A High-Temperature Embedded Network Interface using Software Thread Integration

Alexander Dean\textsuperscript{1} and Dr. R. R. Grzybowski
United Technologies Research Center
East Hartford, CT, USA
deanc@utrc.utc.com

This document describes an experimental implementation of a communication network protocol bridge for a high-temperature (200°C / 392°F) application. The software uses a novel compilation technique called Software Thread Integration (STI) to interleave multiple real-time program threads for efficient concurrent execution, allowing an off-the-shelf, high-temperature, low-speed 8 bit microcontroller to translate messages between a CAN bus and a serial UART data link. The prototype system, which has been built and tested, demonstrates the feasibility of using STI in real-world applications. STI provides the techniques needed to automate the tedious hand-scheduling of time-sensitive code from multiple simultaneous threads so common in cost-driven microcontroller-based applications.

1 Introduction

Software Thread Integration (STI) \cite{DS97, DS98a, DS98b} is a new technology for interleaving program threads and is especially suitable for performing real-time operations. STI produces a single thread which does not need to switch between contexts, yet provides efficient, deterministic performance. This technique complements and improves upon today’s ad hoc methods of hardware to software migration by providing a method for automatically creating efficient real-time concurrent code. In this work we use STI to create a program for an 8051-compatible microcontroller which functions as a communication bridge between a 10 kbaud CAN network \cite{Bos91} and a 9.6 kbaud serial link. Previous work in STI has focused on integrating functions in multimedia applications from simulated systems using high-end 32- and 64-bit microprocessors. This current work presents a complete application which has been built and tested at high temperature using a low-end 8-bit microcontroller, demonstrating STI’s suitability for a wide range of applications.

The ultimate goal of this research is to produce a software development environment which allows programmers to write and maintain individual real-time threads. The tool will automatically merge, transform and pad these threads to create integrated object code which meets all functional and real-time requirements. Because this automation allows developers to work on software at a higher level, systems will be developed more quickly, and more importantly, more complex systems will become feasible, allowing migration of more sophisticated functions into software.

1.1 Hardware to Software Migration

Designers of high-temperature embedded systems (HTES) are constrained by the scarcity of semiconductor components capable of operating in a harsh environment (200°C / 392°F). Low production volumes lead to high prices due to the lack of economies of scale. In addition, these components tend to be mature designs, which prevents HTES designers from benefiting from recent advances in technology. A common alternative to designing exclusively with HT parts is to provide a cooling system to moderate the environment, allowing standard-temperature components to be used. These solutions range from simple (using a larger heatsink) to complex (cooling electronics with jet fuel in an aircraft engine). Many applications are unable to bear the added cost, size, weight, complexity or reliability risk of such a cooling system, leaving high-temperature components as the only option.

Moving a real-time function from dedicated hardware to software (Hardware to Software Migration, or HSM) running on a microcontroller allows a wide variety of processing to be performed, limited only by the speed of the microcontroller and the performance of the software. The potential benefits of eliminating hardware components include better component availability, higher reliability, smaller size, lower weight and lower cost. Using more software can increase function availability and flexibility, and reduce time to market and system redesigns resulting from component obsolescence.

Two factors limit the use of HSM for high-temperature embedded systems. First, the microcontrollers available offer very limited performance (under two million instructions per second), constraining the range of real-time functions which can be

\textsuperscript{1} Also a Ph.D. candidate at Carnegie Mellon University, at adean@ece.cmu.edu
moved into software. Second, the traditional techniques for HSM are complicated and produce inefficient code, further reducing the feasibility of migration. An interrupt-driven approach not only incurs context-switch penalties which limit efficiency with frequent events, but can also introduce significant timing jitter in other tasks, which may be real-time and hence very sensitive to such variations. A busy-waiting solution is tedious to implement, as the code must be implemented in assembly language for timing precision and then measured and padded to ensure events are timed correctly.

1.2 Software Thread Integration

STI addresses these problems by providing a method of integrating multiple threads of control automatically and efficiently. It interleaves multiple threads to create a single integrated thread which offers low-cost or no-cost concurrency. This is useful for performing real-time operations in embedded systems [DS98a]. STI features a set of code transformations which operate on hierarchically-represented threads (in control-dependence graphs, CDGs). This hierarchical form makes control independence among code regions explicit and also allows summary information to be stored at each level. These features simplify code motion, which we use to provide concurrency among multiple threads.

STI of real-time threads involves moving individual instructions or regions of code from one thread (the guest) to execute at specific times in another thread (the host). To do this we begin by predicting the host’s execution schedule and transforming it as needed to regularize it enough to meet the guest thread’s timing jitter limits. The dataflow of each thread is analyzed to determine variable use and eliminate potential register reuse conflicts; after integration the variables are colored into registers. The guest thread is then concatenated to the end of the host thread, which is the degenerate or initial integration. Each region of guest code is then moved into the host to execute at a valid time. When greater timing accuracy is needed we use various code transformations to move the guest regions into subgraphs such as loops and predicates. We have developed a set of transformations [DS98b] which allow efficient motion into subgraphs while optimizing for memory size or speed.

2 Experimental Prototype

A prototype high-temperature automotive controller being developed at UTRC needed a CAN (Controller Area Network) interface, but no appropriate high-temperature hardware solutions exist. Hardware to software migration with traditional software techniques would lead to an unacceptably low CAN data rate on the available high-temperature microcontroller. STI was selected to implement the CAN interface in software.

Figure 1 below shows the hardware of the prototype HSCAN (Hot Soft CAN) system, consisting of a Honeywell HT83C51 high-temperature microcontroller [Hon98a], a memory system and a CAN physical layer interface. The microcontroller operates at 22 MHz, executing at most 1.83 million instructions per second. External memory contains an EPROM, which transfers the program to a high-temperature SRAM (HT6256) [Hon98b] for actual execution. The CAN physical layer is based on a standard room-temperature circuit [Phi95] and modified to use high-temperature components.

The software for HSCAN consists of two threads, one for bit-banged 10 kbps CAN communication (denoted CAN interface thread) and one for communicating with the controller across a 9600 baud serial link (denoted controller interface thread) as seen in Figure 2. The threads jointly manage a pair of message queues for message transmission and reception.

2.1 CAN Interface Thread

Controller Area Network (CAN) [Bos91] is a multimaster communication network protocol designed by Bosch for use in real-time embedded systems. Messages are short (0 to 8 bytes of data), contain an identifier which determines bus access.
through nondestructive arbitration, and provide strong error detection through a 15 bit cyclic redundancy check code and bit stuffing. CAN is popular in the European automotive market and industrial control applications world-wide. Bit rates range up to 1 MBaud. Designing CAN into a real-time distributed system is simplified by the lossless prioritized arbitration method, which allows the network to be modeled as a priority queue. Many protocol controller devices are available for operation at standard temperature ranges, but not at the extreme temperature of this project.

The CAN thread spends its time in one of three states: monitoring an idle bus, sending a message, or receiving one. The CAN bus monitor samples the bus at three times the bit rate (about every 33 µs) to ensure that when a message is received, sampling begins in the middle third of the bit. This timing requirement, combined with the CPU processing speed and code size, results in 22 to 30 µs (40 to 56 cycles) of idle time per 33 µs loop iteration. We use software thread integration to reclaim part of this idle time for controller communication.

The CAN message transmit and receive functions repeatedly call send and receive bit functions, which also perform protocol requirements such as CRC computation, bit stuffing or destuffing, and bus error detection. When transmitting a message, the CPU is idle for 50 to 66 µs (91 to 121 cycles) per 100 µs bit. We reclaim these idle times through thread integration. Note that these idle times fall as the desired CAN bit rate rises or the CPU speed falls.

2.2 Controller Interface Thread

The controller interface thread responds to commands from the serial port to transmit or receive messages and manage the message queue, as in Figure 3a. If an interrupt is used to detect incoming bytes, the service routine will create timing jitter (unless specific compensation code is added). During the longest CAN message, an interrupt service routine (which must be at least 8.7 µs long) could be triggered 13 times, leading to timing errors reaching 113.1 µs. This error will corrupt an incoming or outgoing CAN message. Instead, STI is used to integrate the controller interface routine with three functions of the CAN thread. STI allows the software to be structured in an easily developed and maintained form.

The controller interface thread monitors the UART for incoming message activity. When a byte is received, it must be unloaded before the next arrives to eliminate the possibility of receive buffer overrun. With a 9600 baud serial link, the controller interface thread must be called often enough to service the UART at least every 1.04 ms.

![Figure 3. Control Flow Graphs of Controller Interface](image)

We convert this thread into subthreads (Figure 3b), which are similar to the states of a finite state machine. Each time the controller interface thread is called, a complete subthread executes, possibly enabling a different subthread for the next call to the thread. This simplifies integration in two ways. First, the switching among subthreads restores the control-flow variability lost through integration. This makes it much easier to allow the individual threads to execute at their own paces, rather than in lock-step. Second, a subthread can easily be partitioned into smaller subthreads, which allows a thread to be adapted to fit into short sections of idle time as well as eliminating blocking due to nondeterministic inputs.

In fact, the controller interface thread uses both of these techniques. Each subthread is partitioned to last no more than 78 cycles and contains no more than one potentially blocking UART access. Each subthread with such an access is guarded to prevent its execution if the UART is not ready. This partitioning increases the number of subthreads between UART receive buffer servicing, so the thread must be called more frequently to avoid overruns (three times per 1.04 ms, as at most two subthreads separate receive buffer servicing).
2.3 Software Thread Integration

After the controller interface thread is converted into subthreads it is integrated into three of the CAN functions which make up the CAN interface thread. As mentioned above, the CAN interface thread spends its time monitoring an idle bus or sending or receiving a message. The controller interface thread is integrated to meet its timing requirements in each of these three conditions.

![Figure 4. Periodic CPU activity during unrolled CAN bus sampling loop](image)

When monitoring the idle bus, the CPU has from 40 to 56 cycles of idle time per 33 µs iteration. The controller interface subthreads are too long (78 cycles) to fit into this gap, so the bus monitor loop is unrolled once to double the amount of idle time. In the midst of this idle time the bus must be sampled, so this code is integrated with the controller interface by replication into each of the subthreads at the appropriate time. The bus sampling code is replicated 5 times leading to code expansion of 45 bytes in addition to the 496 bytes of the duplicated controller interface, as shown in Table 1. This replication is less than the total number of subthreads (9) as several share the same padding code into which a copy of the bus sampling code is integrated. If loop unrolling is not possible, the subthreads would need to be further partitioned to fit into the available idle time.

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Size</th>
<th>Thread B</th>
<th>Size</th>
<th>Integrated Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>idle bus sampling</td>
<td>9</td>
<td>541</td>
</tr>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>send_bit</td>
<td>185</td>
<td>684</td>
</tr>
<tr>
<td>controller_interface</td>
<td>496</td>
<td>receive_bit</td>
<td>157</td>
<td>654</td>
</tr>
</tbody>
</table>

Table 1: Code Expansion (in bytes) from Integration with Controller Interface

Integration of the controller interface with send and receive messages functions is straightforward. Because each bit must be sent or sampled at a specific time, the message functions provide a scheduled framework into which the controller interface thread is integrated. The repeated calls to the send_bit() and receive_bit() make them logical candidates for integration. The idle time per bit varies while transmitting a message range from 91 to 121 cycles. The minimum idle time of 91 cycles is longer than the controller interface thread, so degenerate integration with send_bit() is sufficient; the entire thread is appended to the end of the function. The procedure is similar for receive_bit(). This duplication of code leads to significant code expansion.

<table>
<thead>
<tr>
<th>Function</th>
<th>Original Size</th>
<th>Integrated Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_bit</td>
<td>111</td>
<td>684</td>
</tr>
<tr>
<td>receive_bit</td>
<td>100</td>
<td>654</td>
</tr>
<tr>
<td>can_interface_executive</td>
<td>290</td>
<td>831</td>
</tr>
<tr>
<td>controller_interface</td>
<td>541</td>
<td>0</td>
</tr>
<tr>
<td>Other</td>
<td>2478</td>
<td>2478</td>
</tr>
<tr>
<td>Total</td>
<td>3520</td>
<td>4647</td>
</tr>
</tbody>
</table>

Table 2: Total Code Expansion (in bytes)
Integration raises the total code memory size from 3520 to 4647 bytes, or about 32%, as listed in Table 2. This expansion could be reduced through the use of more space-efficient padding techniques. The HT83C51 offers 8 kilobytes of on-chip memory, so this expansion is quite acceptable. In more typical high-volume applications, it is likely that the cost savings of eliminating a dedicated peripheral component will exceed the cost of the microcontroller (if added) and its memory.

### 2.4 Development Time

Software development for HSCAN took about 100 days, with about a quarter dedicated to developing the independent threads without considering real-time requirements. Conversion to real-time operation (through both simulated and actual execution on the target processor) combined with integrating the threads took over half of the total time. The software development effort could be simplified dramatically through the use of an automated tool for STI. More importantly, software maintenance could be made much easier with such a tool, as each thread could be operated upon in its source code format, buffering the user from assembly language concerns, improving portability and extending the useful life of software.

### 3 Conclusions

The prototype system was recently tested. It functioned successfully for an extended period at 200° C and met all program goals. The HSCAN prototype system performed flawlessly in passing messages between the CAN network and the UART without errors, demonstrating the feasibility of the HSM and STI techniques.

Whether driven by cost or function availability pressures, embedded system designers use many ad hoc techniques to implement and interleave real-time functions in software on microprocessors. Software thread integration provides a set of methods which simplify hardware to software migration and make it possible to implement in a tool. Currently under development at Carnegie Mellon University, this tool will enable designers to work at a higher level of abstraction without worrying about manual tweaking of assembly code. At this level designers become much more efficient and can implement (and maintain) more sophisticated embedded real-time systems.

### Acknowledgments

The authors thank the Lear Corporation and the former UT Automotive (now part of Lear) for the opportunity to apply STI/HSM research to their applications, and Knut Roll-Lund, who provided the CAN bitstream decoding function.

### References

- [Hon98b] High Temperature 32Kx8 Static RAM Datasheet, Honeywell, Inc., April 1998
- [Phi95] Databook IC20: 80C51-Based 8-Bit Microcontrollers, Philips Semiconductors, P8XC592 Data, January 1995, p. 1200