PrimeBox PX4 Controller

Technical specification

|  |  |
| --- | --- |
| Publish Version | 1.32 |
| Live link: |  |
| Author | Andrew Taylor-Steen (atsteen@primebox.org) |
| Last edit | Thursday, October 19, 2017 |
| Status | Not Complete |

Contents

[2 Project Design Overview 2](#_Toc496177463)

[2.1 Objectives 2](#_Toc496177464)

[2.2 Project Scope Exclusions 2](#_Toc496177465)

[3 Platform Requirements and Considerations 2](#_Toc496177466)

[3.1 Target Hardware Platform 2](#_Toc496177467)

[3.2 Language Considerations 2](#_Toc496177468)

[3.3 Development and Test Infrastructure Design 3](#_Toc496177469)

[3.3.1 Component Emulation 3](#_Toc496177470)

[3.3.2 IDE Designation 3](#_Toc496177471)

[4 Architecture 3](#_Toc496177472)

[4.1 Program Architecture 3](#_Toc496177473)

[4.1.1 Component Architecture 3](#_Toc496177474)

[4.2 Program Control Flow 9](#_Toc496177475)

[5 Class Models 9](#_Toc496177476)

# Project Design Overview

## Objectives

* + Implement a modular software solution that’s easy to maintain
  + Maintain transparent program logic unless performance requirements force obfuscation
  + Implement libraries that can be compiled to run on both Win32 and AVR (8bit) architectures
  + Include full component emulation in Win32 development environment
  + Achieve 100% functionality of both primary and secondary usage scenarios as defined in the approved functional specification

## Project Scope Exclusions

* + Network communications will not be supported in this release
  + External component support (input or output) will not be included or developed except for the following:
    - DS3231 RTC (i2c)
    - HTU21D Air temperature and humidity sensor (i2c)
    - MicroSD card reader breakout (ICSP)
    - Four channel isolated relay module
    - LCD Display (HD44780 - direct parallel interface)
    - Jog/Select rotary encoder
    - Three button navigation control

# Platform Requirements and Considerations

## Target Hardware Platform

The PX4 codebase will be architected for development in a Win32 environment and deployment to 8bit AVR. Although the codebase will compile and run on All 8bit AVR architectures, this solution targets the ATMega line of microprocessors, and specifically the ATmega2560 package.

Hardware limitations require that compiled binaries must adhere to the following:

* Maximum compiled binary payload size must remain below 240KB
* Maximum runtime memory allocation (static, heap, and Stack) must not exceed 8000 bytes
* Maximum non-volatile memory allocation must remain below 4196 bytes

## Language Considerations

All runtime binaries related to core feature functionally must adhere to ANSI C99 standards. This will ensure compiler support and cross platform compatibility (Development/Deploy). All test plugins and emulators must be compiled and run in Win32 to ensure regression pass compatibility.

## Development and Test Infrastructure Design

Development and functional testing will be completed in a Win32 environment. Functional signoff of completed features must occur against emulated test infrastructure before sanity, regression, and performance/reliability testing is performed on target hardware.

### Component Emulation

All external components must be developed to interface with both physical component representations as well as virtual (emulated) representations. The intentions is to obviate additional hardware changes to enable accelerated parallel testing as well as edge case testing that may be difficult to achieve with standard physical component design.

### IDE Designation

This software solution will be developed in Visual Studio 2015 Community Edition. Target hardware deployment and debugging will be performed using Atmel Studio 7.0. All codebase will be maintained in a GIT repository.

# Architecture

## Program Architecture

The PX4 controller implements a design that encourages low coupling and stack usage optimized for microprocessors with limited memory. Design pattern implementation should reinforce these characteristics while both simplifying codebase as well as adding transparency to program logic.

### Namespace Organization

Interact

Integrate

Process

Data

### Component Architecture

#### Non-Volatile Memory Management

NV memory is managed as a system component instantiated by the IPersistent interface. Like other components, the NvMemoryManager is bridged and may have several concrete abstractions that facilitate committing and restoring object state (from physical memory, configuration files, or other persistent data sources).

***Class associations:***

* IPersistent
* ComponentModule
* ComponentModule\_NvMemoryManager
* ComponentModule\_NvMemoryManager\_Standard
* ComponentModuleImp\_Factory
* ComponentModule\_Imp\_NvMemoryManager\_<Concretelmplementor>

***Pattern implementation***

* ComponentModule\_Imp\_NvMemoryManager is instantiated by a simple factory pattern
* ComponentModule\_NvMemoryManager is an abstract refinement that follows the bridge design pattern

***Component group interaction and dependencies:***

* Data member state (select primitives) of an object can be committed and retrieved from non-volatile memory
* Objects implement IPersistent
  + IPersistent exposes a factory instance of ComponentModule\_NvMemoryManager\_Standard
  + IPersistent requires implementation of RestorePersistentData and UpdatePersistentData
  + RestorePersistentData and UpdatePersistentData invoke ReadPersistentItem or WritePersistentItem
* IPersistent maintains an instance of ComponentModule\_NvMemoryManager\_PX4 refined abstraction
* ComponentModule\_NvMemoryManager\_PX4 bridges implementer ComponentModule\_Imp\_NvMemoryManager\_<Concretelmplementor> which exposes implementations to ReadPersistentItem and WritePersistentItem

#### Data Logging

Data logging is handled by a DataLogger component instantiated by the ComponentModuleGroup. Like other components, the DataLogger is bridged and may have several concrete abstractions that facilitate Logging event information (to local memory or external storage)

***Class associations:***

* DataLogger\_Standard
* ComponentModuleImp\_Factory
* DataLogger\_Imp\_<Concretelmplementor>
* ComponentModuleGroup

***Pattern implementation***

* DataLogger\_Imp\_<Concretelmplementor> is instantiated by a simple factory pattern (ComponentModuleImp\_Factory)
* DataLogger\_Standard is an abstract refinement that follows the bridge design pattern

***Component group interaction and dependencies:***

* ComponentModuleGroup instantiates DataLogger\_Standard
* DataLogger\_Standard instantiates ComponentModuleImp\_Factory\_PX4
* ComponentModuleImp\_Factory\_PX4 returns DataLogger\_Imp\_<Concretelmplementor>
* DataLogger\_Imp\_<Concretelmplementor> logs events called through DataLogger\_Standard using:

#### Component Module Management

Component modules provide the interface by which the system can interact with both physical and virtual objects. Component modules maintain a bridged abstraction which permits dynamic runtime instantiation by simple factories. Concrete component modules are factory initialized and their bridged abstractions are maintained by component module groups.

***Class associations:***

* ComponentModules
* ComponentModuleImp\_Factory
* ComponentModuleGroup

***Pattern implementation***

* ComponentModules maintain abstract refinements that are instantiated using concrete implementations at runtime (bridge pattern). This enables runtime transformations of system component implementation and support for multiple runtime environments from the same codebase.

***Component group interaction and dependencies:***

* Runtime processes invoke ComponentModules through one or more ComponentModuleGroup collections
* ComponentModuleGroups initialize (lazy) and maintain instances of abstract ComponentModules
* ComponentModules are initialized using concrete implementations created by the ComponentModuleImp\_Factory

#### Data Model Aggregation

The controller architecture follows a lightweight MVC pattern. Data is aggregated in the data model and accessed through public exposure and member accessor methods. The data model implements Ipersistent to expose persistent data access methods. The data model also inherits the observer patterns subject class so that other objects can subscribe to modification events made to the data model.

***Class associations:***

* Ipersistent
* IDataModel
* DataModel
* Subject

***Pattern implementation***

* The DataModel serves the model role of a lightweight MVC pattern
* The DataModel inherits the Subject class and implements the publish side of the observer design pattern
* The DataModel maintains a singleton creation pattern to ensure only one data model is exposed on the controller at runtime

***Component group interaction and dependencies:***

* DataModel implements IDataModel
* DataModel implements Ipersistent
* DataModel inherits Subject

#### View-State Management

The lightweight MVC pattern implementation requires viewstate management for all supported display interfaces. Viewstates initialize and process viewstate elements that result in rendered content. Viewstate elements can be shared amongst views and viewstate elements can be nested to provide aggregated processing of elements. Viewstate objects are initialized and maintained by the Viewstate Manager. A Viewstate string generator manages the storage and retrieval textual information processed by viewstate elements.

***Class associations:***

* ViewStateManager
* ViewState
* ViewstateElement
* ComponentModule\_DataLogger\_Standard
* Viewstate\_String\_Generator
* DataModel
* Observer

***Pattern implementation***

* The ViewStateManager manages the view role in a lightweight MVC pattern

***Component group interaction and dependencies:***

* ViewStateManager observes the DataModel and invokes view state updates on notification
* ViewState objects are created by the ViewStateManager and used to invoke the processing of content that’s eventually rendered through the system display interface
* One or more ViewstateElement objects are created by either a parent Viewstate or ViewstateElement to process and prepare display content
* ViewstateElements use the Viewstate\_String\_Generator to retrieve static content which is aggregated into ViewState display content and published by the ViewStateManager
* ViewState and ViewstateElements query the DataModel to perform conditional content formatting

#### Controller Management

The controller manages the main processing loop for the system. The controller is responsible for top-level creation and invocation of core system objects including the command interpreter, task scheduler, task executor, viewstate manager, component modules.

***Class associations:***

* DataModel
* ComponentModuleGroup
* TaskScheduler
* CommandInterpreter
* TaskExecutor
* Viewstate\_Manager

***Component group interaction and dependencies:***

* The Controller instantiates a ComponentModuleGroup to interface with system components
* The Controller invokes an external instance of a TaskScheduler to poll and queue scheduled tasks
* The Controller invokes an external CommandInterpreter to directly schedule new tasks
* The Controller instantiates a TaskExecutor to execute scheduled tasks
* The Controller instantiates a Viewstate\_Manager
* The controller invokes the DataModel to retrieve information relevant to event processing

#### Command Interpretation

The controller system uses a command interpreter to schedule well-formed tasks. The Command interpreter also maintains logic that forces the DataModel to refresh its subscribers when a user input is handled (user expects interface feedback).

***Class associations:***

* TaskScheduler
* TaskItemPool
* DataModel
* ComponentModuleGroup

***Component group interaction and dependencies:***

* The Controller or TaskScheduler invokes task requests on the CommandInterpreter
* The CommandInterpreter verifies that the well-formed task can be added to the TaskItemPool
* The CommandInterpreter invokes TaskItemPool to queue the new task item
* The CommandInterpreter uses the ComponentModuleGroup to interface with system components
* The CommandInterpreter invokes the DataModel to trigger observer notification when a new task requires state refresh

#### Task Handling & Management

All system events begin as tasks that are scheduled and processed by the controller. The number of tasks that can be actively queued is determined by the controller configuration. When a task is executed, it is passed down a chain of task handlers which are given an opportunity to handle that specific task. If a task can’t be immediately handled, it is retried a finite number of times before its abandoned and released from the task pool.

***Class associations:***

* ScheduledTaskDetail
* ScheduledTaskDetailPool
* TimeSignature
* T\_queue
* TaskItemPool
* TaskExecutor
* TaskScheduler
* ITaskHandler
* TaskHandler
* ITaskItem
* TaskItem
* TaskItemPool

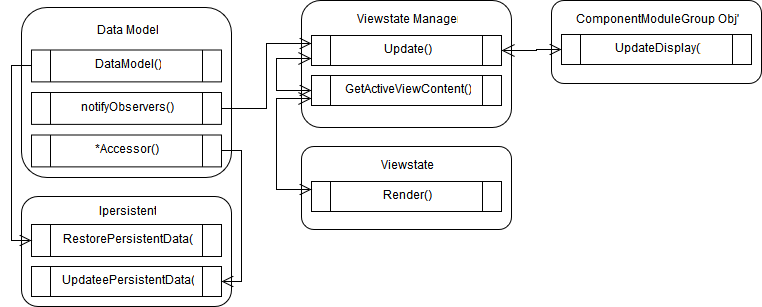
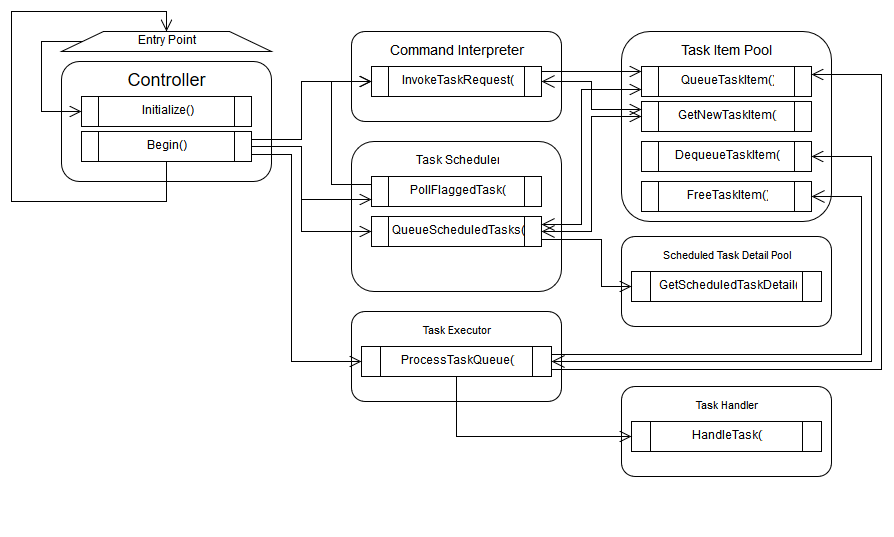
***Pattern implementation***

* TaskHandler object maintain a unidirectional reference chain that implements the Chain of Command design pattern for event handling
* The TaskItemPool is a singleton to ensure only one task item pool exists at runtime

***Component group interaction and dependencies:***

* The main execution loop of the Controller instantiates a TaskScheduler and invokes its member functions to both poll and queue ScheduledTaskDetail items in the ScheduledTaskDetailPool
* A ScheduledTaskDetail maintains the scheduled execution TimeSignature, formal task alias (ID), and active state
* Any ScheduledTaskDetail that’s scheduled to run now is converted into a TaskItem
* ITaskItem declares required virtual method overrides for any TaskItem
* TaskItems can be associated with TaskAlias, ViewstateAlias, and a ScheduledTaskDetail
* TaskItems are cataloged into a TaskItemPool, which maintains a T\_queue of TaskItems
* The main execution loop of the Controller instantiates a TaskExecutor and invokes its member functions to process TaskItems in the TaskItemPool
* The TaskExecutor passes scheduled tasks to a chain of TaskHandler that attempt to handle the task
* The TaskExecutor frees scheduled TaskItems from the TaskItemPool if their handled successfully

## Program Control Flow



# Class Models

* Interfaces
* Allocation Lifecycle
* Persistence
* Dependencies