Indice

Capitolo 1

1.1 Phase 1

1.1.1 Hardware

- Quicrun 1060 brushed ESC.
  PWM frequency 50 Hz (200ms)
  It seems it wants 13 ms for neutral position. I tested with 10 as minimum value (maximum speed backward) and 20 as maximum (maximum speed forward).
  Pinout:
  - blue and yellow cables to the motor.
  - black and red connected to the power supply (battery). Be careful in not inverting them (black is the ground). The power supply should generate 7.2V and the current absorbed by the motor can also reach 2.5-3A.
  - The three small cables are connected to the driver. Black one is the ground, red one has to be connected to 5V and the white one to the PWM generator.
- Servo motor to drive the car direction.
  PWM frequency 50 Hz (200ms)
  It should work with 15 ms as central position, 20 and 10 ms for extremals. I tested it and it works but needs calibration yet (TODO).
- Rotative potentiometer. Watching it with pins in front of you:
  - Right pin on 3.3 V
  - Middle pin is the output (to the ADC)
  - Left pin on ground.
- STM32F4-discovery

  - TIM6: it drives the HighResolutionTimer and can be used as a synchronisation point for the execution cycle.
  - TIM4: drives the PWM generator. Now only the channel 1 (pin D12 connected also to green led) is activated.
  - ADC1: reads the speed reference. Now only the channel 8 is activated (pin B0). It acquires data transferring data using the DMA, but now after each conversion i put a sleep when i should poll on a flag asserting that the DMA has finished its work (TODO).
  - UART4: pins A0=Tx and A1=Rx (maybe is better another UART because the user button is connected to A0 and we want to use it (TODO)). It is used as the error stream flushing error messages.
  - USB-OTG-FS: uses the pins A11 and A12. This stream is used to get the RT diagnostic (much faster than UART).

1.1.2 Software

The code has been developed using MARTe framework libraries. A brief description of GAMs and DataSources involved in this first step follows.

- CoolCarControlGAM: reads the timer as synchronisation point and the ADC value from the potentiometer. The parameters that now can be defined are:
  - MaxMotorRef: the maximum value that can be read from ADC.
  - MinMotorRef: the minimum vale that can be read from ADC.
  - MaxMotorIn: the maximum PWM value (depending on the period configured on PWM timer)
  - MinMotorIn: the minimum PWM value.

  This GAM computes the PWM value linearly from the ADC respecting imposed ranges. Then writes on USB interface the value read from potentiometer and the PWM.

- TimerDataSource (input): Is the synchronisation point of the MARTe cycle. This DataSource supports only one signal specifying the sync frequency.
  - Frequency: is the parameter that has to be configured in the signal linked with this DataSource. A thread posts a semaphore with a frequency specified in the signal.

- USBSignalDisplay (output): It prints infos on USB interface.
  - FrameRate: specifies the print frequency.

A thread has been run printing all the values of the signals linked to this DataSource.

- PWM (output): Handles the PWM hw interface.
  - TimNumber: the timer handling the pwm signal.
  - Channel: the timer channel (max 4 for timer on STM32F4-Discovery)
  - ZeroVal: the value to be set at the beginning

  Now each PWM DataSource can handle a single channel (TODO)

- ADC_DMA (input): Handles the read from ADC hw interface.
  - AdcNumber: the number of ADC used.
  - Channel: the ADC channel used (0-15 on STM32F4-Discovery).

  This DataSource manages the read from ADC using DMA to transfer data in memory.

The DataSources (and their brokers) normally don't manage the peripheral configuration. The configuration of all pheripherals has been generated using the STMCubeMX graphic tool. When opening the tool we must specify which type of board we are using. In this case we must specify (STM32F4-STM32F407/417-LQFP100) and then select the one with 1024 Kb flash and 192 Kb ram. The tool is quite intuitive for pheripheral configuration. To use the full speed clock we have to:

- Go to RCC in the Pinout area and select HSE with Crystal/Ceramic
- Go to the Clock Configuration area and set 8MHz in input frequency, select HSE and set M, N, P, Q to 8, 288, 2, 6 respectively in order to use ADC with the maximum speed (the max ADC speed is 72MHz then the clock speed has to be set to 144MHz which is the double).

It is possible configure all the peripherals paying attention to what we are using truth the MARTe DataSources. In general the DataSources manage only the pheripheral operations (read-write) and not the configuration, but if the pheripheral used by the DataSource has not be initialised that can cause unexpectable behaviors.
For this project we have to enable the TIM6 used as HighResolutionTimer, configure the TIM4 channel 1 for PWM output, the ADC1 channel 8 with the DMA2 stream 0, the USB in Device Only mode and Communication Device class and the UART4. We have also to enable FREERTOS. Once we have activated all the pheripherals in the Pinout area we have to switch in Configuration area to configure them as we wish. In FREERTOS configuration remember to enable USE_RECURSIVE_MUTEXES,

USE_RECURSIVE_MUTEXES and USE_COUNTING_SEMAPHORES. It is recommended using heap_3 as Memory Management scheme, so we don't have to care about the total heap size assigned to the operating system since we are using the standard heap memory area.
Once we have fully configured our board hardware go to Project-Settings and set the desired Project Name and Project Location and select Otger Toolchains in the Toolchain/IDE menu. Then going to Project-Generate code, all the configuration code will be generated in the specified folder.
The tool generates also an XML file with all the paths of the generated files. In the Scripts folder there is the python STM32ExtractFiles.py script which using the XML copies outside all the files.

- python STM32ExtractFiles.py [Project_Folder] [Project_Name] [Destination_Folder]

extracts the files from the folder generated by the tool to the destination folder specified. Use ToolConfiguration folder as destination folder remembering to delete all the *.c and *.h previous files but not other files such as the makefiles. The tool generates a main.c function which will be a little bit changed by the makefile. One of these changes is to add an include of a Addons.hdd files if it exists in that directory. Since all the pheripheral handles are declared statically in the main.c, in the Addons.hdd files we can define functions to get the handles and these functions will be used by the DataSources in order to manage pheripherals operations. The current Addons.hdd is:

```
UART_HandleTypeDef *errorUartHandle = &huart4;

ADC_HandleTypeDef * ADC_Handles[] = { &hadc1, NULL, NULL};

TIM_HandleTypeDef * TIM_Handles[] = { NULL, NULL, NULL, &
    htim4, NULL, &htim6, NULL, NULL };

TIM_HandleTypeDef * Get_TIM_Handle(unsigned int number) {
        return TIM_Handles[number];
}

ADC_HandleTypeDef * Get_ADC_Handle(unsigned int number) {
        return ADC_Handles[number];
}
```

where the defined function Get_TIM_Handle and Get_ADC_Handle are called by the DataSources.
The common STM32 DataSources are defined externally in the MARTe2-hw-testing where we are going to develop step by step the code that can be shared across different projects if they use the same type of hardware.
The makefiles manage all the compilation and linking mechanism across these different folders. The idea is to decouple the code that can be shared across different projects and the custom code of the single project we are developing. Acting on makefiles it

is possible decide the code to be compiled and linked. It is enough go in MARTe and call

- make -f Makefile.gcc

to build the project. The executable output.bin will be generated in MARTe/Build and it can be flashed on the board using programs such as st-link.
In this first phase of the project we have deployed the CoolCarControlGAM taking an input signals from the ADC_DMA data source and an input signal from the TimerDataSource for synchronisation. On output we have a signal writing on PWM data source and timer, adc and pwm signals wrote on the USBSignalDisplay data source in order to visualize diagnostic data. Connecting the rotative potentiometer to the pin B0 (ADC1 channel 0), and the ESC DC motor controller on the pin D12 (pwm signal driven by TIM4 channel 0) we can control the car speed.

1.2 Phase 2

In the first phase all the project has been deployed on a single STM32 board. In this phase we are going to divide the work between two different boards: one connected to the potentiometers and the other connected to the car controllers. The two boards can communicate using the radio module nRF24L01+. This module communicates to the board is connected with truth the SPI serial interface and it is possible managing the communication and setting its parameters writing on the chip registers truth SPI. Follows a brief description of the radio module pinout and behavior.

- VCC: to be connected to 5V
- GND: to be connected to ground
- CSN: to be set low during a spi command, high otherwise.
- CE: enables the receive mode if the PRIM_RX bit of the config register it set; triggers a packet transmission if that bit is zero. In the latter case it is enough raise the CE pin for more than 10us.
- SCK: to be connected to a SCK pin of a SPI pheripheral on the STM32.
- MOSI: to be connected to a MOSI pin of a SPI pheripheral on the STM32.
- MISO: to be connected to a MISO pin of a SPI pheripheral on the STM32.

Another change we have done with respect to the phase 1 is the Addons.hdd and the way we use to retrieve the pheripheral handles configured by the STMCube tool. In particular we have declared a struct containing a keyword-name and a void pointer

for the handle, and a NULL terminated array of these structures containing all the pheripherals handles. The function retrieving the handle takes the pheripheral name in input and just searches a match in the array returning the related handle. Follows the code of the current Addons.hdd file:

```
UART_HandleTypeDef *errorUartHandle = &huart4;

struct identifier{
        const char *id;
        void *handle;
};


const struct identifier ids[]={
        {"ADC1", &hadc1},
        {"TIM6", &htim6},
        {"TIM4", &htim4},
        {"UART4", &huart4},
        {"SPI2", &hspi2},
        {"GPIOD", GPIOD},
        {0,0}
};


void *GetHwHandle(const char *id){
        int i=0;
        while(ids[i].id!=NULL){
                if(strcmp(ids[i].id, id)==0){
                        return ids[i].handle;
                }
                i++;
        }
        return 0;
}
```

Note that we have added the SPI2 pheripheral in order to allows the STM32 to be interfaced with the nRF24L01+ radio module. The SPI configuration is quite standard but the follows parameters deserve a special description:

```
  hspi2.Instance = SPI2;
  hspi2.Init.Mode = SPI_MODE_MASTER;
  hspi2.Init.Direction = SPI_DIRECTION_2LINES;
  hspi2.Init.DataSize = SPI_DATASIZE_8BIT;
  hspi2.Init.CLKPolarity = SPI_POLARITY_HIGH;
  hspi2.Init.CLKPhase = SPI_PHASE_1EDGE;
  hspi2.Init.NSS = SPI_NSS_SOFT;
  hspi2.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_256;
  hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
```

the board has to be configured as master (so providing the clock source truth the SCK pin) and we have set the highest baud rate clock divider obtaining the slowest

communication speed. With this configuration the radio modules work, but we can increase the performance increasing the baud rate speed (TODO).
The pinout of the STM32 boards to be connected with the radio modules it is the same:

- B13 - SCK
- B14 - MISO
- B15 - MOSI
- D15 - CE
- D14 - CSN
- 5V - Vcc
- GND - GND

The nRF24L01+ provides another pin (IRQ) that can be used to trigger an interrupt when a packet has been received or transmitted. Now it is unused because we poll on the status register to get these informations.
The software to manage these modules can be found in the folder 'MARTe2 / DataSources / RadioModule' where we have deployed the code to initialise some registers of the radio chip depending on the parameters that the user can set on the configuration file. In this folder we can find the data source itself with all the functions used to drive the modules.
On the STM32 board which acts as transmitter we have enabled another ADC channel to be connected to another potentiometer in order to control the car direction. The direction on the car is performed using a servo motor drive by a PWM signal with 200ms period. We are using now the following:

- ADC1 channel 8 (B0): connected to the potentiometer for car speed control
- ADC1 channel 9 (B1): connected to the potentiometer for car direction control

It is important highlight that the STMCube tool seems not capable to enable different ADC channels, thus this has to be done manually after the code generation.
Passing to the software, the TransmitterGAM reads the two potentiometers from ADCs and converts them to the related PWM values that have to be used to control car speed and direction. These two values are then packed in a 16 bit integer and transmitted using the nRF24L01+ module. On the other side, the STM32 connected to the car runs the CoolCarControlGAM which simply receives the uint16 from its nRF24L01+ module, retrieves two bytes containing the PWM values and sets them to the PWM interfaces connected to DC motor and servo motor respectively.

1.3 Phase 3

In this phase we are going to introduce the raspberry pi 2 and the kinect camera. I am not sure that use the kinect on the car is the best solution, anyway it is important write somewhere a tutorial explaining how to install it on a raspberry because for sure i will use it in other projects.
The only working drivers i have found are on 'https://github.com/xxorde/librekinect', where basically i understood they provide kernel modules for the kinect. So it is necessary compile again all the rpi kernel and this operation can take a while.
The readme on the site is quite clear but the following can be more completed:

```
# connect to rpi via ssh
ssh pi@<ip>

sudo su
apt-get install build-essential bc ncurses-dev tmux git
cd /usr/src/

# find out which kernel you are using - in my case 3.12.20+
uname -r
# get the source - in my case 3.12.y (change if needed)
wget https://github.com/raspberrypi/linux/archive/rpi-3.12.y.tar.gz
tar xfvz rpi-3.12.y.tar.gz
mv linux-rpi-3.12.y linux

ln -s /usr/src/linux /lib/modules/$(uname -r)/build
ln -s /usr/src/linux /lib/modules/$(uname -r)/source
cd /usr/src/linux

make mrproper

# If using Kernel >= 4.x.x execute:
sudo modprobe configs

# get your config
gzip -dc /proc/config.gz > .config

# building, that is going to take a while!
# nohup allows to continue the make if ssh connection drops
nohup make &
# to see make output
tail -f nohup.out -n 10
```

```
make modules_prepare
make modules_install

# copy the new kernel image
cp /usr/src/linux/arch/arm/boot/zImage /boot/linux-3.12.y

# choose it
echo "kernel=linux-3.12.y" >> /boot/config.txt

reboot

# after reboot

git clone https://github.com/xxorde/librekinect.git
cd librekinect

make
# if an errors appears here, you need to comment a line in the code which
# is unnecessary if the kernel version is quite new
make load


# After loading the modules you should have a new "/dev/videoX" which you can
# use like a web cam. For example:

camorama -d /dev/video0
vlc v4l:///dev/video0
```

Once the kinect drivers have been installed, we can download the openni libraries. Follows the procedure to install it on rpi:

```
# dependencies required by OpenCV
sudo apt-get -y install libpng12-0 libpng12-dev libpng++-dev libpng3 \
libpnglite-dev \
zlib1g-dbg zlib1g zlib1g-dev \
pngtools libtiff4-dev libtiff4 libtiffxx0c2 libtiff-tools \
libjpeg8 libjpeg8-dev libjpeg8-dbg libjpeg-progs \
ffmpeg libavcodec-dev libavcodec53 libavformat53 libavformat-dev \
libgstreamer0.10-0-dbg libgstreamer0.10-0  libgstreamer0.10-dev \
libxine1-ffmpeg  libxine-dev libxine1-bin \
libunicap2 libunicap2-dev \
```

```
libv4l-0 libv4l-dev \
libdc1394-22-dev libdc1394-22 libdc1394-utils \
libatlas-base-dev gfortran swig zlib1g-dbg zlib1g zlib1g-dev

# dependencies required by OpenNI
sudo apt-get -y install libusb-1.0 doxygen freeglut3-dev openjdk-6-jdk
graphviz

# python
sudo apt-get -y install libpython2.7 python-dev python2.7-dev
python-numpy python-pip

# download packages
git clone https://github.com/OpenNI/OpenNI.git -b unstable
git clone git://github.com/avin2/SensorKinect.git -b unstable
git clone https://github.com/PrimeSense/Sensor.git -b unstable

# edit OpenNI/Platform/Linux/Build/Common/Platform.Arm and change
CFLAGS += -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
# to
CFLAGS += -march=native  -mfpu=neon-vfpv4 -mfloat-abi=hard

# and do the same for Sensor and SensorKinect directories
cd OpenNI/Platform/Linux/CreateRedist/
sudo ./RedistMaker.Arm

cd ../Redist/OpenNI-Bin-Dev-Linux-Arm-v1.5.8.5
sudo ./install.sh

# do the same for Sensor and SensorKinect directories replacing if needed
# sudo './RedistMaker.Arm' with './RedistMaker Arm'
# with SensorKinect can appear an error due to a pure virtual function not
# defined in XnSensorDepthGenerator class. It is sufficient define it with
# a dummy 'return 0;' inside the code files.

# Once everything has been installed edit
# '/usr/etc/primesense/GlobalDefaultsKinect.ini' and uncomment and set:
UsbInterface=1
```

Once everything has been installed we can run an example in 'OpenNI/ Platform/ Linux/Bin/ Arm-Release'. Most of the samples require graphical interface, thus if we

are connected via ssh to the rpi, it is necessary add the flag '-X' in the ssh command. If the error 'X error because of wrong autentication' appears, do the following:

```
sudo xauth merge /home/pi/.Xauthority
```