Welcome
to the family...

Technical Introduction

C6808 Code Development System
for Freescale™ RS08

Byte Craft Limited is please to offer the only C compiler for Freescale’s RS08 architecture. With C6808, you can program RS08 completely in C. You can also migrate existing HC08 programs and other C software to RS08 with ease.

C6808 with RS08 support is shipping now.

Read on for details about RS08 support, reference designs, and more information about Byte Craft Limited...
Byte Craft Limited’s C6808 Code Development System includes an optimizing C compiler, optimizing linker, BCLIDE integrated development environment, full documentation, and one year’s full technical support.

C6808 draws on Byte Craft Limited’s experience and expertise in a wide variety of embedded architectures. Translating your programs to different instruction sets is our main business, and we have in-depth understanding of the strengths of modern processors.

Supports all HC08, HCS08, and RS08 devices

Select or change the target device simply by including a header file; code generation changes automatically. All library code is shipped with C sources, making it portable across different devices.

Memory Management

C6808 includes enhanced memory management such as named memory directives. Local address space directives allows the user to maximize the use of RAM, direct the placement of local variables, re-use RAM locations and pass multiple arguments to functions. Other named memory directives add support for variables, allocated externally or internally, that are managed through software drivers.

Features in Detail

- Highly-optimized generated code. Full versions generate ROMable code; demonstration versions generate listing files with assembly.
- #pragma directives. Ports and other resources are declared and protected against improper access.
- Named address spaces, supporting the grouping of variables at specific memory locations.
- Register-oriented data types, for direct access to processor registers when necessary.
- Data structures of arbitrary complexity, including packed bit fields in structures.
Problem Space, Solution Space

Programming in C is working in the solution space, not the problem space. You can implement an algorithm in C and pay attention to the correctness of your calculations, free from preoccupations about registers, pages, and the subtle differences between addressing modes. The only limit on complexity is the available resources of the processor.

The greater breadth of expression available in C permits you to design programs that are meaningful to you, but still compile to valid, optimized executables.

Data Types

Our research has told us that choosing the right data types for the application is key to a successful embedded project. C offers numerous specialized data types, and the coding environment in which to use them. The only limit is the hardware resources of your chosen part.

Type Sizes: C6808 offers signed and unsigned integer types in 8, 16, 24, and 32 bits. Choose data types to suit the solution of your problem, and let the compiler worry about the carry bit.

Fixed point: C6808 supports TR 18037 fixed point and accumulator types. _Fract types represent fractional values between 0 and 1, and _Accum types have small integer components.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short _Fract</td>
<td>8</td>
</tr>
<tr>
<td>unsigned _Fract</td>
<td>16</td>
</tr>
<tr>
<td>unsigned long _Fract</td>
<td>24</td>
</tr>
<tr>
<td>signed short _Fract</td>
<td>s.7</td>
</tr>
<tr>
<td>signed _Fract</td>
<td>s.15</td>
</tr>
<tr>
<td>signed long _Fract</td>
<td>s.23</td>
</tr>
<tr>
<td>unsigned short _Accum</td>
<td>8.8</td>
</tr>
<tr>
<td>unsigned _Accum</td>
<td>8.16</td>
</tr>
<tr>
<td>unsigned long _Accum</td>
<td>8.24</td>
</tr>
<tr>
<td>signed short _Accum</td>
<td>s8.7</td>
</tr>
<tr>
<td>signed _Accum</td>
<td>s8.15</td>
</tr>
<tr>
<td>signed long _Accum</td>
<td>s8.23</td>
</tr>
</tbody>
</table>

Fixed point math is implemented within the compiler itself; no libraries are needed. The listing file allows full inspection of generated code.
**Unambiguous sizes:** C6808 offers ISO types that make it clear what values can be expected.

<table>
<thead>
<tr>
<th>uint8_t</th>
<th>uint16_t</th>
<th>uint24_t</th>
<th>uint32_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>int16_t</td>
<td>int24_t</td>
<td>int32_t</td>
</tr>
</tbody>
</table>

_Bool: C6808 supports _Bool, using bit test and set instructions wherever possible. The deprecated Byte Craft Limited bit type is still supported.

**bits:** Code Development Systems have always offered the bits type, a structure of eight bits that are individually addressable (with constant-specified members).

**Derived Types and Complexity**

C6808 supports single-dimension arrays, structures, unions, and any combination of them. This permits data to reflect the solution, not the hardware. The only limit placed on this complexity is the available resources on the target part.

C6808 also supports bit fields within structures. This particularly complex code generation and optimization problem can help conserve resources in certain circumstances.

**Named Memory**

Where the problem space forces you to dwell on addressing modes, the solution space offers named memory spaces to organize your variable allocations. This separates low-level details about variable access from the high-level design of the program.

Header file declarations organize memory into named sections (see the supplied headers for examples). They can represent scratchpad RAM versus persistent data, or read-only resources separate from the program and its constant objects. They can even be used to issue warnings when one class of objects exceeds memory budgets, though others do not.

Named memory can be more than just storage. C6808 offers user-implemented memory declarations: accesses to variables declared in these areas are implemented using device driver routines. In the past, this type of memory has been used for serial bus communications, Flash programming, and other driver-oriented applications.

Here’s an example of accessing variables declared on an external SPI EEPROM device:
Combining the user’s logical divisions of memory with the different types of physical memory is exactly the type of work most successfully accomplished by an optimizing compiler.

**Hardware Access**

For those times when hardware access is unavoidable, C6808 brings these details into the C language and into the solution space. *There is nothing you can do in assembly that can’t be done in C.*
Register access: Following on from named memory, C6808 offers direct access to processor registers for specialized purposes. Reading and writing variables of (storage class) type `register _A` and `register _CCR` generates stores and loads as appropriate. This applies even to registers that can’t be directly accessed through the instruction set, another benefit of C.

Device-specific instructions: These are brought out as C intrinsics:

<table>
<thead>
<tr>
<th>WAIT();</th>
<th>NOP();</th>
<th>SEC();</th>
<th>BKGD();</th>
<th>SLA();</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOP();</td>
<td></td>
<td>CLC();</td>
<td></td>
<td>SHA();</td>
</tr>
</tbody>
</table>

Reset code: The state of the processor is generated automatically. Variables are initialized, and a user-defined `__STARTUP()` routine is executed if defined. `__STARTUP()` can serve as a debugging-specific shim, compiled in for testing and then simply left out for production.
Why C?

Programming a desktop computer and programming a miniscule embedded processor have a lot in common. Though the resources differ, the underlying model of computing is the same. It makes perfect sense to develop for two computers in the same language, even if the applications are completely different:

- Computers are very good at accounting. A compiler can track actual demand on resources far more closely than a programmer doing so manually. The accounting is redone each time a program is compiled, and all code creation decisions, based on the metrics obtained, are revisited at each compile.

- Compilers use the entire instruction set. The complete list of capabilities and constraints of the instruction set and architecture are embedded in the compiler. As mentioned before, developers are left free to think in the solution space.

- “Programmers’ tricks”, once added to the knowledge base in the compiler, remain available for new applications forever. Furthermore, the compiler is capable of recombining these techniques into expressions not previously anticipated.

    Optimization strategies that involve data combinations, instruction sequences, or placement-specific code, are extremely hard for assembly language programmers to implement. These optimizations are easy for compilers to identify and integrate into the program.

It's not just memory use that's at stake.

Optimization reduces one or more of: ROM usage, RAM usage, or application execution time.

- Compression trades off increases and reductions in ROM, RAM and execution time.

- Optimization allows more application code to be run in the same-sized target processor. Optimization reduces execution time for an application, or allows more code to be executed in the same time. Finally, optimization allows clock speeds to be reduced, in turn reducing EMI and power consumption.

Quick: what’s one thing that RS08 does, that HC08 and HCS08 don't do?

Processor architecture is our specialty. We ask questions like this to investigate the best ways of optimizing code.

The answer?

RS08 can perform single-bit set and clear anywhere in memory—by virtue of its paging mechanism. This feature can make it easier to create and manage Boolean values and structure bit fields, especially when combined with page selection optimization.
Why C for RS08? Here are two examples of the effectiveness of C in the RS08:

- Multiple memory spaces. Moving a variable to another data space changes the type of access and instructions available to reference the data. An optimizing compiler can manage the consequences throughout the program of moving a variable.

- RS08 literal data access, using physical address space, requires 16 to 23 bytes to access a random memory location, owing to the use of the PAGE, X and D[X] registers. High-level language tools can create and maintain a virtual address space on the RS08 that will reduce the access to either 7 or 10 bytes. This reduction is transparent to the programmer.

Compiler technology is especially effective for processors with instruction sets optimized for generated code and processors with pared-down resources. The RS08 core employs both these strategies.

Management of rare resources on the RS08 includes:

- Managing multiple address spaces and address-specific instructions.
- Reusing RAM as much as possible.
- Eliminating unnecessary PAGE, X, and accumulator operations.
- Compiling or eliminating program stack accesses. There are numerous optimizations that involve the Shadow Program Counter in flow control, not all of which are obvious or mutually-compatible.
- Tracking condition code register bits. Testing and setting CC bits requires indirect operations; if the compiler can eliminate this at compile time, so much the better.
- Tracking and overlaying TINY and SHORT address space contents, and reallocating variables between the two spaces depending on operations performed on them.

Insider Knowledge

Our experience tells us that hand-written assembly can reach a 16-to-1 ratio of program code to RAM. For every new RAM variable, typically 16 bytes of program store are needed. Compilers’ superior memory accounting can generally equal or beat the best handwritten assembly ROM to RAM ratios.

The MC9RS08KA2 has a ROM to RAM ratio of 32. With 32 bytes ROM for every byte of RAM, applications on this processor will benefit from every optimization possible to fit their data into the RAM available.
RS08 has the potential to change interrupt programming significantly. Its architecture encourages **event-driven programming** and **run-to-completion routines**.

**From Interrupts to Threads**

Even though there are no traditional dispatched interrupts, we’ve found that RS08’s architecture saves on program complexity. The time and memory saved through eliminating context switching more than compensates for the occasional delay in servicing an interrupt.

We’ve designed two different ways of responding to asynchronous events in C in RS08. Consider them both for your project: the first is simple and traditional, but the second is far more powerful.

**Event-driven programming**

Traditional interrupts divert the processor’s attention to an asynchronous task, and restore it after the interrupt service is complete. HC08 and HCS08 followed on in the HC05 practice of pushing (almost) the entire processor state on the stack during interrupt servicing. The biggest cost of this is the amount of time and resources needed to support this operation. But is it really necessary?

We’ve dealt with a variety of architectures, some of which are run-to-completion—no asynchronous events disturb the processor between logical tasks. Experience has shown us that run-to-completion, while risking delays in response, helps eliminate:

- Expensive preservation of processor state during context switches.
- Complexity in managing local variable values.
- Complexity in interrupting and restarting main line software.

Event-driven programming, where execution takes place only in response to external stimuli and is usually run-to-completion, is a viable alternative.

**RS08-specific event-driven programming**

This method is RS08-specific. It is also the simplest, but not necessarily the most elegant. It simply performs a `WAIT()` and then tests possible sources of interrupt. Interrupt priority is determined by control flow.

Within the main control loop, software performs a `WAIT()` or `STOP()` instruction. When the processor wakes due to a hardware interrupt, code must test all relevant interrupt flags to determine what caused the interrupt, take appropriate action, and loop around to the initial `WAIT()`.

```c
void main(void)
{
    while(1)
    {
        WAIT();
```
if (KBISC.KBF)
{
    // service peripheral’s request
    // and acknowledge
    KBISC.KBACK = 1;
}
else
if (MTIMSC.TOF)
{
    //...and so on.
    MTIMSC.TRST = 1;
}

Existing code would require significant rewriting to make use of this method. This is suggested for new, RS08-specific applications only.

**Threads**

Interrupts on RS08 do not actually interrupt the flow of execution. They may wake the processor from a `WAIT()` or `STOP()` mode, but they can only be handled when software invokes a specific thread of execution.

Byte Craft Limited has designed a threaded programming model that makes event-driven programming in C easy and intuitive. We use the compiler to organize sets of event and state tests in one coherent architecture.

To get the definitions right: a **thread** is a C function with the added semantics of centralized conditional dispatch. The elements of C6808 thread programming are as follows:

- Thread declarations appear as `#pragma` directives. Each thread has its own gating expression that evaluates to a Boolean value. The expression can test interrupt source flags, global variables, or I/O ports.

  ```
  #pragma thread identifier keyword (expression);
  ```

  The keyword is one of `SOFTWARE` or `INTERRUPT`, indicating whether the dispatch should take place at will or after a `WAIT()`.

- C6808 generates a function `__DISPATCH()` that implements the thread dispatch code; the developer calls it at an appropriate point in their kernel.

  In `__DISPATCH()`, when the thread conditions are satisfied, control passes to the matching function. If no threads match, control falls through to the main line code.

Threads grow naturally from the practice of declaring interrupt service routines using `#pragma vector` declarations. Interrupt vectors are selected by the underlying hardware, where threads are selected by compiler-supplied program code. Ultimately, in RS08 all “asynchronous” code must still be dispatched by program flow. Threads can give the appearance of being selected by hardware, but can in fact be selected by dispatch code that references both hardware and software.
A collection of threads can implement a complex peripheral function in an intuitive and straightforward fashion. See the quadrature decoder described later.

**Evaluation**

All thread expressions are evaluated within `__DISPATCH()` in declaration order. When any expression is completely evaluated, and evaluates to `true`, that thread is run immediately. If no expressions are true, execution falls through to the next main line program instruction after the call to `__DISPATCH()`.

**Examples**

**Real Time Interrupt**

Implementing a real time interrupt for a real-time kernel is a matter of choosing the right expressions for `#pragma thread` directives:

```c
unsigned int count = 0;

#define SW_PRESCALE a_value

#pragma thread __TIMRESET() INTERRUPT
 (SRTISC.RTIF == 1 && count == SW_PRESCALE);

#pragma thread __TIM() INTERRUPT
 (SRTISC.RTIF == 1);

void __TIM(void)
{
    count++;  
}

void __TIMRESET(void)
{
    count = 0;
    SRTISC.RTIACK = 1;
}

void main(void)
{
    while(1)
        __DISPATCH();
}
```

**Two sources**

This example is comparable to the RS08-specific design described at the beginning of the article.

```c
#pragma thread KBI() wait (KBISC.KBF == 1);
#pragma thread MTIM() wait (MTIMSC.TOF == 1);
/* interrupt functions */
```
void KBI(void)  
{  
    // actions  
    KBISC.KBACK = 1;  
}  

void MTIM(void)  
{  
    // actions  
    MTIMSC.TRST = 1;  
}  

void main(void)  
{  
    while(1) __DISPATCH();  
}

In this example, only the KBI and MTIM routines will be run, and only after a \texttt{WAIT()}.

\textbf{Future Directions}

\textbf{Threads and Libraries}

One novel use of threads is to distribute interrupt service across several libraries. Each library that requires event servicing can declare its own \texttt{\#pragma thread condition}, complete with macros as expression terms. The compiler will integrate the expressions and thread calls during final compilation.

\textbf{Threads and Vectored Interrupts}

Thread-based design can be integrated with vectored interrupts: a vectored interrupt can perform base-level tasks such as setting one or more of the flags or variables to be tested during thread dispatch.

The benefit: interrupt servicing on different platforms takes a varying amount of time to preserve and restore main-line code state. Using the simplest possible vectored interrupt to set the environment for threads can help minimize the overhead required.
Quadrature Decoder

This example demonstrates a quadrature decoder.

The RS08's keyboard interrupt has options to raise an interrupt on low-to-high or high-to-low transitions. Therefore, we can use the RS08 WAIT() to suspend the processor until a transition appears on one of the input sensors. This design is ideal because it integrates with other interrupt tasks, and can ultimately lower the power consumption of the processor during decoding.

The quadrature decoder routine checks the state of the input pins, and inverts them to determine the next edges to look for. It implements a state table to decode whether the input quadrature signal is moving forward or backward.

```assembly
void QDecode (void)
{
    char save_port, state;
    state = ((KBIES << 2) & 0x0c);
    state = state + (save_port & 0x03); // state Old + new value complemented

    if ((state == 0b1101) ||
        (state == 0b1011) ||
        (state == 0b0010) ||
        (state == 0b0100)) tally++;

    if ((state == 0b1110) ||
        (state == 0b1000) ||
        (state == 0b0001) ||
        (state == 0b0111)) tally--;
}
```
This example also demonstrates a threaded software architecture.

*Look at the complete source and listing file in the accompanying BCLIDE project.*
Intelligent A/D with RS08 and C6808

Introduction

This application note describes an interesting implementation of A/D conversion using the RS08 microcontroller from Freescale and the C6808 Code Development System from Byte Craft Limited.

The RS08 part does not include an A/D converter, but with a little bit of external hardware and programming, an RS08 can perform A/D conversion in engineering units. Using the RS08’s comparator and modulo timer, the A/D conversion can be performed as an interrupt task, allowing the part to perform other software tasks at the same time.

Theory

This method of A/D conversion is ratiometric. Software keeps two counts: the number of positive comparator readings and the total number of readings taken. The ratio between the two indicates the input voltage. The faster the A/D task can take place, the more accurate the readings.

The circuit strives to keep the input voltage close to the switching voltage of the comparator. The comparator is a flexible device that can operate independently of the RS08. Though it can interrupt the RS08 on a change in its output, this design simply polls it. It functions as an op-amp that the processor can watch.

The comparator’s positive input is set to the bandgap reference voltage source (1.218V). Its negative input is tied to the input voltage through a current source resistor. A small capacitor integrates the error current.

The digital output of the comparator is fed back into the input through another current source. Because the voltage input is fed in through the negating input, the output of the comparator is opposite to the difference between the bandgap input and the voltage under test. This will cause the comparator output to oscillate. In a software-only implementation, the same effect is created by constantly inverting an output pin with respect to the input.

The modulo timer task can take place along with other frequent modulo timer tasks. So long as the comparator is sampled at regular intervals, this conversion is not time-constrained.

Design

This is a schematic for the design using the 6-pin RS08KA2 part:
The values of the resistors are selected to determine the range of input values to be measured. The capacitor is non-critical; use a 0.01 to 0.1 µF capacitor.

With a sense voltage on input of 1.218 volts, Vh of 3.0V and Vl of 0V, the operating range is determined by these equations:

\[
V_{\text{min}} = V_s - ((V_h - V_s) \times (R_i / R_f))
\]
\[
V_{\text{max}} = V_s - ((V_l - V_s) \times (R_i / R_f))
\]

In software, the comparator is set to run freely (no interrupts on output changes), and an interrupt loop counts the number of iterations and number of positive readings on the comparator. The measured voltage is calculated as follows:

\[
V_{\text{in}} = \left(\frac{\text{onecount}}{\text{totalcount}}\right) \times (V_{\text{max}} - V_{\text{min}})
\]

but calibrating the device allows you to avoid the division.

**Software**

The required software appears at the end of the document. Some excerpts follow.

Using the RS08’s modulo timer, the conversion can take place as an interrupt task, allowing the processor to perform other (interrupt-driven) tasks at the same time.

**Configuration**

The comparator is configured to run freely, and the modulo timer is set to generate an interrupt.
Event loop

This excerpt shows the while(1) statement that bounds the operating loop and the WAIT() that heads it. We used WAIT() to allow the modulo timer to continue running. Whenever the modulo clock rolls over, the program increments the total and (if needed) positive counts.

while(1)
{

WAIT(); //wait mode allows modulo clock to continue.

/* awaken from interrupt and check sources */
if(timer_overflown())
{
    totalcount++;

    if(comparator_hi())
    {
        positivecount++;
    }
}

At the end of the timer interrupt, the interrupt flag is reset and the loop is continued.

Calibration

The circuit can give an output in engineering units once calibrated. The sample software incorporates calibration, but here are the steps:

1. Apply the maximum known reference voltage to the input.
2. Signal the software to count samples until the count of positive samples is an engineering unit (i.e., 100, 1000, etc.)
3. Software records the count of total samples needed to reach the proper maximum positive count, and returns to normal operation.
4. Apply a voltage to measure to the input.
5. Software performs a count of total samples matching the recorded value.
6. Use the resulting count of positive samples as an engineering value between zero and the calibration maximum count of positive samples.

If you’re tempted to use a low-cost RS08 part in your design, but need full A/D conversion, designing in a software-assisted A/D converter may serve your purposes.

*Look at the complete source and listing file in the accompanying BCLIDE project*
Every Code Development System ships with BCLIDE, a Windows development environment that helps you manage your development projects by keeping all information related to each project together in one place. You can edit, compile and link individual files or the entire project with a few keystrokes. BCLIDE knows the Byte Craft Limited Code Development System; it can generate files for the compiler's use and associate output files generated by the compiler.

You can call any Byte Craft Limited C compiler, the Byte Craft BClink linker, and any other software tools with a single keypress, menu, or toolbar control.

On-line help is available for BCLIDE and the compiler, and you can easily add frequently-used help files to the help menu.

BCLIDE’s editor is particularly flexible. You can create code templates for frequently-used structures and bind them to hotkeys. You can describe block boundary character sequences for quick navigation through your code. And you can set and navigate to bookmarks. All these settings are stored with your project for quick recall.

Best of all, BCLIDE is lightweight: startup times are under 3 seconds. Compare this with some other IDEs on the market…
Curious about "that other product" that Byte Craft Limited offers? Have trouble describing it to clients?

Fuzz-C™ is a stand-alone preprocessor that seamlessly integrates fuzzy logic into the C language. Now your clients can add fuzzy logic to their applications without expensive, specialized hardware or software. The preprocessor generates C code that is both compact and significantly faster than most current commercial fuzzy logic implementations. Fuzz-C provides a practical, unified solution for applications that can benefit from fuzzy logic control systems. Use existing C libraries for program management, keyboard handlers and display functions without change, and implement system control functions using fuzzy rules.

**Fuzz-C™ and Fuzzy Logic**

Crisp numbers (the kind we're all familiar with) simply indicate the magnitude of a measurement. Fuzzy values describe the 'truth' or 'falsehood' (or something in-between) of one or more qualities relating to a crisp value. These membership functions are collected in a linguistic variable declaration complete with membership functions.

```c
/* degrees Celsius */
LINGUISTIC room TYPE int MIN 0 MAX 50
{
    MEMBER cold  { 0, 0, 15, 20 }
    MEMBER normal { 20, 23, 25 }
    MEMBER hot    { 25, 30, 50, 50 }
}
```

Membership functions encode the “coldness” and “hotness” of different crisp temperatures. The reverse is true as well: the consequence variables (which provide us with an ultimate crisp result) relate fuzzy sets to crisp values.

```c
/* A.C. on or off */
CONSEQUENCE ac TYPE int DEFUZZ CG
{
    MEMBER ON    { 1 }
    MEMBER OFF   { 0 }
}
```

These declarations give us the makings of fuzzy logic rules. These rules test the 'truth' or 'falsehood' of the linguistic variable, and specify consequences accordingly.

```c
/* Rules to follow */
FUZZY climateControl
{
    IF room IS cold THEN ac IS OFF
    IF room IS normal THEN ac IS OFF
    IF room IS hot THEN ac IS ON
}
```
Fuzzy logic starts with crisp values, analyzes them according to membership functions, and then uses consequence functions to arrive at a crisp result. There's nothing indeterminate about fuzzy logic: given the same input, you get the same output every time. *The difference is in the ease of coding the algorithm.*

```c
int main(void)
{
    while(1)
    {
        /* find the temperature */
        room = thermostat;
        /* apply the rules */
        climateControl();
        /* switch the A.C. */
        airCon = ac;
        wait(10);
    }
}
```

Using Fuzz-C fuzzy logic in a program is easy. The LINGUISTIC variables (and CONSEQUENCE blocks alike) become variables in your program. Fuzzy logic rules become a function that you call when needed to perform the fuzzy calculation.

Fuzz-C is a flexible system that allows all (scalar) data types supported by your C compiler. It includes standard defuzzification methods, and accepts user-defined ones.

**Fuzzy Logic and C in RS08**

Even in the smallest applications, fuzzy logic can assist in implementation. Here are some suggestions on implementing fuzzy logic on RS08:

- Choose appropriate data types. Fuzzy logic passes its information with **Degree of Membership** values. The underlying type needs to have sufficient dynamic range for the problem at hand, but this isn’t exclusively decided by the input or output types. An unsigned char is a perfectly acceptable degree of membership type.

- Use a fuzzy function as a soft thread. Cause the fuzzy function to run whenever the crisp input variables have a new value. Fuzz-C’s architecture performs only as many calculations as necessary to arrive at a fuzzy result. For example

  ```c
  #pragma thread checkcount WAIT (SIPl.RTI == 1);
  ``

  ```c
  LINGUISTIC MTIMCNT TYPE unsigned char
  {  
    MEMBER low { 0 , modulo/4, modulo/2 }
    MEMBER mid { modulo/4, modulo/2, 3*modulo/4 }
    MEMBER hi { modulo/2, 3*modulo/4, modulo }
  }
  FUZZY checkcount
  {  
    IF MTIMCNT IS low THEN adjust IS up;
    IF MTIMCNT IS high THEN adjust IS down;
  }
  ```
Fuzzy PID controller

This is an excerpt of a fuzzy logic PID controller implemented using Fuzz-C. You can see by the fuzzy rules that the fuzzy kernel is processing the proportional, integral and derivative of the input to drive the output.
/* Fuzz-C Fuzzy Logic Preprocessor
   Fuzzy PID Controller
   Copyright 2005, 2006 Byte Craft Limited
*/
/* for calculating derivative error */
int OldError;

/* external set point */
int Setpoint;

/* the process that reads the ManVar and updates Error */
int Process(void);

/* proportional error */
LINGUISTIC Error TYPE int MIN -90 MAX 90
{
    MEMBER LNegative { -90, -90, -20, 0 }
    MEMBER normal    { -20, 0, 20 }
    MEMBER close     { -3, 0, 3 }
    MEMBER LPositive { 0, 20, 90, 90 }
}

LINGUISTIC DeltaError TYPE int MIN -90 MAX 90
{
    MEMBER Negative { -90, -90, -10, 0 }
    MEMBER Positive { 0, 10, 90, 90 }
}

LINGUISTIC SumError TYPE int MIN -90 MAX 90
{
    MEMBER LNeg { -90, -90, -5, 0 }
    MEMBER LPos { 0, 5, 90, 90 }
}

CONSEQUENCE ManVar TYPE int MIN -20 MAX 20 DEFUZZ cg
{
    MEMBER LNegative { -18 }
    MEMBER SNegative { -6 }
    MEMBER SPositive { 6 }
    MEMBER LPositive { 18 }
}

FUZZY pid
{
    /* large moves for large proportional errors */
    IF Error IS LNegative THEN ManVar IS LPositive
    IF Error IS LPositive THEN ManVar IS LNegative

    /* small moves for changes in error */
    IF Error IS normal AND DeltaError IS Positive
    THEN ManVar IS SNegative
    IF Error IS normal AND DeltaError IS Negative
    THEN ManVar IS SPositive

    /* small moves for large sums of accumulated error */
    IF Error IS close AND SumError IS LPos
    THEN ManVar IS SNegative
IF Error IS close AND SumError IS LNeg
   THEN ManVar IS SPositive
}

void main (void)
{
   while(1)
   {
      OldError = Error;
      Error = Setpoint - Process();
      DeltaError = Error - OldError;
      SumError := SumError + Error;
      pid();
   }
}

*Look at the complete source and listing files in the accompanying BCLIDE project.*
Our compiler technology is the product of twenty years' experience developing code generation strategies for different embedded platforms. We are experts in dealing with processor architectures designed for computer-generated code. Byte Craft compiler code generation technology produces tight, fast code. Byte Craft's tools equal or improve on the code density and execution speed of well-written hand-coded assembler.

Byte Craft Limited works with our silicon partners to develop benchmarks, leading to design wins and silicon sales. Many sales initiatives start with benchmark tests. Get Byte Craft Limited involved in this process; we consistently win benchmarks, and we can demonstrate through this process that we work with our silicon and hardware partners in the best interest of our common customers.

Byte Craft Limited can provide marketing materials, demos, and product information. Our demonstration products generate listing files, with which users can see the code we generate. We can provide machine-readable copies of all of our materials, in a wide variety of electronic formats.

Byte Craft Limited can contribute to joint presentations at customer sites. Byte Craft Limited staff travel around the world for client meetings, sales presentations, and standards body participation.

Byte Craft Limited is located in Waterloo, Ontario, Canada. We are in the Eastern Standard Time zone (EST/EDT, UTC-5).

Download demonstration versions of our products at:
http://www.bytecraft.com/depot/depot.php

Demonstration versions do not create executables, but do create listing files to allow inspection of generated code. They come with full documentation and tutorials.

Industry Presence

Byte Craft Limited sends representatives to selected embedded systems conferences. We decide to attend trade shows on a case-by-case basis, and frequently present in conference sessions. Contact us for more information.
Walter Banks participates in meetings of ISO WG-14, the working group responsible for TR 18037, *Extensions for the Programming Language C to support Embedded Processors*. Walter's leadership has helped advance embedded systems' support in the C language standard.
Publications

We have significant experience in publishing. Our website features two publications free for download:

First Steps in Embedded Systems is a comprehensive general text introducing embedded systems. Fuzzy Logic in Embedded Microcomputers and Control Systems provides a tutorial on fuzzy logic as it pertains to control systems. Both publications can be downloaded from our website in PDF format.

To download: http://www.bytecraft.com/publishing.html

Product Development

Feedback and Improvements

Product development and technical support are intimately linked. The staff members responsible for developing our Code Development Systems also provide front-line technical support. Support calls or correspondence that reveals issues with the compiler result in improvements to all of our products.

Toolchain Innovation

We work actively with simulator and emulator vendors to complete the developers' toolchain. The Byte Craft Limited .COD file format and BCDIRECT embedded connectivity links allow the compiler to communicate all necessary information to downstream tools. We share standards documents on these technologies freely with other vendors where mutual advantage exists.

Standards

Byte Craft Limited Code Development Systems conform to ISO standards for C as much as possible for the target hardware platform. In many cases, a full implementation of standard C is not possible.

A C standard for embedded systems (ISO Technical Report 18037) now formally makes all of the processor registers, address spaces and internal flags available to the C programmer. This makes it possible to match any assembly language statement in C. By writing code in C and not as assembly, every line of a program can benefit from the compiler's optimization and data-flow analysis. This is a big advantage to developers.

Byte Craft's president Walter Banks represented Canada at ISO and was a major player in the development of these standards.
About Byte Craft Limited

Byte Craft Limited is a Canadian company founded by Walter Banks, Jay Majithia, and Surin Kalra in July 1976. Walter Banks became the sole shareholder of Byte Craft Limited in May 1979. Walter Banks remains President of the company to this day.

In the early 1980's, Byte Craft Limited provided consulting development services for embedded systems in industrial, commercial and consumer products. Some of our achievements include the first mouse for the IBM-PC personal computer, environmental data loggers, and early personal data assistants.

Byte Craft Limited developed software tools to support our product development. The software tools we developed grew into a more important part of our business than original product development.

Byte Craft wrote over 90 assemblers in the 1980's, and shipped our first commercial compiler in 1984. We shipped our first C compiler in 1987.

Today, Byte Craft Limited is a software company specializing in development tools for embedded systems processors with unusual architectures or limited resources. Our main products are C cross-compilers targeted to a variety of microcontroller families. We provide innovative solutions for developers, consultants and manufacturers around the world.

Byte Craft Limited leads the industry in embedded software engineering solutions; we produced the first C compilers for new platforms released by many of the leading semiconductor manufacturers worldwide. Byte Craft Limited tools are used by developers in automotive, telecommunications, consumer-product, industrial, and aerospace industries.

We have more than twenty-five years' experience in Embedded Systems, and significant experience in publishing and document management. We are devoting our efforts to putting excellent tools and knowledge into developers' hands.

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Apr. 2006</td>
<td>KZ</td>
<td>Initial version</td>
</tr>
<tr>
<td>2</td>
<td>Oct. 2006</td>
<td>KZ</td>
<td>Release</td>
</tr>
</tbody>
</table>

Copyright © 2006 Byte Craft Limited. All rights reserved.