Towards an Open Source Linux autopilot for drones

Víctor Mayoral Vilches
Center for MicroBioRobotics
Istituto Italiano di Tecnologia (IIT)
Pontedera, Italy

Siddharth Bharat Purohit
Institute of Technology
Nirma University
Ahmedabad(GUJ), India 380015

Philip Rowse
3D Robotics / ProfiCNC
424 Nicholson Street
Black Hill, Vic 3350
Australia

Alejandro Hernández Cordero
Erle Robotics
Ecuador 3, 1 Izquierda
Vitoria-Gasteiz, Spain

Barbara Mazzolai
Center for MicroBioRobotics
Istituto Italiano di Tecnologia (IIT)
Pontedera, Italy

ABSTRACT

In this paper we present an open source Linux autopilot for drones that implements *state of the art* algorithms and supports three kind of vehicles: *copters*, *rovers* and *planes*. We discuss the historical development of the *AutoPilot Multiplatform* (APM) and introduce the changes that have enabled APM to become a Linux-based autopilot through the creation of the new Linux Hardware Abstraction Layer (HAL).

Section 1 introduces APM's history, code and hardware. Section 2 discusses the work performed to convert APM into a fully functional Linux autopilot. Section 3 exposes the results and section 4 provides an overview of the future work expected.

1. INTRODUCTION

Over the last years, there has been several projects that tried to bring up a fully featured Linux autopilot. Among them the most popular initiative is the Paparazzi[6, 14, 7] autopilot which was one of the main sources of inspiration for the Ardupilot project. To the best of our knowledge, none of these initiatives was widely adopted.

Ardupilot project started officially in 2009 with the contributions of Jordi Muñoz and Chris Anderson [4]. The project was initially linked to a hardware platform holding the same name as the software and it used the Arduino development environment.

As more developers joined the project and new hardware platforms appeared it seemed obvious the need of creating an abstaction layer that allowed different boards to be supported by the code. In August of 2012, Pat Hickey created the *AP_HAL* abstraction layer that simplified adding support for new boards. Using this idea, each board would create its own *Hardware Abstraction Layer* (HAL) inheriting the general requirements specified in the *AP_HAL*.

The project grew far from the Arduino editor thereby it was renamed to *APM* which stands for *AutoPilot Multiplatform*. During the last quarter of 2013, Andrew Tridgell[1] made a prototype of *AP_HAL_Linux*, an abstraction layer that allowed to run APM's code in Linux using the BeagleBone Black as the hardware blueprint. Following this work, the authors decided to launch BeaglePilot project to continue with the effort of making APM available for Linux systems. Currently ardupilot project has about **700.000 lines of code**, more than **150 contributors**, **8.4 commits per day on average** (a commit every three hours) and thousands of users worldwide.

1.1 Hardware

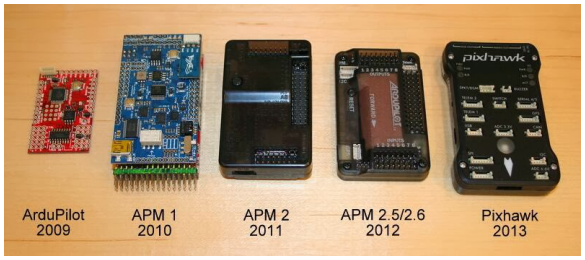

Figure 1: Cronological evolution[4] of the hardware used within the APM project.

[1]Andrew Tridgell is the head developer of ArduPlane.

The first hardware platform appeared in 2009 and shared name with its software side: "ArduPilot". The board included a 16MHz ATMega328 microcontroller (Arduino inspired), a 6-pin GPS connector for the 1Hz EM406 GPS module, six spare analog inputs (with ADC on each) and six spare digital input/outputs to add additional sensors. From this initial concept, the hardware has been updated every year. Figure 1 presents the historical evolution until 2013.

In order to port APM's effort to Linux we used the *PixHawk Fire Cape* (Figure 3) together with the *BeagleBone Black* (Figure 2) development board:

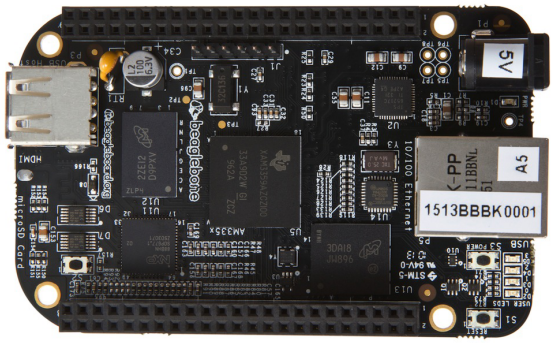

Figure 2: The BeagleBone Black development board.

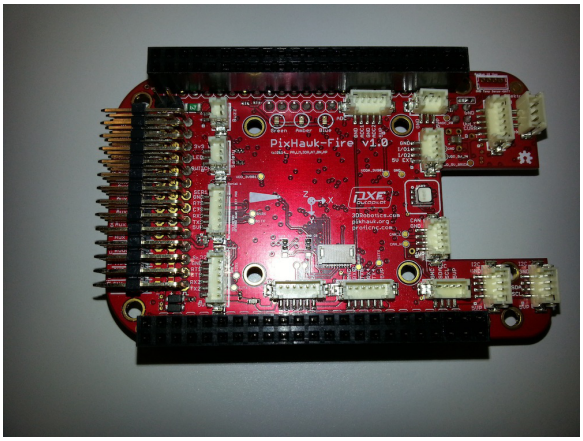

Figure 3: The PixHawk Fire Cape.

The *PixHawk Fire Cape* is a daugther board for the BeagleBone Black that includes the following technologies:

- **MPU6000**: 3-axis gyroscope and 3-axis accelerometer (through SPI bus 1).
- **MPU9250**: 3-axis gyroscope, 3-axis accelerometer and 3-axis magnetometer (through SPI bus 1).
- **LSM9DS0**: 3-axis gyroscope, 3-axis accelerometer and 3-axis magnetometer (through SPI bus 0).
- **MS5611-01BA03**: Barometer (through SPI bus 1).
- **MS5611-01BA03**: Barometer (I2C).
- **BMP250**: Barometer (not placed).

1.2 Code structure

APM code[3] can be divided in four sections: **vehicles's code**, **core libraries**, **sensor libraries** and **other libraries**.

Vehicle's code

```
APMrover2
ArduCopter
ArduPlane
...
```

The code for the 3 different vehicles (rover, copter and plane) are available at the top of the repository and have dedicated folders where the particularities of each vehicle are implemented.

Core libraries

```
libraries/
|--- AP_AHRS
|--- AP_Common libraries
|--- AP_Math vector manipulation
|--- AC_PID
|--- AP_InertialNav
|--- AC_AttitudeControl
|--- AP_WPNav
|--- AP_Motors mixing
|--- RC_Channel
|--- AP_HAL, AP_HAL_AVR, AP_HAL_PX4, AP_HAL_Linux,
...
```

AP_AHRS code estimates the attitude using either a DCM [12] or EKF [10], *AP_Common* includes common utilities for the code while *AP_Math* implements various math functions useful for vector manipulation. *AC_PID* provides a library for PID controllers. *AP_InertialNav* blends together acelerometer data, barometer data and GPS to provide inertial navigation. *AC_AttitudeControl* presents the code in charge of the attitude control. *AP_WPNav* is the waypoint navigation library and *AP_Motors* provides an abstraction for multicopter and traditional helicopter motor dynamics. *RC_Channel* is a library for Radio Control (RC).

The different Hardware Abstraction Layers (HAL) are presented in *AP_HAL*, *AP_HAL_AVR*, *AP_HAL_Linux*, etc.

Sensor libraries

```
libraries/
|--- AP_InertialSensor
|--- AP_RangeFinder
|--- AP_Baro
|--- AP_GPS
|--- AP_Compass
|--- AP_OpticalFlow
...
```

AP_InertialSensor reads gyro & accelerometer data, performs calibration and provides data in standard units (deg/s, m/s) to the main code and other libraries.*AP_RangeFinder* is used for sonar and ir distance sensor interfacing. *AP_Baro*, *AP_GPS* and *AP_Compass* are barometer, gps and triple axis compass interface libraries respectively.*AP_OpticalFlow*

library is utilised to maneouver vehicles indoors using optical flow [9] where there is very low chances of getting gps signal and/or substantial accuracy.

Other libraries

```
|--- AP_Mount, AP_Camera, AP_Relay
|--- AP_Mission
|--- AP_Buffer
...
```

Miscelaneous parts of codebase that doesn't belong to the groups described above are grouped in here. Some examples are the *AP_Mount* library which takes care of the camera mount control or *AP_Mission* library that stores and retrieves mission commands from the EEPROM[2].

1.3 Parameters

APM relies heavily on some configurable parameters[3]. These parameters allow the user to configure the autopilot without the need of recompiling the code. Generally they are stored in the EEPROM but with the new *AP_HAL_Linux* parameters can be stored either in the file system of BeagleBone Black or in the FRAM[4] available in the cape (refer section 2.6).

2. PORTING APM TO LINUX

Many complex Unmanned Aerial Vehicles (UAVs) have usually another processing system on-board that contains an embedded Linux box. This *companion* computer is generally used for onboard complex computing tasks such as image processing or data streaming. Having a separate board makes UAVs larger and usually the wiring complex. We should also not forget that such system also adds as one of the heavy consumer of already very limited Power source.

Besides removing the need of a companion computer, we believe that a Linux port would allow the Linux community to easily jump into the development of applications using drones. BeaglePilot project was launched to port APM's code to Linux-based systems using the BeagleBone and the BeagleBone Black as the "hardware blueprint". The following sections will discuss the work done through the project which contributed mainly to the *AP_HAL_Linux* layer.

2.1 Sensor buses: SPI and I2C

One of the most relevant aspects of an autopilot is the need of predictable timing which is generally needed for sensor sampling. This is traditionally a difficult task to achieve in general purpose Linux systems where the bus access and latencies are not constant.

In order to address this matter, *good* SPI and I2C drivers are a must. These drivers should comply with the latencies shown in Table 1.

Latency	Task
100 ns	SPI bus transitions
1 us	PWM transitions, PPM-SUM input and SBUS
1 ms	IMU sensor input (gyros and accels)
20 ms	Barometer, compass, airspeed, sonar (I2C, SPI and analog).
200 ms	GPS

Table 1: Usual latencies required in an software autopilot

If these latencies are met, the autopilot will be able to fetch the sensor samples and respond in a good manner. Generally, for a capable autopilot system following sensors are necessary:

- 3-axis gyroscope
- 3-axis accelerometer
- 3-axis magnetometer
- barometer
- airspeed

Serial Peripheral Interface (SPI)

The Serial Peripheral Interface or SPI bus is a synchronous serial data link that operates in full duplex mode. Within APM, the bus is used for fast sensor data acquisition. The autopilot uses the SPI bus to fetch information from *gyroscopes*, *accelerometers* and *barometers*. Several *accelerometers* and *gyroscopes* are supported and their code resides under the *libraries/AP_InertialSensor* library. Similarly, the barometers are abstracted under *libraries/AP_Baro*.

The SPI interface for Linux-based systems separates the logic of the bus into two interfaces[5]: *LinuxSPIDeviceManager* which provides an abstraction for APM and *LinuxSPIDeviceDriver* that allows to represent each one of the different sensors. A set of *LinuxSPIDeviceDriver* objects is enlisted under the *static* variable *_device* of the *LinuxSPIDeviceManager* class :

```
LinuxSPIDeviceDriver LinuxSPIDeviceManager::_device
    [LINUX_SPI_DEVICE_NUM_DEVICES] = {

    // different SPI tables per board subtype
    LinuxSPIDeviceDriver(1, AP_HAL::SPIDevice_
        LSM9DS0_AM, SPI_MODE_3, 8, BBB_P9_17,
        10*MHZ,10*MHZ),
    LinuxSPIDeviceDriver(1, AP_HAL::SPIDevice_
```

[2]EEPROM stands for Electrically Erasable Programmable Read-Only Memory and is a type of non-volatile memory used in the hardware autopilots that stores the parameters and waypoints

[3]A full list of parameters is available at `http://copter.ardupilot.com/wiki/arducopter-parameters/`

[4]Ferroelectric RAM (FeRAM, F-RAM or FRAM) is a non-volatile random-access memory similar in construction to DRAM but that uses a ferroelectric layer instead of a dielectric layer to achieve non-volatility.

[5]Refer to */libraries/AP_HAL_Linux/SPIDriver.[h and cpp] for more details*

```
        LSM9DS0_G,  SPI_MODE_3, 8, BBB_P8_9,
        10*MHZ,10*MHZ),
        ...
};
```

Each one of the SPI buses (the BeagleBone Black has 2) is separated through a *semaphore* that avoids two sensor to take the bus simultaneously.

It is worth mentioning that our tests concluded that with the current implementation, the BeagleBone Black needed about 10% of the CPU to run the full set of SPI transactions for all the sensors (about 4000 SPI transactions per second).

Inter Integrated Circuits (I2C or I^2C)

I2C bus has been traditionally used for short distance and low speed communications between a processing unit and its sensors. Compared to SPI which uses $3+n$ lines of communication where n is the number of devices, I2C only uses two[6]. This reduction comes with a protocol that is slower.

Generally, the cabling (external cables) makes I2C less reliable however it is actually *more robust* than SPI. This fact is because SPI has no system for error checking, whereas I2C provides a way to confirm that a device is present. This is the ultimate reason why it is used instead of SPI for external devices over potentially noisy cables[7].

I2C driver for Linux systems is implemented using the *ioctl* system call and currently only supports a single bus per instance. The bus is used to interoperate with several I2C-capable devices. Some examples are EEPROM memory, airspeed sensors, external barometers or external magnetometers.

2.2 UART and MAVLink

The UART driver for the *AP_HAL_Linux* allows to use one of the conventional UART ports[8] for telemetry using the MAVLink protocol[11].

In addition to the traditional serial ports used for telemetry, the Linux port of APM allows to use either a TCP or a UDP socket that simulates the serial port over the network. This functionality was implemented during the development phase and has proved to be useful in many cases since it allows different Ground Control Stations (GCS) to connect to the autopilot using a TCP socket connection provided by Ethernet, Ethernet-over-USB, Wi-Fi, etc. In the TCP case, this functionality is implemented in the function:

_tcp_start_connection(bool wait_for_connection)

that receives a parameter *wait_for_connection* which if *true*, halts the autopilot initialization until it receives an incoming connection[9].

Generally it is not safe to use direct TCP over WiFi for an autopilot since you are likely to lose control (connection) of your vehicle and crash it. We would recommend to use UDP instead. Refer to [3] for the UDP implementation.

2.3 GPIO

The General Purpose Input Output (GPIO) driver takes care of digital I/Os. Some tasks doesn't require much faster responses for example, the status LED control is a pretty straighforward task and it does not have strict timing constraints. On the contrary, the SPI CS control and even for serving the purpose of enable pin GPIO is required to change states with accurate timing. Even a minor overlap between the states of two CS pins, for example, can result into failure in grabbing correct sensor values.
There is a GPIO kernel driver for BeagleBone Black accesible through the *sysfs*[10] interface however, interactions with *sysfs* require read and write system calls which proved to be slow for our interests (approximately 500us). In order to obtain a GPIO driver with a quick response, the GPIO registers of BeagleBone Black are directly accessed using the mmap[11] system call. This technique to change GPIO states provides reponses that can be very accurate compared to sysfs method.

2.4 Output generation

2.4.1 Radio Control (RC) Output using the PRU

RC Output driver for the *AP_HAL_Linux* layer of the APM is divided into two execution spaces: one is the ARM userspace (as a part of Ardupilot executable) and the second one is in the Programmable Real-Time Unit (PRU) space.

The ARM side makes APM to write all the requisites for the PWM output in a block of RAM shared with the PRUs. These requisites are represented through 3 values: *chan_mask*, *period* and *hi_time*. *chan_mask* contains information about the PWM channels that need to be enabled, *period* represents the period in micro-seconds of the PWM signal and *hi_time* represents the micro-seconds for which signal has a digital high value. Although, *period* and *hi_time* are abstracted in micro-seconds by the driver functions, physically, they are written to shared memory in the form of PRU timer ticks.

It should be noted that PRU's clock frequency is 200MHz which is greater than 168MHz in STM32F4 used inside PixHawk and 16MHz in ATMega32U2 used inside APM autopilot board. Furthermore, there are two such systems inside TI's AM335x microprocessor included in BeagleBone Black. With all this processing power some may wonder why not

[6] Note that some SPI devices have additional *data ready* lines per device.

[7] A better alternative than SPI or I2C would be CAN bus which has much better integrity checking (eg. CRCs).

[8] There are 5 UART ports available in the BeagleBone Black however only 4 of them are breaked out. Additionally UART0 (mapped as */dev/ttyO0*) can be used directly from the BeagleBone Black Header P6

[9] There is a work in progress implementation for a non-blocking way of connection available at `https://github.com/erlerobot/ardupilot/tree/uart-tcp-nonblocking-wip`

[10] sysfs is a virtual file system provided by the Linux kernel to interface with kernel drivers

[11] mmap is a system call used to map registers into memory

build most of the autopilot system within the PRU subsystems. The reason is simple: although these processing units have a reasonably fast instruction cycle, they have only 8KB of data RAM (DRAM) and instruction RAM (IRAM) each. With such a small quantity of data and instruction RAM, it is easy to conclude that the PRU subsystems are not meant for full fledged system development but as hard real-time task handlers for applications running inside of the ARM processor (both userspace and kernelspace). PRUs are being utilised to do exactly that: generate 12 PWM signals (on 12 different pins) with independently variable(wrt other channels) *frequency* and *High time* i.e. the time for which the state of PWM pulse is *high*. The resolution of this PWM output is just a couple of microseconds. Such accuracy and resolution is very difficult and complicated to reach while doing the same tasks inside Linux running on BeagleBone Black.

The software running over PRU constantly polls the data inside the shared RAM and puts polled values into a linked list. There is no separate timer peripheral available inside the PRU but each PRU has got *Industrial Ethernet Peripheral* (IEP) with timer as part of this system. IEP timer has a resolution of 5ns and a 32bit register to store the value. This timer is used to serve all the timing related purposes inside PRU space. PRU-ICSS[12] is one of the main reasons why the Beaglebone Black was selected as the *first Linux platform to which APM is ported.*

2.4.2 Tone Alarm

The Tone Alarm library for the *AP_HAL_Linux* allows to create different acoustic tones. It makes use of the *Enhanced High-Resolution Pulse-Width Modulator* (EHRPWM) peripheral inside the AM335x. The driver for using this peripheral is already built inside the kernel and the user just needs to interact with the sysfs interface provided by the driver. The tone format supported by the developed library is *Ring Tone Text Transfer Language* (RTTTL). RTTTL format was formerly used by old cell phone technology. The tones are similar to the ones played by Pixhawk although the format used by pixhawk is that used by PLAY function inside QBASIC.Also, with the autopilot installed inside Linux it gives us the advantage of being able to easily play natural voice audios with higher level formats like *.mp3*. Currently these more advanced compression mechanisms are not supported and is considered as one of the future enhancements.

2.5 Input processing

PPM using PRU

Pulse Position Modulation (PPM) is one of the most common protocols used to transfer actuator setpoint values to vehicle autopilot. Generally, RC receivers use this protocol (either several PPM signals or a single PPM-SUM signal) to send the commands received from the operator to the autopilot. The problem of implementing PPM on Linux machines is similar to the problem in the case of RC Output [13]: the system needs to be hard real-time.
Collecting information from PPM signals requires constant polling of the pins and accurately notifying these changes which is quite difficult to accomplish in the linux userspace. The availability of a second PRU makes it possible to do very minimal but time intensive task of processing the level changes of incoming PPM inputs. It should be noted that the PRU only handles the timing task, no decoding is done by software running on the PRU. Data sharing is done over a *ring buffer* contained in shared memory and the communication is uni-directional: from the PRU to the ARM userspace. Level changes of the PPM signal are pushed into the ring buffer as per their time of occurences with a head pointer incrementing and reseting after an overflow. The data inside the *ring buffer* is read and decoded by *RCInput_PPM* driver with a tail pointer incrementing until it touches the head.

SBus and Spektrum DSMX using PRU

SBus and Spektrum DSMX protocols are proprietary RC protocols from Futaba® and Spektrum® respectively. Nowadays both of these protocols are supported by many third party transceivers. The major motivation of using these technology is that unlike PPM (refer 2.5) they support two way communication and other than solely communicating RC values designated by the user, they can also be transmit telemetry data. The communication on the controller and onboard transceiver happens over serial communication with baud rates of about 100,000 bps for Futaba® SBus and 115,200 bps for Spektrum® DSMX. In Beaglebone Black serial's kernel driver the option of selecting 100,000 bps baudrate is not available. So it was decided to build a serial decoder as a part of ardupilot codebase. PRU's purpose in this case is same as in PPM-SUM method explained in the last section i.e. to collect High time and Low times of the signal and pass it over to userspace via shared RAM. These times are then decoded and stored inside buffer to be available to other parts of ardupilot. Since, the decoder built right inside ardupilot proved to be very efficient, it was decided to do the same with DSMX, even though the baudrate 115200 is supported by Beaglebone Black, so as to save Serial port for other purposes. It should be noted that all three RC inputs(PPM-SUM, SBUS, DSMX) are mapped to a single pin on PXF cape.

2.6 Storage

Parameters as discussed in section 1.3 require a non-volatile storage location and also need to be accessed and changed by the APM system. As we mentioned previously, the storage area of the *Parameters* has been traditionally EEPROM or FRAM but in the Linux port they can also be stored inside the file system and/or FRAM depending on the user preferences.

Storage process is part of the IO thread (refer to section 2.7) which runs at the lowest priority within all threads. The Storage driver serves two main purposes: 1) keep the *load parameter buffer* with the corresponding values inside the FRAM/File-System and 2) update FRAM/File-System storage with the changes demanded by the user. Other than that, parameters other non-volatile storage values including waypoints, fence points and rally points required during autonomous and semiautonomous missions. Although the knowledge of which kind of value being stored is abstracted

[12] Programmable Real-Time Unit and Industrial Communication Subsystem

[13] refer to section 2.4.1

from the storage driver, it needs to keep both buffer and storage updated with coherent values.

2.7 Real-time assessment

In order to address the maximum latencies described in Table 1, the system should be designed with real-time[14] aspects in mind. Most of the time constraints come from getting sensor values. If these are not fetched at the right time it will impact on the Attitude and Heading Reference System (AHRS) calculations and generate a set of wrong control outputs that will cause the vehicles to misbehave.

The following subsections treat three different aspects that address the real-time constraints: kernel latencies, thread priority and the *Programmable Real-Time Units* (PRUs).

Kernel latencies

The Linux kernel has not been designed to meet real-time constraints however there are some techniques that can help tune the kernel to meeting certain deadlines. In this subsection we discuss the experiments performed with the different kernels:

- vanilla kernel (userspace)
- vanilla kernel compiled with the PREEMPT option (userspace)
- RT_PREEMPT patches applied (userspace)
- Xenomai patches applied (userspace)

It is important to note that the tests were performed in the *userspace level*. This is specially relevant for the *Xenomai* case where previous work[2] have proved that this scenario outperforms the rest of the kernels when the applications run in *kernel space*.

In order to generate an intensive computational task the *stress* linux command has been used. This command generates about 100% of computational load which is ideal for testing the different kernels reaction. The tests have been done using *cyclictests*, a benchmarking utility that measures the latency of response to a stimulus. This utility implements the following pseudo-code:

```
clock_gettime((&now))
next = now + par->interval
while (!shutdown) {
  clock_nanosleep((&next))
  clock_gettime((&now))
  diff = calcdiff(now, next)
  # update stat-> min,max,total latency,cycles
  # update the histogram data
  next += interval
}
```

Results are presented in Figure 4 and show that the kernel obtaining the lowest minimum and average latencies is the vanilla (red) one. This result can be understood by the fact that the rest of the kernels include an overhead. This overhead allows them to control the timing in a more precise way however it results in a slightly bigger minimum and average latencies.

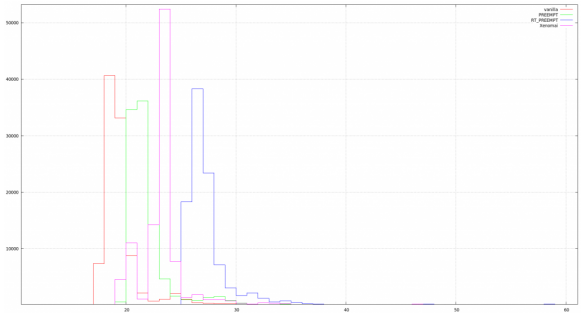

Figure 4: Histogram showing performance tests of different flavours of the Linux kernel v3.8.13: vanilla (red), vanilla compiled with the PREEMPT option (green), PREEMPT_RT patches applied(blue) and a kernel with the Xenomai patches (magenta). The x-axis is measured in micro-seconds (us).

Table 2: Kernel benchmarking results

Kernel type	Min (us)	Avg (us)	Max (us)
vanilla	14	19	193
PREEMPT	16	21	68
RT_PREEMPT	20	27	91
Xenomai	15	23	630

More interesting than looking at the minimum and average latencies is to review the maximum ones presented in Table 2. These values show that the kernel obtaining the lowest maximum latencies is the PREEMPT kernel (green).

Besides the previous results obtained uniquely benchmarking the kernels, we also produced a set of tests using the APM implementation in Linux with the three best performing kernels. Results of this second benchmark test are shown in Table 3. Each kernel is tested under three different loads:

- minimal load[15]
- CPU+I/O+Memory load
- only CPU load

Similarly to the previous case, to overload the Linux OS, the *stress*[16] command was used. It should also be noted that there were other background application consuming resources but were low enough to be ignored.

For each kernel three parameters were registered: %CPU, %MEM and PERF. %CPU and %MEM are self-explanatory, while PERF results can be understood by the following format:

[14] A system's real-time performance is assessed by the time in which it completes the task and the time in which it should have actually completed the task.

[15] By load we mean the additional processes run over OS to consume variety of resources like memory, I/Os and CPU

[16] Refer `http://people.seas.harvard.edu/~apw/stress/`

Table 3: APM_PERF results

Kernel		minimal load			2 CPU-bound processes, 1 I/O-bound process, 1 memory allocator process.			7 CPU-bound processes		
PREEMPT_RT		%CPU	%MEM	PERF	%CPU	%MEM	PERF	%CPU	%MEM	PERF
	General	11.7	7.1	3/4000 4692	17	7.1	9/4000 3075	12.7	7.1	0/4000 2748
	Max observed	12.4	7.1	5/4000 5420	17.5	7.1	20/4000 3139	13	7.1	0/4000 2976
vanilla		%CPU	%MEM	PERF	%CPU	%MEM	PERF	%CPU	%MEM	PERF
	General	10.2	7.1	0/4000 2653	17.4	7.1	93/4000 4573	10.6	7.1	0/4000 2652
	Max observed	10.5	7.1	0/4000 2766	18.7	7.1	119/4000 4386	11.2	7.1	0/4000 2748
PREEMPT		%CPU	%MEM	PERF	%CPU	%MEM	PERF	%CPU	%MEM	PERF
	General	11.2	7	0/4000 2655	17	7	0/4000 2939	11.7	7	0/4000 2666
	Max observed	11.8	7	1/4000 3643	17.3	7	1/4000 3127	12	7	0/4000 2825

$[N_S]/[N_A]$ $[T_M]$

N_S = Number of Slow Main Loops
N_A = Number of Assessed Main Loops
T_M = Maximum time taken out of all assessed Main Loops

These results confirm the previous benchmark results and allow us to claim that: *while using the version 3.8.13 of the Linux kernel in the BeagleBone Black, the PREEMPT kernel works best when it comes to minimum latencies.*

It is interesting to note that the PREEMPT kernel[17] outperforms the PREEMPT_RT one[18] regardless of what previous work afirms [5]:

> The 2.6 Linux kernel has an additional configuration option, CONFIG_PREEMPT, which causes all kernel code outside of spinlock-protected regions and interrupt handlers to be eligible for non-voluntary preemption by higher priority kernel threads. With this option, worst case latency drops to (around) single digit milliseconds, although some device drivers can have interrupt handlers that will introduce latency much worse than that. If a real-time Linux application requires latencies smaller than single-digit milliseconds, use of the CONFIG_PREEMPT_RT patch is highly recommended.

Thread priorities

Having a kernel with a controlled maximum latency is not enough for an autopilot. It could be case that one thread streaming the video takes the processor and makes the autopilot software miss several deadlines. This might easily lead to a possible drift in the attitude of the vehicle and is indeed not desired thereby we need the operating system to be able to priorize the tasks that are most relevant for the autopilot.

Since the Linux kernel schedules threads (the concept of a process is an artificial construct seen mostly by things outside the kernel) we should make sure that the threads running in APM have the right set of priorities. All Linux threads have one of the scheduling policies presented in Table 4.

[17] CONFIG_PREEMPT=y

[18] CONFIG_PREEMPT=y and CONFIG_PREEMPT_RT=y

Table 4: Scheduling policies of the Linux threads

Policy	Description
SCHED_OTHER	Also known as SCHED_NORMAL, is the default policy
SCHED_BATCH	Similar to SCHED_OTHER but with a throughput orientation
SCHED_IDLE	A lower priority than SCHED_OTHER
SCHED_FIFO	A first in/first out realtime policy
SCHED_RR	A round-robin realtime policy

SCHED_OTHER or SCHED_NORMAL are the default scheduling policies for the Linux threads. These threads have a dynamic priority that is changed by the system based on the characteristics of the thread.The threads with a policy SCHED_FIFO will run ahead of SCHED_OTHER tasks[19] in a first-in-first-out series. SCHED_FIFO uses a fixed priority between 1 (lowest) and 99 (highest). The SCHED_RR policy is very similar to the SCHED_FIFO policy. In the SCHED_RR policy, threads of equal priority are scheduled in a round-robin fashion.

APM application threads are configured with with the scheduling policy SCHED_FIFO and with the priorities shown in Table 5. This configuration has proved to be sufficient for stable flights.

Table 5: APM priorities

Name	Priority
APM_LINUX_TIMER_PRIORITY	15
APM_LINUX_UART_PRIORITY	14
APM_LINUX_RCIN_PRIORITY	13
APM_LINUX_MAIN_PRIORITY	12
APM_LINUX_TONEALARM_PRIORITY	11
APM_LINUX_IO_PRIORITY	10

2.8 ROS integration

[19] A SCHED_FIFO thread with a priority of 1 will always be scheduled ahead of any SCHED_OTHER thread.

The Robot Operative System (ROS)[13] is a framework that provides libraries and tools that help software developers create robot applications[20]. As APM transitions to Linux computers, it seems reasonable to provide integration with middlewares such as ROS that would simplify the creation of robotic applications and behaviors.

Besides providing a unifying interface for developing robotics, ROS is based on the the publish-susbsribe paradigm[1]. Currently, while using APM, if one wants to control a gimbal camera using computer vision in Linux we generally set up MAVProxy[21] [15] and write a module for it (for example using the droneapi[8]). This method clearly introduces an unnecessary intermediary: MAVProxy. We believe that removing this intermediary and taking inspiration from the the publish-subscribe paradigm would bring considerable benefits to the autopilot.

With the changes described in section 2.2, APM is now accesible through TCP sockets however the communication with the autopilot is performed using the MAVLink protocol. This means that external programs (such as ROS packages) that wish to speak with the autopilot, should do it using the MAVLink protocol.

We evaluated different available ROS packages[22] that use MAVLink and allow ROS and APM's code to interoperate. The following subsections present this results[23]:

mavlink_ros

The *mavlink_ros* ROS package is a serial-MAVLink-to-ROS bridge. The package creates a node that allows to send and receive MAVLink packets through a serial interface.

The following listing shows a the nodes and topics created by the *mavlink_ros* package:

```
root@erlerobot:~# rosnode list
/mavlink_ros_serial
root@erlerobot:~# rostopic list
/fcu/imu
/fcu/mag
/fcu/raw/imu
/mavlink/from
/mavlink/to
/rosout
/rosout_agg
```

roscopter

roscopter is a ROS package that implements a ROS interface for APM using Mavlink 1.0 interface. It supports controlling the autopilot by overriding the RC input commands, and it publishes all the sensor data. Figure 5 shows the nodes and topics created when working with *roscopter*.

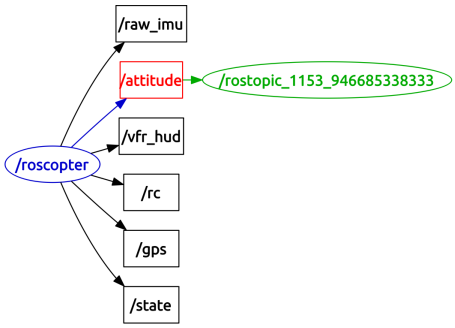

Figure 5: `roscopter` topics and nodes pictured with `rosgraph`

autopilot_bridge

The *autopilot_bridge* ROS package aims to create a bridge for different autopilot protocols. For now it supports only MAVLink. Figure 6 shows a use case.

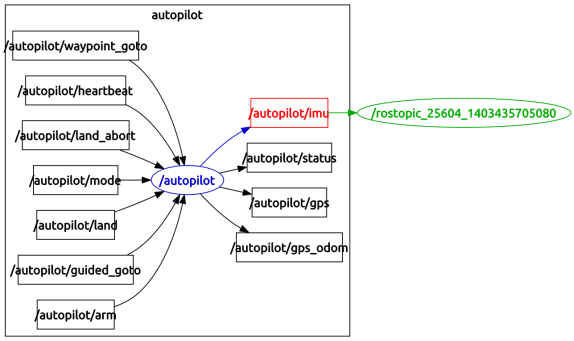

Figure 6: `autopilot_bridge` topics and nodes pictured with `rosgraph`

mavros

mavros is a MAVLink extendable communication node for ROS with a UDP proxy for different Ground Control Stations. This package implemments a MAVLink extendable communication node for ROS that includes the following features:

- Communication with autopilot via serial port, UDP or TCP
- UDP proxy for Ground Control Station
- *mavlink_ros* compatible ROS topics (Mavlink.msg)
- Plugin system for ROS-MAVLink translation
- Parameter manipulation tool
- Mission manipulation tool
- Waypoint manipulation tool
- Safety tool

Figure 7 pictures some of the topics and nodes created when using the package.

[20] For more information about ROS refer to `http://www.ros.org/`

[21] MAVProxy is a fully-functioning GCS for UAV's. The intent is for a minimalist, portable and extendable GCS for any UAV supporting the MAVLink protocol.

[22] Refer to `http://wiki.ros.org/Packages` for a definition and a better understanding of the concept of *ROS package*

[23] For a more detailed analysis refer to `http://erlerobotics.gitbooks.io/erlerobot/en/mavlink/mavlinkros.html`

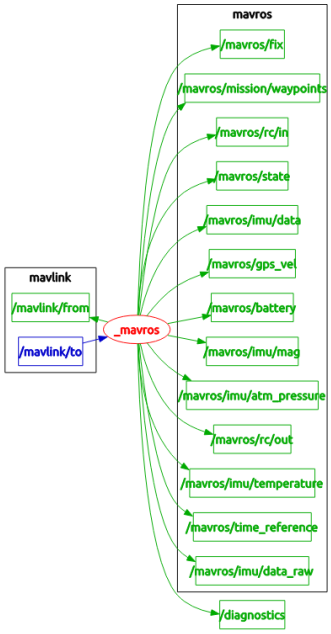

Figure 7: `mavros` **topics and nodes pictured with** `rosgraph`

3. RESULTS

At the time of writing, the three drone platforms supported by APM (copter, plane and rover) have proved to work reliably for simple flights using the Linux port. Still, when compared to the performance obtained by the Pixhawk we perceive that there is room for improvement.

Some have expressed their concerns about whether Linux can meet the real-time requirements while others argue that an embedded microcontroller-based solution will be provide an easier path to certification and at the same time will keep the door open to use the latest available embedded board (as a companion computer).

Although the Linux port needs to mature, we believe that using this approach provides a unifying and well known future for drones and their applications. Our work here has proved that with some tuning, Linux can perfectly be used to meet the real-time requirements needed by an autopilot requiring only about 25% of the processor in the BeagleBone Black. The remaining cycles could be used for other tasks as well as to interoperate with other technologies.

4. FUTURE WORK

Back/front-end interfaces

The the *back/front-end* separation work is primarily motivated by wanting to allow multiple sensors of the same type on a Linux based board (eg. multiple gyros, multiple compasses etc).

ROS integration

For now we are using a serial port and MAVLink-capable ROS packages to interoperate with APM. We also envision the integration of ROS in a deeper level within the autopilot having a separate thread using the ROS API to publish and subscribe directly.

5. ACKNOWLEDGMENTS

This work would not have been possible without the support of many people such as Kevin Hester, Lorenz Meier, Ben Nizette, Jimmy Johnson and Daniel Frenzel. Special thanks to the APM developers and particularly to Andrew Tridgell which has been our main source of support and guidance. We would also like to metion the fantastic support offered by Beagleboard.org developers Jason Kridner, Pantelis Antoniou and their community. The authors would also like to convey thanks to Beagleboard.org, Google, 3DRobotics and Erle Robotics for providing financial means and material.

6. REFERENCES

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 262–272, 1999.

[2] J. H. Brown and B. Martin. How fast is fast enough? choosing between xenomai and linux for real-time applications. In *proc. of the 12th Real-Time Linux Workshop (RTLWS12)*, pages 1–17, 2010.

[3] A. Developers. Apm official repository. `https://github.com/diydrones/ardupilot`, 2014.

[4] A. Developers. History of ardupilot. `http://dev.ardupilot.com/wiki/history-of-ardupilot/`, 2014.

[5] A. Developers. Real-time linux wiki, faq. `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_are_real-time_capabilities_of_the_stock_2.6_linux_kernel.3F`, 2014.

[6] P. Developers. Paparazzi uav. `http://wiki.paparazziuav.org/wiki/Main_Page`, 2014.

[7] G. Hattenberger, M. Bronz, and M. Gorraz. Using the paparazzi uav system for scientific research. In *IMAV 2014: International Micro Air Vehicle Conference and Competition 2014, Delft, The Netherlands, August 12-15, 2014*. Delft University of Technology, 2014.

[8] K. Hester. Drone api. `https://github.com/diydrones/droneapi-python`, 2014.

[9] D. Honegger, L. Meier, P. Tanskanen, and M. Pollefeys. An open source and open hardware embedded metric optical flow cmos camera for indoor and outdoor applications. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1736–1741. IEEE, 2013.

[10] S. Julier and J. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, Mar 2004.

[11] L. Meier. Mavlink: Micro air vehicle communication protocol, 2009.

[12] W. Premerlani and P. Bizard. Direction cosine matrix imu: Theory. *DIY Drones.[Online][Cited: 1 7 2012.]*

http://diydrones. ning. com/profiles/blogs/dcm-imu-theory-first-draft, 2009.

[13] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[14] B. Remes, D. Hensen, F. van Tienen, C. De Wagter, E. van der Horst, and G. de Croon. Paparazzi: how to make a swarm of parrot ar drones fly autonomously based on gps. In *International Micro Air Vehicle Conference and Flight Competition (IMAV2013)*, pages 17–20, 2013.

[15] A. Tridgell. Mavproxy. `http://tridge.github.io/MAVProxy/`, 2014.