RTOS Based Software Architecture for Intelligent Unmanned Systems

Tanmay N. Bichu, Sarthak R. Kaingade, Amit Walambe and Nitin K. Gupta
NavStik Autonomous Systems, Pune, Maharashtra, 411014

With advancement in technology, the research-focus is gradually shifting from development of basic autonomous unmanned systems, towards design of “intelligent” unmanned systems. Along with strides in hardware technology, the algorithms for these machines are rapidly becoming quite complex. In order to manage complexity and build on it, the software engineering practices for unmanned systems need to keep in sync with these changing requirements. This paper presents a summary of work being done at NavStik Autonomous Systems Lab towards this objective. The paper discusses the development of a generic, modular code-base that can be used for deployment of a variety of intelligent systems. This code-base is built on open-source tools and has been made available in open-source for the research community. The availability of this platform would help reduce time from design of algorithms to their deployment on hardware. At the same time, it will reduce entry barrier and encourage further innovation.

I. Introduction

Intelligent Unmanned Systems are a major area of research in recent times. The modern single-board computers provide access to multi-core processors, clocked at several Giga Hertz, in an extremely small and efficient form-factor, making them suitable for use as controllers in intelligent unmanned systems. From the early days of assembly programming for embedded systems, to the use of operating systems for scheduling dozens of tasks in real-time, the software for unmanned systems has evolved to a stage where modularity, abstraction, readability and upgradability take precedence over code-size optimization. Though processors with high processing capabilities are now easily available, the software architecture needs to be designed in a way that it is able to optimally harness the processing power of these processors.

Most traditional systems implement the entire control algorithm, including sensor data-acquisition and processing, as a single thread, which severely limits the scalability of the system. Further, in most cases, the high-level control algorithms directly access the hardware interfaces to the sensors and actuators. This architecture demands a major rewrite of the software whenever there is any change in the hardware configuration (sensors, actuators, or processor) of the system, making it extremely expensive to upgrade the system in future.

The software architecture thus plays a major role in the development of an intelligent unmanned system. The use of a modern real-time operating system addresses the above issues and provides the user with a modular software architecture. A real-time operating system is capable of executing multiple threads in parallel, which enables development of modular applications while still meeting the strict timing constraints of the system. This architecture also provides a lot of other advantages like, scalability, upgradability, usability, multitasking, shorter response time, system management and reliability, which allows the user to focus on application development rather than resource management.

II. Software Architecture Requirements

We now present a summary of requirements from the architecture of software for an Intelligent Unmanned System (IUS). These are common requirements for development of a modern IUS, irrespective of any particular application [1].

A. Real-Time Execution
All the algorithms for system stabilization and control need to be executed in real-time. This further results in real-time performance requirements from sensor data-acquisition and actuator output routines. The software architecture must allow for implementation of these safety-critical routines with required real-time performance guarantees.

B. System Modularity and Scalability
As the applications increase in complexity, it becomes important to be able to program independent modules, with different timing characteristics, that execute in parallel (multitasking) and share the data. The ability to add new modules without any significant overheads in terms of rewriting the exiting modules is highly desirable. Multiprocessor and multi-sensor-extensions must be supported by the operating system in a coherent way.

C. Hardware Upgradability
With advances in technology, newer, better, faster, and cheaper hardware (processors, sensors, actuators) needs to be constantly incorporated in the IUS to keep it relevant with times. A large part of the development effort for any IUS is spent on development of software. The software architecture thus needs to allow easy upgrade of the hardware, as and when it becomes available. In a well-designed software-architecture, the hardware-dependent code must be isolated in the Hardware Abstraction Layer (HAL), making it possible to upgrade the hardware with minimal effort.

D. Code Coherence
In a complex unmanned system with multiple sensors, processors and actuators, the developer should be able to consistently work in the same programming environment from time-critical low-level control to non–time-critical high-level algorithm design.

E. Efficient Code Compilation and Linking
A fast and efficient mechanism to edit, cross-compile, and dynamically link a piece of code with the rest of the framework improves the testing effectiveness of the developer.

F. Inter-System Communication
It order to implement collaborative control strategies and other algorithms on network of IUS, the architecture must provide a set of libraries and standard protocols for exchange of data between different unmanned systems.

G. Multi-Modal User Interfaces
Interfaces of this type make use of multiple communication modalities like speech, motion, gestures or visual-feedback. This requires architecture with enough generalization capabilities, in order to support additional serial lines, display units and analog or digital I/O signals.

H. Ease of Use
The software architecture must exhibit good readability and should be easy to understand. Researchers from different backgrounds should also be able to learn it in a short time. An easy to learn programming language and operating system favors faster integration into courses, exercises, student competitions, and real-life deployments.

I. Man-Machine Interfacing
The architecture must provide a means for on-the-fly visualization of various on-board parameters e.g. system state at a particular instant, its time-history, and parameters for debugging or task supervision purposes.

J. Safety and Reliability
With the ever-increasing presence of computers controlling critical systems—critical to missions, the environment, human lives, or the society—the safety of such systems is a prime concern. The failure of the system needs to be detected in time and corrective action must be initialized for its recovery. The attention to
safety should span all aspects of the software constellation, from high-level behaviors to system-software primitives.

K. Security

As large part of IUS deployments are in public safety organizations, being able to maintain confidentiality is a very important requirement. Also, IUS have a huge potential for misuse when placed in the wrong hands; both from security and privacy point of view. Hence, the software architecture should provide high level of security in terms of access restrictions, ability to track and protection of the data.

The requirements outlined above present a lot of design challenges: abstraction layers with well-defined interfaces, concurrence between tasks sharing different timing characteristics, black box-like components, dynamic behavior, reliability, safety, scalability and economic aspects. Unfortunately, software frameworks do a poor job in supporting general robot-programming patterns. The management of time as a system resource, the safe composition of software modules, the type-safety, the management of dynamic memory, all add up to a great part of the task of writing software for an IUS. There is no real reason not to have the required features embedded in the operating system. In fact, deadline-driven scheduling has already been widely recognized as a safer alternative to interrupts-driven systems [2]. Priority based kernels are more suitable to support dynamic control applications with varied computational requirements [3]. A good embedded operating system should provide common programming constructs like dynamic memory management, inter-process communication (IPC) etc., thus setting the applications free from implementing proprietary solutions which are inherently complex to develop, debug and maintain in the long run. The difficult task of designing and implementing high-level algorithms for an IUS should not be further complicated by these common, low-level (speaking in terms of a layered software architecture) problems.

III. Architecture

The proposed system architecture of an intelligent unmanned system using a real-time operating system is shown in Fig. 1. The architecture can be broadly structured into four different layers. The control algorithms, processing of sensor data, computation of actuator outputs according to the desired state, is done in the application layer. This is also the user space wherein the user can write and execute his applications. These applications run over the RTOS layer. The RTOS layer consists of the actual operating system software. It consists of the file system, kernel, APIs, different stacks like system stack and network stack and high-level
drivers. The Hardware Abstraction Layer (HAL) is a key element in this architecture [4]. It consists of the low-level drivers and all the hardware-specific libraries i.e. the Board Support Packages (BSPs). The lowest layer is the actual microcontroller on which the entire system is running.

One of the key elements of this architecture is the central database management system, which provides a publish-subscribe model for exchange of data between different applications [5]. All the input data, output data and system state data is stored into this database by the applications. For instance, the sensor drivers read the sensor data and write it to the database. The control algorithms read data from the database, carry out all the necessary computations and then determine the actuator outputs necessary, which are written back to the database. The actuator drivers read this data and then actuate accordingly. As a result, data sharing between different applications becomes very easy to maintain. If any new data is to be shared between the applications, the database just needs to be updated to store the new data.

Another important advantage of using this architecture is the scalability. All the low-level drivers and the hardware specific code are present only in the Hardware Abstraction Layer [6]. All the above layers are independent of the hardware. Thus, any change in the hardware will not affect the OS or the applications directly. By changing the HAL, the same applications and the OS can be implemented on any hardware. For instance, if the processor is changed/upgraded, by just including the BSPs for the new processor in the HAL, the entire software stack can be executed on the new processor. If a particular sensor on the board is replaced, which makes use of a different communication bus, as opposed to the previous sensor, all that is required is a change in the high-level driver of the sensor to support the new communication bus. This makes the entire architecture scalable and upgradable to support any hardware platform, which is an extremely useful feature for any IUS implementation. Effectively, a large part of the software developed for a system would remain unchanged, even when the hardware is upgraded or modified in future.

Modularity is another necessary factor for easy code maintenance and upgradability. This software architecture supports vertical as well as horizontal modularity. Vertical modularity means that every layer in the architecture is independent. Changing any layer in the architecture will not affect the working of any other layer, as discussed above. At the same time, the architecture also supports horizontal modularity i.e. all the modules in any single layer are also independent of each other. In the application layer, the interfaces between different applications are standardized and frozen. The algorithms inside the individual applications can now be modified to suite the requirements, without breaking the functionality of the rest of the code. Thus, adding a new control algorithm becomes extremely easy, which enables researchers to experiment with and implement new algorithms, easily.

IV. Implementation of the Architecture

The proposed architecture was implemented as an autopilot system on NavStik\(^1\) (Fig. 2), which is one of the most advanced and smallest autopilot hardware platform available [7]. The entire autopilot system is implemented as an open-source project in NuttX RTOS\(^2\) based on PX4 autopilot firmware\(^3\). The high-level drivers for all these sensors are implemented in NuttX. Using the data from the sensors, the Navigation, Guidance and Control algorithms [3] compute the current system state, the desired state and the outputs required

---

2. https://www.nuttx.org/
to achieve the desired state, respectively. All these parameters are stored in a central publish-subscribe data management unit, ORB (Object Request Broker). Using the MAVLINK protocol, all this data is sent to the Ground Control Station, where the parameters are displayed in a graphical format on the open-source software for ground control station, QGroundControl.

The specifications for the NavStik module, are presented in Table 1.

**Table 1 Specifications of NavStik Module**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>STM32F4 Cortex M4 (32 Bit, 168 MHz) with FPU and DSP</td>
</tr>
<tr>
<td>Sensors</td>
<td>- 3-axis Accelerometer (Invensense MPU60x0)</td>
</tr>
<tr>
<td></td>
<td>- 3-axis Gyroscope (Invensense MPU60x0)</td>
</tr>
<tr>
<td></td>
<td>- 3-axis Magnetometer (Honeywell HMC5883)</td>
</tr>
<tr>
<td></td>
<td>- Static Pressure Sensor (Bosch BMP180)</td>
</tr>
<tr>
<td></td>
<td>- GPS (Ublox Max 6Q)</td>
</tr>
<tr>
<td></td>
<td>- Airspeed Sensor (Measurement Specialties MS4515)</td>
</tr>
<tr>
<td>Communication Interface</td>
<td>Modular Design, Variety of Interface Boards Available</td>
</tr>
<tr>
<td></td>
<td>(UART, I2C, SPI, USB, PWM)</td>
</tr>
<tr>
<td>Computer-on-Module (COM) Support</td>
<td>Plug-in Support for Gumstix/Overo® COM</td>
</tr>
<tr>
<td></td>
<td>Other modules can be connected over UART/USB</td>
</tr>
<tr>
<td>Size</td>
<td>59 mm x 18 mm (2.3 inch x 0.7 inch)</td>
</tr>
<tr>
<td>Weight</td>
<td>~ 4 grams (excluding Differential Pressure Sensor)</td>
</tr>
<tr>
<td></td>
<td>~ 5.4 grams (including all sensors)</td>
</tr>
</tbody>
</table>

---

\(^4\) http://qgroundcontrol.org/mavlink/start  
\(^5\) http://qgroundcontrol.org/start
The application layer consists of all the navigation, guidance and control algorithms. These algorithms are implemented as different modules, which can be enabled or disabled, as necessary. The modules have standardized interfaces, which means that they are independent of what autopilot algorithm is actually running inside the modules. This can be interpreted as horizontal modularity of the architecture. The user can easily develop and incorporate new modules that integrate with the rest of the framework. The ORB module acts as a central data management unit for sharing of data between different modules. The ORB database consists of different topics, which are nothing but structures consisting of similar type of data grouped together. Any module can subscribe to a particular topic to be able to publish (write) and copy (read) the required data from that ORB topic. The HIGHRES_IMU as shown in the QGroundControl screenshot (Fig. 4) is an ORB topic containing all raw data from Inertial Measurement Unit (IMU).

Fig. 4 QGroundControl User Interface

The OS layer consists of the NuttX RTOS. NuttX is a modular, micro-kernel RTOS with POSIX and ANSI standards compliance and a small footprint. It has inbuilt device drivers like USB, SPI, I2C, CAN, serial, ADC and DAC. It also offers support for PWM drivers, which is an important feature for autopilot systems.

The HAL layer has support for different processors scalable from 8-bits to 32-bits. Based on the hardware platform used, modifications are required to be made in two main configuration files, namely `defconfig` and `board.h`. The `defconfig` file contains all the hardware specific definitions, such as the processor used and the interfaces to be configured. The `board.h` file contains the processor specific configurations such as the crystal frequency, timer configurations and pin configurations. Hence, if the processor architecture of the chosen hardware platform is already supported, porting the OS only requires appropriate changes in these two files.

V. Implementing “Intelligence” Layer

The architecture described above provides us with a modular, scalable, upgradable, real-time software-base for implementing lower levels of autonomy, like waypoint navigation, for an IUS. However, for implementing higher levels of “intelligence”, like vision-based navigation, object tracking, multiple-system coordination, the processing power offered by a single processor might not be adequate. While most of these algorithms need high computational power, they are typically, not required to be implemented in a hard real-time framework, as they do not directly talk to critical sensors or actuators. All the critical tasks, required for platform stabilization and control, are implemented in the lower-level autonomy layers, which must run in real-time.

Based on these observations, a framework for implementing the higher-level “intelligence” layer for IUS has been proposed for multi-processor systems, as shown in Fig. 5. Here, the real-time processor handles the lower-
level autonomy functions, such as stability, navigation, and control. This processor interfaces to the critical sensors and actuators and also implements all the fail-safe algorithms. The other processors runs non real-time operating systems and offer heavy processing power, in order to implement vision, higher-level data fusion, and decision-making algorithms. These processors, in turn, command the real-time processor, to control the system as desired. The software architecture must support such an interface between processors to effectively implement the desired strategy. Such a logical separation of tasks further helps in modular implementation of complex intelligent unmanned systems.

Fig. 5 Architecture for Implementation of Higher Levels of Autonomy

As an example, NavStik platform can be used to implement an intelligent aerial vehicle to use visual information for tracking a given moving target, as follows. A single-board computer, like, Gumstix, Beagle-board, Raspberry Pi, etc. can be easily interfaced to NavStik using the UART or USB ports. While NavStik can efficiently handle the real-time tasks and maintain vehicle stability and control, the camera can be interfaced to the single-board computer. With the processing power available on the single-board computer, and the vision libraries, like, OpenCV\(^6\) the image data can be processed and desired position of the IUS (in order to continue tracking the target) can be computed. This position is sent to NavStik as the next waypoint. NavStik makes use of the low-level real-time algorithms to control and guide the vehicle to this desired waypoint. This cycle is repeated to continuously track the desired moving target.

VI. Conclusion

A modular software architecture for an Intelligent Unmanned System has been proposed and implemented. This architecture, based on real-time operating system, has numerous advantages like, scalability, upgradability, real-time performance, code-coherence, readability, and ease of use. The architecture makes use of the publish-subscribe model to provide an effective means of sharing data between applications, with different timing characteristics. Strategies for implementation of “intelligence” layer on multi-processor architectures have also been presented. This architecture has been deployed on NavStik hardware, to implement an autopilot system for an unmanned aerial vehicle. The entire code-base has been released in open-source for other researchers to quickly get started and accelerate their work towards development of advanced intelligent unmanned systems.

References


\(^6\) http://opencv.org/


