Ultra-Low-Power Application Development with RCore C and Assembler

Alan Rooks, Etienne Cornu

AMI Semiconductor Inc. | www.amis.com

Abstract

Assembly language programs are efficient, but slow to develop and difficult to maintain. A higher-level language like C can resolve these issues, but for ultra-low-power applications where efficiency is critical, how can the developer contend with the C compiler's relatively inefficient code generation, in a modular, maintainable way? This paper shows how the RCore C compiler allows C and assembler interfaces to be combined cleanly, for rapid development of ultra-low-power applications that are at once efficient and maintainable.

Introduction

Assembly language is the obvious design choice for ultra-low-power application development based on a system with a programmable controller core such as the RCore DSP (1), because the most efficient code can be written. However, there are well-known problems with assembler:

1. It is too time-consuming for prototyping, development, test, and maintenance.
2. Clean interfaces—which promote modularity, maintainability, and code re-use—can be difficult to design and enforce in assembly language.
3. The learning curve for effective assembler programming is too steep for engineers who are new to the system. For a new design with an embedded RCore, development engineers must understand the system's peripherals, I/O interfaces, and possibly other embedded cores, in addition to programming the controller core (RCore). The situation is similar for customers using ASSPs, such as BelaSigna® 250, where the WOLA coprocessor is a second large block to absorb.

The C programming language (2) could help with these issues, but some fundamental obstacles to using C for highly resource-constrained DSP applications are:

1. Many DSPs, including RCore, have an instruction set and memory architecture that are not a good fit for the C language, so the generated code is sub-optimal.
2. The data types and operations (e.g., multiply-accumulate) of the DSP are different from those of C, so effective signal-processing code is difficult to write in C.
3. If there are existing libraries that are optimized for use in assembly language (math routines, hardware control), or existing signal-processing modules written in assembler, these can be difficult to assimilate with C calling conventions, data layout, and register usage.

The RCore C compiler (3) was designed to encourage and support integration with assembly-language modules and interfaces to reap the benefits of both environments. This paper shows—with a BelaSigna 250 signal-processing example—how to use RCore C and its assembly-language "glue" for development of ultra-low-power applications that are:

- Cycle-efficient for signal processing.
- Quick to prototype, develop, test, and change.
- Modular and maintainable.

Problem Domain Example: BelaSigna 250 Applications

Audio applications for BelaSigna 250 provide examples of real-world embedded programs with ultra-low power budgets, running on an embedded RCore DSP. Such applications are typically difficult to implement with a C-based design for the reasons stated above, so the BelaSigna 250 environment provides a challenging target for a port to RCore C. This section gives background on the problem at hand by providing a brief overview of the BelaSigna 250 hardware and its assembly-language software framework.

A. The BelaSigna 250 Chip

BelaSigna 250 is an ultra-low-power audio processing system developed by AMIS for portable audio applications (4). Fig. 1 shows an overview of the blocks comprising BelaSigna 250.

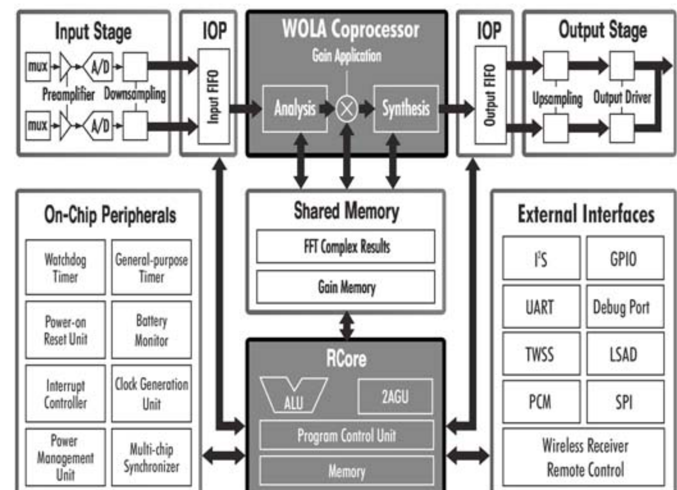

Fig. 1 BelaSigna 250 Block Structure

© 2007 AMI Semiconductor, Inc.

The RCore DSP is the system controller and performs general-purpose signal processing functions, while the WOLA coprocessor is dedicated to vector-based filtering computations. The mixed-signal chip also contains full audio input and output stages, several peripherals and I/O interfaces, and other dedicated subsystems. Reference (5) describes how this architecture is used to improve sound quality in audio headsets.

B. BelaSigna 250 Application Software Framework

The BelaSigna 250 hardware architecture is mature, so a significant body of software expertise and infrastructure exists; all of it written in assembler or intended for interfacing with assembly programs (6). In summary, the software is layered from the bottom up as follows:

- A "hardware symbols" header file gives names to all addresses, bit positions, and numeric values related to the chip.
- Low-level libraries with associated headers provide interfaces for math primitives, peripherals, interfaces, and the WOLA coprocessor.
- Code blocks or modules are larger-grain components, such as filters, that are built from the primitives.
- Algorithms are code sets that combine lower-level components to implement high-level functionality such as noise reduction or echo cancellation.
- Applications group the algorithms and other controlling code into a functioning system.

C. Creating a BelaSigna 250 Application

The BelaSigna 250 software framework demonstrates that with time and effort modularity *can* be achieved in assembler. This highly structured assembly environment presents a challenge (and an excellent example) for integration with C; see **Hoisting BelaSigna 250 Interfaces into the C World**, below.

In the context of the BelaSigna 250 software framework, an engineer developing a new application for BelaSigna 250 would typically be implementing a new algorithm (such as a decoder for the AAC codec) and creating or modifying an application to integrate the new algorithm with existing ones. These activities—which could occur in an AMIS design team or a customer's development lab—require knowledge and experience with the RCore DSP, other processors and peripherals in the system, the software framework and how it supports parallel processing for high performance and low power, and the characteristics of the existing algorithms, *in addition to* the specific expertise with the new algorithm.

This learning curve is not particular to BelaSigna 250; it is typical of RCore-based applications which are resource constrained. Climbing that learning curve by creating new algorithms in RCore assembly language is a daunting task for developers, especially if the algorithm is specified (or a reference implementation is available) in a language like C.

The RCore C Language

The RCore C compiler implements ISO C90 (the 1990 C standard) plus several extensions for the RCore DSP, but with no floating-point support. It generates code for the BelaSigna 250 version of the DSP core (1), and is incorporated in the SignaKlara® Tools Integrated Development Environment (7). RCore C is a *freestanding* implementation; C library facilities that depend on an operating system are not provided (3).[1]

A. The Problem: Inefficient Signal-Processing Code

The RCore C implementation achieves the basic goal of a C compiler: code portability. For example, a portable Bluetooth protocol stack was compiled with no substantial modifications. However, as expected, the generated code is less than optimal in some key areas:

- Small loops are slow because the `REP` instruction (for single-instruction loops) is not generated.
- Multiply-accumulate (MAC) loops do not use the optimal pipelined approach, so each MAC takes several cycles instead of a single cycle.
- The accumulator's guard bits (the AE register) are unavailable in C, so if guard bits are needed, all operations must be done using 32-bit `long` values.
- Accessing stack-based data such as function parameters and local variables is cumbersome because the RCore's instruction set has no stack-offset addressing, and stack address computations displace expression results from the accumulator.

These deficiencies are generally not critical during the prototyping stage of development when correct, bit-true results are more important than performance.

In a worst-case scenario such as a sum-of-products loop in a filter, these issues can compound to produce code that is up to two orders of magnitude slower than the best assembly code, and uses much more program memory. Small sections of code with large performance issues like this typically occupy 5% of the code while using 95% of the execution time (embedded signal-processing applications tend to follow a more extreme form of the 90/10 rule), so they are obvious areas for improvement when development moves into the production phase. RCore C facilities for enhancing performance are described in the following sections.

B. Facilities for Improving Generated Code

Several extensions to standard C allow developers to improve the performance of the generated code. They can all be used conditionally (using the C preprocessor) so that the code still compiles with a workstation-based compiler for offline exploration, and can be compiled without optimizations for the

[1] To date no operating system has been written for any RCore-based device, to the authors' knowledge, because memory and CPU cycles are so limited.

© 2007 AMI Semiconductor, Inc.

purpose of regression-testing the optimized code. In brief:

- The `register` keyword is more flexible: both local and global variables can be bound to specific machine registers. The first two function parameters are passed in registers as well, which reduces the accessing overhead.
- Functions can specify that local variables be stored statically to reduce overhead for stack accesses.
- Data can be declared in all three memory spaces and can be further declared to reside in low XMEM or YMEM, for quick access.
- Control-flow extensions support low-overhead looping and indirect branching with jump tables.

C. Interfacing with Assembly Code

When code uses data types and machine structures that do not fit the C model—such as the sum-of-products example above—assembly language can be used to achieve the performance requirements. The traditional method for integrating assembly modules—where assembler code is separately compiled, linked into the executable and called from C—suffers from the following drawbacks:

Complexity – the code to handle the compiler's function-calling sequence often involves managing many details about the stack and register usage. The details are often bypassed by compiling (to assembler) a skeleton version of the function in C to use as a starting point, but even this approach can fail when the details of the calling sequence depend on the registers used, whether the function calls another function, etc.

Susceptibility to change – changes to the parameter list, compile-time options, or even a new version of the compiler can require small changes throughout the assembly module. Stack offsets and register usage can change, often with no warning given by the compiler or assembler.

Lack of modularity – the assembly module is in a separate source file; therefore, it is not right at hand when the C code that calls it is edited, and vice versa. No automated check ensures that the C declaration of the function matches its assembler implementation. The assembler function must be external, so C's file scope cannot be used to hide the function from code that need not be aware of it.

Coarse granularity – assembly-language components can only be accessed through C's function-call interface. If the C programmer really just needs to execute one or two machine instructions, the overhead of a function call is too great.

To address these issues, RCore C provides *inline assembly language*, which integrates assembly-language code directly into a C-code file. Inline assembly language usually appears within a C function body, and is written within special delimiters, similar to a multi-line comment:

```
/$
    CLB A          // Normalize A
    SHFT A
 $/
```

Fine-grain assembler use is obviously supported. However, the real power of inline assembler comes from its symbolic connections to the C code in which it is embedded. Assembly instructions can refer to C elements as follows:

@variable – expands to a register name for register variables, a stack offset for function parameters and local variables, or a label for statically-allocated variables (and functions).

@sizeof(expression) – expands to a constant with the value that the C `sizeof()` operator would return for the expression.

@struct-type.member – produces the offset of the member within its `struct` type. Nested `struct` members are allowed.

One common idiom with inline assembler is to define a function in C, but implement the whole executable body of the function with assembly code:

```
int func(register int a,
         register XMEM int *xp)
{
    register YMEM int *yp;
    register int r @ REG_AH;
    /$
        // Assembler using @a, @xp, @yp, @r
        ...
     $/
    return r;
}
```

This is an alternative to the traditional approach (separate assembler functions) that resolves all of the drawbacks of that approach. The C compiler manages the stack frame and register allocation, while the assembly code is tightly but symbolically integrated with the surrounding C code in the source file, so it is robust when changes occur. The compiler also checks calls to the function for correctly-typed arguments.

Interrupt handlers can be defined in RCore C by using the `_INTERRUPT` keyword in a function definition. This allows the developer to implement first versions of low-level interrupt handling code in C, without worrying about which registers to save, how to save and restore status, or re-enabling interrupts. Interrupt functions also work well with inline assembler.

Hoisting BelaSigna 250 Interfaces into the C World

We now look at porting the assembler interfaces of the BelaSigna 250 software framework to C, using the facilities of RCore C described above. We will walk through the layers as they are described in *BelaSigna 250 Application Software Framework* above, looking at examples from each layer.

The hardware symbols header file was designed for use in assembly language, but the symbols are useful in BelaSigna 250 C programs, too. It can be `#include`'d in the usual way:

© 2007 AMI Semiconductor, Inc.

```
#include <sk25_hw.inc>
```

All of the symbols can now be used in the C program. This "cheat"—using an assembly-language header file unchanged in a C program—illustrates a final aspect of RCore C's rich facilities for cooperation with assembly language: the RCore assembly language tools use (and have always used) the same C preprocessor to provide assembler macros, symbolic names for constants, conditional compilation and file inclusion, as the RCore C compiler. This C preprocessor provides the usual standard facilities, plus extensions like multi-line macros to facilitate its use for macro assembler programming. These are available in C programs as well as assembly language.

The second layer—low-level libraries for math primitives, system facilities and the WOLA—are accessed similarly, using `#include` to pull in the assembly-language headers:

```
#include <bat.inc>
#include <boss.inc>
#include <wola.inc>
```

These headers contain symbolic names like the hardware symbols header, but they also implement assembler macros and have associated libraries containing assembled functions. The assembler macros can be used directly in a C function, with inline assembly. Sometimes it is easiest to simply steal existing assembler code and use pieces of it in C:

```
void SystemInitialize(void)
{
  /$
        // Set up interrupt vectors
    Set_Int_Vect(D_VECT_IOBLOCK, _iop_isr)
        // Analog block configuration...
        // High voltage mode
    Write_AReg(A_PSU_CTRL, POWER_CFG)
    Sys_Delay(100)
        // Configure sampling frequency
    Write_AReg(A_ADC_CTRL, ADC_CTRL_CFG)
    ...
        // Configure and start IOP
    Set_Iop_Cfg(IOP_CFG)
  $/
}
```

Library functions can be accessed similarly. Macros that contain `.extern` directives allow the function to be called from inline assembler, while a C declaration for a function must be written to call it from C. A wrapper combines the two:

```
int BAT_db_rms(register int db_val)
{
    register int result @ REG_AH;

    /$
        RES ST, 7       // Signed mode
        LD AH, @db_val  // Get parameter
        DB_RMS          // Call _db_rms
     $/

    return result;
}
```

Here, a wrapper function invokes the `DB_RMS` macro from the Basic Algorithm Toolkit (BAT); the macro expands to a call to the library's `_db_rms` function that the macro declared with a `.extern` directive. The rest of the C application uses the C interface `BAT_db_rms()` to call the BAT function.

The third layer in the BelaSigna 250 software framework consists of existing computational blocks (e.g., filters) that can be integrated in the same fashion as the low-level libraries. New blocks can be added, using C and assembler as desired or convenient, in the style of the previous examples.

A full BelaSigna 250 application can now be assembled. If a new algorithm (fourth layer) is to be implemented for the application, its routines will likely be prototyped in C to speed development. Once the algorithm is behaving correctly, the time-consuming sections can be optimized as necessary.

The application layer integrates code and data from the various algorithms that the application uses. The application's `main()` function typically executes initialization routines for the algorithms and the system as a whole, and then enters its main loop that coordinates execution of the various routines. These are usually interleaved with interrupt handlers, which can also be written in C for tight integration with the application.

In summary, the assembler interfaces of the BelaSigna 250 software framework were not in fact *ported* to C, in the sense of being re-written in C. Rather, they were directly integrated with C code, as is. The rich inline assembly language facility and the shared C preprocessor allow RCore C and assembly language to be integrated seamlessly. This combined approach accelerates initial development as well as performance tuning. Optimized code is modular and maintainable because related elements are kept together and the strengths of the two languages are leveraged: the efficiency and notational brevity of macro assembler programming, with the automatic code generation and type checking of C.

Benefits of C for Testing and Teaching

The preceding sections have focused on selective optimization of RCore C programs to satisfy the efficiency requirements of production ultra-low-power application code. The benefits of C through the development lifecycle have been discussed—exploring pre-production algorithms on the hardware entirely in C, optimizing incrementally, simpler maintenance—but always with a focus on dealing with the inefficiencies of C by optimizing, typically using assembly language.

Some of the code that is associated with an ultra-low-power application *never* needs to be optimized, however. Using C for these code sets is especially beneficial since they are not shipped as part of the application or product, and thus need not be optimized:

- Test scaffolding code and unit-test modules often run on test jigs or development platforms where power consumption and timing are not critical;

© 2007 AMI Semiconductor, Inc.

correct output results are the key.

- Reference versions of code blocks or whole algorithms are typically written entirely in C for portability and ease of understanding. This original form can be retained and used for regression-testing optimized versions of the code.
- Utility programs for product setup, configuration, calibration, or problem diagnosis usually do not have strict performance requirements.
- Example or template code must be clear and easy to understand, primarily. It might be provided by an internal design team to a customer as a starting point for the customer's product integration efforts (involving optimization), for example.

C is an obvious benefit for these programs, to further accelerate application development, test, and maintenance.

Conclusions

We have investigated the benefits of RCore C in the development of ultra-low-power applications. Poor code generation is a common problem for C implementations on DSP cores, so we evaluated RCore C's optimizing extensions and assembly-language integration using the BelaSigna 250 application software framework. Our goal was to determine whether RCore C is an effective language for development of real-world ultra-low-power applications that are very efficient, yet still modular and maintainable.

We conclude that RCore C is an excellent tool for developing ultra-low-power applications because its extensions and seamless integration with assembly language provide the benefits of both environments:

- Rapid prototyping, development, testing, changing.
- Modular, re-usable, maintainable code.
- Ultimate efficiency, when necessary and without the usual drawbacks of separately-compiled assembler.
- Ability to use existing assembler-based interfaces directly in C.

References

(1) AMI Semiconductor, *RCore DSP Architecture Manual*, SignaKlara Tools EDK 4.1, 2007.

(2) B. W. Kernighan and D. M. Ritchie, *The C Programming Language, 2nd Edition*, Prentice Hall, 1988.

(3) AMI Semiconductor, *RCore C User's Guide for BelaSigna 250*, SignaKlara Tools EDK 4.1, 2007.

(4) AMI Semiconductor, *Hardware Reference Manual for Orela® 4500 and BelaSigna 250*, SignaKlara Tools EDK 4.1, 2007.

(5) K. Tam and E. Cornu, "System architecture for audio signal processing in headsets", *Proc. of the Global Signal Processing Times Conference (GSPx)*, 2005.

(6) E. Cornu, T. Soltani and J. Johnson, "A framework for automatic generation of audio processing applications on a dual-core system", *Proc. of the Global Signal Processing Times Conference (GSPx)*, 2005.

(7) AMI Semiconductor, *Integrated Development Environment User's Guide*, SignaKlara Tools EDK 4.1, 2007.

For more information and sales office locations, please visit the AMI Semiconductor web site at: www.amis.com

Worldwide Headquarters
AMI Semiconductor, Inc.
2300 Buckskin Road
Pocatello, ID 83201 USA
Tel: +1.208.233.4690

European Headquarters
AMI Semiconductor Belgium BVBA
Westerring 15
B-9700 Oudenaarde, Belgium
Tel: +32 (0) 55.33.22.11

Copyright © 2007 AMI Semiconductor, Inc. All rights reserved. AMI Semiconductor, BelaSigna, Orela, and SignaKlara are either trademarks or registered trademarks of AMI Semiconductor, Inc. All other brands, product names and company names mentioned herein may be trademarks or registered trademarks of their respective holders.