

Best Practices for Appliance Control Architecture

**Dale Mayes
Home Port Engineering, LLC
20004 103rd Court NE
Bothell, WA 98011
U.S.A.**

Telephone: 425.876.1915

Fax: (484)970-0941

E-Mail: DMayes@HomePortEngineering.com

Web Address: www.HomePortEngineering.com

ABSTRACT

By having a robust software design, a system can be more cost effective, faster to implement, and easier to maintain. Control software must also be structured to minimize the impact, and quickly recover from any effects of noise. Typically, noise affects a micro's ports and interrupt-registers, but the contents of any memory location can be suspect.

This paper/presentation explores some of the best practices for designing appliance control systems. It also introduces three views needed for an adequate description of a control's software architecture:

- *Design Overview diagram*
- *Thread of Control diagrams*
- *Data and Control Flow diagram*

BEST PRACTICES TO MINIMIZE THE EFFECTS OF NOISE ON SOFTWARE

By building certain precautions into a program's structure, the operational effects of external noise can be minimized. Each processor family has its own inherent weakness that must be factored into the design. In some cases, additional hardware filters are required on certain ports. For other devices, a special feature may become erratic under harsh conditions. The following best practices for implementing a noise immune design will generally improve the robustness of any processor.

Internal Registers

At reset, initialize all registers. Never assume they will all be 0s or Fs. RAM should be cleared and seeded with initial values.

At the top of the main loop, there are several things that should be done continually to help recover from noise:

- Reset the stack pointer. Since this is the main loop, there should not be anything on the stack.
- Mask off all unused interrupts and re-enable any applicable interrupts.
- For I/O used as inputs, re-assert the appropriate value to enable the input operation.

All vectors must be mapped. Unused interrupts can just jump to some benign function. If coding in assembly, adding a function label just before the return instruction of a utilized interrupt is a code-free possibility. If coding in C, you can declare an empty interrupt routine. Your optimizing compiler should automatically perform something similar to the assembly trick. Prior to the final product release, all unused program locations should be filled with a single byte instruction. No-ops are acceptable if the last memory location does something safe, or jumps to a recovery function. A recovery function can either restart the system, or trap the error by looping until the watchdog kicks-in.

Interrupts

If possible, avoid external interrupts. Noise can cause multiple interrupts that might overflow the stack. If the timer interrupt rate is fast enough,

instead of enabling the external interrupt, poll the input for its status.

If an external interrupt must be utilized, enable the interrupt for as short a time as possible. (e.g. When monitoring feedback for phase firing a magnetron relay, only enable the interrupt when closing the relay.)

Do not rely on any feature that can only be initialized once, like some timer interrupt's auto-reload functions. Rewrite the internal timer interrupt load registers continuously (if possible). This allows the processor to recover if noise affects the timer load register.

Iterative or reentrant interrupts can be very dangerous. Never have loops in an interrupt. A well-designed interrupt gets-in, quickly does something useful, and gets-out. Disabling interrupts in the main loop should be avoided.

In order to have enough processor cycles to execute the main loop, a program should never spend more than 60% of its time in the interrupts. A good way to determine the percent of time the system is processing interrupts is to temporarily commandeer a port. Add some debug code at the very top of every interrupt that turns the port on. Just prior to exiting the interrupts, turn the port off with some more debug code. An oscilloscope connected to the port gives a quick visual of the time spent processing interrupts. Be aware of the overhead to enter and exit an interrupt. A typical processor pushes five or more registers, and performs an indirect long jump going into the interrupt service routine. It then has to pop all the registers, and do another long jump to return. If you have frequent short interrupts, you may be unwittingly spending all of your CPU

cycles jumping around but not accomplishing much else.

Ports

De-bounce all inputs. This will reduce the chance of noise causing false readings. It may also allow for the removal of additional hardware filters.

If possible, periodically rewrite output ports and update their status registers. To ensure the outputs are in their correct state, continuously evaluate and refresh their value.

Unless you disable interrupts, it is not possible to reliably drive a shared output port from both an interrupt and the main loop.

Never leave unused ports floating, tie them appropriately high or low.

Watchdog and Time Base Monitoring

Do not rely too heavily on a built-in watchdog that does not have its own clock. If the micro's clock fails, how will the internal watchdog reset the system's outputs?

Kick the watchdog in the main loop, not in an interrupt. In most cases, if the main loop gets lost, the processor will still be properly accessing the timer interrupts.

If zero cross is used as a time base for safety related outputs, its loss should be detected and these outputs turned off until zero cross returns.

External Memory and Serial Communications

In a noisy environment, addressed busses, like I2C, are far superior to a microwire type buss.

For extra protection when using external EEPROMs:

- Precede each READ instruction with an “Erase/Write Disable” instruction.
- Each read should actually be a read verify. In other words, the information should not be used until the same data is seen twice in a row.
- The EEPROM's checksum should be verified periodically, either when a large block of data is brought into RAM, or prior to the start of a cycle.
- Any loops used for writing or reading should include time outs.

KEY DEVELOPMENT PRACTICES FOR ROBUST SOFTWARE

Developing robust software takes more than a couple of pages of coding techniques and tricks. Only by creating a repeatable process, will your organization be able to consistently realize truly robust software.

Implementation

Have two modes of the program; the development mode where errors and unexpected events are trapped and the deployment mode where errors and unexpected events are automatically recovered from gracefully.

The hardware engineer, PCB designer, and software engineer should collaborate on port assignments. Typically, what is desirable for the board layout is good for the software. For example, both like grouping similar outputs on the same port and using sequential pins for display busses.

The software developer should also work with the usability engineers to understand the underlying customer drivers. What is desirable for the customer (consistent operation) is typically advantageous for the software. For instance, always requiring the “start key” to accept new data vs. an on-the-fly edit with no start key, can help to reduce or avoid complex exception handling in the code.

Institute coding standards and development guidelines. It will make reviews more productive and enable other developers to assist on a project as needed.

When commenting a program, a theory of operation statement in the header is more effective than lots of general comments. In other words, do not have a comment on every line.

Use meaningful names for functions and variables. Establish a naming convention that easily differentiates between local and global variables names.

Planning

How accurate are your software development schedules? Do you know what your department's production rate is? If you do not know what your (pick a metric - lines of code, function points, feature points...) per person month is, how is your next schedule going to be any more accurate? For small-embedded systems, like an appliance control, weeks per kilobyte of ROM is a simple and effective metric. It avoids all the debates over what is a line of code and the complexity of calculating 'points'. If you can not extract historical data to estimate an initial X weeks per

kilobyte, just pick a number. Do a project, and then adjust the value.

How do you know if a project is on track? Will it fit in the ROM size available? Simple metrics will provide an early warning. As part of the design phase, establish budgets for each module's size and time. Size includes RAM, ROM, and EEPROM usage. Time covers both runtime (processor utilization estimates) and development time.

Build the system iteratively. Focus first on the code that will interface with the controller's hardware, next develop some of the basic appliance cycles, and lastly implement all the exceptions and cycle interactions.

Tools and development environment

Institute common tools across the developers, and ensure they all have the same version.

Use a configuration management system. It will free you to experiment with new ideas, because it is easy to jump back to the "last known good" version of code if the experimental code does not workout.

How conducive is the developer's space to productive code generation? In the book "Peopleware", Tom DeMarco and Timothy Lister, documented that a developer's work environment accounted for a 3 to 1 coding improvement.

Testing and Reviews

A system should have three testers. The developers must always have the first opportunity to test their code. This will accomplish two things. It gives the coder a chance to correct any silly mistakes. It also gives developers

ownership of the quality. Next, it is crucial to have someone other than the developer test the code. It is not that developers are not good testers; it is just that any misinterpretations they had in creating the code will be carried into testing. Finally, a quality assurance or final systems test should be done outside the design group. Once the code moves away from the developer, formal issue tracking should be employed, as well as, configuration management to track in what version of code a bug was found.

Peer reviews, design reviews, and code inspections can be very effective. These are not only efficient ways to find errors in the current code, but it can help a group share best practices and train junior developers. Utilizing a strict syntax-checking tool can save time in the code reviews.

Before you can have an effective design review, you need to have a documented design. This does not mean you should have reams of flowcharts showing every anticipated if-then-else statement. Three views are needed to adequately describe a software architecture. The Design Overview diagram sets the context of the control system. The Thread of Control diagrams show what processing will take place where. The Data and Control Flow diagram provides a comprehensive view of the data and process interaction.

DESIGN OVERVIEW DIAGRAM

As any software engineer can attest, the microprocessor is the center of the universe. So it follows that the

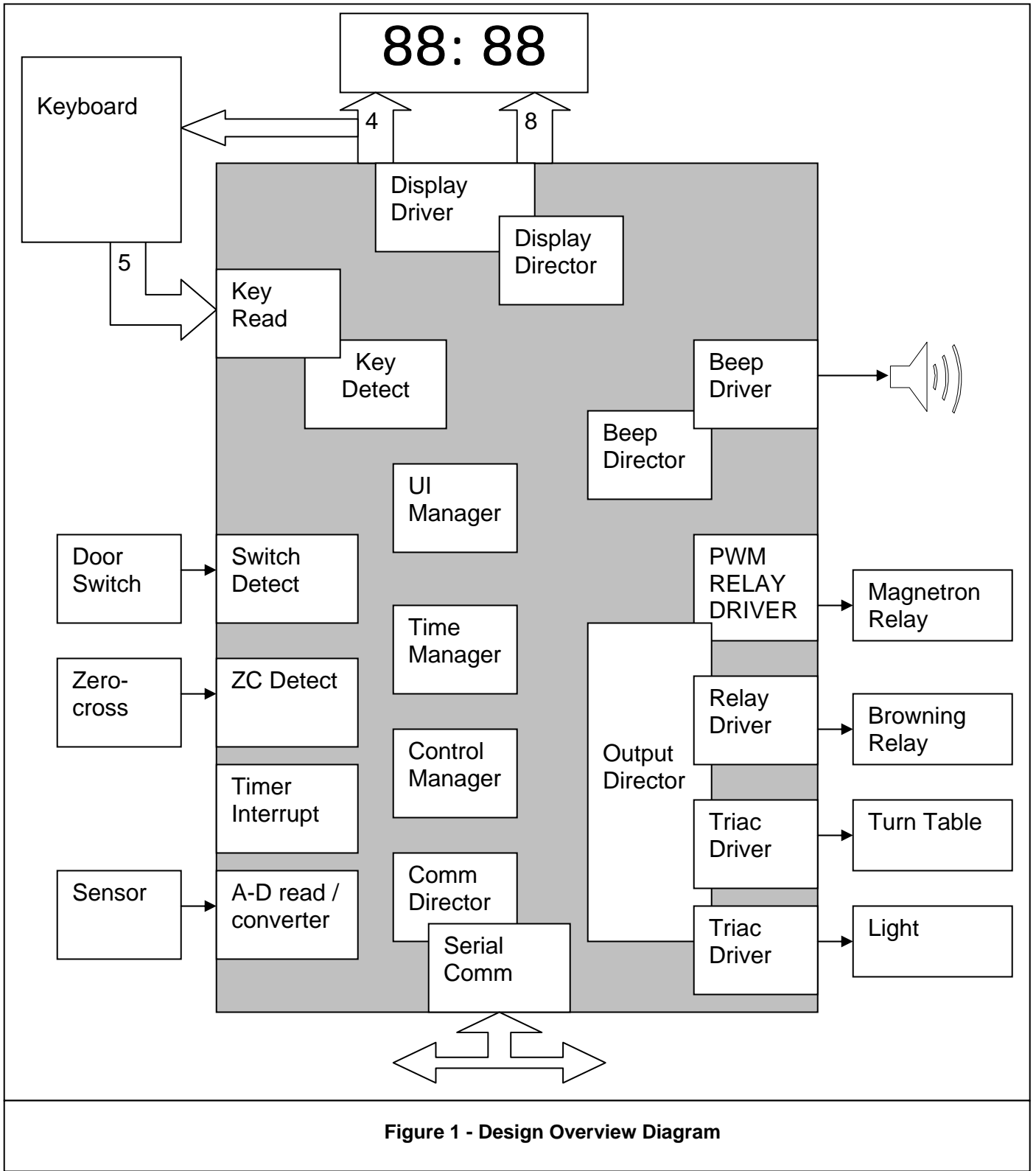


Figure 1 - Design Overview Diagram

micro would be at the center of the Design Overview Diagram, see Figure 1 on the preceding page. This diagram identifies the physical devices at the end points of a fictitious microwave oven control system. The diagram also identifies the major software modules located in the gray microprocessor. Ideally, each of the physical device blocks would include a short paragraph describing the interface. For example, the door switch's description would include:

The door switch reflects the status of the microwave oven's door. A value of 0v at the microprocessor indicates that the door is closed and a value of 5v indicates the door is open.

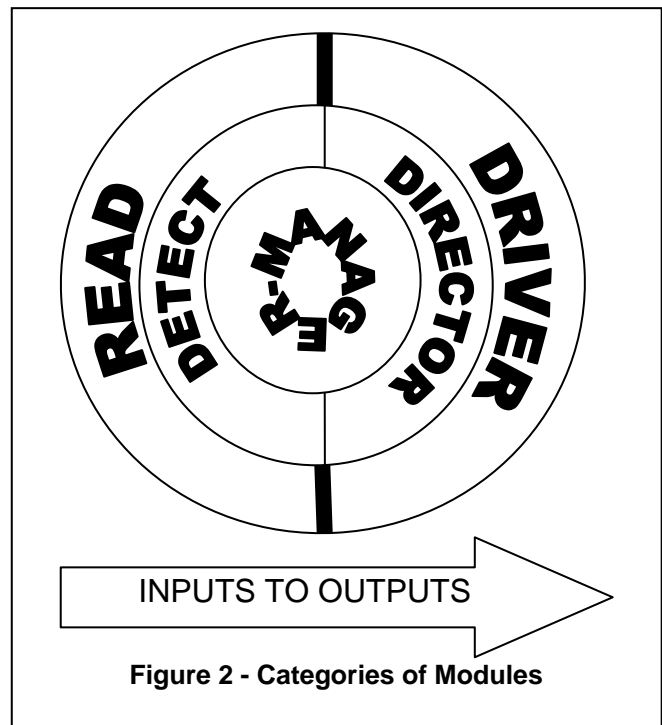
The software modules on the Design Overview Diagram fall into five categories. See Figure 2 for how these categories are logically organized.

Sometimes the read and detect operations co-exist in one module.

- READ – Captures input values for the detect modules to de-bounce.
- DETECT – De-bounces, filters, and formats input values for the manager and director modules.
- MANAGER – The main logic for the control.
- DIRECTOR – Prioritizes and validates output requests.
- DRIVER – Output to devices.

A good architecture will provide ample opportunities for reuse. Notice in Figure 2 how the Manager modules are isolated from the Read and Driver modules, and visa versa. This enables the Read and Driver modules in the outer ring of Figure 2 to be easily reused

across products. In other words, a relay drive module does not care if it is in a dishwasher or a washing machine. The Manager modules at the center of the diagram can be easily reused within a model line because the Detect and Director modules format and convert data for the Manager.



THREAD OF CONTROL DIAGRAMS

Flowcharts are the best way to represent the thread of control for the main loop and all the interrupts. The flowcharts should not be too detailed. They should just show where the major processing is taking place for each of the software modules identified in the Overview Diagram. State Charts are useful for documenting cycle logic and user interface transitions. Some of the modules will have functionality in a

couple locations. For instance, the Timer Manager typically has some functionality in an interrupt and some in the main loop.

There are two common main loop structures, Expanding and Slotted, see Figure 3 and Figure 4.

Both structures consist of a start-up (reset) branch, and a common top branch.

Typically, the main loop is loosely coupled to a time source. While the main loop is waiting for a time tic (zero cross is used in the example) it loops through the Top of Main section. The Top of Main section is a good place for noise immunity operations discussed previously.

Expanding Main Loop structure

The Expanding Main Loop, Figure 3, grows depending on what conditions are met. The “Once per ZC” (zero cross) section is executed every 16.66ms for 60 Hz systems. There could then be a “Once per 250ms” section that is executed once for every 15 “Once per ZC” iterations. There will typically be a “Once per Second” block. The sections are not limited to just time. Usually there is a “Handle New Key” block of code that is executed whenever a new key is detected.

The “Once per ZC tic” should be a counter, not a flag. In other words, the time to get around the longest possible main loop can be greater than 16.66 ms. The main loop can ‘catch-up’ because the time to get through the “Once per ZC” block is much less than 16.66 ms.

Some of the advantages of this structure include its ease of adapting and implementation. The number of

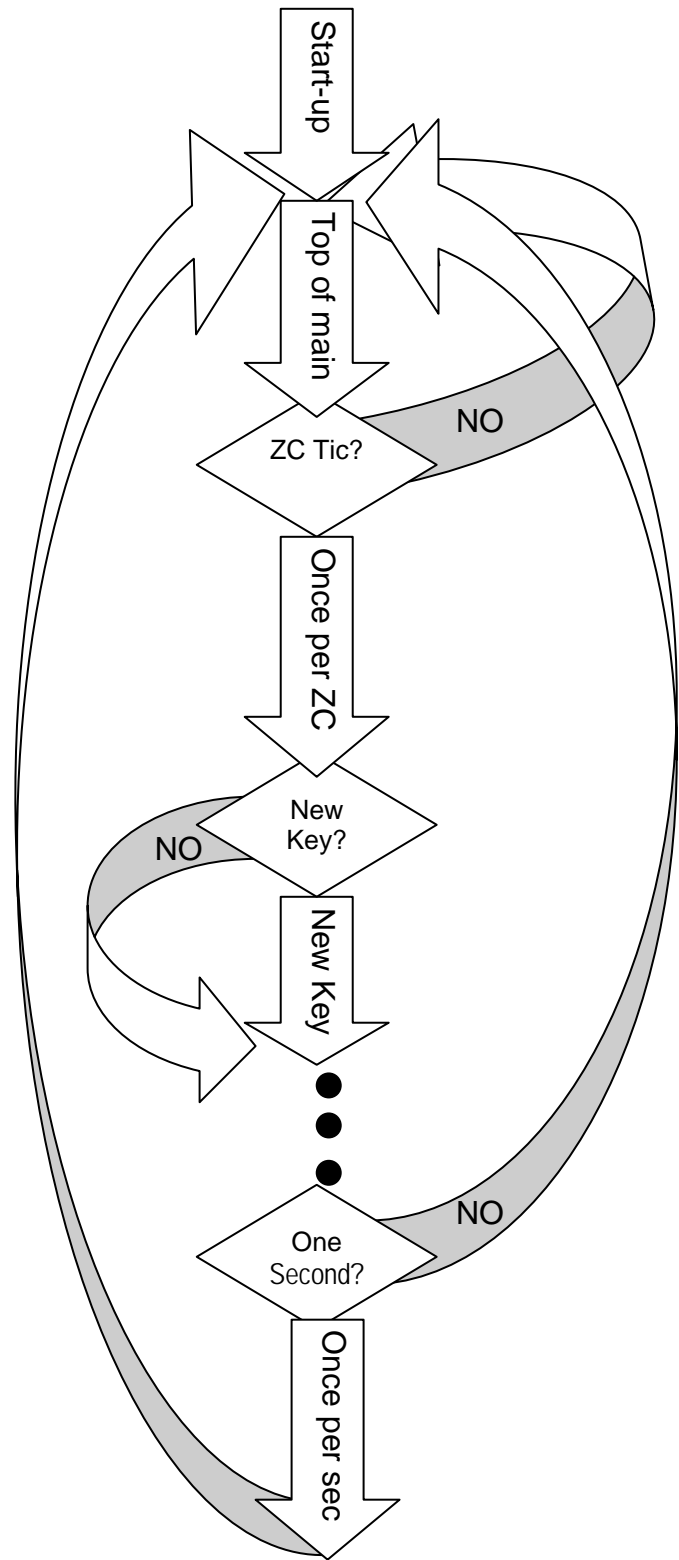


Figure 3 - Expanding Main Loop Structure

sections and their time bases or conditions for entry can be adjusted for your programs needs.

By kicking the watchdog in one of the sections, both the main loop integrity and the source of the timing tic can be verified.

The downside of the Expanding Main Loop structure is that it is difficult to predict what the longest path through the main loop will be. This can cause the timing to be indeterminate for brief periods. Like when the loop is 'catching-up' from a long main loop pass, the "Once per ZC" section could actually be executing every 2-3 ms instead of every 16.6 ms.

Slotted Main Loop structure

In the Slotted Main Loop structure, the loop executes the Once per ZC, One Slot and the End of Main each pass.

By adjusting the number of slots to a convenient multiple of the tic rate, it is easy to build a time base inherently into the loop. There is nothing wrong with having empty slots to get to a good multiple.

The slotted structure has two sections for 'faster' processing, the Once per ZC and End of Main.

This structure tends to be more deterministic in its time for a main loop pass. This is both a blessing and a curse. Because each module has a time budget, it is easy to see if a module is consistently exceeding its time allowance. However, because each module has a time allowance, it is another thing for the developer to monitor. If one module is consistently too long, it may need to be partitioned across two slots.

By kicking the watchdog in the bottom of the Slotted Main Loop, both the main loop integrity and the source of the timing tic can be verified.

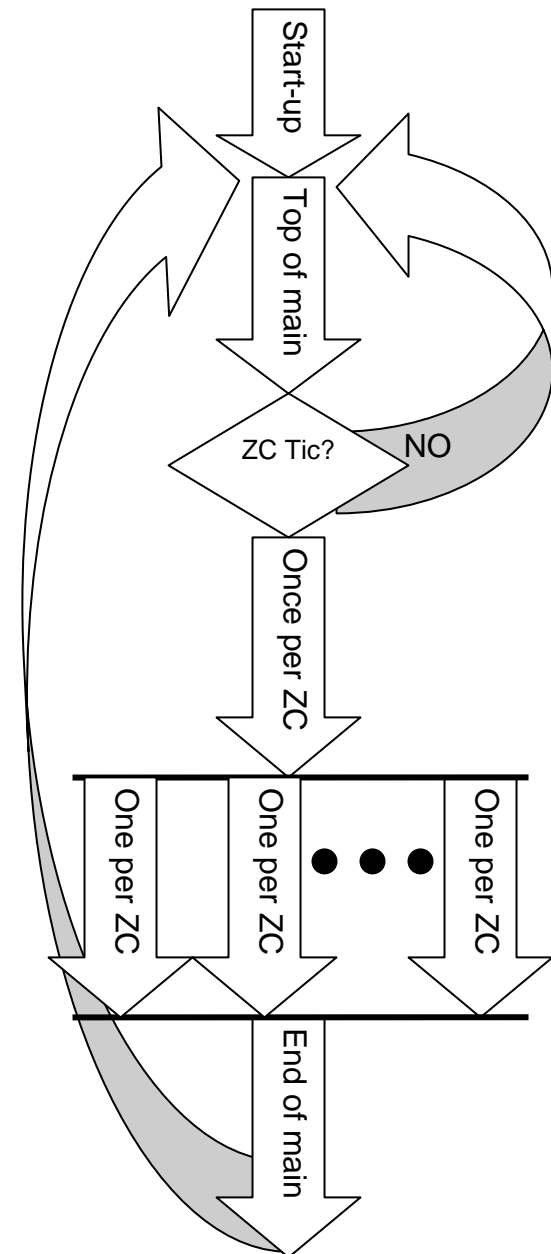


Figure 4 - Slotted Main Loop Structure

This slotted structure can also be applied to an interrupt. If an interrupt is consuming too many of the processor's cycles, the tasks can spread over several slots thereby reducing the number of tasks that are executed in a given interrupt pass.

CONTROL AND DATA FLOW DIAGRAM

The Control and Data Flow diagram represents how the major modules interact. It contains information on both the control threads, (the thin lines) as well as, the data paths (the fat lines). See Figure 5. It is actually a hybrid of a Data Flow Diagram and a Control Flow Diagram as described in Pressman's "Software Engineering a Practitioners Approach".

The boxes represent the modules. The ovals represent the global data structures the modules use to communicate. The arrowheads provide perspective on data access. An arrow into a data element represents writing data. An arrow from a data element represents a read. Figure 5 is very cramped; the diagram is typically done on larger paper.

One of the goals of a good design is to ensure there is efficient partitioning with minimal overlap in responsibilities between modules. Think about your job. How effective are you if you have three or four bosses telling you what to do?

The Control and Data Flow example demonstrates how the diagram surfaces potential problem areas. Notice how two relay drivers are both trying to manipulate the same port byte. As stated in the best practice section, this is not possible unless interrupts are

disabled, which is not a good practice. It would be better to have the main loop's driver write its pin's data to a port mirror byte. The relay driver in the interrupt can then 'OR' its pin's value with the mirror byte and write the port.

This diagram is also good for prioritizing design reviews. Any module, which uses more than four inputs, should be reviewed. As well as any modules that split their functionality between two threads of control. Any modules interacting with a data structure that is manipulated by multiple modules should also be reviewed.

DETAILED DESIGN

These three diagrams can be used iteratively to refine a design. The Design Overview diagram can set the context of a module in the system. The Thread of Control diagrams would show what processing will take place in what functions. The Data and Control Flow diagram provides a comprehensive view of the local and global data, and process interaction within a module.

REFERENCES

DeMarco T. and Lister T. (1987) *Peopleware: Productive Projects and Teams*. New York: Dorset House Publishing Co.

Pressman, R. (1992) *Software Engineering a Practitioners Approach (third edition)*. New York: McGraw-Hill, Inc.

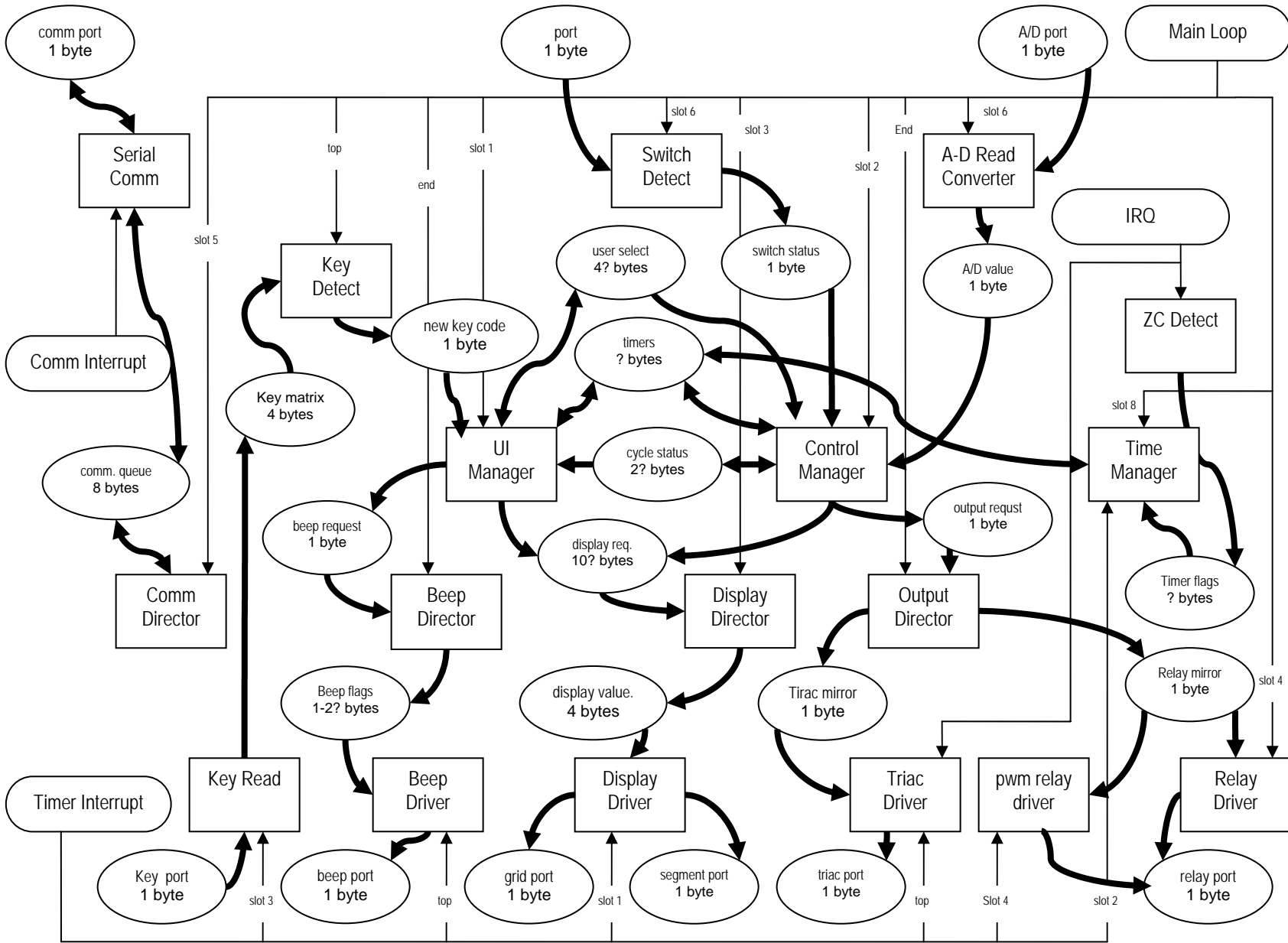


Figure 5 – Control and Data Flow Diagram

CONCLUSION

Developing robust software takes more than a couple of pages of coding techniques and tricks. Only by creating a repeatable process, will your organization be able to consistently realize truly robust software.

By building certain precautions into a program's structure, the operational effects of external noise can be minimized

Design reviews are very effective for finding errors. They are also a great way for a group to share best practices and train junior developers.

Before you can have an effective design review, you need to have a documented design. This does not mean reams of flowcharts. Three views are needed to adequately describe a software architecture. The Design Overview diagram sets the context of the control system. The Thread of Control diagrams show what processing will take place where. The Data and Control Flow diagram provides a comprehensive view of the data and process interaction. These three diagrams can be used iteratively to refine a design.