A Pattern Language for Simple Embedded Systems

Introduction

A pattern language is a set of cooperating patterns that combine to provide solution guidelines for a problem in a particular context resolving the identified forces. This is a pattern language for simple embedded systems.

The world has thousands of applications running on millions of micro-controllers that utilize simple executives or "bare metal" applications. These applications typically are built around one of three configurations. First, the pre-emptive multi-tasking system is the most complex. Second, the co-operative multi-tasking system, requiring tricky synchronization planning. Third, the carousel or super-loop foreground/background processing system, is the smallest and simplest design.

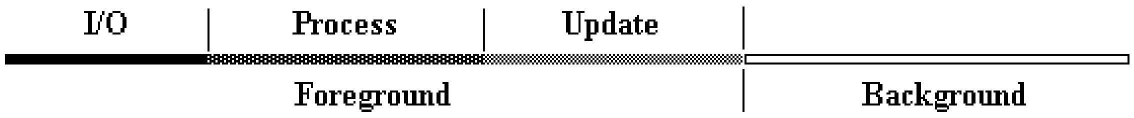
The solution space includes process controllers, consumer boxes (e.g. VCRs, microwaves, stoves, etc.), automotive modules, and many other applications. These systems often have limited user interaction, accept data from a variety of sensor inputs, and control a few outputs. The design of all embedded applications is challenging as it often has hard timing restrictions, limited processing resources, and severely limited RAM and program memory. Many of these systems do not use Real Time Operating Systems (RTOSs) for various reasons including recurring licensing costs and/or RAM and ROM resource demands. A good multi-tasking solution can be very elegant, but difficult to design and debug.

The pattern language is for the world of simple embedded systems that lie below the threshold of an executive or RTOS. We begin with that part of the language that defines the framework for the third configuration, *The Carousel*. All the other patterns can fit into this framework to provide a comprehensive embedded system that is easy to understand, develop, debug, and maintain due to its simplicity and consistency of operation. This pattern language does not address initialization or shut down of the system as this time.

The Carousel

...assume that you have chosen to design your embedded system based upon a fixed cycle time. Within the cycle, processing needs to be allocated for reading system inputs, processing the new inputs into the internal system state, preparing a new set of outputs reflecting the new state, and setting system outputs.

How should *The Carousel* organize the processing time?



Organization of continually repeated activities in an embedded system requires some coordination with outside interfaces and processing to modify the system state as the inputs change and events occur. Most of these systems require that some of their primary inputs and outputs occur at fixed frequencies with little or no jitter in the external update times.

The Carousel normally operates as a contiguous block of processing. Use of sequential processing means that most design considerations related to multi-threaded/multi-tasked solutions are eliminated or significantly simplified with critical regions becoming "automatic", and semaphores and other multi-tasking cooperation strategies unnecessary. The only caveat is to be careful in areas that could have data changes caused by asynchronous events. Simply disabling interrupts for the minimum duration to protect the critical lines of code can protect these areas.

The organizing of the activities within each cycle of the carousel is critical to most systems. There are three major groupings of activities forming the foreground of the cycle. The first group of activities contains synchronous data I/O activities. Hardware peripherals are read and outputs are set at this time to have them occur at known frequencies with little or no jitter. No additional processing of the inputs is performed at this time. This first stage requires that the total execution time be kept very short, preferably less than the time that any *Asynchronous Activities* such as serial communication ports can be disabled. In the strictest of cases, this group of activities would take exactly the same portion of each cycle.

The second group of activities contains the event processing activities. The samples are processed as necessary and then this is where objects within the system look at the processed inputs and apply necessary processing to change the system internal state. The sequence of processing usually proceeds from the outside of the system (the peripherals and samples) to the inside (system state). This group of activities usually takes a minimal amount of time on most cycles assuming that changed data events causing updates occur at a rate much slower than *The Carousel Rate*.

The third group of activities contains the updating activities. This is where objects modify the data that will be output during the first section of the next cycle. This section includes two sub-sections that perform any filtering on the data inputs and prepare all data outputs. The previous event processing is separated from the updating so that the system should be in a consistent state prior to updating the outputs.

Therefore:

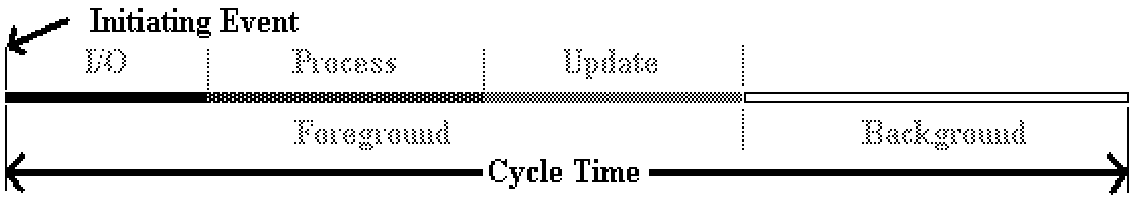
Separate the external state sampling from the processing of the samples. Separate the processing of the samples from the internal state update. Separate the internal state update from the new external state generation.

Once the activities are grouped, the frequency of *The Carousel* needs to be established to identify the processing resources available for the activities.

The Carousel Rate

...now *The Carousel* has allocated the different processing activities within the cycle, we need to decide what event initiates the cycle and how frequently the event occurs. The cyclic execution of the system requires determination of a fundamental frequency of operation. This time base sets the processing capacity, the response time, and the sensor sampling rates. There are many possible sources for a time base such as a Real Time Clock chip, a timer/counter hardware interrupt, a video retrace signal, or any other continual event.

How fast should *The Carousel* "spin" for your system?



The Carousel cycle initiation is usually tied to a high priority interrupt to maintain the low jitter sampling. This interrupt is masked off after the time critical portion of the I/O activity so as not to inhibit other lower priority *Asynchronous Activities* during the remainder of its execution.

The Carousel Rate provides a stable synchronization to some continual external event. The results of this temporal coupling is to provide predictability in some or all of the following areas:

1) stable, low jitter sensor sampling (e.g. integrating rate sensors to calculate position accurately);
2) time based messaging protocols (e.g. messaging at a fixed frequency);
3) synchronized video display update (e.g. avoid "snow" by updating video memory during retrace periods);
4) hard limit response times on asynchronous events (e.g. update an output within some time limit); and
5) stable, low jitter outputs (e.g. soft modem outputs).

The frequency selection comes from examining the synchronous activity rates, both input, and output, and usually selecting the fastest rate as a base rate or possibly the lowest common multiple rate for multiple activities. The base rate must also consider the response time to any critical asynchronous event. The usual operating scheme is to gather the input from *Asynchronous Activities* and deal with it at a fixed position in the cycle so the maximum latency to handle the event is one cycle time. Careful consideration at this point may allow some *Asynchronous Activities* to be implemented as polled activities versus interrupts, simplifying shared data access concerns to sequential access.

Activities that are not operating at the base rate or not driving the lowest common multiple, can be deferred for consideration in *Too Fast For Me*, however their processing demands should be estimated at this point to avoid nasty surprises in processing loads.

Therefore:

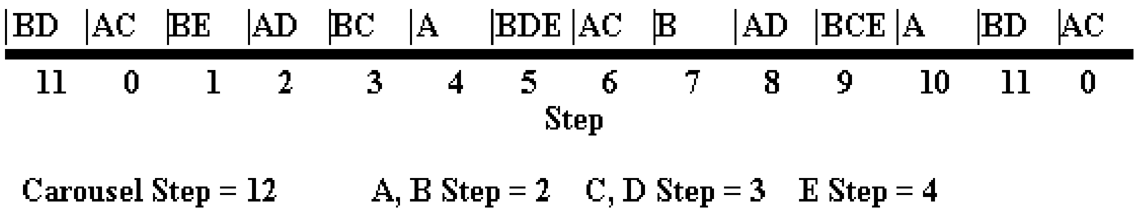
Examine the rates required for all synchronous events to establish a base rate for *The Carousel*. Determine that critical response latencies are met and processing resources are not over-allocated.

Once you have established *The Carousel Rate*, *Asynchronous Activities* and *Too Fast For Me* activities can be inserted into *The Carousel*.

Too Fast For Me

...assume that some of the inputs, outputs, and processing activities do not need to be performed at *The Carousel Rate*, we need to integrate these activities into *The Carousel*.

How should slower activities be integrated into *The Carousel*?

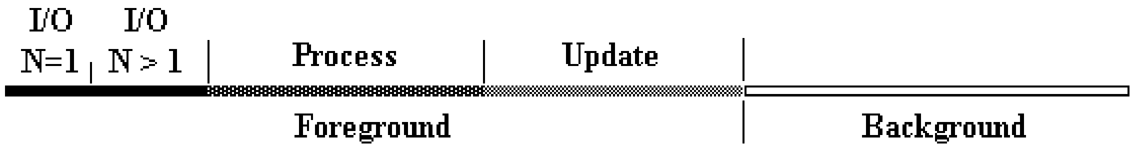


Many of the activities identified in *The Carousel Rate* may not require attention on every cycle. These activities need to be allocated to avoid overloading the processor in any single cycle.

These activities can be adjusted to a related divisor of the base rate. The rate may be a half, a tenth, or some other fraction. This means performing these slower activities on every N cycles where N is greater than or equal to 1. The cycle now requires a "step" counter for this purpose.

The allocation of sub rate activities can be performed to equalize processing loads by interleaving activities with common multiples in their time base. The simplest example would be performing Activity A (N=2) when "step" is even and Activity B (N=2) when "step" is odd. For this example, step would only need to cycle in the range of (0..1). It is common to need a step (0..9) or (0..11) in many systems. Big Calculation or any other processing activity may also use this step value.

Activities that operate very infrequently (N>~20) may use there own internal counters to determine when they need to execute, rather than forcing the more frequent sub rate activities to use mod functions or case statements to determine if activity should occur on a given cycle.



The sub rate activities are put in their processing groups of *The Carousel*. The positioning of these activities is only important in the I/O group. The slower activities are inserted after the full rate I/O activities so as not to change their timing. The ordering of multiple slow rate activities in a single cycle is usually faster to slower, but it may be changed around depending on which activities have a requirement for low jitter.

Therefore:

Select a step counter to handle those events that operate at a fraction of *The Carousel Rate*. Assign the slower activities in an interleaved manner to equalize the foreground processing in each cycle. Position lower rate I/O activities after the full rate I/O activities.

Interleaving and distribution of *Too Fast For Me* activities across multiple cycles evens out levels of foreground and background processing. Next the *Background Activities* are selected and prioritized.

Asynchronous Activities

…assuming that some activities cannot be polled or require a response time that is much faster than can be met at *The Carousel Rate*, we need to handle these activities with interrupts.

How should interrupts fit into *The Carousel*?

The priority of each interrupt needs to be considered weighing response time against the duration of the I/O activity of *The Carousel*. The frequency of the interrupts must also be considered to estimate the impact on total processing resources.

The length of time that an interrupt can be ignored is ideally greater than the time required by the I/O activity of *The Carousel*. If that is not the case, then the trade-off between priorities must be performed. If the interrupt is higher priority, try to reduce the interrupt period to as short a time as possible by separating the recording of the event from any associated processing if possible. In the case of communication interfaces, the normal implementation is to send or receive data in the interrupt, while queuing of a message to transmit, or detection of a complete input buffer and any other related processing occurs during the process portion of *The Carousel*.

Therefore:

Identify any *Asynchronous Activities* with relative priorities and frequencies. Determine whether they will impact *The Carousel* I/O activity critical time segment or add an unacceptable processing load to the system. Implement any interrupts with as little processing as possible inside the interrupt handler by deferring input processing and pre-computing outputs as possible.

After this point, the primary drivers for system timing have been satisfied. The only likely further influence might be *Big Calculation* if it is part of the design. Now the *Background Activities* can be examined to determine what to do when you are finished *The Carousel* cycle and waiting for the next initiating event.

Background Activities

...since *The Carousel* does not consume all of the interval with its processing, we need to define what happens during the remainder of the time. This background time segment can easily handle a queue of prioritized activities.

What should I do while I'm waiting around?

The *Background Activities* selected for this time will be prioritized. All the activities will run to completion except the lowest priority activity. A decision is made at the completion of each background activity to select the next highest priority activity. If the lowest priority is being executed, it will only re-examine the queue at the start of the background segment.

The enabling and queuing of background activities is done within *The Carousel* by gathering all the inputs to assure data coherency, and then queuing the activity. The enabling could be in response to some detected event, state change, or staleness notification.

Therefore:

Identify each possible background activity and assign priorities. Process the highest priority activity when the current activity completes, or if the lowest priority activity is executing, wait for the start of the next background segment to re-examine the queue.

Common choices for the background activities include:

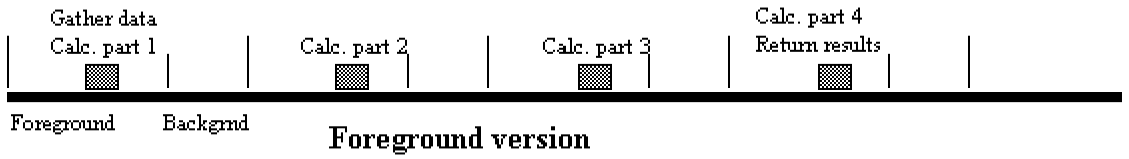
1) *Big Calculation* to handle a large execution time calculation;
2) *Nap Time* to conserve power in the system; and
3) *Maintenance Check* to verify system health.

Once the *Background Activities* are set, then processing assigned to the three foreground activity groups of *The Carousel* can be detailed.

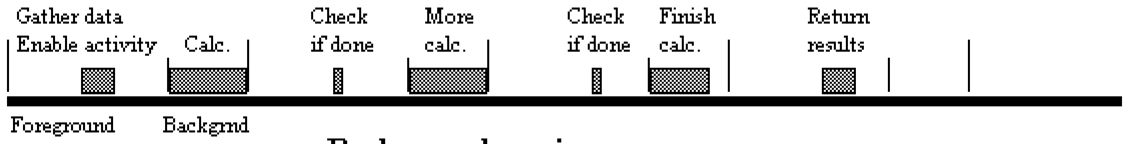
Big Calculation

...assume now that your system has a calculation that requires a large execution period. This time demand could be the equivalent of many cycle times. There are many options as to when to allocate the processing depending on the acceptable latencies in getting the result. Typical calculation types would include PID calculations in process controllers, robot path calculations, and Kalman filters in GPS systems.

How should the *Big Calculation* be integrated into *The Carousel*?



Foreground version



Background version

The *Big Calculation* may be either a continually updated result, or it may be an event driven activity that demands execution on an infrequent basis. The latency requirements of the calculation results may also force a particular update method.

The *Big Calculation* usually requires that all the input data is collected at the start so the data is temporally coherent for the calculation. Exceptions to this will include some types of predictive filters that perform more accurately if they utilize the most recent data for each portion of the calculation. The end of the *Big Calculation* should also be synchronous as partial updates of the output data may lead to glitches in other outputs. E.g. Calculation of latitude, longitude, and elevation in a GPS should be returned together as some other part of the system may use this *Big Calculation*'s latitude with last *Big Calculation*'s longitude and generate an incorrect interim value. This implies that the *Big Calculation* data collection and data return activities are part of "foreground" activities to ensure the coherency.

The allocation of the *Big Calculation* processing is determined by considering the following three variations. Calculations that require continual updating while maintaining *The Carousel* can be divided into known, fixed size portions that are executed in the "foreground" during sequential cycles in the processing portion of *The Carousel*. The result is a fixed latency and constant update rate. Should the goal be to minimize latency within *The Carousel*, the solution is to utilize the "background" time slot to perform the calculation using all available spare processing time to provide the most rapid calculation without disrupting *The Carousel* while sacrificing a constant update rate. This "background" assignment of the *Big Calculation* may be enabled either as required, or continuously depending on the result requirements. The third case for a *Big Calculation* requiring minimum latency, is to execute it to completion upon identifying the triggering event and interrupt *The Carousel* for the duration. This corrupts the cyclic time base so only use this version for very infrequent calculations where *The Carousel* can restart between calculations.

Therefore:

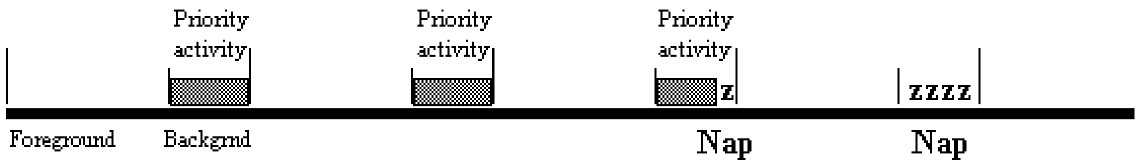
Examine the latency and the update frequency for the *Big Calculation* to select a processing allocation strategy as foreground, background, or interrupt. Gather the data, crunch it, and then return it.

Once the *Big Calculation* is allocated, most other activities should be simple to fit into *The Carousel*.

Nap Time

...assuming that the embedded system has power saving requirements typical of thermal considerations and/or battery powered systems, the "background" processing time can be used to put the micro-controller and possibly associated peripheral hardware into an idle or power-down mode. These modes are typical of many micro-controllers, but may not appear in all varieties. The differentiation between the two modes is that idle mode keeps the peripherals alive, while power-down mode is a complete shutdown.

How should I take a nap?



Nap Time mode determination is usually made by the requirement to handle external asynchronous events. Some consideration should also be given to the restart time from the selected mode as the system may be required to respond very rapidly to asynchronous events.

If asynchronous events can occur during the "background" time or the peripherals require power to operate, idle mode is used, otherwise power-down mode may be used. It should be noted that often the hardware has different criteria for waking up from each mode, so they are usually not interchangeable in a given system.

Nap Time is usually a background activity and as such it will be assigned the lowest priority. Once the system is napping, it is assumed that the foreground must do some processing before any higher priority activities can be enabled, so the system naps until the next cycle initiating event.

Therefore:

Examine the requirements for handling asynchronous events and infrequent processing. Select the power saving mode and have a nice nap.

Once this activity has been selected as the low priority activity in *Background Activities*, the system can often reduce the average power demand.

Maintenance Check

...if the system has requirements to perform sanity checking for safety and/or functionality reasons, this activity should occur regularly. The scope of the checking performed here needs to be determined here along with the reaction to events and the frequency of monitoring.

When should I perform a maintenance check and what should be included?

Maintenance Check is performed outside the flow of *The Carousel*, so this activity should be limited to testing for system level conditions. Ongoing testing and diagnostics for sensors and outputs are found in the *Noisy World* and *Perfect World* patterns. The selection of possible checks in *Maintenance Check* activities could include the following:

1) System calibration;
2) Power stability check;
3) Remaining battery life check;
4) ROM sum check or cyclic redundancy check;
5) Communication link(s) check;
6) Peripheral Built In Test signal checks;
7) Maintenance port processing; and
8) Over temperature monitoring.

This pattern extracts much of the worldly concerns from the processing occurring within *The Carousel*. The activities identified here are typical of embedded systems that are a little higher up the food chain and demand some system level diagnostics. Selection of other maintenance activities for any system will usually be driven by customer requirements beyond our control. The *Maintenance Check* activities tend to be pervasive in any system, so any picture is of limited value.

If the system cycles at a rate much higher than it’s required response time to error conditions, The error checking can be distributed across multiple cycles as in *Too Fast For Me*. Battery life and over temperature checks are usually in this category. The paranoid system would assign *Maintenance Check* to *Background Activities* and use it as the lowest priority activity to keep all the checks as current as possible. Sum checks, maintenance port processing, and system calibration may be in this category. The most common system type records any status data when accessing each sensor/peripheral that may fail and integrating the results into the system state during the processing activity group. This usually includes communication link checks and peripheral Built In Test signal monitoring.

It is usually a good idea to use a consistent “polarity” on reporting all checks by using an IsOK() method for all faults and warnings.

Therefore:

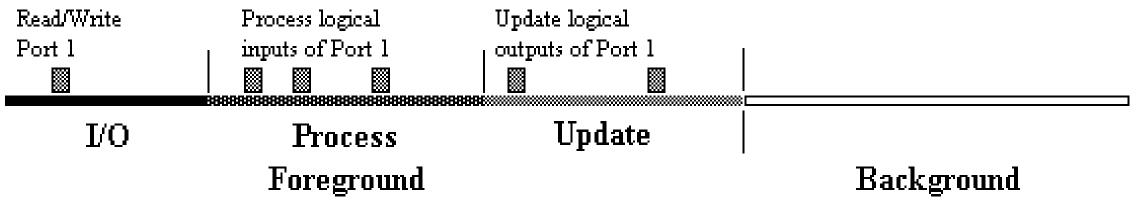
Select the collection of maintenance activities required for the system. Assign the collection and processing activities within *The Carousel* as appropriate.

Selection of these maintenance activities broadens the core system and provides an opportunity for small scale reuse as many of the activities are dependent on the base hardware and not necessarily on the specific application.

Sharing

...assuming that inputs and outputs of the system share common pieces of hardware, the multi-channel sensors, multi-bit digital I/O ports, and communication busses. The data going over each of these interfaces must be multiplexed or de-multiplexed to share the hardware. It can also be very time consuming to have each portion of the common hardware accessed separately be each interested party.

How should shared input/output interfaces be handled?


Read/Write Port 1
Process logical inputs of Port 1
Update logical outputs of Port 1
I/O
Process
Update
Foreground
Background

I/O pins on micro-controllers are often accessed as byte wide devices meaning that any read/update/write sequence must be protected from any interrupts that may affect the port update sequence. Many peripheral chips often have a serial interface that may contain multiple pieces of data. E.g. A multi-channel serial ADC. Coordination of access to these peripherals is essential for a stable system.

The safest and simplest way to deal with the shared I/O devices is to implement virtual I/O devices. *Sharing* provides a place to manipulate all of an interface in a single coherent access that is later separated into logical portions to be used by each interested party during the process and update groups of *The Carousel*. The changes of the I/O state is done during the update portion of the carousel cycle while the physical update occurs in the synchronous portion of the carousel. This single access provides for even tighter synchronization of input and outputs while performing them in an efficient manner to help reduce the total time taken in the I/O activities.

For peripherals such as the serial multi-channel ADC, *Sharing* can handle the reading of the serial interface in a method similar to *Too Fast For Me* and cyclically read a single channel per cycle. *Sharing* would de-multiplex the consecutive channels of the ADC and then call the processing components for each datum.

Therefore:

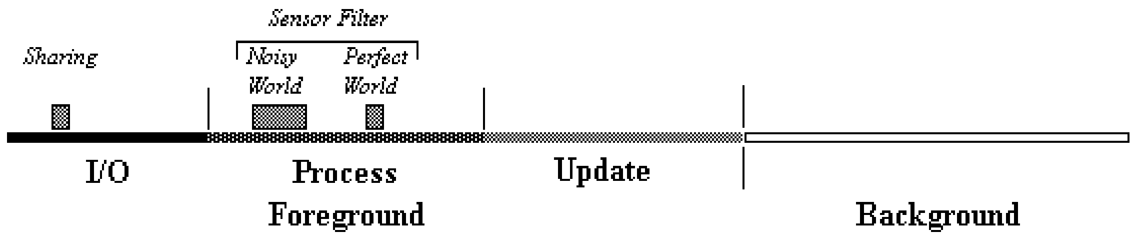
Separate the logical I/O groupings and physical I/O groupings with virtual I/O handling to make each access simple and consistent.

Sharing of the embedded systems I/O makes the interface internally consistent while enhancing efficiency.

Sensor Filter

...now the embedded system monitors some inputs signals, many of which are not clean, discrete, stable signals. A certain amount of processing or filtering is required to prepare the signal for use in the rest of the system. This activity must be integrated into *The Carousel*.

How should I handle input signals.



The *Sensor Filter* is responsible for making the external sensor inputs usable to the remainder of the embedded system. The requirements for processing the sensor come from both system specifications and hardware limitations.

The *Sensor Filter* separates the sensor conditioning into two distinct areas. The first is the *Noisy World*, which deals with the hardware and real world problems of processing the sensor data. The second is the *Perfect World*, which deals with application and processing restrictions derived for the application of the sensor data. The sensor data is handled first by the *Noisy World*, and then by the *Perfect World*.

Therefore:

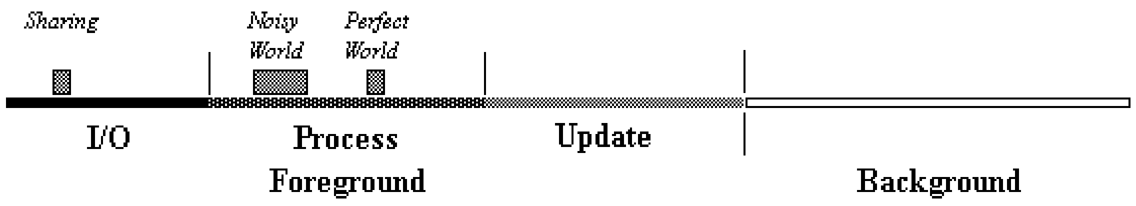
Separate the *Sensor Filter* activities into two groups, the *Noisy World* and the *Perfect World*. Process the data through each filter consecutively.

The results of the separation of processing means that the system can access either the actual conditioned sensor reading or a further filtered, guaranteed version of the sensor.

Noisy World

...now the actual values read from the embedded system inputs must be conditioned to extract the signal from the noise. This activity must be integrated into *The Carousel*.

How should I condition input signals.



The *Noisy World* signals suffer from disturbances, distortions, and input sensor limits. These problems need to be resolved to condition the signal for use by the *Perfect World*. Any concerns regarding the stability of signal sampling have been handles already in *The Carousel*.

The activity for *Noisy World* is to extract the most accurate signal from the input. The activities here should provide a stable signal with some resilience to transient faults and noise. Processing of the input could include a sensor/converter health check, anomaly clipping, a multiple sample averaging, dead banding, and result re-scaling.

The first filter is the sensor/converter health check to verify that the conversion hardware believes that the signal conversion is valid. This is typical of Analog to Digital Converters that indicate when a signal is outside the conversion voltage range. In this case the system may count up a number of consecutive conversion failures before declaring a fault.

The next three filters deal with noise at various levels. The first noise filter to handle large noise spikes is anomaly clipping. It ignores apparently valid samples that differ significantly from the signal's current average when the signal is not expected to change rapidly. This is typical of a temperature sensor measuring room temperature. In this case, the system may try resetting the average if it detects a number of consecutive "anomalies" or it may declare a fault.

The second noise filter is an averaging filter. Multiple sample averaging lowers the noise in the process at the expense of reducing the signal bandwidth. A four sample average would increase the apparent resolution by 2 bits, but reduce the bandwidth by one quarter. This filter settles out low level noise in the input.

The third noise filter is dead banding. Dead banding is the reverse of the averaging filter in that it ignores low level noise and reduces the apparent resolution. It is most common in position feedback systems where the system avoids hunting by not attempting to command a new position if the current position is within some dead band around the desired position.

The last filter is result re-scaling. Most sensors read in some scale that is not directly useful to the application and the value must be converted to some other units. As an example, a pressure sensor that returns a sixteen bit value may require an equation to convert binary to some useful units (p.s.i. or kPa) and linearize the conversion.

Therefore:

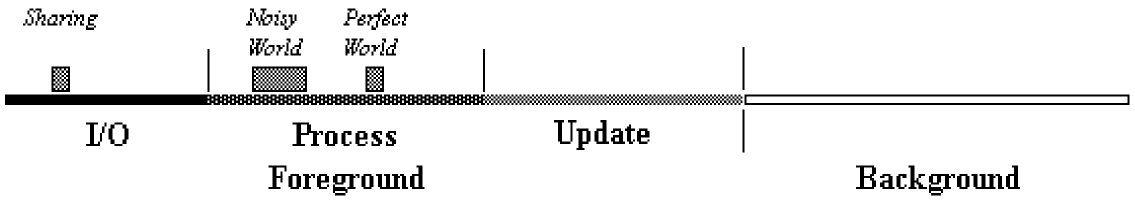
Select the processing necessary to extract useful data. Condition the signal and set or clear any detected fault states.

Careful control of the view of the *Noisy World* at this time provides the internals of the system with stable signal on which to base further processing.

Perfect World

...now the stable values read from *Noisy World* must be conditioned to provide a signal within expected limits and ready for use by the remainder of the embedded system. This activity must be integrated into *The Carousel*.

How should I handle the conditioned input signals.



The *Perfect World* signals need to be strictly controlled to provide stability for all other users of the filtered input. It is usually required that a valid sensor reading be always available.

The activity for a *Perfect World* output is to provide a carefully controlled version of the output from a *Noisy World*. The activities here should provide a signal inside an expected range under all conditions. Processing of the filtered signal could include cross-coupling, soft limiting, and default/override.

The first filter is cross-coupling. Cross-coupling means combining multiple inputs to create or calibrate a sensor. This may be using the output of an air temperature sensor to generate a correction coefficient for an air pressure sensor, or combining the air temperature and pressure sensors to derive an air density sensor.

The next filter is soft limiting. The sensor output from *Noisy World* usually will be able to generate valid readings outside the range that the system considers acceptable. As an example, the air temperature sensor may read down to -40 degrees, but the system may wish to declare a warning or a fault if the temperature goes below 0 degrees. A soft limit filter would clamp the sensor data to the 0 degrees limit for use by the remainder of the system.

The final filter is the default/override filter. This filter would decide how to handle faults reported by *Noisy World*. It will typically select some benign default value to be used if the sensor is malfunctioning. This is also where the sensor value would be overridden. This override could be an operator input to override the sensor, or to provide a estimate for a faulty sensor, or to provide known sensor vales during testing and debugging.

Therefore:

Select the processing necessary to provide a stable internal sensor representation.

Careful control of the view of the *Perfect World* provides the internals of the system with stable signal on which to base further processing.

Related Materials

Design Patterns for Avionics Control Systems, Doug Lea; March 1995
http://gee.cs.oswego.edu/dl/acs/acs/acs.html

Conclusion

The preceding is the core of *The Carousel* pattern language for simple embedded systems. Further extensions to this language should include patterns covering initialization, shutdown, processing optimizations (particularly replacing floating point and trigonometric calculations), error reporting, and the large area of user interfaces.

I would like to thank Dennis DeBruler who ably shepherded this paper from its initial ramblings into its current state. I hope to actually meet him some day.

Mark Bottomley

mark.bottomley@cdott.com

Computing Devices Canada
M/S 5215
3785 Richmond Road
Nepean, Ontario
K2H 5B7
Canada

1-613-596-7235