ABSTRACT

This application report contains the results from benchmarking the MSP430 against microcontrollers from other vendors. IAR Embedded Workbench™ development platform was used to build and execute, in simulation mode, a set of simple math functions. These functions were executed on each microcontroller to benchmark different aspects of the microcontrollers' performance. In addition, both Dhrystone and Whetstone analyses have been included.

1 Embedded Benchmark Suite

This section has results for simple and less intense math functions. Figure 1 shows the total code size in bytes for each microcontroller with no optimization and with full optimization.

Figure 1. Total Code Size for Embedded Benchmark Suite
Figure 2 shows the total cycle count for each microcontroller with no optimization and with full optimization. Note that some architectures use an internal CPU clock divider. In these architectures, the total execution time for the code is the clock divider multiplied by the total instruction cycle count. This clock divider is not included in the total cycle count numbers presented here. See Appendix A.1 for more information regarding CPU clock dividers.

![Bar Chart]

Figure 2. Total Instruction Cycles for Embedded Benchmark Suite

The MSP430FG4619 differs in architecture from the MSP430F149 and has the MSP430X CPU. The MSP430X CPU can address up to 1-MB address range without paging. In addition, the MSP430X CPU has fewer interrupt overhead cycles and fewer instruction cycles, in some cases, than the MSP430 CPU. The MSP430X CPU is completely backward compatible with the MSP430 CPU. Code size and cycle count values are shown in Appendix A.
Table 1 shows the total code size and the total instruction counts for each microcontroller, normalized against the MSP430FG4619, for the Embedded Benchmark Suite.

Table 1. Normalized Results for Embedded Benchmark Suite

<table>
<thead>
<tr>
<th>MICROCONTROLLER</th>
<th>TOTAL CODE SIZE</th>
<th>TOTAL INSTRUCTION CYCLE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNOPTIMIZED</td>
<td>FULLY OPTIMIZED</td>
</tr>
<tr>
<td></td>
<td>MSP430FG4619</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>MSP430F149</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>PIC24FJ128GA</td>
<td>1.80</td>
</tr>
<tr>
<td></td>
<td>PIC18F242</td>
<td>1.75</td>
</tr>
<tr>
<td></td>
<td>8051</td>
<td>2.53</td>
</tr>
<tr>
<td></td>
<td>H8/300H</td>
<td>1.94</td>
</tr>
<tr>
<td></td>
<td>MaxQ20</td>
<td>1.38</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>HCS12</td>
<td>1.40</td>
</tr>
<tr>
<td></td>
<td>ATmega8</td>
<td>1.49</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FULLY OPTIMIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSP430FG4619</td>
</tr>
<tr>
<td></td>
<td>MSP430F149</td>
</tr>
<tr>
<td></td>
<td>PIC24FJ128GA</td>
</tr>
<tr>
<td></td>
<td>PIC18F242</td>
</tr>
<tr>
<td></td>
<td>8051</td>
</tr>
<tr>
<td></td>
<td>H8/300H</td>
</tr>
<tr>
<td></td>
<td>MaxQ20</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI</td>
</tr>
<tr>
<td></td>
<td>HCS12</td>
</tr>
<tr>
<td></td>
<td>ATmega8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>UNOPTIMIZED</th>
<th>FULLY OPTIMIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSP430FG4619</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>MSP430F149</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>PIC24FJ128GA</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>PIC18F242</td>
<td>5.79</td>
</tr>
<tr>
<td></td>
<td>8051</td>
<td>5.64</td>
</tr>
<tr>
<td></td>
<td>H8/300H</td>
<td>3.22</td>
</tr>
<tr>
<td></td>
<td>MaxQ20</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>HCS12</td>
<td>2.49</td>
</tr>
<tr>
<td></td>
<td>ATmega8</td>
<td>1.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FULLY OPTIMIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MSP430FG4619</td>
</tr>
<tr>
<td></td>
<td>MSP430F149</td>
</tr>
<tr>
<td></td>
<td>PIC24FJ128GA</td>
</tr>
<tr>
<td></td>
<td>PIC18F242</td>
</tr>
<tr>
<td></td>
<td>8051</td>
</tr>
<tr>
<td></td>
<td>H8/300H</td>
</tr>
<tr>
<td></td>
<td>MaxQ20</td>
</tr>
<tr>
<td></td>
<td>ARM7TDMI</td>
</tr>
<tr>
<td></td>
<td>HCS12</td>
</tr>
<tr>
<td></td>
<td>ATmega8</td>
</tr>
</tbody>
</table>

Appendix B includes the code names and a brief description of their functionality used for this benchmarking.
In order to exhibit the performance of each of the microcontrollers under intense math operations, the benchmarking of a Finite Impulse Response (FIR) filter that requires multiply and accumulate (MAC) is included in this report. Also included are the results of the Dhrystone and Whetstone benchmarks. Code size and cycle count values are shown in Appendix A.

Figure 3 shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of an FIR filter.
Figure 4 shows the cycle count for each microcontroller, with no optimization and full optimization, for the implementation of an FIR filter.

**Figure 4. Cycle Count For FIR Filter Operation**

Table 2 shows the total code size and the total instruction cycle count for each microcontroller, normalized against the MSP430FG4619, for the FIR filter operation.

**Table 2. Normalized Results for FIR Filter Operation**

<table>
<thead>
<tr>
<th>MICROCONTROLLER</th>
<th>TOTAL CODE SIZE</th>
<th>TOTAL INSTRUCTION CYCLE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNOPTIMIZED</td>
<td>FULLY OPTIMIZED</td>
</tr>
<tr>
<td>MSP430FG4619</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>1.006</td>
<td>1.006</td>
</tr>
<tr>
<td>PIC24FJ128GA</td>
<td>1.61</td>
<td>1.52</td>
</tr>
<tr>
<td>PIC18F242</td>
<td>2.02</td>
<td>1.99</td>
</tr>
<tr>
<td>8051</td>
<td>2.08</td>
<td>2.04</td>
</tr>
<tr>
<td>H8/300H</td>
<td>1.41</td>
<td>1.38</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>1.56</td>
<td>1.47</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>1.52</td>
<td>1.52</td>
</tr>
<tr>
<td>HCS12</td>
<td>1.91</td>
<td>1.90</td>
</tr>
<tr>
<td>ATmega8</td>
<td>1.33</td>
<td>1.35</td>
</tr>
</tbody>
</table>
The Dhrystone benchmark gauges the performance of a microcontroller in handling pointers, structures, and strings. Figure 5 shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of this code.

Figure 5. Code Size In Bytes For Dhrystone Analysis
Figure 6 shows the cycle count for each microcontroller, with no optimization and full optimization, for the Dhrystone analysis.

![Cycle Count for Dhrystone Analysis](image)

**Figure 6. Cycle Count For Dhrystone Analysis**

Table 3 shows the total code size and the total instruction cycle count for each microcontroller, normalized against the MSP430FG4619, for the Dhrystone analysis.

**Table 3. Normalized Results for Dhrystone Analysis**

<table>
<thead>
<tr>
<th>MICROCONTROLLER</th>
<th>TOTAL CODE SIZE</th>
<th>TOTAL INSTRUCTION CYCLE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNOPTIMIZED</td>
<td>FULLY OPTIMIZED</td>
</tr>
<tr>
<td></td>
<td>UNOPTIMIZED</td>
<td>FULLY OPTIMIZED</td>
</tr>
<tr>
<td>MSP430FG4619</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>1.03</td>
<td>1.03</td>
</tr>
<tr>
<td>PIC24FJ128GA</td>
<td>1.40</td>
<td>1.51</td>
</tr>
<tr>
<td>PIC18F242(1)</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>8051</td>
<td>2.42</td>
<td>2.32</td>
</tr>
<tr>
<td>H8/300H</td>
<td>1.71</td>
<td>1.92</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>1.88</td>
<td>1.98</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>1.27</td>
<td>1.19</td>
</tr>
<tr>
<td>HCS12</td>
<td>0.98</td>
<td>1.07</td>
</tr>
<tr>
<td>ATmega8</td>
<td>1.74</td>
<td>1.76</td>
</tr>
</tbody>
</table>

(1) The 30-day trial version of the IAR compiler did not support the memory model required for Dhrystone analysis.
The Whetstone benchmark attempts to measure the performance of both integer and floating-point arithmetic in a variety of scientific functions. The code has a mixture of C functions to calculate the sine, cosine, exponent, etc., of fixed-point and floating-point numbers. Figure 7 shows the code size for each microcontroller, with no optimization and full optimization, for the implementation of this code.

Figure 7. Code Size In Bytes For Whetstone Analysis
Figure 8 shows the cycle count for each microcontroller, with no optimization and full optimization, for the Whetstone analysis.
Table 4 shows the total code size and the total instruction counts for each microcontroller, normalized against the MSP430FG4619, for the Whetstone analysis.

**Table 4. Normalized Results for Whetstone Analysis**

<table>
<thead>
<tr>
<th>MICROCONTROLLER</th>
<th>TOTAL CODE SIZE</th>
<th>TOTAL INSTRUCTION CYCLE COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNOPTIMIZED</td>
<td>FULLY OPTIMIZED</td>
</tr>
<tr>
<td>MSP430FG4619</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>PIC24FJ128GA</td>
<td>0.74</td>
<td>0.70</td>
</tr>
<tr>
<td>PIC18F242(1)</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>8051</td>
<td>1.40</td>
<td>1.33</td>
</tr>
<tr>
<td>H8/300H</td>
<td>0.72</td>
<td>0.71</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>0.98</td>
<td>0.82</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>1.52</td>
<td>1.61</td>
</tr>
<tr>
<td>HCS12</td>
<td>1.09</td>
<td>1.13</td>
</tr>
<tr>
<td>ATmega8</td>
<td>1.07</td>
<td>0.72</td>
</tr>
</tbody>
</table>

(1) The 30-day trial version of the IAR compiler did not support the memory model required for Whetstone analysis.

Appendix B includes the code and a brief description of functionality used for this benchmarking.
Appendix A Background Information

This appendix includes the actual values for all of the benchmarking discussed in this report.

A.1 Processor Clock vs Instruction Cycle Clock Considerations

MCU architectures have different associations between the processor input clock frequency and the actual instruction cycle clock frequency. Ideally, one processor clock cycle fed into the CPU would result in one instruction being executed. However, in some cases, an additional CPU internal clock divider is used (Table 5). Then, multiple processor clock cycles are necessary to execute a single instruction. This is important to consider when determining the system clock frequency that is needed to achieve a given task. Note that higher clock frequencies generally also lead to a higher power consumption due to increased CMOS logic switching losses.

Table A-1. Table 5. CPU Clock Divider

<table>
<thead>
<tr>
<th>Microcontroller</th>
<th>CPU Clock Divider</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430FG4619</td>
<td>1</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>1</td>
</tr>
<tr>
<td>Microchip PIC24FJ128GA</td>
<td>2</td>
</tr>
<tr>
<td>Microchip PIC18F242</td>
<td>4</td>
</tr>
<tr>
<td>Generic 8051</td>
<td>1...12(1)</td>
</tr>
<tr>
<td>Renesas H8/300H</td>
<td>2</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>1</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>1</td>
</tr>
<tr>
<td>Freescale HCS12</td>
<td>2</td>
</tr>
<tr>
<td>Atmel ATmega8</td>
<td>1</td>
</tr>
</tbody>
</table>

(1) 8051 architectures typically use a divider of 12. However, some improved architectures can execute a subset of instructions in as little as one clock cycle per instruction.
A.2 Compiler Information And Detailed Results

The C compiler bundled with IAR Embedded Workbench Integrated Development Environment (IDE) was used to build the benchmarking applications. Evaluation copies of the IDE were obtained for each microcontroller from IAR Systems’ web site located at http://www.iar.com. Table A-2 lists the C compiler version used to build the benchmarking applications for each microcontroller.

All applications were built with compiler optimization set to “none” and to “full”. This was done to utilize the compiler’s ability to build efficient code, which has had a great impact on the results.

Table A-2. C Compiler Versions

<table>
<thead>
<tr>
<th>Microcontroller</th>
<th>IAR C Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSP430FG4619</td>
<td>3.41A</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>3.41A</td>
</tr>
<tr>
<td>Microchip PIC24FJ128GA</td>
<td>2.02&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>Microchip PIC18F242</td>
<td>3.10A</td>
</tr>
<tr>
<td>Generic 8051</td>
<td>7.20C</td>
</tr>
<tr>
<td>Renesas H8/300H</td>
<td>1.53I</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>1.13C</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>4.31A</td>
</tr>
<tr>
<td>Freescale HCS12</td>
<td>3.10A</td>
</tr>
<tr>
<td>Atmel ATmega8</td>
<td>4.12A</td>
</tr>
</tbody>
</table>

<sup>(1)</sup> For this device, the current Microchip MPLAB C30 compiler was used. An IAR compiler for the PIC24x was not available at the time of publishing this application note.

The following pages include the actual values for all of the benchmarking discussed in this report. Table A-3 and Table A-4 show the code size in bytes for each of the microcontrollers for every math operation without optimization and with full optimization, respectively.
### Table A-3. Code Size In Bytes Without Optimization For Simple Math Operations

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>MSP430FG4619</th>
<th>MSP430F149</th>
<th>PIC24FJ128GA</th>
<th>PIC18F242</th>
<th>8051</th>
<th>H8/300H</th>
<th>MaxQ20</th>
<th>ARM7TDMI</th>
<th>HCS12</th>
<th>ATmega8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit Math</td>
<td>236</td>
<td>232</td>
<td>345</td>
<td>174</td>
<td>266</td>
<td>400</td>
<td>326</td>
<td>684</td>
<td>95</td>
<td>152</td>
</tr>
<tr>
<td>8-Bit Matrix</td>
<td>120</td>
<td>126</td>
<td>450</td>
<td>368</td>
<td>499</td>
<td>492</td>
<td>348</td>
<td>416</td>
<td>217</td>
<td>394</td>
</tr>
<tr>
<td>8-Bit Switch</td>
<td>220</td>
<td>200</td>
<td>486</td>
<td>238</td>
<td>305</td>
<td>498</td>
<td>200</td>
<td>532</td>
<td>197</td>
<td>378</td>
</tr>
<tr>
<td>16-Bit Math</td>
<td>176</td>
<td>172</td>
<td>333</td>
<td>266</td>
<td>478</td>
<td>398</td>
<td>240</td>
<td>684</td>
<td>107</td>
<td>210</td>
</tr>
<tr>
<td>16-Bit Matrix</td>
<td>136</td>
<td>166</td>
<td>558</td>
<td>634</td>
<td>693</td>
<td>572</td>
<td>460</td>
<td>432</td>
<td>301</td>
<td>532</td>
</tr>
<tr>
<td>16-Bit Switch</td>
<td>220</td>
<td>198</td>
<td>480</td>
<td>342</td>
<td>519</td>
<td>534</td>
<td>186</td>
<td>532</td>
<td>215</td>
<td>424</td>
</tr>
<tr>
<td>32-Bit Math</td>
<td>272</td>
<td>268</td>
<td>462</td>
<td>486</td>
<td>1050</td>
<td>646</td>
<td>316</td>
<td>644</td>
<td>324</td>
<td>352</td>
</tr>
<tr>
<td>Floating-Point Math</td>
<td>1134</td>
<td>1132</td>
<td>1266</td>
<td>1322</td>
<td>2346</td>
<td>1176</td>
<td>1200</td>
<td>1868</td>
<td>2082</td>
<td>1096</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>204</td>
<td>248</td>
<td>519</td>
<td>732</td>
<td>707</td>
<td>554</td>
<td>480</td>
<td>476</td>
<td>270</td>
<td>518</td>
</tr>
<tr>
<td>Total</td>
<td>2718</td>
<td>2742</td>
<td>4899</td>
<td>4762</td>
<td>6863</td>
<td>5270</td>
<td>3756</td>
<td>6268</td>
<td>3808</td>
<td>4056</td>
</tr>
</tbody>
</table>

### Table A-4. Code Size In Bytes With Full Optimization For Simple Math Operations

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>MSP430FG4619</th>
<th>MSP430F149</th>
<th>PIC24FJ128GA</th>
<th>PIC18F242(1)</th>
<th>8051</th>
<th>H8/300H</th>
<th>MaxQ20</th>
<th>ARM7TDMI</th>
<th>HCS12</th>
<th>ATmega8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit Math</td>
<td>214</td>
<td>210</td>
<td>297</td>
<td>170</td>
<td>233</td>
<td>344</td>
<td>230</td>
<td>636</td>
<td>83</td>
<td>134</td>
</tr>
<tr>
<td>8-Bit Matrix</td>
<td>106</td>
<td>106</td>
<td>366</td>
<td>324</td>
<td>398</td>
<td>412</td>
<td>252</td>
<td>392</td>
<td>188</td>
<td>354</td>
</tr>
<tr>
<td>8-Bit Switch</td>
<td>218</td>
<td>198</td>
<td>393</td>
<td>208</td>
<td>305</td>
<td>444</td>
<td>192</td>
<td>452</td>
<td>162</td>
<td>350</td>
</tr>
<tr>
<td>16-Bit Math</td>
<td>162</td>
<td>158</td>
<td>285</td>
<td>286</td>
<td>452</td>
<td>352</td>
<td>204</td>
<td>636</td>
<td>76</td>
<td>198</td>
</tr>
<tr>
<td>16-Bit Matrix</td>
<td>110</td>
<td>130</td>
<td>465</td>
<td>692</td>
<td>504</td>
<td>482</td>
<td>328</td>
<td>396</td>
<td>262</td>
<td>434</td>
</tr>
<tr>
<td>16-Bit Switch</td>
<td>218</td>
<td>196</td>
<td>390</td>
<td>282</td>
<td>493</td>
<td>478</td>
<td>184</td>
<td>452</td>
<td>174</td>
<td>382</td>
</tr>
<tr>
<td>32-Bit Math</td>
<td>258</td>
<td>254</td>
<td>375</td>
<td>542</td>
<td>909</td>
<td>574</td>
<td>288</td>
<td>620</td>
<td>323</td>
<td>342</td>
</tr>
<tr>
<td>Floating-Point Math</td>
<td>1122</td>
<td>1120</td>
<td>1194</td>
<td>1400</td>
<td>2190</td>
<td>1104</td>
<td>1172</td>
<td>1832</td>
<td>2082</td>
<td>1088</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>176</td>
<td>220</td>
<td>426</td>
<td>676</td>
<td>536</td>
<td>482</td>
<td>398</td>
<td>428</td>
<td>219</td>
<td>490</td>
</tr>
<tr>
<td>Total</td>
<td>2584</td>
<td>2592</td>
<td>4191</td>
<td>4580</td>
<td>6020</td>
<td>4672</td>
<td>3248</td>
<td>5844</td>
<td>3569</td>
<td>3772</td>
</tr>
</tbody>
</table>

(1) For some functions, the 30-day trial version of the IAR compiler produced larger code sizes with full optimization than it did with no optimization.
Table A-5 and Table A-6 show the cycle count for each of the microcontrollers for each math operation without optimization and with full optimization, respectively.

### Table A-5. Cycle Count Without Optimization For Simple Math Operations

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>MSP430FG4619</th>
<th>MSP430F149</th>
<th>PIC24FJ128GA</th>
<th>PIC18F242</th>
<th>8051</th>
<th>H8/300H</th>
<th>MaxQ20</th>
<th>ARM7TDMI</th>
<th>HCS12</th>
<th>ATmega8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit Math</td>
<td>250</td>
<td>261</td>
<td>107</td>
<td>141</td>
<td>212</td>
<td>240</td>
<td>175</td>
<td>87</td>
<td>97</td>
<td>134</td>
</tr>
<tr>
<td>8-Bit Matrix</td>
<td>2370</td>
<td>2497</td>
<td>2927</td>
<td>7310</td>
<td>14898</td>
<td>10228</td>
<td>6196</td>
<td>2122</td>
<td>6858</td>
<td>2523</td>
</tr>
<tr>
<td>8-Bit Switch</td>
<td>33</td>
<td>32</td>
<td>76</td>
<td>49</td>
<td>112</td>
<td>96</td>
<td>42</td>
<td>51</td>
<td>51</td>
<td>39</td>
</tr>
<tr>
<td>16-Bit Math</td>
<td>223</td>
<td>233</td>
<td>108</td>
<td>332</td>
<td>542</td>
<td>254</td>
<td>201</td>
<td>102</td>
<td>108</td>
<td>288</td>
</tr>
<tr>
<td>16-Bit Matrix</td>
<td>3140</td>
<td>3270</td>
<td>3183</td>
<td>26533</td>
<td>23868</td>
<td>11252</td>
<td>9012</td>
<td>2890</td>
<td>8650</td>
<td>9506</td>
</tr>
<tr>
<td>16-Bit Switch</td>
<td>32</td>
<td>31</td>
<td>74</td>
<td>87</td>
<td>314</td>
<td>102</td>
<td>35</td>
<td>51</td>
<td>54</td>
<td>45</td>
</tr>
<tr>
<td>32-Bit Math</td>
<td>569</td>
<td>589</td>
<td>564</td>
<td>1259</td>
<td>3854</td>
<td>520</td>
<td>440</td>
<td>109</td>
<td>267</td>
<td>750</td>
</tr>
<tr>
<td>Floating-Point Math</td>
<td>771</td>
<td>795</td>
<td>789</td>
<td>1049</td>
<td>3339</td>
<td>1548</td>
<td>644</td>
<td>205</td>
<td>5508</td>
<td>1663</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>4500</td>
<td>4706</td>
<td>3203</td>
<td>32096</td>
<td>19856</td>
<td>14018</td>
<td>9624</td>
<td>3424</td>
<td>8034</td>
<td>8417</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11888</strong></td>
<td><strong>12414</strong></td>
<td><strong>11031</strong></td>
<td><strong>68856</strong></td>
<td><strong>6695</strong></td>
<td><strong>38258</strong></td>
<td><strong>26369</strong></td>
<td><strong>9041</strong></td>
<td><strong>29627</strong></td>
<td><strong>23365</strong></td>
</tr>
</tbody>
</table>

(1) The 30-day trial version of IAR compiler for some functions did produce larger numbers with full optimization, as compared to no optimization.

### Table A-6. Cycle Count With Full Optimization For Simple Math Operations

<table>
<thead>
<tr>
<th>APPLICATION</th>
<th>MSP430FG4619</th>
<th>MSP430F149</th>
<th>PIC24FJ128GA</th>
<th>PIC18F242</th>
<th>8051</th>
<th>H8/300H</th>
<th>MaxQ20</th>
<th>ARM7TDMI</th>
<th>HCS12</th>
<th>ATmega8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Bit Math</td>
<td>233</td>
<td>243</td>
<td>75</td>
<td>136</td>
<td>176</td>
<td>152</td>
<td>130</td>
<td>64</td>
<td>68</td>
<td>110</td>
</tr>
<tr>
<td>8-Bit Matrix</td>
<td>875</td>
<td>1009</td>
<td>1051</td>
<td>2193</td>
<td>2590</td>
<td>4362</td>
<td>1140</td>
<td>475</td>
<td>1559</td>
<td>984</td>
</tr>
<tr>
<td>8-Bit Switch</td>
<td>32</td>
<td>31</td>
<td>61</td>
<td>49</td>
<td>112</td>
<td>62</td>
<td>38</td>
<td>20</td>
<td>46</td>
<td>38</td>
</tr>
<tr>
<td>16-Bit Math</td>
<td>210</td>
<td>219</td>
<td>73</td>
<td>339</td>
<td>526</td>
<td>172</td>
<td>183</td>
<td>79</td>
<td>60</td>
<td>266</td>
</tr>
<tr>
<td>16-Bit Matrix</td>
<td>811</td>
<td>945</td>
<td>1115</td>
<td>6461</td>
<td>4294</td>
<td>4746</td>
<td>1508</td>
<td>475</td>
<td>2073</td>
<td>1488</td>
</tr>
<tr>
<td>16-Bit Switch</td>
<td>31</td>
<td>30</td>
<td>60</td>
<td>87</td>
<td>318</td>
<td>66</td>
<td>34</td>
<td>20</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>32-Bit Math</td>
<td>556</td>
<td>575</td>
<td>510</td>
<td>1284</td>
<td>2622</td>
<td>388</td>
<td>425</td>
<td>97</td>
<td>235</td>
<td>731</td>
</tr>
<tr>
<td>Floating-Point Math</td>
<td>762</td>
<td>786</td>
<td>741</td>
<td>1085</td>
<td>2127</td>
<td>1416</td>
<td>629</td>
<td>187</td>
<td>5470</td>
<td>1654</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>2550</td>
<td>2762</td>
<td>1384</td>
<td>5283</td>
<td>5880</td>
<td>10468</td>
<td>2214</td>
<td>839</td>
<td>2732</td>
<td>2396</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>6060</strong></td>
<td><strong>6600</strong></td>
<td><strong>5070</strong></td>
<td><strong>16917</strong></td>
<td><strong>18645</strong></td>
<td><strong>21832</strong></td>
<td><strong>6301</strong></td>
<td><strong>2256</strong></td>
<td><strong>12284</strong></td>
<td><strong>7711</strong></td>
</tr>
</tbody>
</table>
Table A-7 shows the code size in bytes and cycle count and for each of the microcontrollers for every math operation without optimization and with full optimization.

Table A-7. FIR, Dhrystone, and Whetstone Code Size and Cycle Counts

<table>
<thead>
<tr>
<th>MCU</th>
<th>FIR FILTER&lt;sup&gt;1&lt;/sup&gt;</th>
<th></th>
<th>DHRYSTONE</th>
<th></th>
<th></th>
<th>WHETSTONE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CODE SIZE</td>
<td>CYCLES</td>
<td>CODE SIZE</td>
<td>CYCLES</td>
<td>CODE SIZE</td>
<td>CYCLES</td>
<td>CODE SIZE</td>
</tr>
<tr>
<td></td>
<td>UNOPT</td>
<td>OPT</td>
<td>UNOPT</td>
<td>OPT</td>
<td>UNOPT</td>
<td>OPT</td>
<td>UNOPT</td>
</tr>
<tr>
<td>MSP430FG4619</td>
<td>1020</td>
<td>1008</td>
<td>113308</td>
<td>108527</td>
<td>1272</td>
<td>838</td>
<td>166457</td>
</tr>
<tr>
<td>MSP430F149</td>
<td>1026</td>
<td>1014</td>
<td>118170</td>
<td>113399</td>
<td>1310</td>
<td>864</td>
<td>190834</td>
</tr>
<tr>
<td>PIC24FJ128GA</td>
<td>1638</td>
<td>1530</td>
<td>127125</td>
<td>122692</td>
<td>1779</td>
<td>1266</td>
<td>93920</td>
</tr>
<tr>
<td>PIC18F242</td>
<td>2058</td>
<td>2006</td>
<td>245704</td>
<td>182210</td>
<td>***</td>
<td>***</td>
<td>***</td>
</tr>
<tr>
<td>8051</td>
<td>2116</td>
<td>2056</td>
<td>330640</td>
<td>321781</td>
<td>3075</td>
<td>1946</td>
<td>732532</td>
</tr>
<tr>
<td>H8/300H</td>
<td>1440</td>
<td>1392</td>
<td>285580</td>
<td>271964</td>
<td>2173</td>
<td>1607</td>
<td>454518</td>
</tr>
<tr>
<td>MaxQ20</td>
<td>1592</td>
<td>1478</td>
<td>176720</td>
<td>167583</td>
<td>2393</td>
<td>1661</td>
<td>207905</td>
</tr>
<tr>
<td>ARM7TDI</td>
<td>1548</td>
<td>1528</td>
<td>37827</td>
<td>33114</td>
<td>1616</td>
<td>1000</td>
<td>83798</td>
</tr>
<tr>
<td>HCS12</td>
<td>1945</td>
<td>1917</td>
<td>1045982</td>
<td>1035934</td>
<td>1244</td>
<td>900</td>
<td>208648</td>
</tr>
<tr>
<td>ATmega8</td>
<td>1356</td>
<td>1358</td>
<td>365837</td>
<td>352894</td>
<td>2210</td>
<td>1474</td>
<td>240320</td>
</tr>
</tbody>
</table>

<sup>1</sup> The FIR filter code has been modified for correct operation from the previous version which lacked the MAC operations.
Appendix B Benchmarking Applications

B.1 Benchmarking Applications

In order to benchmark various aspects of a microcontroller's performance, the following set of simple applications was executed in simulation mode for each microcontroller.

8-bit_math.c — source file containing three math functions. One function performs addition of two 8-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

16-bit_math.c — source file containing three math functions. One function performs addition of two 16-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

32-bit_math.c — source file containing three math functions. One function performs addition of two 32-bit numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

floating_point_math.c — source file containing three math functions. One function performs addition of two floating-point numbers, one performs multiplication, and one performs division. The "main()" function calls each of these functions.

8-bit_switch_case.c — source file with one function containing a switch statement having 16 cases. An 8-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

16-bit_switch_case.c — source file with one function containing a switch statement having 16 cases. A 16-bit value is used to select a particular case. The "main()" function calls the "switch" function with an input parameter selecting the last case.

8-bit_2-dim_matrix.c — source file containing 3 two-dimensional arrays containing 8-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

16-bit_2-dim_matrix.c — source file containing 3 two-dimensional arrays containing 16-bit values—one of which is initialized. The "main()" function copies values from array 1 to array 2, then from array 2 to array 3.

matrix_multiplication.c — source file containing code, which multiplies a 3 x 4 matrix by a 4 x 5 matrix.

dhry.c — source file containing code, which does the Dhrystone analysis.

whet.c — source file containing code, which does the Whetstone analysis.
B.2 Benchmarking Application Source Code

The following are the C source-code files for the benchmarking applications used in this document.

8-bit Math.c

/*******************************************************************************
* Name : 8-bit Math
* Purpose : Benchmark 8-bit math functions.
*******************************************************************************/

typedef unsigned char UInt8;

UInt8 add(UInt8 a, UInt8 b) {
    return (a + b);
}

UInt8 mul(UInt8 a, UInt8 b) {
    return (a * b);
}

UInt8 div(UInt8 a, UInt8 b) {
    return (a / b);
}

void main(void) {
    volatile UInt8 result[4];

    result[0] = 12;
    result[1] = 3;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}
Benchmarking Application Source Code

8-bit 2-dim Matrix.c

/***************************************************************
* Name : 8-bit 2-dim Matrix
* Purpose : Benchmark copying 8-bit values.
***************************************************************/

typedef unsigned char UInt8;

const UInt8 m1[16][4] = {
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12},
    {0x12, 0x56, 0x90, 0x34},
    {0x78, 0x12, 0x56, 0x90},
    {0x34, 0x78, 0x12, 0x56},
    {0x90, 0x34, 0x78, 0x12}
} ;

void main (void)
{
    int i, j;
    volatile UInt8 m2[16][4], m3[16][4];

    for(i = 0; i < 16; i++)
    {
        for(j=0; j < 4; j++)
            {
            m2[i][j] = m1[i][j];
            m3[i][j] = m2[i][j];
            }
    }
    return;
}
8-bit Switch Case.c

/******************************************************************************
*/
/*
*   Name    : 8-bit Switch Case
*   Purpose : Benchmark accessing switch statement using 8-bit value.
*   */
/******************************************************************************/

typedef unsigned char UInt8;

UInt8 switch_case(UInt8 a)
{
    UInt8 output;

    switch (a)
    {
    case 0x01:
        output = 0x01;
        break;
    case 0x02:
        output = 0x02;
        break;
    case 0x03:
        output = 0x03;
        break;
    case 0x04:
        output = 0x04;
        break;
    case 0x05:
        output = 0x05;
        break;
    case 0x06:
        output = 0x06;
        break;
    case 0x07:
        output = 0x07;
        break;
    case 0x08:
        output = 0x08;
        break;
    case 0x09:
        output = 0x09;
        break;
    case 0x0a:
        output = 0x0a;
        break;
    case 0x0b:
        output = 0x0b;
        break;
    case 0x0c:
        output = 0x0c;
        break;
    case 0x0d:
        output = 0x0d;
        break;
    case 0x0e:
        output = 0x0e;
        break;
    case 0x0f:
        output = 0x0f;
        break;
    case 0x10:
        output = 0x10;
        break;
    } /* end switch*/
}
Benchmarking Application Source Code

```c
return (output);
}

void main(void)
{
    volatile UInt8 result;
    result = switch_case(0x10);
    return;
}
```
16-bit Math.c

/*******************************************************************************
* Name : 16-bit Math
* Purpose : Benchmark 16-bit math functions.
*******************************************************************************/

typedef unsigned short UInt16;

UInt16 add(UInt16 a, UInt16 b)
{
    return (a + b);
}

UInt16 mul(UInt16 a, UInt16 b)
{
    return (a * b);
}

UInt16 div(UInt16 a, UInt16 b)
{
    return (a / b);
}

void main(void)
{
    volatile UInt16 result[4];

    result[0] = 231;
    result[1] = 12;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}
16-bit 2-dim Matrix.c

/*******************************************************************************
*                             Name : 16-bit 2-dim Matrix
*                             Purpose : Benchmark copying 16-bit values.
*                             ********************************************************************************
*/

typedef unsigned short UInt16;

const UInt16 m1[16][4] = {
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234},
    {0x1234, 0x5678, 0x9012, 0x3456},
    {0x7890, 0x1234, 0x5678, 0x9012},
    {0x3456, 0x7890, 0x1234, 0x5678},
    {0x9012, 0x3456, 0x7890, 0x1234}
};

void main(void)
{
    int i, j;
    volatile UInt16 m2[16][4], m3[16][4];

    for(i = 0; i < 16; i++)
    {
        for(j = 0; j < 4; j++)
        {
            m2[i][j] = m1[i][j];
            m3[i][j] = m2[i][j];
        }
    }
    return;
}
16-bit Switch Case.c

/*******************************************************************************
*                              Name  : 16-bit Switch Case
*                              Purpose : Benchmark accessing switch statement using 16-bit value.
*                              
*******************************************************************************/

typedef unsigned short UInt16;

 UInt16 switch_case(UInt16 a)
{
    UInt16 output;

    switch (a)
    {
        case 0x0001:
            output = 0x0001;
            break;
        case 0x0002:
            output = 0x0002;
            break;
        case 0x0003:
            output = 0x0003;
            break;
        case 0x0004:
            output = 0x0004;
            break;
        case 0x0005:
            output = 0x0005;
            break;
        case 0x0006:
            output = 0x0006;
            break;
        case 0x0007:
            output = 0x0007;
            break;
        case 0x0008:
            output = 0x0008;
            break;
        case 0x0009:
            output = 0x0009;
            break;
        case 0x000a:
            output = 0x000a;
            break;
        case 0x000b:
            output = 0x000b;
            break;
        case 0x000c:
            output = 0x000c;
            break;
        case 0x000d:
            output = 0x000d;
            break;
        case 0x000e:
            output = 0x000e;
            break;
        case 0x000f:
            output = 0x000f;
            break;
        case 0x0010:
            output = 0x0010;
            break;
    }
    return output;
}
Benchmarking Application Source Code

    break;
    } /* end switch*/
    return (output);

void main(void)
{
    volatile UInt16 result;

    result = switch_case(0x0010);
    return;
}
32-bit Math.c

/*******************************************************************************
*
*     Name    : 32-bit Math
*     Purpose : Benchmark 32-bit math functions.
*
*******************************************************************************/

#include <math.h>

typedef unsigned long UInt32;

UInt32 add(UInt32 a, UInt32 b)
{    return (a + b);
}

UInt32 mul(UInt32 a, UInt32 b)
{    return (a * b);
}

UInt32 div(UInt32 a, UInt32 b)
{    return (a / b);
}

void main(void)
{    volatile UInt32 result[4];

    result[0] = 43125;
    result[1] = 14567;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}
Benchmarking Application Source Code

Floating-point Math.c

/*******************************************************************************
 * Name : Floating-point Math
 * Purpose : Benchmark floating-point math functions.
 ********************************************************************************/

float add(float a, float b)
{
    return (a + b);
}

float mul(float a, float b)
{
    return (a * b);
}

float div(float a, float b)
{
    return (a / b);
}

void main(void)
{
    volatile float result[4];
    result[0] = 54.567;
    result[1] = 14346.67;
    result[2] = add(result[0], result[1]);
    result[1] = mul(result[0], result[2]);
    result[3] = div(result[1], result[2]);
    return;
}
Matrix Multiplication.c

/*******************************************************************************
* Name : Matrix Multiplication
* Purpose: Benchmark multiplying a 3x4 matrix by a 4x5 matrix.
*   Matrix contains 16-bit values.
*******************************************************************************/

typedef unsigned short UInt16;

const UInt16 m1[3][4] = {
    {0x01, 0x02, 0x03, 0x04},
    {0x05, 0x06, 0x07, 0x08},
    {0x09, 0x0A, 0x0B, 0x0C}
};

const UInt16 m2[4][5] = {
    {0x01, 0x02, 0x03, 0x04, 0x05},
    {0x06, 0x07, 0x08, 0x09, 0x0A},
    {0x0B, 0x0C, 0x0D, 0x0E, 0x0F},
    {0x10, 0x11, 0x12, 0x13, 0x14}
};

void main(void)
{
    int m, n, p;
    volatile UInt16 m3[3][5];

    for(m = 0; m < 3; m++)
    {
        for(p = 0; p < 5; p++)
        {
            m3[m][p] = 0;

            for(n = 0; n < 4; n++)
            {
                m3[m][p] += m1[m][n] * m2[n][p];
            }
        }
    }
    return;
}
FIR Filter.c

/*******************************************************************************
* Name        : FIR Filter
* Purpose     : Benchmark an FIR filter. The input values for the filter
* is an array of 51 16-bit values. The order of the filter
* 17.
*******************************************************************************/

#ifndef MSP430
#include "msp430x14x.h"
#endif
#include <math.h>

#define FIR_LENGTH 17

const float COEFF[FIR_LENGTH] =
{ -0.000091552734, 0.000305175781, 0.004608154297, 0.003356933594,
  -0.025939941406, -0.044006347656, 0.063079833984,
  0.290313720703, 0.416748046875, 0.290313720703,
  0.063079833984, -0.044006347656, -0.025939941406, 0.003356933594,
  0.004608154297, 0.000305175781, -0.000091552734};

/* The following array simulates input A/D converted values */

const unsigned int INPUT[] =
{ 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
  0x0200, 0x0200, 0x01C0, 0x0180, 0x0140, 0x0100, 0x00C0, 0x0080,
  0x0040, 0x00C0, 0x0080, 0x00C0, 0x0100, 0x0140, 0x0180, 0x01C0,
  0x2000, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000, 0x0800,
  0x0080, 0x0040, 0x00C0, 0x0080, 0x00C0, 0x0100, 0x0140, 0x0180,
  0x01C0, 0x0200, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400, 0x1000,
  0x0800, 0x0080, 0x0040, 0x00C0, 0x0080, 0x00C0, 0x0100, 0x0140,
  0x0180, 0x01C0, 0x0200, 0x2400, 0x2000, 0x1C00, 0x1800, 0x1400,
  0x1000, 0x0800, 0x0080, 0x0040};

void main(void)
{
  int i, y; /* Loop counters */
  volatile float OUTPUT[36], sum;

  #ifdef MSP430
  WDTCTL = WDTPW + WDTHOLD; /* Stop watchdog timer */
  #endif

  for(y = 0; y < 36; y++)
  {
    sum=0;
    for(i = 0; i < FIR_LENGTH/2; i++)
    {
      sum = sum+COEFF[i] * ( INPUT[y + 16 - i] + INPUT[y + i] );
    }
    OUTPUT[y] = sum + (INPUT[y + FIR_LENGTH/2] * COEFF[FIR_LENGTH/2] );
  }
  return;
}
Dhry.c

/*******************************************************************************
* Name: Dhrystone
* Purpose: Benchmark the Dhrystone code. This benchmark is used to gauge
* the performance of the microcontroller in handling pointers,
* structures and strings.
*******************************************************************************/

#include <stdio.h>
#include <string.h>
define LOOPS 100
#define structassign(d, s) {d = s}

typedef enum (Ident1, Ident2, Ident3, Ident4, Ident5) Enumeration;
typedef int OneToThirty;
typedef int OneToFifty;
typedef unsigned char CapitalLetter;
typedef unsigned char String30[31];
typedef int Array1Dim[51];
typedef int Array2Dim[51][51];

typedef struct Record
{
    struct Record *PtrComp;
    Enumeration Discr;
    Enumeration EnumComp;
    OneToFifty IntComp;
    String30 StringComp;
};

typedef struct Record RecordType;
typedef RecordType * RecordPtr;
typedef int boolean;

#define NULL 0
#define TRUE 1
#define FALSE 0

# define REG register

int IntGlob;
boolean BoolGlob;
unsigned char Char1Glob;
unsigned char Char2Glob;
Array1Dim Array1Glob;
Array2Dim Array2Glob;
RecordPtr PtrGlb;
RecordPtr PtrGlbNext;
RecordType rec1, rec2;

Enumeration Func1(CapitalLetter CharPar1, CapitalLetter CharPar2)
{
    REG CapitalLetter CharLoc1;
    REG CapitalLetter CharLoc2;
    CharLoc1 = CharPar1;
    CharLoc2 = CharLoc1;
    if (CharLoc2 != CharPar2)
        return (Ident1);
    else
        return (Ident2);
}

boolean Func2(String30 StrParI1, String30 StrParI2)
{
    REG OneToThirty IntLoc;
    REG CapitalLetter CharLoc;
Benchmarking Application Source Code

```c
IntLoc = 1;
while (IntLoc <= 1)
    if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
        CharLoc = 'A';
        ++IntLoc;
    if (CharLoc == 'W' && CharLoc <= 'Z')
        IntLoc = 7;
    if (CharLoc == 'X')
        return(TRUE);
else
    if (strcmp(StrParI1, StrParI2) > 0)
        IntLoc += 7;
        return (TRUE);
    else
        return (FALSE);
}

boolean Func3(Enumeration EnumParIn)
{
    REG Enumeration EnumLoc;
    EnumLoc = EnumParIn;
    if (EnumLoc == Ident3) return (TRUE);
    return (FALSE);
}

void Proc7(OneToFifty IntParI1, OneToFifty IntParI2, OneToFifty *IntParOut)
{
    REG OneToFifty IntLoc;
    IntLoc = IntParI1 + 2;
    *IntParOut = IntParI2 + IntLoc;
}

void Proc4(void)
{
    REG boolean BoolLoc;
    BoolLoc = Char1Glob == 'A';
    BoolLoc |= BoolGlob;
    Char2Glob = 'B';
}

void Proc5(void)
{
    Char1Glob = 'A';
    BoolGlob = FALSE;
}

void Proc6(Enumeration EnumParIn, Enumeration *EnumParOut)
{
    *EnumParOut = EnumParIn;
    if (! Func3(EnumParIn) )
    *EnumParOut = Ident4;
    switch (EnumParIn)
    {
    case Ident1: *EnumParOut = Ident1; break;
    case Ident2: if (IntGlob > 100) *EnumParOut = Ident1;
    else *EnumParOut = Ident4;
    break;
    case Ident3: *EnumParOut = Ident2; break;
    case Ident4: break;
    ```
case Ident5:  *EnumParOut = Ident3;
}

void Proc3(RecordPtr *PtrParOut)
{
  if (PtrGlb != NULL)
    *PtrParOut = PtrGlb->PtrComp;
  else
    IntGlob = 100;
  Proc7(10, IntGlob, &PtrGlb->IntComp);
}

void Proc1(RecordPtr PtrParIn)
{
  #define NextRecord (*(PtrParIn->PtrComp))
  structassign(NextRecord, *PtrGlb);
  PtrParIn->IntComp = 5;
  NextRecord.IntComp = PtrParIn->IntComp;
  NextRecord.PtrComp = PtrParIn->PtrComp;
  Proc3(&NextRecord.PtrComp);
  if (NextRecord.Discr == Ident1)
  {
    NextRecord.IntComp = 6;
    Proc6(PtrParIn->EnumComp, &NextRecord.EnumComp);
    NextRecord.PtrComp = PtrGlb->PtrComp;
    Proc7(NextRecord.IntComp, 10, &NextRecord.IntComp);
  }
  else
  structassign(*PtrParIn, NextRecord);
  #undef NextRecord
}

void Proc2(OneToFifty *IntParIO)
{
  REG OneToFifty IntLoc;
  REG Enumeration EnumLoc;
  IntLoc = *IntParIO + 10;
  for(;;)
  {
    if (Char1Glob == 'A')
    {
      --IntLoc;
      *IntParIO = IntLoc - IntGlob;
      EnumLoc = Ident1;
    }
    if (EnumLoc == Ident1)
      break;
  }
}

void Proc8 (Array1Dim Array1Par, Array2Dim Array2Par, OneToFifty IntParI1, OneToFifty IntParI2)
{
  REG OneToFifty IntLoc;
  REG OneToFifty IntIndex;
  IntLoc = IntParI1 + 5;
  Array1Par[IntLoc] = IntParI2;
  Array1Par[IntLoc+1] = Array1Par[IntLoc];
  Array1Par[IntLoc+30] = IntLoc;
  for (IntIndex = IntLoc; IntIndex <= (IntLoc+1); ++IntIndex)
    Array2Par[IntLoc][IntIndex] = IntLoc;
  ++Array2Par[IntLoc][IntLoc-1];
  Array2Par[IntLoc+20][IntLoc] = Array1Par[IntLoc];
  IntGlob = 5;
void Proc0 (void)
{
    OneToFifty IntLoc1;
    REG OneToFifty IntLoc2;
    OneToFifty IntLoc3;
    REG unsigned char CharLoc;
    REG unsigned char CharIndex;
    Enumeration EnumLoc;
    String30 String1Loc;
    String30 String2Loc;
    extern unsigned char *malloc();
    long time(long *);
    long starttime;
    long benchtime;
    long nulltime;
    register unsigned int i;

    for (i = 0; i < LOOPS; ++i);
    PtrGlbNext = &rec1; /* (RecordPtr) malloc(sizeof(RecordType)); */
    PtrGlb = &rec2; /* (RecordPtr) malloc(sizeof(RecordType)); */
    PtrGlb->PtrComp = PtrGlbNext;
    PtrGlb->Discr = Ident1;
    PtrGlb->EnumComp = Ident3;
    PtrGlb->IntComp = 40;
    strcpy(PtrGlb->StringComp, "DHRYSTONE PROGRAM, SOME STRING");
    strcpy(String1Loc, "DHRYSTONE PROGRAM, 1'ND STRING"); /*GOOF*/
    for (i = 0; i < LOOPS; ++i)
    {
        Proc5();
        Proc4();
        IntLoc1 = 2;
        IntLoc2 = 3;
        strcpy(String2Loc, "DHRYSTONE PROGRAM, 2'ND STRING");
        EnumLoc = Ident2;
        BoolGlob = ! Func2(String1Loc, String2Loc);
        while (IntLoc1 < IntLoc2)
        {
            IntLoc3 = 5 * IntLoc1 - IntLoc2;
            Proc7(IntLoc1, IntLoc2, &IntLoc3);
            ++IntLoc1;
        }
        Proc8(Array1Glob, Array2Glob, IntLoc1, IntLoc3);
        Proc1(PtrGlb);
        for (CharIndex = 'A'; CharIndex <= Char2Glob; ++CharIndex)
            if (EnumLoc == Func1(CharIndex, 'C'))
                Proc6(Ident1, &EnumLoc);
        IntLoc3 = IntLoc2 * IntLoc1;
        IntLoc2 = IntLoc3 / IntLoc1;
        IntLoc2 = 7 * (IntLoc3 - IntLoc2) - IntLoc1;
        Proc2(&IntLoc1);
    }
}

void main(void)
{
    Proc0();
}
Whet.c

>Description

Whetstone

Purpose
Benchmark the Whetstone code. The code focuses on scientific functions such as sine, cosine, exponents and logarithm on fixed and floating point numbers.

*****************************************************************************/
#include <math.h>
#include <stdio.h>

PA(float E[5]);
P0(void);
P3(float *X, float *Y, float *Z);

float T,T1,T2,E1[5];
int J,K,L;
float X1,X2,X3,X4;
long ptime,time0;

main ()
{
    int LOOP,I,II,JJ,N1,N2,N3,N4,N5,N6,N7,N8,N9,N10,N11;
    float X,Y,Z;
    T = .499975;
    T1 = 0.50025;
    T2 = 2.0;
    LOOP = 1;
    II = 1;
    for (JJ=1;JJ<=II;JJ++)
    {
        N1 = 0;
        N2 = 2 * LOOP;
        N3 = 2 * LOOP;
        N4 = 2 * LOOP;
        N5 = 0;
        N6 = 2 * LOOP;
        N7 = 2 * LOOP;
        N8 = 2 * LOOP;
        N9 = 2 * LOOP;
        N10 = 0;
        N11 = 2 * LOOP;

        /* Module 1: Simple identifiers */
        X1 = 1.0;
        X2 = -1.0;
        X3 = -1.0;
        X4 = -1.0;
        if (N1!=0)
        {
            for(I=1;I<=N1;I++)
            {
                X1 = (X1 + X2 + X3 - X4)*T;  
                X2 = (X1 + X2 - X3 + X4)*T;  
                X3 = (X1 - X2 + X3 + X4)*T;  
                X4 = (-X1 + X2 + X3 + X4)*T; 
            }
        }

        /* Module 2: Array elements */
        E1[1] = 1.0;
        E1[2] = -1.0;
        E1[3] = -1.0;
        E1[4] = -1.0;
        if (N2!=0)
Benchmarking Application Source Code

```
{ 
  for (I=1;I<=N2;I++)
  {
  }
}

/* Module 3: Array as parameter */
if (N3!=0)
{
  for (I=1;I<=N3;I++)
  {
    PA(E1);
  }
}

/* Module 4: Conditional jumps */
J = 1;
if (N4!=0)
{
  for (I=1;I<=N4;I++)
  {
    if (J==1) goto L51;
    J = 3;
    goto L52;
  }
  L51:
  J = 2;
  L52:
  if (J > 2) goto L53;
  J = 1;
  goto L54;
  L53:
  J = 0;
  L54:
  if (J < 1) goto L55;
  J = 0;
  goto L60;
  L55:
  J = 1;
  L60:
}

/* Module 5: Integer arithmetic */
J = 1;
K = 2;
L = 3;
if (N6!=0)
{
  for (I=1;I<=N6;I++)
  {
    J = J * (K-J) * (L-K);
    K = L * K - (L-J) * K;
    L = (L - K) * (K + J);
    E1[I-1] = J + K + L;
    E1[K-1] = J * K * L;
  }
}

/* Module 6: Trigonometric functions */
X = 0.5;
Y = 0.5;
if (N7!=0)
{
  for (I=1;I<=N7;I++)
  {
    X = T*atan(T2*sin(X)*cos(X)/(cos(X+Y)+cos(X-Y)-1.0));
    Y = T*atan(T2*sin(Y)*cos(Y)/(cos(X+Y)+cos(X-Y)-1.0));
  }
}
```
Benchmarking Application Source Code

} }

/* Module 7: Procedure calls */
X = 1.0;
Y = 1.0;
Z = 1.0;
if (N8!=0)
{
    for (I=1;I<=N8;I++)
    {
        P3(&X,&Y,&Z);
    }
}

/* Module 8: Array references */
J = 1;
K = 2;
L = 3;
E1[1] = 1.0;
E1[2] = 2.0;
E1[3] = 3.0;
if (N9!=0)
{
    for (I=1;I<=N9;I++)
    {
        P0();
    }
}

/* Module 9: Integer arithmetic */
J = 2;
K = 3;
if (N10!=0)
{
    for (I=1;I<=N10;I++)
    {
        J = J + K;
        K = J + K;
        J = K - J;
        K = K - J - J;
    }
}

/* Module 10: Standard functions */
X = 0.75;
if (N11!=0)
{
    for (I=1;I<=N11;I++)
    {
        X = sqrt(exp(log(X)/T1));
    }
}

PA(E) float E[5];
{
    int J1;
    J1 = 0;
          J1 = J1 + 1;
          if ((J1 - 6) < 0) goto L10;
Benchmarking Application Source Code

return;

P0()
{
    E1[J] = E1[K];
    E1[K] = E1[L];
    E1[L] = E1[J];
    return;
}

P3(X,Y,Z) float *X,*Y,*Z;
{
    float Y1;
    X1 = *X;
    Y1 = *Y;
    X1 = T * (X1 + Y1);
    Y1 = T * (X1 + Y1);
    *Z = (X1 + Y1) / T2;
    return;
}
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DSP</td>
<td>Broadband</td>
</tr>
<tr>
<td>Interface</td>
<td>Digital Control</td>
</tr>
<tr>
<td>Logic</td>
<td>Military</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Optical Networking</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Security</td>
</tr>
<tr>
<td>Low Power Wireless</td>
<td>Telephony</td>
</tr>
<tr>
<td></td>
<td>Video &amp; Imaging</td>
</tr>
<tr>
<td></td>
<td>Wireless</td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2006, Texas Instruments Incorporated