Be Alert and Alarmed: A Design for a Real-Time Uncommon Speed Alarm

Simon Thornton
Student No.: 4031377

October 18, 2004

1 Introduction

This document presents a design for an adaptive real-time uncommon speed alarm. The input to the program is the data produced by a laser range finder. This data is in the form of 360 distance readings taken at half-degree intervals. It is possible that some readings may not return a value. Throughout this document a set of readings from the range finder is referred to as a frame. In designing the program no assumptions were made about the placement or field of view of the range finder.

Using these input frames the program tracks objects that move within its field of view in order to determine their speed and direction. If this vector value is outside a certain range from the average vector in the object’s current location then an alarm is raised. This alarm indicates the location of the object, its measured vector and the expected vector in the location. This motion tracking must be done within the time constraints presented by the output of the range finder.

The following sections describe the details of the design, both from a logical and temporal perspective (sections 2 and 3 respectively). A high level overview of the system is initially given. The individual stages of the program are then described in more detail, as are the synchronisation objects. As part of the logical analysis additional constraints or parameters for the program are also outlined. The temporal analysis presents the behaviour of the system at run-time.

In designing the program primary consideration was given to modularity, the re-usability of basic components, the accuracy and consistency of the results and the scalability of concurrent sections. The rationale for giving primacy to these factors and an evaluation of the merits of the design based on other criteria is given in section 4.
2 Logical Description

2.1 Overview

The basic structure of the system is a pipeline architecture. This pipeline is split into four stages, three of which are continuously repeated as part of the main control loop. In order to identify and track objects the program establishes a reference frame containing the readings from the range finder when no moving objects are in view. This calibration is done during the Initialisation stage, which also sets up the data structures and initialises all the tasks. The reference frame is then used by the Identification stage, in combination with the current frame to produce a set of objects. These objects are then passed to the Correlation stage which looks for them in the subsequent frame in order to establish their current motion. These updated objects are then passed to the Integration stage which checks them against the average vectors for the object’s current location and raises an alarm if any unusual vectors are found. The entire process then begins again with objects being identified in what was the subsequent frame (and is now the current one). The latter of these three stages form the main control loop of the program. Each stage is discussed in greater detail below.

The execution described above means that the tracking of objects over more than one time step\(^1\) is implicit rather than explicit. This implicit tracking greatly simplifies the structure of the program as the internal representation of an object is not required to persist for more than one iteration of the main control loop. At the same time none of the information required by the program is lost, because all that is required to find an object’s vector is its position in two consecutive frames.

2.1.1 Static Architecture

The pipeline structure described above is illustrated in figure 1. The dotted arrows show the control flow at a conceptual level, with data being fed in the Identification stage and being output by the Integration stage.

The actual data flow within the system is also shown in figure 1 (the solid arrows). This differs from the usual pipeline architecture. It can be seen that each new frame from the laser range finder is actually used at both the Identification and Correlation stages. This “dual-input” is required because correlation (which utilises the current frame) is performed upon the set of objects identified in the previous frame. At the same time a new set of objects is identified in the current frame. This is possible because at an implementation level each of the three main stages is performed in parallel by a different task. In other words each of the three stages in the main control loop represent at least one task within the system.\(^2\)

---

\(^1\)Where a time step is the time between one sample frame and the next.

\(^2\)For this reason the words stage and task are sometimes used interchangeably in this document.
In addition to this concurrency between stages, the multiple data flows from the CurrentObjects data structure into the Correlation stage and out to the UpdatedObjects data structure indicate that concurrency is possible within this stage by having more than one correlation task running. This additional concurrency was included in the design because correlation is potentially the most computationally expensive part of the system. Parallelism within this stage therefore allows more accurate results to be calculated. At the same time the use of concurrency requires synchronisation between and within stages to ensure time constraints are met and that accuracy is consistent.

2.1.2 Data and Communication Structures

The description above does not describe the data structures shared between stages (CurrentObjects and UpdatedObjects) in any detail. In practice both of these data structures could be implemented as protected first-in-first-out queues. In combination with the synchronisation objects outlined below this is enough to effectively shift data from one stage to the next, whilst allowing all stages to be working in parallel.

The other data structure required by the program is the one used by the Integration stage to identify unusual speeds or directions. To do this the program must have a data structure that corresponds to the world being scanned by the range finder. In this case the program maintains an internal representation of the range finder’s field of view in the form of a Cartesian plane with the range finder at the origin. This plane is overlayed with a grid which divides the plane into squares. This representation is illustrated in figure 2. The dimensions of this grid determine the resolution of the representation, which in this case is the number of cm$^2$ per square. Each grid square contains the average vector for objects moving within that square and the Cartesian coordinates of the square’s lower left corner.
2.2 Initialisation Stage

This stage is responsible for initialising all the data structures. It is performed by the same task that executes the identification stage once the main loop of the program has been entered. During the Initialisation stage only one task is executing so no synchronisation is needed. The steps performed during this stage are shown below:

begin
  Initialise Tasks and Synchronisation Objects;
  Initialise Grid;
  Startup Range Finder;
  Calibrate Reference Frame;
  Get First Sample;
  Detect_Object_Set(Reference, Sample, CurrentObjects);
end

Of these steps two require further explanation. The first is the calibration of the reference frame, which is discussed below. The second is the initial identification of objects. This is done in order to fill the CurrentObjects queue with objects for the correlation tasks to track as soon as the next frame arrives from the range finder. Placing this step in the initialisation stage essentially completes a the first iteration of the first stage of the pipeline.

The initialisation of the grid may simply involve zeroing all the vectors or, if data from a previous session has been written to a file by the program it may involve reading in this data so that the grid starts with non-zero values.

2.2.1 Reference Frame

When the first frame is received from the laser range finder it is not known if it contains values that correspond to an object or not. In order to calibrate the reference frame correctly it is therefore necessary to average the values over a
number of frames, or to continue receiving frames until there is no movement detected between frames. Either approach could be used, though in this case an average over the first 2 seconds of data (10 frames) will be used as this has a bounded calibration time. The program is designed in such a way that the procedure used to calibrate the reference frame can be changed without affecting the rest of program. This would allow any algorithm from simply grabbing the first available frame, to tracking the differences between frames until there is no movement to be implemented.

2.3 Identification Stage

As described above the Identification stage uses the current frame and the reference frame to generate a set of objects. It is executed by the same task that executed the Initialisation stage. The steps for the stage are shown below:

```
while program_is_active loop
    Get_next_frame(Sample);

    --Synchronisation steps
    Abort Updates;  (1)
    TriggerWait(Sample);  (2)
    TriggerIntegrate;  (3)
    Reset Updates;

    --Indentification
    Detect_Object_Set(Reference, Sample, NewObjects);

    --Place new objects in correlation queue
    select
        Wait_till_empty(CurrentObjects);  (4)
        while not NewObjects.Is_Empty loop
            NewObjects.Dequeue(O);
            CurrentObjects.Enqueue(O);
        end loop;
    or
        delay until two frames arrived;
        Empty NewObjects;
    end select;

end loop;
```

An explanation of these steps is given below.
2.3.1 Synchronisation Steps

The Abort Updates call is used to force any correlation tasks waiting in the UpdateMonitor\(^3\) to exit the protected object without completing the current update round. The TriggerWait call then performs a rendezvous with all of the correlation tasks. At the same time the new sample frame is passed to each of the correlation tasks through the FrameMonitor. A rendezvous is used to ensure that all the correlation tasks have emptied their local queues into the shared UpdateObjects object. The next step triggers an iteration of the Integration task. Finally, the UpdateMonitor is reset to allow the correlation tasks to enter again. Each of these synchronisation steps is numbered to illustrate the corresponding steps in the stages described below.

2.3.2 Identification Step

This step is the main operation for this task. The program is designed in such a way that any algorithm for identifying moving objects can be ‘dropped’ into place, so long as it meets the interface. In this case a relatively simple algorithm is used. It simply identifies consecutive readings that differ from the reference frame by more than a threshold amount. These movement clusters are then classified as an object, so long as the number of consecutive readings is at least equal to a minimum length. An object identified in this way is then placed in the queue that was passed to the procedure. This process of identifying clusters of movement is completed by scanning across the entire frame.

Once the identification step has been completed the new objects must be copied from the local NewObjects queue into the shared CurrentObjects queue. To ensure that objects from different frames are not confused by the correlation tasks the identification task waits to receive a signal from each correlation task indicating that it knows the CurrentObjects queue is empty before copying the new objects across. The delay statement is used to ensure data consistency should a frame be lost.

2.4 Correlation Stage

The Correlation stage has the potential to be the most computationally expensive stage in the system. For this reason the program was designed to allow concurrency within the stage, reducing the number of objects each correlation task must deal with. Allowing for multiple tasks increases the complexity of the synchronisation with the identification task as the FrameMonitor and UpdateMonitor must now be aware of the number of correlation tasks to ensure that all tasks receive any communications. This information is provided to the synchronisation objects during initialisation.

The steps for the Correlation stage are as follows:

while program_is_active loop

\(^3\)These synchronisation objects are discussed below at section 2.6
-- Generate rough solution for each object --
while not CurrentObjects.Is_Empty loop
    CurrentObjects.Dequeue(O);
    Correlate(O, Status, Found);
    if Found then
        LocalObjects.Enqueue(O);
    end if;
end loop;

SignalID_Task; (4)

-- Use remaining time to refine solutions
select
    SampleWait(Status); (2)
    while not LocalObjects.Is_Empty loop
        LocalObjects.Dequeue(O);
        UpdatedObjects.Enqueue(O);
    end loop;
    then abort
    while True loop
        for each object in LocalObjects loop
            Refine(O, Status, Resolution);
        end
        Start_Update; (1)
        end loop;
    end select;
end loop;

2.4.1 Correlation Steps

To ensure that every object is tracked at least roughly through each frame two
different correlation steps are used. The first gives a rough estimate of the
new position and vector of the object. This algorithm should be capable of
calculating positions for large numbers of objects before the next frame arrives.
Again, the program is designed to allow any algorithm to be implemented at
this stage. The easiest way to do this rough correlation is to simply scan for the
object a certain number of readings either side of its original location, identify
the scan point that produces the lowest correlation value and, if this correlation
value is below a threshold, calculate the object’s new position and vector before
returning it as found. If the minimum correlation value is above the threshold
the object is returned as not found and discarded.

During this stage of correlation it is not possible to abandon object tracking
so it is important that this algorithm returns an answer quickly. If all of the
objects are not dealt with before the next frame arrives then a frame will be
dropped by the system.

In order to correlate an object a correlation task removes it from the Cur-
rentObjects queue and places it on a local queue if it is found in the new frame. Once all objects have been roughly correlated and are therefore stored in one of the correlation tasks local queues or discarded the correlation tasks signal the identification task to indicate that the CurrentObjects queue can be refilled. This concurrent removal of objects from the CurrentObjects queue is the reason that it must be a protected object.

### 2.4.2 Refinement

The second part of the correlation stage refines the results from the first part. It involves each correlation task looping over the objects in its local queue. Once every task has refined the solution in every object it has all the tasks commit to an update. At any point before all tasks have completed their refinement the current round can be abandoned by the arrival of a new frame. However, once all tasks begin an update they cannot be aborted until they have completed updating their local objects. This is achieved by placing the update code within the protected UpdateMonitor.

This commit-or-abort style refinement ensures that all the objects tracked during any given iteration are tracked to a consistent accuracy. This prevents any drift in measurements from occurring by having certain objects repeatedly correlated at a lesser accuracy in a given area.

In this way the rendezvous between the correlation and identification tasks can occur at three times. When a new frame arrives while a correlation task is refining an object’s tracking then the refinement is immediately abandoned by the `select..then..abort` statement, which corresponds to the TriggerWait call made by the identification. Alternatively a correlation task may be waiting in the UpdateMonitor, in which case it exits without updating its local objects. Thirdly, all objects may have committed to an update, in which case each will complete its update and abandon its refinement processing. Once the refinement stage has been completed for any of these reasons the updated objects are copied out of the task’s local queues into the queue shared with the Integration stage. This concurrent copying is the reason the UpdatedObjects queue must be protected.

Again the refinement algorithm used can be easily changed. In this case correlating at a higher resolution during each refinement stage can be used to provide ever more accurate results. This involves interpolating an objects position between range finder readings and then correlating at a certain number of spots along this interpolation.

### 2.5 Integration Stage

The integration stage is the most straightforward of the three stages in the main loop. It is takes the objects from the UpdatedObjects queue and checks them against the grid data structure to see if they have an unusual speed or direction. The object’s vector is then used to update the average vector for the objects location. The remaining grid squares are then updated so that
they will eventually decay to zero over some specified time frame (if no objects pass through them). Once this is done the task waits for a signal from the identification task before starting over. These steps are shown below:

```plaintext
while program_is_active loop
    WaitIntegrate; (3)
    for each object in UpdatedObjects loop
        check for alarm;
        modify grid;
        end;
    update grid;
end loop;
```

If the grid is large then it may take along time to update. If it is so big that it is impossible for one task to update it in a single timestep then additional tasks could be used, so long as the grid also implements ensures that each individual square is kept consistent. Alternatively, the task could be rewritten so that it only updates part of the grid each iteration. This would mean average values still decayed over time while a large grid still met timing constraints.

### 2.5.1 Raising The Alarm

The most important function of the Integration stage is to check the objects for unusual speeds. This is done by comparing the calculated vector of the object with the average vector in the grid square that the object is currently in. If these vectors differ in their headings by more than 45 degrees or 15 KM/H then an alarm will be raised.

### 2.6 Synchronisation Structures

Three objects are used to provide synchronisation between the tasks in the system. The FrameMonitor synchronises the correlation tasks with the identification tasks. The UpdateMonitor is used for synchronisation between correlation tasks and the IntergrateMonitor is used for synchronisation between the identification task and the integration task.

#### 2.6.1 Frame Monitor

The FrameMonitor includes a rendezvous and wait point. To implement these the object must know how many correlation tasks are running so it has a private variable that is set during the Initialisation stage. The rendezvous then holds the triggering task, in this case the identification task, until the queue length at the waiting entry is equal to the number of tasks.

This mechanism also allows each correlation task to update its sample frame to the most recent frame during the rendezvous. This is done by having the triggering task copy the frame to a private variable within the object before it
releases the the waiting tasks. As each waiting task leaves the object it takes a new copy of the object’s private variable, thus updating its frame.

The wait point for the identification task is used to ensure that all correlation tasks know the CurrentObjects queue is empty before it is filled again. It works in a similar way to a semaphore. Each of the correlation tasks increments a private counter within the object. When this counter equals the number of correlation tasks the waiting identification task is release, resetting the counter before leaving the object.

### 2.6.2 Update Monitor

The UpdateMonitor works to implement a commit-or-abort cooperation between correlation tasks. To ensure that a correlation task cannot be pulled out of it execution point during an update, the updating of the objects is performed within a protected object. Furthermore, to ensure that once one task begins updating all of them will update all tasks must be queued at an internal queue within the object before updating can begin. When all tasks are internally queued the identification task is unable to enter the object and abort the update. However, if not all tasks are ready to update those that are internally queued can be aborted.

This behaviour is implemented by means of a private counter and two flags within the object. The counter allows the object to know when all tasks are internally queued. Once this is the case the updating flag is set. This locks the object to the identification task, preventing the abort flag being set. If however, the update is to be aborted before all tasks are ready then the identification task enters and sets the abort flag. This locks the object to tasks that are not yet ready to update. It also releases all the internally queued tasks. These tasks check the reason for their release however and exit the object without updating. The internal routines of the object are shown below.

```haskell
entry Start_Update when not Updating and not Abort_Now is begin
   if Queue’Count = TaskCount - 1 then
      Updating := True;
   end if;
   requeue Queue;
end Start_Update;

entry Abort_Update when not Updating is begin
   Abort_Now := True;
end Abort_Update;

entry Queue when
```
Updating or Abort_Now is begin if not Abort_Now then Update Objects; end if; if Queue'Count = 0 then Updating := False; end if; end Queue;

procedure Reset is begin Abort_Now := False; end Reset;

procedure Set_Task_Count (I : in Integer) is begin TaskCount := I; end Set_Task_Count;

Placing the update procedure within a protected object like this has the disadvantage of serialising the updates of the correlation tasks. This is not much of a problem however, as updating an object only involves copying a refined result to the definite result variable within an object. When compared to the computational cost of correlating at a resolution beyond that of the range finder the serialisation of the update mechanism is a relatively insignificant cost.

2.6.3 Integration Monitor

The IntegrateMonitor is a simple wait point for the integration task. When it is triggered by the identification task it iterates once and waits for the next trigger to arrive. The trigger is reset by the integration task as it exits the object meaning that it can miss its deadline. This will cause problems with the decaying averages, however all objects will still be dealt with.

2.7 Additional Specifications/Tuning Parameters

This section lists some additional constraints and parameters upon the system that need to be defined, either arbitrarily to provide the working specification for the system or based on empirical testing.

- The number of objects trackable at once - This value is determined at a logically by the size of the object queues and temporally by the rough correlation time. Exceeding the first will result in an error, while exceeding the latter will result in a dropped frame. A queue size of 30 is appropriate.
The size of objects trackable - This value will be determined logically by the array used to store a copy of an objects distance values. Larger objects will take longer to correlate and will therefore also pose a temporal constraint. A maximum size of 30 is appropriate.

The distance reading noise threshold and minimum object length - These values are used in identifying objects in a frame. If they are set too high then objects may be missed, while too low and ‘ghost’ objects will appear. A noise value of 5 with a minimum length of 2 are good starting values.

The alarm thresholds - These are the maximum divergences from average that are allowed. Reasonable values would seem to be 45 degrees heading differences and 15 KM/H speed differences.

The number of tasks - The minimum number of tasks is three. The system has been designed to scale up with the number of available processors.

The grid resolution - This is determined by the dimensions of the grid. It directly relates to the time taken to integrate a set of objects and perform an update of all grid squares. A 50x50 grid gives a reasonable resolution over a 200m² plane.

3 Temporal Description

To some extent the execution of the system has been described above. However, this section graphically illustrates the run-time behaviour of the system.

3.1 Temporal Behaviour

Figure 3 illustrates the run-time behaviour of the system. The four columns labelled Id, Cor1, Cor2 and Int represent the identification, first and second correlation and integration tasks respectively. The circles labelled Frame, Update and Integrate represent the synchronisation objects. Each column of circles represents the same object, they are just redrawn for clarity. The part of the figure above the dotted line represents the Initialisation stage. It illustrates that the tasks other than the identification task are suspended immediately after their initialisation.

The part of figure below the line represents a single iteration of the main control loop. Each arrow represents a call to a synchronisation object with the numbered arrows corresponding to the numbered calls found in section 2.3. In this particular case only a single refinement is made by the correlation tasks to the initial rough correlation before a new frame arrives (the arrival is implicit at the bottom when the identification task begins executing again).

The figure shows how the synchronisation objects are used to coordinate the various stages of the pipeline even as they execute concurrently. The short execution time of the integration task is explained by the fact that on this particular iteration there are no objects for it to integrate into the grid.
Figure 3: The run time execution of the system using two correlation tasks. The flow of time is down the page. The part above the dotted line is the Initialisation stage. Running tasks are a square block. Suspended tasks are a dotted line. Synchronisation objects are circles. Each column corresponds to a single object, the multiple circles are just for clarity. The lines indicate calls to synchronisation objects. The flow of data is not shown.
By contrast, the filling of the CurrentObjects queue during the initialisation stage means that the correlation tasks have a full queue to work with. The figure also shows that once the correlation tasks have been awoken the only time they are suspended is when waiting to commit to an update.

4 Design Rationale

A number of factors were identified in section 1 as the driving forces behind the choices made in designing the system. These were: modularity, reusability of basic components, accuracy and consistency of results and scalability of concurrent sections.

The reason for wanting to create a modular design, where important algorithms could be changed without affecting the execution of the program, is that pattern recognition is an entire field of research in itself. This means that there will be a number of ways of identifying and tracking objects, each with differing performance and accuracy. The design given above allows an algorithm to be implemented to meet a specified interface and then inserted into the program with the system enforcing the temporal constraints.

Reusable basic components was also considered important because representation of entities such as Cartesian points and vectors and the use of a grid are not unique to this program. These basic components are included in such a way that they do not need any information specific to the system. For example a generic grid package could be used with an implementation specific instantiation providing all the information via the grid element package.

In any system the accuracy of the results is always important, though in this case achieving maximal accuracy conflicts with the requirement that an alarm be raised at the earliest possible time. The decision was made not to check for an alarm before the most accurate results had been obtained. This decision was taken in order to separate the functions of the three stages of the main control loop cleanly. The result is a clearer design.

Perhaps the decision that had the most impact on the final design was the decision to only track any given object for a single timestep. This means that data only needs to travel in one direction in the system, so long as a “dual input” pipeline was used. As noted above this implicit tracking simplifies the architecture without discarding any of the information required by the system.

The result is that the synchronisation points are clearly defined within the system. The streamlined architecture results in a design that can be scaled from 3 to N tasks, depending upon the number of available processors, fulfilling the primary concern of scalability, though in this case the scalability is achieved within the correlation stage, not between stages. None-the-less, the computational cost of correlation makes this precisely the point at which maximal concurrency is required.

The other important decision was the making of no assumptions about the field of view in the range finder. Whilst in this case the test data all came from a specific run of the range finder none of the information known about the
area the range finder was viewing was included in the design. This is important because it means that the system can be used in any environment where the detection of unusual speed or direction is required.

Another criterion that could be applied to an evaluation of the system is the expandability of the design. In this case cleanly defined and separated stages of the pipeline mean that design could be expanded, or changed, by adding another stage or replacing an existing one. This is possible because the communication between stages is via shared objects and is synchronised.

The reliability of the design to some extent hinges on the size of the grid. If the entire grid cannot be updated in a time step then the program can never meet its deadlines. Section 2.5 mentioned ways of addressing this problem. The other important part of the design is rough correlation. If this takes more than a time step all the time the program cannot meet its deadline and will begin dropping frames. If it is less than this on average and only occasionally misses the deadline due to a spike in the number of objects identified then it is possible to miss a deadline, have less time available for the next iteration and eventually catch up with the range finder.

To briefly summarise: the key points of the design are the explicit tracking of objects over one time step and the implicit tracking over longer intervals and the generality of the solution because no assumptions from the test data were used. The first point leads to a clear, simple design that can be extended or changed easily and uses maximal concurrency, while the second point leads to a design that can be used to track unusual motion in any environment.