HyperLogic

TECHNICAL NOTES

A series of informal papers on technical topics for customers and friends of HyperLogic Corporation

Number 4
Revision 4

Developing Embedded Systems with CubiCalc RTC's Fuzzy Run-time

When you're trying to meet both time and space constraints with a limited-capability processor, you can't afford to fight tools too. CubiCalc RTC was designed to gracefully handle certain problems faced by programmers developing ROM-based embedded systems. This note describes those aspects of the RTC run-time architecture.

Run-Time Organization

The CubiCalc RTC run-time system consists of a run-time generator and an inference engine that evaluates fuzzy rules.

The run-time generator doesn't create an executable program or a C function — it produces only symbols and pre-initialized data structures (C structs and arrays) defining the rules and membership functions, plus a few other useful data items. The executable code — the actual inference engine — is a separate data-driven library that works with the generated data structures.

The inference engine is written in C. It's provided in object code form in the standard RTC product, with source code available separately. The object code libraries provided with CubiCalc RTC support both Microsoft and Borland C and C++ compilers for 80x86 processors. If you are using a different compiler or processor, you need the run-time source code so you can compile it for your target.

Since the executable code is separate from the system-specific definitions, only data structures change from one project to another. This approach has many advantages over the alternative of generating in-line executable code, particularly for embedded system development.

Code Management

Because the RTC inference engine is common to all your projects, you can work with the executable code independently. The code stays the same when you change rules or fuzzy sets, so any tweaking is a onetime effort. You don't have to keep editing the code to make the same changes over and over again.

This is not so great an issue in a more luxurious environment, but embedded system developers don't have it so easy. Your compiler may be limited somehow. It could be as new as the processor (Version One Point Uh-Oh). And you might not have any choices. The chip supplier could be the only one that supports the processor!

So separating the executable code and keeping it stable can be a real advantage. You might need to insert special #pragma statements to fine-tune optimization or control the memory layout. Or you might need to work around a compiler bug or limitation. You could even translate a portion into assembly language if necessary.

With the RTC run-time, you can optimize the code, compile it, and test it once then use the object code again and again. If your compiler is updated but is still compatible with prior object code, you don't need to worry about new compiler bugs breaking the existing code — just keep using the same precompiled library.

Symbol Management

The RTC run-time is careful to avoid symbol conflicts, one of the bugbears of purchased components.

Maximum use is made of the "static" keyword to hide symbols, both in generated data structures and in the run-time library. In most cases, only a single global symbol appears outside the data structure file. In the run-time library, user-callable functions are prefixed with "fz_". The few non-static internal functions have an initial underscore.

If you combine multiple rule bases within a single executable image, you can define your own symbol prefix to distinguish them. For example, you might have different rule bases for two motors. You could assign prefixes of "M1_" and "M2_" to ensure that all the generated symbols differ. The run-time generator then uses the prefix on all of the project's definitions.

Even if two rule bases use different inference methods, there are no symbol conflicts in the fuzzy executable code itself. The code for all the inference methods can coexist in the same executable image.

Memory Layout

The RTC run-time specifically distinguishes ROM-able information from that which must be in RAM. Of course, the code goes in ROM. Almost certainly the generated data structures are in ROM. But the input and output variables must be writable! And the fuzzy computations need some scratch space (usually 20 bytes or less).

You set up the writable structures yourself — they're "outside" the fuzzy module. So you can allocate them dynamically and reuse the space for something else in between your calls to the inference engine. Or you can just use regular static or automatic C variables. Either way, you decide for yourself.

If you have two different ROM speeds or banked ROM, you can even put the code and data in different areas. The code can go in faster ROM or you can bank-select the data structures for the current rule base.

Predicting Requirements

Resources are always tight in embedded systems: processor cycles, RAM, ROM, schedule time. You can't afford surprises. And the RTC architecture gives you predictability that you just can't get with a tool that generates executable code.

Since the run-time code is fixed, when you've compiled it one time you can tell how much ROM space it takes; it won't change when your rules change.

The only other ROM space needed is for the generated data structures. The CubiCalc RTC "Compile" dialog shows you how much ROM is needed for the options you've selected. So you can make your trade-offs up front as part of your design and test process.

If RTC generated in-line C code, whenever your rules changed you'd have to produce the run-time, then compile it and study listings or the link map to see the ROM usage.

RAM requirements vary by only a few bytes as your project changes. The run-time generator creates a C "include" file that has, among other things, a #define for the amount of RAM scratch space.

When you're computing required resources, another issue is stack space. Since the run-time code is always the same for any particular inference method, you can compute stack requirements once instead of studying generated in-line functions every time your rules change.

Mixing Inference Methods

When you execute fuzzy rules, the algorithms are similar from one system to another, but general inference methods differ slightly in the way they perform some steps. There's a code base needed to support each particular method, but most of the code is common to all.

The run-time generator and the organization of internal functions in the library work together to ensure that only *needed* code is included in the final system. There's no repetition of the common code. And with standard library utilities, there's no unused code either.

So if you add another rule base to your program, the incremental memory cost is just the size of the data structures for the new rule base and one or two small functions for another inference method.

Reentrant Run-Time Code

The RTC run-time is completely reentrant. When you call the fuzzy inference engine, you pass it the address of a data structure you've set up; everything it does is based on this structure. There are no static variables.

This is useful if you have multiple tasks doing fuzzy control. For example, you might have multiple independent rule bases for different devices. Or you might use the same rule base to drive multiple instances of the same general type of device. Either way, there's no need for a semaphore to force "single-threading" through the inference engine.

Run-Time Source Code

The RTC fuzzy inference engine is pure computation, so the source code is independent of the target environment. No C run-time functions are needed (except possibly math functions in the floating point version); it doesn't do any file I/O or memory allocation.

The integer run-time requires very little of the processor and runs quite well on 8-bit micros. Most of the code uses only 8-bit arithmetic, though some methods need a few 32-bit *long* operations. The usual data type is *unsigned char*, but you can change it and customize the run-time generator for your own choice.

The C source code is conditionally compilable for either ANSI or the older K&R style. Generated C data structures are compilable with either.

Simulation Facilities

The initial design and analysis stage is really important when you're facing tight constraints or controlling an expensive piece of equipment. The interactive shell in CubiCalc RTC lets you do a lot of work before you generate the run-time. It is much more than just an editor for writing fuzzy rules and converting them to code.

Many fuzzy tools claim to support simulation when they just run the fuzzy rules and create a graph from the results. But CubiCalc has a built-in programming language that lets you simulate *the responses of the target system*. Or you can compute error, preprocess input data, post-process output results, or whatever.

With such a rich environment, there's often no need for further tuning in the target system where resources are so limited. And if someone else is writing the rules, he need not be a programmer to use the interactive shell.

Embedded system programmers don't necessarily care much about programming under Windows. But you might want to use CubiCalc RTC's Dynamic Link Library version of the inference engine for quick proof of concept demonstrations using Windows-based rapid development tools such as Visual Basic. The DLL can execute rules in integer, so you can see effects of discretization in the prototype.

Summary

CubiCalc RTC is a general purpose fuzzy development tool with excellent support for embedded system programming. Some of its advantages are those of CubiCalc itself. But the run-time system was designed with careful attention to the constraints of programming real-time ROM-based systems on low-end processors.

HyperLogic Corporation, PO Box 300010, Escondido, CA 92030-0010 USA
http://www.hyperlogic.com — Telephone (760) 746-2765 — Fax (760) 746-4089