StrongARM™ Computational Performance Benchmarks

An Analysis of FFT Algorithms

InHand Electronics, Inc.
30 West Gude Drive, Suite 100
Rockville, MD 20850
240-558-2014
info@inhandelectronics.com
www.inhandelectronics.com

January, 2000
(text revised January 2002)

Copyright © 2000-2003 InHand Electronics, Inc. All Rights Reserved.

Copyright © 2003 InHand Electronics, Inc. All Rights Reserved.

InHand Electronics, Elf, Elf2, Elf3, Fingertip, Fingertip3 and BatterySmart are trademarks of InHand Electronics, Inc.

All other products or services mentioned herein may be trademarks of their respective owners.

No part of this document or the information contained within may be adapted or reproduced in any form except with the prior written permission of InHand Electronics, Inc.

Table of Contents

1. Introduction 1
2. Experimental Setup 3
   2.1. Hardware Platform 3
   2.2. Operating System (OS) Configurations 3
      2.2.1. No OS 3
      2.2.2. Windows CE 3
   2.3. Test Software 3
      2.3.1. FFT Algorithm Reference 4
      2.3.2. Waveform Data 4
      2.3.3. FFT Types 4
      2.3.4. Higher-Performance FFT Algorithms 5
3. Results 7
4. Appendix A – Windows CE RFFT Performance 9
5. Appendix B – No OS RFFT Performance 11
6. Appendix C – CFFT and ICFFT Performance 15
7. Appendix D – The Effect of Multi-Tasking in Windows CE 17
8. Appendix E – Comparison with Other Windows CE Devices 19
9. Appendix F – How To Achieve Performance Improvements 21
   9.1. CPU-Level Hardware 21
   9.2. Board-Level Hardware 21
   9.3. OS Software 21
   9.4. Algorithm Software 21

Executive Summary

This document summarizes the results of experiments performed on a StrongARM-based embedded platform to estimate its computational performance under various operational conditions. Appendices provide detailed results and further analysis of the computational performance data.

Tests were performed using Fast Fourier Transform (FFT) algorithms, which are a staple of modern digital signal processing (DSP) systems. The algorithms were adapted from public domain sources and were implemented within "instrumented" software applications, so as to measure the computational speed of execution. Performance was measured in two operating system configurations: no operating system and with the Windows® CE operating system.

The experiments show that to calculate a 16384-point, fixed-point FFT on real time domain data requires approximately 62 milliseconds with no operating system and approximately 107 milliseconds with Windows CE. A 16384-point, fixed-point inverse FFT on complex frequency domain data requires approximately 127 milliseconds with no operating system and approximately 302 milliseconds with the Windows CE operating system.

1. Introduction

Many embedded applications require the use of DSP algorithms to analyze waveforms, in both real-time and off-line situations. As FFTs are a representative DSP algorithm and are typically compute-intensive, an analysis of the expected computational performance of FFTs under various usage scenarios was undertaken. The results compare and contrast fixed versus floating point FFTs of various sizes and types, and under two operating system configurations. Depending on the performance requirements of your application, you may or may not need additional hardware "acceleration" (e.g., incorporation of a programmable DSP chip on a daughtercard) in order to meet your timing goals.

This document details the experiments undertaken, an analysis of the results, and some general conclusions. Appendices provide more detailed analysis of the results and some recommendations on how to improve performance. The appendices additionally include performance comparisons with other Windows CE devices, and with the hardware platform under various Windows CE configurations.

Please keep in mind that the results shown are for comparison purposes only. Your own results may vary as a function of many factors.

©InHand Electronics, Inc.

2. Experimental Setup

This section details the hardware platform, operating system configuration, and FFT algorithms that were used in the experiments.

2.1. Hardware Platform

A hardware platform with an Intel® SA-1100 (StrongARM®) CPU operating at a speed of 206 MHz and with 16MB of 70ns asynchronous DRAM was used for the experiments.

2.2. Operating System (OS) Configurations

Tests were performed with the hardware platform in an embedded, no OS configuration and in a Windows CE OS configuration.

2.2.1. No OS

This configuration provides insight into the raw computational performance of the StrongARM CPU. Without an OS, with interrupts turned off, and by executing completely out of DRAM, no software overhead is at issue and the results show the relative effects of FFT size and CPU features (e.g., the size of the data cache) on computational performance.

The software for this configuration was created in the ARM® Software Development Toolkit (SDT), version 2.50, and was tested via the ARM Debugger for Windows. Communications with the hardware platform during debug and analysis were accomplished via the Angel Debug Monitor.

2.2.2. Windows CE

This configuration provides insight into the computational performance of the hardware platform, when using a well-known third-party OS. For this configuration, a Flash-based build of Windows CE 2.11 was created. OS overhead, such as timer ticks, task-switching, and other hardware interrupts was minimized by raising the thread priority of the experimental software to the highest priority in the OS, effectively locking out other threads. Timer tick interrupts still occurred (at 25ms intervals), but other hardware interrupts (e.g., touch-screen, etc.) were minimized by not disturbing the hardware platform during the experiments.

The software for this configuration was created in Microsoft's Visual C++ development package (version 6.0), with the Windows CE Toolkit for Visual C++ add-on. Communications with the hardware platform during debug and analysis were accomplished by having the software send timing results to a DRAM-based file.

2.3. Test Software

The test software was written in the C programming language. The same code base was used for both the no OS and the Windows CE configurations, with macros used to selectively add/remove code for the different configurations.

Additionally, tests were performed on both floating-point and fixed-point versions of the algorithms. Again, macros were used to selectively add/remove code for the fixed- and floating-point versions. The fixed-point versions stored inputs, outputs, and intermediate results in 32-bit memory locations, to preserve accuracy.

Each test was performed by repeatedly calling the appropriate FFT function, bracketed by high-resolution (microsecond accuracy) timing measurements. Except where noted, the FFT function was tested a total of ten times for each scenario, and minimum, mean, and maximum timing measurements were stored.

2.3.1. FFT Algorithm Reference

The FFT algorithm was derived from software published along with the following reference book:

> *C Language Algorithms for Digital Signal Processing*, by Paul Embree & Bruce Kimble, copyright 1991, Prentice Hall.

The book provides radix-2 FFTs and inverse FFTs, along with a modified algorithm for doing the FFT of real data (zero imaginary component).

2.3.2. Waveform Data

The waveform data was simulated. A series of four sine waveforms of different frequencies were combined and scaled to a 16-bit fixed-point representation for the input. The data was then stored in an array containing 32-bit values. No filters or windowing functions were applied to the input data.

2.3.3. FFT Types

Three types of FFT tests were employed: an FFT on real data, an FFT on complex data, and an inverse FFT on complex data. All algorithms were in-place, meaning that they replaced the input data with the output data.

Except where noted in the results, all algorithms were entirely written in the C programming language, assumed the input array was already in memory, pre-calculated the W array (twiddle factors), and performed sorting of the bit-reversed output. The results of the algorithms were provided in complex format, and the algorithms did not directly calculate the magnitude, log magnitude, or phase of the result, though these naturally could be derived from complex output.

2.3.3.1. CFFT (complex FFT)

This algorithm implements a standard radix-2 FFT, with a pre-calculated W array. It performs a degenerate butterfly with no multiplications on the first iteration of each main loop, followed by complete butterflies for remaining iterations. After the FFT is calculated, it is in bit-reversed order in the array, so the array is sorted to provide output in ascending frequency order.

2.3.3.2. RFFT (real FFT)

This algorithm is a modification of the standard CFFT algorithm specified above. It is an optimization for the real input sequences (no imaginary component to the input) that are common in analysis of time domain waveforms. The optimization consists of performing the CFFT on the sequence, treating the even values in the input array as real and the odd values as imaginary. This results in an FFT of N/2 complex points, for an input array of length N real points. After the FFT is performed, the result is recombined using a series of pre-calculated coefficients.

2.3.3.3. ICFFT (inverse complex FFT)

This algorithm is identical to the standard CFFT algorithm described above, with a simple sign change in the W array resulting in the inverse FFT being calculated. The bit-reversed result is sorted, and a scaling factor is applied to the final result.

2.3.4. Higher-Performance FFT Algorithms

No representations are made that these algorithms are optimal relative to other public domain algorithms. Many books and Internet-based sources for FFT algorithms exist, including – but not limited to – the following:

- Fastest Fourier Transform in the West (FFTW): http://www.fftw.org
- Signal Processing Using C++ (SPUC): http://spuc.webjump.com
- Vector/Signal/Image Processing Library (VSIPL): http://www.vsipl.org

3. Results

The following table summarizes the results for a 16384-point FFT, using the CFFT, RFFT, and ICFFT (except where noted, all results are the maximum time required to perform the FFT algorithm):

OS	Math	RFFT (ms)	CFFT (ms)	ICFFT (ms)	Notes
N/A	Fixed	59	116	116	Without re-ordering/scaling
N/A	Fixed	62	122	127	
N/A	Fixed	51	102	105	Butterfly acceleration
N/A	Float	153	266	273	
Windows CE	Fixed	101	251	328	Without re-ordering/scaling
Windows CE	Fixed	107	287	302	
Windows CE	Float	785	1549	1659	

Several general trends emerge. As expected, the RFFT requires about half as many calculations as the CFFT, since the RFFT is actually the CFFT on a data set of half the size. Additionally, the CFFT and ICFFT require about the same number of calculations, since the algorithms are identical except that the ICFFT adds a scaling factor calculation. The floating-point versions execute much more slowly than the fixed-point versions, with floating-point execution on Windows CE especially slow, relative to embedded floating-point execution. This result is probably due to limited optimization of the Windows CE floating point libraries for the StrongARM, whereas the embedded floating point libraries developed by ARM are likely more highly optimized.

For the versions with scaling (for the ICFFT) and bit-reverse reordering disabled, the results are slightly improved. As some implementations of FFTs may not require bit reversal (e.g., performing an FFT and then performing an inverse FFT on the same data results in the final result being in the same sort order as the original input), this reduction in computation time may be appropriate.

One test on the embedded version replaced the C software of certain FFT butterfly calculations with accelerated assembly code that takes advantage of ARM's built-in instructions for 32x32 multiplies and 32x32 multiply-accumulates. These instructions yield 64-bit results. The results are significantly improved, pointing to the possibility that a finely tuned algorithm written in assembly language would provide better results. Also, in the future, when the Windows CE software development tools provide easier incorporation of assembly language code, assembly language acceleration of the FFT can be tested under Windows CE as well.

In comparing the general performance of the no OS algorithms versus those performed under Windows CE, the Windows CE versions are typically 2-3 times slower (with the exception of floating-point versions, which are markedly slower). The reduction in computational performance is most likely a direct result of the overhead of a multi-tasking OS and the relative ability of the ARM SDT and Visual C++ compilers to perform speed optimizations.

In addition, in a real-world application, where the user is interacting with the touchscreen, acquiring data from a PCMCIA card, displaying waveforms, etc., results will vary.

4. Appendix A – Windows CE RFFT Performance

The Windows CE results for RFFTs at various point sizes are:

N	Fixed (ms)	Float (ms)
1024	3.097	33.215
2048	7.852	74.954
4096	21.948	167.511
8192	48.341	363.011
16384	106.893	785.254

As the radix-2 FFT algorithm is an O[N * LOG(N)] process, at successively larger point sizes, we expect to see an increase in the length of computation of slightly greater than a factor of two. This result is indeed seen in the table above.

The fact that the floating-point RFFT takes 8-10 times longer to calculate for a given point size indicates that it should be used with great caution. If your application requires high precision calculations, you might consider the 32-bit fixed-point results shown here as adequate. If not, Windows CE does have provision for a 64-bit integer, which can greatly improve accuracy without incurring floating-point processing overhead.

Notwithstanding the relatively poor floating-point performance, the ability to implement a 1024-point fixed-point FFT in about three milliseconds does open up the possibility of performing FFTs in real-time, side-by-side with data acquisition. At three milliseconds per FFT, the resulting 330 FFTs per second of 1024 points each translates into a theoretical sampling rate of over 300ksps. Even with a conservative estimate of data acquisition overhead and other OS factors, real-time acquisition and FFT analysis at rates approaching100 ksps seems likely.

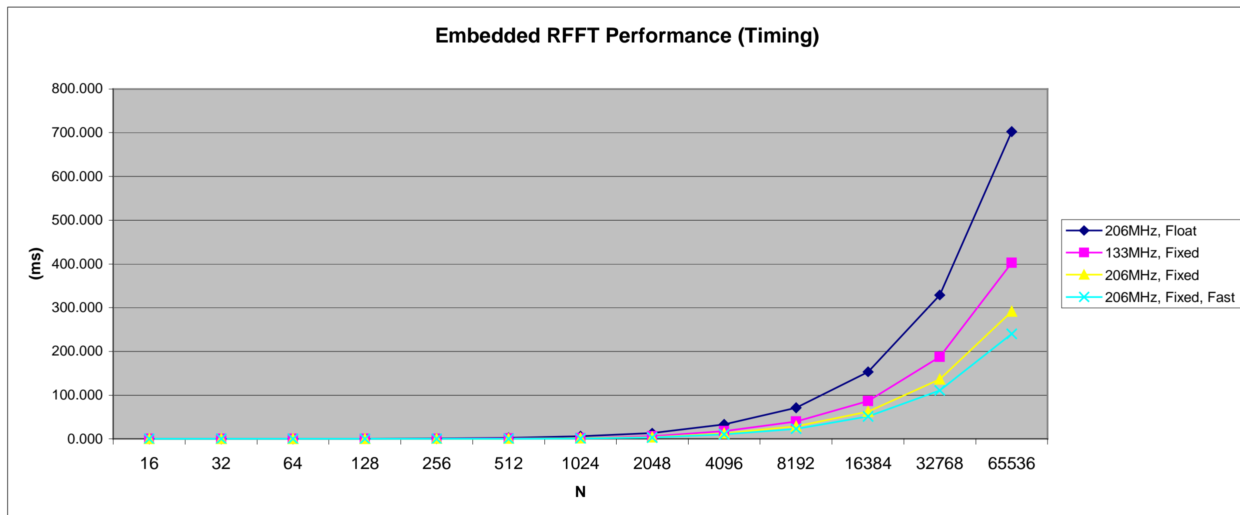
©InHand Electronics, Inc.

5. Appendix B – No OS RFFT Performance

The following table shows the no OS results (in milliseconds) for RFFTs at various point sizes:

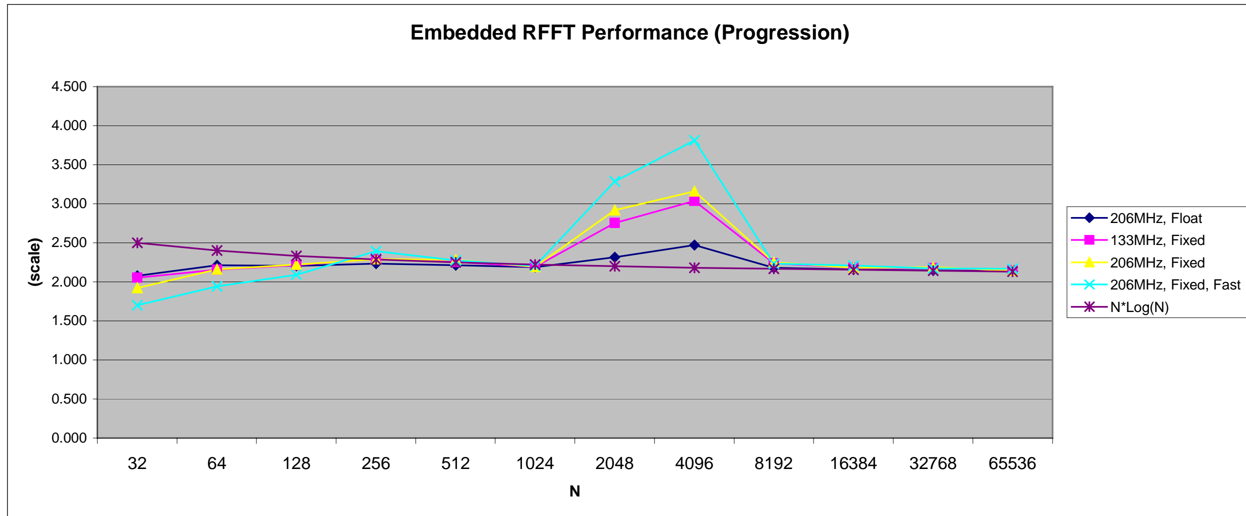
N	Fixed w/ ASM (ms)	Fixed (ms)	Float (ms)
16	0.010	0.013	0.052
32	0.017	0.025	0.108
64	0.033	0.054	0.239
128	0.069	0.120	0.527
256	0.165	0.275	1.177
512	0.375	0.628	2.605
1024	0.826	1.376	5.696
2048	2.716	4.016	13.194
4096	10.349	12.688	32.588
8192	23.115	28.568	71.019
16384	51.037	62.214	153.373
32768	110.768	135.981	329.046
65536	239.935	291.413	702.389

The results are graphed below.



The first column results are for those tests in which the FFT butterfly was augmented by replacing the C language code with ARM assembly instructions that perform 32-bit multiplications. The second column results are directly comparable with those for Windows CE in that similar code is used to execute the algorithms. The difference in performance is mainly a result of the overhead of the OS, though the compiler's ability to optimize code also plays a factor. The third column floating point results, while several times slower than the fixed point results, are still markedly faster than those for Windows CE. This is most likely due to poorly optimized floating point libraries for the Windows CE compilers.

One interesting factor in the fixed-point results is the role of the StrongARM's 8KB data cache. Notice in the graph below how the length of time that the algorithm requires at low values of N generally increases by a factor of two at low values of N (1024 points and lower) and at high values of N (4096 points and higher).



However, in the range 1024-4096 points, the length of time required to calculate the RFFT increases at a much greater rate (about a factor of three). This is the result of the data cache size. At low values of N, the data for the algorithm can be entirely contained within the cache. Since no OS is present to cause context-switches and cache misses, the cache is essentially completely at the algorithm's disposal. Thus, the effective rate at which data is retrieved from memory is governed by the cache access speed (about 5ns at a 206MHz CPU clock speed). As N gets larger, the cache can no longer completely contain the data required for the algorithm. As such, cache misses start occurring and some data must be retrieved from DRAM at a much slower access speed (about 30ns for in-page accesses).

In FFT algorithms which typically "jump" throughout the data at varying increments, the number of cache misses can be relatively large. Thus, as N ranges between 1024-4096 points, the effect of cache misses is seen in the corresponding increase in length of computation. After 4096 points though, the algorithm becomes largely governed by DRAM accesses, which is why it stabilizes again and increases in computational time again are approximately a factor of two.

This analysis is further shown by analyzing the table below, which compares algorithm performance when the CPU is operating at 133MHz versus 206MHz.

N	133MHz Fixed (ms)	206MHz Fixed (ms)
16	0.019	0.013
32	0.039	0.025
64	0.084	0.054
128	0.186	0.120
256	0.427	0.275
512	0.963	0.628
1024	2.111	1.376
2048	5.813	4.016
4096	17.645	12.688
8192	39.497	28.568
16384	86.119	62.214
32768	187.579	135.981
65536	402.467	291.413

Again, you can see that between 1024-4096 points, the increase in computational performance for the 133MHz version is approximately a factor of three. Additional analysis reveals that when the algorithm is largely in-cache (1024 points and lower), the increase in performance for the 206MHz version over the 133MHz version is about 33%, as would be expected, given the relative clock speeds. Once the algorithm is governed by DRAM accesses (4096 points and above), the increase in performance is lower (about 25%) which is due to fact that DRAM accesses require a fixed amount of time (about 30ns), regardless of the CPU speed.

The implications of this result are appropriate for both the no OS configuration and the Windows CE configuration. In general, an FFT algorithm will perform at a much higher rate when it is accessing data from on-chip memories. Of course, this is why programmable DSP chips typically have lots of on-chip memory. Analogously, as future CPUs used in the hardware platform incorporate larger data caches, it can be expected that performance on data-intensive algorithms such as the FFT will be boosted as well.

©InHand Electronics, Inc.

6. Appendix C – CFFT and ICFFT Performance

The following table shows CFFT performance at various point sizes for the Windows CE configuration.

N	Fixed (ms)	Float (ms)
1024	6.526	53.263
2048	19.221	125.322
4096	42.869	279.844
8192	95.615	620.226
16384	286.893	1548.999

The following table shows ICFFT performance at various point sizes for the Windows CE configuration.

N	Fixed (ms)	Float (ms)
1024	6.891	61.717
2048	20.005	142.186
4096	44.506	316.544
8192	98.995	698.792
16384	301.834	1659.290

The following table shows CFFT performance at various point sizes for the no OS configuration.

N	Fixed w/ ASM (ms)	Fixed (ms)	Float (ms)
16	0.012	0.015	0.048
32	0.024	0.036	0.119
64	0.053	0.085	0.291
128	0.119	0.199	0.699
256	0.279	0.472	1.648
512	0.627	1.073	3.799
1024	2.127	3.100	9.364
2048	9.267	11.155	24.797
4096	21.023	25.218	55.361
8192	46.698	55.886	121.901
16384	102.370	122.396	266.015
32768	222.519	265.859	576.358
65536	478.979	575.760	1238.643

The following table shows ICFFT performance at various point sizes for the no OS configuration.

N	Fixed w/ ASM (ms)	Fixed (ms)	Float (ms)
16	0.014	0.019	0.065
32	0.028	0.041	0.144
64	0.058	0.092	0.335
128	0.130	0.211	0.769
256	0.305	0.497	1.785
512	0.678	1.120	4.054
1024	2.285	3.357	9.832
2048	9.682	11.602	25.894
4096	21.731	26.159	57.308
8192	48.297	57.804	126.119
16384	105.166	126.505	273.414
32768	228.117	274.359	590.546
65536	493.441	591.090	1271.303

©InHand Electronics, Inc.

7. Appendix D – The Effect of Multi-Tasking in Windows CE

One of the reasons that a multi-tasking OS such as Windows CE slows computational performance (relative to no OS at all) is the effect of time-slicing and thread prioritization. A multi-tasking OS typically has a timer interrupt that occurs many times per second and causes the OS kernel to switch tasks from one thread to another. This switching – or time-slicing – allows a device to appear to be doing many things at a time, which often results in a more pleasurable user experience. Ironically, this perceived improvement in performance actually reduces performance on any single important thread, as it must share CPU time with other threads. A real-time OS can reduce or eliminate the effect of time-slicing by ensuring that high priority tasks always complete before low priority tasks are given CPU time.

Windows CE gives the real-time software developer some control over thread execution, by allowing the setting of thread priorities. All threads start out with a standard priority level, but threads can be boosted in priority. Boosting to the highest priority has the effect of locking out all other threads. In the FFT tests, other threads are effectively locked out by boosting the priority of the FFT thread to the highest in the system and also by not disturbing the hardware platform during tests (e.g., no touch-screen interrupts, etc.). Thus, while timer interrupts still occur and cause the kernel to perform some time-slicing evaluation, since the FFT thread is the highest priority, it wins any contest for CPU time.

In the tests, with the highest thread priority, ten iterations of each FFT were performed and it was generally found that the variation between the minimum and maximum FFT calculation time was negligible. This result indicates that the kernel is indeed giving all CPU time to the FFT thread. To characterize performance relative to other thread priorities, tests were done comparing the performance of the FFT algorithms when executed within a normal priority thread (in this case, the test was iterated 100 times). The maximum times required to complete the RFFT are as shown below:

N	**Fixed with Highest Priority (ms)**	**Fixed with Normal Priority (ms)**	**Float with Highest Priority (ms)**	**Float with Normal Priority (ms)**
1024	3.097	3.106	33.215	52.396
2048	7.852	7.832	74.954	77.516
4096	21.948	21.998	167.511	202.640
8192	48.341	48.930	363.011	438.262
16384	106.893	110.377	785.254	874.019

The results show the significant impact that thread priority has on execution time, especially for longer executing algorithms. As the execution time gets longer, more opportunities for thread interruption occur.

The average times required to complete the RFFT are as shown below:

N	Fixed with Highest Priority (ms)	Fixed with Normal Priority (ms)	Float with Highest Priority (ms)	Float with Normal Priority (ms)
1024	3.089	3.070	33.203	33.524
2048	7.831	7.792	74.920	75.307
4096	21.930	21.963	167.462	168.263
8192	48.320	48.452	362.923	364.708
16384	106.653	107.274	785.182	790.268

These results indicate that even though thread priority has a significant impact on the bounded maximum time required to complete the RFFT, the average time does not change appreciably.

Another potential impact on performance is the time interval between timer ticks, which is the time at which the kernel reevaluates threads for time-slicing. To evaluate the effect of timer tick interval, the results for 25ms intervals (the standard used in the tests) were compared with results with a timer interval of 5ms (all at highest thread priority). The maximum required time to complete the RFFT are shown in the table below:

N	Fixed with 25ms Timer Tick (ms)	Fixed with 5ms Timer Tick (ms)	Float with 25ms Timer Tick (ms)	Float with 5 ms Timer Tick (ms)
1024	3.097	3.094	33.215	33.261
2048	7.852	7.894	74.954	75.103
4096	21.948	22.024	167.511	167.689
8192	48.341	48.534	363.011	363.423
16384	106.893	107.323	785.254	786.764

Interestingly, increasing the frequency of timer interrupts does not appreciably increase the RFFT execution time.

8. Appendix E – Comparison with Other Windows CE Devices

The performance of the hardware platform on the RFFT was compared to the performance of the same software on a Cassiopiea® E-105 palm-size PC. The E-105 is a palm-size PC manufactured by Casio®, containing a MIPS® R4000 CPU running at 131MHz, 32MB of DRAM, and Windows CE 2.11. The E-105 has a timer interval of 5ms, while our hardware platform has a timer interval of 25ms (RFFT results for 5ms timer intervals on the hardware platform are detailed in a prior Appendix).

Results for the RFFT are shown in the table below:

N	InHand Platform, Fixed (ms)	E-105, Fixed (ms)*	InHand Platform, Float (ms)	E-105, Float (ms)*
1024	3.097	5.000	33.215	40.000
2048	7.852	10.000	74.954	75.000
4096	21.948	40.000	167.511	185.000
8192	48.341	100.000	363.011	405.000
16384	106.893	235.000	785.254	895.000

* - The E-105 does not make the high-performance timer (QueryPerformanceCounter function) available to the application developer, so results are based on the main timer interval (granularity of 5ms).

The InHand platform clearly outperforms the E-105. Little is known of the internals of Windows CE on the E-105 (e.g., other tasks executing) and of the hardware on the E-105 (e.g., DRAM type, Flash memory type, etc.). Hardware and software modifications on both the InHand platform and the E-105 could affect results.

 ©InHand Electronics, Inc.

9. Appendix F – How To Achieve Performance Improvements

Performance can be improved by focusing on various hardware and software subsystems. Some of these improvements are beyond the hardware platform's direct control, and must be implemented by the manufacturers of the various subsystems.

9.1. CPU-Level Hardware

An increase in the size of the data cache will have the most impact on performance, especially for larger FFTs. An increase in the CPU clock speed also will improve computational performance. On small FFTs that can be contained within the data cache, the increase will generally scale with the CPU clock speed. As the FFT becomes large enough to cause cache misses, the improvements also will be governed by the speed of DRAM accesses.

Note that the program cache also has an impact on performance, but since the FFT algorithm is rather small (less than 1.5KB), relative to the size of the program cache (16KB), increases in its size will have minimal impact, unless an OS is switching between many threads of execution.

9.2. Board-Level Hardware

An increase in the access speed of the DRAM will have a great impact on performance for large FFTs that cannot be completely contained in the data cache. Currently, the in-page access time for DRAM is about 30ns, which is about six times slower than that for the cache (when the CPU is operating at 206MHz). Using higher-speed DRAM and synchronous DRAM will make a significant difference.

Additionally, limiting "traffic" on the main memory bus will improve performance for large FFTs. Congestion from devices such as LCD controllers (the SA-1100 LCD controller stores the frame buffer in DRAM) can decrease memory bandwidth.

9.3. OS Software

A reduction in task-switching overhead will improve performance. One way to accomplish this is to reduce the frequency of the timer tick that is used for time-slicing (currently 25 ms). The negative impact of reducing this frequency is that the hardware platform may become less responsive to user input. Additionally, interrupts for other devices (e.g., touchscreen, PCMCIA) could be selectively turned off to eliminate their impact on performance.

9.4. Algorithm Software

Improvements in performance for the algorithm used for these tests are most likely to appear by fine-tuning it in assembly language, taking advantage of the ARM instructions for fixed-point multiplication. The algorithm also can be improved, depending on the amount of fixed point "headroom" that is needed for accuracy. For example, if the input data is only 8-bits, the use of 16-bit numbers to store intermediate results might improve performance over storing 32-bit numbers. Modifying the algorithm for radix-4 or radix-8 calculations will also improve performance by reducing the number of arithmetic operations required.

Additionally, a further analysis of the many other FFT algorithms that exist may be prudent to determine the best one for a given application. For example, ARM provides a DSP library that includes an optimized FFT algorithm in source code format.

 ©InHand Electronics, Inc.