Benchmarking Real-time Determinism in Windows CE

Chris Tacke, eMVP
CE Product Manager, Applied Data Systems

Lawrence Ricci, eMVP
Business Development Manager, Applied Data Systems

Abstract

The real time or deterministic performance of Windows CE has been extensively investigated for applications in industrial control systems. With the release of CE.NET, engineers are asking if the new OS is more or less agile than its widely used predecessors, 2.12, and 3.0. This white paper first establishes the real-time performance of CE 3.0 on the 'industry standard' StrongARM™ platform, and then compares it in detail to CE.NET, and its predecessor CE 2.12

Real-Time performance was tested by using a standard function generator to create a hardware interrupt on the device. The device then starts an IST which immediately sets a GPIO line high and sets a Windows event. Further, an application thread receives that event and sets another GPIO line high. We tested and are reporting several timing differentials with reference to the generated hardware interrupt and the time at which the output GPIO lines actually went high.

These are 'hard tests' for 'hard real time'. There are no measurements made 'internal' to the hardware/software platform, and there are no semantic games on the nature of hard and soft real time. The results, measured with high speed, high precision instruments, were surprising, suggesting that specifications for CE real time performance were too conservative. CE as an RTOS performs far better then generally discussed.

Introduction

The tests we ran are simple and direct. We send a signal to a StrongARM-based controller running Windows CE of various versions, and measure how long it takes to respond. The “latency time” and “jitter time” of this response are the quantified measures of determinism

As a separate test, we follow the methodology of Dupre and Barcos published at http://www.isa.org/~pmcd/acs/brtd.html. This test does not measure latency and jitter, but does measure ‘saturation’ which we define as the repetitive interrupt loading where the system becomes saturated and its response becomes non-deterministic.

Finally, to more accurately reflect the environment of a real device in actual operation, we introduce a load in the form of the “Polygons” program. The Latency, Jitter and Saturation are measured under load.

We like these tests because of their reliance on ‘full loop’ testing- beginning with the stimulus of a system input and measuring the system output. The tests make no assumptions about the ‘internals’ of the device. The tests make no references to ‘interrupt latencies’ or ‘context switch times’; it is a basic measurement of the full system- both hardware and software performance.

The reader should note how this test might map to the actual limiting case for real time performance of the OS in a target application.

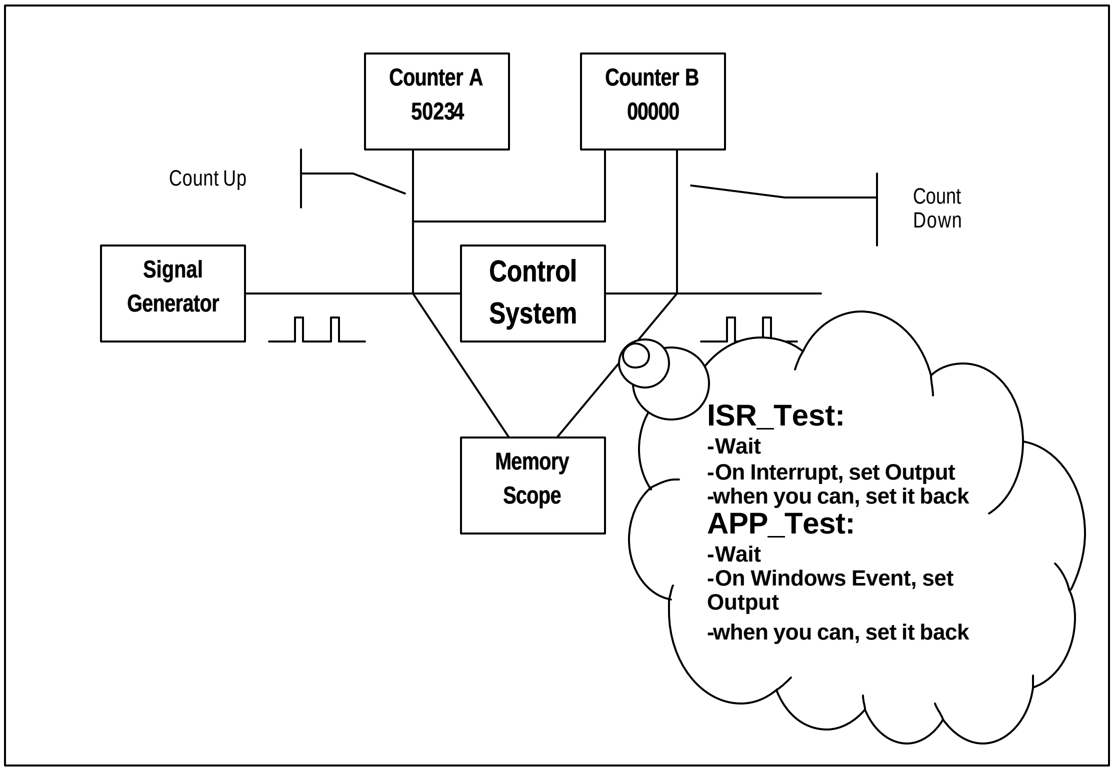
On the one hand, the ‘control logic’ in this test is very low overhead, simple C++ Programs that read a processor-IO input on interrupt or sense a windows event and set an output. Also, a real controller would have additional latencies associated with the hardware multiplexing to a ‘real world’ process I/O subsystem. Likewise, this test is independent of networking and calculation loads, and it is independent of the overhead associated with a soft-control executive. It should be noted that all these loads would typically be independent of OS used, so we remain focused, as intended, on raw OS performance.

On the other hand, the system tested has not been optimized for real-time performance and carries the full “Max ALL” CE build, with all CE utilities installed. Also, the StrongARM provides the additional capability of continually providing the graphics for a VGA flat panel display, highlighting its performance even further.

Test Equipment

Our test setup follows that of Dupre and Barcos, with the addition of a high-speed memory scope to let us measure latency, jitter and thread run time.

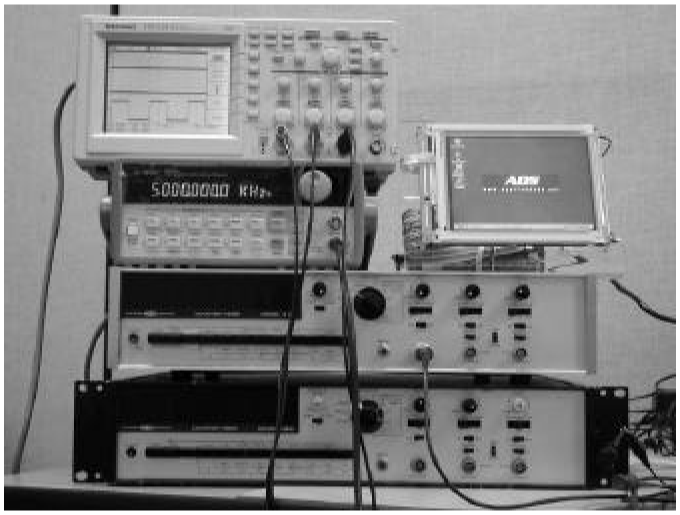
Figure 1- Initial Test Setup



In our setup the following equipment was used

Control System ADS Graphic Master Development System
Signal Generator Agilent 33210A
Counters A, B Systron Donner 6150
Oscilloscope: Tektronix TDS 224

Figure 2 Photograph of test setup



Code may be hardware independent, but embedded systems programmers seldom are

Graphics Master Test Platform

The Graphic Master is a StrongARM-based single board embedded computer furnished by Applied Data Systems, Inc. (ADS), in an 'application ready' format (all device drivers, BSP, full suite of CE peripherals were loaded). The system is SA 1110/1111 running at 206 MHz, with 32 Meg Flash and 32 Meg RAM. The system includes a high-speed 8 bit micro-controller often used as a real time front-end I/O processor, but this was not used in the tests. It also includes Ethernet, 7 serial ports, a CAN bus, PCMCIA and Compact Flash. General purpose I/O lines were used for input and output.

The CE Builds on the Graphics Master used for this test were

CE 2.12
CE 3.0
CE.NET

In all cases, this is the 'standard' out of the box configuration with fully graphics, desktop, control panels, networks, browsers etc. While an optimized build for performance was available by ADS, it was decided to utilize a non-optimized build for this analysis to understand a "worst case" scenario for the Graphics Master.

The interrupt line we used for the test is most normally used for a 'power on/power off' pushbutton to signal transition in and out of power saving 'sleep' mode. In this service, it is useful to have some small filters on the board to eliminate contact bounce from the typical membrane pushbutton. Once we started running the test, it became clear that the OS real time performance on the Graphics Master was much quicker than we expected, and was occurring at time intervals inside the 'contact bounce' of a button, so we removed the filters from the board. It should be noted that this type of selective depopulation is a routine operation for ADS systems as purchased by OEM's.

More detailed specification of the Graphics Master can be found here: http://www.applieddata.net/products_master.asp

To help the reader understand the general performance level of this system, we benchmark using "Polygons" for each OS, and compare these to other systems. This is not 'apples to apples' comparison since IPAQ is QVGA and therefore has only ¼ the number of pixels to update, but it does provide a general indication of the efficiency of the OS BSP. For comparison, we also ran Polygons on a slightly different version of Graphics Master running 3.0 with a QVGA display, and a desktop Pentium running CEPC

Figure 3 Polygon Benchmarks

Polygons/Second	CE 2.12	CE 3.0	CE.NET
Graphics Master (VGA 8bit)	**170**	**751**	**537**
Graphics Master Variation (QVGA 8bit)		**980**	
3630 iPaq (QVGA 16bit)		**260**	

As of this date (March 2002) this is an early release of the CE.NET BSP for this system; some improvement is expected over the next few months.

The particular 'control logic' for this series of tests was of two types. One routine (ISR_TEST) was a few lines of code linked directly to the ISR. When the interrupt occurred, the routine ran with the ISR and set an output on the Graphics Master. This would be the type of code used for the most demanding real-time applications and runs at 'interrupt priority'. A typical application for this type of real-time code would be something that must happen very quickly, for example the register store and shutdown required to put a system into sleep as a result of a power failure, or an application with a high-speed 'stream' of interrupts, for example a pulse encoder.

The second routine APP_TEST was analogous to typical 'control packages' which communicate only with the standard OS API and messaging system. For this routine, the ISR sets a Windows™ event. When the event is recognized by the APP_TEST routine it runs, turns a different output on, and they shuts it off as soon as it finishes running. This routine ran quite well in our tests and requires no knowledge of the driver structure to implement, so is the better choice for most applications.

Code for APP_TEST test is as shown if Figure 4. It is identical code for all versions of the OS, written in C++.

Figure 4 APP_TEST

```
#define LED_ON               1
#define LED_OFF                     0
#define ITERATIONS           1000000

DWORD EventThread(LPVOID threadParams);

HANDLE  hPort                = INVALID_HANDLE_VALUE;
HANDLE  hPwrOffEvent   = INVALID_HANDLE_VALUE;
HANDLE   hThreadEvent  = INVALID_HANDLE_VALUE;
TCHAR   szEvent[]            = _T("PWROFF");
TCHAR   szThreadEvent[]      = _T("DONE");
TCHAR   szLEDName[]          = _T("LED3:");

int WINAPI WinMain(   HINSTANCE hInstance,
                                      HINSTANCE hPrevInstance,
                                      LPTSTR    lpCmdLine,
                                      int       nCmdShow)
{
       HANDLE  hThread;

       hThreadEvent = CreateEvent(NULL, FALSE, FALSE, szThreadEvent);
       hThread = CreateThread(NULL, NULL, EventThread, NULL, 0, NULL);

#if (_WIN32_WCE < 300)
       SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);
#else
       CeSetThreadPriority(hThread, 0);
       CeSetThreadQuantum(hThread, 0);
#endif

       WaitForSingleObject(hThreadEvent, INFINITE);

       return 0;
}

DWORD EventThread(LPVOID threadParams)
{
       BYTE   ledStatus;
       DWORD  dwSize;
```

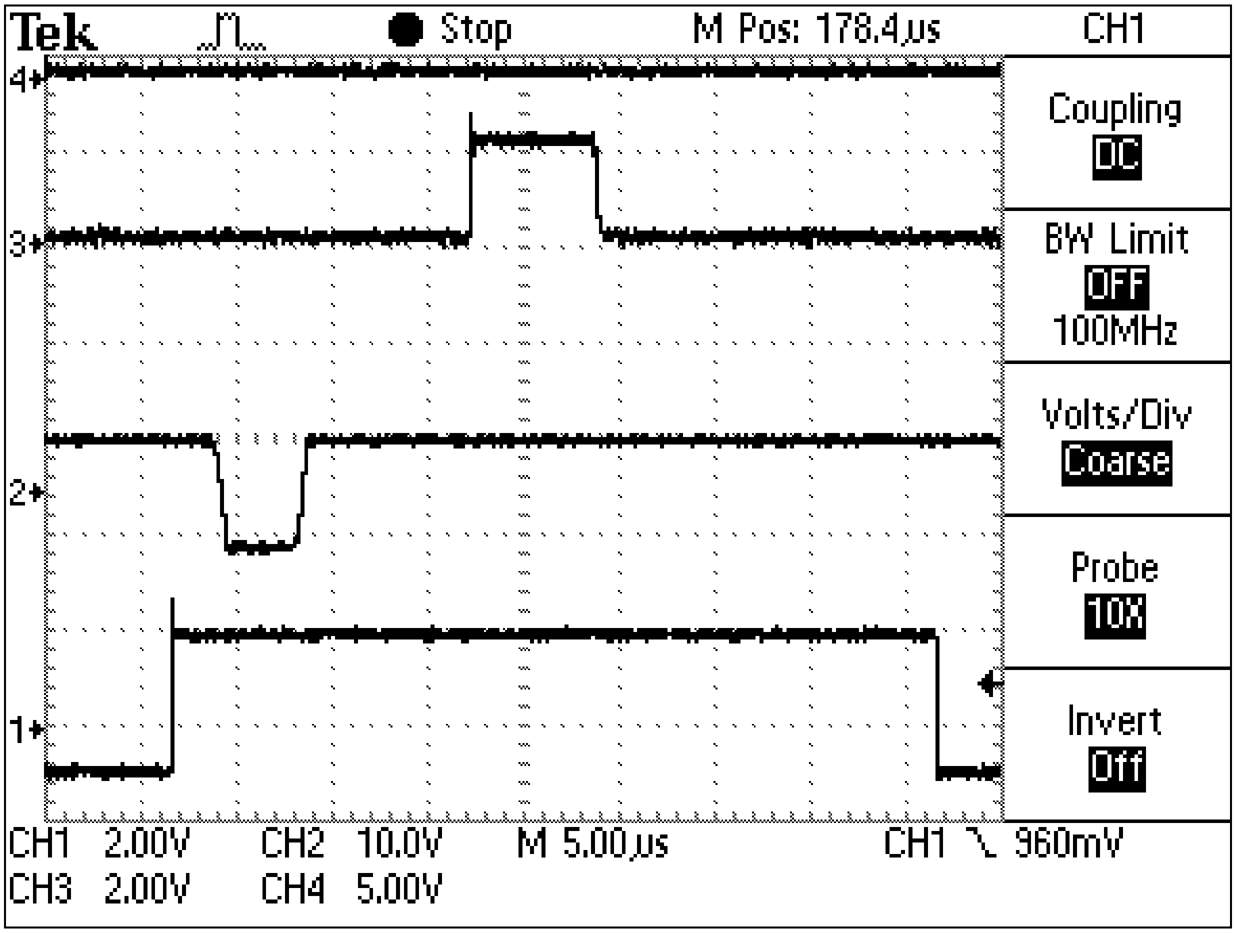
```
        DWORD   dwRet;

        // open LED
        hPort = CreateFile(szLEDName, GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0,
                           NULL);

        // check for success
        if(hPort == INVALID_HANDLE_VALUE)
        {
               RETAILMSG(TRUE, (_T("\r\nFailed to open LED: %d\r\n"),
                                   GetLastError()));
               return -1;
        }

        // ensure LED is off
        ledStatus = LED_OFF;
        WriteFile(hPort, &ledStatus, 1, &dwSize, NULL);

        // create event
        hPwrOffEvent = CreateEvent(NULL, FALSE, FALSE, szEvent);

        // check for success
        if(hPwrOffEvent == INVALID_HANDLE_VALUE)
        {
               RETAILMSG(TRUE,(_T("Failed to create event: %d\r\n"),
                                GetLastError()));
               return -1;
        }

        for(int i = 0 ; i < ITERATIONS ; i++)
        {
               // wait for interrupt-driven event
               dwRet = WaitForSingleObject(hPwrOffEvent, INFINITE);

               // turn on LED
               ledStatus = LED_ON;
               WriteFile(hPort, &ledStatus, 1, &dwSize, NULL);

               // turn off LED
               ledStatus = LED_OFF;
               WriteFile(hPort, &ledStatus, 1, &dwSize, NULL);
        }

        // close port
        CloseHandle(hPort);

        // destroy event
        CloseHandle(hPwrOffEvent);

        SetEvent(hThreadEvent);

        // destroy event
        CloseHandle(hThreadEvent);

        return 0;
}
```

Detailed Tests of CE 3.0

While the Graphics Master runs all OS variations, we used 3.0 here as a base case to outline the detail of the methodology. We set up the apparatus with the controller comfortably monitoring and passing on square wave inputs at a frequency of 5000 Hz, (200 micro second period peak to peak for pulses) with pulse generator set at 20% duty cycle (each pulse 40 microseconds wide). The frequency of 5000Hz was chosen to be comfortably less than saturation, but still more than expected of a 32 bit Graphic system in real-world applications.

The input pulse signal was input as channel 1(bottom trace) to the memory scope, acting as the trigger. The Interrupt-linked ISR_TEST was channel 2 (middle trace), and the windows event linked APP_TEST was channel 3 (upper trace).

Figure 5 - 3.0 Latency, unloaded scenario



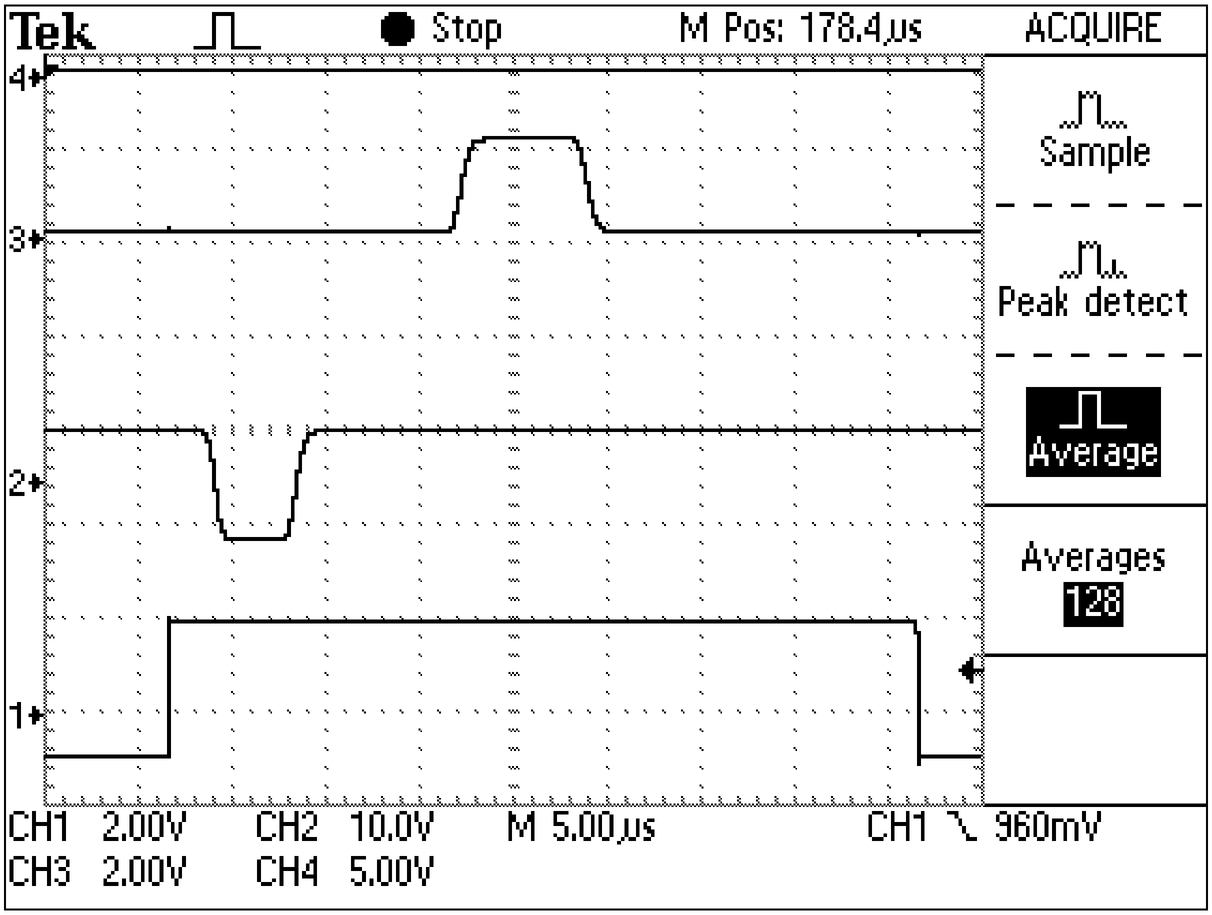
As you can see, the interrupt linked ISR_TEST output lagged the input signal by about 2.5 microseconds. The windows-event linked APP_TEST output lagged the interrupt input by 16 microseconds.

Most readers will note these numbers are far inside any specification limits typically discussed by Microsoft or most professionals. To help quantify this performance in meaningful terms, consider the latency (propagation delay) in cable is about 1 nanosecond per foot. This means the ISR_TEST response occurred as if the entire Graphics Master (StrongARM+CE) system were replaced with a 2,500 foot cable. Most real world designs do not find determinism a meaningful problem in a spool of cable. Alternately, consider that the lions share of real-time processes occur outside the period of one millisecond, for example the specification of industrial SOE recorders is 1ms resolution +/- 1ms accuracy. If you are viewing the above chart on normal paper, the one-millisecond mark is about 15 feet to your right. CE 3.0 is clearly, in most environments, well inside the time window used to discuss determinism.

Since determinism is a 'statistical' property, we needed to measure this latency over a extended number of samples to quantify "Jitter" or the variation in latency. It would have been nice to make a statistical distribution plot of the start of rise time and the start of completion time, but we did not have suitable equipment,. Nevertheless, we were able to make a closely related measurement.

The scope used for testing has a feature to average samples, giving us a good way to measure this Jitter. The "rise time" of the controller output is sharp, only a few nanoseconds, so as the pulse would jitter back and forth, an average would be created for the last group of samples. The average for two pulses, jittered back and forth, would be a 'one step' stairway to the top of the pulse. For three staggered pulses, two steps and so on. For the 128 samples offered by the scope, this gave the averaged output a smooth 'slope' up to a flat-topped peak. The time duration of the 'slope' is taken as the Jitter.

Figure 6 Jitter 3.0 System under no load

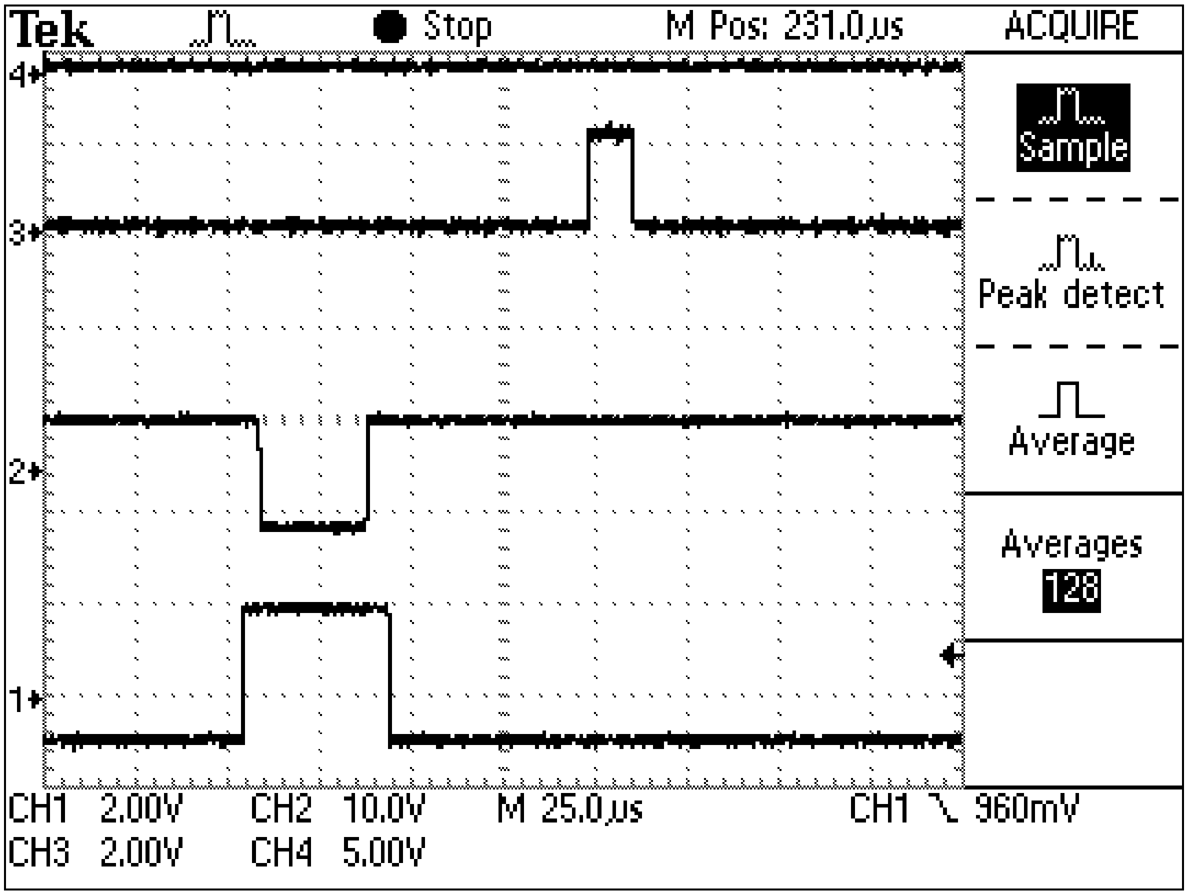


Here we can see that 95% or more of the "jitter" or variation in the latency tends to occur within half a division, or about 2.5 microseconds. This is essentially the same for both the ISR_TEST and APP_TEST outputs of the unloaded system. In this test, we do, however, see a few pixel high 'step' trailing each of these outputs, suggesting that there were some outliers, which later we measure under load.

This system in this test was 'unloaded'. No programs except ISR_TEST and APP_TEST were running, and the system was not being driven to saturation. The realistic case would include some background load for graphics, algorithmic processing, communalizations, etc. To simulate this load, we use the Polygons test program shipped with Windows CE and running at top priority. This program presents intensive integer compute demands on the CPU and contends with APP_TEST for processing at the highest priority level. In our experience, Polygons is a far more intensive load than most Graphics MMI/SCADA applications. The nature of the load it presents on the CPU in a StrongARM system is largely integer arithmetic, much like a control program. In short, Polygons is on balance quite a realistic load.

Repeating the first test, Figure 1, we see essentially no change for the system under load.

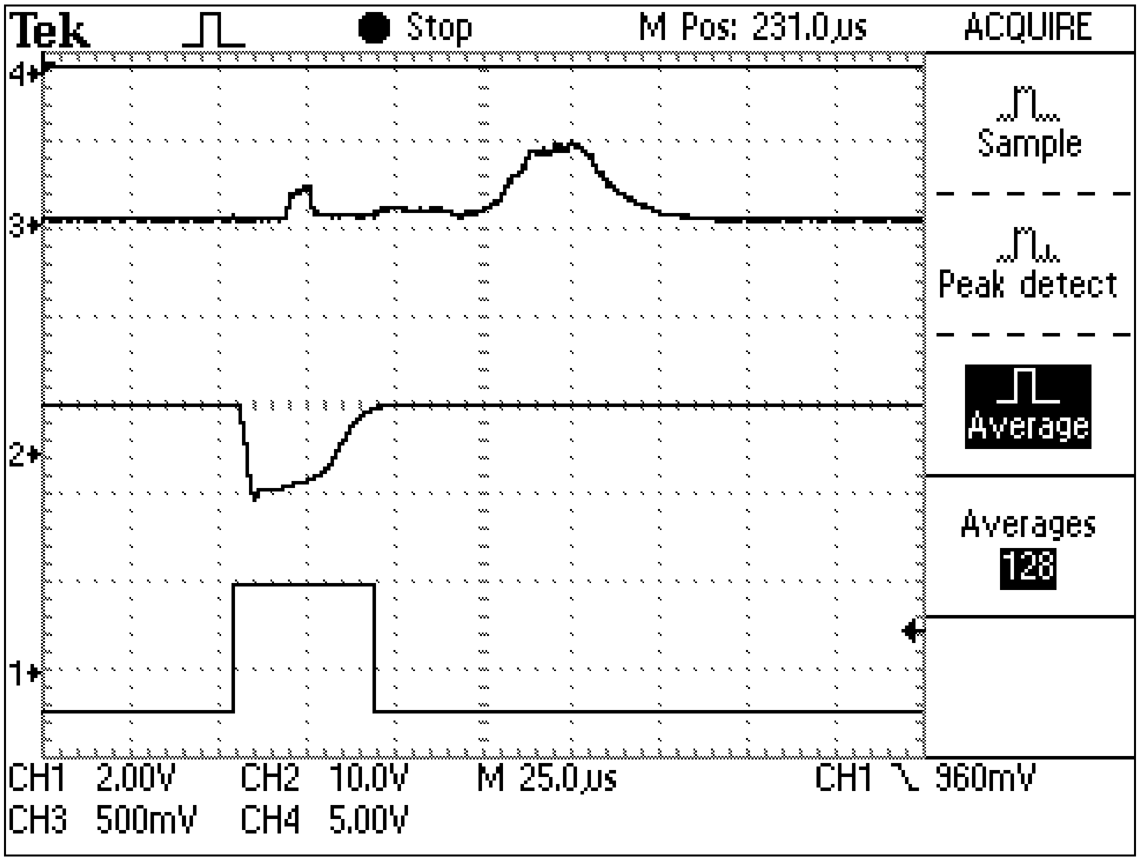
Figure 6 - 3.0 System Under Load



This is the same chart as Figure 5, this time with the system under load from Polygons and with the time axis compressed X10 to 25 microseconds per division. We would expect the ISR_TEST to be the same (it is) because it runs at OS/Interrupt priority. The APP_TEST, however, now contend at parity priority with Polygons so it shows a much higher latency, about 133 microseconds, depending on how the OS spreads CPU cycles between two identically prioritized tasks.

The next test is to measure the average response under load.

Figure 6 3.0 Average Response E 3.0 Under Load

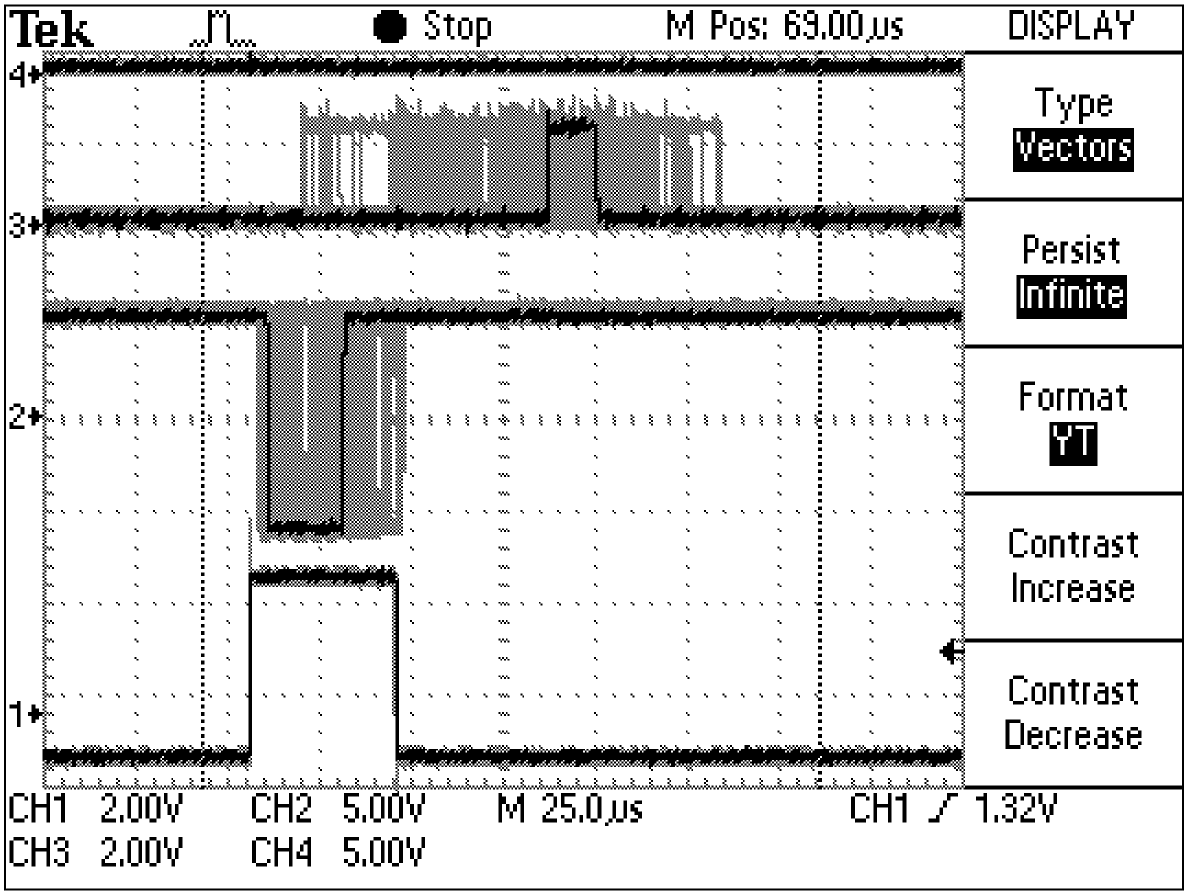


Here we can see the ISR_TEST, linked to the interrupt, is still very deterministic with respect to when it starts, even under load. Its completion time, however, becomes more varied, as expected. Both IRS_TEST and APP_TEST are deterministic as to when they signal output "on", but they are dependant on code run time as to when they shut output "off".

The response to the Windows-Event scheduled APP_TEST is in some ways the opposite. The OS is very deterministic in ensuring the APP_TEST will run within the 133, microsecond window even contending with Polygons, but a measurable number of executions occur much earlier, at peaks about 15 microseconds and 30 microseconds. So, we should understand that with for Windows CE 3.0, determinism of an interrupt-linked process is best interpreted with respect to how quickly they start. Determinism of windows-event scheduled processes is best interpreted with respect to when they *must* start, with the time interval understood more as a 'back' wall.

Apart from the average, an analysis of determinism needs to understand the behavior of outliers, the small number or responses (early or late) that are far from the mean. To measure theses outliers, we use the persistence feature of the scope which records and overwrites ALL traces for a total many hundreds of thousands of traces. The result does show outlier behavior:

Figure 7 CE 3.0 Under Load, thousands of samples



Here we can see the deterministic behavior measured over many tens of thousands of cycles. The dark line is the last recorded response. Here we can see that all interrupt-linked processes were complete within about 50 microseconds and all windows event linked processes were complete within about 125 microseconds.

Maximum Interrupt Frequency and Saturation Frequency for Three Systems

After the above detailed investigation of CE 3.0, we can now compare the Maximum Frequency and Saturation Frequency of all three CE Versions

In the manner of Dupre and Barcos, the test is set up to measure the maximum frequency (MF) that the controller can accept input and faithfully reproduce output. This is measured separately for both ISR_TEST linked to interrupts and APP_TEST linked to windows events.

We also consider the 'saturation frequency' (SF) of the controller an important variable, indicating when processing real time inputs preempts other programs in the system.

Variable	Name	Definition
MF-ISR	Max Frequency	The input frequency at which the output of a ISR-linked process is seen to miss (any) input pulses
MF-ISR	Max Frequency	The input frequency at which the output windows-event linked profess is seen to miss (any) input pulses
SF	Saturation Frequency	The input frequency at which the system can no longer run Polygons

This test is the place where we will introduce a direct comparison of the three version for the CE operating systems because is shows so clearly the effect of the real-time recode between 2.12 and 3.0.

Figure 8 Dupre / Barcos Test Results

	MF-ISR interrupt linked	MF-APP windows linked	SF Polygons Dies	Comment
CE 2.12	20Hz	19 Hz		Notice! 20 Hz NOT kHz!
CE 2.12 Loaded	20Hz	19 Hz	N/A*	Polygons never stopped
CE 3.0	127 KHz	28 kHz		
CE 3.0 Loaded	127 KHz	15 kHz	85 kHz	
CE.NET	47 kHz	6.1 kHz		
CE.NET Loaded	47 kHz	6 kHz	40 kHz	

Here it becomes very clear that CE 2.12 was a very different animal before it went into its cocoon and emerged as some sort of super sonic 3.0 butterfly. CE 2.12 drops off the end' of this test at frequency of only 20 cycles per second, corresponding to the system time clock set, in this case, at a 50 ms 'tic'. Outside this load rate CE 2.12 behaves quite nicely, but inside this window it is non responsive. In short, a mixed choice for real time processing.

CE 3.0 however is another matter. We were not expecting anything like this interrupt-linked performance. We were watching a full graphic system servicing interrupts comfortably above 100 kHz, and even running background graphics at 80 kHz. If we had shut off the APP-TEST routine, it doubtless would have reached even higher frequencies.

When we placed the unit on a scope, we could see that that system continued to behave completely determinately until the frequency increased to the point that the ISR did not complete before the next ISR (on the same interrupt line) repeated. In other words, it was real time to the point that it was processing interrupts just about as fast as the CPU could compute. All this time the system was processing APP TEST and servicing other real time processes like the system clock and memory refresh. It is also notable that the load from Polygons had no effect on the interrupt linked processing at all; the system comply preempted this background application.

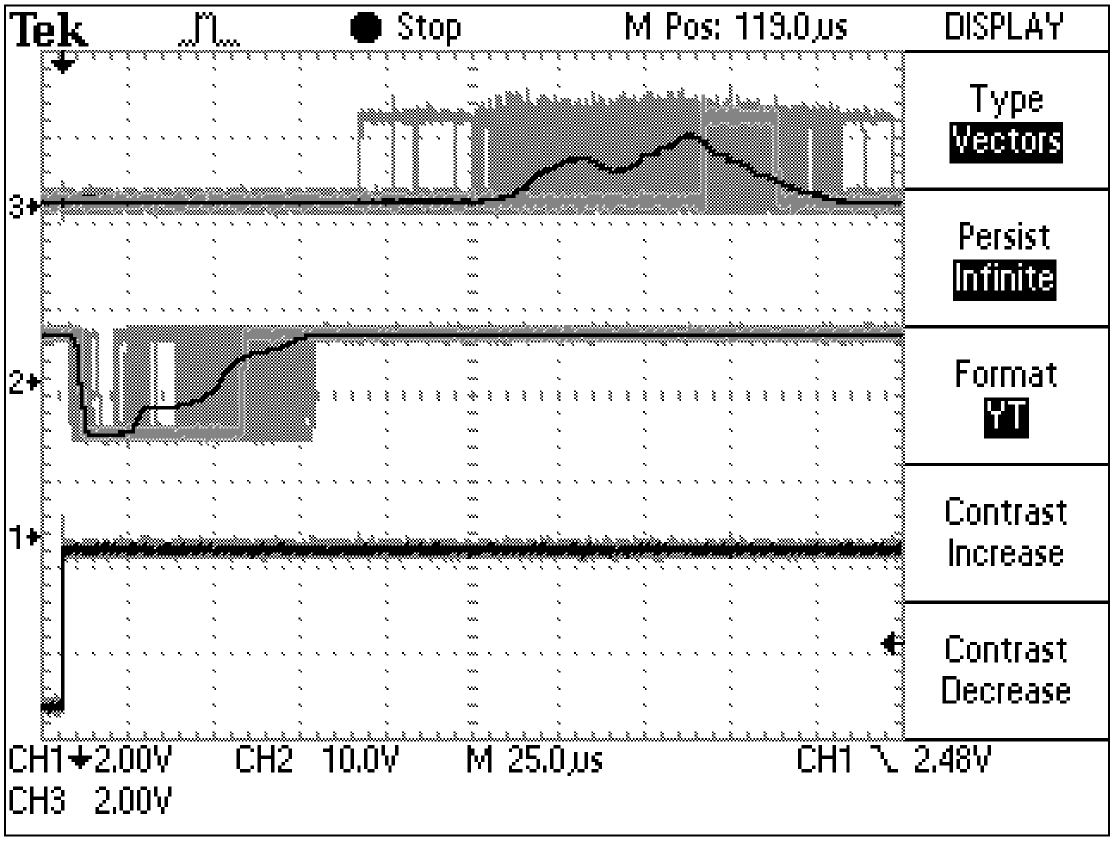
CE.NET is still very agile, even though it carries a lot more baggage in terms of network support and graphics- the perennial bug-bear of real time systems. CE.NET serviced interrupts right up to 47 kHz . While not exactly a threat to the AM band on your radio, this would have been a useful frequency for Marconi. For the vast range of events that are associated with electrical or mechanical devices, a frequency of 47 kHz, corresponding to a period of 20 microseconds, should be quite sufficient.

What is notable, and actually very positive with respect to the real time performance of CE.NET, is the way the maximum frequency of the windows event linked APP_TEST changed under background load. We can see that for CE.NET, the MF changed only about 2% when Polygons was running. In CE 3.0 MF_APP changed more than 50% as a result of backgrounds load. This means that for the form of real time programming most often used (windows event linked tasks) CE.NET is more deterministic than CE 3.0. Later, we will see the CE.NET situation with Latency and Jitter was even better.

This improvement is because some changes were made to the CE.NET interrupt handling to better arbitrate between two processes sharing the same priority- in this case APP_TEST and Polygons. This code seems to work very well to improve the repeatability and determinism of the system. We expect it is the presence of this additional logic that slows the maximum frequency of the interrupt paced system. If so, the trade-off is probably worth it. There probably are not too many processes that require a performance window between the ~10 microsecond performance of 3.0 and the ~20 microsecond performance of CE.NET.

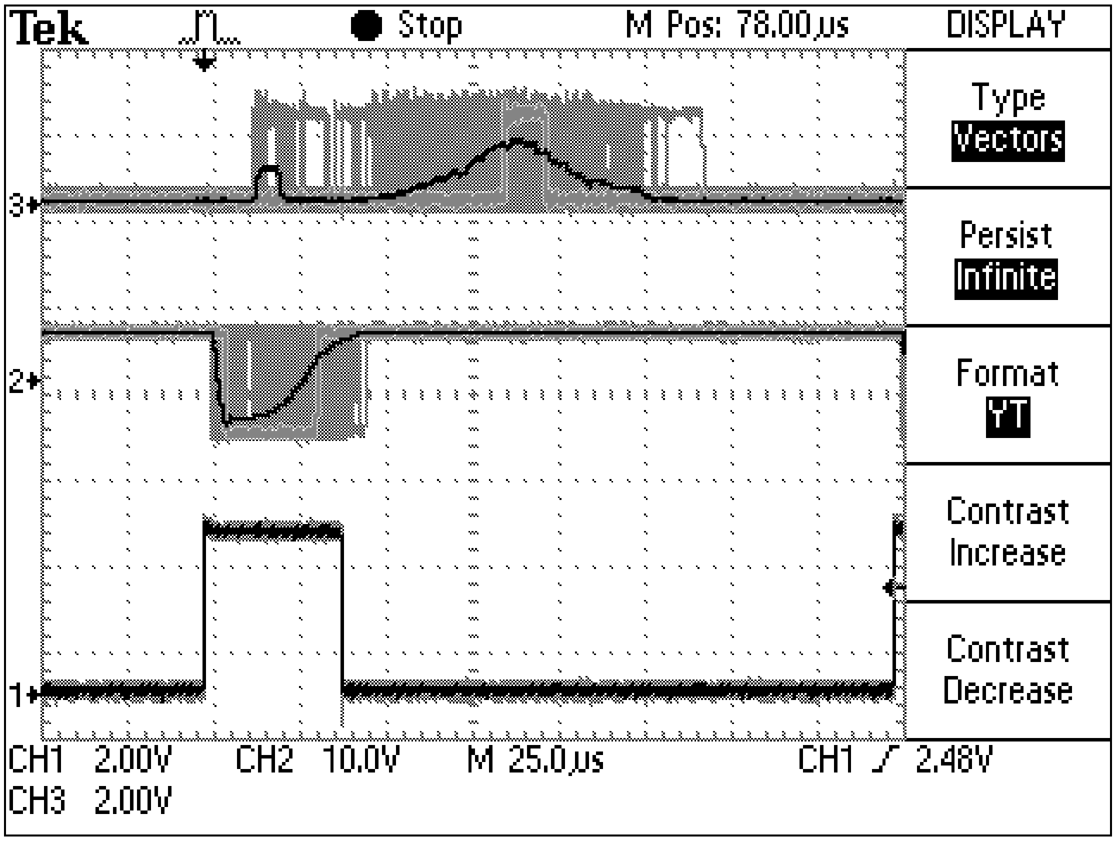
Graphic Comparison, Three Systems.
Here we condense the measurements onto a single chart grid to facilitate comparison of the three OS's that were run on the Graphics Master. All systems were running polygons. Note the horizontal axis is 25 microseconds per division.

Figure 9 CE 2.12 Overall Performance



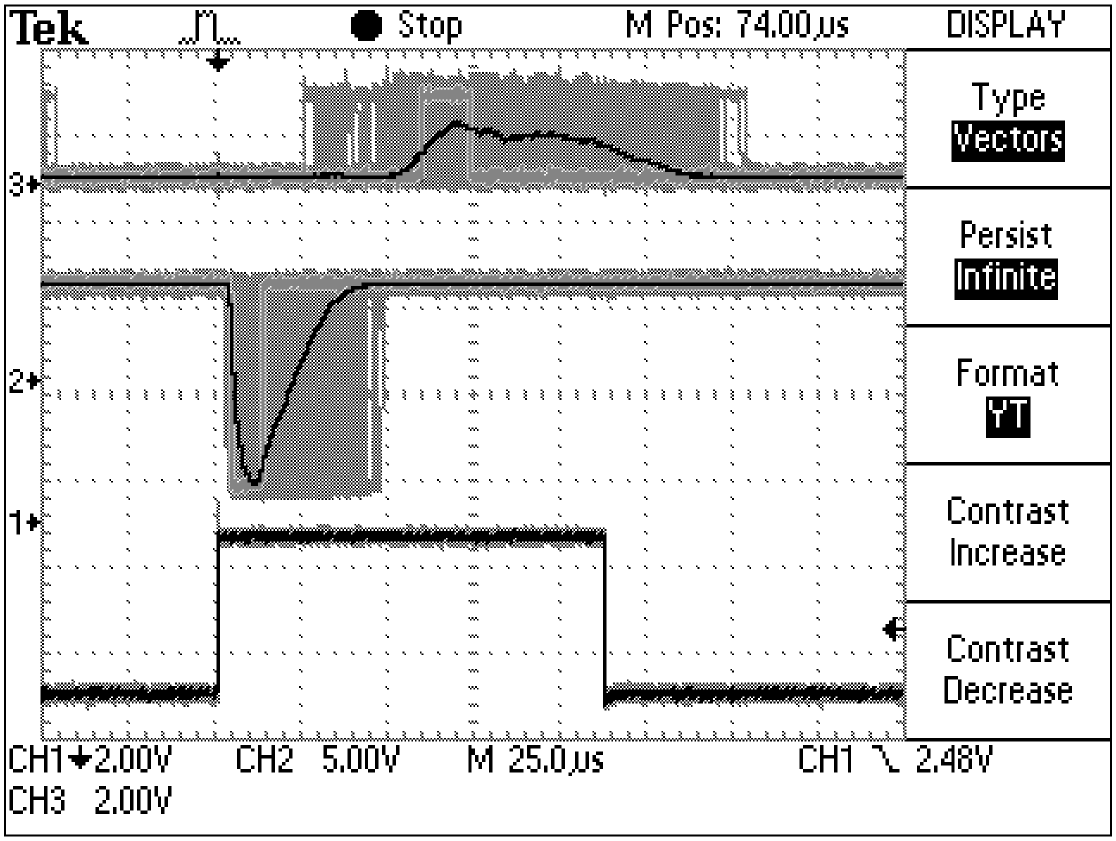
In this figure we overlay the average (dark black trace) the last (gray trace) and the montage of tens of thousands of traces (gray area), which catches the outliers. This is a 'loaded' system with Polygons running while the test was done. In this case, the test was run at a leisurely 19Hz because of the low saturation rate of CE 2.12.

Figure 9 CE 3.0 Overall Performance under Load



This is the response of a 3.0 system, again under load, but this time managing quite well at 5000 Hz. Notice the interesting response of the event-linked task, sometimes 'beating polygons to the CPU' and running just 10 microseconds after interrupt, the rest of the time waiting for polygons to finish and running about 100 microseconds after interrupt.

Figure 10 CE.NET Overall performance under load


Tek Stop M Pos: 74.00µs DISPLAY

Type Vectors

Persist Infinite

Format YT

Contrast Increase

Contrast Decrease

CH1 2.00V CH2 5.00V M 25.0µs CH1 ↘ 2.48V

CH3 2.00V

Here we see CE.NET under load. We notice that both the interrupt linked ISR_TEST and the windows event linked APP_TEST are more tightly defined, more deterministic in response. While almost as quick as CE 3.0, CE.NET is a little bit more predictable, especially for the APP_TEST under load.

Finally we can summarize several charts in a single table

Figure 11 Latency and Jitter; three Operating Systems

	ISR Start (ms)				**APP Start (ms)**			
	Max	Min	Jitter	Latency	Max	Min	Jitter	Latency
CE 2.12 **	6	2.6	3.4	3	162	51	111	61
CE 2.12** Loaded	11	3.9	7.1	4.8	201	117	84	135
CE 3.0	5.2	1.4	3.8	2.5	17.6	14.4	3.2	21
CE 3.0 Loaded	7	2	5	5	133	16	117	71
CE.NET	5	2	3	3	170	24	146	47
CE.NET Loaded	12.2	2.8	9.4	6.3	125	46	79	53

We think the place most engineers should focus is the performance of the 'loaded' system. Whether loaded by graphics, as in this case, or loaded with some sort of control/optimization algorithm, real time in a CE system will usually mean real time in a system also performing other tasks. Conversely, very few devices would invest a 32 bit

CPU, much less a full multi-tasking OS, in a single function device. In this measure, performance under load, CE.NET comes out very well.

Under load, the system actually becomes MORE deterministic. The "Jitter" on the APP_TEST actually reduces, not only to less than a loaded 3.0 system, but less than a CE.NET system under no load!! The actual Latency numbers are also quite good, with an interrupt linked response starting 6.3 microseconds after interrupt, and a windows even linked response starting only 53 microseconds after interrupt. Again, we point out this is an early build of the BSP for this system; we expect some improvement over the coming months.

Summary and Recommendations

We were very pleased with the real time response of CE 3.0 and CE.NET on the Graphics Master platform. While for certain tests CE 3.0 seemed to have an edge, we think that for the realistic cases- Windows™ events linked tasks on loaded systems- CE.NET is the clear choice. We also think the series of tests give a good general guideline for CE application to real time process. If the time intervals are measured in milliseconds, don't analyze too deeply, you are probably OK. If the time intervals are measured in tens to hundreds of microseconds, think about it and perhaps test your target system and BSP. If time intervals are single-digit microseconds, test carefully and consider hardware-based interrupt handling. If needed, optimizing the Graphics Master for interrupt handling would also be of tremendous value.