Lock-by-Wire Door Lock Detailed Design

Steven Lawrance
Version 1.0
November 26, 2006

Document Revisions

Revision	Changes
0.9	Initial draft
1.0	Final version

Table of Contents

Introduction

This document describes the detailed design for the lock-by-wire door lock system using a process of gradual refinement with traceability. The requirements and use cases are captured into an appropriate architecture. The architecture is then refined into an object-oriented detailed design. An overview of the software's requirements appears below.

The software will take inputs from multiple sources to drive the desired states of output devices. The desired states will drive physical mechanisms, and sensors from those physical mechanisms will feed input back into the software as the actual states. The system's current status can be displayed on a notification output along with any alerts that the user should know about. When necessary, the software can activate backup unlocking mechanisms to force the door to unlock in the event of a primary locking mechanism's failure to unlock.

The system generally classifies users into three different categories: trusted users, untrusted users, and installers. Trusted users are typically either authenticated and authorized or simply on the trusted side of the door, such as the inside of a house's front door. Untrusted users are those who are not authorized to affect the door's state when the door is locked. This models the outside of a residential door or the inside of a prison cell door.

The following table summarizes the system's timing requirements in terms of requirements and timeouts, sorted by increasing order of deadline.

Requirement	Deadline
The system shall recognize and prioritize input signals within 10ms	10 milliseconds
The backup unlocking mechanism shall begin to be activated within 50ms of determining that the backup unlocking mechanism should be activated	50 milliseconds
When the fire alarm input signal indicates that the building is on fire, then the door shall begin to disengage its primary locking mechanisms within 50ms	50 milliseconds
The primary locking mechanism shall change its state from locked to unlocked or vice-versa within 3 seconds by default unless if its configuration states different maximum timeout values	3 seconds
When the manual intervention signal is set, it will remain set for five seconds since the time that the signal becomes unset	5 seconds
When a trusted user is trying to open the door and the system determines that the primary locking mechanisms failed, then the system will wait for 15 seconds before it engages the backup unlocking mechanisms	15 seconds
Software updates can occur in five minutes or less	5 minutes

For more information on the background of this project, its requirements, and its quality attributes, please refer to the requirements and specification document for the lock-by-wire door lock system.

Architecture

Context

Collectively, the requirements of this system call for a real-time control system that drives output devices based on inputs, a configuration, and output feedback. Some important decisions are based on time-based information, such as how long a door knob is held down. Other decisions are formed by an analysis of the separate input states and the configuration, which are discussed in requirements R4 and R10. Generally, the system is trying to get the output states to match their appropriate desired states for a given set of inputs, which is documented in requirements R1 and R2. When that cannot be performed after a timeout, the desired output states might change to other values for either alerting or forced unlocking, which is covered in requirement R8. Manual intervention, which is specified in requirement R6, is considered an input device. This general relationship is depicted in Figure 1, which shows the high-level context diagram.

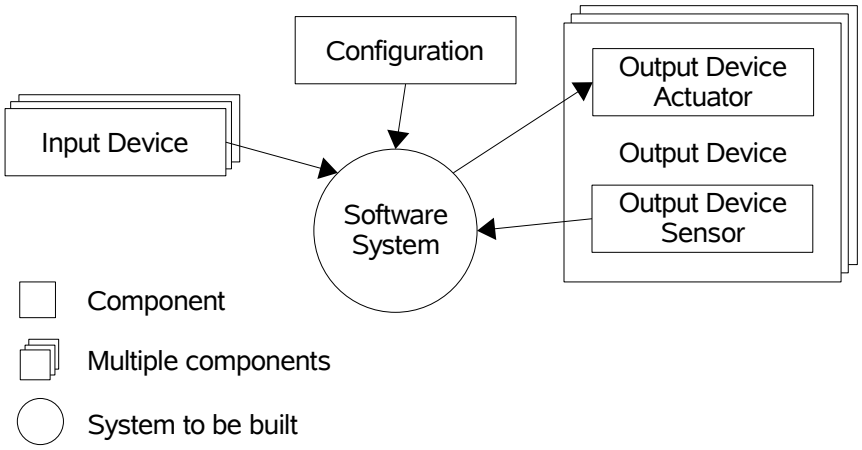

Figure 1: Control System High-Level Context Diagram

Abstract States

The system maintains several abstract states, each which are modeled separately. Figure 2 shows the locked and unlocked states. The system starts with its current state. When a locked door needs to be unlocked, it tells the proper output device to unlock and waits until it either unlocks or a timeout occurs. Depending on other states, the backup unlock mechanism might engage if the primary locking mechanism fails to unlock. Note that the locking operation does not have an emergency locking mechanism in this system.

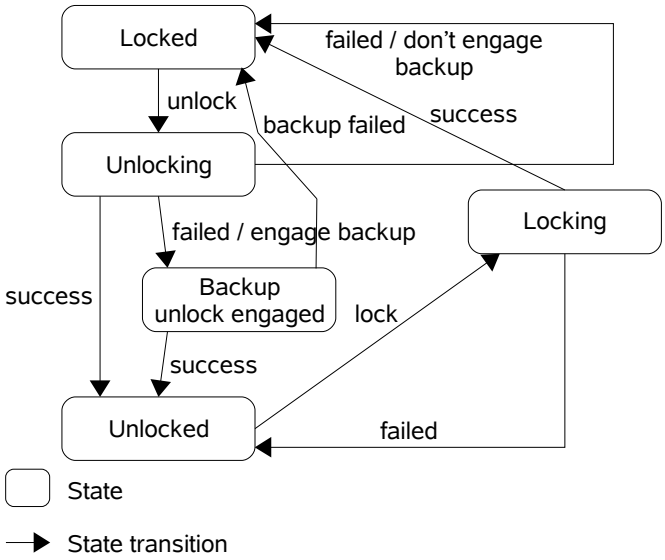

Figure 2: Locked and Unlocked States

Another state tracked by the system is the door's open versus closed state. This is depicted in Figure 3 on the next page. The door is either open or closed, though intermediate state-changing states are included to permit the software system to wait for optional door opening and closing mechanisms to change the door's closed versus opened state. A sliding door with a motor to open and close the door is one example of such as door.

When locked, the door cannot open or close. In swinging doors, however, locking a door that is open should not be possible. That restriction will exist in the configuration so that sliding doors can be locked open while swinging doors cannot.

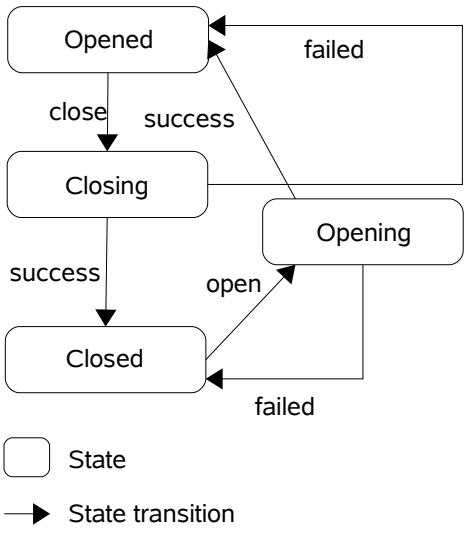

Figure 3: Opened and Closed States

Visibility

To give the user visibility into the software system, which satisfies requirements R7 and R9, both an alerting output and a status display exist. These outputs do not have feedback inputs. Their states simply reflect the states that the system tells them to reflect.

Prioritization

When considering the input device and output feedback states, prioritization needs to take place to consider some states over others, which is documented in requirements R3 and R5. To accomplish this, a set of conditions are checked in a predefined order to determine what the desired lock states should be. These conditions are documented in use case UC5.1.

Cyclic Executive

The software system will run as a cyclic executive that polls inputs for their states, determines the desired output states, and, if needed, changes the output states. When specific events occur, timeout variables will be set so that a different branch can be taken if more than a timeout's time has elapsed. This is depicted in the pseudocode below.

1. Save input states and output feedback states into variables
2. Determine new desired states, including timeout-based states
3. Actuate output devices to make their states eventually become the desired states

Note that the configuration will need to be read into variables before the cyclic executive begins, which is specified in use case UC10.1. The configuration is read in once per system initialization. As a result, when changes occur to the configuration, the system will need to be reset for the new configuration to take effect.

Note that determining the new desired states is split into two different tasks in order to meet the original timing requirements from the software requirements and specification. The four highest-priority decision points are processed separately from the other decision points. The others are processed less frequently, but only if the high-priority decision points didn't signal a final decision by returning true, which is explained later in this section. This permits the low-priority decision points to be preempted by higher-priority tasks in the absence of processor-provided preemptive multitasking. The StatusDisplay decision point is called directly by the cyclic executive only if enough remaining time exists before the next deadline, making this the lowest priority task in the system.

The next section discusses the detail for each component that the cyclic executive uses.

High-Level Design

The design of the architectural components exists in this section.

Saving Input States

Saving the input states into variables is a matter of reading in the values on both the input and the output feedback pins (use case UC2.1) and then storing those values into bit and string variables. In this version, however, implementation of serial strings is not necessary, leaving only bits for inputs. Only those input and output feedback pins that are enabled in the configuration will be read.

Pins that are marked as ignored in the configuration will not be instantiated and thus will be ignored by the decision processors. This detail helps satisfy the configurability quality attribute QA5, which stipulates the configurability of which pins to listen to.

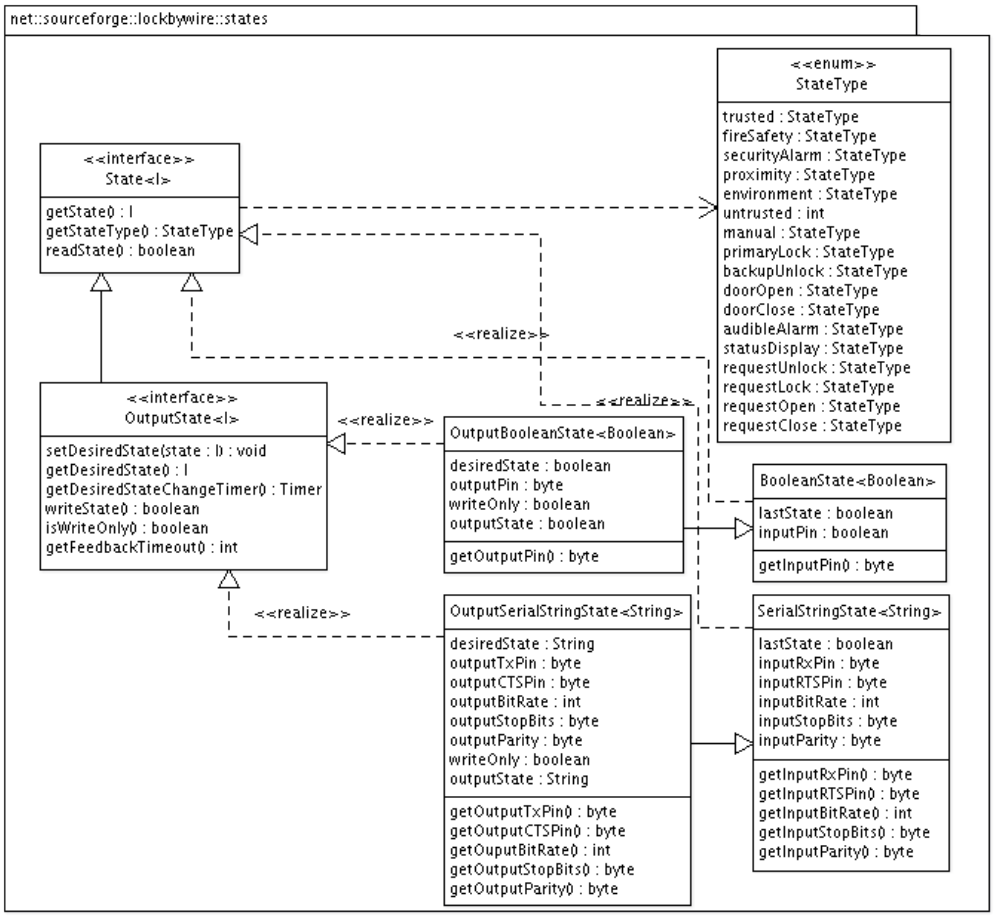

Figure 4: Module view of the states package

Figure 4 above shows how the states are represented architecturally in a module view. Each state has a StateType that corresponds with the type of input or output device that the State object encapsulates. The OutputState interface extends the State interface to include the desired state as well as the getDesiredStateChangeTimer() method that returns a Timer object that was started when the last change to the desired state occurred. Note that setting the desired state of an output to its preexisting desired state will not reset the Timer object. Knowing how much time has

elapsed since the desired state was most recently changed will let the decision point modules know when to time-out their prior operations, which is explained later.

In the current system, all states are boolean except for the status display, which is a text string. Each State object instance maintains an in-memory state of the encapsulated input. Each OutputState also stores its desired state and, when the desired state is set, will reset its desired state modification timer if the new desired state does not equal the old desired state.

Some output states are write-only, meaning that they have no feedback inputs. Such output devices are marked as such in the configuration and will return true when isWriteOnly() is called. Output devices that have a feedback input will return false when isWriteOnly() is called. Note that calling getState() on a write-only output device will return an undefined value.

The State interface includes two ways to obtain a device's state: getState() and readState(). The getState() method will return the most recently read state. If readState() had not been called before the first getState() call, then getState()'s return value is undefined. The readState() method returns true if the state had changed since the last call to readState() or false if it remains the same as it was before. When the cyclic executive decides to save the states of input devices into variables, it will call readState() on all devices, optionally including write-only devices. Calling readState() on a write-only device will always return false.

Devices that interface with the microcontroller using a single boolean value will typically use BooleanState and OutputBooleanState to read and write values, respectively. Because output devices typically have feedback inputs to ascertain the success or failure of an output actuation, OutputBooleanState extends BooleanState to permit easy reading and writing of values from and to the same device.

Devices that interface with the microcontroller using serial communications can use the SerialStringState and OutputSerialStringState. The status display is one such device in the current design, though it's a write-only device, making implementation of the input parts of SerialStringState unnecessary at this time. Parity values for getInputParity() and getOutputParity() are to be defined as constants and can be replaced by an enum class if desired.

This design permits future extensibility by making the state readers and writers generic with clear guidelines as to when to return a cached value and when to query the actual input device for its value, which is the difference between getState() and readState(), respectively. More input and output device types can be added by adding a new StateType enumeration value and then adjusting the decision point modules to consider this new input and adjust its desired state. If a device's state value type is neither binary nor text, then new State and OutputState objects can be created to accommodate the new state value type.

Determining Desired States

To determine the desired states for the output devices (use case UC5.1), several steps need to take place. A series of decision point modules will get executed to determine the desired locked/unlocked and closed/opened states. The order that the modules gets processed in is very important to the system's quality attributes, including safety (QA1), responsiveness (QA2), and security (QA4), especially with respect to timing requirements. Trusted human interactions (use case UC3.1), fire safety, and security alarm inputs have very tight millisecond-level deadlines.

From an architectural perspective, each decision point module implements a common interface that eases future extensibility. Because modules can be enabled and disabled via configuration, the configurability quality attribute QA5 is satisfied. This architecture permits the door software

to operate on a wide range of deployments, including sliding doors, swinging doors, and prison cell doors.

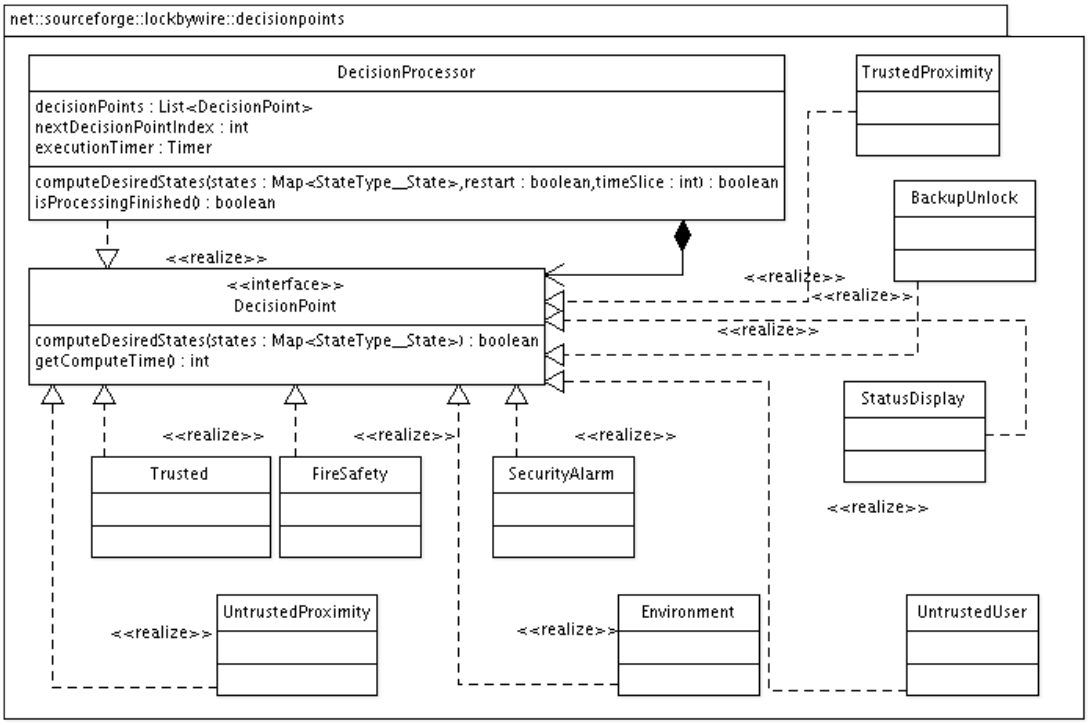

Figure 5: Module view of the decision point package

As shown in Figure 5, every DecisionPoint module implements the computeDesiredStates() method, which returns true to signal a final decision from the decision module or false if the decision processor should query subsequent decision modules. The input and output states are passed in as a Map of StateType objects to State objects.

DecisionPoint objects can use timing information from a state via the return values from getLatestDesiredStateChangeTime() and getFeedbackTimeout(). If more than the timeout has elapsed since the desired state changed, then appropriate abstract state transitions can take place, such as activating the backup unlocking mechanism during a fire emergency.

The cyclic executive calls the three-parameter variant of the computeDesiredStates() method on the DecisionProcessor for both the high-priority and low-priority decision processors. Note that the cyclic executive will need to create these two DecisionProcessors as separate instances to make scheduling easier. The three-parameter variant of the computeDesiredStates() method permits the cyclic executive to pass in time slice information.

In the three-parameter computeDesiredStates() method, the time slice tells the decision processor how many 100µs increments it must execute within, which the cyclic executive calculates from the length of time taken by the higher-priority tasks and the next deadline. If the readState() method returned true on any state to signal an input state change since the last call to the decision processor's computeDesiredStates() method, then the reset argument must be set to true; otherwise, reset should be set to false. The reset parameter will tell the method whether it should start from the beginning of the decisionPoints list when false or if it should resume from where it previously left off when true. The method will execute decision points until either one returns true, the amount of remaining time in the time slice is less than the number of 100µs increments returned by the next decision point's getComputeTime() method, or all decision points have been

executed. The isProcessingFinished() method returns true if the last call to computeDesiredStates() was able to process the last decision point. If no decision points are registered, isProcessingFinished() always returns true. The executionTimer is used to measure how much time has been spent executing decision points, which gets reset on each call to the three-argument computeDesiredStates() method.

Overall, the DecisionProcessor calls the computeDesiredStates() method on the configuration-enabled DecisionPoint modules until either one returns true or all have returned false. If one DecisionPoint module returns true, then computeDesiredStates() in the DecisionProcessor returns true; otherwise, it returns false. With this design, DecisionProcessor acts as an aggregation of DecisionPoints. As a result, the DecisionProcessor's implementation of getComputeTime() will return the amount of worst-case time required to execute all of its decision points, which it computes by adding the return values of calls to getComputeTime() on all of its registered decision point instances. One interesting future possibility created by this architecture that is not used in this current design is that DecisionProcessors can call other DecisionProcessors via the DecisionPoint interface. DecisionProcessor satisfies use case UC5.1.1.

Setting the Output States to the Desired States

After the desired states are updated, which is determined when either computeDesiredStates() returns true or isProcessingFinished() returns true on both the high-priority and the low-priority decision processors, the cyclic executive will call writeState() on all OutputState objects. That method call will actuate an OutputState's output device if the desired state differs from the last-set output state, which is represented by the outputState field in the output state object implementations (use case UC1.1). After writeState() sets the output state, it sets outputState equal to the desired state so that subsequent calls to writeState() won't re-actuate the output device until the desired state changes. The writeState() method returns true if the device's actuated output state was changed or false if it was not changed.

State Compositions

The abstract states discussed earlier are derived concretely from the individual binary and text states. In the current design, these states are the door's open/closed status, the lock's locked/unlocked status, the audible alarm's state, and the status display's text. These abstract states are determined using the following combinations of concrete states and their transitions. Each abstract state can be displayed in the status display to reflect the door's current status.

- Door opening: The doorOpen desired state is true and the doorOpen feedback state is false. The doorOpen feedback state will be true when it the door is sufficiently opened as determined by hardware sensors
    - If either the doorOpen feedback state transitions to true within the configured timeout period or the doorClose feedback state, if configured, is false after the configured timeout period expires, the door's state transitions to opened
    - If the doorOpen feedback state does not transition within the configured timeout period and the doorClose feedback state, if configured, is true, then the door's state transitions back to closed
- Door opened: The doorOpen desired and feedback states are both true
    - The door's state can transition to "closing" when the "closing" condition is met

- Door closing: The doorClose desired state is true and the doorClose feedback state is false
  - If the doorClose feedback state transitions to true within the configured timeout period, then the door's state transitions to closed
  - If the doorClose feedback state does not transition within the configured timeout period, then the door's state transitions back to opened
- Door closed: The doorClose desired state is true and the doorClose feedback state is true
  - The door's state can transition to "opening" when the "opening" condition is met
- Lock unlocking: The primaryLock desired state is false and the primaryLock feedback state is true
  - If the primaryLock feedback state transitions to false within the configured timeout period, then the lock's state transitions to unlocked
  - If the primaryLock feedback state does not transition within the configured timeout period and either the fireSafety input is true or both the manual and trusted input states are true, then the lock's state transitions to "backup unlock engaged." Note that if the manual and trusted input states are true, then the timeout period is extended by 15 seconds after activating audibleAlarm and updating statusDisplay
  - If the primaryLock feedback state does not transition within the configured timeout period and the previous set of conditions did not apply, then transition the lock's state back to locked
- Lock unlocked: The primaryLock desired state is false and the primaryLock feedback state is false
  - The lock's state can transition to "locking" when the "locking" condition is met
- Backup unlock engaged: The backupUnlock desired state is true and the backupUnlock feedback state is false
  - If the backupUnlock feedback state transitions to true within the configured timeout period, then the lock's state transitions to unlocked
  - If the backupUnlock feedback state does not transition within the configured timeout period, then the lock's state transitions back to locked
- Lock locking: The primaryLock desired state is true and the primaryLock feedback state is false
  - If the primaryLock feedback state transitions to true within the configured timeout period, then the lock's state transitions to locked
  - If the primaryLock feedback state does not transition within the configured timeout period, then the lock's state transitions back to unlocked
- Lock locked: The primaryLock desired state is true and the primaryLock feedback state is true
  - The lock's state can transition to "unlocking" when the "unlocking" condition is met

Decision Points

The following table associates decision point modules with their use cases, which can then be used to refine this design into pseudocode. This table is sorted by execution order.

Decision Point	Use Cases
BackupUnlock	UC8.1 and UC8.2
Trusted	UC3.1 and UC5.1.2
FireSafety	UC5.1.3
SecurityAlarm	UC5.1.4
TrustedProximity	UC5.1.5
UntrustedProximity	UC5.1.6 and UC5.1.7
Environment	UC5.1.8
Untrusted	UC5.1.9
StatusDisplay	UC7.1 and UC9.1

Platform Technologies

This section discusses the technologies that will be used to implement this design and includes the processor and language suggestions and minimum requirements.

Processor

The software system requires a hefty number of input pins, a fair number of output pins, periodic checking of input states, and, as a derived requirement from the reliability, safety, and security quality attributes, low power consumption so that it can operate in the absence of electricity. Operation during a power failure will require either a rechargeable battery, capacitors, flywheels, or a combination.

To accommodate and balance the requirements and quality attributes while permitting the use of an object-oriented language such as Java, this design document recommends the Parallax Javelin microcontroller, which is available at http://www.parallax.com/javelin. This processor features 16 pins that can be used for either input or output. Binary, analog, and serial inputs and outputs are permitted. For performance, the Javelin processor is able to execute serial I/O in the background, which substantially reduces the execution time of updating the status display. This processor runs all instructions in a single thread using Java-like language semantics. Garbage collection does not exist, so any objects that are allocated remain allocated until the device is reset, which works well with the architecture's use of primitive method return values. According to the documentation, the Javelin can be reprogrammed up to one million times, which should be more than sufficient for most deployment scenarios as no state needs to be persisted to EEPROM except for the configuration that an installer manually loads as part of use case UC10.2.

This processor costs around $80 per unit in bulk. While less expensive solutions are possible, trade-offs exist in lower-end solutions, and this processor offers attributes that help promote this system's functional requirements and quality attributes, which is explained below.

Although this processor permits only a single thread of execution, it does perform some tasks in the background such as serial I/O. It also has a background timer that supports multiple timer instances, which can help with the cyclic executive for rate monotonic scheduling.

If all the features in the current design are enabled in the configuration, then 22 pins are required:

- One pin for the following inputs: trusted, fireSafety, securityAlarm, proximity, environment, untrusted, manual, requestOpen, requestClose, requestLock, and requestUnlock
- Two pins – one output and one feedback input – for the following outputs: primaryLock, backupUnlock, doorOpen, and doorClose
- One pin for the following output: audibleAlarm
- Two pins for the following output to handle CTS (clear to send) and Tx (transmit): statusDisplay

Because 22 is greater than 16, some multiplexing might be required. Fortunately, the Javelin processor supports analog-to-digital (ADC) conversion that can be used in combination with a multiplexer chip to handle more than one digital input on the same pin. A time penalty exists in the analog-to-digital conversion, so only a small set of analog values that can be counted quickly should be used. It's possible to multiplex more than 2 input devices as this process requires two pins on the Javelin. A reasonable number might be 8, though this should be verified through

experimentation to ensure that the ADC process is not too slow with 8 inputs being multiplexed. An alternative to multiplexing could be to use another similar microcontroller with more input pins. Another possibility is to reuse the main COM port that is used for programming the Javelin for the status display and not using one of the low-priority pins, such as environment, which reduces the I/O pin requirements a little. The doorOpen and doorClose output pins could be digitally multiplexed as they are the inverse of each other, further enforcing their mutual exclusion. Their feedback input pins, however, can continue to be separate or even multiplexed with an analog signal.

The configuration can be implemented in the Javelin as it has sufficient memory that is rewritable, satisfying the constraint of being able to handle different timeouts for different output devices. Configuration reprogramming and software updates can be achieved with a COM port connection to a computer with the Javelin. Due to this ease of configuration and software updating, quality attribute QA6 can be met, which specifies that updates and configurations should be possible within five minutes. The COM port must be accessible in some way without having to deconstruct the door to completely adhere to QA6, preferably from a trusted side of the door. Note that if the status display is also connected to the COM port, then the status display might display random garbage while the Javelin is reprogrammed. A switch that turns off the status display during reprogramming can help avoid that and also remove possible current draws from the status display on the serial lines during reprogramming.

Language

Due to the object-oriented design of this system, using an object-oriented language is advantageous as it can help ensure that an implementation follows this design document, but this is not absolutely necessary. Due to the presence of interfaces, collections, and implementations, using an object-oriented language should be strongly preferred.

On other processors, using C++ or possibly even a full Java is possible, though other attributes and considerations such as cost and complexity need to be looked into.

Input Resistor-Capacitor (RC) Circuits

In addition to the Javelin microcontroller and the supporting hardware, the input pins might need debouncing and/or delay circuits. One popular and inexpensive way to accomplish this is through a resistor-capacitor (RC) circuit on each input pin that either connects to a simple ground-conducting switch or has delay requirements.

For simple ground-conducting switches, the value of the pin read on the microcontroller can be inverted in software so that a closed switch is interpreted as a true value. An open switch would then be interpreted as a false value after the capacitor charges up.

For inputs that have associated hold delays such as the manual input, the RC circuit can discharge when the input goes high, which is possible with either an inverter or an already-inverted input. The RC time constant can then be high enough to make signal build back up to a CMOS high value after five seconds, satisfying use case UC6.1. Similar to the ground-conducting debouncing circuits, the state of the input pin can be inverted easily in software.

Schedulability Analysis

Using the Parallax Javelin as discussed in the previous section, which executes 8000 instructions per second according to the documentation, this section performs a schedulability analysis to ensure that this system will work on this processor and meet its real-time requirements.

The following table lists the period, the estimated number of instructions in a worst-case branching scenario, and execution time of each task.

ID	Task	Period	Instr.	Exec Time
1	Recognize and prioritize input signals	10ms	30	3.75ms
2	Process the high-priority decision points (BackupUnlock, Trusted, FireSafety, and SecurityAlarm)	40ms	60	7.50ms
3	Actuate the output devices to reflect the desired states	50ms	50	5.00ms
4	Process the low-priority decision points (TrustedProximity, UntrustedProximity, Environment, UntrustedUser)	250ms	140	17.50ms
5	Update the status display using the StatusDisplay decision point	500ms	40	5.00ms

The high-priority decision points are split from the low-priority decision points to give preference to them and to increase the likeliness of meeting the original timing requirements. This set of tasks happens to be schedulable as evidenced by the analysis below:

$$U_T = \sum_{n=1}^{5} \frac{C_n}{T_n} = \frac{3.75}{10} + \frac{7.50}{40} + \frac{5.00}{50} + \frac{17.50}{250} + \frac{5.00}{500} = 0.7425$$

$$U(n) = n(\sqrt[n]{2} - 1) = U(5) = 5(\sqrt[5]{2} - 1) = 0.7435$$

$$U_T \leq U(5) \text{ because } 0.7425 \leq 0.7435$$

Because the Parallax Javelin microcontroller does not have a scheduler, a simple rate monotonic scheduler needs to be created in the cyclic executive. Fortunately, the architecture of this system permits the high-priority decision points to be split easily from the others. Using a timer in the Javelin, it's possible for the scheduler to know how much slack time exists for both the high-priority and low-priority decision processors after executing higher-priority tasks. The cyclic executive then tells a decision processor to execute within the calculated execution. This permits the decision processors to spread their executions over several cycles within a single-threaded environment.

The timing diagram in Figure 6 on the next page visually demonstrates the schedulability of the system when all tasks are ready to run at time zero. This diagram assumes that as many branches as possible are taken in each task, making this a worst case analysis. Because it's schedulable in the worst case, the system is schedulable in less-than-worst case scenarios, too.

The decision processor is able to execute decision points based on available time to help the cyclic executive achieve its timing objectives and implement a form of preemption. Inputs are checked frequently and high-priority decision points are processed in case if an important state change needs to occur to meet the original timing requirements. Lower-priority decision points are processed if a higher-priority decision point did not mark a decision as final, which means that all higher-priority decision point modules returned false on computeDesiredState().

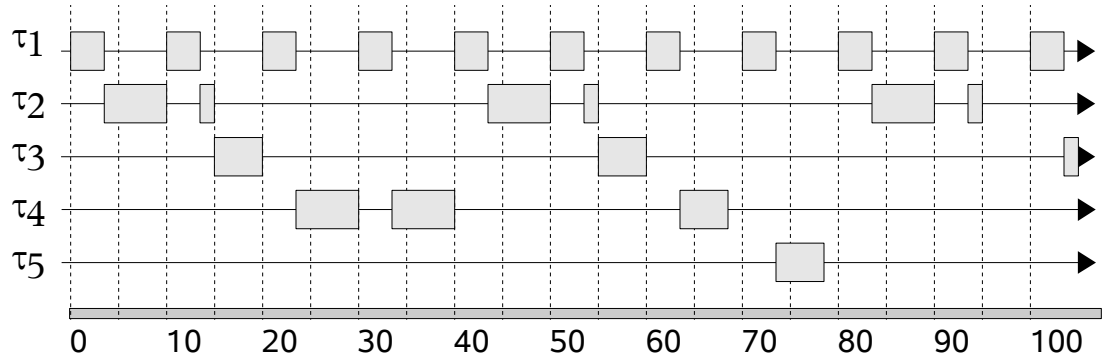

Figure 6: Timing diagram of the schedule

With a schedulable system, the original timing requirements of reading inputs within 10ms, actuating output devices in an emergency within 50ms, and engaging backup unlocking mechanisms in a fire emergency within 50ms hold true. In addition to this, trusted input can easily meet its timing requirements when used with fast-actuating locking mechanisms as it's also considered a high-priority decision point. This design permits emergency actions to preempt low-priority processing such as prioritizing a fire emergency over opening a door to let warm air inside and within a real-time boundary.

References

Javelin Stamp Manual, version 1.0. Parallax, Inc., http://www.parallax.com/dl/docs/prod/javelin/JavelinStampMan1-0.pdf