# Navico Pre-Interview Test

Please answer all questions to the best of your ability and provide as much detail as possible. We will discuss your answers in more detail if you are selected for a face to face interview.

**Candidate Name: Reza Amani**

## General

1. You have been asked to debug a random crash in a very large program with millions of lines of code. Initial analysis indicates that the crash occurs on a different line of code each time.

How would you debug this problem? Describe any tools that you may have used to aid with this type of investigation in the past

### Answer

|  |
| --- |
| I would try these:   1. Unexpected interrupt: Is there any potential source of interrupt whose ISR has not been defined? I would define a null ISR (e.g. log an error only) for all unexpected interrupts. 2. Stack overflow: I may use one of stack check programs. Also, if dynamic memory allocation is utilised, I would have a look at heap size and any possibility of memory leakage. 3. Hardware Faults: It can be a hardware issue, like a surge noise in system or a spike on sensitive pins (RESET, VCC, etc.) 4. Repetitive patterns: I would search for common events in different crash times to find a clue. For example, if every time that the CPU crashes, something especial in hardware happens, or always it crashes when we are in a specific state in SW, or crash always occurs after receiving a message. Using an event logger helps to find any repetitive pattern. 5. Backtrace: If the CPU ends up in a HardFault error, I would use a realtime debugger with backtrace capability (like GDB). I would put a BreakPoint in the hardfault handler, and then using the backtrace command I would be able to see the history of program flow in the stack. At this stage, if the stack is corrupted, a stack overflow is the problem. If no, I can see what were the last modules that CPU has been executing before the crash. |

1. What is the most inventive or innovative thing you’ve done (in a SW development context)? **Pick one** and describe what problem were you trying to solve and the solution you came up with?

### Answer

|  |
| --- |
| Last year, we had a timing problem. We had to respond to an event in less than 7us, while the primary implementation took 400 us and after full optimisation, 200us.  I used an oscope and some GPIO’s as flags to determine which part of the code consumed most of the time; and it was a couple of formulas based on the inputs that were captured once the event was received.  My solution for the problem was calculating all possible outputs and saving them in an LUT. Then, once the event was triggered, we just used the inputs as indexes for the LUT to catch the correct output.  That solved the problem. But, in a later stage of the projects, we ran out of the memory. Half of the memory was filled with the LUT data and this made me think about an alternative approach.  Further investigation showed that the CPU was very weak in terms of division and floating point operations. The immediate idea was turning our floating-point algorithm to a fixed point one. Based on my experience in DSP projects, a new implementation of the algorithm was provided without any division and other floating point operation. The output of this new one along with a small modification in hardware was good enough to save the day. |

## C++ Programming

1. Which design patterns have you used in the past? Pick **one** and provide a **simple** C++ example of how this is used.

### Answer

|  |
| --- |
| Strategy (for a hobby project), Singleton, FSM (if state machine is counted as a design pattern. Using an FSM platform class including events, entry routine and exit routines) and Publisher-Subscriber. |
| Observer / publisher-Subscriber:  In this design pattern, any publisher may report (i.e. publish) events and then those subscribers (observers) that have subscribed to the published event will be notified and may execute a callback to react.  The main idea here is that publishers and subscribers don’t need to know each other (apart from a simple interface) and this helps reach better abstraction. It doesn’t matter how many observers have subscribed to a particular event. And even we can add or remove some observers in runtime. Because of independence between publishers and subscribers, each one can be reused independently.  Simple example in pseudo-code:  In the publisher side:  Class Publisher  {  List< Subscriber\*> subscriber\_list;  Public:  Virtual void Register(Subscriber\*); //adds the subscriber to the list of subscribers  Virtual void UnRegister(Subscriber\*); //removes the argument from the above list  Virtual void Update(); //calls the update function of all subscribers.  //If we have more than one event, the above functions should have another argument to determine the intended event  …  }  //Now any publisher inherits from the above class:  Class PublisherModule1 : public Publisher  {  ….  }  In the subscriber side:  Class Subscriber  {  Public:  Virtual void Update(Publisher\*); //reaction to the event. Having the publisher as an argument is needed only if more than one publisher can send the event to this subscriber (in order to distinguish between them)  …  }  Class SubscriberModule1: public Subscriber  {  …  }  In our project with C language, we implemented in another way; We had a 3rd module responsible for handling all stuff regarding events and subscriptions. It included an enum of all events and a list of callback functions from subscribers, Register functions that subscriber needed to link their callbacks to events, and finally ReportEvent functions that the publishers used to trigger the events. When an event was triggered, this coordinator module was responsible for calling the callback functions of associated subscribers. |

1. Imagine you are working in the team responsible for rendering charts on Navico’s products. A chart represents an area on a map (similar to Google maps). For the purpose of this exercise assume that a map consists of multiple charts drawn on top of each other, similar to the picture below.



Assume a chart has the following attributes.

* Position in space and size is specified by its top left coordinate (x1,y1) and its bottom right coordinate (x2,y2).
* Colour of the chart is specified in RGB, with each value between 0 – 255 (e.g. R=255, G=0, B=0 would imply a red chart)

1. Write a class called **Chart** that represents the above.
2. Write a class called **View** that can contain a maximum of 2 charts at a time
3. Implement a method **DoChartsOverlap()** in the **View** class which checks if the charts overlap.
4. Implement a method **GetColour(X,Y)** in the **View** class that will return the RGB colour of a given coordinate. If two charts overlap the colour of the point should be the average of the two colours.

You may assume the X and Y axis starts at 0 and has a maximum value of 100 for all of the above tasks.

### Answer

|  |
| --- |
| #define maxP 100  #define maxRGB 255  struct point  {  uint x,y;  };  struct color  {  uint R,G,B;  };  class Chart  {  public:  color rgb;  point p1, p2;  Chart(point \_p1, point \_p2, color \_rgb)  {  assert(\_p1.y<=maxP && \_p2.y<=maxP && \_rgb.R<=maxRGB && \_rgb.G<=maxRGB && \_rgb.B<=maxRGB);  assert(\_p1.x<\_p2.x && \_p1.y<\_p2.y);  p1=\_p1;  p2=\_p2;  rgb=\_rgb;  }  bool IsPointIn(point p)  { //returns true if the point falls in the chart  return p.x>p1.x && p.x<p2.x && p.y>p1.y && p.y<p2.y;  }  point GetTopRight()  { //returns the 3rd corner; the top right corner  point p={p2.x, p1.y};  return p;  }  point GetBottomLeft()  { //returns the 4th corner; the bottom left corner  point p={p1.x, p2.y};  return p;  }  };  class view  {  private:  Chart\* pCharts[2]={NULL,NULL};  uint NumCharts=0;  public:  bool AddChart(Chart\* pCh)  { //returns true if we have room to accept a new chart  if(NumCharts>=2)  return false;  else  {  pCharts[NumCharts++] = pCh;  return true;  }  }  bool DoChartsOverlap()  {  if(NumCharts!=2)  return false;  else if( pCharts[0]->IsPointIn(pCharts[1]->p1) || pCharts[0]->IsPointIn(pCharts[1]->p2) || \  pCharts[0]->IsPointIn(pCharts[1]->GetTopRight()) || pCharts[0]->IsPointIn(pCharts[1]->GetBottomLeft()) )  return true; //one of 4 corners of chart1 falls in chart0; so they overlap  else if( pCharts[1]->IsPointIn(pCharts[0]->p1) || pCharts[1]->IsPointIn(pCharts[0]->p2) || \  pCharts[1]->IsPointIn(pCharts[0]->GetTopRight()) || pCharts[1]->IsPointIn(pCharts[0]->GetBottomLeft()) )  return true; //one of 4 corners of chart0 falls in chart1; so they overlap  else  return false; //none of corners of charts falls in the other chart. So, they don't overlap  }  color GetColor(point p)  {  color white={0,0,0}; //Assumed that in the absence of both charts, the color is white  color average = {(pCharts[0]->rgb.R+pCharts[1]->rgb.R)/2, \  (pCharts[0]->rgb.G+pCharts[1]->rgb.G)/2, \  (pCharts[0]->rgb.B+pCharts[1]->rgb.B)/2 };  if(pCharts[0]->IsPointIn(p))  return pCharts[1]->IsPointIn(p) ? average : pCharts[0]->rgb;  else  return pCharts[1]->IsPointIn(p) ? pCharts[1]->rgb : white;  }  }; |

1. How would you test your classes? Write some pseudocode for the unit tests you might write.

### Answer

|  |
| --- |
| TEST\_CHART\_ISPOINTIN  {  Chart ch( (point){10,10}, (point){30,40}, (color){0,0,0} );  Point p\_border={10,20}, p\_in={20,20}, p\_out={5,20};  Assert( ! Ch.IsPointIn(p\_border) );  Assert( Ch.IsPointIn(p\_in) );  Assert( ! Ch.IsPointIn(p\_out) );  }  TEST\_CHART\_GetTopRight  {  Chart ch( (point){10,10}, (point){30,40}, (color){0,0,0} );  Assert( ch. GetTopRight()==(point){30,10} );  }  TEST\_CHART\_GetBottomLeft  {  Chart ch( (point){10,10}, (point){30,40}, (color){0,0,0} );  Assert( ch. GetBottomLeft()==(point){10,40} );  }  TEST\_VIEW\_ADD  {  View v;  Chart ch1( (point){0,0} , (point){10,10} , (color){ 10,10,10} );  Chart ch2( (point){5,5} , (point){15,15} , (color){ 20,0,0} );  Assert( v. NumCharts == 0 );  Assert( v. AddChart(ch1) );  Assert( v. NumCharts == 1 );  Assert ( v. AddChart(ch2) );  Assert( v. NumCharts == 2 );  Assert ( ! v. AddChart(ch2) );  }  TEST\_VIEW\_DOOVERLAP  {  View v;  Chart ch1( (point){0,0} , (point){10,10} , (color){ 10,10,10} );  Chart ch2( (point){5,5} , (point){15,15} , (color){ 20,0,0} );  Assert( ! v.DoChartsOverlap() );  v. AddChart(ch1);  Assert( ! v.DoChartsOverlap() );  v. AddChart(ch2);  Assert( v.DoChartsOverlap() );  }  TEST\_VIEW\_DONTOVERLAP  {  View v;  Chart ch1( (point){0,0} , (point){10,10} , (color){ 10,10,10} );  Chart ch2( (point){15,15} , (point){25,25} , (color){ 20,0,0} );  v. AddChart(ch1);  v. AddChart(ch2);  Assert( ! v.DoChartsOverlap() );  }  TEST\_VIEW\_GETCOLOR  {  View v;  Chart ch1( (point){0,0} , (point){10,10} , (color){ 10,10,10} );  Chart ch2( (point){5,5} , (point){15,15} , (color){ 20,0,0} );  v. AddChart(ch1);  v. AddChart(ch2);  assert( v. GetColor ( (point){50,50} == (point){0,0,0} );  assert( v. GetColor ( (point){15,15} == (point){0,0,0} ); //assuming that borders do not belong to the charts  assert( v. GetColor ( (point){1,1} == (point){10,10,10});  assert( v. GetColor ( (point){11,11} == (point){20,0,0});  assert( v. GetColor ( (point){6,6} == (point){15,5,5});  } |