Modernizing a Security Alarm System

Spinellis, Diomidis

DOI
10.1109/MS.2025.3539364

Publication date
2025

Document Version
Final published version

Published in
IEEE Software

Editor: Diomidis Spinellis
dds@aueb.gr

Modernizing a Security Alarm System

Diomidis Spinellis

IN CONTRAST TO physical objects and living things, software doesn't deteriorate with the passage of time. While we age and our shoes fall apart, digital storage ensures that the software's bits stay immutable. And yet, software needs substantial maintenance over time, owing to changes in its environment.[1] Advancing technology and new requirements prompt us to modernize the software to keep it relevant. Here, I show how these changes happen in practice by describing the evolution and modernization of a burglar alarm security system I first developed a quarter-century ago. All code and its changes are available as open source software at https://github.com/dspinellis/Kerberos.

A burglar alarm system receives its input over diverse sensors, such as reed switches placed on windows or doors, infrared or microwave detectors looking over entire rooms, and photoelectric beams covering passages. A control panel processes these signals and when it detects an intrusion, it can sound sirens or bells and notify the owners or the authorities.

In the fall of 2000, I developed an alarm's control unit as part of an experiment to integrate several home automation functions on a single system.[2] The platform was a secondhand IBM PC with a leisurely clock speed of 100 MHz and 64 MB of RAM running the FreeBSD operating system (OS). For the alarm input and output (I/O), I used an industrial automation peripheral card, for which I wrote a barebones (polling-only) kernel-level device driver. I also built a printed circuit board to interface the card's 5-V I/O signals to the alarm's sensors and actuators via opto-isolators and relays. For performance reasons, I implemented the alarm's control software in the C programming language. To simplify the alarm system's configuration, I developed a domain-specific language (DSL) that models its operation as a state machine and allows the specification states, event transitions, and corresponding actions. For example, a set of transitions specify the following: when a "leave home" command is entered, wait for the main door to open and close, and then enter the "armed" state. Actions can make sensors active, sound sirens, or send out notifications. A small Perl script compiled the DSL into performant C code.

Basing the alarm system on a full-fledged computer and a DSL rather than a microcontroller, as was the case with the commercial units of the time, allowed me to provide several functions that were not typically available in proprietary systems. These included sophisticated rules for distinguishing home members from intruders, automation scenarios for arming and disarming, and several types of intruder notifications. Later, I added further integrations, such as lowering the central heating's thermostat when the occupants are absent.

In 2016, I realized that the system was nearing the end of its life. The OS release I was using was no longer maintained, modern releases could not run on its hardware, and getting spare parts in the event of a hardware failure would be tricky. Consequently, I decided to port the system on a modern platform, namely a Raspberry Pi: a small inexpensive single-board computer. This had the advantage of low power consumption, integrated I/O ports, while still running a powerful OS (Linux). Thanks to the alarm's configuration via a DSL, only one C control file needed substantial changes. In total, from the system's 13 files and 1,039 lines, I only changed four files, removing 115 lines and adding 60 new ones. A small upgrade in 2022 involved the transition to a rack mounted chassis, a more powerful

Digital Object Identifier 10.1109/MS.2025.3539364
Date of current version: 11 April 2025

Raspberry Pi model, and the consequent required change of the employed I/O library.

Iteration Number Four

This year I decided the stars had aligned in a way that would allow me to modernize the system in two important ways. The first was to replace its existing interface that worked by creating and monitoring files on disk with the well-established Representational State Transfer (REST) interface. A REST API (application programming interface) would make it easier to monitor and control the system from diverse clients and applications and to provide a more interactive web interface. The second was to monitor sensors through interrupts rather than polling—a tricky-to-implement feature I had put off from the system's inception. The polling-based method for monitoring some external input involves the software running a loop, continuously checking the input device's state. This is inefficient, because the software continuously performs some action. In my case, over a period of 217 days the alarm control program (daemon) spent about 10 CPU days mostly on polling. The needlessly consumed power reduces the amount of time the alarm can run on the backup power supply. Depending on the time waited between successive checks (needed to reduce CPU load), polling can also involve substantial latency; one second in my case.

A better way to monitor I/O involves hardware interrupts. Under this method, the hardware can be setup to execute some instructions whenever an event occurs (e.g., a sensor trigger). Thus, the software can be idle most of its time, waiting for an interrupt. Typically, interrupts are handled at the layer of the OS, which provides higher-level abstractions via APIs. User programs interface with this facility in two ways. The most common one involves submitting an I/O operation that blocks the program until the OS completes it (blocking I/O). The other involves submitting a nonblocking I/O request that will notify its completion asynchronously—through a polled event or a signal. Higher-level libraries use this interface to support a callback function that will be invoked when the I/O request completes. This so-called asynchronous I/O has been popularized by Node.js, React.js, and other JavaScript frameworks.

Typically, interrupts are handled at the layer of the OS, which provides higher-level abstractions via APIs.

In the case of the alarm system, although both versions of the hardware I used supported interrupts, I never invested the effort needed to implement it. In the case of the original hardware controller, I would need to write a much more complex device driver, while for the Raspberry Pi, I considered it a waste to invest more effort in the alarm's C language implementation, which was already beginning to appear arcane.

The enablers for implementing both features were Python's packages that support interrupt-based I/O and the provision of a RESTful web server interface. Porting the alarm control system from C to Python would allow me to kill two birds with one stone. It would make the system available in a more feature-full and popular ecosystem, while also allowing me to implement the two features I wanted.

A New Design

When replacing a legacy system, the challenge is to retain the knowledge embedded in its code.[3] Fortunately, in my case this was mostly contained in the DSL file. As this had served me well over the years, I decided to retain it, extend it, and base the new design around it.

Starting from the DSL file processing, I continued concurrently building and designing the system in a bottom-up fashion. Every time I saw that the existing design was not serving its purpose I refactored it accordingly. For instance, the event queue was initially part of the state transition module, which was also importing functionality from the port handling module. However, once I started handling input events, it became obvious that port handling would also need to access the event queue, introducing a cyclic dependency. Consequently, I abstracted the event queue into a small separate module.

To avoid the cost and complexity of devising the required data abstractions in C, the original code handled events by compiling the DSL-specified actions and state transitions into

one C switch statement for each state, like the following:

```
static void
proc_ST_wait_for_door_open(void)
{
  syslog(LOG_INFO, "Waiting for door open");
  for (;;) {
   switch(get_event()) {
   case EV_ActiveSensor:
     state = ST_door_open;
     return;
```

web server, which handles transport layer security and access control. The Flask server runs on a separate Python thread, allowing the (almost) concurrent handling of REST requests with other alarm handling tasks. (Python's threads don't offer preemptive multitasking but allow switching between them when waiting for I/O, which is exactly what is needed here.) Route decorators simplified request handling, as can be seen in the following excerpt implementing the /state REST HTTP request:

```
@app.route("/state", methods=["GET"])
def rest_status():
  return jsonify(
    {
      "state": State.get_state().get_name(),
    }
  )
```

> When replacing a legacy system, the challenge is to retain the knowledge embedded in its code.

Python's class support allowed me to abstract the representation of states into objects, each with its actions and transitions tied to it. This clarified the state machine logic by moving it from the obscurity of the DSL compiler into an explicit loop:

```
while cls.state.get_name()! = "DONE":
  if not cls.state.has_direct_transition():
    # Block until an event is available
    event = event_queue.get()
  else:
    # Execute entry actions
    event = None
  new_state_name = cls.state.process_event(event)
  new_state = cls.get_instance_by_name(
      new_state_name)
  if new_state! = cls.state:
    cls.state = new_state
    cls.state.enter()
```

I implemented the REST interface with Python's Flask package. Because the Flask server lacks many features required for production use, I restricted its access to a production-quality

I also handled the blocking reads of sensor raw edge transition events through another thread. This waits for an I/O event to occur, checks whether the sensor is currently associated with an event, e.g., an active or a delayed sensor event, and then queues the corresponding event for further processing. The following code excerpt illustrates this:

```
while True:
  # Blocking read of sensor level changes
  for event in request.read_edge_events():
    port = get_instance_by_bcm(
      event.line_offset)
    event_name = port.get_event_name()
    if event_name:
      event_queue.put(event_name)
```

While coding I looked for opportunities to exploit Python's features. One major change involved the addition to the DSL file of the hardware configuration, which was previously hard-coded in C. This was enabled by defining a base class for I/O ports and subclasses for sensor and actuator objects. Another improvement offered a facility for incorporating and executing Python code in the DSL file. This could be easily run in the correct context (that of the state transition engine module) by means of Python's eval() function supplied with the module's dictionary. These changes decoupled the dependency of the legacy state transition engine on the hardware interface and other APIs.

Once the implementation stabilized, I automated code formatting with the Black tool and introduced static analysis with Pylint. Although early in my career I disliked automatic code formatters, preferring precise manual formatting, I've come to value not wasting mental energy on it. I enforced both with a locally run precommit hook and with continuous integration checks via GitHub Actions. I didn't put these in place from the start, because while I find that automated code checks act as valuable safety railings when a project is relatively stable, at its beginning their demands can be distracting, impeding prototyping, rapid progress, and experimentation.

Obligatory GenAI Section

In common with almost everything I do nowadays, I used extensively generative AI (ChatGPT 4o) as an assistant. Through more than 150 prompts, I got advice regarding Python packages to use, coding patterns, APIs, execution errors, suggestions

for the dynamic injection of imports, Pylint fixes, and even a couple of complete code segments. You can find the complete interaction log shared online at https://chatgpt.com/share/679cd35b-8088-8011-ba5d-18db9f03b8dc. The most extensive help I received concerned the translation from Perl into Python code of the script that converted the alarm specification DSL into C event-handling code. The code required several additional prompts and manual adjustments to correct and perfect it, but ChatGPT saved me considerable time and mind-numbing work.

Another helpful AI-derived code chunk concerned the unit tests for the I/O port handling code. I wrote most of the unit tests concurrently with the corresponding code. Naively, I didn't write unit tests for the I/O port handling routines, considering them trivial. After finding (what I thought to be) a fault in the port handling code, I decided to write complete unit tests to provide an additional test for its fix. I handed the task to ChatGPT, which after a couple of clarification prompts regarding the name of the test package to use and the tested module's name, gave me a 116-line file with ten unit tests. Most were correct and run without a hitch. For two tests it did not consider some global state that was required for the test, and for another one it got wrong the names of the mocked functions. Frustratingly, when I tried to see if the tests failed when removing the fix, I found that they didn't. It turned out that the fault was associated with incorrect unit test initializations. Still, the unit tests uncovered another fault, so they proved useful. In programming the story is often more complex than what appears at first sight.

The responses I got from ChatGPT were not always correct. In several cases it hallucinated method names and constants. However, these were easy to recognize through unit tests and fix based on the online documentation, which continues to be as valuable as ever.

The biggest failure of generative AI was the recommendation of a deprecated package (*RPi.GPIO*). I only realized this quite late, when I deployed the system on the actual hardware and the interrupt-triggered callback routine failed to execute. By searching the web for similar issues, I found out that this package had not been updated to support modern hardware. To rectify the misleading ChatGPT directions required an extensive redesign of I/O handling. A follow-up question to compare *RPi.GPIO* with the modern and portable *gpiod* package which I adopted gave me better advice. In retrospect, my mistake was that the prompt for the initial recommendation was a closed question to compare the officially supported *gpiozero* package against *RPi.GPIO*. Instead, I should have prompted ChatGPT with an open-ended question for available alternatives and decide among them based on my judgment.

In all, rather than replacing me as a programmer, ChatGPT reduced tedious work and allowed me to work more productively by focusing on the substantial stuff. Its mistakes, which I have seen rising when moving from teaching examples and open source software into proprietary code, also highlighted the constant need for a human expert to guide and oversee its operation.

Implementing and modernizing the burglar alarm security system taught me three important lessons. First, the advantage of having a strong architectural core—in this case, the configuration DSL. Second, the improvements readily achieved by upgrading a system to modern technologies—here, Python as a catalyst for dynamic hardware configuration, a REST interface, and hardware-driven sensing. Third, the numerous rewards and challenges of pairing with a generative AI sidekick.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, 104 34 Athens, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

References

1. M. M. Lehman, "Laws of software evolution revisited," in *Proc. Eur. Workshop Softw. Process Technol.*, Berlin, Germany: Springer-Verlag, 1996, pp 108–124.
2. D. Spinellis, "The information furnace: Consolidated home control," *Pers. Ubiquitous Comput.*, vol. 7, no. 1, pp. 53–69, May 2003, doi: 10.1007/s00779-002-0213-8.
3. P. G. Armour, *The Laws of Software Process: A New Model for the Production and Management of Software*. New York, NY, USA: Auerbach, 2003.