card slot. What you DO need:

1. Preformated microSD card. This should be formatted using your PC or MAC ideally for DOS FAT16 but can be FAT32.
2. SPI.h library
3. SD.h library

It is important that the SD card be properly formatted. Generally, you will get much better performance by NOT using the utilities on your Macintosh or Microsoft Windoze PC. The SD Card Association has a web site at https://www.sdcard.org/downloads/formatter_4/. They actually have some very good software for formatting these cards that greatly improves access speed for the Arduino.

The two libraries have actually made it into the Arduino IDE so you do not actually have to get them and add them.

```
#include<SPI.h>
#include<SD.h>
```

You also need to establish pin 10 as your **Slave Select** pin

```
pinMode(10, OUTPUT);
```

Note that the library already establishes the SS constant as pin 10.

```
  // see if the card is present and can be initialized:
if (!SD.begin(SS))
      {
      Serial.println("Card failed, or not present");
      return;
      }
  else Serial.println("card initialized.");
```

To use an SD card, we are fortunate to have the SD library written by William Greiman. The SD.h library gives us two new "classes" you can use in your programs. A class provides additional commands specific to the device that are not normally part of the basic programming language. They "extend" the language to access the functions of the new hardware.

The two new classes are the **SD** class and the **FILE** class.

SD CLASS

The SD class provides functions for accessing the SD card and manipulating its files and directories.

- begin()

```
SD.begin(10) //opens communications to SDcard
```

- exists()

```
SD.exists(logger.txt); //Returns TRUE if file
logger.txt exists.  8.3 filenames.
```

- mkdir()

```
SD.mkdir(root/jack/stuff) //Creates directory
```

- open()

```
File dataFile=SD.open(example.txt); //Opens
example.txt file for READ beginning at byte one. Note
definition of dataFile as type File.  This is your
file handle.

File dataFile=SD.open(/JACK/example.txt, FILE_WRITE)
//Opens file for both reading AND writing starting at
END of file
```

- remove()

```
SD.remove(/JACK/example.txt) //Removes file
```

rmdir()

```
SD.rmdir(/JACK/STUFF) //Removes directory
```

FILE CLASS

The File class allows for reading from and writing to individual files on the SD card.

- available()

```
dataFile.available(); //This would return the number
of bytes in the dataFile instance of the File class
(returned by SD.open())
```

- close()

```
dataFile.close(); //Closes the file designated by the
dataFile file handle. Writes any data written to the
file to the SD card.
```

- flush()

```
dataFile.flush(); // Ensures that any bytes written to
the file are physically saved to the SD card. This is
done automatically when the file is closed.
```

- peek()

```
dataFile.peek(); //Returns the value at the current
file position but does not increment file pointer.
```

- position()

```
dataFile.position();//Returns the position of the
current file pointer (long integer)
```

- print()

```
dataFile.print(data, BASE);// Print data to the file,
which must have been opened for writing. Prints
numbers as a sequence of digits, each an ASCII
character (e.g. the number 123 is sent as the three
characters '1', '2', '3'). Optional BASE of BIN, DEC,
OCT or HEX
```

- println()

```
dataFile.println (data, BASE);// Same as print()
but it terminates string with carriage return and
newline characters.
```

- seek()

```
dataFile.seek(pos); //Set file pointer to pos in file
```

- size()

```
dataFile.size(); // Return size of file in bytes
```

- read()

```
dataFile.read(); //Returns next byte in file or -1 if
end
```

- write()

```
dataFile.write(data); // the byte, char, or string
(char *) to write

dataFile.write(buf, len); // buf: an array of
characters or bytes len: the number of elements in buf
```

- isDirectory()

```
dataFile.isDirectory();//Returns true if directory
```

- openNextFile()

```
dataFile.openNextFile(); //If dataFile is dir, returns
next file in directory
```

- rewindDirectory()

```
dataFile.rewindDirectory(); //Resets to first file in
directory
```

```
#include<SD.h>

const int SS = 10;  //Set SS as pin 10

File datafile;

if (!SD.begin(SS))

     {

     Serial.println("Card failed, or not present");

     return;

     }

  else Serial.println("card initialized.");
```

```
dataFile = SD.open("datalog.txt", FILE_WRITE);

  // if the file is available, write to it:
if (dataFile)
     {
      dataFile.println("Hello World…");
      dataFile.close();


if (SD.exists("datalog.txt"))
     {
     Serial.println(datalog.txt exists.");
     SD.remove("example.txt");
     }



 FILE myFile = SD.open("datalog.txt");
  if (myFile)
     {
        while (myFile.available())
          {
               Serial.write(myFile.read());
          }

  myFile.close();


uint32_t volumesize;
Sd2Card card;
SdVolume volume;
SdFile root

  Serial.print("\nVolume type is FAT");
  Serial.println(volume.fatType(), DEC);
  Serial.println();

  volumesize = volume.blocksPerCluster();     //
clusters are collections of blocks
  volumesize *= volume.clusterCount();        //
we'll have a lot of clusters
```

```
  volumesize *= 512;                                    //
SD card blocks are always 512 bytes
  Serial.print("Volume size (bytes): ");
  Serial.println(volumesize);
  Serial.print("Volume size (Kbytes): ");

  Serial.println("\nFiles found on the card (name,
date and size in bytes): ");
  root.openRoot(volume);
```