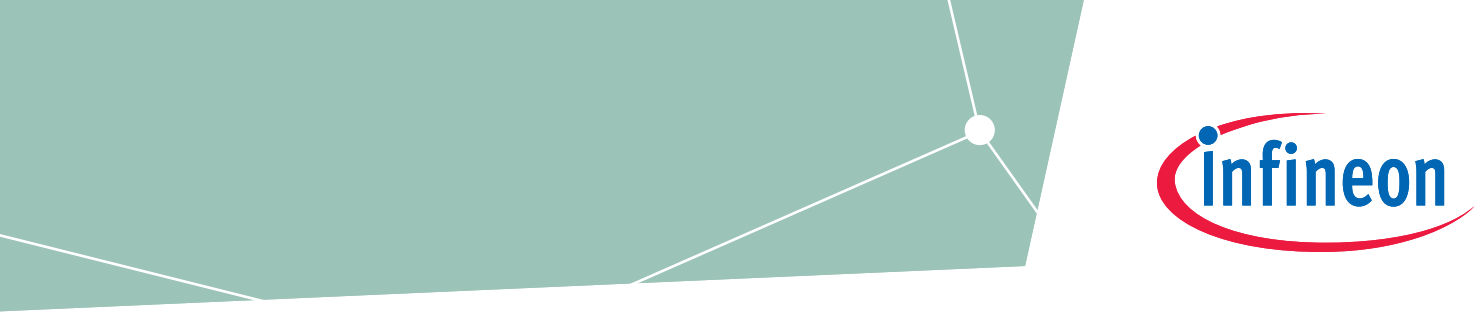


ShieldBuddy Cookbook

Alina Steinberger

Abstract

OK, the Shieldbuddy (124-5257) is basically just another Arduino but is it? While it looks a bit like an Arduino MEGA or Due, you might notice that the CPU is a bit bigger (176 pins) and the PCB is red for danger. Certainly the connector arrangement is the same and it works with the Arduino IDE. However what is not obvious is that the processor is running at 200MHz and hidden inside the LQFP package there are in fact three of them, along with 4MB of FLASH, 128kb of data flash and 500k of RAM.

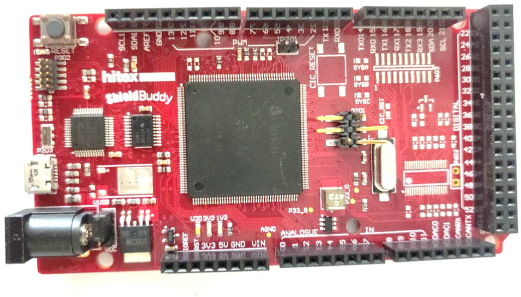


Figure 1: ShieldBuddy Hardware

Most Arduino-style boards use AVR or ARM/Cortex processors which are fine for basic messing about with micros - these chips are everywhere in consumer gadgets and devices. The Shield-Buddy is different, having the mighty Infineon Aurix TC275 processor. These are normally only to be found in state of the art engine management systems, ABS systems and industrial motor drives in your favourite German car. They rarely make it out into the daylight of the normal hobbyist/-maker world and to date have only been known to a select few at Bosch, BMW, Audi, Daimler-Benz etc.. Hitex UK decided to change all that and bring this awesomely powerful chip to a wider audience. A few years ago we had a placement student with us who was using the Arduino Uno to control 4 servos in a model aeroplane. Unfortunately the combination of real time servo control and serial comms to the RC receiver was overloading the 16MHz AVR Atmega 328p, resulting in a lot of grief and frustration. To fix this problem and give more processing power than he could ever need, we decided to drop the Aurix TC275 from a confidential automotive project we had underway onto an Arduino Due format board. The ShieldBuddy was born and the placement student's problems were solved using just one CPU core. The other 2 were just twiddling their thumbs.[1]

Contents

1 Installation

- A PC with Windows Vista or later
- The following programs (Have to be installed in the right order. Further information can be found on the following pages.)
  - The Aurix free toolchain
  - The standard 1.6.11 Arduino IDE
  - The ShieldBuddy Multicore IDE.

1.1 Aurix free toolchain

First of all you have to install the Aurix free toolchain. For that you have to enter the website and follow the instructions:

`http://free-entry-toolchain.hightec-rt.com`

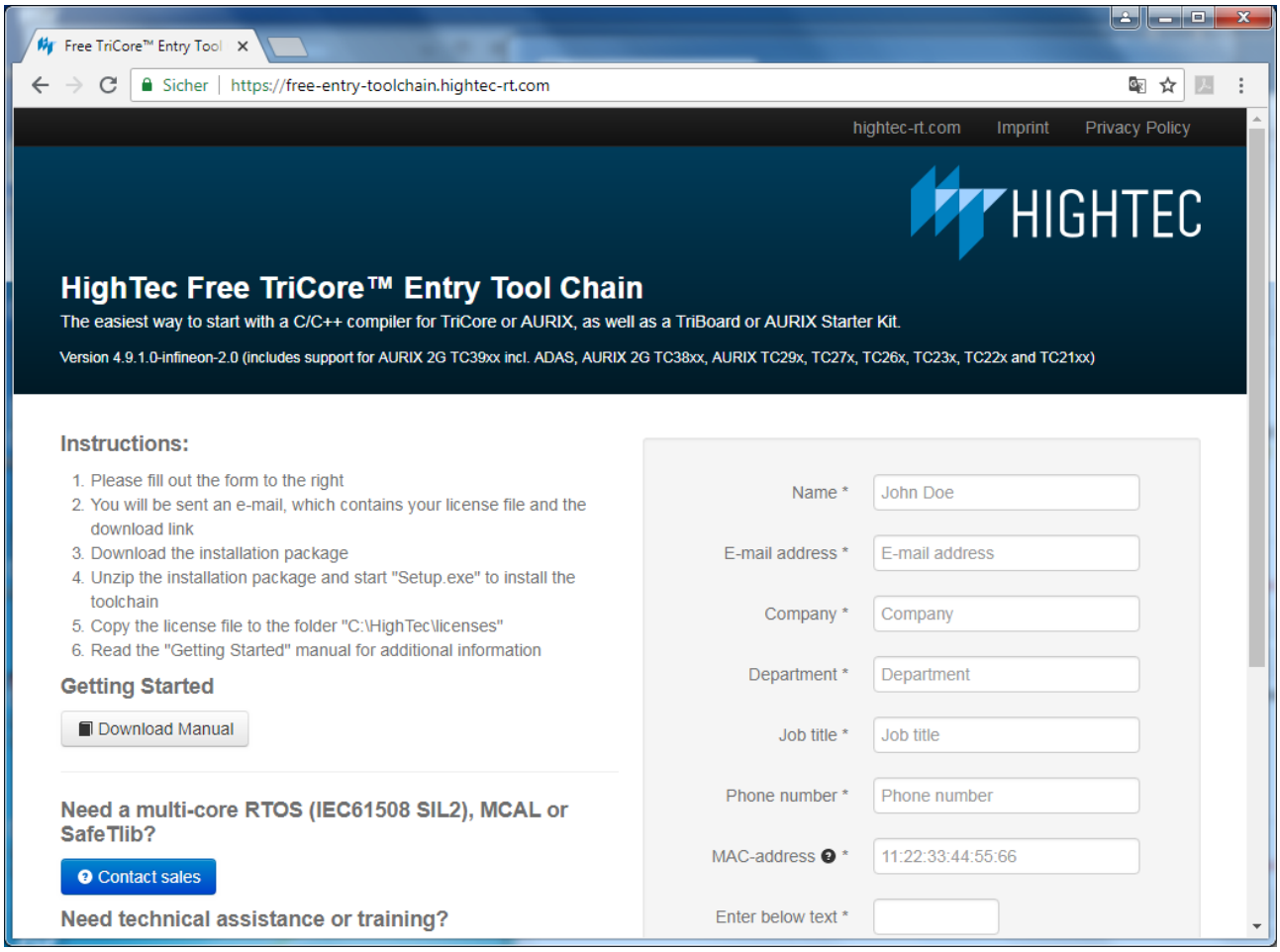


Figure 2: Free Toolchain

Download the installation package on the website or from your email, create the Folders “C:\HighTec\licenses” and save the license.

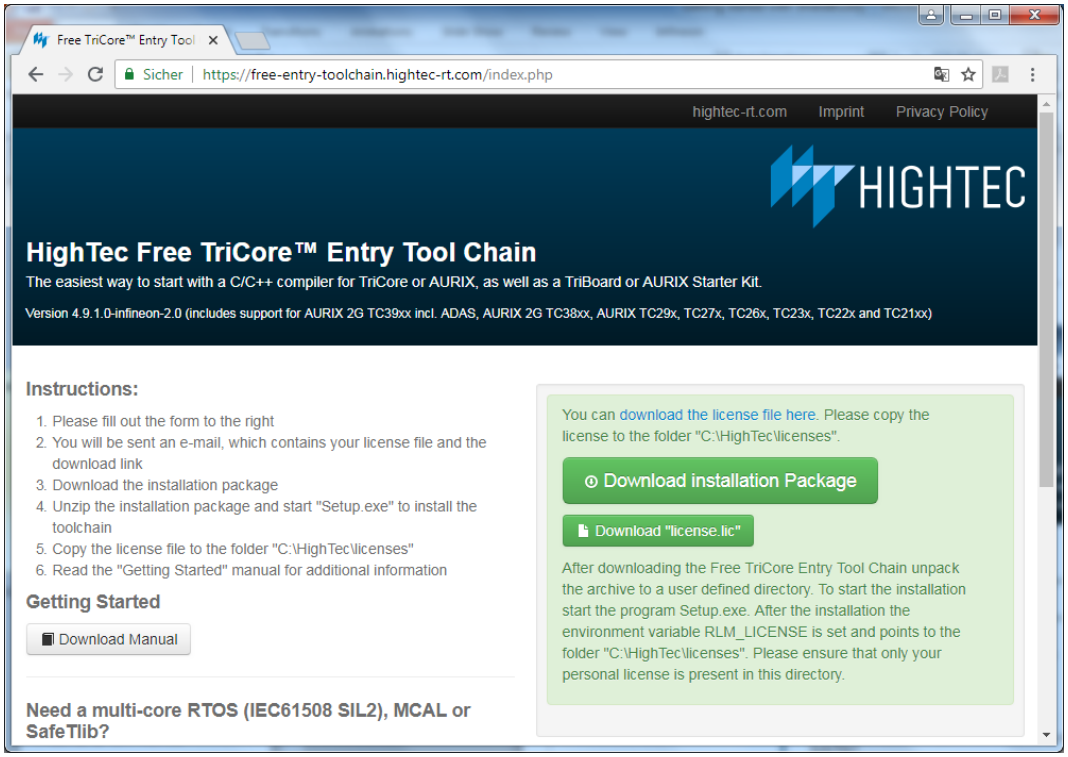


Figure 3: Download Toolchain

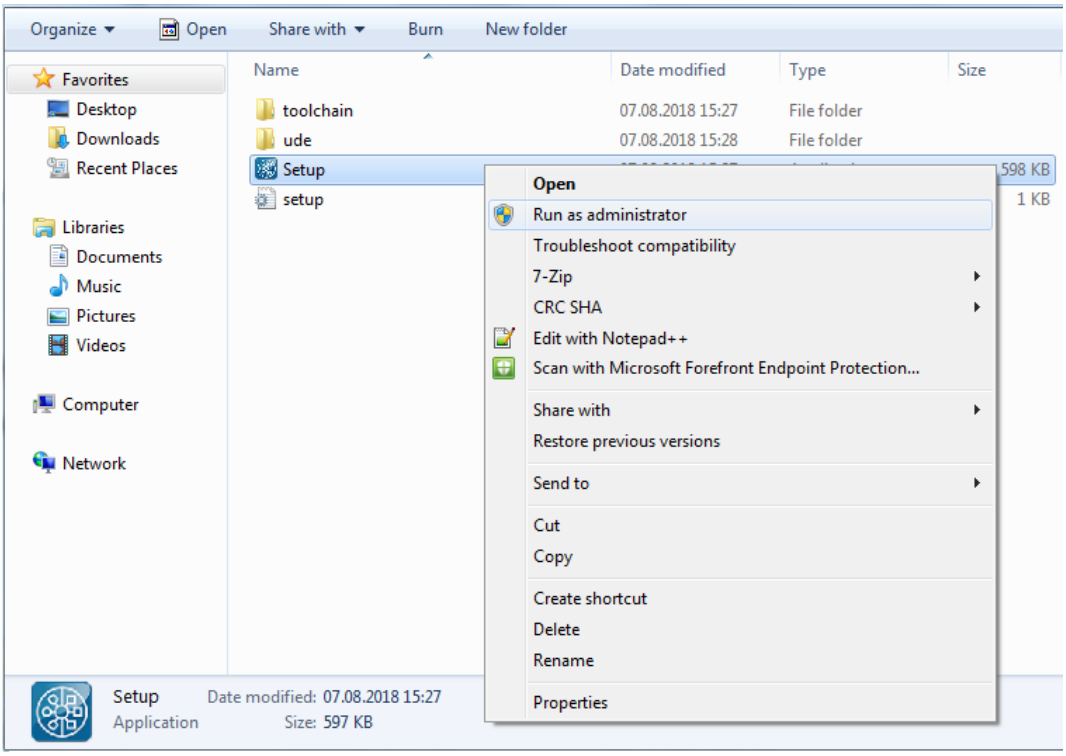


Figure 4: Run as Admin

Unpack the downloaded .zip and run it as administrator as seen in figure 4.

1.2 Arduino IDE

Download the Arduino IDE.

`http://arduino.cc/download.php?f=/arduino-1.6.11-windows.exe`

Run the downloaded file as Administrator and install Arduino IDE to the default directory! You may find that the Arduino IDE thinks that the ShieldBuddy has two COM-ports. If this happens, choose the one with the higher number!

1.3 ShieldBuddy Multicore IDE

Use the following link to download the ShieldBuddy Multicore IDE.

`http://www.hitex.co.uk/fileadmin/uk-files/downloads/ShieldBuddy/ShieldBuddyMulticoreIDE.zip`

Unzip this to a temporary directory using the zip password “ShieldBuddy” (you will need a software like 7-zip or WinRAR for that).

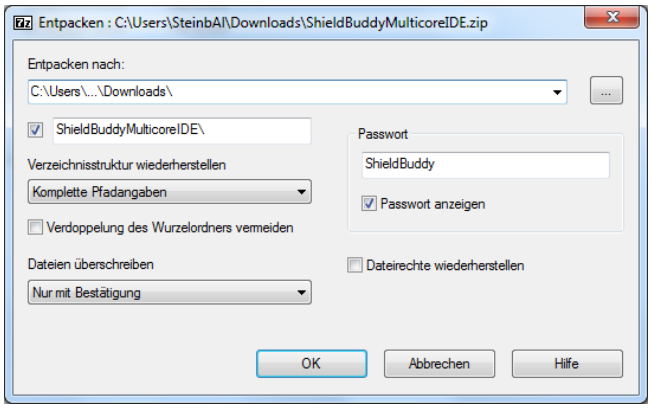


Figure 5: unzip multicore

Run the installer as administrator and use the password “ShieldBuddy” to copy the IDE onto your PC. After installing the ShieldBuddy IDE the Installer for Infineon Memtool will launch automatically If you are using a 32 bit Windows version change the destination location of Memtool to: C:\Program Files (x86)\Infineon\Memtool 4.6 For a 64 bit Windows version the default directory should work.

2 First Steps

2.1 Connect your board

To connect your ShieldBuddy you need a micro USB cable. (The cable most Android smartphones are charged with). The USB connection with the PC is necessary to program the board and not just to power it up. The ShieldBuddy can draw power from either the USB or an external power supply. To be able to program the board it must be adjusted to be supplied over USB (the Jumper must be connected the way it is shown below). Three LEDs (white, yellow, blue) should go on.

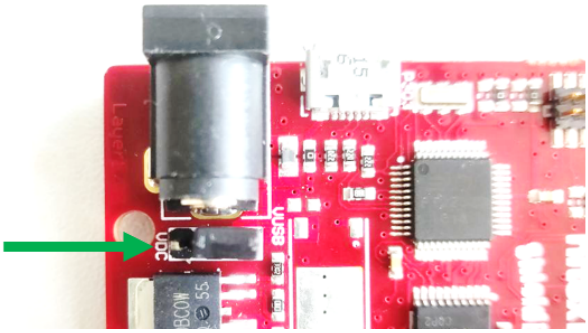


Figure 6: supply jumper

2.2 Open your first sketch

The easiest way is to program the ShieldBuddy to use the Arduino IDE. Most of the commands stay the same, there are just some small changes (like instead of Serial.println(“ ”) for output on the Serial Monitor SerialASC.println(“ “) is used) but more about that later. Now let’s start at the beginning. First we open the main_serial Sketch as shown below. This program only does one thing. Read in data from the Serial Monitor and returns the read information back to the serial monitor.

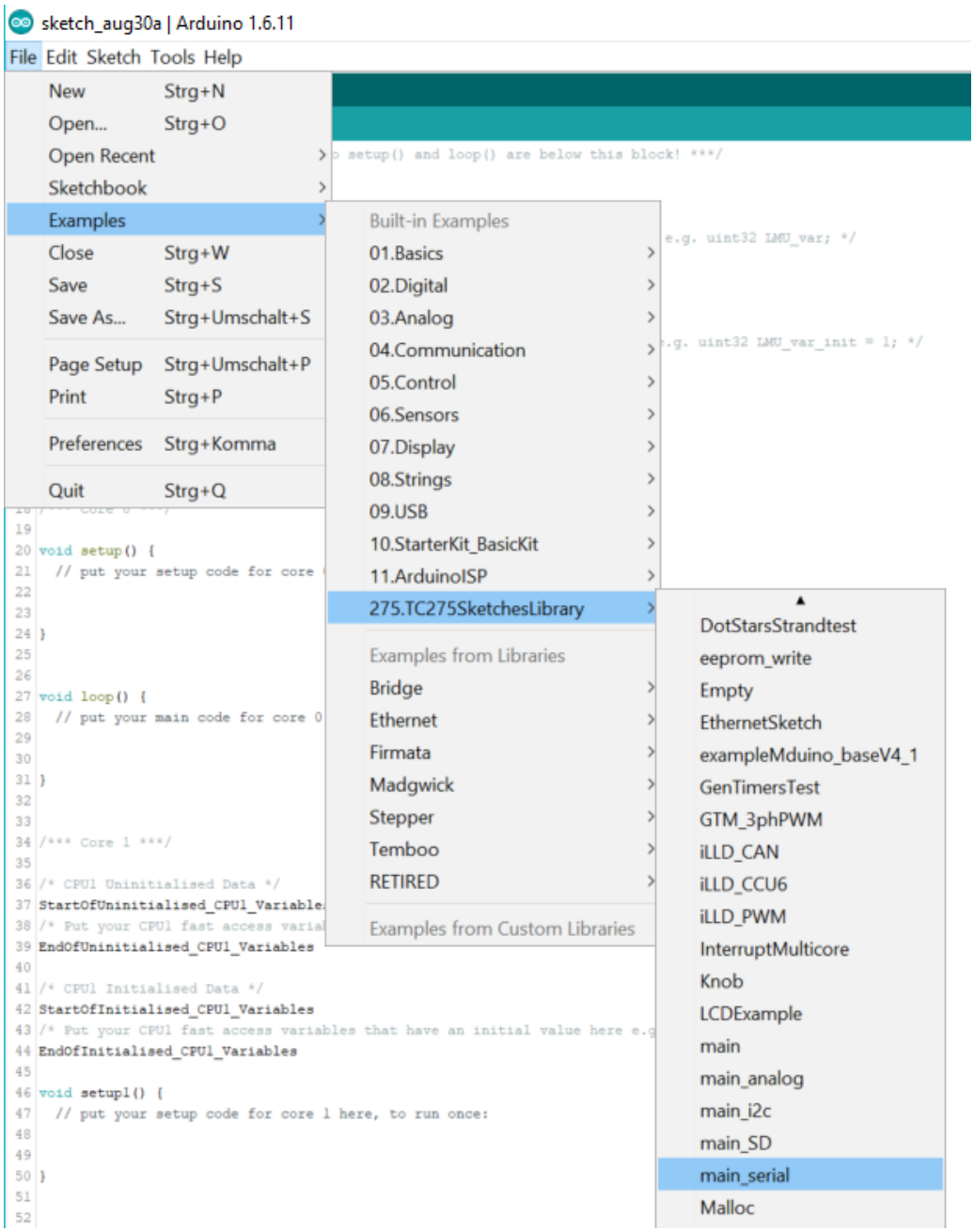


Figure 7: open example

2.3 Upload your first sketch

Before uploading the sketch on the board, it is necessary to select which Board and which Port you are using. You'll need to select the entry in the Tools > Board menu that corresponds to your board. You have a ShieldBuddy TC275.

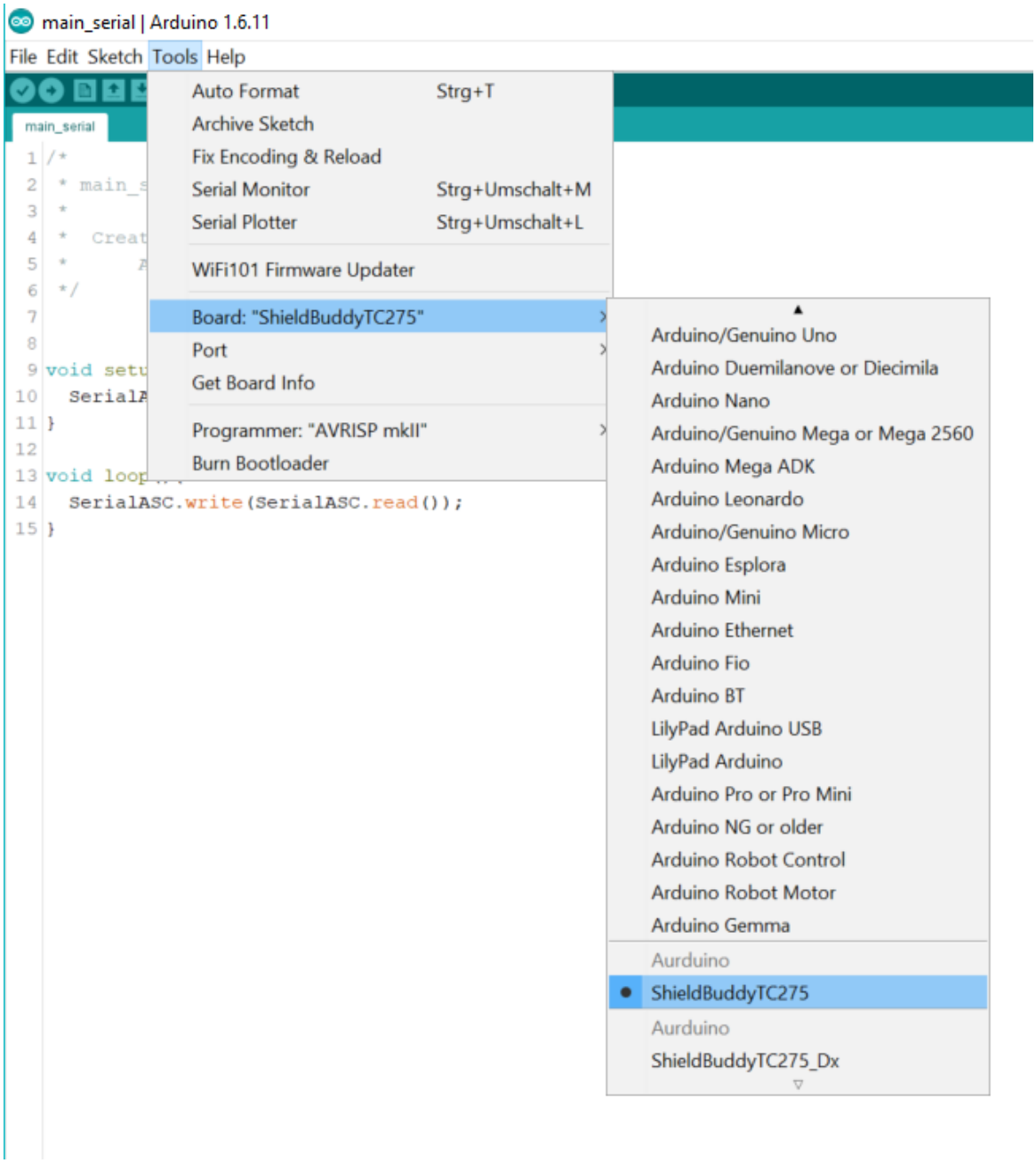


Figure 8: select board

3 Serial Ports

The Arduino has the Serial class for sending data to the UART which ultimately ends up as a COM port on the host PC. The ShieldBuddy has 4 potential hardware serial ports so there are now 4 Serial classes. The default Serial class that is directed to the Arduino IDE Serial Monitor tool becomes SerialASC on the ShieldBuddy. Thus Serial.begin(9600) becomes SerialASC.begin(9600) and Serial.print("Hi") becomes SerialASC.print("Hi") and so on.

The serial channels are allocated as per:

SerialASC	Arduino FDTI	USB-COM	Micro USB
Serial1	RX1/TX1 Arduino	J403	pins 17/16
Serial0	RX0/TX0 Arduino	J403	pins 15/14
Serial	RX/TX Arduino	default J402	pins D0/D1

Any of the serial channels can be used from any core but it is not a good idea to access the same serial port from more than one core at the same time – see the later section on multicore programming.

4 Using more cores

In the main_serial example only the core0 is used. When you want to use more than just one core you need to take the standard structure given when you create a new sketch.

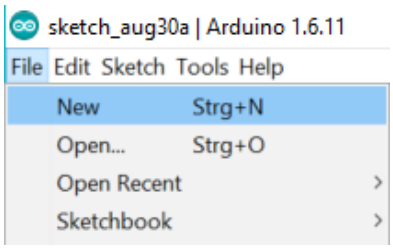


Figure 9: create sketch

The difference between the three cores is, that the core0 is a bit slower and has less Interrupts then the other cores. Core1 and core2 contain the same functionalities and are equally fast. The standard structure consists of the following parts:

- LMURam_Variables
- Setup and loop for core0
- CPU1_Variables
- Setup and loop for core1
- CPU2_Variables
- Setup and loop for core2

4.1 LMURam_Variables

This part must hold the global variables,which are the variables each core must be able to access. There are two different sections in the LUMRam_Variables the uninitialized variables, which don't have to have a certain start value. More about the multicore memory map see ShieldBuddy user manual p. 25 [2].

```
StartOfUninitialised_LMURam_Variables
int global;
EndOfUninitialised_LMURam_Variables
```

And there also are the Initialised variables which need to have a certain start value.

```
StartOfInitialised_LMURam_Variables
int global=0;
EndOfInitialised_LMURam_Variables
```

4.2 Setup and Loop for core0

The setup and loop function are working the same way as for normal Arduino. The setup is called once at the start of the program and the loop is repeated endlessly. In this core there are less interrupts available (chapter 5) and it is a bit slower than the other cores.

4.3 CPU1_Variables

This part is structured the same way as the LMURan_Variables with the uninitialized and initialized variables, the only difference that only the core1 can access variables declared here.

4.4 Setup and Loop for core1

This is again the same as for core0.

4.5 CPU2_Variables

Here variables are declared which are only needed in core2.

4.6 Setup and Loop for core2

This is the same as for core0.

5 Interrupts

There are different interrupts available for the ShieldBuddy.

5.1 CoreInterrupt

These interrupts are mostly used to communicate between cores. An easy example for that would be if one core is inputting data and one is displaying data. When the first core is finished inputting the wanted data, it interrupts the second core at whatever it is doing that moment and starts the interrupt service routine ISR where the second core outputs the data. This function is very important when memory or certain peripherals are accessed by more than one core. For example, it is not possible that two cores access a variable at the same time. So, it is common that one core is interrupting the next core when it is finished writing the variable and the next core can access it afterwards.

```
/* Create an interrupt in core 1 - is called in the setup */
CreateCore1Interrupt(Core1IntService);
/* Trigger interrupt in Core1 now! */
InterruptCore1();
```

5.2 TimerInterrupt

Timer interrupts are used to trigger an event continuously at a certain time rate which can be seen in the codesnippet below,

```
/* 10ns per bit count */
CreateTimerInterrupt(ContinuousTimerInterrupt,10000,STM0_inttest);
```

or they just trigger an event once after a certain time has passed.

```
/* Run STM0_inttest once, 100us in the future */
CreateTimerInterrupt(OneShotTimerInterrupt, 10000, STM0_inttest);
```

Here you can see a difference between core0 and the cores 1 and 2. The core1 and core2 both have two timer interrupts available and are created as seen below. [2]

```
/* Make STM1_inttest0() function run every 100us */
CreateTimerInterrupt0_Core1(ContinuousTimerInterrupt, 10000,
   STM1_inttest0);
```

5.3 PinInterrupt

A pin interrupt is triggered when a certain event occurs. The following Interrupt is triggered when at the in_pin any change occurs and the function PIN_interrupt() is called.

```
attachInterrupt(digitalPinToInterrupt(_in_pin), PIN_Interrupt, CHANGE););
```

At the ShieldBuddy it is possible to create PinInterrupts at the following Pins:

2, 3, 15, 18, 20, 52

5.4 Enablina/Disabling Interrupts

It is possible to disable all interrupts using:

```
noInterrupts();
```

This will also stop the delay() and other timer-related functions. Interrupts can be re-enabled using:

```
interrupts();
```

More general information about Interrupts can be found in the ShieldBuddy user manual p.20-24 [2]

6 Make your own library or class

A lot of people do not know that in ArduinoIDE it is possible to use classes. The reason for that might be that Arduino calls them libraries and there is no possibility to create the class files directly in the Arduino IDE surface. You need to use another code editor like NotePad++ and then you can create the C++ class files .cpp and .h . After creating the class files, you need to add them to the same folder then your .ino file and when you reopen your Arduino-file the class files can also be accessed and edited from this interface.

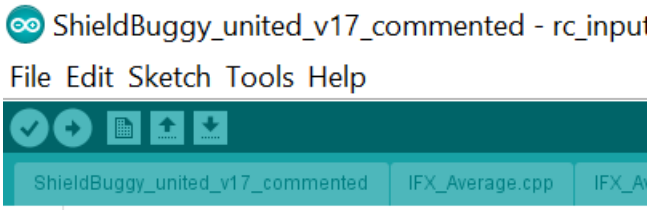


Figure 10: classes in Arduino IDE

More information about Arduino libraries and classes can be found here

`https://www.arduino.cc/en/Hacking/LibraryTutorial`

7 SPI comnnection

See manual p.33 & 34 [2]

8 I2C

Manual p 32 [2]

9 EEProm

Manual p 33 [2]

10 CAN

Manual p 30-31 [2]

11 PWM/ analog pins

Manual p 30-31 [2]

Like the Arduino, the ShieldBuddy uses PWM to generate analog voltages. The PWM frequency is only around 1kHz on the Arduino. The ShieldBuddy frequency is 390kHz when using 8-bit resolution. Whilst this is great for AC waveform generation, audio applications etc., it can be too high for some power devices used for applications like motor control. The useArduinoPwmFreq() function will set the PWM frequency to 1.5kHz so that motor shields etc. should work without problems.

11.1 Custom PWM frequency

It is also possible to set any PWM frequency using the useCustomPwmFreq() function:

```
/* Use 4000Hz carrier */
useCustomPwmFreq(4000);
```

The maximum frequency that may be set is 390kHz. The minimum is 6Hz. If you want to change the PWM frequency after calling analogWrite(pin,dutycicle), use the following functions:

```
AnalogOut_2_Reset();
// Allow analog channel 2 to be altered
useCustomPwmFreq(3900);
// Change to 3900Hz carrier
analogWrite(2, 128);
// Write 50% duty ratio at 3900Hz carrier
```

11.2 ADC Resolution

The default resolution for ADC conversion results is 10 bits, like on an ordinary Arduino. On the ShieldBuddy you can set 8-bit or 12-bit conversions if required, using the analogReadResolution () function.

```
/* Set default VADC resolution 10 bits */
analogReadResolution (10u);
```

To set 12-bits of resolution,:

```
analogReadResolution (12u);
```

To set 8-bits:

```
analogReadResolution (8u);
```

11.3 Fast Update Of AnalogOut() Function

In situations where the duty ratio has to be updated very frequently, it is often better to update just the duty ratio register in the PWM system for the particular channel in use rather than using the normal analogWrite(). This can be done using macros like:

```
AnalogOut_2_DutyRatio = 128;
```

The value used must be within the range allowed by the resolution you are using. For the default 8-bit, this is 0- 255; for 10-bit this is 0-1023 and so on. For this to work, you must have used the normal analogWrite(pin, dutycicle) for that channel at least once e,g.

```
analogWrite(2, 128);
```

11.4 DAC0 and DAC1 pins

These Arduino pins are specifically for accurate digital to analog conversion. They have a fixed 14-bit resolution (0-16383) and a 6.1kHz PWM frequency.

```
analogWrite(DAQ0, 8192); // Set 2.5V on DAC0 pin
analogWrite(DAQ1, 4096); // Set 1.25V on DAC1 pin
```

12 Including extern libraries

To include extern libraries you need to have the library saved on your computer. To include it go to Sketch, include Library and Add .zip library. Then you have to select the memory location of the wanted library.

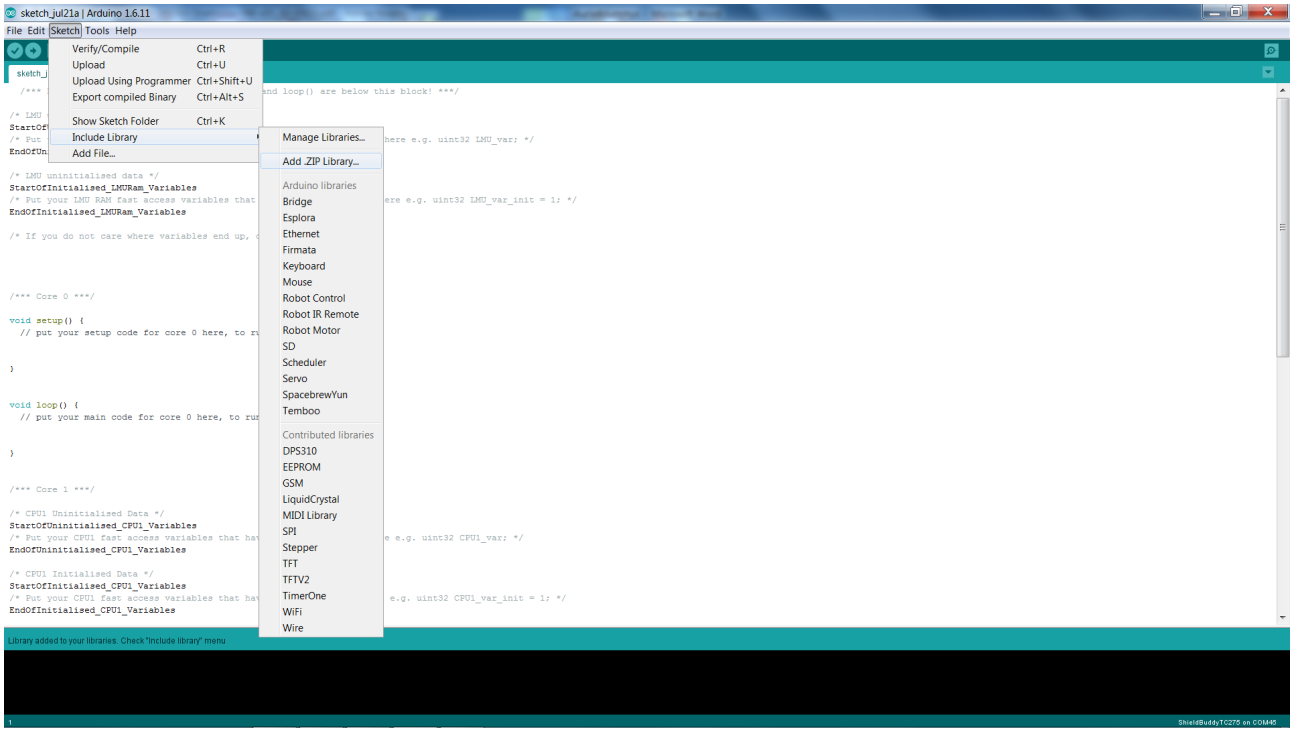


Figure 11: include library

To check the status of the library go to “Sketch/Include Library/Manage Library”. It will appear in the list as “Installed” and possibly you can also select a version in the library.

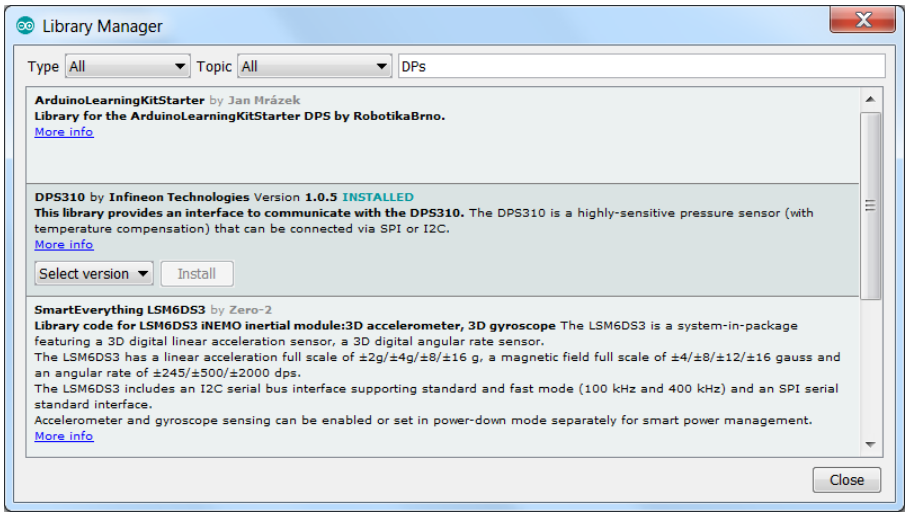


Figure 12: Library manager

13 Projects for the Shieldbuddy

Starting with simple easy Project to get to know how to work with the ShieldBuddy and leading to more and more complex ones.

13.1 LED blink

In this chapter a small example is given how a LED blink routing can be realized. This is accomplished by using the functions digitalWrite() and delay();

```
int ledPin = 8;   // Integer variable for ledPin, initialized at 8

/*** Core 0 ***/
void setup() {
pinMode(ledPin, OUTPUT);  // Initialize the digital pin as an output
}

void loop() {
digitalWrite(ledPin, HIGH);// turn the LED on --> HIGH is 5 V on the pin
delay(250);                      // wait for 250 ms
digitalWrite(ledPin, LOW); // turn the LED of --> LOW is 0 V on the pin
delay(1000);
}
```

In the next picture the output of the LED pin can be seen.

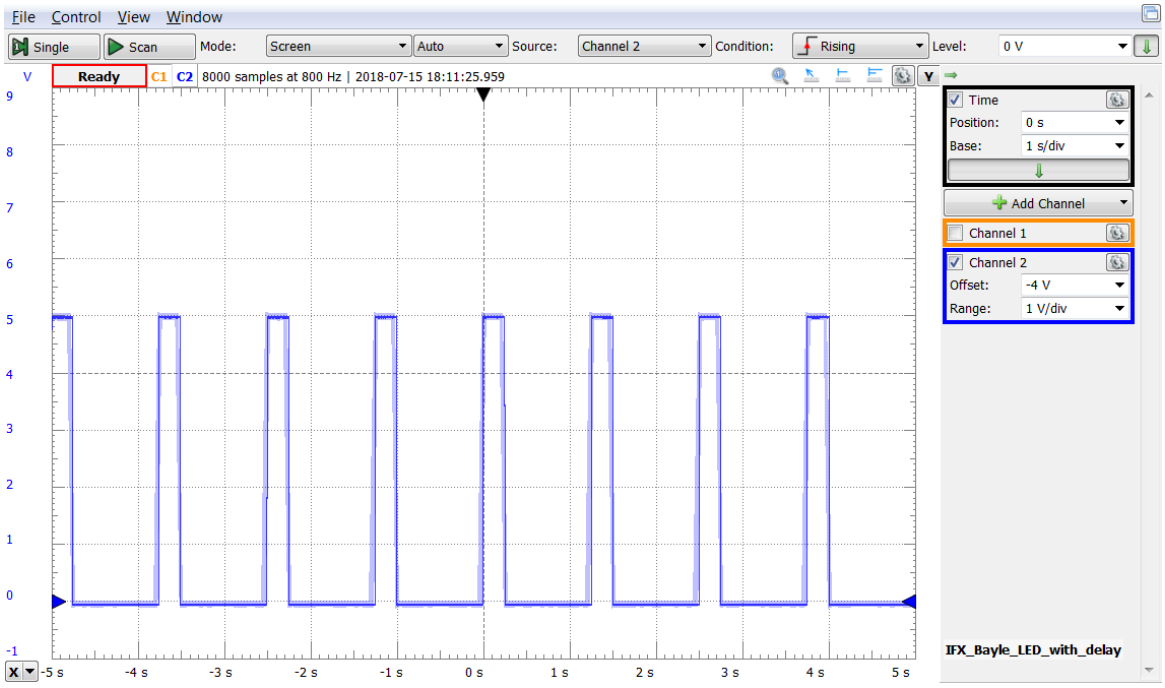


Figure 13: LED pin putput

13.2 Traceroutine with serial output

The example of the last chapter is used further and we want to output the status of the LED via the serial connection to the computer.

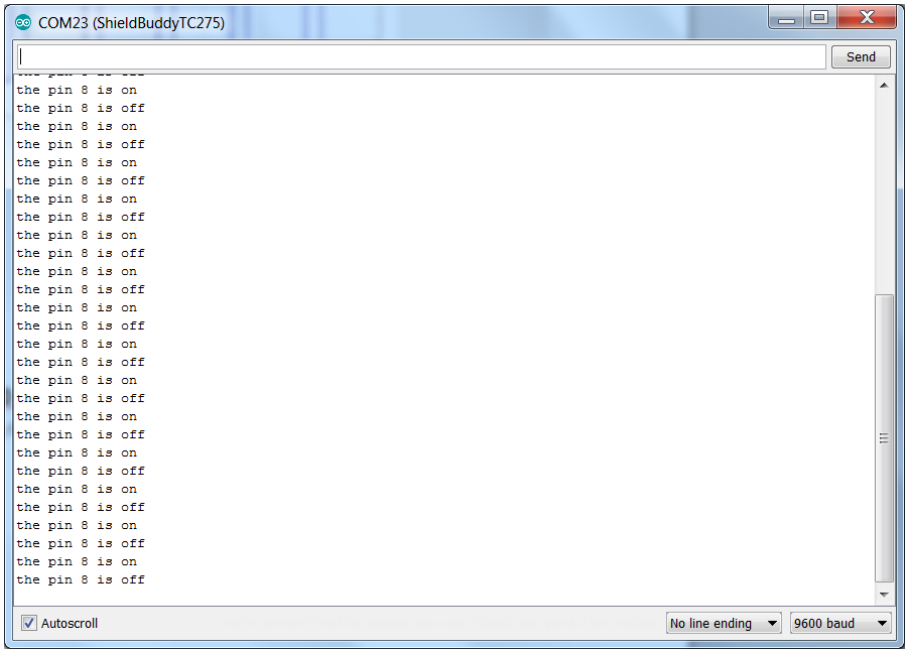


Figure 14: traceroutine

Compared to the use of a normal Arduino board we have to use SerialASC here, instead of Serial.

```
int ledPin = 8;   // Integer variable for ledPin, initialized at 8

/*** Core 0 ***/
void setup() {
pinMode(ledPin, OUTPUT);  // Initialize the digital pin as an output
Serial.begin(9600);    // Serial communication setup at 9600 baud, THE
   COMMAND IS DIFFERENT FROM STANDARD ARDUINO
}

void loop() {
digitalWrite(ledPin, HIGH);   // turn the LED on --> HIGH is 5 V on the
   pin
Serial.print("the pin ");
Serial.print(ledPin);      // Print pin value
Serial.println(" is on");  // println gives a carriage return and a
   newline
delay(250);                   // wait for 250 ms
digitalWrite(ledPin, LOW);    // turn the LED of --> LOW is 0 V on the
   pin
Serial.print("the pin ");
Serial.print(ledPin);      // Print pin value
Serial.println(" is off");
delay(1000);
}
```

13.3 Morse-SOS Functions

Generally, Functions in ShieldBuddy Programming are the same as in C and with the Arduino. In this chapter a simple use case for functions is shown by creating a Morse SOS signal on the yellow LED of the ShieldBuddy.

```
int pin = 13;   // The pin 13 connected to the yellow led

void setup()
{
pinMode(pin, OUTPUT);
}

void loop()
{
dot(); dot(); dot();
dash(); dash(); dash();
dot(); dot(); dot();
delay(3000);
}

void dot()
{
digitalWrite(pin, HIGH);
delay(250);
digitalWrite(pin, LOW);
delay(250);
}

void dash()
{
digitalWrite(pin, HIGH);
delay(1000);
digitalWrite(pin, LOW);
delay(250);
}
```

In the Codesnippet above the functions dot(), for a short blinking signal and dash(), for a long blinking signal, can be seen.

13.4 Morse-SOS Library

As mentioned in chapter 6 it is possible to create classes in Arduino by creating libraries. This example shows how to implement the Morse-program with a library. There is also the definition of a class of objects in the Morse.h library. The Morse.h library is stored in the same folder as the .ino file. In this case the correct command is #include “Morse.h”, not #include <Morse.h>. The following codesnippet shows the .ino program where the created library is implemented.

```
#include "Morse.h"  /* That's a difference to the original, they used
   brackets
in this case he lib is in the local folder */
Morse morse(13);    //  Generate an object "morse" of the class "Morse"

void setup()
{
}

void loop()
{
morse.dot(); morse.dot(); morse.dot();
morse.dash(); morse.dash(); morse.dash();
morse.dot(); morse.dot(); morse.dot();
delay(3000);
}

Every Libraray or class in Arduino like in C++ consists of a .cpp and a .
   h file.
The content of the library Morse.h is:

#ifndef Morse_h
#define Morse_h

#include "Arduino.h"

class Morse
{
public:
Morse(int pin);
void dot();
void dash();
private:
int _pin;
};

#endif
```

And there is the file Morse.cpp:

```
#include "Arduino.h"
#include "Morse.h"

Morse::Morse(int pin)
{
pinMode(pin, OUTPUT);
_pin = pin;
}

void Morse::dot()
{
digitalWrite(_pin, HIGH);
delay(250);
digitalWrite(_pin, LOW);
delay(250);
}

void Morse::dash()
{
digitalWrite(_pin, HIGH);
delay(1000);
digitalWrite(_pin, LOW);
delay(250);
}
```

13.5 Use of global, local and static variables

In this chapter you can find some examples how to use the different kinds of variables available with the ShieldBuddy and where the differences are. In this example we show the differences between global, local and static variables. When you try out the code yourself you will see that the value of static local variables is persistent between the calls of a function. The global variable is saved after the end of every function and you can access the actual value from everywhere. And the local variable is gone after the function has ended and a new variable with the same name is created when the function is started again.

```
int myGlobalVariable = 1;                    // Global variable, can
   changed in every function
// const int myGlobalStaticVariable = 1;     // This global function can
   not be changed

void setup() {
SerialASC.begin(9600);
}


void loop() {
SerialASC.print("the global variable is: ");
SerialASC.println(myGlobalVariable);
SerialASC.print("the modified local variable is: ");
SerialASC.println(myFunctionLocal(myGlobalVariable));   // call of the
   function in the println-command
SerialASC.print("the modified static variable is. ");
SerialASC.println(myFunctionStatic(myGlobalVariable));  // call of the
   function in the println-command

}

int myFunctionLocal(int argumentLocal) {        // the result of the
   function is used in the print statement --> int
int aLocalVariable;
return (aLocalVariable += argumentLocal);     // every call of the
   function add up myGlobalVariable
}

int myFunctionStatic(int argumentStatic) {      // the result of the
   function is used in the print statement --> int
static int aStaticVariable;
return (aStaticVariable += argumentStatic);   // every call of the
   function add up myGlobalVariable
}
```

No you can try to change the value of the global function within the function and then change the type to const and look at the different outputs you get.

13.6 Generate a PWM for a Servo

Simple program to generate a PWM Signal with the delay function.

```
/* simple program for Servo without the Servo lib
*  from "Arduino notebook"
*  Important program because the importance of timings is visible. Higher
    CPU load would shift the pattern
*  --> Interrupt or task scheduler
*/


int servoPin = 13;       // servo connected to digital pin 2
int myAngle;            // angle of the servo roughly 0 - 180
int pulseWidth;         // servoPulse function variable

void setup()
{
pinMode(servoPin, OUTPUT);    // sets pin2 as output
}

void servoPulse(int servoPin, int myAngle)
{
pulseWidth = (myAngle * 10) + 600;  // determines delay in microseconds
digitalWrite(servoPin, HIGH);       // set servo high
delayMicroseconds(pulseWidth);      // microseconds pause
digitalWrite(servoPin, LOW);        // set servo low
}
void loop()
{
//servo starts at 45 deg and rotates to 135 deg
for (myAngle=45; myAngle<=135; myAngle++)
{
servoPulse(servoPin, myAngle);              // send pin and angle
delayMicroseconds(19400 - 10 * myAngle);    // refresh cycle, modified,
   empirical formula
}

for (myAngle=135; myAngle>=45; myAngle--)
{
servoPulse(servoPin, myAngle);              // send pin and angle
delayMicroseconds(19400 - 10 * myAngle);    // refresh cycle, modified,
   empirical formula
}
}
```

On the scope you will get the following output:

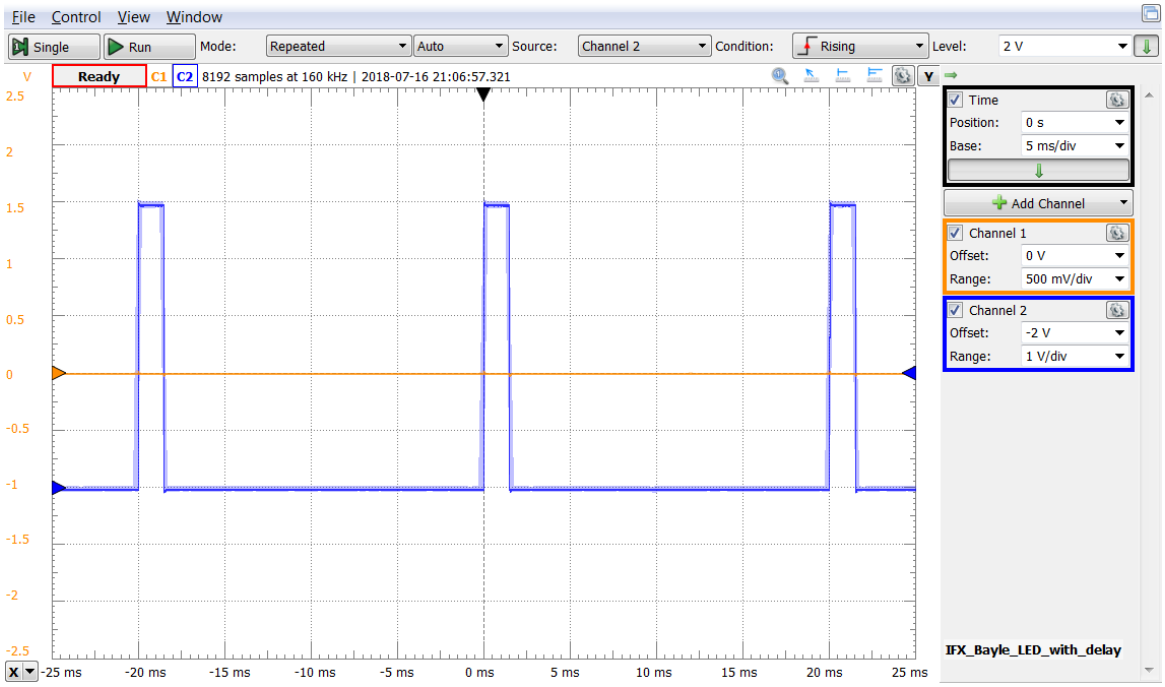


Figure 15: PWM for servo

The high signals are getting wider and smaller periodically.

13.7 Pressure Sensor DPS310

The communication with the pressure sensor of infineon works over I2C. The Homepage of the pressure sensor is:

`https://www.infineon.com/cms/de/product/sensor/barometric-pressure-sensor-for-consumer-applications/dps310/`

The latest version of the Arduino library (.zip) can be downloaded from:

`https://github.com/Infineon/DPS310-Pressure-Sensor`

To install the library download as a .zip file open the Arduino IDE and click “Sketch/Include Library/Add .ZIP Library” and include it as described in the chapter 12 “Include extern Libraries”.

13.8 Engine speed sensor

To measure the rotation speed of an engine without inbuild sensor it is possible to build a sensor yourself. At the used engine the gear shaft is visible and as seen in the picture below, a button magnet can be glued to it sideways

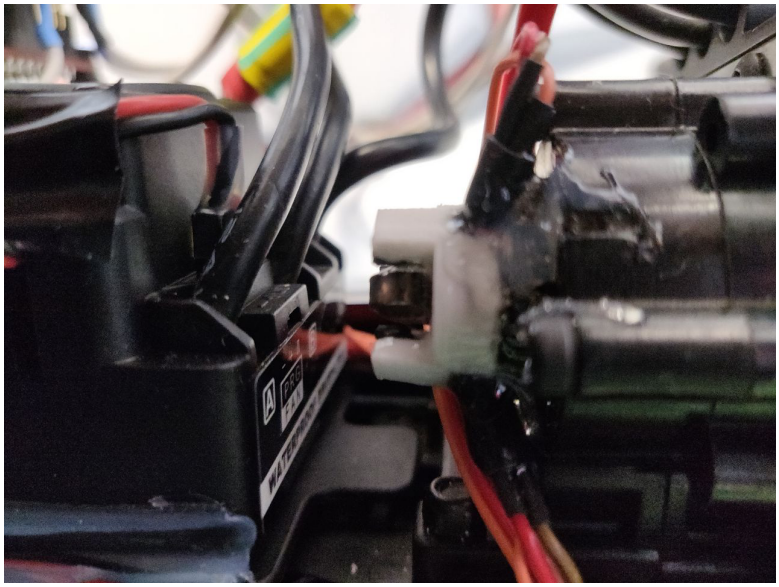


Figure 16: Installation of the engine sensor

Because we want to know the speed and direction of the engine we need three hall effect sensors fixed around the spinning magnet. One sensor is needed to determine the speed of the engine and the two others, depending which one is active when the sensor for the speed turns on, tell us which direction the engine is spinning.

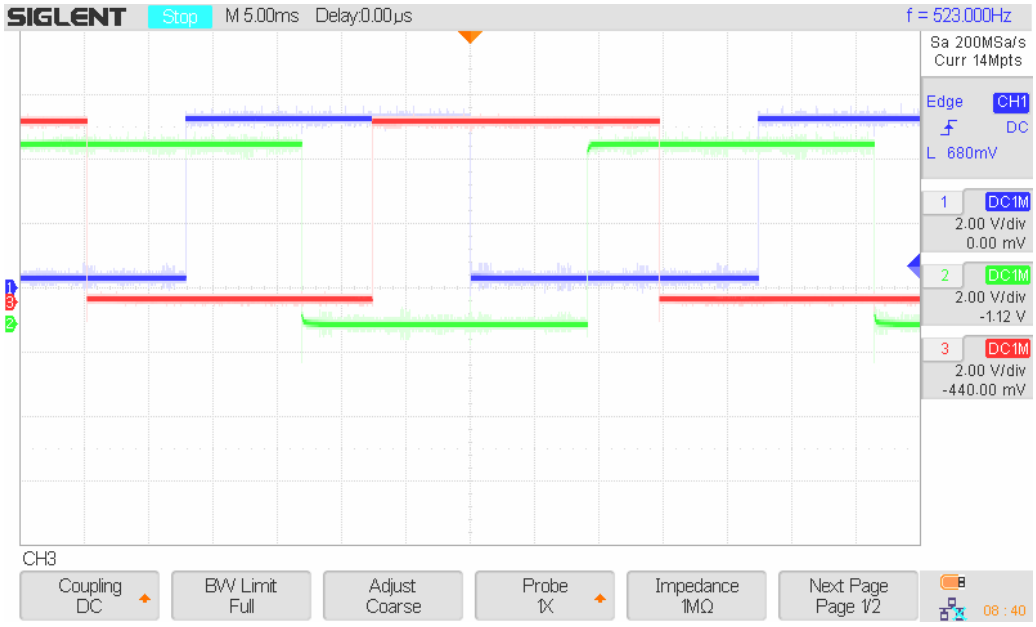


Figure 17: output of the hallsensors

In the software it is implemented with a pin interrupt which measures the time between the interrupts and saves it as period. With the period, the gear ratio and the tyre circumference, the speed of the vehicle can be calculated. And the sign of the speed can be added when the other value of the other two sensors are read in.

```
int ev_pin_A=20;//pin interrupt must be possible on this pin
int ev_pin_B=39;
int ev_pin_C=41;
enum engine_direction{
forward,backward,standing
};
engine_direction dir=standing;

int engine_velocity=0;
int engine_period=0;
unsigned long last_irq0=0;

void setup() {
// put your setup code for core 0 here, to run once:
// put your setup code for core 1 here, to run once:
pinMode(ev_pin_A, INPUT);
pinMode(ev_pin_B, INPUT);
pinMode(ev_pin_C, INPUT);
attachInterrupt(digitalPinToInterrupt(ev_pin_A), engine_isr, RISING);
SerialASC.begin(9600);
SerialASC.println("setup");
}

void loop() {
// do something with the input velocity
}

static void engine_isr()
{
  unsigned long act_irq0=micros();
  engine_period = act_irq0 - last_irq0;
  last_irq0=act_irq0;
  engine_velocity=11/((float)engine_period)*pow(10,6);

  if(digitalRead(ev_pin_B)==HIGH)
  {
  dir=backward;
  engine_velocity=engine_velocity*(-1);
  }else if(digitalRead(ev_pin_C)==HIGH)
  {
  dir=forward;
  }else{
  dir=standing;
  }
}
```

13.9 Flying/Driving ShieldBuddy

The flying ShieldBuddy is a shield for the ShieldBuddy which holds various sensors and modules which can be easily connected by just mounting the shield.

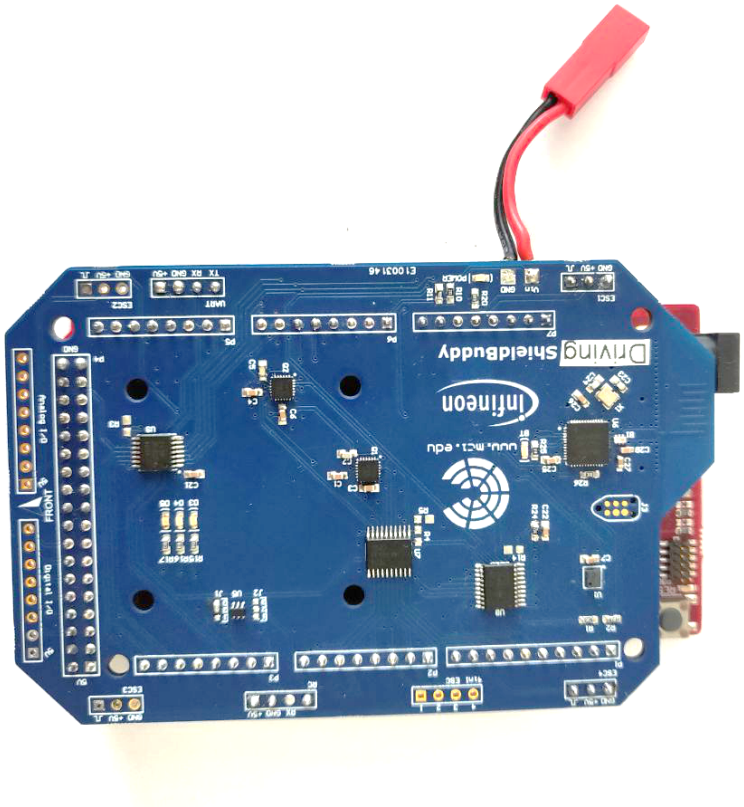


Figure 18: Flying/Driving ShieldBuddy

13.9.1 Nordic Bluetooth

The Nordic Bluetooth module creates a low energy Bluetooth connection which can be used with several other low energy capable devices. A tested out example for a receiver is a smartphone with the application adafruit bluefruit. To send information over Bluetooth when the flying ShieldBuddy is plugged in the ShieldBuddy is simple. It is only necessary to send the data over the Serial1 interface with a baud rate of 38400. The only thing to mind is that at the start of the program the ShieldBuddy must send something to the Bluetooth module or it will go into sleep mode.

An example for using the nordic bluetooth chip on the flying ShieldBuddy is given in the following codesnippet. Here data from the Serial Monitor is read in and send via bluetooth. It can be received by the earlier mentioned smartphone application or other devices.

```
void setup() {
// put your setup code for core 0 here, to run once:
Serial.begin(38400);
SerialASC.begin(38400);
}
```

```
void loop() {
// put your main code for core 0 here, to run repeatedly:
if(SerialASC.available()>0){
Serial.write(SerialASC.read());
}else{
Serial.write(0);
}
delay(200);
}
```

13.9.2 IMU

No working program was created till now so hopefully further information will follow in future.

13.10 Distance2Fly Radar

The Distance2Fly Radar can recognize a target that lays between 0,6 and about 20 metres. Additionally, it is able to tell what speed a target is moving towards or away from the radar.

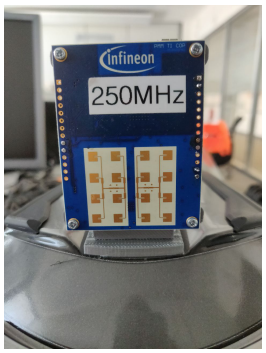


Figure 19: Distance2Fly Radar

13.10.1 Communication Protocol

To communicate with the ShieldBuddy the Distance2Fly Radar uses an UART connection with the baud rate of 115200. To get the data from the radar, the Radar data_in pin must be connected with a free RX-pin of the ShieldBuddy

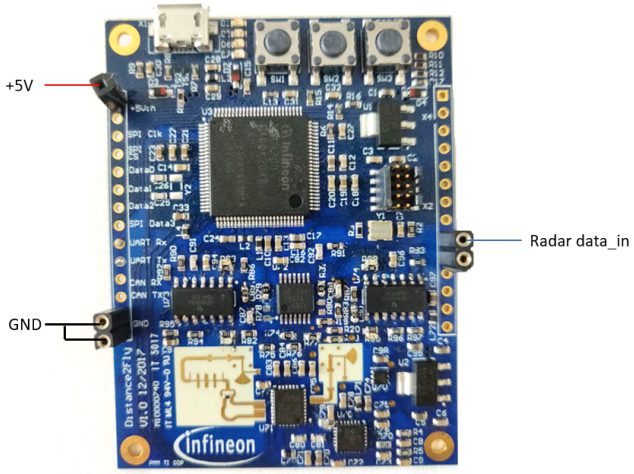


Figure 20: Pins on the distance2fly radar

To transmit the obtained information the radar surrounds the sent data with a start- (0xaa) and stopbyte (0xbb). For a good resolution it uses two bytes per dataset to transmit it (that is why in the following transmitted dataset some bytes have a hb, high byte and lb, low byte in their naming). Furthermore, to recognize transmission errors a checksum is also included in the sent data. In the end the transmitted dataset looks as followed.

<0xaa><distance_hb><distanc_lb><velocity_hb><velocity_lb><checksum><0xbb>

The checksum is calculated by simply oring the transmitted distance and velocity bytes.

13.10.2 Received Data

To process the used data in each case the high byte must be shifted eight bit and the low byte is added to the resulting int16 number. The distance should be saved in an unsigned int because it can't be negative, and the velocity must be saved in a signed int because the target can also move away.

13.10.3 Example program

In this example an own radar-class (see chapter 6) was created with the function receive_radar_data(). The function reads in the distance and velocity data from the radar and saves these in variables created in the main-class.

main-class:

```
uint16_t distance = 0;
uint16_t velocity = 0;
volatile bool error = false;

void setup() {
  Serial1.begin(115200);
}

void loop() {
  static int availables = 0;
  availables=Serial1.available();
  for(int i=0;i<availables;i++)
  {
    uint8_t in =Serial1.read();
    if (in == RADAR_START_BYTE)
    {
      if(!Radar.receive_Radar_Data(&distance, &velocity)) //Receive data
         via UART Pin16/17
      {
      error_radar_transfer=true;
      break;
      }
    }
  }

}
```

radar-class:

```
boolean Radar_Interface::receive_Radar_Data(uint16_t *distance_m, int16_t
    * velocity_ms)
{
  uint8_t incomming_bytes[_NUM_RADAR_BYTES] = {0};

  int received=1 ;

  while(received<_NUM_RADAR_BYTES)
  {
    if (Serial1.available())
    {
      incomming_bytes[received] = Serial1.read();
      received++;
    }
  }

  if (incomming_bytes[_NUM_RADAR_BYTES - 1] == _RADAR_STOP_BYTE)
  {
    distance_ =incomming_bytes[1];
    distance_ =(distance_<<8)|incomming_bytes[2];
    velocity_ = incomming_bytes[3];
    velocity_ = (velocity_<<8)|incomming_bytes[4];

    if((distance_!=65535)&&(velocity_!=32767))
    {
      *distance_m=distance_;
      *velocity_ms=velocity_;
    }
    if((incomming_bytes[1]^incomming_bytes[2]^incomming_bytes[3]^
       incomming_bytes[4])==incomming_bytes[5]){ //check checksum
      return true;
    }else{
      return false;
    }
  }
  return true;
}
```

13.11 ShieldBuggy

The ShieldBuddy is a remote-controlled buggy, which reads in the data from different sensors with the help of Aurix ShieldBuddy. Accordingly, it returns a PWM signal to the engine of the car. The various components of the car are described in this document. The main focus is set on how to put the buggy in operation and how small changes can be accomplished.

13.11.1 ShieldBuddy

With its three cores the ShieldBuddy is controlling the buggy. Because eventually the driving Shield-Buddy is mounted on the ShieldBuddy, most of the connections are made there. Therefore, only the SPI connection must be added. Additionally, the type of supply must be selected by putting the supply jumper on the outer two pins, to use the ShieldBuggy or on the inner two to upload a new program via USB.

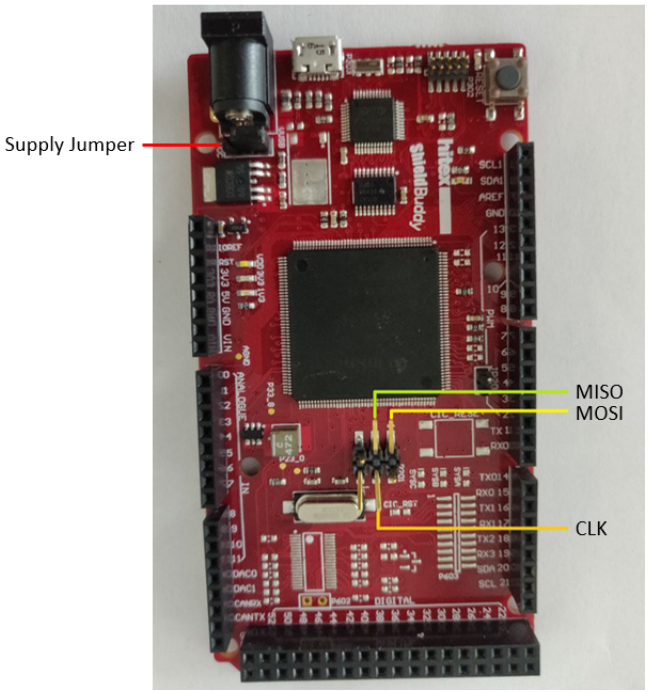


Figure 21: shieldbuddy on buggy

Where the SPI connections can be found on the driving ShieldBuddy, is shown in the next chapter.

13.11.2 Driving ShieldBuddy

The driving ShieldBuddy is mounted on the ShieldBuddy and connections to engine, remote control, lights and radar are made here.

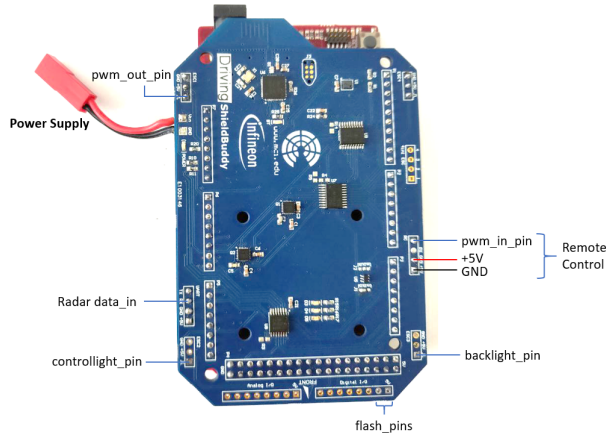


Figure 22: shield on buggy topside

The SPI connection to the ShieldBuddy as mentioned in the first chapter can be found on the bottom side of the driving ShieldBuddy. The last thing worth mentioning at this point is the yellow cable you can see in the next picture which connects the pwm_in_pin to a pin, which is reachable from the outside.

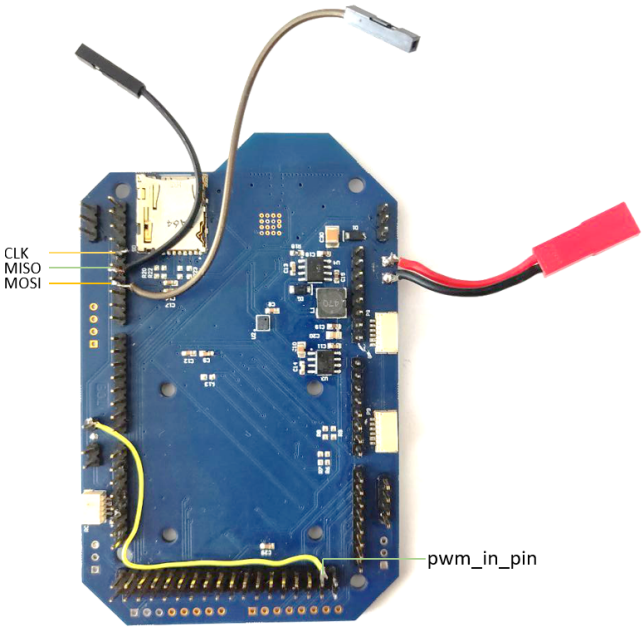


Figure 23: shield on buggy bottom

13.11.3 Remote control/ receiver

The buggy is steered with a remote control over radio waves. The receiver provides the data in three channels and needs to be supplied at least with one 5V and on GND connection. At channel1 the

data for the steering is delivers and must be directly connected with the servo. At channel2 the data for the speed is delivered which must be connected to the pwm_in_pin of the flying ShieldBuddy.

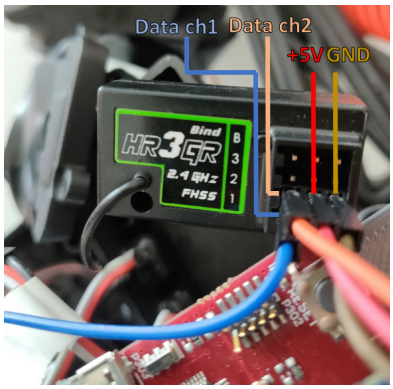


Figure 24: receiver

In the finished version the cables are colour coded and can be seen in the next picture. The speed signal (ESC) is realized with a yellow and green plug and the control signal with a blue plug. The supply is connected to the supply distributor above.

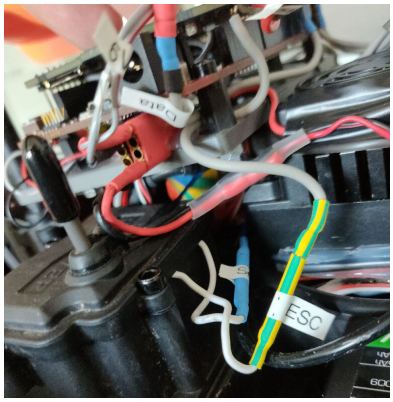


Figure 25: Wiring of the receiver

On the underside of the ShieldBuddy cables are mounted for connecting the velocity signal from the remote control to the controller. The plugs are color-coded again.

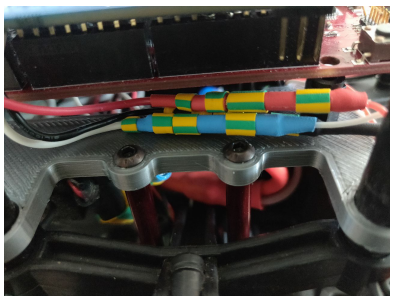


Figure 26: Wiring of the receiver to the ShieldBuddy

13.11.4 Servo

The servomotor is responsible for steering the buggy. It directly gets the data from the receiver channel1. The servo is supplied with 5V from the battery. The conductors are lowering the current spikes which in otherwise would affect the other components of the buggy.

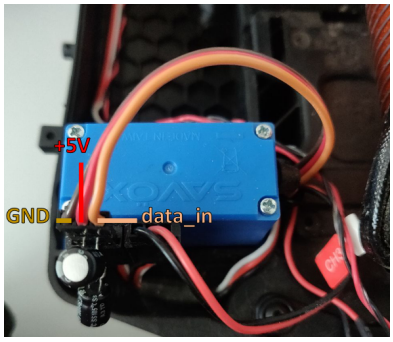
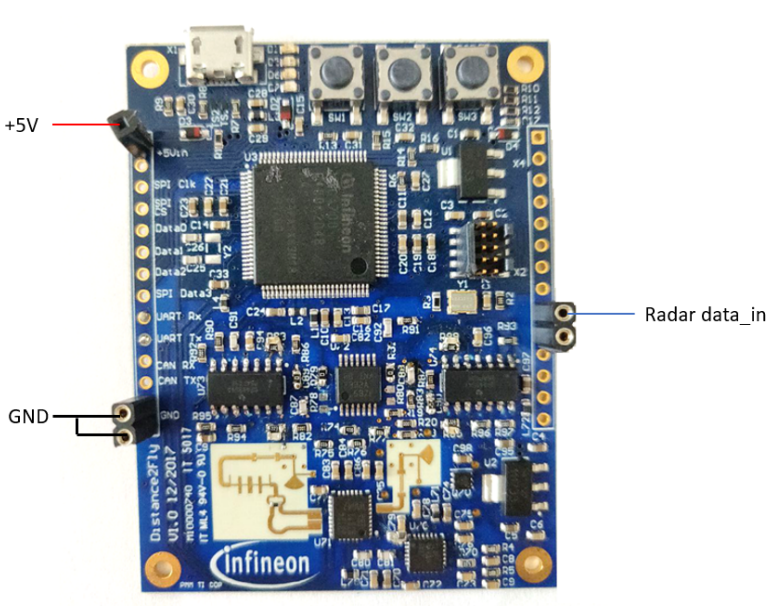


Figure 27: Servo

To connect the servo with the receiver fix cables are mounted in the chassis of the buggy.

13.11.5 Radar

For detection possible obstacled the Distance2Fly Radar was used for this project. The Radar measures the distance and velocity of a detected target and is connected via the Radar data_in pin to the driving ShieldBuddy. It has to be supplied with 5V and is mounted on the roof of the ShieldBuggy. More information how the connection th the radar works and how to program it, can be found in chapter 13.10.



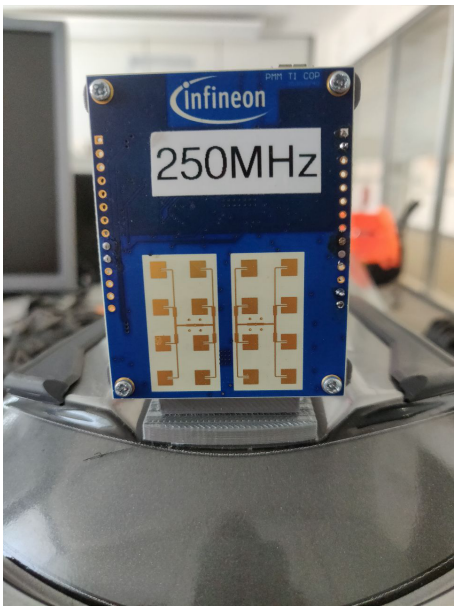


Figure 28: Radar

13.11.6 Velocity

To figure out the velocity of the engine we build a speed sensor because in the used engine there was none implemented. How exactly the sensor was implemented and how the software works can be seen in the Example “engine speed sensor” in chapter 13.8.

13.11.7 Lighting

In the ShieldBuggy it is possible to control two blue flashlights at the front- and backside, a green control light at the top and red brake lights at the back of the buggy. The relatively strong LEDs are operated with drivers (Infineon BCRs), which are placed on the left frontside of the buggy.

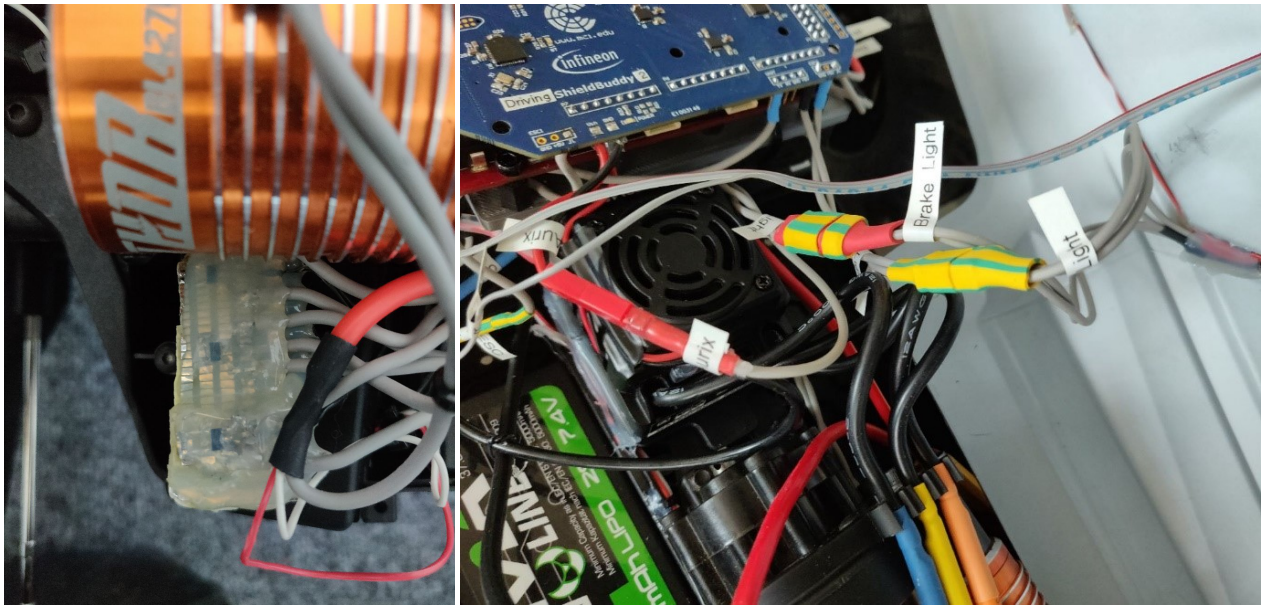


Figure 29: Lighting

The plugs for the lights are color coded and also labeled so only the cabled with the same colors and labels have to be plugged together as seen in the poicture above.

With the current program (chapter 8: Program) the flashlights are always flashing when the buggy in switched on, the control light is switched on when the radar detects a target and the brake light is glowing brighter when the buggy is braking in front of a target.

13.11.8 Wiring and plugs

This chapter gives you a brief overview of the different connectors and what the different color codes and connector combinations mean.

There are two types of plugs which are used in the ShieldBuggy. first the bigger XT30 plugs used for the lights on the cover of the buggy.

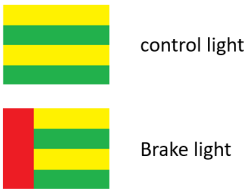


Figure 30: colorcode XT30

Second the small banana plugs used for most of the connections to the driving ShieldBuddy.

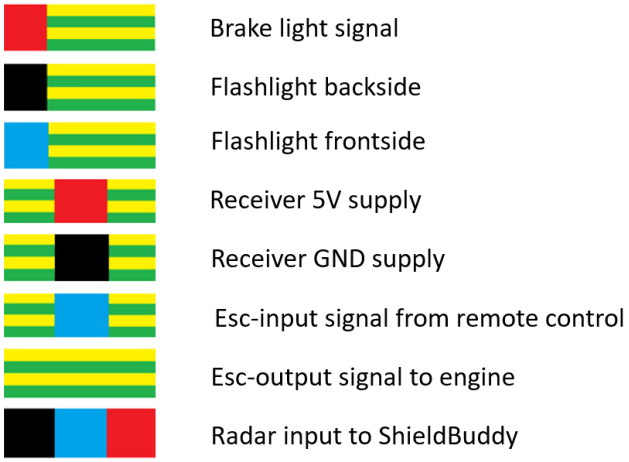


Figure 31: colorcode bananaplugs

13.11.9 Algorithm

The driving algorithm of the ShieldBuggy works as can be seen in the following diagram. As long as no obstacle closer than 1,2m is recognized or the car drives too fast with an obstacle further away the car is in the state DRIVE and the rc_throttle of the remote control is passed through to the engine.

As soon as an obstacle is too close and the buggy is too fast, the braking process starts as it can be seen on the righthand side of the diagram. When the buggy drives very slowly it can get as close as 1,2 meters then it goes directly to the state STAND and can only drive backwards till the distance is big enough again.

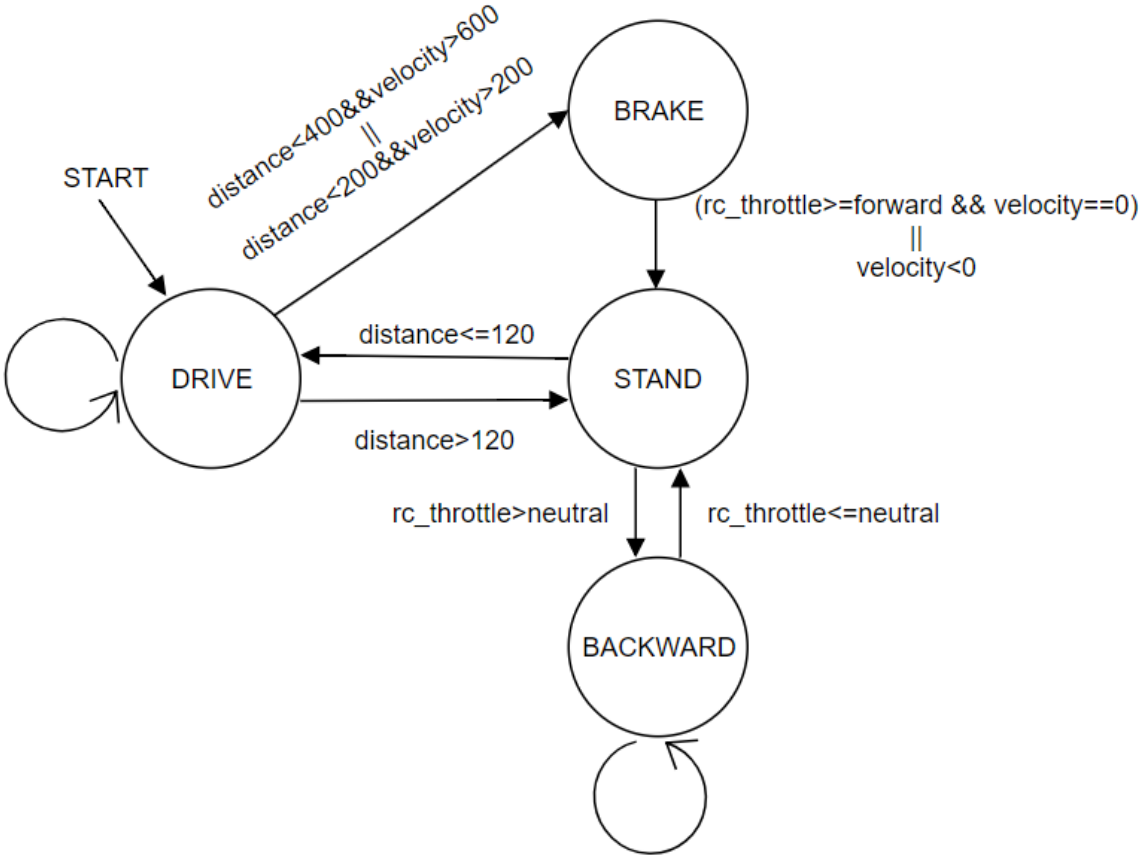


Figure 32: State diagram

13.11.10 Programm

As already mentioned at the beginning the ShieldBuggy is operated with the ShieldBuddy. The three cores share their tasks as followed.

13.11.10.1 Core0

Reads the input from the receiver and outputs a PWM signal according to the data other cores have input.

For this the library rc_input was created, where several scenarios are handled, like what happens when the remote control loses the contact with the buggy or when a too close target is detected.

13.11.10.2 Core1

Controls the lighting of the Buggy.

For that reason, the library lighting was created. In this library it is possible to easily change the rate at which the flashlight is turning on and off or at which intensity the brake and the control light should glow.

Also, for example with the method brake(true) the brake light can be switched on (or with false off) from every core. the methods flash(bool) and safemode(bool) exist for the other lights.

13.11.10.3 Core2

Reads in the IMU and Radar-data and outputs some data over Bluetooth (at the moment the radar data) which can be displayed on the smartphone application ‘Adafruit Bluefruit’.

In this core the distance (in cm) and speed (in cm/s) the car will brake at can be changed.

For the communication with the Radar the library Radar_Interface was developed. To transmit a string via Bluetooth the method n_b.nordic_transmit(StrOut); is used so also other information than the radar data can be sent that way

List of Figures

References

[1] rs online, “Bringing multicore to the arduino world with shieldbuddy tc275.” https://www.rs-online.com/designspark/bringing-multicore-to-the-arduino-world-with-shieldbuddy-tc275.

[2] H. U. Limited, “Shieldbuddy tc275 user manual.” https://docs-emea.rs-online.com/webdocs/159d/0900766b8159d23a.pdf.

Published by
Infineon Technologies AG
85579 Neubiberg, Germany

Order Number: B000-H0000-X-X-7600
Date: MM / 2015