DEBUGGING USING COMPLETE SYSTEM SIMULATION

BY ALAN HARRY

Embedded software developers can now escape from the constraints imposed by the availability of the target hardware. Using an electronic spirit level as an example, Alan shows us how to use the Crossware 8051 Virtual Workshop to simulate a complete target system.

In "Developing a Virtual Hardware Device"[1] Michael Smith gave a breakdown of the time spent on the various activities of the software development process. He allotted 215% to "Waiting for Hardware". We think this is an underestimate and that if you can proceed with the software development in the absence of hardware, time-to-market can be reduced in some cases by an order of magnitude.

As you will see from this article, it is now possible to develop and debug a complete embedded program for the 8051 microcontroller or one of its variants without any target hardware at all. The benefits extend beyond time-to-market to encompass the complete software lifecycle.

At the beginning of this year we finished the development of a medical product. This used high power lasers to treat the patients skin and so had to be carried out in strict accordance with FDA guidelines. The development was completed in a total of 4 months and the software, which was designed using Real-Time UML[2], was developed and verified almost entirely using simulation.

The software was ready long before the hardware and when the prototypes were handed over to the client we were able to continue to provide support and to fully test our enhancements before e-mailing them to the client. Maybe one day we will be able to tell you more about this project but for the time being our client wishes to continue to enjoy the perception that its new product was developed by its in-house team.

A few years ago it was a different story. We developed an instrument, shown in photo 1, to measure the slope and height of a flowing liquid. This uses three lasers and three position sensitive detectors, which measure the displacement of the reflected beams.

The instrument is positioned above the liquid, mounted on a frame that is not necessarily horizontal. Therefore the instrument needs to know its absolute orientation in space so that it can compensate for its own tilt. We embedded two ceramic tilt sensors inside the instrument to measure this tilt, calibrating them throughout their range on an adjustable tilt stage.

The height of the liquid is displayed on a graphic LCD and an image of a bubble is used to indicate the 2-axis slope. The operator is required to adjust the screws supporting the tank that contains the liquid until the bubble is in the centre of a circle printed on the display and until the height offset is zero. Using the image of a bubble made it immediately familiar to the operator who previously used a spirit level.

The software development process was not easy. We had to have a special board made so that we could get started with the display. In order to move our bubble around without unacceptable delay, we had to create six images and page the correct one into view at the right location by manipulating the graphics origin.

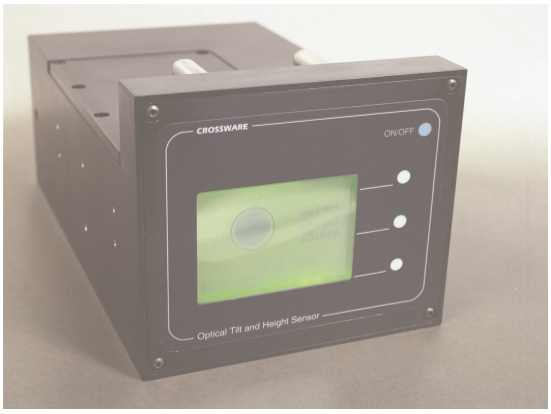

Photo 1. The Crossware Optical Tilt Sensor measures the slope and height of the surface of a flowing liquid. Embedded tilt sensors allow the instrument to monitor its own orientation in space.

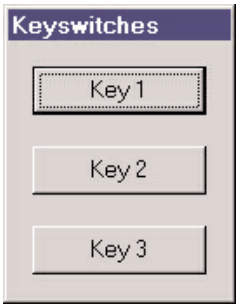

Photo 2. The three keyswitches will be positioned immediately to the right of the display. Legends on the display will change as the functions of the keyswitches change.

This was difficult to get right when our only guide was what we could or could not see on the display.

We had to wait for the final board before we could test other features and we could not carry out any full testing until the complete mechanical unit, a precision assembly which took a long time to make, was finished. We then found ourselves on the critical path with everyone else waiting for us to sort out our problems.

We recently revisited this project to see how we would develop the software today and so that we could provide a real life example for users of our development tools.

COMPLETE SYSTEM SIMULATION

The software that we use to simulate complete target systems is our 8051 Virtual Workshop. This simulates the 8051 instruction set, timer/counters, UART, interrupts, etc. and it also has an interface that allows it to be extended. This extension interface was originally implemented to allow us to rapidly add support for different 8051 variants. We soon realised how powerful it would become if we developed it further and made it easy for users to add their own extensions.

An extension takes the form of a DLL which supports some or all of the interface calls. It is given a special filename so that the Virtual Workshop will recognise it as an extension. The interface is a set of C function calls and so any DLL written in C can be an extension. However, in our own extensions we create a CExtensionState C++ object and immediately convert the C call into a CExtensionState function call.

By writing the DLL in a particular way and using Microsoft Visual C++ to build it, we can create an extension that integrates seamlessly with the Virtual Workshop and the rest of our development environment. We can add dialog boxes, windows and menu items using the Microsoft graphical editor and Class Wizard. We can also support the Virtual Workshop's Capture State command allowing the complete state of the target system to be captured and later restored.

Any number of extensions can be added. The Virtual Workshop looks for esim0.dll, esim1.dll, esim2.dll, etc. and loads each in turn as it finds them. This allows an extension to be developed specific to a particular peripheral. The same extension can then be re-used elsewhere if the same peripheral is being used in a different target system.

Each extension will receive calls in turn from the Virtual Workshop and it can handle the ones that it chooses. It is also possible for extensions to communicate with each other and we will describe how this can be done using named pipes.

LISTING ONE. *The Virtual Workshop can detect a falling edge on a port pin and if appropriate generate an interrupt.*

```
#define P2 0XA0
#define P3 0XB0

#define KEY1PIN 0X02 // P2.1
#define KEY2PIN 0X04 // P2.2
#define KEY3PIN 0X08 // P2.3

void CExtensionState::GetPortPins(BYTE nPortAddress, BYTE* pnPins,
BYTE* pnHandledPins, BOOL bSimulating)
{
  switch (nPortAddress)
  {
    case P2:
      if (m_pKeys->IsKey1Pressed()) // interrogate the dialog box
      {
        *pnPins &= ~KEY1PIN;                                    // clear pin
        *pnHandledPins |= KEY1PIN; // pin handled
      }
      else
      {
        *pnPins |= KEY1PIN;       // set pin
        *pnHandledPins |= KEY1PIN; // pin handled
      }
      if (m_pKeys->IsKey2Pressed()) // interrogate the dialog box
      {
        // clear appropriate pin and trigger EX0 with a falling edge
        *pnPins &= ~KEY2PIN;                                    // clear pin
        *pnHandledPins |= KEY2PIN; // pin handled
      }
      else
      {
        *pnPins |= KEY2PIN;       // set pin
        *pnHandledPins |= KEY2PIN; // pin handled
      }
      if (m_pKeys->IsKey3Pressed()) // interrogate the dialog box
      {
        *pnPins &= ~KEY3PIN;                                    // clear pin
        *pnHandledPins |= KEY3PIN; // pin handled
      }
      else
      {
        *pnPins |= KEY3PIN;       // set pin
        *pnHandledPins |= KEY3PIN; // pin handled
      }
      break;
    case P3:
      // trigger an interrupt if any key is pressed
      if (m_pKeys->IsKey1Pressed() || m_pKeys->IsKey2Pressed() ||
m_pKeys->IsKey3Pressed())
      {
        // P3.2 goes low for external interrupt 0
        *pnPins &= ~0X04;
// clear P3.2
        *pnHandledPins |= 0X04;    // P3.2 handled
      }
      else
      {
        // P3.2 high
        *pnPins |= 0X04;                                        // set P3.2
        *pnHandledPins |= 0X04;    // P3.2 handled
      }
      break;
  }
}
```

LISTING TWO. *The developer can set breakpoints in and single step through interrupt routines just as for normal routines.*

```
void _interrupt IVN_INTERRUPT0 _using 1 KeyPress()
{
  unsigned char i;
  unsigned char nKeyPressed;
  unsigned char nKeyMask;
  unsigned char nKeyMaskCheck;
  _ie0 = 0;                  // clear the interrupt flag
  nKeyMask = _p2;            // read port 2
  ResetWatchDog();
  for (i = 0; i < 10; i++);  // delay
  // read port 2 a second time to debounce the keys
   nKeyMaskCheck = _p2;
  if (nKeyMask == nKeyMaskCheck)
  {
    // The same data was read both times,
    // so assume valid keypress
    switch (nKeyMask)
    {
      case 253:      // 11111101
        KeyOneResponse();
        break;
      case 251:      // 11111011
        KeyTwoResponse();
        break;
      case 247:      // 11110111
        KeyThreeResponse();
        break;
    }
  }
}
```

A VIRTUAL ELECTRONIC SPIRIT LEVEL

So to return to our objective, we are going to simulate a complete sub-set of our measurement instrument. For the purposes of this example, we will leave out the lasers and the position sensitive detectors. We will simulate the display, A/D converter, tilt sensors, battery and keyswitches to create a virtual electronic spirit level.

When all of this is finished we will have a set of component programs all working together as shown in figure 1. We will briefly describe all of these components in turn before moving to the details of our target board specific extensions.

The Embedded Development Studio (estudio.exe) is the development environment that provides project management and editing facilities and allows you to compile and assemble your source code. Whenever you select 8051 as the target microcontroller it will load and initialise the 8051 Virtual Workshop (sim.dll).

There is a lot of interaction between the Embedded Development Studio and the Virtual

Workshop. The Embedded Development Studio tells the Virtual Workshop all about the target program and source level break points. The Virtual Workshop places additional windows, menus and toolbars in the Embedded Development Studio environment.

When the 8051 program is ready to run, the user selects an appropriate command such as Go or StepInto and the Virtual Workshop starts to do its work. It asks the Embedded Development Studio where it should look for extensions and it loads and initialises any that it finds. It loads the 8051 program and starts to simulate it instruction by instruction. At this point all DLLs are running and all are getting calls from the Virtual Workshop as the simulation proceeds.

Extension esim.dll provides support for the extra features provided by the Dallas DS2250. This includes a watchdog timer, additional interrupts, banked ram, etc. It comes as part of the Virtual Workshop package and is automatically selected when the DS2250 variant is chosen in the Embedded Development Studio.

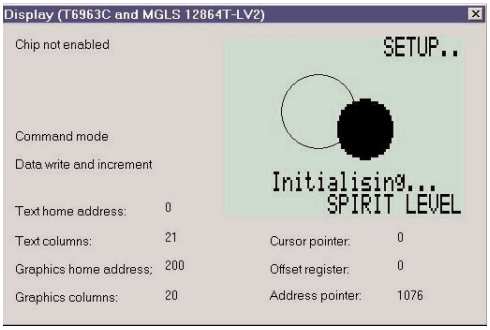

Photo 3. The image of the LCD and the status of the display driver attributes allow the developer to easily see if the embedded program is functioning as intended.

Then we have the four custom extensions esim0.dll, esim1.dll, esim2.dll and esim3.dll which we will develop specifically for our target system.

Finally mfc42.dll contains the Microsoft Foundation Classes - a comprehensive C++ interface to Microsoft Windows. It is important to realise that this DLL is being used by all components. Without this link it would not be possible for the components to integrate graphically with each other in such a seamless way.

It is worth mentioning at this point that when you are developing a Virtual Workshop extension you can run it in the Microsoft debugging environment. When you first run it from the Start Debug menu, you will be asked what EXE program your DLL is associated with. You will specify estudio.exe and the whole Embedded Development Studio environment will fire up. You can set breakpoints in your DLL, and execution will halt whenever they are reached. You can then single step though it and observe its behaviour.

LISTING THREE. *Writing to an external address at or above 8000 hex. will access the display data bus. The HandleCommand routine does the hard work.*

```
BOOL CExtensionState::SetXDataMemory(int nAddress, BYTE nValue, BOOL bSimulating)
{
  if (m_bChipEnabled && nAddress >= 0X8000)
  {
    if (m_bCommandMode)
    {
      // program is writing a command byte
      m_nCommand = nValue;
      HandleCommand();
    }
    else
    {
      // program is writing a data byte
      if (m_bDataAutoWrite)
      {
        // place the byte in memory and increment the memory pointer
        m_Memory[m_nAddressPointer++] = nValue;
        if (g_pDisplayDlg)
        {
          // show the updated memory pointer in the dialog box
          g_pDisplayDlg->SetAddressPointer(m_nAddressPointer);
        }
      }
      else
      {
        // keep track of the lst two data bytes for use
        // by the next command
        m_nData[1] = m_nData[0];
        m_nData[0] = nValue;
      }
    }
    m_nBusy = 4;   // time 4 micro-second busy period
    // Tell the Virtual Workshop that this extension has
    // handled SetXDataMemory by returning TRUE
    return TRUE;
  }
  return FALSE;
}
```

FOUR CUSTOM EXTENSIONS

We will now describe our custom extensions in turn, starting with the simplest esim3.dll and working our way backwards to esim0.dll.

The Virtual Workshop comes with an AppWizard. This program interacts with the Microsoft environment so that when you create a new Microsoft C++ project you have the choice to create a Crossware 8051 Virtual Workshop Extension. When you do, the source code for a complete extension will be created, immediately ready for you to customise and build.

To customise esim3.dll, we will create a modeless dialog box containing three buttons (photo 2). This is done using the normal Microsoft graphical tools with the outline code and variables being generated by the Microsoft Class Wizard. We need to make it a modeless dialog box by adding the Create call to the class constructor, remembering also to make its style visible. It all takes about 10 minutes.

We then need to modify the CExtensionState class by adding code to create the dialog box object and to interrogate its buttons whenever CExtensionState::GetPortPins is called. And that is essentially it. There are some cosmetic features you might like to add such as saving the position of the dialog box so that you can restore it to the same place on your screen when the extension is next started.

In total, it takes about 30 minutes to create a set of virtual keyswitches.

The code that we need to add to GetPortPins depends of course on the electronic circuit that we are simulating. The circuit is shown in figure 2. This shows each keyswitch connected to its own microcontroller port pin and all of them also con-

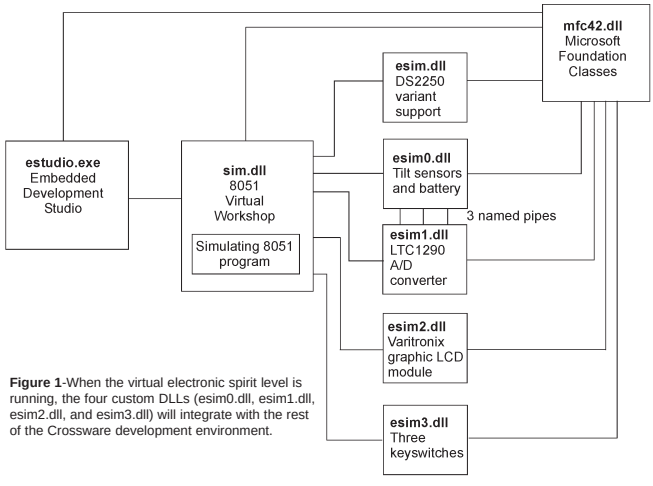

Figure 1-When the virtual electronic spirit level is running, the four custom DLLs (esim0.dll, esim1.dll, esim2.dll, and esim3.dll) will integrate with the rest of the Crossware development environment.

nected to INT0. So pressing a keyswitch causes an interrupt and the 8051 program can respond to this interrupt and interrogate port 2 to determine which keyswitch caused it.

GetPortPins is called repeatedly after each instruction is simulated. This allows the Virtual Workshop to be sensitive to falling edge, rising edge and level sensitive external interrupts. The extension therefore only needs to apply the correct levels to the pins that it controls. It does this by setting or clearing appropriate bits of *pnPins and telling the Virtual Workshop that it has set or cleared a particular bit by setting the corresponding bit *pnHandledPins.

Listing 1 shows the complete GetPortPins function and listing 2 shows the code of the 8051 interrupt function that reads the keys.

GRAPHIC LCD DISPLAY MODULE

Now we will turn our attention to the extension for the LCD display. You can see from the circuit diagram that the data bus of the display is connected to port 0 of the microcontroller and the /WR and /RD pins of the display and microcontroller are connected together. The microcontroller will therefore write to or read from the display whenever it writes to or reads from off-chip external data memory. For our DS2250, this is whenever the xdata address is above 8000 hex. We will therefore use CExtensionState::SetXDataMemory to determine if the display is being written to, and CExtensionState::GetXDataMemory to determine if the display is being read.

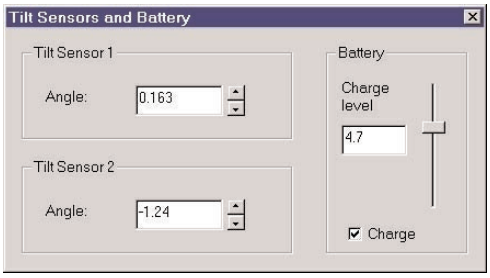

Photo 5. Three separate threads, one for each sensor and one for the battery, service named pipes so that the A/D converter can retrieve suitably scaled values from this dialog box.

You can also see that the chip enable of the display is connected to P2.5. We will use CExtensionState::SetPortPins to keep track of this pin. We can then ignore all reads and writes unless the display is enabled. Similarly we will monitor P2.6 to determine whether the display is in command mode or data mode. Listing 3 shows code for SetXDataMemory. HandleCommand does all of the interpretation of the display driver commands and is partly shown in listing 4.

You will notice variable g_pDisplayDlg. This points to a dialog box (photo 3) which displays the LCD and its attributes. As with the keyswitches, this dialog box and associated code can be quickly created using the Microsoft graphics editor and Class Wizard.

However, since we are displaying graphics in the dialog box we have to create a CWnd object and subclass it to a placeholder in the dialog box. Our CWnd object will then get all of the Windows messages that the placeholder would have received and it can draw an image of the display into the placeholders window area. Sub-classing can be easily done with a single call to the MFC function SubclassDlgItem.

You can see in photo 3 that as well as an image of the display, we are also showing the attributes and state of the display driver chip. This makes development of our 8051 program much easier because we can directly see if it is doing the things that we expect it to be doing.

We also have to take account of timing. The display driver requires 4 micro-seconds to process a byte. Our 8051 program will poll the displays status byte to determine when

LISTING FOUR. *The data bytes have already been received when the command byte has been written. These have been stored and can be used if the command needs them.*

```
void CExtensionState::HandleCommand()
{
   CString strCommand;
   switch (m_nCommand & 0XF0)
   {
      case CONTROL_WORD_SET:
         switch (m_nCommand & 0X0F)
         {
            case TEXT_HOME_ADDRESS_SET:
               strCommand = "Text home address set";
               m_nTextHomeAddress = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetTextHomeAddress(m_nTextHomeAddress);
               break;
            case TEXT_AREA_SET:
               strCommand = "Text area set";
               m_nTextArea = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetTextArea(m_nTextArea);
               break;
            case GRAPHICS_HOME_ADDRESS_SET:
               strCommand = "Graphics home address set";
               m_nGraphicsHomeAddress = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetGraphicsHomeAddress(m_nGraphicsHomeAddress);
               break;
            case GRAPHICS_AREA_SET:
               strCommand = "Graphics area set";
               m_nGraphicsArea = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetGraphicsArea(m_nGraphicsArea);
               break;
            default:
               strCommand = "Invalid command";
               break;
         }
         break;
      case DATA_READ_WRITE:
         switch (m_nCommand & 0X0F)
         {
            case DATA_WRITE_AND_INCREMENT:
               strCommand = "Data write and increment";
               m_Memory[m_nAddressPointer++] = m_nData[0];
               if (g_pDisplayDlg)
               {
                  g_pDisplayDlg->UpdateDisplay(m_Memory, m_nAddressPointer - 1);
                  g_pDisplayDlg->SetAddressPointer(m_nAddressPointer);
               }
               break;
            case DATA_WRITE_AND_DECREMENT:
               strCommand = "Data write and decrement";
               m_Memory[m_nAddressPointer—] = m_nData[0];
               if (g_pDisplayDlg)
               {
                  g_pDisplayDlg->UpdateDisplay(m_Memory, m_nAddressPointer + 1);
                  g_pDisplayDlg->SetAddressPointer(m_nAddressPointer);
               }
               break;

      .....
      .....
      .....

      default:
         strCommand = "Invalid command";
         break;
   }
   if (g_pDisplayDlg)
      g_pDisplayDlg->ShowCommand(strCommand);
}
```

the display is ready to receive another byte and so our simulation needs to incorporate a busy flag in order to cater for this.

We use CExtensionState::IncMachineCycles to time out the busy flag. IncMachineCycles is called after each instruction is simulated with an argument that contains the number of machine cycles elapsed since the last call. Therefore it allows us to implement cycle accurate features in our extension.

THE A/D CONVERTER

Moving on to esim1.dll, the A/D converter that this extension simulates is driven by port 1 of the microcontroller. Our extension for this device will therefore use CExtensionState::SetPortPins to monitor the output from this port and CExtensionState::GetPortPins to send data back to it.

To help us get the simulation right, we have constructed a UML style statechart depicting the operation of the device. This is based on the description in the manufacturer's data sheet and is shown in figure 3.

The code in SetPortPins and GetPortPins is based upon this statechart and, if we had had the knowledge to construct this statechart a few years ago when we were developing the original instrument, it would have helped us with the development of our original 8051 program too.

Photo 4 shows the dialog box that we have constructed for the A/D converter. This displays the attributes of the device and also allows us to directly enter data representing the voltage level on its inputs.

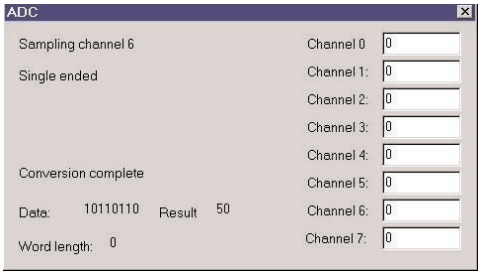

Photo 4. The simulating A/D converter can fetch its inputs from the dialog box edit fields or from named pipes running anywhere else on the system.

NAMED PIPES

However, we really want the inputs to the A/D converter simulation to come from the tilt sensors and battery and, for re-usability reasons, these are supported in a separate extension. To communicate between the two extensions we have used named pipes. Extension esim0.dll creates three named pipes, which it calls TiltSensor1, TiltSensor2 and Battery, and esim1.dll connects to these named pipes and requests data from the appropriate one when it needs an input level.

Named pipes work system wide and the operating system looks after the details. To the program, they behave just like files. To service a pipe, esim0.dll must create a separate thread. It then loops continuously and most of the time waits for a return from the ReadFile function. When this function returns, it is in response to a request for data and so the thread gathers the data and sends it back to the requestor using the WriteFile function.

Extension esim0.dll will therefore create three separate threads and there will be three separate channels of communication between esim0.dll and esim1.dll. The tilt sensors and

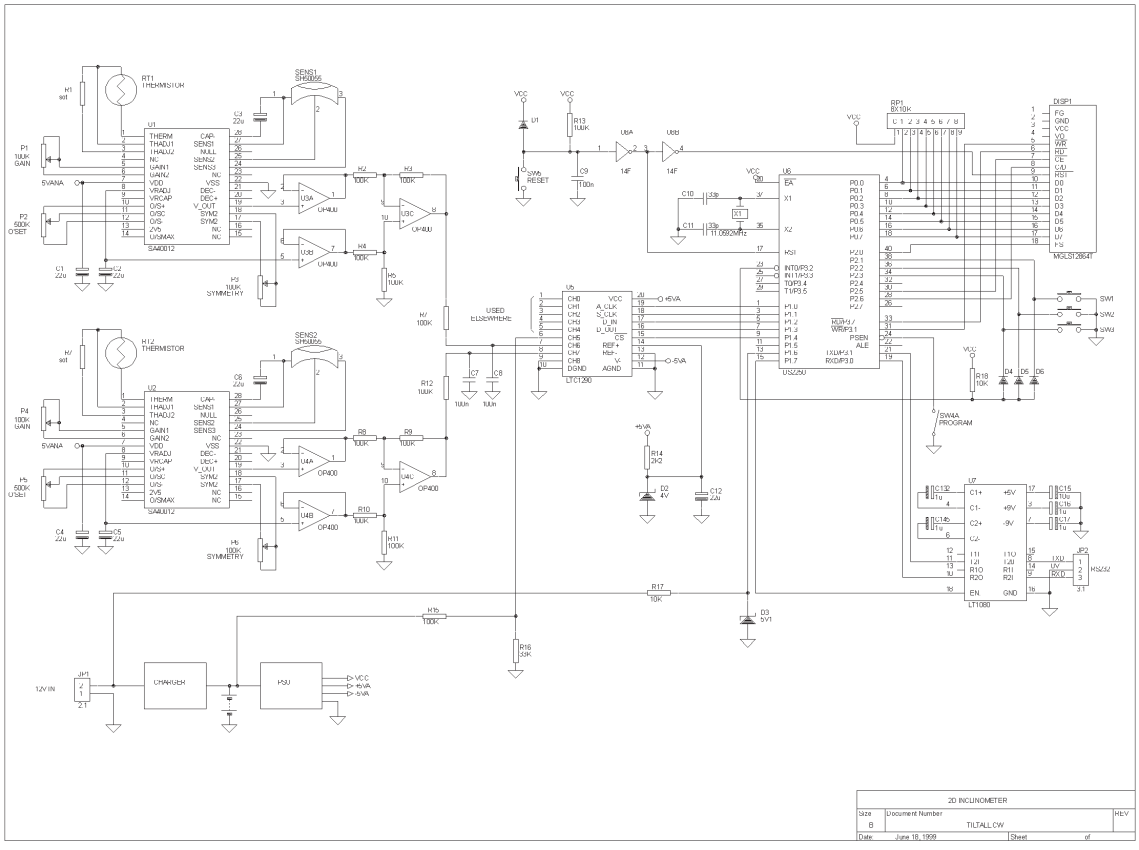

Figure 2. *The electronic spirit level is a sub-set of the Crossware Optical Tilt Sensor. It uses two cermic incinometers to detect orientation in two axes. A moving image of a bubble on the graphics LCD makes it immediately familiar to users.*

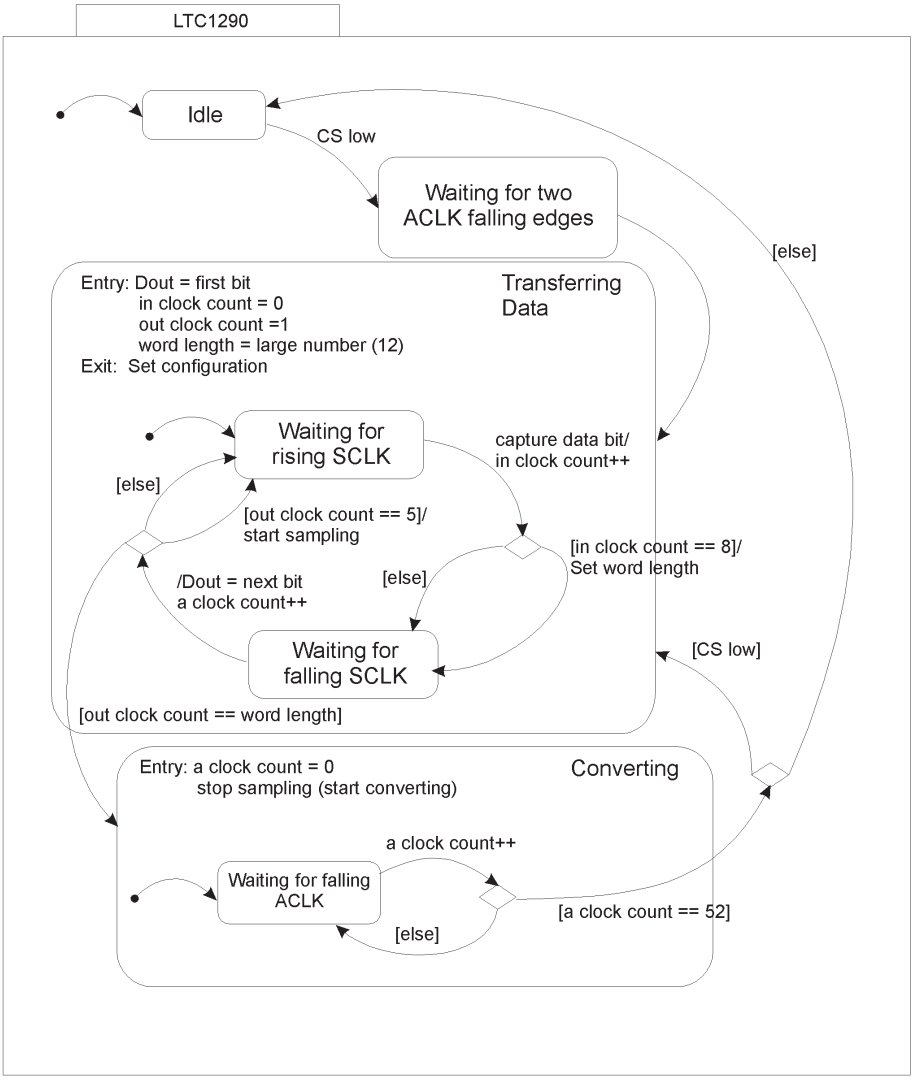

Figure 3. *A manufacturer's data sheets will usually give a written description of the operation of a device. It is often easier to understand the detailed operation of the device if this description is translated into a UML style statechart. This statechart shows an approximationof the LTC1290 A/D converter.*

battery charge can then be separately controlled using the dialog box shown in photo 5.

We simulate drain on the battery by using the IncMachineCycles function in which we decrement a variable representing the state of charge at a rate which corresponds with the rate of simulation. Similarly we simulate charging by incrementing this same variable whenever the charge check box is checked.

We use the MFC function fxBeginThread to start each new thread. This is quick and easy but care must be taken when accessing Windows functions. Any action on a window handle other than using it to post a message is likely to fail. We therefore program the dialog box to independently keep the three variables m_nTiltSensor1, m_nTiltSensor2 and m_nBattery up-to-date and use a CCriticalSection to ensure that these variables are not accessed simultaneously from separate threads.

DEBUGGING WITHOUT HARDWARE

With all of our extensions in place we can now run a complete simulation of our target system and use it to develop our 8051 program. We can add additional features to the extensions to trap error conditions or to automate testing.

You will notice, if you examine the source code for the custom extensions in detail, that we have not simulated everything. In particular, there are many features in the display driver that we do not use and so have not added support for. The objective is to speed up development and support the verification and life cycle processes. Spending time providing features that will not be used does not support that objective.

It took two to three days to develop our four extensions, with the display extension requiring the majority of this effort. The benefits of being able to see the internal attributes of the display driver and A/D converter, to be able to test a wide range of tilt sensor inputs, to be able to automate the testing process, to be able to do all this independently and at any time makes it worthwhile even where the real hardware is available.

If you are really looking for benefits, how about emailing your requirements specification and Virtual Workshop extensions to a low cost resource on the other side of the world and getting them to develop your program while you spend some time on the beach. Then again, developing your embedded program is not going to be quite as difficult as it otherwise would, and you will see quite enough of the beach while you are waiting for the hardware to catch up.

Alan Harry is the founder and managing director of Crossware, a developer of programmer-friendly C cross compilers and other development tools for embedded systems based on the 8051, ColdFire, 68000, CPU32 and other chip families. He also heads up a multi-disciplinary product development consultancy working on leading-edge developments for international clients.

REFERENCES

1. Michael Smith, Developing a Virtual Hardware Device, Circuit Cellar Ink, 64, 36-45, November 1995.

2. Bruce Powel Douglass, Real-Time UML, Developing Efficient Objects for Embedded Systems, Addison-Wesley, ISBN 0-201-32579-9.

SOURCES

8051 Virtual Workshop
Crossware Products
Old Post House
Silver Street
Litlington
Royston
Herts
SG8 0QE
UK

LISTING FIVE. *Using named pipes is a convinient way of communicating between extensions. The operating system does the hard work of connecting the ends and routing data between them.*

```
// the routine in esim0.dll that is running three times
// in three separate threads

CCriticalSection g_CriticalSection;

UINT PipeRoutine(void* pParam)
{
   const char* pszName = (const char*)pParam;
   CString strPipeName;
   strPipeName.Format("\\\\.\\pipe\\%s", pszName);
   HANDLE hPipe = CreateNamedPipe(strPipeName,
      PIPE_ACCESS_DUPLEX,  // dwOpenMode
      PIPE_TYPE_MESSAGE |
      PIPE_READMODE_MESSAGE |
      PIPE_WAIT,
      PIPE_UNLIMITED_INSTANCES,
      BUFSIZE,
      BUFSIZE,
      PIPE_TIMEOUT,
      NULL);
   if (hPipe == INVALID_HANDLE_VALUE)
   {
      CString strMessage;
      strMessage.Format("Could not create pipe %s", strPipeName);
      AfxMessageBox(strMessage);
      return 0;
   }
   BOOL bConnected = ConnectNamedPipe(hPipe, NULL) ? TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);
   if (!bConnected)
   {
      // exit thread
      CString strMessage;
      strMessage.Format("Could not connect to pipe %s", strPipeName);
      AfxMessageBox(strMessage);
      return 0;
   }
   CSingleLock AccessToExtensionState(&g_CriticalSection);
   while (nExtensionCount > 0)
   {
      char chRequest[BUFSIZE];
      char chReply[BUFSIZE];
      DWORD nBytesRead, nReplyBytes, nWritten;
      BOOL bSuccess = ReadFile(hPipe, chRequest, BUFSIZE, &nBytesRead, NULL);
      if (!bSuccess || nBytesRead == 0)
         break;
      AccessToExtensionState.Lock();
      g_pExtensionState->GetAnswerToRequest(chRequest, chReply, &nReplyBytes, pszName);
      AccessToExtensionState.Unlock();

      bSuccess = WriteFile(hPipe, chReply, nReplyBytes, &nWritten, NULL);
      if (!bSuccess || nReplyBytes != nWritten)
         break;

   }
   FlushFileBuffers(hPipe);
   DisconnectNamedPipe(hPipe);
   CloseHandle(hPipe);
   return 0;
}

// CExtensionState constructor in esim0.dll
// tilt sensors and battery extension

CExtensionState::CExtensionState()
{
   ....
   ....
   ....
   g_pExtensionState = this;  // let threads access class functions
   AfxBeginThread(PipeRoutine, (void*)"TiltSensor1");
   AfxBeginThread(PipeRoutine, (void*)"TiltSensor2");
   AfxBeginThread(PipeRoutine, (void*)"Battery");
   ....
   ....
}

// CExtensionState constructor in esim1.dll
// A/D Converter extension

CExtensionState::CExtensionState()
{
   ....
   ....
   ....
   for (int i = 0; i < NO_OF_CHANNELS; i++)
   {
      m_hPipe[i] = INVALID_HANDLE_VALUE;
   }
   m_hPipe[5] = OpenNamedPipe("Battery");
   m_hPipe[6] = OpenNamedPipe("TiltSensor1");
   m_hPipe[7] = OpenNamedPipe("TiltSensor2");
}
```

Tel: + 44 (0) 1763 853500
Fax: + 44 (0) 1763 853330
Web: http://www.crossware.com

Graphics LCD Module
MGLS12864T-LV2
VL ELECTRONICS, INC.
3250 Wilshire Blvd., Suite 1901,
Los Angeles, CA 90010,
U.S.A.
Tel: (213) 738-8700
Fax: (213) 738-5340
Web: http://www.vle.com

LCD Driver Chip T6963C
Toshiba America
Electronic Components, Inc.
3700 Crestwood Parkway
Suite 460
Duluth, GA 30136
Tel: (770) 931-3363
Fax: (770) 931-7602
http://www.toshiba.com

A/D Converter LTC1290
Linear Technology Corporation
720 Sycamore Drive
Milpitas, CA 95035
Tel: (408) 432-1900
Fax: (408) 434-0507
http://www.linear-tech.com

SH50055 Tilt Sensor and Driver
Spectron Glass and Electronics Inc.
595 Old Willets Path
Hauppauge, NY 11788
Phone: (516) 582-5600
Fax: (516) 582-5671
http://www.spectronsensors.com

T2250 Soft Microcontroller
Dallas Semiconductor
4401 South Beltwood Parkway
Dallas, TX 75244
Tel: (972) 371-4167
Fax: (972) 371-3715
http://www.dalsemi.com/