ABSTRACT

KUMAR, NAGENDRA J

STI Concepts for Bit – Bang Communication Protocols

(Under the direction of Dr. Alexander Dean)

In the modern times, embedded communication networks are being used in increased number of embedded systems to provide more reliability and cost effectiveness. Designers are forced to limit and minimize the size, weight, power consumption, costs and also the design time of their products. However, network controller chips are also expensive and hence moving functionality from hardware to software cuts down the costs and also makes custom fit protocols easier to implement.

Traditional methods of sharing a processor are not adequate for implementing communication protocol controllers in software because of the processing required during each bit. The available idle time is fine grain compared to the bit time and is usually small for even the fast context switching techniques (e.g. co-routines) to run any other thread. Without some scheme to recover this fine-grain idle time, no other work in the system would make any progress.

Software Thread Integration (STI) provides low cost concurrency on general-purpose microprocessors by interleaving multiple threads of control (having real-time constraints) into one. This thesis introduces new methods for implementing communication protocols in software using statically scheduled co-routines and software thread integration. With co-routines, switching from primary to secondary threads and vice versa can be done without incurring a penalty as severe as "context – switching". This technique will be been demonstrated on the SAE J1850 communication standard used in off- and on-road land-based vehicles. These methods also minimize the number of co-routine calls needed to share the processor thereby enabling finer-grain idle time to be recovered for use by the secondary thread. Increased number of compute cycles implies

- Improved performance of the secondary thread and
- Reduced minimum clock speed for the microprocessor.

Thus, now more secondary thread work can be done and also the minimum clock speed required of the processor is reduced. These factors enable the embedded system designers to use processors more efficiently and also with less development effort.

STI Concepts for Bit-Bang Communication Protocols

by

Nagendra J. Kumar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER ENGINEERING

Raleigh

May 2003

Approved by

_____	_____
Dr. Eric Rotenberg	**Dr. Thomas M. Conte**

_____

Dr. Alexander G. Dean, Chair of Advisory Committee

BIOGRAPHY

Nagendra J Kumar was born on July 4, 1976 in Bangalore, India. He graduated with a B.E degree in Electronics and Communication Engineering from University Visvesvaraya College of Engineering, Bangalore, India, in July 1997. After graduation, he worked first as a software design engineer at Tektronix Engineering Development India in Bangalore for 2 years and then as a Member Technical Staff – Developer in Sharp Software Development India for a year.

He then joined the masters program in computer engineering at North Carolina State University, Raleigh, NC. There he was a part of the STI for Bit Bang Communication protocols project and worked on his thesis under the direction of Dr Alexander Dean.

ACKNOWLEDGEMENTS

Graduate studies at North Carolina State University has been a truly gratifying experience and I would like to use this opportunity to thank one and all who made this possible.

First and foremost, I would like to thank my advisor, Dr Alexander G. Dean for giving me the opportunity to work on this project. Dr Dean envisaged the idea of software thread integration and has been the motivating factor behind my work. He provided valuable guidance and insights, resulting in this research. I got an opportunity to develop wonderful insights in the field of embedded systems under his guidance. Apart from being my guide, he has always encouraged independent thinking and helped me in all my endeavors. I thank him for that.

I thank the committee members, Dr Thomas Conte, Dr Greg Byrd and Dr Eric Rotenberg, for reviewing my thesis and providing valuable comments. I would also like to thank my other instructors at NCSU for their guidance in technical subjects.

I express my sincere thanks to all my friends and colleagues at NC State, who made my stay here at NCSU, a memorable one. My special thanks to my roommates, Raghavendra and Rajesh, and to my friends Mohit, Drubha and Saurabh, for their endless help and motivation.

Finally I thank my parents, Dr J. B. Subramaniam and Mrs Hemalatha, for their constant support and encouragement, to further my studies. They have played an active role in shaping my career and were always there when I needed them.

This material is based upon work supported by the National Science Foundation under Grant No. 0133690.

I conclude, by thanking one and all for making this happen.

TABLE OF CONTENTS

LIST OF FIGURES.....vi

1 Background : 1
1.1 Why Thread Integration 1
1.2 Typical Hardware Environment 1
1.3 Software Program Structure 2
1.4 BBCP Thread Code Structure 4
1.5 Timeline and Timing Constraints 5
1.6 Idle Time Allocation: 6
1.7 Fine Grain Scheduling: 7
1.8 Code Structure: 9
1.9 Contributions 11
1.10 Thesis Organization 11
2 Co-Call Technique For Integration 12
2.1 Goals for Code Transformation: 12
2.2 Co-calls and Co-routines: 12
2.2.1 Definition 12
2.2.2 Program States affected during switching 13
2.3 Primary thread preparation: 14
2.3.1 Definitions 14
2.3.2 Idle time analysis 15
2.3.3 Idle Time in Bit Level Functions 17
2.3.4 Message Level Functions 17
2.3.5 Transformations 18
2.4 Secondary thread 19
2.4.1 Structure 19
2.4.2 Do's and Don'ts 20
2.4.3 Preparation 21
2.4.4 Transformations 28
3 J1850 Protocol 39
3.1 Introduction 39
3.2 Signaling scheme 39
3.3 Arbitration: 40
3.4 Frame Format 41
4 Demonstration of Integration techniques on J1850 44
4.1 Overview 44
4.2 System Architecture 44
4.2.1 Hardware Architecture 44
4.2.2 Program Structure 46
4.2.3 Bus Interface Thread 49
4.3 Program Analysis 51
4.3.1 Primary Code – Idle Time 51
4.3.2 Secondary Thread – Determinism 53
4.4 Integration 53

4.4.1 Preparations 53
4.4.2 Transformations 55
4.4.3 Transformation Methodology 61
5 Results & Analysis 62
5.1 Simulation Environment: 62
5.2 Results: 63
5.2.1 Timing Evaluation: 63
5.2.2 Code Expansion: 65
5.2.3 Performance Improvement compared to Interrupt Based approach: 67
6 Summary & Future Work 70
6.1 Summary: 70
6.2 Future Work: 71

Bibliography……72
Appendix I……73
Appendix II……...88

List Of Figures

Figure 1-1 Typical communication system block diagram……1
Figure 1-2 Modified Block diagram……...2
Figure 1-3 Typical Implementation……....2
Figure 1-4 Host Interface Thread structure……3
Figure 1-5 BBCP thread structure……..3
Figure 1-6 Typical send bit operation timeline…...……..5
Figure 1-7 Idle Time distribution - case 1……6
Figure 1-8 Idle Time distribution - case 2……6
Figure 1-9 Idle Time distribution - case 3……7
Figure 1-10 CFG of primary and secondary threads…..……10
Figure 1-11 Context switching using interrupts…..……...10
Figure 1-12 Context switching using STI……..……11
Figure 2-1 Context switching through co-calls……13
Figure 2-2 Processor Idle Time……15
Figure 2-3 Cocall Insertion...……...16
Figure 2-4 Idle Time in bit-level functions…...……..17
Figure 2-5 Removing Intervening guest code……...……...18
Figure 2-6 Integration Process……..19
Figure 2-7 Primary and secondary thread switching……20
Figure 2-8 Subroutines in Secondary thread……21
Figure 2-9 CFG for code having blocking I/O calls……23
Figure 2-10 CDG for the modified blocking I/O loop code……...25
Figure 2-11 Predicate Padding in secondary……..26
Figure 2-12 Padded Secondary Thread……..27
Figure 2-13 CFG for Blocking I/O loop code……..……..29
Figure 2-14 Modified CDG……..……..30
Figure 2-15 Switching between Primary threads..……32
Figure 2-16 Synchronization..……33
Figure 2-17 Guarded Cocalls.……....34
Figure 2-18 PLBF Code.……35
Figure 2-19 Secondary Code..……35
Figure 2-20 "Raw" Secondary code..……..36
Figure 2-21 Secondary thread with Intervening code..……...37
Figure 2-22 Guarded Intervening Code..……....37
Figure 2-23 Register For Synchronization..……...38
Figure 3-1 J1850 bit symbol timings……40
Figure 3-2 J1850 Frame format……41
Figure 4-1 J1850 Node Setup……...44
Figure 4-2 J1850 Hardware Setup……45
Figure 4-3 Overall Simulation setup……46
Figure 4-4 Communication Interface……....46
Figure 4-5 TX. & RX. Message Queue Data Structure……47
Figure 4-6 IFR Queue Data Structure……...47
Figure 4-7 Host Interface Thread State Diagram……..48

Figure 4-8 BBCP thread code structure……49
Figure 4-9 Timeline for Send function…….50
Figure 4-10 Complete Timeline for send bit……..51
Figure 4-11 Intervening Guest Code Removal……..55
Figure 4-12 Cocall Based switching between Primary and Secondary……..…58
Figure 5-1 Test Setup..……..…62
Figure 5-2 Processor Timeline..……...67
Figure 5-3 Comparison between approaches for Send..……..69
Figure 5-4 Comparison between approached for receive..……..69

1 Background :

1.1 Why Thread Integration

Thread integration is useful in embedded system design when multiple threads of control flow are to be run on a single micro-controller. Design constraints in embedded system design are highly restrictive. Designers are forced to limit and minimize the size, weight, power consumption, costs and also the design time of their products. These design constraints could still be met by moving functionality from hardware to software (known as Hardware to Software Migration). HSM involves writing "Guest" functions to replace the hardware and use some means to ensure that the functions execute on time. Thread integration involves interleaving these multiple time critical "real – time" threads into a single thread.
In the modern times, embedded communication networks are being used in an increased number of embedded systems to provide more reliability and cost effectiveness. Since protocol controller chips are expensive, HSM for these applications helps in reducing system costs.

1.2 Typical Hardware Environment

In most of the current embedded communication systems, dedicated hardware is used to implement any protocol of choice. A typical setup is illustrated by the block diagram shown in figure 1.1.

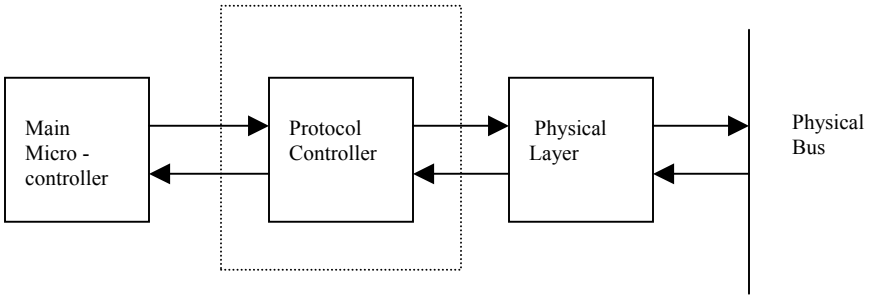


Figure 1-1 Typical communication system block diagram

In the above diagram, the main micro-controller issues commands instructing the protocol controller the commands to perform. The commands may be in the form of sending a message, receiving a message, reporting the status or any other customized command. The protocol controller is a piece of hardware acting as an interface between the main micro-controller and the physical bus. Its main functionality is to interpret the commands being issued by the micro-controller and complete the operations requested by initiating a sequence of actions according to the protocol being supported. The physical layer translates the bits being sent according to the specifications of the protocol.
Since the protocol controller is implemented in hardware, its design is highly customized to the protocol being implemented. In addition, these components tend to be mature designs. Hence this prevents the

designers from benefiting from the recent advances in the specific technology. Also most of the network controller chips available in the market today are expensive. For many applications, low production volumes of these chips lead to high prices due to lack of economies of scale. Hence moving these functions from dedicated hardware to software(HSM), running on a micro-controller, allows a variety of processing to be performed limited only by the speed of the micro-controller and hence the performance requirements of the protocol. But assuming the processor is fast enough to meet the demands of the protocol, there are direct benefits to HSM. The unit cost is lower, system is small and weighs less, reliability increases and new and proprietary functions could be easily added even after the design is mature. Now with the micro-controller replacing the dedicated hardware based protocol controller, the block diagram changes as follows:

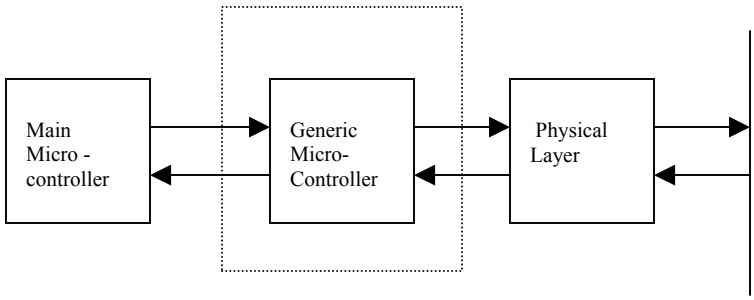


Figure 1-2 Modified Block diagram

In the above diagram, a generic micro-controller is chosen that is fast enough to meet all the deadlines of the protocol being implemented.

1.3 Software Program Structure

In order to implement a protocol controller in software, two threads need to be implemented. One thread (known as the "host") interacts with the main micro-controller and the other is responsible for putting bits on the bus according to the specific protocol being implemented. The communication between the two threads is by means of a message queue, one implemented in each direction. This is illustrated in figure 1.3.

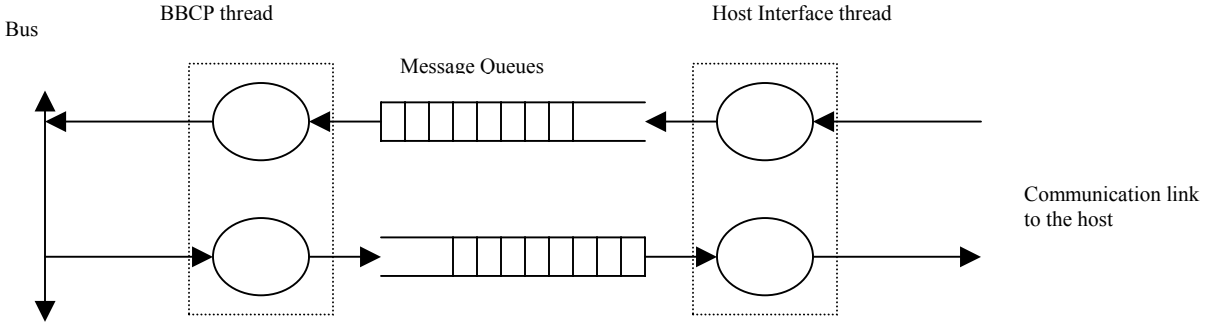


Figure 1-3 Typical Implementation

The thread that interfaces with the host, also called the host interface thread, is responsible for receiving commands from the host and if necessary communicating them to the other thread, also known as the BBCP (Bit Bang Communication Protocol) thread, by putting the command on to the message queue. The BBCP threads monitors the queue at periodic intervals and performs the functions requested by the host interface. It also receives any messages sent on the bus and puts it on the queue for the host interface thread to receive it.

A typical structure of the host interface thread is as shown in figure 1.4

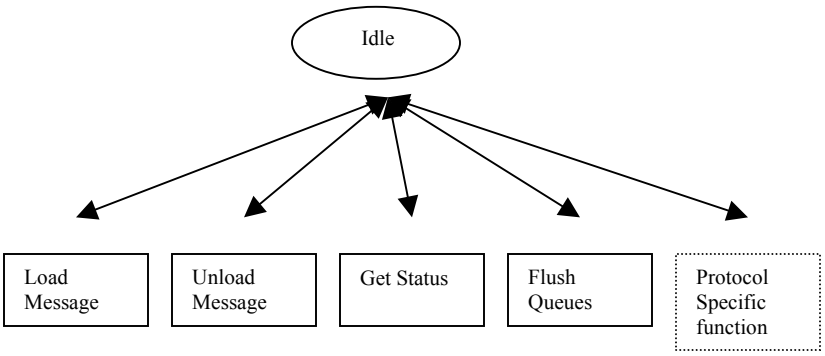


Figure 1-4 Host Interface Thread structure

A host interface thread is usually implemented as a state machine with the state transition being triggered by the commands sent by the host. The thread starts out in the idle state. Based on the command sent by the host, it switches to one of the states shown in the figure. For example, when the host sends a command instructing the thread to send a message, the thread first reads the message to be sent and then communicates the message on to the BBCP thread by putting it on to the queue. The states shown in the figure are not necessarily present for all protocols. Based on the protocol being implemented, the states may increase or decrease.

Similarly, the structure of a typical BBCP thread is as shown in figure 1.5

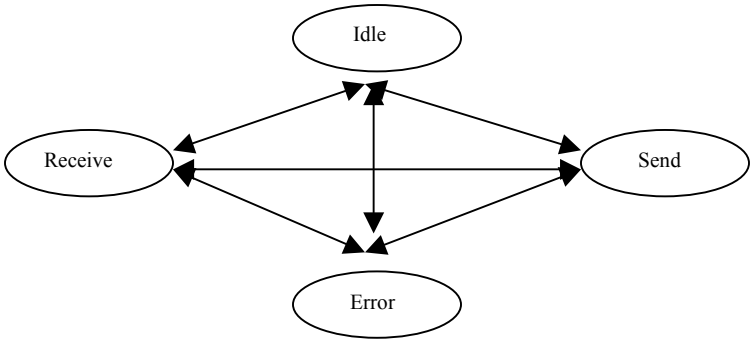


Figure 1-5 BBCP thread structure

The BBCP thread is also implemented as a state machine. The number of states present depends on the specific protocol being implemented. But in most of the protocol implementations, the basic states present are as shown in the figure 1.5. The transition between states is triggered by a flag that is set either

- By a function that polls the message queue for any messages to be sent or
- By a function that is called when a bit is to be received or
- By a function that is called to implement some protocol specific feature.

1.4 BBCP Thread Code Structure

The basic functionality of the BBCP thread is implemented as a state diagram shown in figure 1.5. To implement this state machine, the code is structured in form of layers. These layers interact with each other in order to transmit or receive a bit. This interaction can be depicted by means of a timeline as shown in the figure 1.6 for transmitting a message.

Executive or Manager Function: This is the top-level function that runs a finite state machine to monitor an idle bus, send a message or receive a message. As shown in the figure, when a request to transmit is received, the manager layer uses a subroutine call to pass on the request to its lower layer viz. the messaging layer.

Message Level Function: This is the middle layer that is called by the executive or manager function. This contains all the message oriented functions e.g. send_message, receive_message. These functions are responsible for encoding/decoding the message as per the protocol and then pass them to the corresponding layers. As shown in the figure, when sending a message, the send_message function forms the frame to be sent as per the protocol specifications and sends the frame to the lower layer bit by bit. Similarly, when receiving a message, the receive_message function puts the decoded message into the message queue to be picked up by the host interface thread. In addition to these, depending on the protocol, the messaging layer may also be responsible for calculating the CRC (when sufficient time to do that does not exist in bit level layer) and checking whether the received CRC is same as the calculated CRC.

Bit Level Function: This is the bottom layer that is responsible for putting bits on the bus. This contains functions like send_bit and receive_bit. These functions are called by the message level functions when a message is to be sent or received. The bit level functions may do additional tasks like bit stuffing, calculating and appending CRC / Parity depending on the protocol being implemented, multiple sampling of the bus and voting on the received samples (while receiving a bit) and checking whether the bus matches the bit being transmitted (while transmitting a bit). The timeline in figure1.6 shows some of the actions performed by a send bit function.

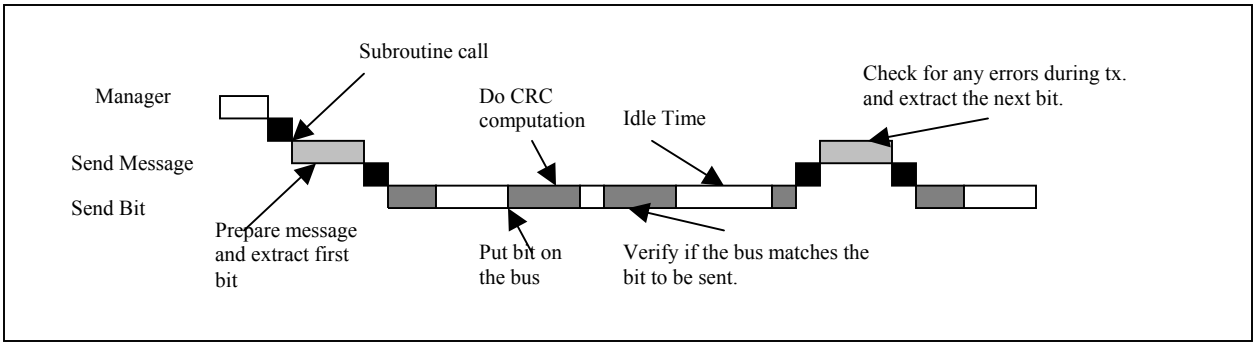


Figure 1-6 Typical send bit operation timeline

1.5 Timeline and Timing Constraints

All protocols specify the rate at which the bits are to be sent or received on the bus. Hence when the protocol controllers are written in software, care has to be taken to ensure that the message or the bit level functions do not put/receive a bit from the bus at a rate different from the specified rate. This introduces tight deadlines/tolerances for the BBCP thread. Since the BBCP thread has rigorous real-time constraints, it is also known as the "primary" thread. The tolerance specified is usually ¼ of the bit time or less. As a result of this timing deadline, idle time is introduced into the primary thread. This can also be seen in the timeline shown in figure1.6. When transmitting a bit, the send bit function has to perform activities like putting the bit on the bus and then checking whether the bus matches the bit being sent. The bus can be checked only after a time corresponding to twice the time taken by the bit to propagate along the length of the bus. Hence these are activities that need to be performed at two separate and fixed instants of time. This timing constraint introduces an idle time as seen in the figure. Idle time can also be introduced by several other factors like forcing the bit level functions to last for a fixed duration of time in order to remove any variability in the bit rate of the protocol. The idle time present may be highly fragmented and hence available in only small pieces. This is due to the fact that the message level/bit level functions may have to perform activities at different instants and thus fragmenting the idle time.

The host interface thread, on the other hand, has looser real-time constraints than the BBCP thread. The only real-time constraints it may have would be to poll the communication link to the host at periodic intervals. For example, if the communication link happens to be an UART, then the host interface threads polls the UART for any incoming message activity. Now when a byte is received, it must be unloaded before the next one comes in to prevent the UART receive buffer from being overrun. With a 9600 baud link, the host interface thread must be called enough to service the UART at least every1.04 ms. This implies that the idle time present here is in much larger chunks than that present in the primary thread and hence is worth reclaiming.

1.6 Idle Time Allocation:

While meeting the real-time constraints of the protocol being implemented, idle time is introduced in the primary thread. The constraint that a real-time primary thread has to satisfy for sending / receiving a bit is:

$$T_{bit\_after} + T_{message} + T_{bit\_before} = T_{bit} \qquad \text{(Equation 1)}$$

Where

T_{bit_after} = Time between a bit is put/received from the bus and the bit-level function returns.

$T_{message}$ = Time taken by the message layer to execute between bit layer function calls.

T_{bit_before} = Time between the bit-level function starts executing and a bit is put/received from the bus.

T_{bit} = Bit time (1/Bit rate of the protocol).

In order for the above equation to be satisfied, idle time could be introduced at a number of places. The figure1.7 shows one such distribution of the idle time.

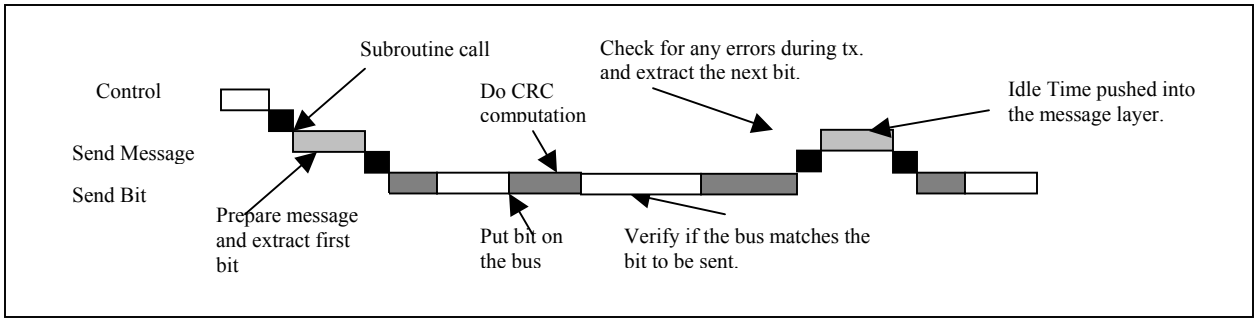


Figure 1-7 Idle Time distribution - case 1

Here the idle time is present only in the bit-level functions. This is the case when the idle time gets added to Tbit_after. Now equation1 1 changes as follows: indent

$$T_{bit\_after} + Tidle + T_{message} + T_{bit\_before} = T_{bit} \qquad \text{(Equation 2)}$$

Where

Tidle = Idle time inserted into the thread to meet the equation 2

If the bit-level functions are made to return immediately after sending/receiving a bit, then the idle time is pushed up into the message layer functions. Now Tidle is introduced as follows:

$$T_{bit\_after} + T_{message} + Tidle + T_{bit\_before} = T_{bit} \qquad \text{(Equation 3)}$$

This is shown in figure 1.8.

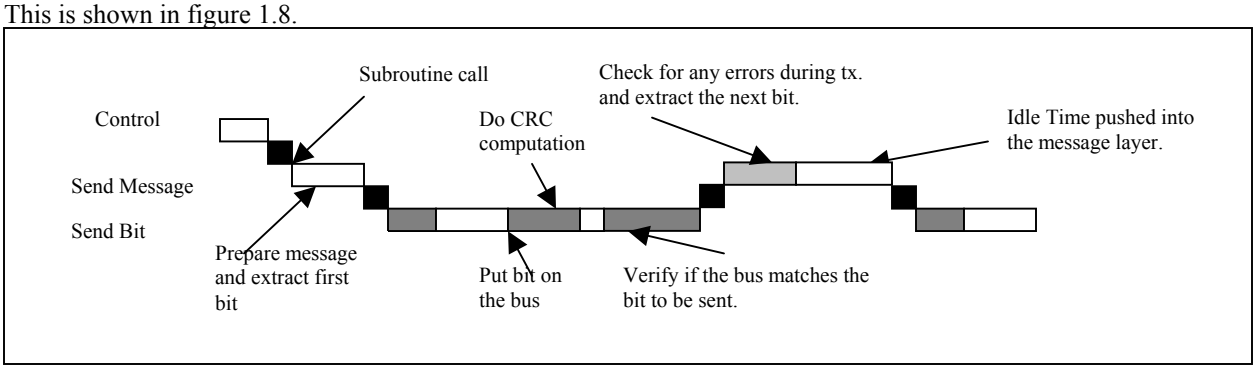


Figure 1-8 Idle Time distribution - case 2

Idle time could also be introduced as shown:

$$T_{bit\_after} + Tidle + T_{message} + Tidle + T_{bit\_before} = T_{bit}$$

Here the idle time is divided between the bit-level and message-level functions. This again can be seen in the figure 1.9.

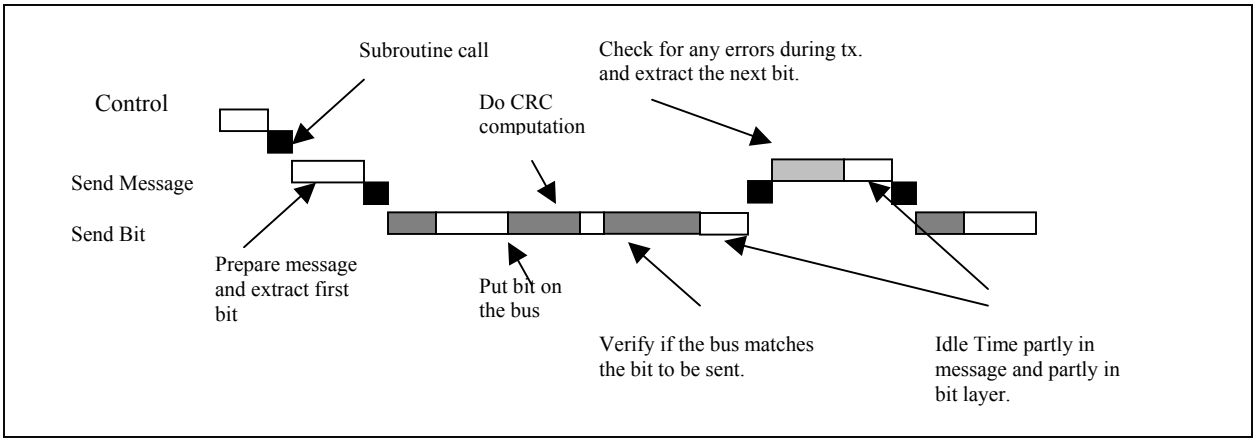


Figure 1-9 Idle Time distribution - case 3

In order to reclaim the idle time most efficiently, the first two options are more desirable. The third option scatters the idle time and hence retrieving it becomes highly cumbersome and non-elegant. Also for the sake of simplicity, the first option is more desirable. Hence, in order to simplify the code structure, the idle time is confined to the bit-level functions. Thus the bit-level functions are padded in order to meet equation 1.

1.7 Fine Grain Scheduling:

Since the primary thread ends up with idle time in the bit level functions due to the real-time constraints, the host interface thread could be scheduled to execute during this idle time. Since the idle time is highly fragmented and hence is fine grain, the scheduling scheme chosen has to ensure that the instructions are executed at the right times even after integration. This scheduling scheme is responsible for switching control from the primary thread to the host interface thread when an idle time slot is encountered. During the idle time slot, the host interface thread executes before switching control back to the primary at the end of the idle slot. Thus the switching overhead introduced by the scheduling mechanism should be kept to a minimum in order to maximize the execution length of the host interface thread segment and to ensure that the real time instructions in the primary thread are executed at the correct time (within the prescribed tolerances).

There are several techniques currently being used to schedule the real time threads, once they are moved from hardware to software. Some of these techniques are listed:

Interrupts: This is the most common method used to schedule real time events. Interrupts are scheduled to happen when the guest threads are supposed to run. One common application where HSM is currently being used is in implementing protocol controllers and I/O interfaces in software rather than in hardware. This provides serious advantages in the form of cost effectiveness and the robustness of software while suffers from the drawback that the baud rates supported are limited. In these applications, the real – time requirement is when a bit is to be sent or received. Hence to ensure this, interrupts have to be issued at periodic intervals corresponding to the bit rate of the protocol being supported. Setting a timer that generates interrupts at each of its count could do this. During the interrupt service routine of the timer interrupt, instructions are executed that put/sample bits from the bus. Generally the processor runs at a frequency much greater than the bit rate. Sometimes, in order to support full – duplex operation, the bit time is divided into multiple (say N) sub-bit times. Now the timer is made to run at N times the bit rate for the supported baud rate. In all these cases, the maximum baud rate that can be supported is essentially determined by the amount of time it might take the micro-controller to complete all operations associated with transmitting and receiving a bit.

Using interrupts however, limits the bit rates than can be supported. This is because the number of cycles available for the processor to execute the send/receive functionality between two timer interrupts is now reduced. Also interrupts incur the penalty of having some context switching time at the time of entering and leaving the interrupt service routine. This time is fixed irrespective of the frequency at which the processor is operating. However, if the processor frequency is reduced or the bit rate of the protocol being executed is increased, the number of cycles available to do useful work is again reduced.

Busy Waiting : When the guest does not require all of the processor's throughput, there is plenty of idle time scattered throughout. When the chunks of idle time slices are large, interrupts could be used to schedule other events (to execute other work). But, on the other hand, when the chunks of idle time become finer, there is not sufficient time to do context switching. Now Busy waiting or padding can be used. Busy waiting or padding can also be used to schedule events whose occurrence is predictable and some amount of latency can be tolerated in its detection. One application where busy waiting is used is in implementing a bus interface. Here all the events occur at a specific time and hence padding could be used to schedule their occurrence. For example, for a particular bit rate being supported, the bus interface knows when to send a bit or receive a bit. Hence, after sending/receiving a bit, a NOP instruction loop could be used to wait before the next bit is sent/received. Busy waiting can be very tedious to implement due to the multiple iterations of the count cycles/pad/verify/adjust process required. Also since NOP's are being to fill up the time between two events, the processors efficiency is being wasted. NOP's also increase the code size, though using delay loops make this code expansion negligible.

Using software polling to detect the trigger event is also less expensive than using interrupts in terms of the resources used and the processing time taken. But on the other hand, there are cases where using interrupts makes a certain design possible whereas using polling could cause unacceptable delays. For example,

taking actions on human inputs could be done using polling because humans would not notice a 5-millisecond delay, but the same would prove dangerous when trying to control an antilock braking system

1.8 Code Structure:

The different multi-tasking schemes currently being used to invoke the real-time thread can be categorized into two broad categories – Preemptive and cooperative.

Preemptive: The preemptive schemes for scheduling the threads cause the thread being executed to stall in order to facilitate the execution of the real-time thread. Using interrupts is one way of doing this. In inherently multithreaded approaches, an operating system could be used to schedule different threads. An OS, in turn, uses interrupts to preempt the executing threads and schedule different ones. Interrupts can be scheduled to occur either by means of a timer or activity on the bus. If a timer is used, then it is made to generate interrupts at periodic intervals corresponding to the bit rate of the protocol. On the other hand, a certain activity on the bus can also be used to generate the interrupt. This is commonly used when receiving a bit. In any of these cases, when an interrupt occurs, the interrupt service routine saves the information required to resume the stalled thread and then calls the next thread.

Cooperative: In this scheme for scheduling, the threads are structured as segments with each segment executing before relinquishing control to a segment of the other thread. The division of threads into segments can be achieved by means of a finite state machine that advances with each call. Hence each state corresponds to the execution of one segment. The duration of each state depends on whether the scheduling is static or dynamic.

In static scheduling, each segment is structured to execute for a fixed time. Hence during each state of the FSM, a segment of fixed duration executes before relinquishing control. On the other hand, in dynamic scheduling, the duration of each segment is not fixed. There are a number of checkpoints inserted in each segment that determines whether the segment should relinquish control or not. Hence during each state, a segment is executed that has a number of "yield" points where a check is performed on a condition. For example the condition may be to see if a bit is available on the bus. Only if the condition is true, the segment executes a return and facilitates the execution of the next state. Cooperative scheduling could be implemented either by subroutines or coroutines.

In bit bang communication protocols, the primary thread (i.e. the send and receive bit functions) has fine grain idle time that could be utilized. These idle time slots could be utilized by calling the host interface thread. Calling the host interface or the secondary thread could be done by either of the scheduling schemes described above. In each of the schemes, the transition between the primary and secondary threads requires a context switch that consumes valuable processor time. In order to execute the secondary thread in the idle time slots of the primary, the secondary thread needs to be divided into segments. The communication between the primary and secondary can be seen in the figure 1-10.

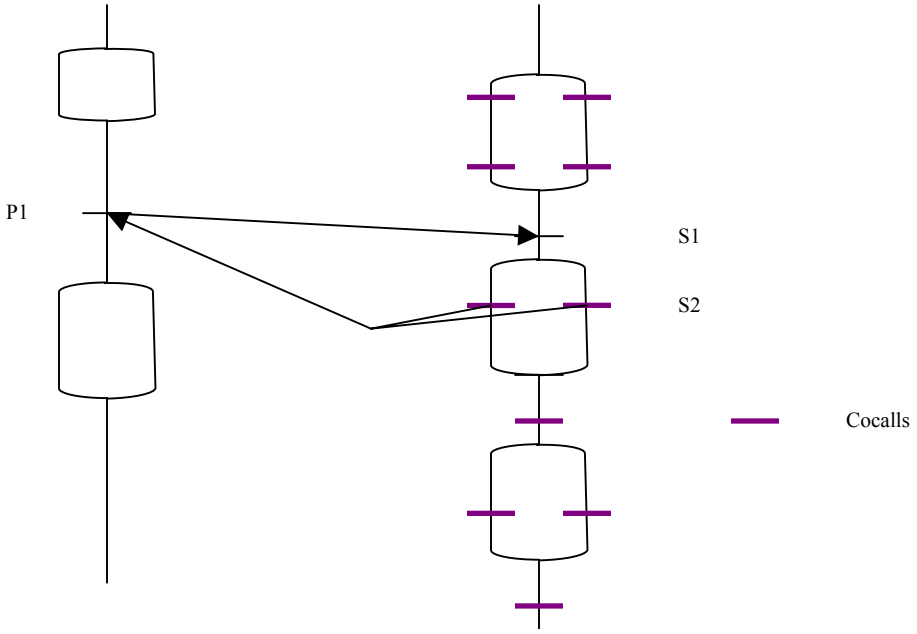


Figure 1-10 CFG of primary and secondary threads

Figure 1-10. shows the control flow graphs of the primary and secondary threads. The control flow graph for the secondary thread shows a number of entry/yield points. These are the points where the secondary thread starts/yields control back to the primary thread. As seen from the figure, when the primary thread finishes executing until point P1, it encounters an idle time slot. Now control is passed to the nearest entry point in the secondary thread (point S1 in the figure). When the secondary now finishes executing until point S2, it yields control back to the primary thread.

The placement of the yield points depends on the time taken for each context switch and the idle time duration. If the context switch takes a large number of cycles, then sufficient time may not exist for the secondary thread to proceed, especially for fine grain idle time. If the idle time slot is short enough, then the entire slot may be used up for just switching from the primary to the secondary and then back to the primary.

The timeline when an interrupt is used to perform a context switch to the secondary thread is shown in figure 1-11.

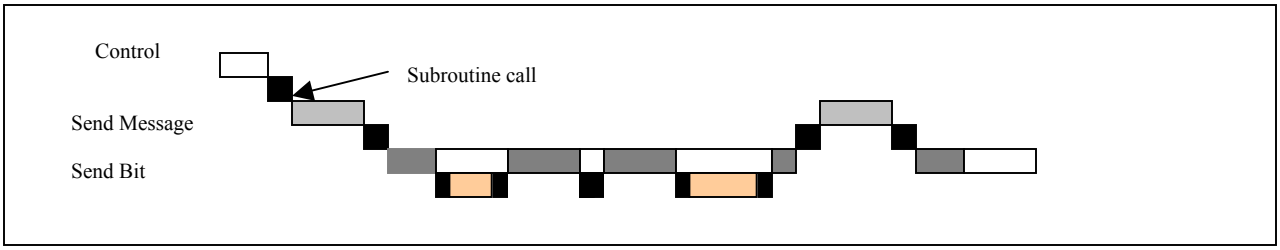


Figure 1-11 Context switching using interrupts

As seen from the timeline, for fine grain idle time, most of the idle time is consumed by the context switch instructions in the interrupt service routine.
Using STI, the number of context switches can be reduced by grouping the primary instructions into multiple locations in the secondary thread. Now the timeline with this approach can be seen in figure 1-12.

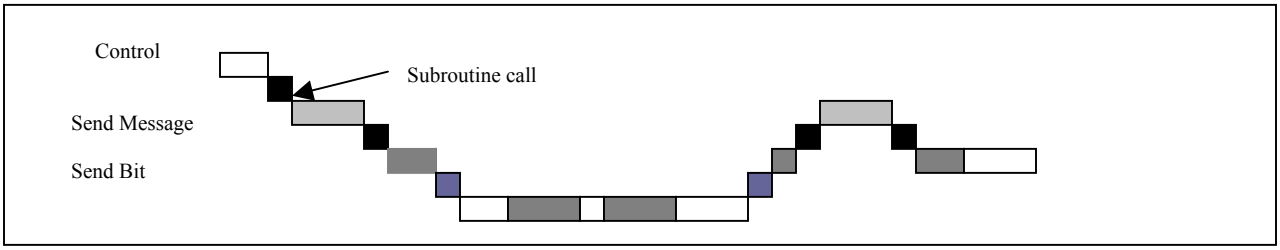


Figure 1-12 Context switching using STI

1.9 Contributions

This thesis introduces new methods for implementing communication protocols in software using statically scheduled co-routines and software thread integration. With co-routines, switching from primary to secondary threads and vice versa can be done without incurring a penalty as severe as "context – switching". This technique will be been demonstrated on the SAE J1850 communication standard used in off- and on-road land-based vehicles. These methods also minimize the number of co-routine calls needed to share the processor thereby enabling finer-grain idle time to be recovered for use by the secondary thread. An increased number of compute cycles implies

- Improved performance of the secondary thread and
- Reduced minimum clock speed for the microprocessor.

Thus, now more secondary thread work can be done and also the minimum clock speed required of the processor is reduced. These factors enable the embedded system designers to use processors more efficiently and also with less development effort.

1.10 Thesis Organization

Chapter 2 gives an overview of current protocol controller implementations in hardware and software and as to how co-calls could be used for integration. Chapter 3 gives a brief introduction of the J1850 protocol. The proposed theory and techniques for integration are discusses in chapter 4. Chapter 5 discusses how the proposed technique for integration could be used for the J1850 protocol. Chapter 6 presents the experimental results. Chapter 7 summarizes the thesis and proposes future work

2 Co-Call Technique For Integration

2.1 Goals for Code Transformation:

Code transformations are required for both the primary and secondary threads before any integration can be done. For the primary threads, most of the transformation is in the bit level functions and some in the message level functions.

The bit level functions are modified to ensure that each function takes a constant amount of time. Thus two versions of the primary thread are created, one with dedicated padding and the other with coroutine calls. The version with coroutine calls is used when a secondary thread needs to be called during the idle time slots. This version also may contain padding to remove timing jitter introduced by conditional or looping instructions. The secondary thread's functions also have coroutine calls embedded within them to ensure that control is returned to the primary at the correct instant. Primary threads having dedicated padding are used when there is no suitable secondary thread to be called during the idle time slots.

The message level functions are modified with padding or code motion to ensure that the bit level function is called at periodic intervals (the duration of the bit on the bus). The timing variation between the calls is removed this way as well.

The secondary thread functions also undergo some amount of code transformation to make integration easier. Just as with primary thread functions, the secondary thread functions are padded to remove any timing jitter. Once the timing information is known accurately, co-calls are inserted at periodic intervals within the secondary thread to yield control back to the primary.

2.2 Co-calls and Co-routines:

2.2.1 Definition

A co-call operation is used to transfer control between two processes. A co-call is effectively a call and return instruction combined into one operation. From the point of view of the process executing the co-call, the operation is equivalent to a procedure call and from the point of view of the process being called, the co-call operation is equivalent to a return operation. Thus, unlike subroutines, when the second process co-calls the first, control resumes not at the beginning of the first process, but immediately after the co-call operation. If the two processes execute a sequence of mutual co-calls, control will transfer between the two processes in the following fashion:

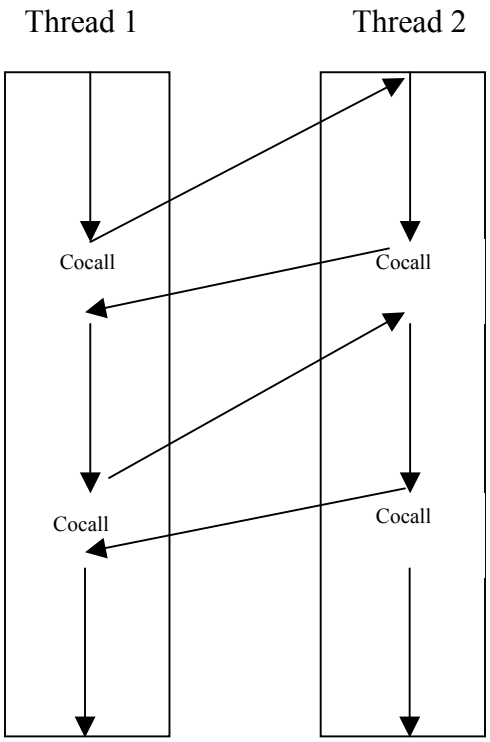


Figure 2-1 Context switching through co-calls

As seen from fig 2.1, co-calls decouple the progress of the two threads. Due to the nature of thread integration, co-calls are more suitable for thread integration than subroutines.

2.2.2 Program States affected during switching

Each thread in a program has some data associated with it. When using a context switch to jump from one thread to another, this data needs to be saved to be restored later. In most of the cases, the data associated with a thread typically includes:

Current Program Counter (PC) value

State information stored in Status register (SREG)

Stack Pointer value

Register values

Global values

In order to do a context switch, not all the data associated with a thread needs to be saved. If data locations of the two threads overlap, then the corresponding data needs to be saved.

When using co-calls for context switch, the threads need to save enough information to start executing from the location the threads had left off before. The thread being executed does not run to completion, but executes for a fixed amount of time before relinquishing control to the other. To allow each thread to keep its call stack intact, multiple stacks are implemented, one for each thread. For simplicity, the registers can be partitioned, with each thread being allotted a fixed number of registers. With this approach, the only data that needs to be saved include:
Program Counter – to enable jumping back and forth between threads
Stack Pointer – To enable each thread having its own stack
Status Register – To keep state information intact for each thread.
Thus a typical co-call implementation for AVR 8-bit micro-controllers would be:

```
in __T1SREG__, __SREG__
in __T1SPL__, __SPL__
in __T1SPH__, __SPH__
ldi __T1PCL__,lo8(PC)
ldi __T1PCH__,hi8(PC)

out __SREG__, __T2SREG__
out __SPL__, __T2SPL__
out __SPH__, __T2SPH__
mov  r29, __T2PCL__
mov r30, __T2PCH__
ijmp
```

The above code represents a co-call implementation for switching between two threads T1 and T2. Registers having __T1 prefix correspond to thread T1 and with __T2 correspond to T2. Also as seen from the code, fixed registers are allotted statically to the threads T1 and T2 so that the threads register data do not overlap. For the above implementation, the context switch takes 20 cycles. Thus for integration, the primary thread must have a idle time period greater than 2*20 = 40 cycles.

2.3 Primary thread preparation:

2.3.1 Definitions

- T_{Bit} is defined to be the duration of a bit on the bus. This value depends on the bit rate specified by a protocol and may be different for different protocols. For some protocols like J1850, the bit duration on the bus is variable depending on the bit to be sent and is a multiple of a minimum duration. Here we use the minimum bit duration as T_{Bit}.

- T_{CS} is defined as the time taken for a context switch as implemented with a co-routine call.

2.3.2 Idle time analysis

Idle time is introduced in the primary threads due to the real-time constraints imposed on them. For example, in bit bang communication protocols, for a given bit-rate and clock frequency, instructions that sample or control the bus impose explicit timing requirements based on the protocol. These instructions serve as timing anchors. Instructions have the implicit timing requirement of executing in order. These explicit and implicit timing constraints introduce idle time into the primary thread.
The idle time introduced in the primary thread is often highly fragmented as shown in the processor timeline given in figure 2.2.

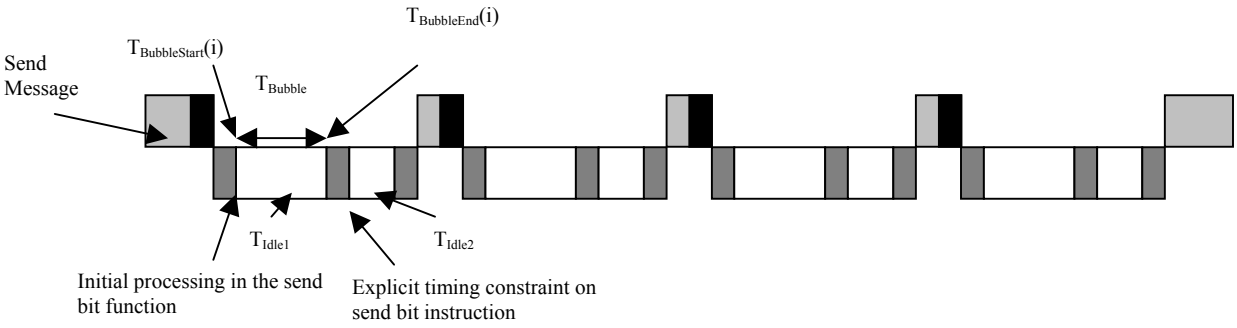


Figure 2-2 Processor Idle Time

The shaded blocks in the figure indicate the period of activity. The non-shaded regions indicate the idle time slots. An idle time bubble is characterized by three parameters:
$T_{BubbleStart}(i)$ = Start of the idle time bubble "i".
$T_{BubbleEnd}(i)$ = End of the idle time bubble "i".
$T_{Bubble}(i)$ = Duration of the idle time bubble "i".

$$= T_{BubbleEnd}(i) - T_{BubbleStart}(i)$$

In order to be able to integrate secondary thread with the primary (during the idle time slots of the primary), the idle time fragments have to be of some minimum length. This "minimum length" depends on the time taken to execute a co-call to transfer control to a secondary thread. Not all fragments need to satisfy this minimum length criterion. If the primary thread has several idle time fragments then at least two of the fragments have to be longer than the minimum length and the other fragments can be of any arbitrary length. In the first fragment satisfying the minimum length, a co-call is executed transferring control to the secondary thread and in the last such fragment, a co-call transferring control back to the primary is executed. On the other hand, if the primary thread has just one idle time chunk, then it should be long enough to execute two co-calls. As seen in figure 2.3, the PLBF has three idle time chunks. The first idle time chunk T_{Idle1} is not long enough to execute a co-call. The second and third idle time chunks, T_{Idle2} and

T_{Idle3}, are however long enough for a co-call. Hence co-calls, transferring control back and forth between the primary and secondary threads, are inserted in these time slots as shown in figure 2.3. For example, for the co-call code shown before, the idle time fragments have to be 20 cycles long for the case

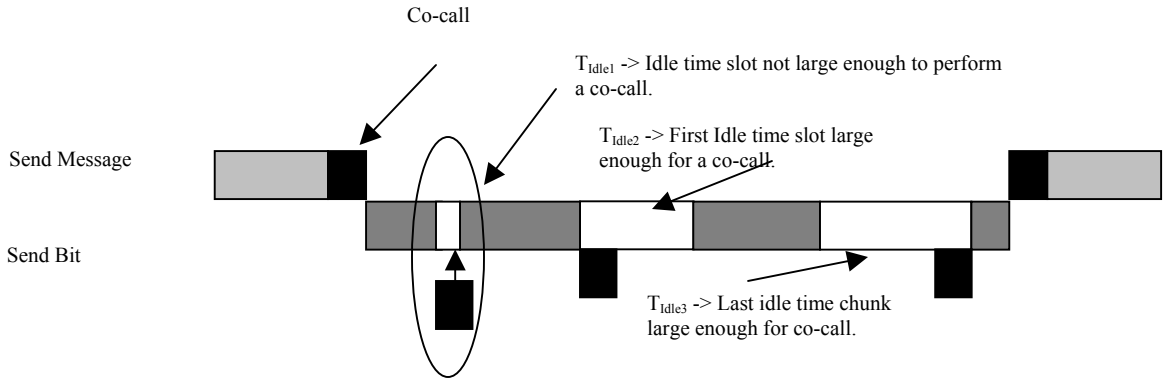


Figure 2-3 Cocall Insertion

when the primary thread has several idle time chunks and 40 cycles when there is just one idle time slot in the primary.

The fragmented idle time can be best described mathematically by the "Segment time" or $T_{Segment}$. The segment time is defined as the difference between the end of the last idle time slot long enough to do a co-call (denoted by b) and the start of the first idle slot long enough for a context switch (denoted by a). The segment idle time is the sum of idle time durations, from a to b, within a bit level function minus two times the overhead incurred to execute a context switch (co-call). This time is useful when calculating the actual time available for executing a secondary thread segment. In other words, if

$a = \min(i)\ T_{Bubble}(i) > T_{CS}$ (First Piece of idle time large enough for a co-call)

and

$b = \max(i)\ T_{Bubble}(i) > T_{CS}$ (Last piece of idle time long enough for a co-call)

then

$$T_{Segment} = T_{BubbleEnd}(b) - T_{BubbleStart}(a)$$

Within $T_{Segment,}$ $T_{SegmentIdle}$ is the amount of idle time that is actually available for useful secondary thread work. It can defined as

$$T_{SegmentIdle} = \left(\sum_{i=a}^{b} T_{Bubble}(i) \right) - 2 * T_{CS}$$

2.3.3 Idle Time in Bit Level Functions

The idle time is distributed among the layers comprising the primary thread viz. the messaging and the bit layer functions. For simplicity, the functions are modified so as to place as much idle time as possible in the bit level functions. This is accomplished by padding the bit level functions so as to execute a "return" as late as possible. This is as shown in the figure below.

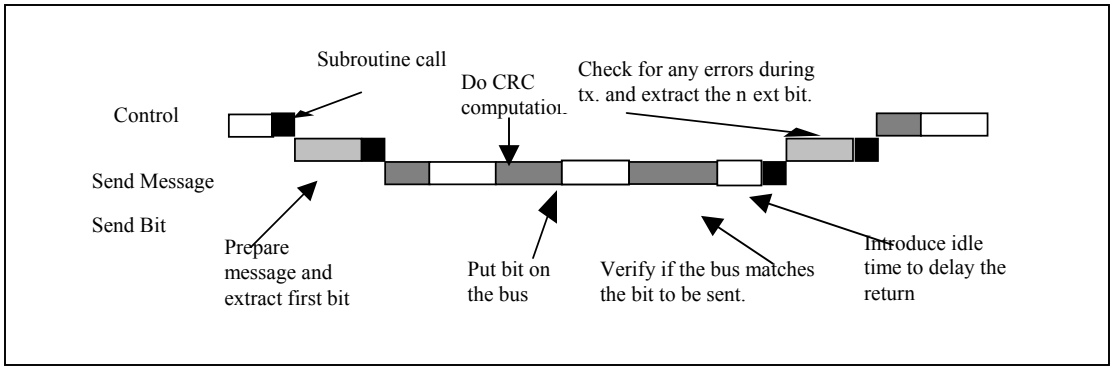


Figure 2-4 Idle Time in bit-level functions

Due to the idle time introduced, the bit level functions are now made to execute for a worst-case duration. Ideally the bit level functions are modified so that they take a constant amount of time to execute. However, this condition is not always satisfied. For example in J1850 protocol, variable pulse width modulation is used to send each bit. Hence there are two bit durations, 64 us and 128 us, used to encode "1" and "0". The bit level function consists of conditional code that checks the duration and level of the previous bit sent and correspondingly transmits the current bit according to VPW encoding. Thus based on the path taken while sending a bit, the function can have varying execution times. Correspondingly the idle time introduced in the bit level functions also varies with the encoding used to transmit the bit.

2.3.4 Message Level Functions

Message level functions need to be modified to ensure that the calls to the bit level functions occur at multiples of the bit time of the protocol i.e. multiple of T_{Bit}. The transformation comprises either padding or code motion. This removes any timing jitter between the calls to the send bit functions.
Padding can be used to equalize the times to the worst case time between calls to the bit level functions. Alternatively, code motion can be used to speculatively execute certain instructions early, thereby spreading out work between bit-level function calls to minimize the worst case. Code motion is desirable when worst case padding causes the time between two calls to bit level functions to exceed T_{Bit}.
The receive message functions also need to be modified with padding at the start of the function to ensure that the receive bit function starts sampling the bus at the correct instant.

2.3.5 Transformations

The bit level functions need to undergo some code transformations to allow integration of the secondary thread code. The transformations can be best described by diagrams shown in figures 2.5 and 2.6.

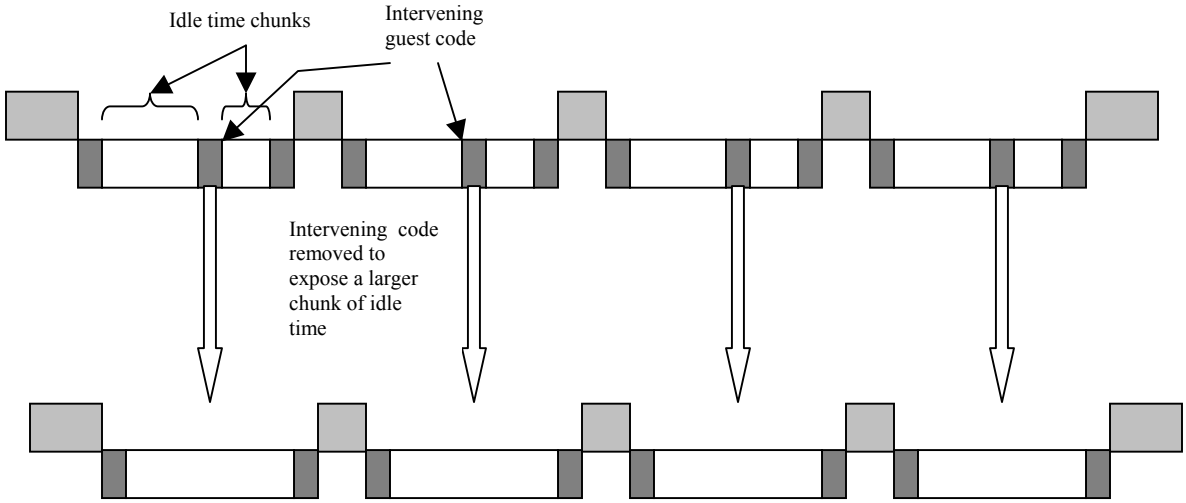


Figure 2-5 Removing Intervening guest code

As explained before, the idle time in the bit level functions is highly fragmented and the idle time chunks are too small to be utilized for any other work. Hence to recover more idle time, the functions are modified by removing the intervening guest code and saving it for later replication into the secondary thread. This is depicted in figure 2.5.

Next co-calls are inserted during the idle time slots to yield control to the secondary. The secondary thread is also modified by inserting co-calls to return control back to the primary at the end of the idle time slot. The entire process is depicted in figure 2.6.

In most communication protocols, there are multiple bit level functions (BLF) that make up the primary thread. These functions may have the same or different idle time periods. If the idle time periods are the same, then the same secondary thread can be integrated with all the PLBFs. In the typical case, however, the idle time slots of the PLBFs are of different durations. Hence separate versions of the secondary thread are needed. Each version is integrated with a PLBF. For example, most of the communication protocols have at-least two PLBFs – send bit and receive bit. Hence in this case two versions of secondary thread are created. One will be integrated with the send-bit function and the other with the receive-bit function. In each case the transformations are the same; differences result from varying segment time durations and the amount of primary thread work.

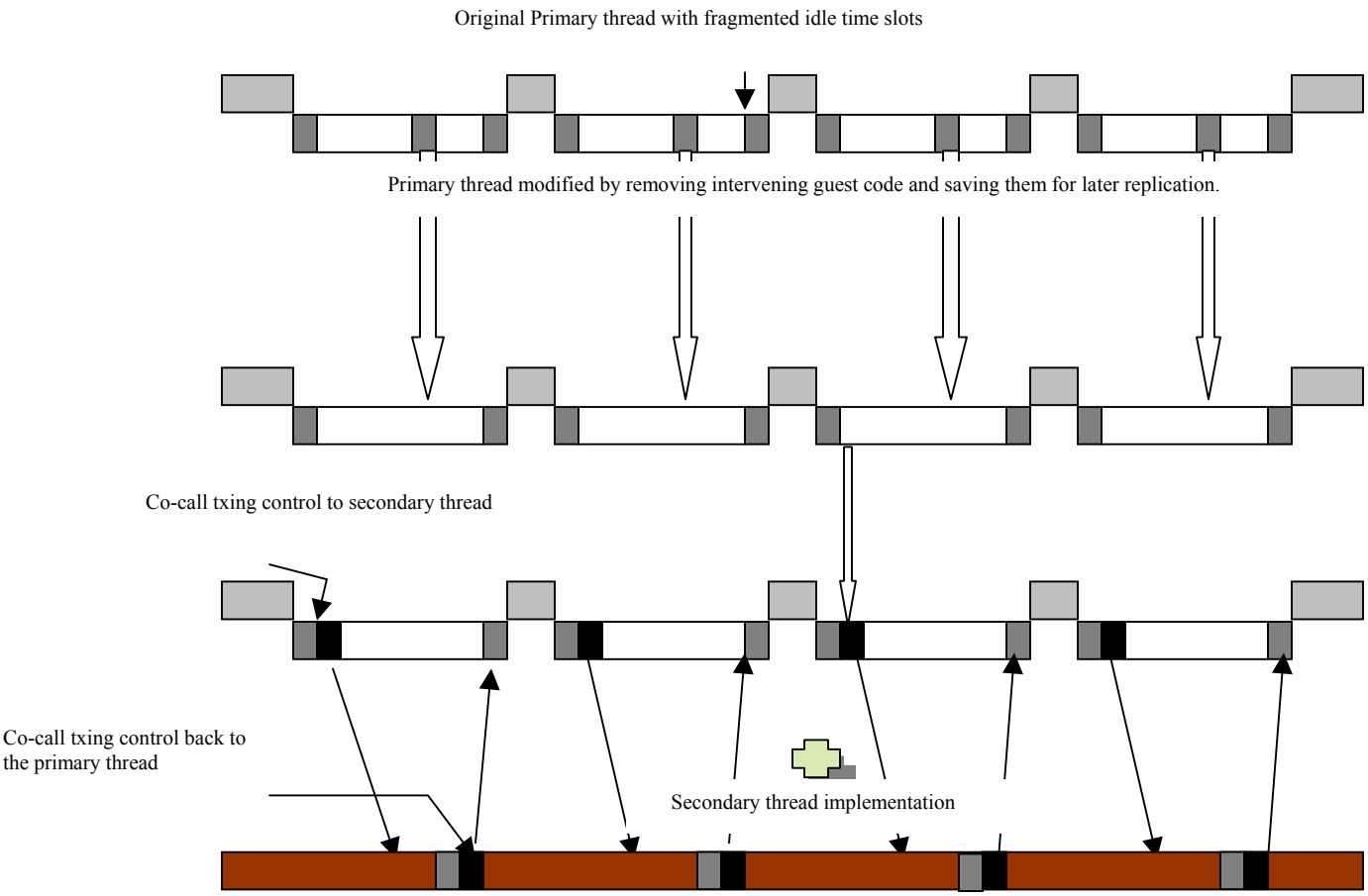


Figure 2-6 Integration Process

2.4 Secondary thread

2.4.1 Structure

The secondary thread has to be structured appropriately to be able to run during the idle time slots of the primary thread. In communication protocols, the secondary thread acts as an interface between the main host controller and the Bus interface. Hence it is also known as the “host interface” thread. This thread constantly monitors the interface to the host controller to check for commands. Thus, the secondary thread is implemented as an infinite loop. The commands sent by the host controller may be to send a message, receive a message or to report the status. Depending on the protocol being implemented, these commands may vary.

Since the secondary thread constantly monitors the host controller interface, real time requirements exist in the form of servicing the communication link to the host controller at regular intervals. For example when an UART is used as the communication link and a byte is received on the link, the host controller thread has to unload the byte before the next arrives to eliminate the possibility of receive buffer overrun. With a

9600-baud link, the controller interface thread must be called enough to service the UART at least every 1.04ms.

In order to meet the real-time requirements and also to run the secondary thread in the idle time slots of the primary, the secondary thread is divided into segments. Each segment is bounded by co-calls that transfer control back and forth between the primary and secondary threads. The interaction between the primary and secondary threads can be shown by the timeline in figure2.7.

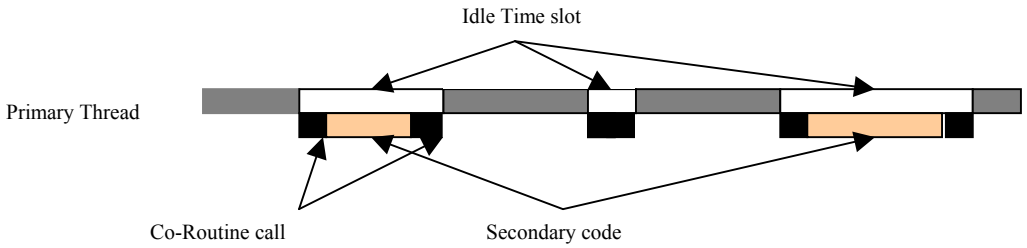


Figure 2-7 Primary and secondary thread switching

As shown in the figure, the actual amount of time that the secondary thread executes during the idle time slots is

$T_{sec} = T_{idle} - 2 * Tcs$

Where

T_{sec} = Secondary thread execution duration

T_{idle} = Duration of the idle time slot

T_{cs} = Time taken for a context switch through co-calls.

The division of the secondary thread into segments starts from the infinite loop. Before the integrated thread is run, the secondary thread has to be initialized by executing until the infinite loop.

2.4.2 Do's and Don'ts

In order to be able to integrate the two threads, there are some restrictions on the way the secondary thread is written. These are:

a) The code should not have any sub-routine calls. Given the control flow graph of the secondary thread, it should be easy to predict the start and end of the thread through each of it branches. Sub-routine calls introduce timing variability in the code since:
   - if the duration of sub-routine is not known beforehand, then actual execution time cannot be predicted in advance.

- If a subroutine is called multiple times, finding a proper location for a co-call or integrated code becomes more complicated. This is shown in figure 2.8.

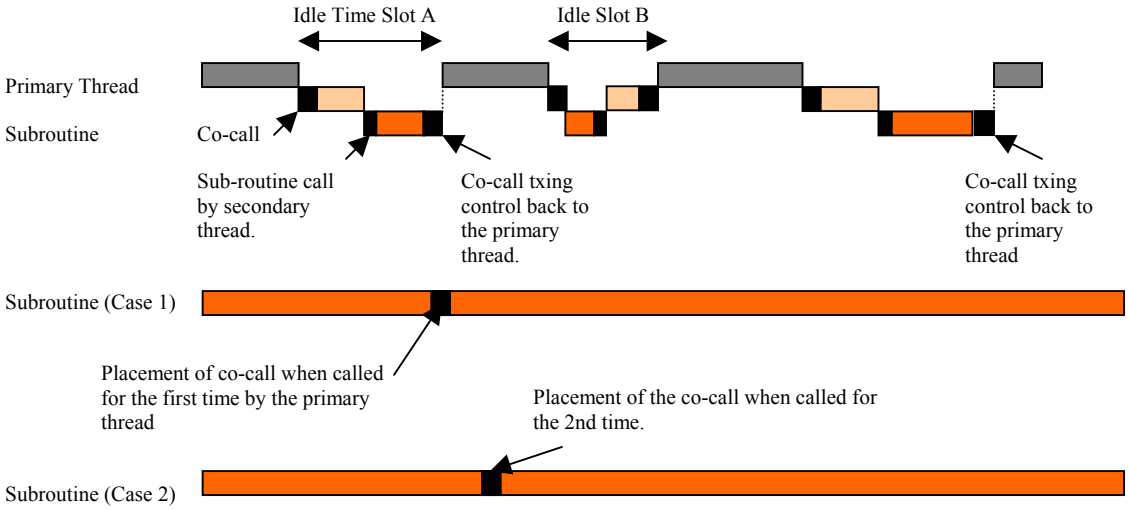


Figure 2-8 Subroutines in Secondary thread

As seen from the figure, the secondary thread, when invoked during an idle time slot A of the PLBF, calls a subroutine. After executing for certain time, the subroutine executes a co-call to return control to the PLBF. During idle time slot C, the secondary thread calls the subroutine for the second time. A co-call is again executed to return control to the PLBF. As seen from the figure, the placement of co-calls in the subroutine to return control to the primary thread is at different instants of time. This may be true for future invocations of the subroutine by the secondary thread. Hence the process of inserting co-calls in the secondary thread becomes cumbersome. A simpler method would be to pad the secondary thread so that the subroutine always executes a co-call at the same time. But this is left for future work.

Thus all sub-routines should be in-lined within main program.

b) The code can contain blocking I/O loops even though they produce timing uncertainty. Blocking loops occur when the secondary thread monitors the host controllers interface for any new commands. By padding, the uncertainty produced by blocking I/O loops can be removed.

2.4.3 Preparation

Before the secondary thread can the integrated with the primary, the actual secondary thread structure must be modified. These modifications are necessary to make the secondary thread ready to be split into segments. The modifications are:

a) **Pad Jitter:** All timing variability in the code has to be removed. This enables an accurate estimation of the execution of each instruction within the secondary thread. Once the execution of

the execution schedule of the thread is known, context switch instructions can be added at the correct places to switch back and forth between the secondary and primary threads. Padding is usually done when the branches of a conditional instruction are not of equal length or when an instruction is required to be executed only at a certain time. Padding is accomplished by inserting NOP instructions.

b) **Pad Blocking Loops**: Blocking Loops produce timing uncertainty within the secondary thread. This uncertainty can again be removed by padding. A blocking loop occurs when a loops exit condition is not affected by the loop's code, but instead an external process or hardware. For example, when the secondary thread is waiting for a command to be issued by the host interface, a blocking I/O loop is introduced.

To implement this a loop waits for a flag, indicating data reception, to be set in an I/O control register of the communication link between the micro-controller and the host interface. For example, in an Atmega103 8- bit micro-controller, when receiving a byte through an UART, the loop checks on the RXC bit of the USR register. The data is read from the UDR register only after the bit is set. An example of this loop is:

```
main:
        ldi r28,lo8(__stack - 1)
        ldi r29,hi8(__stack - 1)
        out __SP_H__,r29
        out __SP_L__,r28
        mov r8,r28
        mov r9,r29
        sec
        adc r8,__zero_reg__
        adc r9,__zero_reg__
        mov r11,r9
        mov r10,r8
        ldi r27,lo8(-1)
        mov r4,r27
        clr r5

/* Blocking I/O Loop */

.L32:
        sbis 11,7                   /* Checks to see if the "data available (RXC)" bit in the status register (USR) is set */
        rjmp .L32           /* If no data is received, wait for it */
        in r24,12           /* If data is received, read it from UDR and store it in  register 24 */
.L33
        nop
        nop
        nop
        ret
```

As shown in the sample code, the loop exit criterion checks for the bit 7 in register USR. If bit is cleared, control is passed back to label .L32. If the bit is set, the control advances to the next instruction. The problem with this structure is when the secondary thread is integrated with the primary thread. Using a context switch, control is passed to the secondary thread from the primary. After executing for a fixed amount of time, control is to be passed back to the primary. During integration, the context switch instructions need to be placed at certain periodic intervals within the secondary thread. This requires precise timing information on the control flow of the program. The control flow graph of the code given above is as shown in figure 2.9

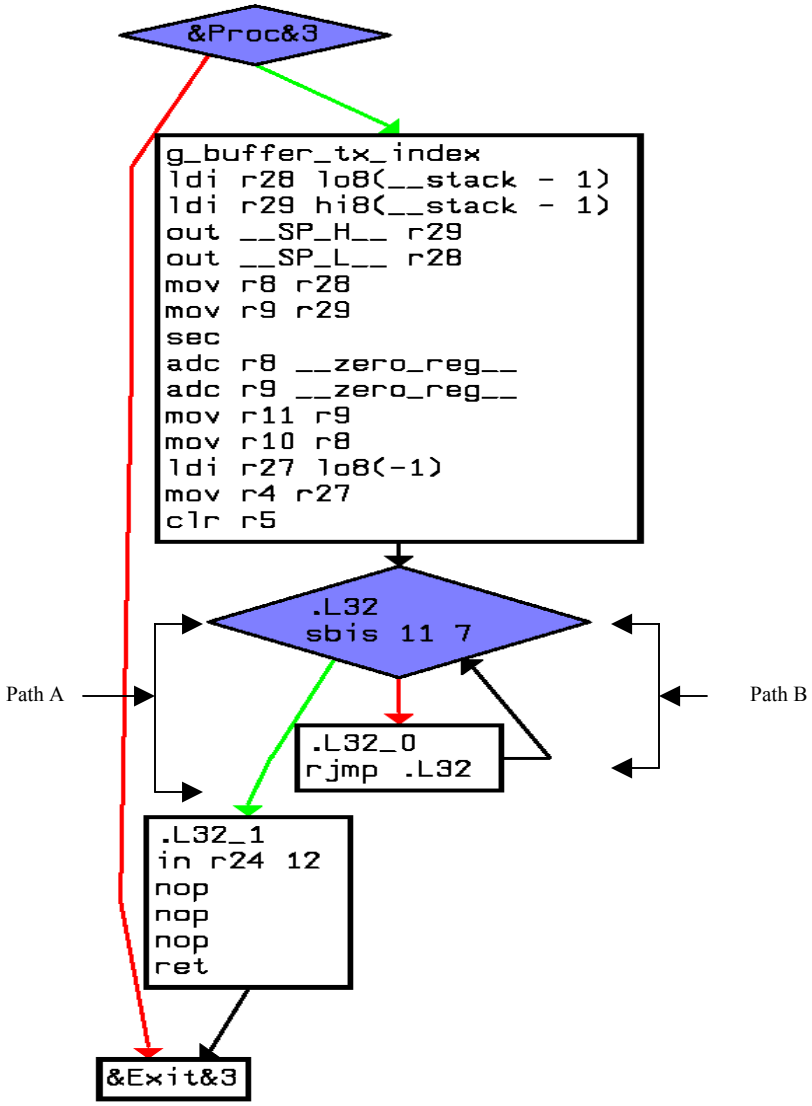


Figure 2-9 CFG for code having blocking I/O calls

As seen from the CFG, when the path A is taken, the time taken to execute the instruction stream is 3 cycles. But for the path B, the time taken to execute the stream is indeterminate. This poses a problem when inserting co-calls into the secondary thread to return control to the primary thread. To remove this timing variability, padding code needs to be inserted in path B. The padding inserted is just enough to ensure that the host interface executes for duration equal to the cocall period before returning control to the BLPF. After the cocall code, some padding is necessary to ensure that the number of cycles after the cocall and before the blocking I/O loop remain the same. This is required to ensure that the blocking I/O loop and the subsequent code executes at the same instant no matter what path the program takes while execution. The final modification carried out on the blocking I/O loop code is on the "skip" instruction. This instruction determines the path to take depending on a bit of an I/O register. The range of this instruction is one instruction. Thus, if the skip condition evaluates to true, the next instruction is skipped. Usually for a blocking I/O loop, when data in the interface register is valid, the skip condition evaluates to true. Hence, a blocking I/O loop comes in when the condition evaluates to false. Since the padding needs to be inserted when the condition is false and the range of the skip instruction being just one instruction, the skip condition is inverted. The modified code and its corresponding CDG is as shown in figure 2.10

```
.global     main
            .type       main,@function
main:
/* prologue: frame size=1 */
            ldi r28,lo8(__stack - 1)
            ldi r29,hi8(__stack - 1)
            out __SP_H__,r29
            out __SP_L__,r28
/* prologue end (size=4) */
            mov r8,r28
            mov r9,r29
            sec
            adc r8,__zero_reg__
            adc r9,__zero_reg__
            mov r11,r9
            mov r10,r8
            ldi r27,lo8(-1)
            mov r4,r27
            clr r5
.L32:
            sbis 11,7
            rjmp .L32
            in r24,12
.L33
            nop
            nop
            nop
            ret
```

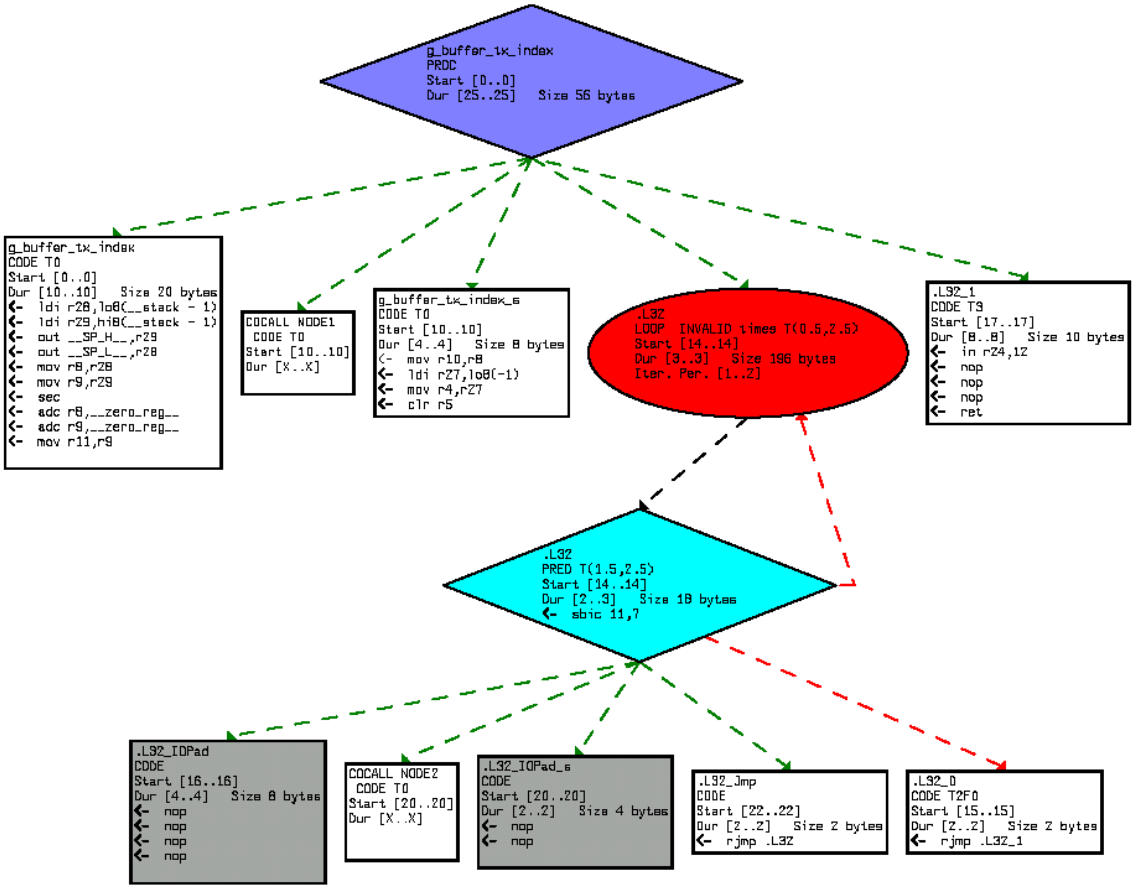


Figure 2-10 CDG for the modified blocking I/O loop code

The above code has a cocall duration of 10 cycles. Since the true condition of the skip instruction terminates the loop, the skip instruction is inverted. Thus “sbis 11,7” is now converted to “sbic 11,7”. Now the true condition causes the loop to be executed and the false condition terminates the loop. When the blocking I/O loop code is entered, 4 cycles have been executed after “Cocall 1” and before the loop. Hence 6 cycles need to be executed before a cocall can be executed. This padding is inserted in the “true” path of the skip condition as seen in the figure. After the cocall, 2 cycles are inserted so that the code node .L32_1 starts at the same instant irrespective of the path taken during execution.

The padding code in a blocking I/O loop insures that control is returned back to the primary thread when the exit criterion for the loop fails. Hence only a single iteration of the loop is executed in each idle time slot of the primary thread. Also, each secondary thread segment being executed in an idle time slot can have no more than one blocking I/O loop. Now as seen in figure 2.10, co-calls could be placed in at periodic intervals within the secondary without having to worry about the timing jitter brought about by the indeterminate duration of the blocking I/O loops.

c) **Predicate Node Padding:** In order to switch between the primary thread and secondary thread, cocalls have to be inserted within the secondary thread at regular intervals. The periodicity of this insertion depends on the actual idle time available in the primary thread to do useful work. In other words,

$$T_{sec} = T_{idle} - 2 * Tcs$$

Where

T_{sec} = Secondary thread execution duration

T_{idle} = Duration of the idle time slot

T_{cs} = Time taken for a context switch through co-calls.

T_{sec} is the periodicity of cocall insertion. In order to be able to insert cocalls within the secondary, precise timing information is required on the control flow of the program. Since predicate nodes generate timing variability, padding has to be done to remove this uncertainty.

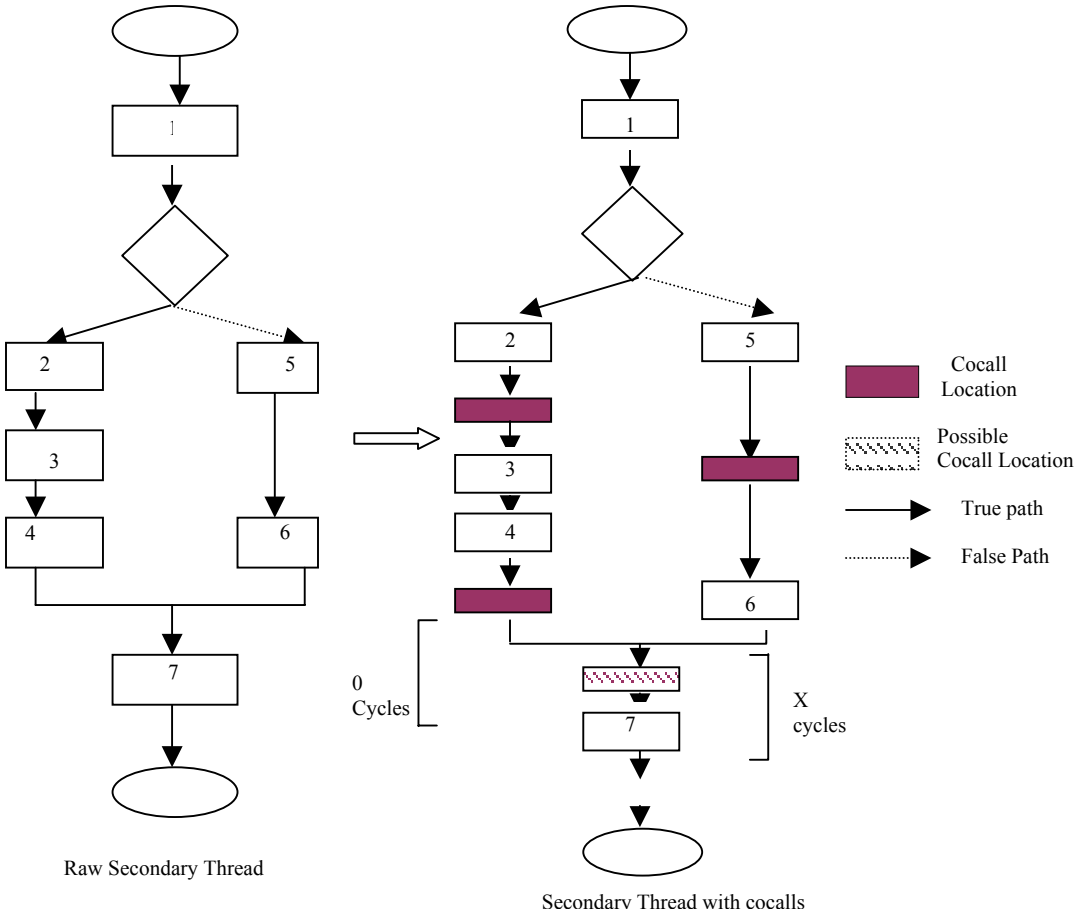


Figure 2-11 Predicate Padding in secondary

However since padding is required to ensure that cocalls are executed at the right places, the two branches of the predicate node causing the jitter need not be padded to the same amount.
The amount of padding inserted should ensure that the time left after the last cocall insertion within both the nodes are the same. This is illustrated in the figure 2.11.

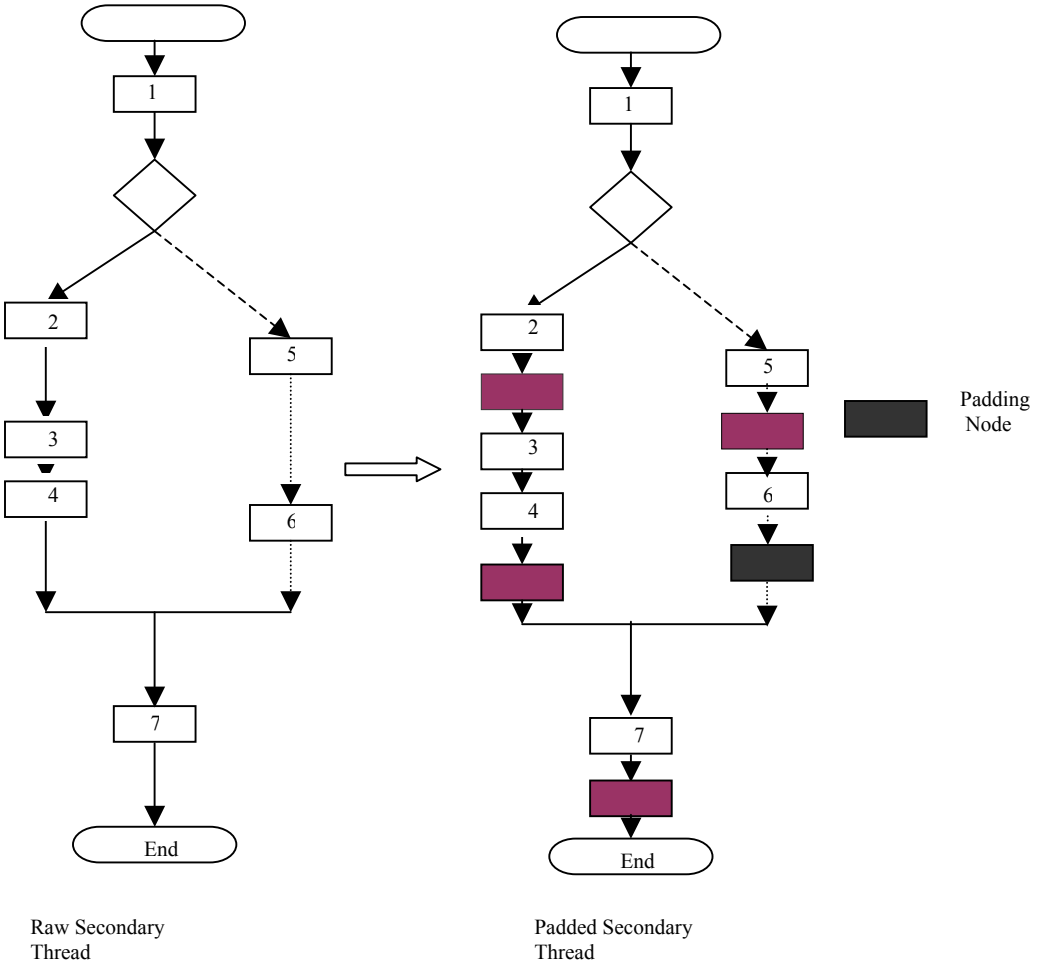


Figure 2-12 Padded Secondary Thread

As seen from the figure, the "raw" secondary thread code has timing variability due to the predicate node. When the True node of the predicate node is taken, there is no time left after cocall insertion. However when the False path is taken then time left after cocall insertion is X. To remove this timing variability, a padding node of duration X is inserted as shown in figure 2.12. This ensures that no matter which path is taken the cocalls are always inserted at the right places. This is of course true only when the predicate node duration is greater than the time to go for

inserting a cocall. If not, the predicate node is padded to make the two branches last for the same duration.

d) **Loop Padding:** If the secondary thread has any loops, then the cocall insertion becomes slightly complicated. To insert cocalls within a loop, the loop iteration in which the cocalls have to be inserted is to be determined. This requires a thorough data flow analysis of the code. An alternative would be to pad the loop so that a single iteration of the loop lasts for atleast a cocall duration. This eliminates the need for any data flow analysis, but results in code expansion. Padding is also inserted to ensure that time after the last cocall within the loop is same as the time left for a cocall insertion before the start of the loop.

2.4.4 Transformations

2.4.4.1 Basics

Integration is achieved by placing co-calls in both the primary and secondary threads so as to enable context switching between them. For the primary thread, co-calls are placed in the idle time slots to switch to a secondary thread. For the secondary thread, co-calls are placed at periodic intervals within it. This, again, depends on the actual idle time available in the primary thread to do useful work.

2.4.4.2 Segments

Segment denotes the length of the secondary thread that can be executed in an idle time slot of the primary thread.

As described before, for the sake of simplicity, most of the idle time in the primary thread is made to occur in the bit level functions (for communication protocols). The idle time present in these functions is also highly fragmented. Hence a better measure of the amount of idle time present in the primary thread to be utilized for secondary thread execution is the segment idle time ($T_{SegmentIdle}$).

As described before ,$T_{Segment}$ was defined as

$$T_{Segment} = T_{BubbleEnd}(i) - T_{BubbleStart}(j)$$

Where

$T_{BubbleStart}(i)$ = Start of the first idle time bubble “i” whose duration is greater than the cocall length and

$T_{BubbleEnd}(j)$ = End of the last idle time bubble “j” whose duration is greater than the cocall length.

Also, within $T_{Segment,}$ $T_{SegmentIdle}$ is the amount of idle time that is actually available for useful secondary thread work. It is defined as

$$T_{SegmentIdle} = \left(\sum_{i}^{j} T_{Bubble}(i) \right) - 2 * T_{CS}$$

For the purpose of integration, co-calls are inserted into secondary thread every $T_{SegmentIdle}$.

2.4.4.3 Effect of Blocking I/O Loops

When the secondary thread contains blocking I/O loops, then the timing estimates on the placement of co-calls within the secondary thread requires some consideration. The CFG of a typical blocking I/O loop is as shown in figure 2.9 that is copied here again for convenience shown in fig 2-13. As explained before, there can be two paths through the loop. When execution follows path A, then the time taken to execute from Point A to B is known accurately. On the other hand, when path B is

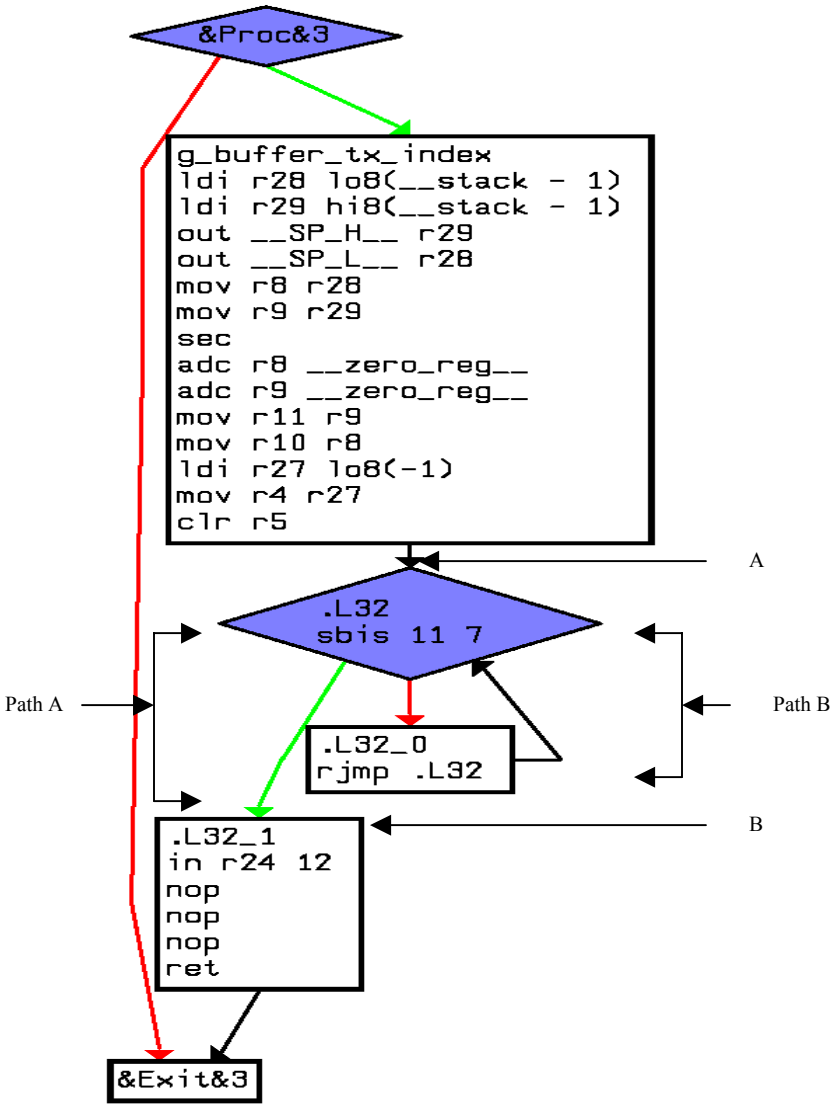


Figure 2-13 CFG for Blocking I/O loop code

followed, the execution time from point A to B is not known. But as explained before, breaking up the loop back path rectifies this. The Loop-back path is padded to last for at least $T_{SegmentIdle}$ and then a co-call is executed to return to the primary thread. Some additional padding is also added as explained before. Thus once the blocking I/O loop code is modified as explained before, cocalls can be inserted at the appropriate instants of time.

This approach of inserting padding and co-calls in the feedback loop simplifies the calculation of timing estimates for the placement of co-calls. The modified CDG with cocalls inserted, is as shown in figure2-14.

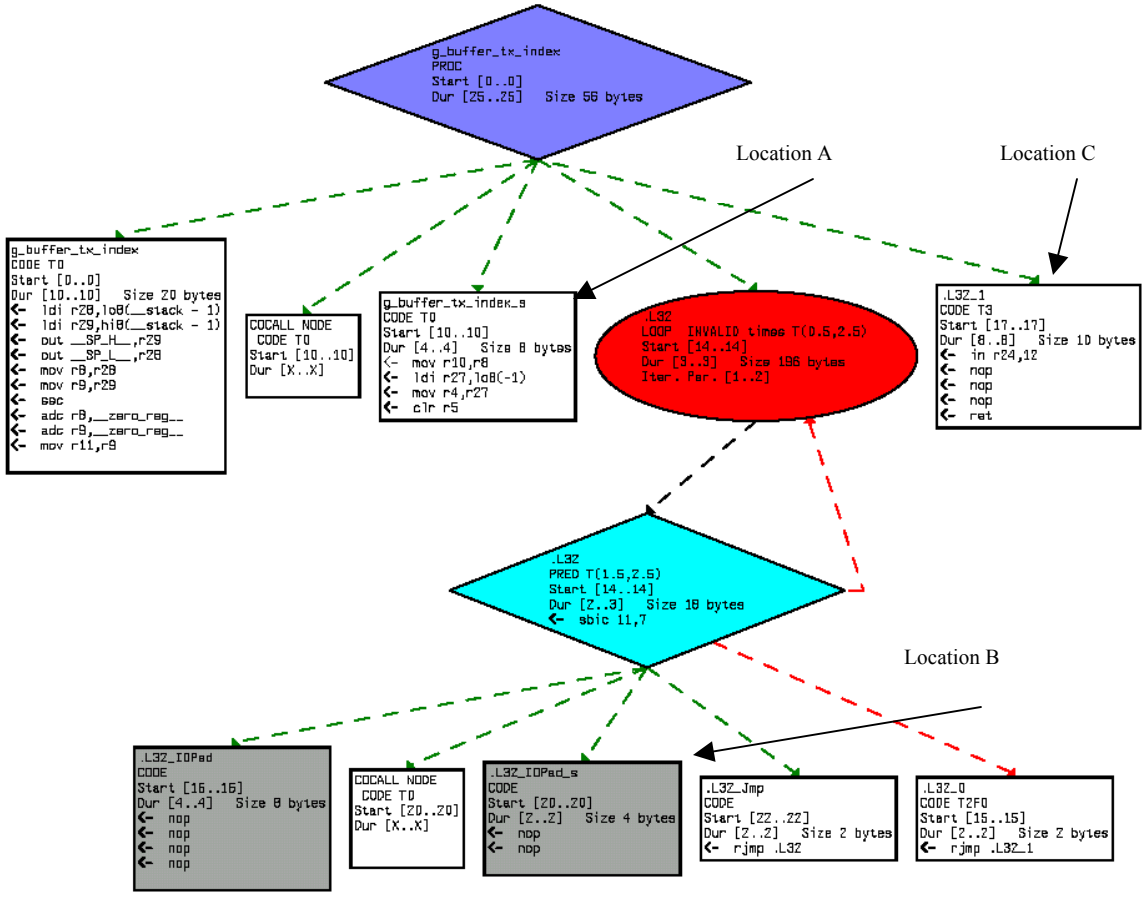


Figure 2-14 Modified CDG

Now as seen in figure 2-14, if the time between "Location A" and "Location C" is same as the time between "Location B" and "Location C", then the only thing to consider while placing co-calls is the time taken to execute the code segment between point A and point B. In most blocking loops, this code segment contains just two instructions, a skip instruction and a rjmp instruction, to transfer control out of the loop. Thus, for the purpose of calculating cocall duration, the blocking I/O loop can be thought of as executing for a duration of 1(skip evaluating false) + 2 (rjmp instruction) = 3 cycles.

2.4.4.4 Co-Call Insertion:

Co-calls are inserted into the primary and secondary threads to perform context switching during the idle time slots of the primary thread. For the primary thread, the co-calls are inserted during the idle time slots. For the secondary thread, co-calls are inserted periodically at $T_{SegmentIdle}$ intervals.

In communication protocols, there is usually more than one PLBF implemented. For example, in most of the communication protocols, the send_bit and the receive_bit functions are called by the primary thread. Also, in some protocols, control may jump from one PLBF to another without the earlier thread executing to completion. For example, in the J1850 protocol, an IFR response request causes the receiving node to temporarily abort its reception and transmit a response to the transmitting node. After the transmission is complete, the node goes back to the receiving mode. This switching between the PLBF's poses a problem when they have to be integrated with a single secondary thread. The PLBF's may have the same or different idle time periods. In the simplest case, when the PLBF's have same idle time periods, the single secondary thread could be called to execute at the idle time slots. On the other hand, when the idle time slots are of different duration, a single secondary thread cannot be called without some modifications. This is mainly because of two reasons:

a) The amount of secondary thread executed during the idle slots of each primary thread is different. Each segment of the secondary thread lasts for $T_{SegmentIdle}$. Hence, co-calls are inserted in the secondary thread at a periodicity equal to $T_{SegmentIdle}$. For two different primary threads, this value may be different. This is as shown in figure 2.15

   In the figure, T1(P1), T2(P1) and T3(P1) represent the fragmented idle time slots of PLBF P1 while T1(P2), T2(P2) and T3(P2) represent the idle time slots of the PLBF P2. In order to extract more idle time, the intervening code in the PLBF is removed. Thus, the actual time available to do useful secondary thread work, or $T_{SegmentIdle}$ is equal to the sum of the idle time slots. This is shown in the figure 2-13. Here, an assumption is made that all the idle time slots are longer than the minimum idle time period. (A minimum idle time slot is defined as one whose duration is long enough to execute at least one cocall). Hence when a switch is to be made from one PLBF to the other, the secondary thread segments now need to execute for $T_{SegmentIdle}$ corresponding to the new primary thread.

b) Since the $T_{SegmentIdle}$ for PLBF's may be different, the cocalls do not coincide, complicating a switch from one to another. Consider figure 2.16 that depicts the secondary thread implementations for different primary threads.

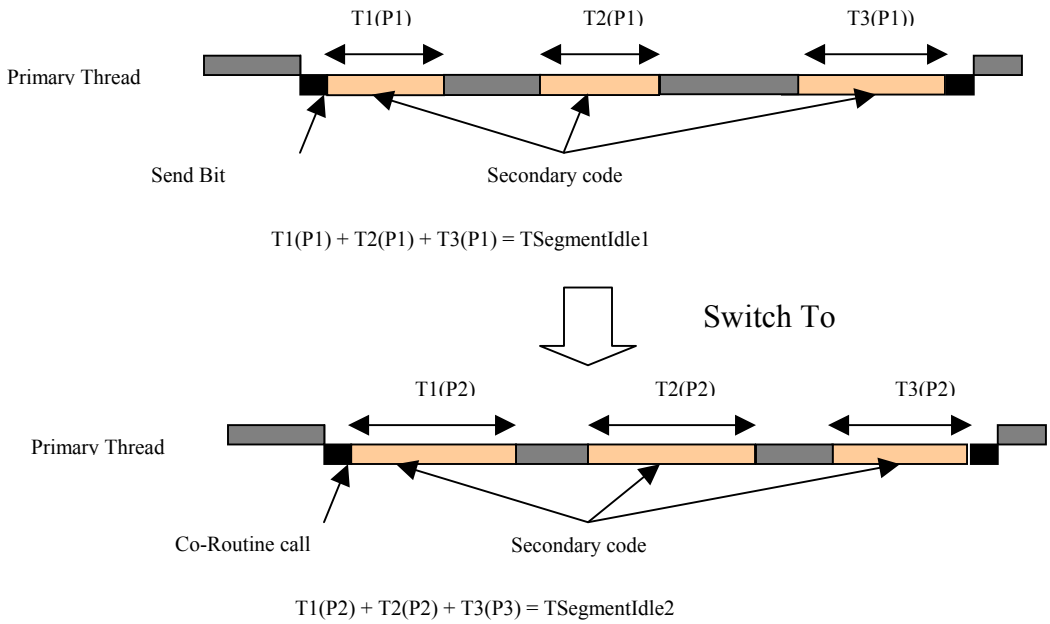


Figure 2-15 Switching between Primary threads

The first is the normal secondary thread without any co-calls inserted. The next two versions denote secondary threads with co-call placements corresponding to the idle time slots of PLBF's P1 and P2. In the first one, PLBF has an idle time slot of duration T1 whereas in the second, the idle time slot is of duration T2. Suppose initially, PLBF P1 executes which in turn calls secondary thread S1. Each segment of the secondary runs for duration T1 before returning control back to P1. When point "A" is reached within the secondary, it becomes known that PLBF P2 has to start executing. When P2 executes, it calls secondary thread S2 during its idle time slot. Now as seen in the figure

$$T2(P2) > T1(P1)$$

Hence the co-call placements in S1 and S2 are different as seen in the figure. When P2 calls S2 for the first time, either S2 can start executing from point "B" or from point "C" for the secondary thread to get synchronized. If execution starts from point "B", then a small portion of the secondary (Node 1) is re-executed incorrectly, potentially corrupting the program. On the other hand, if we choose to start the execution from "C" then a small portion of the code has to be executed(Node 2) before the control is passed to S2 at "C".

From the above discussion, an obvious solution to simplify the switching between primary threads is to use three separate secondary thread implementations. Two secondary thread implementations have co-call placements corresponding to $T_{SegmentIdle1}$ and $T_{SegmentIdle2}$ while the third implementation is for synchronization. Now before switching from one version of the secondary thread to the other, the

synchronizing thread is called to execute until the nearest sync point. Then control is passed over to the relevant secondary thread.

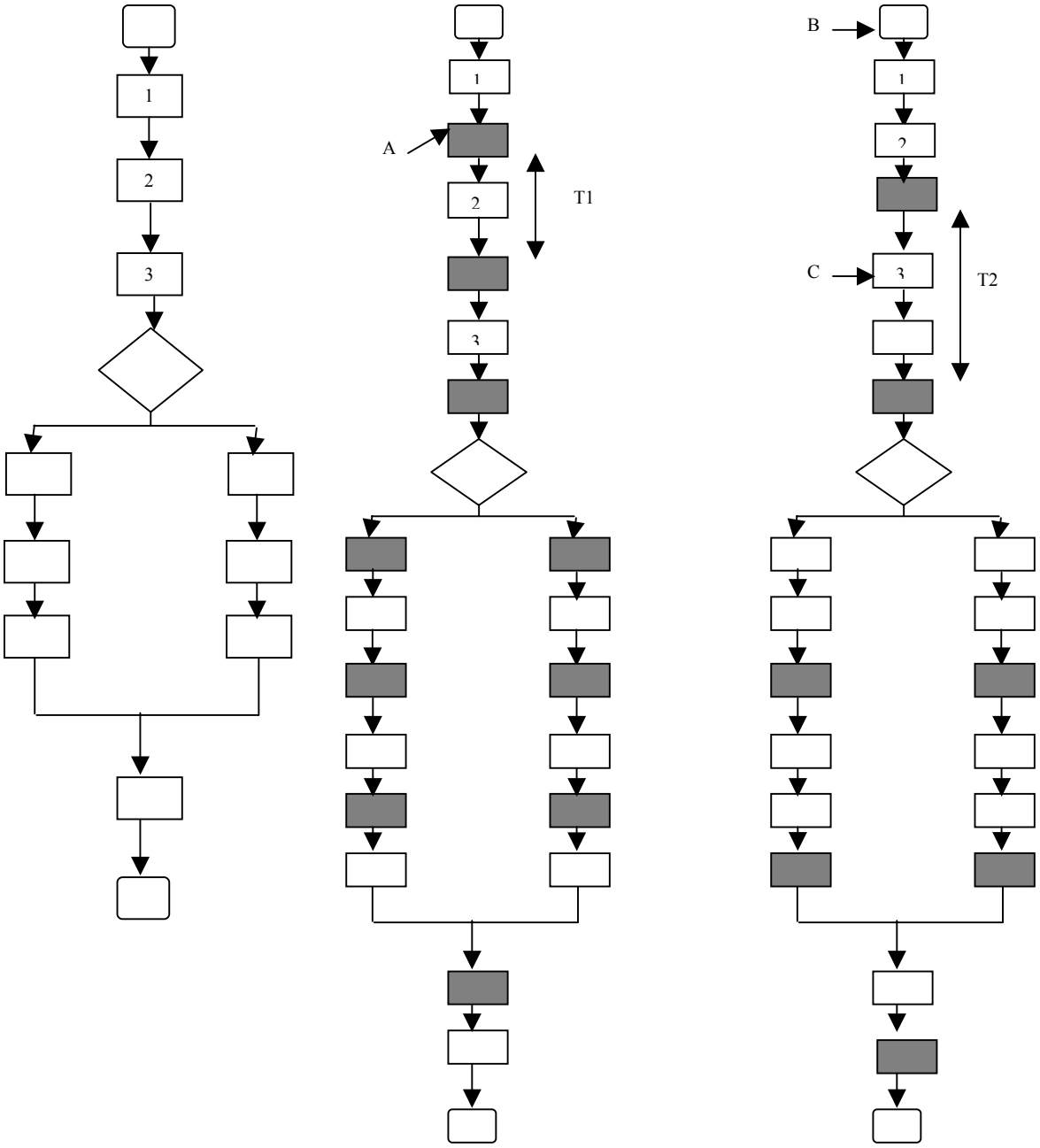


Figure 2-16 Synchronization

This scheme of having separate secondary threads for each PLBF (having different idle times) has the obvious disadvantage of code explosion. Each version of the secondary thread may execute for only a short duration before switching to the other version. To overcome this, a single secondary thread could be used with "guarded" co-calls. Thus portions of the secondary thread code are executed conditionally.

To use a single secondary thread for all the primary threads, the secondary thread must contain co-calls corresponding to $T_{SegmentIdle}$ of all the PLBF's. Since the PLBF's can have different idle time durations, the secondary thread requires co-call placement at different locations for different PLBF's. Thus, for the secondary,

$$N(co\text{-}calls) = N(P1) + N(P2) + \ldots\ldots + N(Pn)$$

Where

N(co-calls) = Number of co-calls in secondary.

N(Pi) = Number of co-calls due to primary thread i.

This is of-course true only when none of the co-calls coincide. When come co-calls due to different primary threads coincide, then the above relation does not necessarily hold.

For a given primary thread, not all the co-calls in the secondary have to be executed. This requires the co-calls to be guarded. Hence now a portion of the idle time is spent executing conditional code, which determines whether the next co-call is for the current primary thread, or not. The secondary thread is structured now as follows:

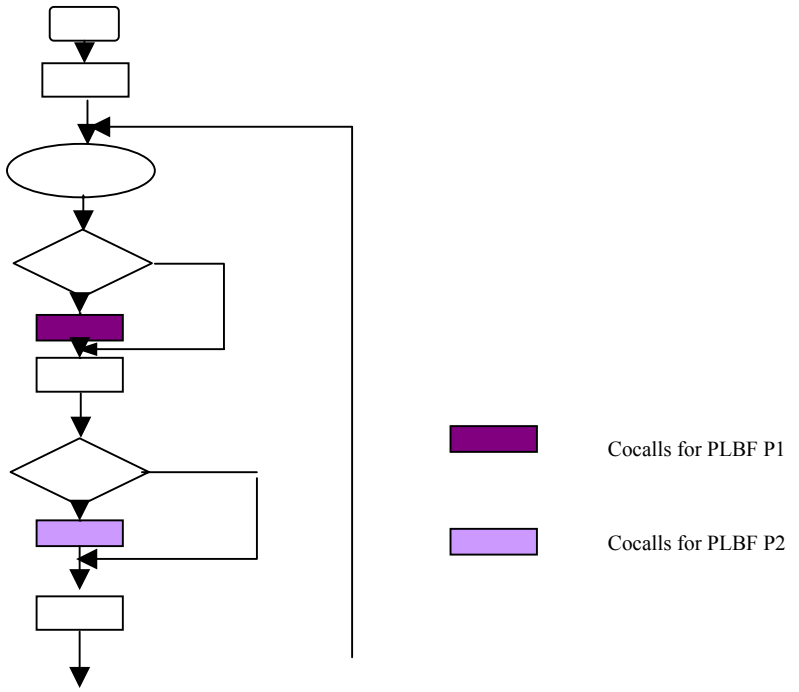


Figure 2-17 Guarded Cocalls

As seen in the above figure, the secondary thread has interleaved co-calls to the various PLBF's. Each co-call is guarded by some conditional code so that the co-call executes only when called by the appropriate primary thread.

2.4.4.5 Intervening guest Code insertion

When the idle time in primary thread is fragmented and many of these chunks have idle time durations greater than the cocall period, then the guest code between the first and last such idle chunks is removed. This is now inserted at several places within the secondary thread such that they would execute at almost the same time as they would have executed in the primary thread. Consider the PLBF code given in figure 2-18. that has be to integrated with the secondary thread code given in figure 2-19

```
.global     receive_bit
            .type       receive_bit,@function
receive_bit:
            in __tmp_reg__,__SREG__
            cli
            out __SREG__,__tmp_reg__

_AVR_First_Sample:
            in r20,0x16
            out 0x1B,r20

_AVR_Second_Sample:
            in r2,0x16
            out 0x1B,r2
            sts sample_value1,r2
```

```
_AVR_Third_Sample:
            in r2,0x16
            out 0x1B,r2
            sts sample_value2,r2
_AVR_End_Pad:
            mov r24,r20
            lds r18,sample_value1
            lds r19,sample_value2
            and r24,r18
            and r18,r19
            or r24,r19
            and r20,r19
            or r24,r20
            clr r25
            ret
```

Figure 2-18 PLBF Code

```
.global     main
            .type       main,@function
main:
            ldi r28,lo8(__stack - 0)
            ldi r29,hi8(__stack - 0)
            out __SP_H__,r29
            out __SP_L__,r28
            in r18,12
            in r24,12
            tst r18
            breq .L55
            mov r24,r25
            subi r24,lo8(-(4))
            cpi r24,lo8(3)
            breq .L56
            out 12,r18
            mov r25,r24
            subi r25,lo8(-(60))
.L24:
            cp __zero_reg__,r25
            brge .L54
            ldi r18,lo8(100)
.L40:
            in r24,12
            subi r24,lo8(-(10))
            cpi r24,lo8(100)
```

```
            brge .L50
            out 12,r24
.L39:
            subi r25,lo8(-(-1))
            brne .L40
            rjmp .L54
.L50:
            out 12,r18
            rjmp .L39
.L56:
            out 12,r25
            ldi r25,lo8(2)
            rjmp .L24
.L55:
            tst r24
            breq .L57
            out 12,__zero_reg__
            mov r25,r18
            subi r25,lo8(-(10))
            rjmp .L24
.L57:
            out 12,__zero_reg__
            mov r25,r18
            rjmp .L24
.L54:
            nop
            nop
```

Figure 2-19 Secondary Code

Let us assume that the following parameters have to be observed during integration.

Parameter	Number of cycles	Description
T_{Sec} = 10 cycles	10	Duration of the secondary thread segment execution
$T_{CocallExec}$	5	Execution time for a single cocall
$T_{LenIdle}$	12	Length of the first idle time slot greater than TcocallExec
T_{IG} (_AVR_Second_Sample)	12	Start of the Intervening Guest (IG) "_AVR_Second_Sample" w.r.t the previous implicit guest
T_{IG}(_AVR_Third_Sample)	2	Start of the Intervening Guest "_AVR_Third_Sample" w.r.t "_AVR_Second_Sample"

The node _AVR_First_Sample in the PLBF is an implicit node and serves as the timing anchor while calculating the idle time. From the above values we see that the Intervening Guest "_AVR_Second_Sample" should execute $(T_{LenIdle} - T_{CocallExec})$ = 7cycles after the start of the each secondary thread segment.

Before integration, the CDG of the secondary thread is as shown in figure 2-20. Once the secondary is integrate with the PLBF, the intervening guest codes "_AVR_Second_Sample" and "_AVR_Third_Sample" are inserted at the appropriate locations within the secondary (yellow boxes) as shown in figure 2-21

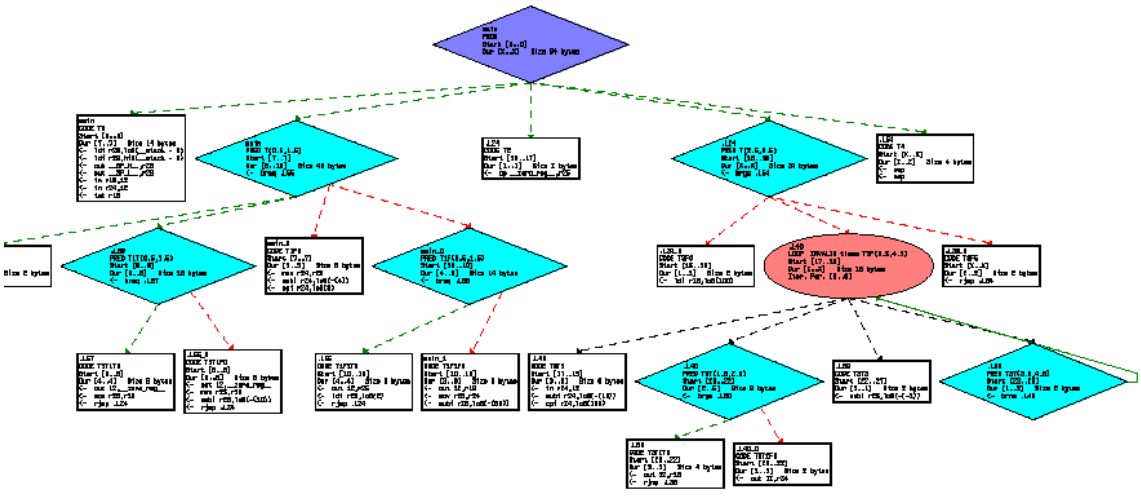


Figure 2-20 "Raw" Secondary code

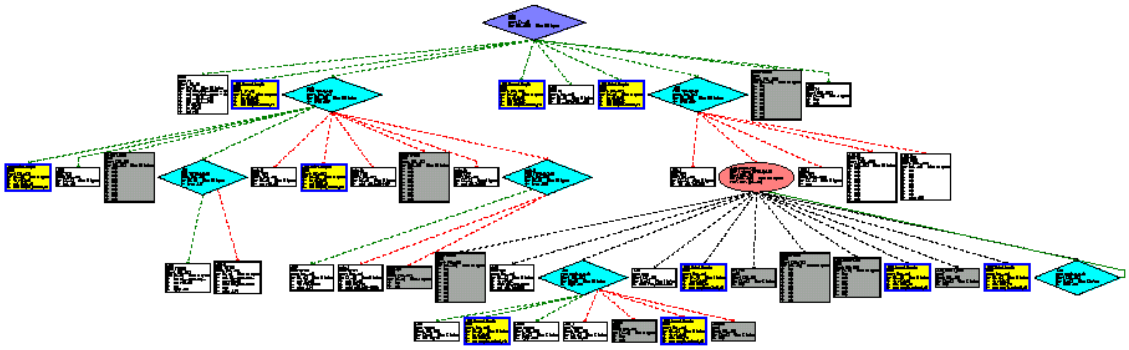


Figure 2-21 Secondary thread with Intervening code

In order to use a single secondary thread for all the PLBF's, the intervening code of all the PLBF's also has to be integrated within the secondary thread at the appropriate instants. As with the cocalls, to prevent the wrong code from being executed, the interleaved code has to be guarded. Thus, conditional code is put in before the primary code to check whether the code is for the PLBF being executed or not. This is shown in figure 2-22.

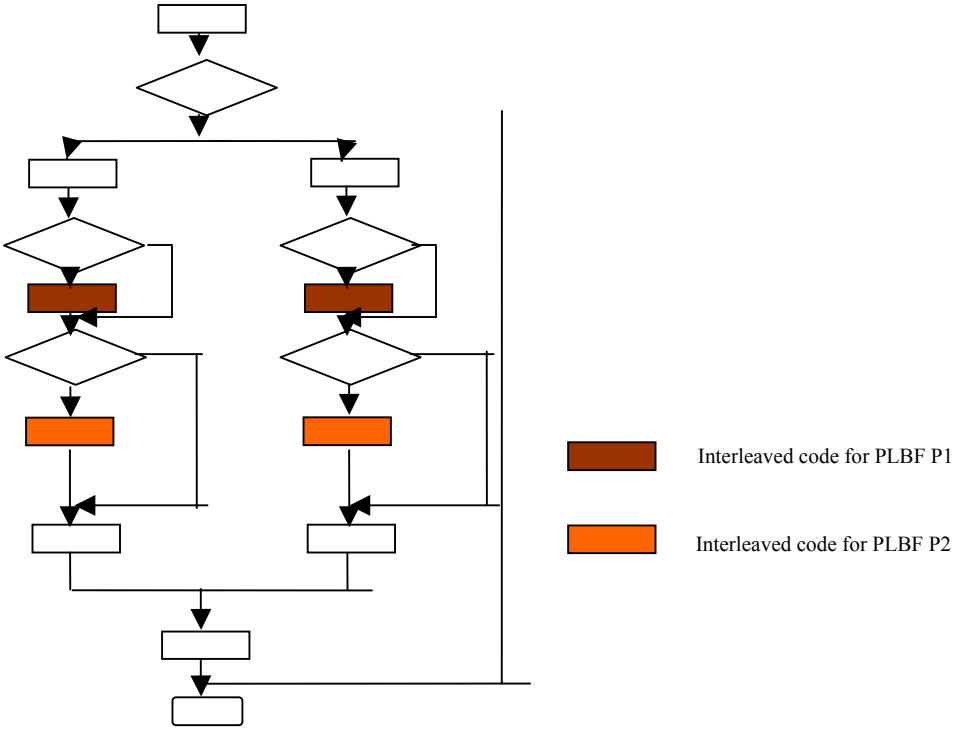


Figure 2-22 Guarded Intervening Code

Resynchronization is still required when the control is switched from one PLBF to another while the execution of the secondary is still incomplete. This can be accomplished by using a dedicated set of flags whose value determines the type of action to perform. The number of flags used in the register depends on the number of PLBF's being run. In this work STI is being demonstrated on a protocol PLBF's,. send_bit and receive_bit. Hence, only 4 flags are required for this purpose. Embedded processors typically feature bit-test instructions. This allows the use of a single variable (memory location or a register) to hold many or all of the flags and also allows quick flag testing. The flags are held in a register shown below:

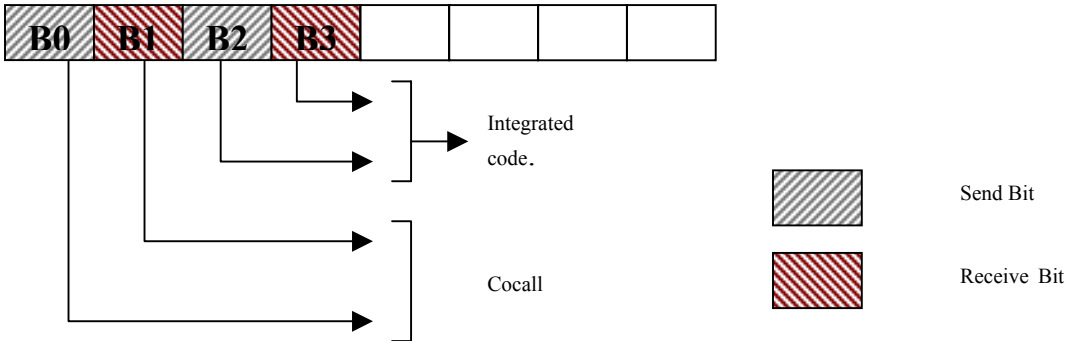


Figure 2-23 Register For Synchronization

The first 2 flags determine which co-calls are to be executed. The number of bits used is equal to the number of PLBF's being run. In this case, we are in the process of executing 2 PLBF's. These are

a) send_bit and
b) receive_bit.

Thus a "0 1" combination implies that next a receive_bit co-call must be executed. Note that the combination "1 1" is invalid.

The next 2 bits in the register determine whether or not to run the integrated secondary thread code. Thus, a "0 0" value would suggest executing no integrated code. Instead it would simply execute until reaching an enabled cocall based on the value of the first 2 bits. This enables resynchronization among segments for different PLBF's. If the first 2 bits read "1 0", then this would imply running through the secondary thread code until the synchronization point for the send_bit thread is found. Obviously this resynchronization code cannot be called from the idle time of any primary thread since the time between any two successive resync points of the primary threads may be greater than the available idle time of the primary thread from where it may be called. Hence this resync code is called by the higher-level functions (e.g the FSM or send_msg). A "1 0" value for the Intg bits would suggest running the send_bit integrated code from one of the resync points. As in the case of the previous 2 bits, a value of "1 1" for Intg bits is illegal since one cannot execute both the send_bit and the receive_bit integrated code at the same time.

3 J1850 Protocol

3.1 Introduction

SAE J1850 protocol is a communications standard used in off- and on- road land-based vehicles. Attributes of the J1850 protocol include an open architecture, low cost, master-less, single level bus topology. The SAE J1850 standard is a Class B protocol. Class B supports data rates as high as 100 Kb/s and typically supports intermodule, non-real time control and communications. SAE J1850 supports two main alternatives, a 41.6 Kb/s PWM approach and a 10.4 Kb/s VPW approach.
J1850 is an intermodule data communication network for the sharing of parametric information passed in frames between vehicle electronic modules connected to the Class B bus.

3.2 Signaling scheme

There are two different alternatives for implementing the SAE J1850 protocol. One is a high speed 41.6 Kb/s Pulse Width Modulation (PWM), two wire differential approach. The other is a 10.4 Kb/s Variable Pulse Width (VPW) single wire approach. Since the 10.4 kb/s approach is more prevalent, this scheme is used for demonstrating the coroutine based STI.
VPW is the encoding scheme of choice for all 10.4 kb/s J1850 applications. VPW offers one of the lowest radiated emissions encoding schemes possible due to the minimization of bus transitions per data bit. Some other key attributes of VPW are its ability to compensate for clock mismatch and ground offsets, fixed transition and sample points, the low number of transitions per bit and the fact that it is well suited for arbitration.
VPW communicates via time dependent symbols. Each logic 1 or logic 0 contains a single transition, can be either at the ACTIVE or PASSIVE level and have one of the two lengths, either 64us or 128us(t_{NOM} at 10.4 Kb/s baud rate), depending on the encoding of the previous bit. The bus can transition from a low potential to a high potential, or vice versa; but it the amount of time that the bus stays in its high or low potential that determines what a particular symbol is.
All the VPW bit lengths stated below are typical values at a 10.4 kb/s bit rate.
A logic 0 is defined as either:

– An active-to-passive transition followed by a passive period 64 µs in length, or
– A passive-to-active transition followed by an active period 128 µs in length.

This is shown in the figure 3.1
A logic 1 is defined as either:

– An active-to-passive transition followed by a passive period 128 µs in length, or
– A passive-to-active transition followed by an active period 64 µs in length

This is shown in figure 3.1

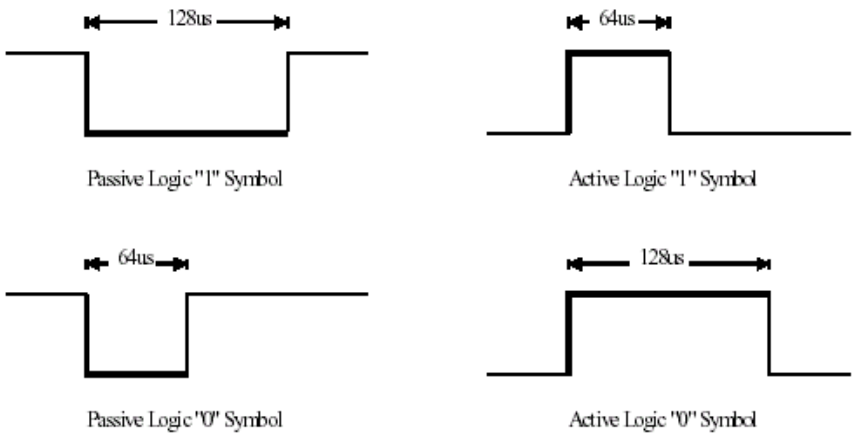


Figure 3-1 J1850 bit symbol timings

3.3 Arbitration:

SAE J1850 standard gives network allocation to each user node based upon the concept of arbitration. Arbitration is a process of determining which of the two or more of the nodes can continue to transmit when both or all the network nodes begin transmission simultaneously. Since all J1850 messages are asynchronous in nature, the message sending devices have to determine when a message transmission can begin and this has to be done with no pre-defined timing between messages.

The J1850 protocol supports CSMA/CR arbitration. CSMA/CR arbitration is a “non-colliding” scheme that supports master-less links. Before any node attempts a transmission, it first “listens” to the J1850 bus for a pre-set amount of time (“Carrier Sense”). If the J1850 bus is busy, then the “listening” node waits until the current message is complete before trying again. Because the J1850 protocol is peer-to-peer, offering equal network access to every node, “Multiple Access” (MA) capability is supported. “Multiple Access” means more than one node may begin transmission at the same time. “Collision Resolution” (CR) allows multiple transmitting nodes to all talk at the same time, and resolves the issue of who ultimately controls the bus through the utilization of message prioritization. Message prioritization is accomplished by allowing an active symbol to win over a passive symbol.

When a transmitting node(s) transmits a passive symbol but sees an active symbol, the transmitting node(s) knows that some other node of higher prioritization is transmitting too, (the node driving the active symbol). Any node that transmits a passive symbol on the J1850 bus but sees an active symbol has “lost arbitration”. Once arbitration is lost, the losing node(s) then stops transmitting and continues to function as a receiver. The node(s) that “won arbitration” continue to transmit, checking each bit of the current message and dropping out when necessary until just one node is left. Checking is done for every bit. This process is referred to as bit-wise arbitration.

All the nodes that have lost arbitration for this message can try again after the complete transmission of the current message, arbitrating again for the control of the bus.

3.4 Frame Format

All messages transmitted on the J1850 bus are structured according to the format shown in the figure 3.2

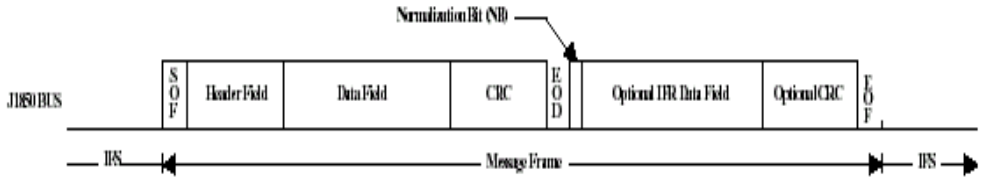


Figure 3-2 J1850 Frame format

The J1850 standard specifies that each message has a maximum length of 101 PWM (Pulse Width Modulation) bit times or 12 VPW (Variable Pulse Width) bytes, excluding SOF,EOD,NB and EOF with each byte transmitted most significant byte (MSB) first. The dynamic VPW J1850 message frame can contain one to eleven bytes of data. Both the header field and the data field must be contained within the one to eleven byte limit of a VPW J1850 message frame.

In the following descriptions, all VPW symbol lengths are typical values at a 10.4 kbps bit rate.

SOF — Start-of-Frame Symbol

All messages transmitted onto the J1850 bus must begin with a long-active 200-us period SOF symbol. This indicates the start of a new message transmission. The SOF symbol is not used in the CRC calculation.

HEADER FIELD

The first byte is designated as the header field. However, the header field can be modified to be either one or three bytes in length. The header field contains crucial information about what a receiving node should expect in the proceeding message frame. The entire byte is utilized as a message identifier with a single byte header field.

The first three most significant bits of the header field are priority bits. These three bits are capable of designating eight levels of priority with a binary 000 being highest and a binary 111 being the lowest in priority. These first three bits readily serve the arbitration process without having to influence any other header bit designations.

Following the priority bits is the header bit, or H-bit. This bit conveys whether or not this particular message is going to have a single byte header, or a three byte header. If the H-bit is set to “zero”, then a three byte header should be expected. If the H-bit is set to a “one”, then only a single byte header should be expected.

The next header bit designates whether an In-Frame Response, (IFR), is requested or not. This bit is known as the K-bit. If the K-bit is set to “zero”, then an IFR is requested. If the K-bit is set to a “one”, then no IFR

is requested. Following the K-bit is a bit utilized to designate an addressing type. This bit is called the Y-bit. Two types of addressing are accommodated with this J1850 protocol. One is a functional type of addressing, the other is a physical type of addressing. How the rest of the message is interpreted is dependent upon how this bit is set. If the Y-bit is set to "zero", then functional addressing is to be used. If the Y-bit is set to a "one", then physical addressing is to be used. Functional addressing has a higher priority than does physical addressing, providing all preceding bits are the same. Functional addressing is an addressing scheme that labels messages based upon their operation code or content. Physical addressing is an addressing scheme that labels messages based upon the physical address location of their source and/or destination(s).

The last two bits of the single byte header are the message type bits, or ZZ-bits.

These two bits tell any receiving node(s) what format the rest of the message is going to take. These two bits, combined with the Y-bit and the K-bit, offer up to sixteen different message types that can be transmitted via the single byte header. The Z1-bit helps to designate whether or not this message uses extended addressing. If the Z1-bit is set to a "one", then a receiving node(s) knows that the fifth byte of this message is the extended address of this function. The Z0-bit indicates whether or not data is contained in this message. If the Z0-bit is set to a "zero", then data should be expected with this message. If the Z0-bit is set to a "one", then no data is contained within this message. A three byte header is similar. With a three byte header, the H-bit is set to a "zero" to indicate that this is a three byte header. All other bits within the first byte operate essentially the same. The second byte of a three byte header contains a target address. The target address can be either functionally addressed or physically addressed. The third and last byte of a three byte header contains the physical address of the source of this message. Because the source address must always be unique, arbitration is always resolved by the end of the third byte whenever three byte headers are utilized.

Data — In-Message Data Bytes

The data bytes contained in the message include the message priority/type, message ID byte (typically the physical address of the responder), and any actual data being transmitted to the receiving node. Messages transmitted onto the J1850 bus must contain at least one data byte, and, therefore, can be as short as one data byte and one CRC byte. Each data byte in the message is eight bits in length and is transmitted MSB to LSB (least significant bit).

CRC — Cyclical Redundancy Check Byte

This byte is used by the receiver(s) of each message to determine if any errors have occurred during the transmission of the message. During transmission, the CRC is calculated and the CRC byte is appended to the message transmitted onto the J1850 bus. The receiver performs CRC detection on any messages it receives from the J1850 bus. CRC generation uses the divisor polynomial $X^8 + X^4 + X^3 + X^2 + 1$. The remainder polynomial initially is set to all 1s. Each byte in the message after the start-of-frame (SOF) symbol is processed serially through the CRC generation circuitry. The one's complement of the remainder then becomes the 8-bit CRC byte, which is appended to the message after the data bytes, in MSB-to-LSB

order. When receiving a message, the receiver uses the same divisor polynomial. All data bytes, excluding the SOF and end of data symbols (EOD) but including the CRC byte, are used to check the CRC. If the message is error free, the remainder polynomial will equal $X^7+X^6+X^2$ = $C4, regardless of the data contained in the message.

EOD — End-of-Data Symbol

Immediately after the CRC byte has been transmitted, an End Of Data (EOD) symbol is transmitted. The EOD symbol is a long 200-µs passive period on the J1850 bus used to signify to any recipients of a message that the transmission by the originator has completed.

IFR — In-Frame Response Bytes

Directly after the EOD symbol, receiving node(s) can opt to immediately respond to the message. This response is called an In-Frame Response (IFR) and contributes to the suite of error handling options that the J1850 protocol supports.

An IFR provides a platform for remote receiving nodes to actively acknowledge a transmission. Receiving node(s) append a reply to the end of the transmitting nodes original message frame. IFRs allow for increased efficiency in transmitting messages since a receiving node may respond within the same message frame that the request originated. A transmitting node request for an IFR is denoted by the K-bit in the transmitting nodes header field. All receiving nodes are required to check the K-bit and respond accordingly. This response must come within the elapse time between an EOD symbol and an EOF symbol, which is approximately 80ms. The beginning of an IFR response is denoted by the receiving node(s) transmission of a Normalization Bit (NB). The NB provides an active separation between the passive EOD symbol and the first data bit of the IFR response.

EOF — End-of-Frame Symbol

This symbol is a long 280-µs passive period on the J1850 bus and is longer than an end-of-data (EOD) symbol, which signifies the end of a message. Since an EOF symbol is longer than a 200-µs EOD symbol, if no response is transmitted after an EOD symbol, it becomes an EOF, and the message is assumed to be completed. The EOF flag is set upon receiving the EOF symbol.

IDLE — Idle Bus

An idle condition exists on the bus during any passive period after expiration of the IFS period (for example, >300 µs). Any node sensing an idle bus condition can begin transmission immediately.

4 Demonstration of Integration techniques on J1850

4.1 Overview

The techniques suggested for integrating primary and secondary threads (using cocalls) have been demonstrated on the J1850 automotive protocol. The protocol is implemented in software and is simulated on an Atmel AVR Atmega103 microcontroller. The software implementation consists of two threads, host interface thread (talking to the main host receiving commands and sending information) and the bus interface thread (responsible for framing/decoding message frames and sending/receiving bits from the bus). The host interface thread uses an UART built in the **Microcontroller** for communicating with the main controller and the bus interface thread uses general purpose I/O ports for sending/ receiving bits from the J1850 bus. Since the **Microcontroller** runs at a faster speed compared to the bit rate of the protocol, idle time is introduced in the PLBF's of the bus interface thread. To utilize this idle time, cocalls are used to switch between the bus interface thread and the host interface thread. In some cases, the idle time is fragmented. In such cases, the interleaved code in the PLBF (causing the idle time to be fragmented) is removed, thereby exposing more idle time and integrated with the host interface thread so as to execute at the same instant.

4.2 System Architecture

4.2.1 Hardware Architecture

J1850 data communication network uses a master-less, single-level bus topology. The nodes involved in the network share a common bus and use a bit-by-bit arbitration scheme to transmit messages on the bus.

4.2.1.1 J1850 Node Design:

The block diagram of a J1850 node is as follows:

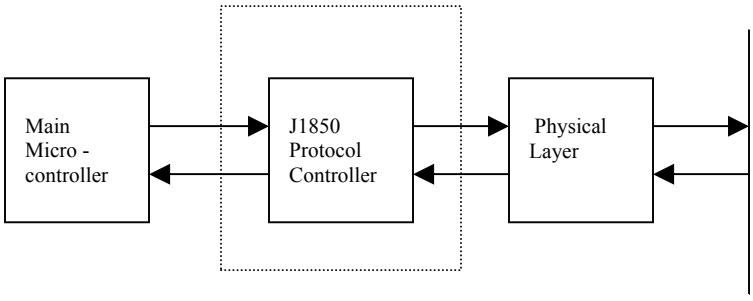


Figure 4-1 J1850 Node Setup

As seen in figure 4.1, the J1850 protocol controller is a piece of hardware acting as the interface between the main controller and the J1850 bus. It is responsible for receiving commands from the main controller and translating these commands into appropriate actions. These may be to send a message, receive a message, send status back to the controller or other protocol specific features. If the protocol controller is implemented as a dedicated hardware, then the design is highly customized to the protocol being implemented. For more flexibility, the protocol controller is implemented in software and run on a generic microcontroller. With this approach, a variety of processing could be carried out limited only by the speed of the microcontroller. Thus any microcontroller that is fast enough to meet the real time deadlines of the protocol and providing means to communicate with the main controller and the J1850 bus could be used for running the protocol. For the purpose of simulation, an Atmel Atmega103 8-bit microcontroller was used. Thus the block diagram of the hardware used for simulation is as shown:

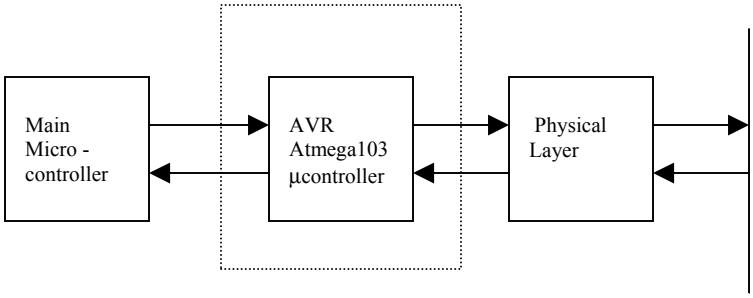


Figure 4-2 J1850 Hardware Setup

4.2.1.2 MCU Details

The Atmega103 is a low-power 8-bit microcontroller based on the AVR RISC architecture. It is capable of executing most instructions in a single clock cycle thereby achieving throughputs approaching 1MIPS per MHz. The microcontroller provides the following features:

- 128K bytes of in-system programmable flash.
- 4K bytes EEPROM
- 4K bytes SRAM
- 32 general purpose I/O lines
- 8 input lines,8 output lines
- Real-time counter
- 4 flexible timers/counters with compare modes and PWM
- UART, programmable watchdog timer with internal oscillator and a SPI serial port.

A full suite of program and system development tools including C compilers, macro assemblers, program debugger/simulators, in-circuit emulators and evaluation kits also supports the Atmega103 AVR.

For simulation, the UART is used to communicate with the main controller and a general purpose I/O port is used to transmit/receive bits. Thus the complete model used for simulation is as shown

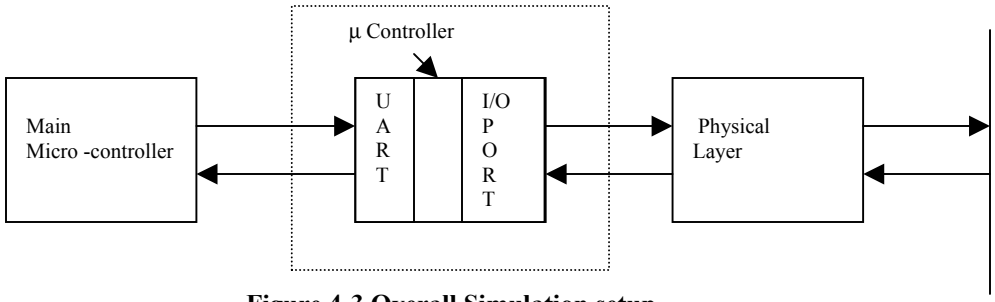


Figure 4-3 Overall Simulation setup

4.2.2 Program Structure

4.2.2.1 Overall Software Architecture

In order to implement the J1850 protocol controller in software, two threads are written. The first thread, called the host interface thread, interacts with the main controller through the UART. This thread is responsible for receiving commands from the host and if necessary communicating them to the other thread, also known as the BBCP (Bit Bang Communication Protocol) thread. The communication between the two threads is by means of message queues. This is as shown in figure 4.4

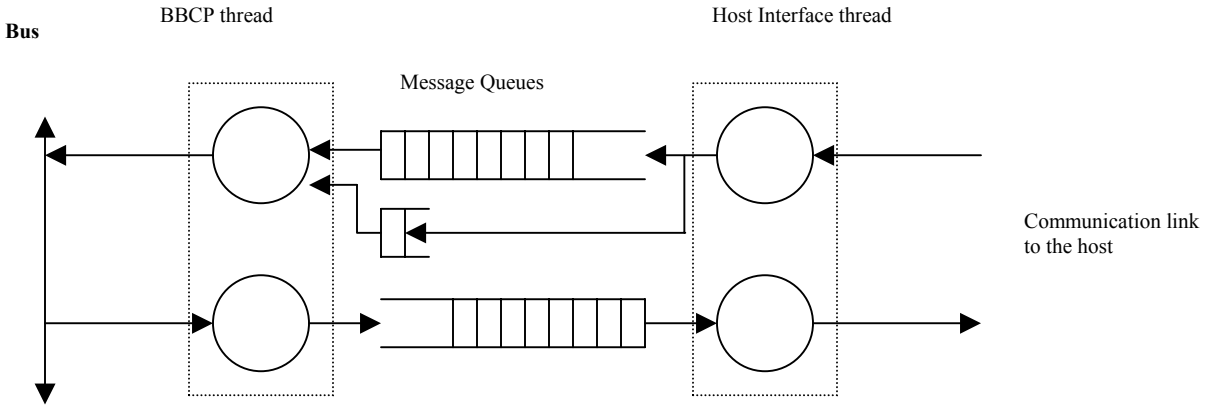


Figure 4-4 Communication Interface

The BBCP threads monitors the queue at periodic intervals and performs the functions requested by the host interface. It also receives any message sent on the bus and puts it on the queue for the host interface thread to receive it.

4.2.2.2 Message Queues

The message queues are used as the data interface between the host interface thread and the BBCP thread. As seen in figure 4.4, three message queues are implemented.

The TX Message queue is used to transfer the data from the host interface thread to the BBCP thread. It can hold upto a maximum of 16 J1850 data frames.

The RX Message queue is used to transfer the data from the BBCP thread to the host interface thread. This too can hold up to a maximum of 16 J1850 data frames. The data structure of the J1850 TX and RX queues is as shown in figure 4.5.

```
typedef struct J1850_Data_Buffer {
 BYTE_t Valid;                  // Entry is valid or not
 BYTE_t ProtocolID;             // Protocol ID. For J1850, the ID is 1
 BYTE_t MessageID;              // The message index within the queue
 BYTE_t IFR;                    // Used to indicate that an IFR response is desired for this frame.
 BYTE_t Length;                 // Length of the frame
 BYTE_t Data[20];               // Actual data to be sent or received.
}J1850_Transmit_Q,J1850_Receive_Q;
```

Figure 4-5 TX. & RX. Message Queue Data Structure

The IFR data queue is used to hold the IFR response sent by the main controller to the host interface thread. This data needs to be sent when the message being received by the BBCP threads requests for it. This queue can hold a maximum of one IFR data response. The data structure of this queue is as follows:

```
typedef struct J1850_IFR_Buffer {
        BYTE_t Valid;       // IFR Data validity
        BYTE_t Length;      // Length of the message
        BYTE_t Data[10];    // The actual IFR Data
}J1850_IFR_Buffer;
```

Figure 4-6 IFR Queue Data Structure

4.2.2.3 Host Interface thread

The typical structure of a host interface thread is as shown in figure4.7.

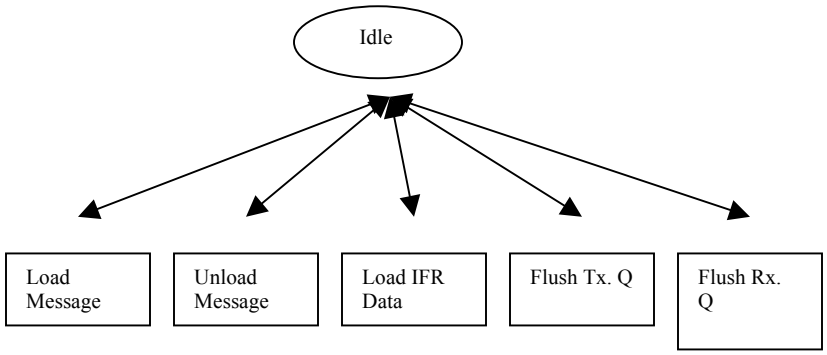


Figure 4-7 Host Interface Thread State Diagram

The host interface thread is implemented by means of a state machine. The transition between the states is triggered by the commands received by the host. The thread starts out in the idle state and based on the command received by the main controller, switches to an appropriate state. The host interface thread uses an UART running at 9600 baud to communicate with the main controller. This requires that the host interface thread be called once at least every 1.04 ms to service the UART. As shown in figure4.7, the J1850 host interface thread implements five states. These are :
Load Message: The host interface thread switches to this state when a "load message" command is received from the main controller. Here the thread receives the length and the data to be sent from the main controller and puts it on the TX message queue to be picked up by the BBCP thread.

Unload Message: In this state, the thread checks to see if any data is present in the RX message queue. This is done by checking the "Valid" bit. If data is available, then it initiates transfer of this data to the main controller though the UART.

Load IFR Data : This state is usually entered when a frame requesting IFR is received by the BBCP thread. In this state, the host interface thread expects IFR data to be sent by the main controller. First the length of the data is sent and then the data itself.

Flush TX Queue : In this state, the thread squashes all the entries of the TX queue by setting the Valid bit to 0.

Flush RX. Queue : In this state, all the entries of the RX Queue are squashed by setting the valid bit to 0.

4.2.3 Bus Interface Thread

The main functionality of the BBCP thread is to send/receive messages. Hence the thread contains code to send a frame and also receive a frame. But at an instant of time, only one of these two activities can occur. Hence, the send and receive functionalities of the BBCP thread are implemented as separate routines. The send and receive routines have real-time constraints and hence must be scheduled to occur at the correct instants. Thus a scheduling function is required to decide which functionality to call and when. In order to be able to switch between these routines, the BBCP thread is structured as a finite state machine shown below.

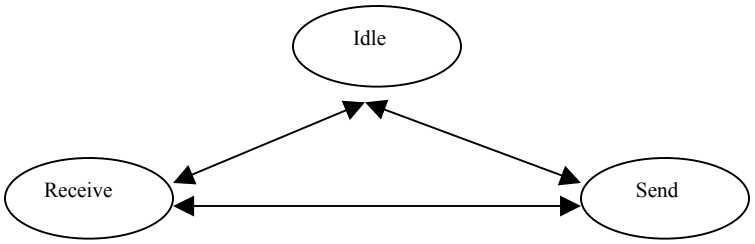


Figure 4-8 BBCP thread code structure

The transition between states is decided by events that occur either on the J1850 bus or the message queue between the BBCP thread and the host interface thread. To manage the transition between states and to maximize idle time, the send and receive functionalities are structured in form of layers. Each layer communicates with its upper and lower layer to realize its task. The layers comprising the send / receive functionalities are:

a) **Executive or Manager Function**: This is the top-level function that runs a finite state machine to monitor an idle bus, send a message or receive a message. As shown in the figure 4.9, when a request to transmit is received, the manager layer uses a subroutine call to pass on the request to its lower layer viz. the messaging layer. This function is different for send and receive units of J1850. For the send functionality, the manager function executes a polling loop that monitors the message queues for any frames to be sent. The manager function for the receive functionality is executed as an Interrupt service routine (ISR). The ISR is called when the J1850 bus makes a low-high transition signaling the beginning of SOF for a new J1850 frame. Some specific characteristics of the manager functions pertaining to J1850 are:

Send Manager Function:

- Keeps polling on the "Valid" bit of the message queue to check whether any messages are there to be sent.
- When data arrives, passes the data on to the message function.

Receive Manager Function:

- An ISR, called when the SOF is sent on the bus.
- Calls the Receive message to begin sampling the bus at the right instant.

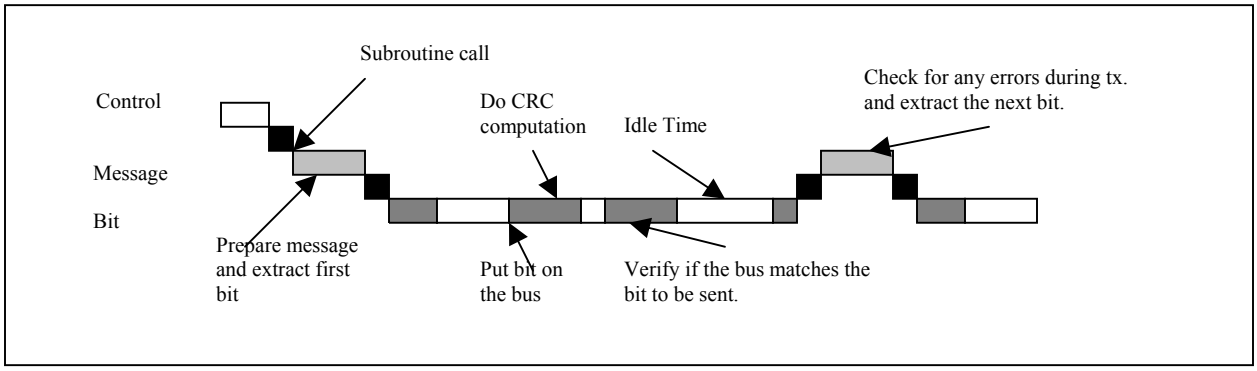


Figure 4-9 Timeline for Send function

b) **Message Level Function:** This is the middle layer that is called by the executive or manager function. This contains all the message oriented functions e.g. send_message, receive_message. These functions are responsible of encoding/decoding the message as per the protocol and then pass them to the corresponding layers. As shown in the figure 4.9, when sending a message, the send_message function forms the frame to be sent as per the protocol specifications and sends the frame to the lower layer bit by bit. Similarly, when receiving a message, the receive_message function puts the decoded message into the message queue to be picked up by the host interface thread. Some other characteristics of the message level functions with respect to J1850 are:

Send Message:

- Handles sending IFR data when called during message reception.
- Handle errors during transmission.

Receive Message:

- Calculates and checks CRC after the bits comprising the frame have been received.
- Executes a tight polling loop at the start of the function to determine the first falling edge. This determines the start of the data reception.
- Executes a padding loop to wait for 31 us after the first falling edge.
- Calls the receive bit function every 64us after the first sample. This ensures correct sampling since the bit durations on the bus can be either 64us or 128us.

c) **Bit Level Function**: This is the bottom layer that is responsible for putting/receiving bits from the bus. This contains functions like send_bit and receive_bit. These functions are called by the

message level functions when a message is to be sent or received. In addition to just sending / receiving bits, the bit level functions also do some additional tasks in J1850.

Send bit :

- Calculates the CRC while transmitting a bit.
- Samples the bus to determine whether it is free to transmit a bit (arbitration) before actually transmitting it.
- Switches between the pulse widths of 64us and 128us while sending alternate bits. This is required since J1850 uses VPW modulation for signaling.

Receive Bit:

- Samples the bus multiple times (3 times) while receiving a bit. The receive bit function "votes" on the bits received and decides on the value having the maximum "votes" i.e. 2.

4.3 Program Analysis

4.3.1 Primary Code – Idle Time

Since the **Microcontroller** is run at a speed (8MHz) much faster than the bit rate of J1850 (10.4kbps), idle time is introduced into the send and receive functions. The amount of idle time introduced is dependent on the bit rate of the protocol and the time taken for the send / receive function to execute. Figure 4.10 shows a timeline for a send bit functionality.

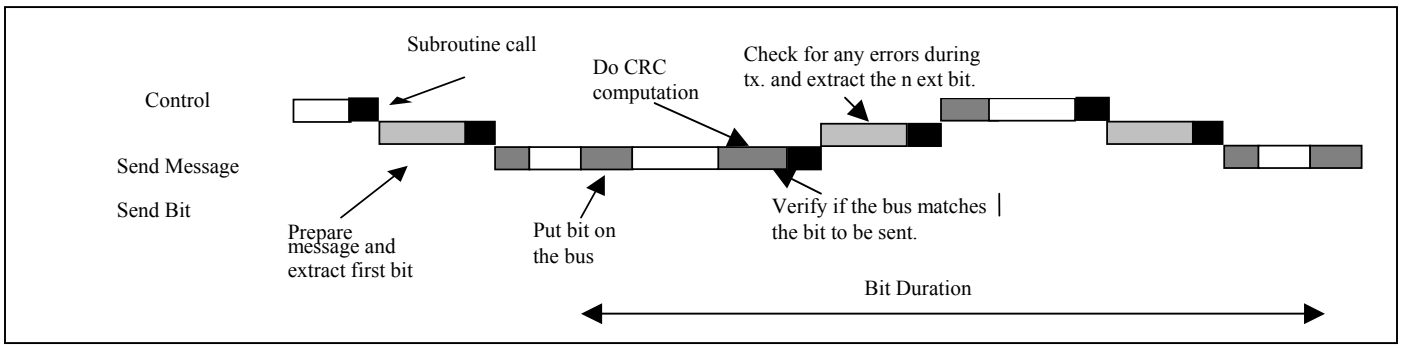


Figure 4-10 Complete Timeline for send bit

As seen from the figure 4.10, there are strict timing constraints on the bit timing as to when they have to be put on the bus. This results in periods of processor inactivity or idle time introduced. In J1850, the duration of a signal on the bus is variable. For example the SOF lasts for 200us, "1"

and “0” can be represented either by 128us or 64us and EOD –EOF separation can be 100us. As a result the idle time introduced is variable depending on the signal being sent. The idle time introduced varies from 328 cycles for a 64us signal to 1533 cycles for a 200us signal. These values are true for the send functionality and a clock speed of 8MHz. For the receive functionality, the idle time is for 433 cycles. The idle time does not vary for the receive functionality since the bus is sampled every 64us. A complete breakdown of the idle time values for the send and receive functionalities is given below:

Send :

Signal Value(in us)	Idle Time(in cycles)
200	1533
128	839
100	733
64	328

Receive:

Signal Value (in us)	Idle Time (in cycles)
64	433

As seen from the above values, there is a difference between the idle time values for the send and receive functionalities even for the same signal values. This can be explained by means of the following equation

$$T_{bit\_after} + T_{idle} + T_{message} + T_{message} = T_{bit}$$

Where

T_{bit_after} = Time between a bit is put/received from the bus and the bit-level function returns.

$T_{message}$ = Time taken by the message layer to execute between bit layer function calls.

T_{bit_before} = Time between the bit-level function starts executing and a bit is put/received from the bus.

T_{bit} = Bit time (1/Bit rate of the protocol).

For the send and receive functions, even though T_{bit} may be the same, $T_{bit\_after,}$ $T_{message}$ and $T_{message}$ are different. This results in different idle time values.

4.3.2 Secondary Thread – Determinism

The secondary or the host interface thread needs to be predictable in its timing for the insertion of cocalls at periodic time intervals. However, timing jitter is introduced in the secondary due to two main factors:

Blocking I/O loops:

Blocking I/O loops are used in the secondary thread to wait for data on the UART. The exact duration of these loops are not known beforehand and hence effectively produces an infinite amount of jitter. This jitter is removed by padding the blocking I/O loops so to execute only for one iteration before returning control to the PLBF. In our J1850 implementation, there are total of 9 blocking I/O loops. The amount of padding inserted varied from one PLBF to the other. When the host is integrated with send PLBF, the host executes for 24 cycles before returning back control. Hence the total padding introduced is 162 cycles. On the other hand, when integrated with receive PLBF, the host executes for 100 cycles before returning control. Hence in this case, the total padding introduced is 846 cycles.

Predicate Nodes:

The presence of predicate nodes introduces timing jitter within the host thread especially when the “true” and “false” branches are not of the same length. The jitter introduced increases with the cocall duration. In order to eliminate this jitter, the true and false paths of the secondary are padded to last for the same time. This simplifies the process of inserting cocalls. Again the amount of padding required to eliminate the jitter varies for the send and receive PLBF. When integrated with the send PLBF, the padding required to eliminate jitter is 176 cycles whereas with the receive PLBF, the padding required to eliminate jitter is 772 cycles.

4.4 Integration

4.4.1 Preparations

Before the primary and secondary threads can be integrated, they have to be modified to prepare them for integration. The modifications required are somewhat similar for the two threads.

4.4.1.1 Primary thread Modifications

The modification of the primary thread is generally in the form of padding. Padding is required in the message and bit level functions to eliminate timing jitter resulting from predicate nodes and to ensure correct timing of the bits. The type of padding done is different for message and bit level padding.

Message Level Padding: Padding is required in the message level functions to ensure that the calls to the bit level functions occur at multiples of the bit time of the protocol i.e. multiple of T_{Bit}. This removes any timing jitter between the calls to the send bit functions. This equalizes the time between send bit calls to the worst-case time between calls to the bit level functions. In J1850, the worst-case time between send bit calls is 45 cycles. Hence all code between send bit calls is made to last for 45 cycles by padding. Similarly, for receive message function, the worst-case time between receive bit calls is 44 cycles.

Bit Level Padding: Padding is required in bit level functions again to remove jitter. Jitter may be introduced in bit level functions due to the presence of predicate nodes. The timing invariability introduced by these nodes is removed by padding. In J1850, jitter is introduced in send bit calls due to CRC computation. This is removed by padding.

4.4.1.2 Secondary Thread:

Before integrating, the host interface thread also needs to be modified to make it ready for integration. The modification carried out again is in the form of padding. This is explained in more detail in section 2.4.3. Padding inserted in the secondary thread if of three types:

Predicate Node Padding:

Predicate node padding is required to ensure that the time left after the last cocall insertion within both the branches of the predicate node are the same. This ensures that no matter which path is taken the cocalls are always inserted at the right places. This is of course true only when the predicate node duration is greater than the time to go for inserting a cocall. If not, the predicate node is padded to make the two branches last for the same duration, as explained in section 2.4.3. For J1850, the total predicate padding inserted is 311 cycles for the receive functionality and 69 cycles for send.

Loop Padding:

In order to ensure that each of the loops within the secondary thread has at least once cocall inserted within it, loop padding has to be inserted. The total loop padding inserted is 413 cycles for the receive functionality and 112 cycles for send.

Blocking I/O loop Padding:

Blocking I/O loops are used to continuously wait on a port (by looping) until data arrives on that port. Since the wait time is not known beforehand, this produces effectively infinite jitter. To remove this jitter the blocking I/O loop is padded to execute for just one iteration and return control back to the PLBF. Padding is required both before and after the cocall. The padding before the cocall is required to wait until a cocall period before returning control back to the PLBF while the padding after the cocall is required to equalize the time (after the cocall) to be the same as the number of cycles executed before entering the blocking I/O loop.

For J1850 send bit PLBF, the padding added due to blocking I/O loops is 162 cycles while for the J1850 receive bit the padding added due to blocking I/O loops is 846 cycles.

4.4.2 Transformations

4.4.2.1 Control Flow Reconciliation

After the primary and secondary thread have been prepared for integration, some control flow changes may be required to be done on both the threads for actual integration. The types of changes required are different for both the threads.

4.4.2.1.1 Primary Thread:

The primary thread may be transformed depending on how the idle time is fragmented. The idle time in the bit level functions is highly fragmented and the idle time chunks are too small to be utilized for any other work. Hence to recover more idle time, the functions are modified by removing the intervening guest code and saving it for later replication into the secondary thread. This is depicted in figure 4.13.

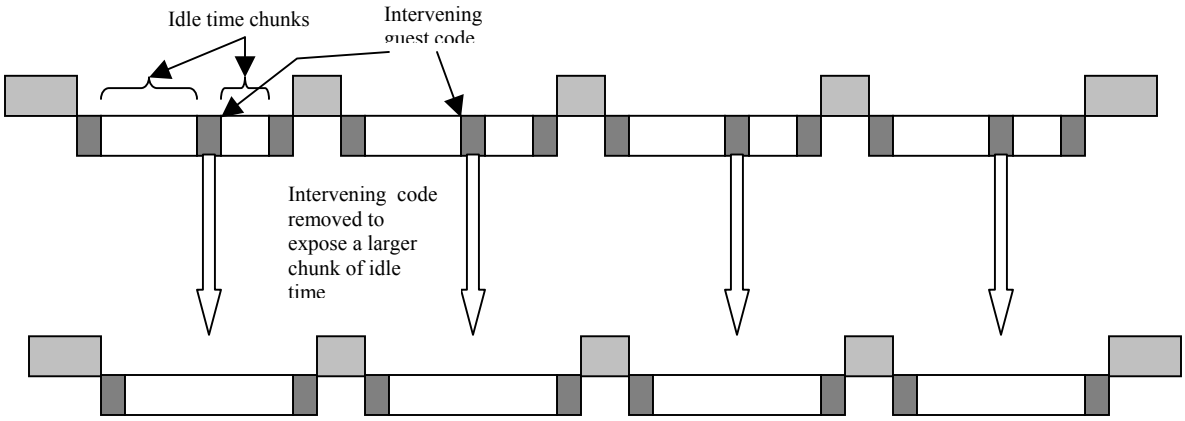


Figure 4-11 Intervening Guest Code Removal

In J1850 send bit function, though the idle time is fragmented, all idle time chunks except one are too small to be used. There are two idle time chunks lasting for 74 cycles and 328 cycles. For an idle time slot to be useful, it should be at least as long as one cocall duration and the cocall duration for J1850 lasts for 153 cycles. Since the 74 cycle idle time chunk is not long enough to execute a cocall, it is not used for integration. Hence intervening guest code removal is not possible for the send bit functionality of J1850. In receive bit function however, the idle time is again fragmented due to multiple sampling of the bus. But in contrast to the send bit function, there are several idle time chunks whose durations are greater than the cocall length. These idle

time chunks last for 158 cycles, 36 cycles and 239 cycles respectively. Since the 158 cycle and 239 cycle idle time chunks are greater than the cocall length of 153 cycles, they can be used for inserting cocalls. The guest code between these two idle time slots could be removed, exposing a larger idle time period.

4.4.2.1.2 Secondary Thread:

There are two types of control flow transformations required on the secondary thread for J1850. These are node replication and loop splitting.

Node Replication:

Node replication is required when there is some intervening guest code that has to be inserted within the secondary thread. As described in the transformations for the primary thread, when the idle time in primary thread is fragmented and many of these chunks have idle time durations greater than the cocall period, then the guest code between the first and last such idle chunks is removed. This is now inserted at several places within the secondary thread such that they would execute at almost the same time as they would have executed in the primary thread. In J1850 intervening guest code insertion is required only for the receive functionality since there is none for the send functionality. The intervening guest nodes are the bus sampling instruction (second and third samples) code nodes. The number of times these have to be inserted within the secondary depends on the number of cocalls being inserted into the secondary.

Note 1 :

During Intervening guest code insertion, the guest code is put at the nearest instant possible to its desired location. Sometimes the intervening guest cannot be put at the exact location where it needs to be. This is when:

a) The intervening guest code location coincide with a control flow changing instruction e.g branch. Consider the following code snippet:

```
brne _PC_ + 2   <----  Intervening code insertion location
rjmp .Lcodelabel
```

In the above code if the intervening guest is inserted in the location shown, the control flow of the program is disturbed. Hence the guest is inserted before the code.

b) The guest location is in the middle of an instruction whose execution time is more than a cycle. Suppose the guest needs to be inserted after a cycle but the next instruction happens to be a store which executes for 2 cycles. In this case, the guest is inserted before the multi-cycle instruction

4.4.2.1.3 Loop Splitting:

Loop splitting is required for Blocking I/O loops in the secondary thread. Blocking I/O loops are used to continuously poll on a port until data arrives at that port. Since this causes timing variability with respect to insertion of cocalls, the loops are modified such that they execute just

one iteration before executing a cocall back to the PLBF. Since the blocking I/O loops do nothing but wait for data, padding has to be inserted to extend its duration until a cocall period.

4.4.2.2 Data Flow Reconciliation

Since integration results in the primary and secondary threads co executing, strict data flow independence has to be maintained between the two threads. The easiest and less costly approach (with respect to no of cycles required) is to partition the register file and assign dedicated registers to the primary and secondary thread. However this is not possible when the threads use almost all the registers for their work. This can be rectified by either of the two following approaches:

a) Push/ Pop strategy : This is the most simple strategy that could be employed to maintain data flow independence. The primary and the secondary threads can have their own stacks and when switching from one thread to the other, all the registers are saved and loaded with the values of the other thread. However the main disadvantage with this approach is the number of cycles required to perform the save/restore operation.
b) Register Reallocation: This is slightly more involved than the previous one. The strategy involves determining the registers being used at the time of switching in one thread and reallocating the same registers with unused ones in the other thread.

For J1850, the Push / Pop strategy is used for data – flow independence. 28 registers of the AVR Atmega03 are saved before switching to the other and restored upon receiving control back.

4.4.2.3 Cocall Insertion

This is the final step in the integration process. Cocalls are used to switch control back and forth between the PLBF of the primary thread and the secondary thread. This is illustrated as shown in the figure 4-14.

As seen from the figure, during an idle time slot of the PLBF two cocalls are inserted : one at the start of the idle time slot in the PLBF and the other at the end of the idle time slot in the secondary. For efficient utilization of the idle time, the cocalls have to be short so as to maximize the secondary thread execution time. In the J1850 implementation, the cocalls last for 153 cycles and its implementation is as shown :

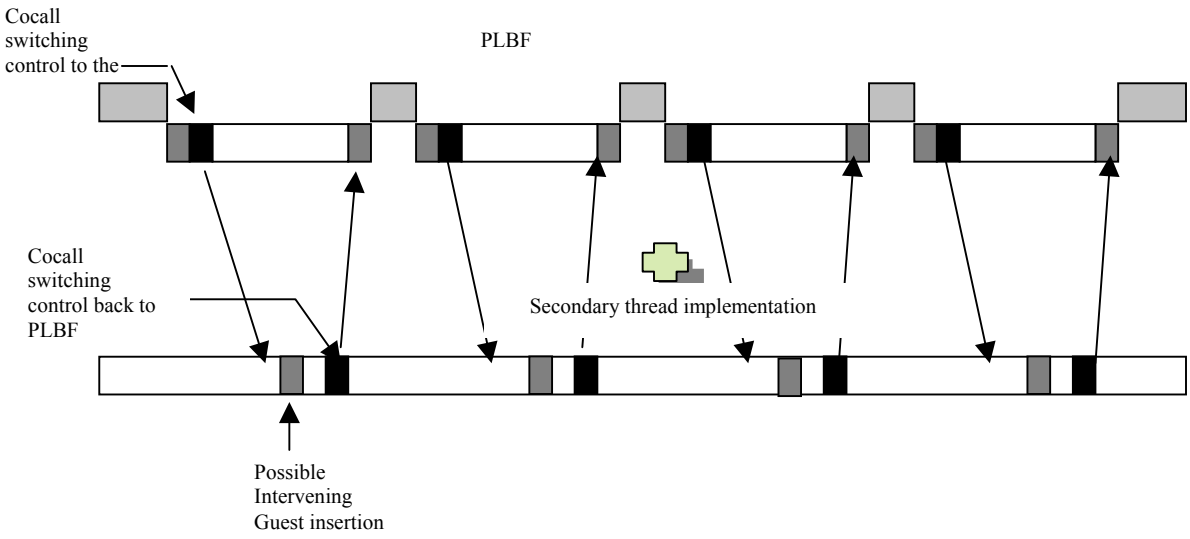


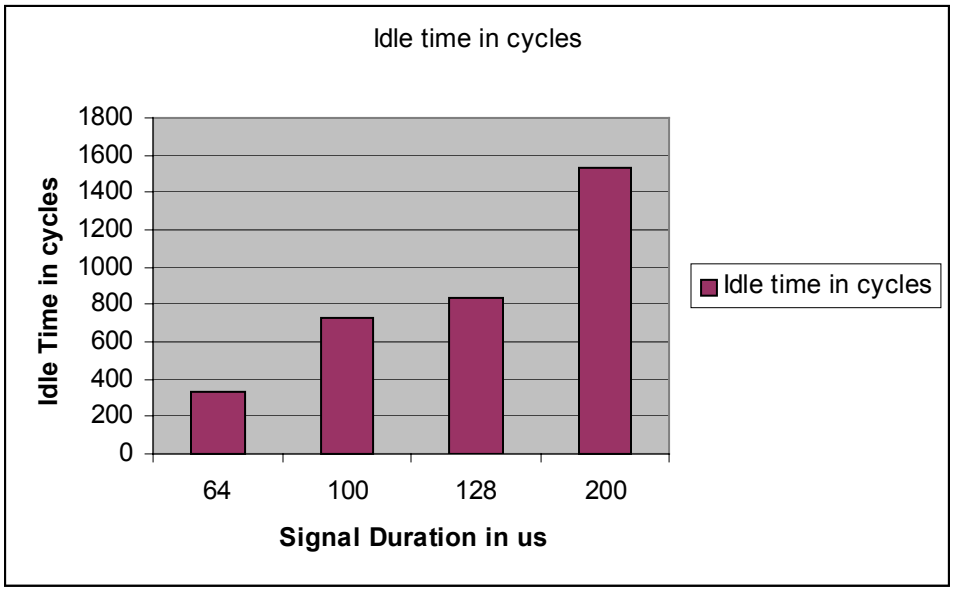
Figure 4-12 Cocall Based switching between Primary and Secondary

```
push r4
push r5
push r6
push r7
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
push r16
push r17
push r18
push r19
push r20
push r21
push r22
push r23
push r24
push r25
push r26
push r27
push r28
push r29
push r30
push r31
```

```
in r2,__SREG__
sts _T2_SREG_,r2
in r2,__SP_L__
sts _T2_SPL_,r2
in r2,__SP_H__
sts _T2_SPH_,r2
lsr r31
ror r30
sts _T2_PCL_,r30
sts _T2_PCH_,r31
lds r2,_T1_SPL_
out __SP_L__,r2
lds r2,_T1_SPH_
out __SP_H__,r2
lds r2,_T1_SREG_
out __SREG__,r2
pop r31
pop r30
pop r29
pop r28
pop r27
pop r26
pop r25
pop r25
pop r23
pop r22
pop r21
pop r20
pop r19
pop r18
pop r17
pop r16
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop r7
pop r6
pop r5
pop r4
lds r30,_T1_PCL_
lds r31,_T1_PCH_
ijmp
```

Since switching between the two threads involves save/restore operation of all the registers, the execution time of the cocalls is significantly high. By register reallocation this duration can be decreased.

When the idle time of a PLBF is too high, then multiple cocall switchings are made to jump back and forth between the PLBF and the secondary thread. This is required especially for the send functionality since the idle time varies from 328 cycles to 1533 cycles for different signal durations on the bus. The distribution of the idle time in the send bit function is shown below:



In order to have a single host thread segment execution between two cocalls, cocalls are inserted within the secondary thread corresponding to the lowest idle time available. For J1850 send, this value is 328 cycles. Since cocalls themselves take 152 cycles to execute, (328 – 2*152) = 24 cycles are available for doing secondary thread work between cocalls. This effectively implies that the host interface thread is called every 64us. Hence the condition for servicing the UART (once every 1.04 ms) is satisfied. For other idle time durations within the PLBF, the secondary thread is called multiple times. If any time remains after doing multiple calls to the secondary, the remaining time is filled up with padding. For example, in J1850 send, for the idle time corresponding to 1533 cycles, the secondary thread is called 4 times. The time remaining after calling the secondary thread i.e 1533 – (328 * 4) = 221 cycles is filled up with padding.
For the J1850 receive functionality, the idle time is constant is equal to 433 cycles. Thus the actual time to do secondary thread work is (433 – 2*152) = 129 cycles. Here 100 cycles are chosen to execute the secondary thread in each segment the remaining time i.e 29 cycles is filled up by padding.

Note 2:

While inserting cocalls within the secondary thread, sometimes the cocalls cannot be inserted at the exact location where it needs to be. This is when:

a) Registers r30 and r31 are being used in the secondary thread. The cocall code uses registers r30 and r31 to perform a switch to the other thread. Since these registers cannot be saved/restored, cocalls are inserted either before or after the code block using r30 and r31. This introduces some timing jitter.

b) Code before the start of a loop and the code at the end of a bounded loop cannot be made equal. This usually results in a jitter of a single cycle.

c) Code at the beginning and end of Blocking I/O loop cannot be made equal. This again results in a jitter of single cycle.

4.4.3 Transformation Methodology

The transformations, described above, are carried out on the primary and secondary threads part by automation and part manually.

4.4.3.1 Automated tasks

A post pass compiler "Thrint" is used to automatically perform the following tasks:

b) Parse the assembly files for the primary and secondary threads and generate their CDG's.
c) Get the timing information on the threads and analyze them.
d) Analyze the primary thread and pad away any timing jitter.
e) If the idle time is fragmented and many of the fragments have duration greater than a cocall duration, the intervening guest code is extracted and saved for later replication into the secondary thread.
f) Analyze the secondary thread and do the transformations. This involves proper padding of the predicate nodes, loop nodes and split nodes where cocalls have to be inserted.
g) Insert the cocalls and the intervening guest code at the proper locations. Thrint does not, however, automatically handle most of the cases mentioned in Note 1 and Note2. It however handles case (b) in Note 1. The algorithm for cocall and intervening guest code analysis and insertion is given in appendix 1.

4.4.3.2 Manual Transformations

The manual transformations required for integration are:

a) Change the cocall and intervening guest location, if necessary, to satisfy conditions mentioned in Note 1 and Note 2.
b) Pad the blocking I/O loops to the correct amount and insert Intervening Guest code in the case of receive functionality.
c) Generate the final integrated assembly file. Though thrint generates the CDG of the final integrated output, it does not generate the corresponding assembly file. This assembly file needs to be generated manually.

5 Results & Analysis

5.1 Simulation Environment:

To simulate the effect of thread integration on J1850 protocol, Atmel's AVR toolkit was used. This toolkit mainly comprises of the AVR studio that is used for simulating the discrete and integrated code. AVR-GCC compiler was used to compile the "C" programs to AVR assembly code. The AVR Studio supports all AVR microcontrollers including Atmega103. It simulates not only the CPU but nearly all the on-chip I/O modules and memory as well as the I/O ports. The J1850 protocol is compiled using AVR-GCC and the resultant assembly code is run through the simulator.

In order to observe the effects of Integration, 3 J850 frames were generated and run through the simulator. The 3 Frames had the following parameters:

a) Maximum Length Frame: Header Length = 3 Data Length = 8 CRC length = 1. This is the longest message that could be sent.

b) Typical Frame: Header Length = 1, Data Length = 7, CRC Length = 1.

c) Minimum Length Frame: Header Length = 1, Data Length = 0, CRC Length = 1. This is the smallest frame that can be sent using the J1850 protocol.

The send routines (discrete and integrate versions) are used encode the messages into frames and put them on the bus and the receive routines (discrete and integrated versions) are used to decode these values put by the send routines. This setup is as shown in figure 5.1.

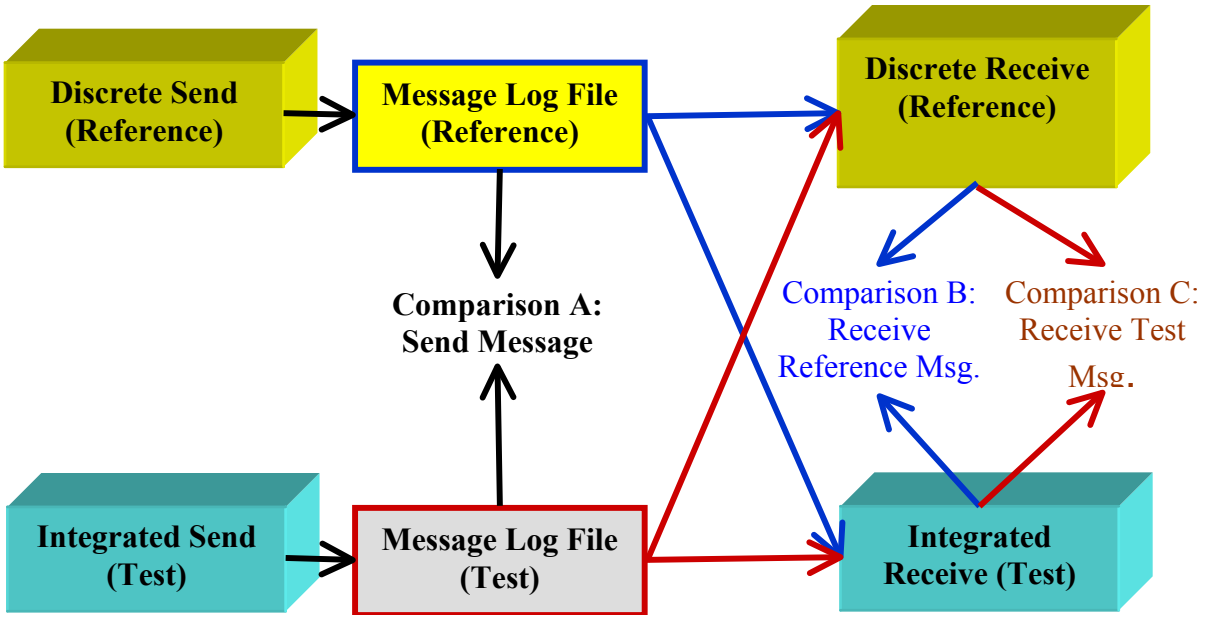


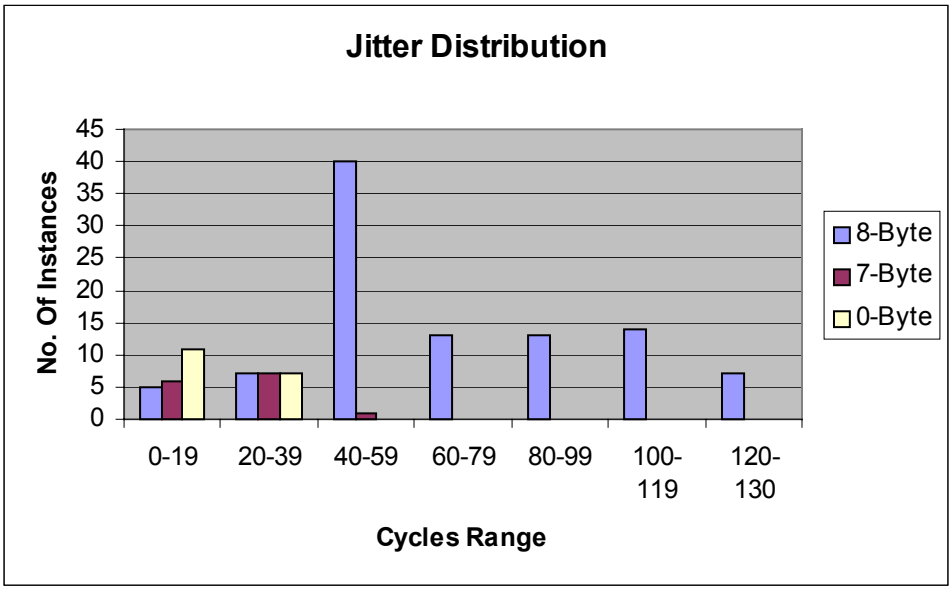
Figure 5-1 Test Setup

The comparisons are used to determine the timing variation or jitter between the samples put out by the discrete padded or "Gold" routines and the integrated routines. . In each of the cases, the messages were correctly recovered and the timing jitter was within tolerable limits.

5.2 Results:

5.2.1 Timing Evaluation:

5.2.1.1 Discrete Vs. Integrated Send Routines:

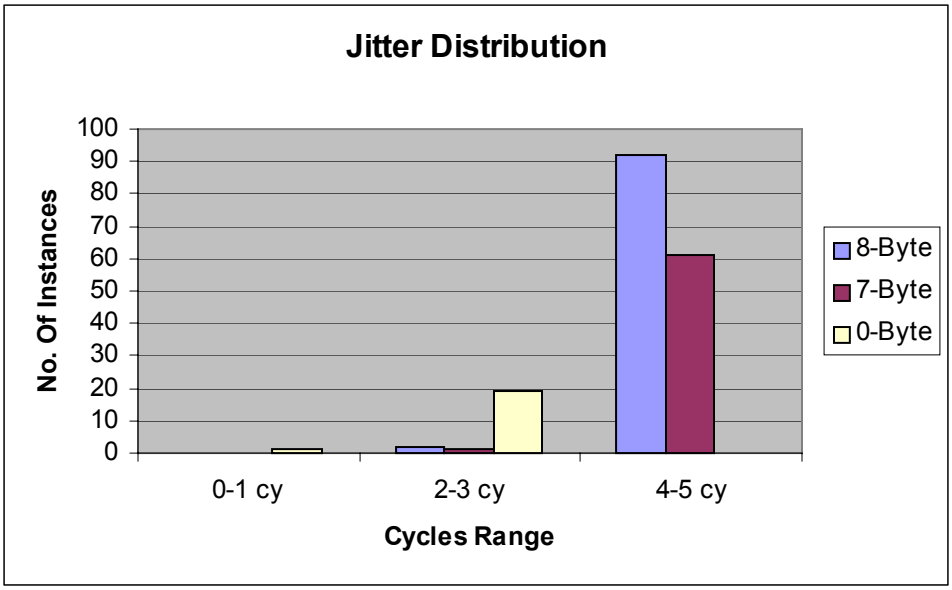


For the messages sent, the jitter introduced is as tabulated as shown:

	8- Byte	7- Byte	0- Byte
Maximum Jitter (in cycles)	125	25	39
Minimum Jitter (in cycles)	5	1	1

As seen from the table, the maximum jitter introduced is around 125 cycles for the 8-byte message. This translates to around 15 µs for 8MHz **Microcontroller** and is a 25% variation for a 64us signal. But this variation is within J1850 specifications. The amount of jitter introduced is also a function of $T_{SegmentIdle}$ or the periodicity of cocall insertion within the host interface thread. The smaller the $T_{SegmentIdle}$, the greater the jitter introduced. Jitter is also dependent on the path of execution within the secondary thread. Some paths produce more jitter while others may not produce any.

5.2.1.2 Discrete Vs. Integrated Receive for Discrete Send Input:

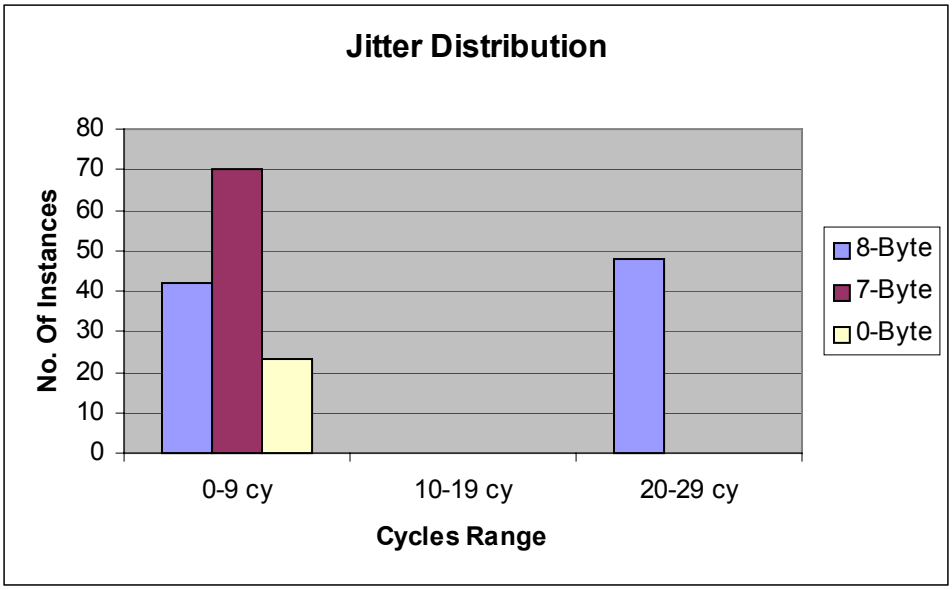


The timing variations between the discrete and integrated versions of the receive routines for discrete send input are tabulated as shown:

	8- Byte	7- Byte	0- Byte
Maximum Jitter (in cycles)	5	4	3
Minimum Jitter (in cycles)	1	1	1

The above values indicate the difference between the sampling instants for the discrete and the integrated receive routines. As seen from the values, jitter between the two versions of the receive routine is quite small. The maximum jitter is around 5 cycles that translates to less than a microsecond variation. This is mainly because the $T_{SegmentIdle}$ is quite large (100) when the host interface thread is integrated with the receive routine.

5.2.1.3 Discrete Vs. Integrated Receive for Integrated Send Input:



The timing variations between the discrete and integrated versions of the receive routines for integrated send input are tabulated as shown:

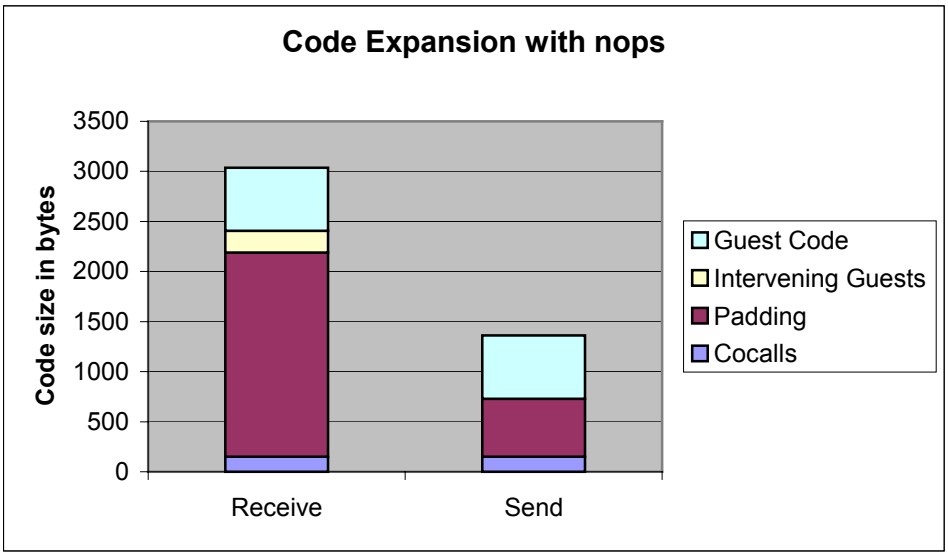
	8- Byte	7- Byte	0- Byte
Maximum Jitter (in cycles)	28	3	3
Minimum Jitter (in cycles)	2	1	1

The maximum jitter is around 28 cycles for the 8-byte message. This is just a 4 microsecond variation between the discrete and integrated versions of the receive routine.

5.2.2 Code Expansion:

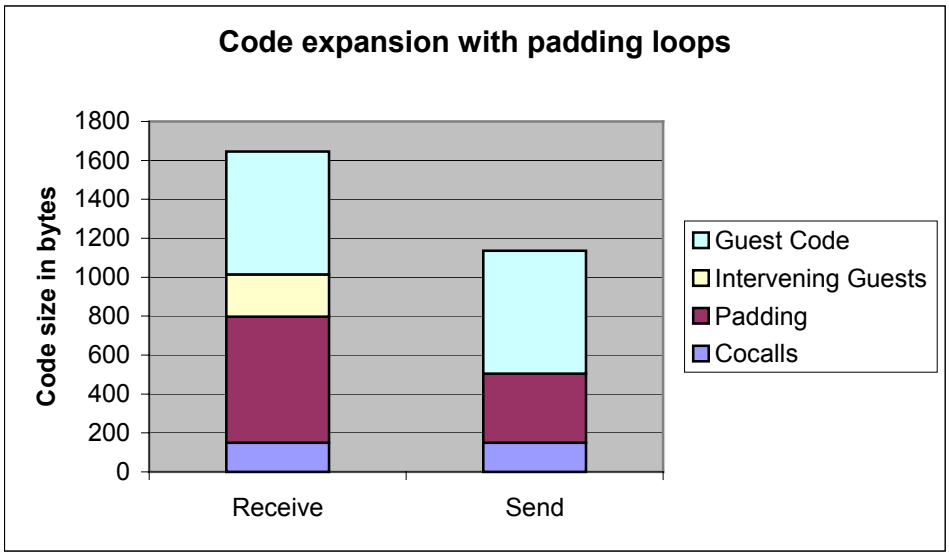
Code expansion is mainly due to the insertion of padding in the threads to regularize the timing. Padding can be done either using Nops or using padding loops.

5.2.2.1 Code Expansion due to Nops:



As seen from the histogram, the code size increases by more than 4 times when the host interface thread is integrated with the receive routine and by around 2 times when integrated with the send routine. Most of the code expansion is due to the padding. When padding is done using Nop's, it accounts for more than 50% of the code expansion.

5.2.2.2 Code Expansion due to Padding Loops:



When padding loops are used for padding rather than Nop's, there is a smaller code size expansion as seen from the histogram. Now the integrated code size increases by less than 50% and within this, padding accounts for less than 50% of the code expansion.

5.2.3 Performance Improvement compared to Interrupt Based approach:

When interrupts are used for context switching, the progress through the secondary or the host interface thread is not as much compared to the "intervening guest code removal" approach due to the following reasons:

a) Context switching needs to be done during each idle time slot
b) Idle time slots smaller than the context switching times cannot be used
c) The secondary threads needs to be structured as a FSM. This introduces extra overhead due to the initial "state determining" code.

For the two approaches, the secondary thread code executed during the idle time slots of the PLBF's can be expressed quantitatively, based on the following terms:

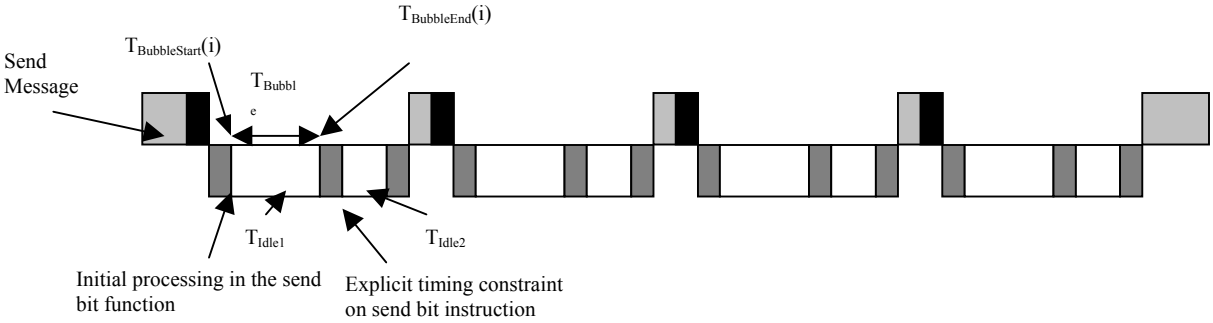


Figure 5-2 Processor Timeline

$T_{BubbleStart}(i)$ = Start of the idle time bubble "i".

$T_{BubbleEnd}(i)$ = End of the idle time bubble "i".

$T_{Bubble}(i)$ = Duration of the idle time bubble "i".

$= T_{BubbleEnd}(i) - T_{BubbleStart}(i)$

T_{CSIGR} = Time taken for a context switch when cocalls are used for context switching.

T_{CSInt} = Time taken for a context switch when interrupts are used for context switching.

$T_{SegmentIdle}$ = Amount of idle time that is actually available for useful secondary thread work between transmission / reception of two bits.

= For the intervening guest code removal approach,

$$T_{SegmentIdle} = \left(\sum_{i=a}^{b} T_{Bubble}(i) \right) - 2 * T_{CSIGR}$$

where

a = min(i) $T_{Bubble}(i) > T_{CS}$ (First Piece of idle time large enough for a co-call)

and

b = max(i) $T_{Bubble}(i) > T_{CS}$ (Last piece of idle time long enough for a co-call)

For the interrupt based switching approach

$$T_{SegmentIdle} = \begin{cases} \sum_{i=a}^{b} (T_{Bubble}(i) - 2 * T_{CSInt}) & (2 * T_{CSInt}) > T_{Bubble}(i) \\ 0 & (2 * T_{CSInt}) <= T_{Bubble}(i) \end{cases}$$

a = first idle time bubble (for fragmented idle time) between transmission and reception of two bits.

b = last idle time bubble between the transmission and reception of two bits.

N_{Bytes} = Total number of bytes sent/ received by the PLBF's.

$N_{Cocalls}$ = Number of cocalls executed between transmission / reception of two bits.

T_{Sec} = Total cycles of secondary thread executed

$= \{ 8 * (N_{Cocalls} * T_{SegmentIdle}) \} * N_{Bytes}$

Secondary thread executed

in 1 byte transmission / reception

For J1850 protocol,

$N_{Cocalls}$ = 1 for send bit (64µs) and receive bit

= 2 for send bit (128µs).

T_{CSIGR} = 146 cycles (refer to section 4.4.2.3)

T_{CSInt} = 151 cycles (refer appendix II)

Based on these values, the progress through the secondary thread or T_{Sec} for the two approaches is as shown in the graphs in figs. 5.1 and 5.2 for the send and receive functionalities. As seen from the graphs, the progress through the secondary thread is much faster using the "Intervening Guest Removal" approach compared to the "Interrupt-based" approach. Within the "Intervening Guest Removal" approach, the progress is much quicker when $N_{Cocalls}$ is higher.

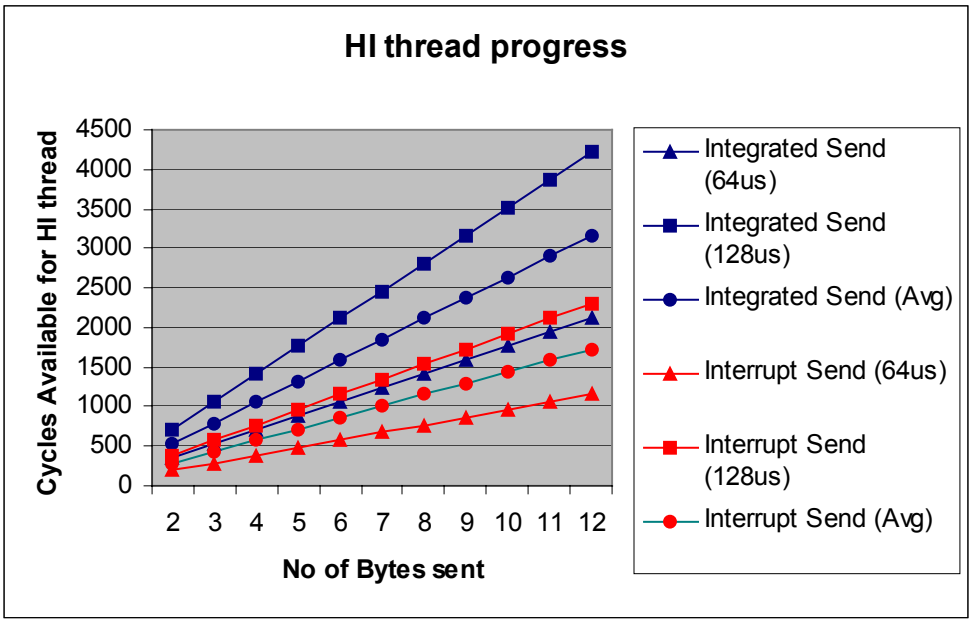


Figure 5-3 Comparison between approaches for Send

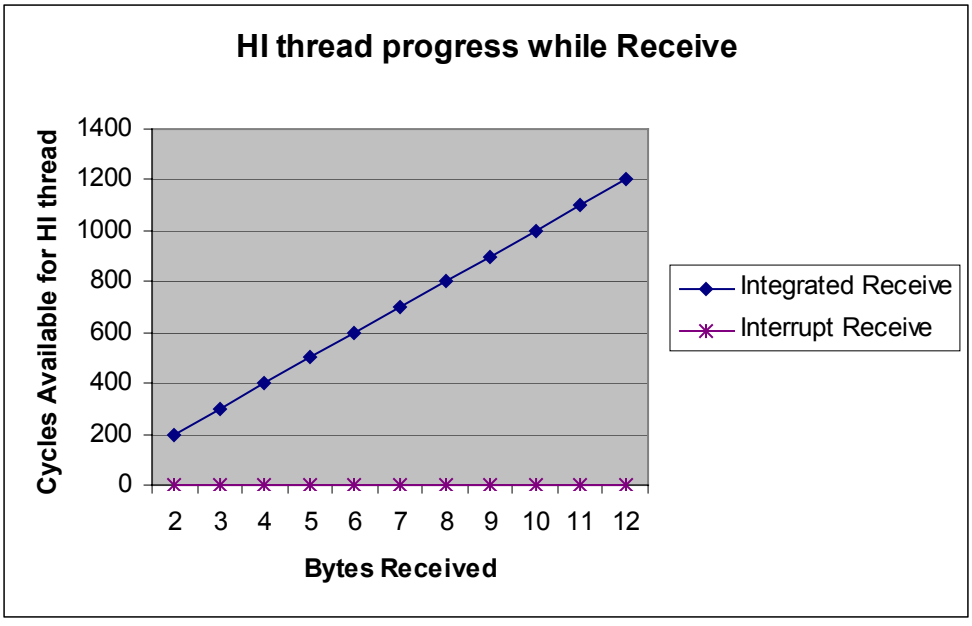


Figure 5-4 Comparison between approached for receive

6 Summary & Future Work

6.1 Summary:

Since hardware implementations of protocol controllers are highly customized to the protocol being implemented, moving functionalities from hardware to software provides a variety of processing options, limited only by the speed of the microcontroller on which the protocol is run. But assuming the processor is fast enough to meet the demands of the protocol, there are direct benefits to moving functionality to software. The unit cost is lower, system is small and weighs less and new and proprietary functions could be easily added even after the design is mature.

Software implementation of protocol controllers usually consists of implementing two threads : a Bus Interface thread responsible for putting bits on the bus and a Host interface thread, responsible for receiving commands from a main controller and communicating them to the bus interface thread. Since the bit rates of the protocols are fixed, there are strict timing constraints on the Bus Interface thread as to when it can put/ receive bits from the bus. This introduces periods of processor inactivity or idle time to be introduced within the thread. Since there may be several such timing constraints while sending or receiving a bit, the idle time is fragmented. The idle time can be better utilized by having the host interface thread execute within this idle time period. This necessitates a context switching mechanism that is responsible for switching between the threads. However in order to meet the real time deadlines and to efficiently utilize the idle time for host interface thread work, the switching overhead of the context switching scheme needs to be minimal. This is especially true if the idle time is fragmented since number of switches per bit is high. Using interrupts does not help since it not only introduces a substantial switching overhead but also requires switching to be done during each of the idle time fragments. Thus the switching overhead is not minimized.

This thesis introduces a scheme for efficiently utilizing the idle time by minimizing the number of context switches required per bit. Cocalls are used to perform the context switching, introducing lesser context-switching overhead compared to interrupts. By using cocalls the host interface thread need not be structured as an FSM that is required when using interrupts. The key idea introduced is the removal of intervening guest code that not only exposes more idle time for secondary thread work but also requires that only two context switches be performed between transmission/ reception of bits. The intervening guest code is inserted at the appropriate instants within the host interface thread. Thus the scheme could be used even to extract fine grain idle time and use it for host interface thread work.

To verify the performance using this approach, a J1850 protocol controller was implemented using both the scheme suggested and the scheme using dedicated padding. The dedicated padding scheme insures that the bits are always sent/received at the correct instants. A comparison between the two schemes showed that the “intervening guest code removal” approach always transmits/ receives bits correctly and introduces a timing jitter that is less than 25 % for transmit and less than 1% for receive. These values occur when the protocol is run on a 8MHz **Microcontroller** and for certain values for cocall periods, as explained in

section 4.4.2.3. This scheme was also compared to the interrupt-based scheme for implementing the protocol, as explained in section 5.2.3. Results showed that the secondary thread advances 45% faster for send compared to the interrupt based scheme. The interrupt-based scheme makes no progress in the secondary thread, while the integrated scheme makes progress.

6.2 Future Work:

Currently the "intervening guest code" removal approach requires separate copies of the host interface thread when integrated with the send and receive functionalities. This introduces considerable code expansion. This could be improved by using a single host interface thread and using "guards" to determine if an intervening guest code or cocalls need to be executed for that particular implementation or not. This is explained in more detail in sections 2.4.4.4 and 2.4.4.5.

The switching mechanism currently requires that all the registers be saved/ restored before a switch is made. This causes the number of cycles required for switching to be high (~153). Improvements could be made in this end by using register reallocation rather than direct save/ restore. The post pass compiler "thrint" could be modified to handle this.

The intervening guest code and the cocalls cannot always be placed at the appropriate timing instants within the host interface thread due to code constraints explained in sections 4.4.2.1.2 and 4.4.2.3. This introduces jitter which manifests itself while sending / receiving bits. A tool could be written that automatically estimates the jitter introduced when a certain path of execution is taken through the host interface thread. This tool could also provide information as to the cocall duration that would minimize the jitter introduced within the host interface thread.

BIBLIOGRAPHY

[1] Oliver, John D. **"Implementing the J1850 Protocol"**, Intel Corporation.

[2] Dean, A. **"Compiling for Concurrency: Planning and Performing Software Thread Integration,"** 23rd IEEE Real-Time Systems Symposium, December 3-5, 2002, Austin, TX.

[3] Dean, A., Shen, J.P. **"System-Level Issues for Software Thread Integration: Guest Triggering and Host Selection,"** 20th IEEE Real-Time Systems Symposium, December 1-3, 1999, Phoenix, Arizona.

[4] Embacher, Martin. **"Replacing Dedicated Protocol Controllers with Code Efficient and Configurable Microcontrollers —Low Speed CAN Network Applications"**, Application Note 1048, National Semiconductor, May 1997.

[5] Glenewinkel, Mark**. "Interfacing the MC68HC705J1A to 9356/9366 EEPROMs"**, Application Note 1241, Motorola semiconductors, 1996.

[6] Goodhue, Greg. **"A software Duplex UART for the 751/752"**, Application Note 446, Philips Semiconductors, June 1993.

[7] Dean, A., Grzybowski, R.R. **"A High-Temperature Embedded Network Interface Using Software Thread Integration,"** Second International Workshop on Compiler and Architecture Support for Embedded Systems (CASES'99) October 1-3, 1999, Washington, D.C.

[8] Dean, A., Shen, J. P. **"Techniques for Software Thread Integration in Real-Time Embedded Systems,"** 19th Real-Time Systems Symposium, Madrid, Spain, December 2-4, 1998.

[9] Dean, A., Shen, J. P. **"Hardware to Software Migration with Real-Time Thread Integration,"** EuroMicro Workshop on Digital System Design, Vasteras, Sweden, August 25-27, 1998.

[10] Dean, A., Shen, J. P. **"Thread Integration for Error Detection and Performance,"** 3rd IEEE International On-Line Testing Workshop, Crete, Greece, 1997, pp. 7-11.

APPENDIX I

Following is the algorithm for inserting cocalls and intervening guest code within the host interface thread.

The variable CocallStepping gives the number of cycles left before the cocall code could be inserted. Initially is set to Cocall Duration.

The Do_Host_Analysis_And_Insert_Cocall is called twice. First, it is called to perform an analysis of the code and perform padding, if necessary. The second call inserts cocalls at the appropriate instants in the code.

Variable Analysis is set to 1 when analysis on the code needs to be done. When set to 0, Cocalls and Intervening guest code are inserted.

Intervening Guest Code insertion is determined by a set of variables, most important of which are the Guests_Duration array and Guest_Index variable. Guests_Index array contains the number of cycles left before an intervening guest code is inserted. Guest_Index keeps track of the Intervening guests. When all the intervening guests are inserted within a cocall period, Guest_Index is reset to zero and the Guests_Comp_For_CocallPeriod is set to One. This is again reset to zero when a cocall is executed, implying the end of a cocall period.

Variables Repeat_Node_For_InterGuest and Repeat_Node_For_HostNode are used to determine whether a particular node needs to be visited more than once. A particular node needs to be visited more than once when during the first visit, the CocallStepping or Guests_Duration[Guest_Index] turns out to be zero. Hence after inserting the cocall, the node needs to be analyzed again for possible cocall insertion or intervening guest code insertion within it.

```
PROCEDURE Do_Host_Analysis_And_Insert_Cocall (int CocallStepping, int Guest_Index)
Begin
        Child = Get Children of Procedure
        Switch(Type of child)
        Begin
                Case Code :
                        If(!Analysis)
                        Begin
                           If(!Not_Exec_InterGuest_Code)
                           Begin
```

```
        If(!Guest_Durations[Guest_Index])
        Begin
           If(!Guests_Comp_For_CocallPeriod)
             Begin
                Insert Intervening Guest code before the current child
                Repeat_Node_For_InterGuest = 1
                Reset Guest Info structures
                If (last guest node)
                Begin
                   Guests_Comp_For_CocallPeriod = 1
                End
             End
        End
        Else if(Code Duration > Guest_Durations[Guest_Index])
        Begin
           If (!Guest_Comp_For_CocallPeriod)
           Begin
              Split Code Node at the Guest Duration
              Insert Intervening Guest Code in front of split node
              Reset Guest Info structures
               If (last guest node)
               Begin
                  Guests_Comp_For_CocallPeriod = 1
               End
        End
        Else                        // If Guests_Duration[] is > Code Duration
        Begin
             If (!Guests_Comp_For_Cocall_Period)
             Begin
                Guests_Duration[Guest_Index] -= Code Duration
             End
        End
     End
End
// Analysis for inserting Cocalls
If (!Not_Exec_HostNode_Code)
Begin
```

```
        If(!CocallStepping)
        Begin
          If(!analysis)
          Begin
            Insert Cocall
          End
          Repeat_Node_For_HostNode = 1
          Repeat_Node_For_InterGuest = 1
          Guests_Comp_For_CocallPeriod = 0
          CocallStepping = CocallDuration
        End

        Else if ((CocallStepping < Node Duration) )
        Begin
          Split the node at CocallStepping
         CocallStepping = CocallDuration
      End
      End
      Else
      Begin
        CocallStepping -= Node Duration
      End
      Break
Case Loop:
        If(Analysis)
        Begin
          If(BlockingIOLoop)
          Begin
        // A Blocking IO loop is considered as an atomic code of duration 4 cycles.
        //Hence if CocallStepping or //Guests_Duration[] is less than 4, then the
        //cocalls or the intervening guests are inserted before the code.

      // The Blocking IO loop lasts for a duration of 3 cycles, for the case when the
      //loop condition is not satisfied.

              If((CocallStepping – 4) < 0)
              Begin
```

```
                    Insert Padding equal to CocallStepping
                    CocallStepping = CocallDuration – 3;
            End
            Else
            Begin
               CocallStepping -= 3;
            End
    End
    Else
       Temp_Cocall_Hold = CocallStepping;
       Cycles_Start_Of_Loop_Before_Cocall = CocallStepping
       Cycles_End_Of_Loop_Before_Cocall =
       Do_Host_Loop_Analysis(CocallStepping)
       Cycles_To_Pad = Cycles_End_Of_Loop_Before_Cocall –
       Cycles_Start_Of_Loop_Before_Cocall

       //  Cycles at the end of loop < Cycles at the beginning. Hence padding added
       //to the end of the loop to make the timing same.

      If(Cycles_To_Pad > 0)
     Begin
        Insert (padding = Cycles_To_Pad) at the end of loop
        Cycles_End_Of_Loop_Before_Cocall = Cycles_To_Pad
     End

// if Cycles_To_Pad is negative, then cycles at the end of the loop is greater than
//those at the beginning. Hence padding is inserted so as to add cocalls and then
//again padding is put in to make the cycles at the end of the loop same as those
//at the beginning.

     Else if (Cycles_To_Pad < 0)
     Begin
            Pad for (duration = Cycles_End_Of_Loop_Before_Cocall +   2) at the
            end of loop.
            Cycles_To_Pad = CocallDuration -
            Cycles_End_Of_Loop_Before_Cocall  +   2
            If (Cycles_To_Pad < 0 )
```

```
            Begin
              Cycles_To_Pad =0
              Cycles_End_Of_Loop_Before_Cocall = Cycles_To_Pad
                Insert (padding = Cycles_To_Pad) at the end of loop
             End
             Else
             Begin
                    Cycles_End_Of_Loop_Before_Cocall = CocallDuration –
                    (Cycles_End_Of_Loop_Before_Cocall + 2)
             End
            CocallStepping = CocallDuration –
            (Cycles_End_Of_Loop_Before_Cocall + 1)

End                         // End of If(Analysis)
Else
Begin                       // Start of  (!Analysis).
   If (Blocking IO Loop)
   Begin
      If(!Not_Exec_InterGuest_Code)
      Begin
/*
Since the Blocking IO loop code has to be executed   atomically, cocalls or
intervening guest code cannot be inserted in between the code. If the intervening
guest code needs to be inserted in between, it is put before the start of the code
thereby inducing a jitter or at most 4 cycles.
*/

            If((!Guests_Duration[Guest_Index] ) OR
            (Guests_Durations[Guest_Index] < 4))
            Begin
               If(!Guests_Comp_For_CocallPeriod)
               Begin
                  Insert Intervening Guest Code before this node
                  Reset Guest Info structures
                  If (last guest node)
                  Begin
                     Guests_Comp_For_CocallPeriod = 1
```

```
                End
              End
            Else
            Begin
              If(!Guests_Comp_For_Cocall_Period)
              Begin
                Guests_Duration[Guest_Index] -= 3
              End
            End
          End

        Else
        Begin
          If(!Not_Exec_HostNode_Code)
          Begin
            If(!CocallStepping)
              Begin
                Insert Cocall Before this node
                Repeat_Node_For_HostNode = 1
                Repeat_Node_For_InterGuest = 1
                Guests_Comp_For_CocallPeriod = 0
                CocallStepping = CocallDuration
              End
              Else
              Begin
                CocallStepping -= 3
              End
            End
        End
End                         // End of If(Blocking IO loop)
Else                        // If (other node types)
Begin
   CocallStepping = Do_Host_Loop_Analysis (CocallStepping)
   CocallStepping += 1    // Loop back criteria is not satisfied.
// Hence branch takes one cycle to jump
// out
   If (!Guests_Comp_For_Cocall_Period)
```

```
            Begin
                Guests_Durations[Guest_Index] += 1;
            End
        End                         // End of If(!Analysis)
        Break                       // End of Loop Node handling code
Case Proc:
        For each Child
        Begin
                Repeat_Node_Visit = FALSE;
                CocallStepping =
                Do_Host_Analysis_And_Insert_Cocall(CocallStepping)
                If((!Repeat_Node_For_InterGuest) AND
                (Repeat_Node_For_HostNode))
                Begin
                   Not_Exec_InterGuest_Code = 1
                End
                Else
                Begin
                   Not_Exec_InterGuest_Code =0
                End
                If((Repeat_Node_For_InterGuest) AND
                (!Repeat_Node_For_HostNode))
                Begin
                   Not_Exec_HostNode_Code = 1
                End
                Else
                Begin
                   Not_Exec_HostNode_Code =0
                End
                If((!Repeat_Node_For_InterGuest) AND
                (!Repeat_Node_For_HostNode))
                Begin
                   Advance child pointer
                End

        Break
Case Pred:
```

```
If((!Analysis) AND (!Guests_Duration[Guest_Index]))
Begin
   If(!Guests_Comp_For_CocallPeriod)
              Begin
                 Insert the Intervening Guest Code before this node
                 Reset Guest Info structures
                 If (last guest node)
                 Begin
                    Guests_Comp_For_CocallPeriod = 1
                End
                Repeat_Node_For_InterGuest = 1
                Repeat_Node_For_HostNode = 1
              End
            End                              // End of (!Guests_Durations[Guest_Index]
            If((!Analysis) AND (Guests_Duration[Guest_Index] < 2))
            Begin
              /*
                    Since the predicate node takes a maximum of 2 cycles to jump
                    to the appropriate node, the intervening code is put ahead of
                    the predicate node introducing a maximum jitter of 2 cycles.
             */
              if(!Guests_Comp_For_Cocall_Period)
              Begin
                 Insert the Intervening Guest Code before this node
                 Reset Guest Info structures
                 If (last guest node)
                 Begin
                    Guests_Comp_For_CocallPeriod = 1
                 End
                Repeat_Node_For_InterGuest = 1
                Repeat_Node_For_HostNode = 1
              End
            End
            Else if(!CocallStepping)
            Begin
              If(!Analysis)
              Begin
```

```
        Insert Cocall
    End
    Repeat_Node_For_InterGuest = 1
    Repeat_Node_For_HostNode = 1
    Guests_Comp_For_CocallPeriod = 0
    CocallStepping = CocallDuration
End
// If CocallStepping is less than 2 cycles, then the cocall is inserted
//before the predicate node introducing a maximum jitter of 1 cycle.
Else if((CocallStepping < Node Duration)
      If(CocallStepping – 2 < 0)
      Begin
          Insert Padding to get CocallStepping = 0
         CocallStepping = CocallDuration
         Repeat_Node_For_HostNode = 1
         Repeat_Node_For_InterGuest = 1
     End
  Else if(! Loop closing predicate)
 Begin
   Temp_CocallStepping = CocallStepping
   Save the Intervening Guest Code State

// If CocallStepping is greater than the true path duration of predicate
//node, then cocalls have to be inserted with the true path.

   If((CocallStepping – 2) > True Path Duration)
   Begin
      CocallStepping -= 2
      If((!Analysis) AND (Guests_Duration[Guest_Index))
      Begin
         Guest_Duration[Guest_Index] -=2
      End
      True_remain =     Do_Host_Pred_Analysis(TV_T,CocallStepping)
   End
   Else
```

```
// If the CocallStepping is less than True path duration, then, a test is
//done on the Intervening guest variables to check whether they need to
//be inserted within the true path.
   Begin
          True_remain = CocallStepping – 2 – True Node Duration
          If((!Analysis) AND (Guests_Duration[Guest_Index] > True
          Path Duration))
          Begin
             If(!Guests_Comp_For_CocallPeriod)
            Begin
               Guests_Duration[Guest_Index] -= 2
               Do_Host_Pred_Analysis(TV_T,CocallStepping)
            End
        End
     End
   CocallStepping = Temp_CocallStepping
   Restore Intervening Guest Code State

// If CocallStepping is greater than the false path duration of predicate
//node, then cocalls have to be inserted with the false path.

   If((CocallStepping – 1) > False Path Duration)
   Begin
      CocallStepping -= 1
      If((!Analysis) AND (Guests_Duration[Guest_Index]))
      Begin
          If(!Guests_Comp_For_CocallPeriod)
          Begin
             Guests_Duration[Guest_Index] -= 1
          End
      End
      False_remain =
      Do_Host_Pred_Analysis(TV_F,Temp_CocallStepping – 1)

   End
   Else
   Begin
```

```
// If the CocallStepping is less than false path duration, then, a test is
//done on the Intervening guest variables to check whether they need to
//be inserted within the false path.

            False_remain = Temp_CocallStepping – 1 – False Node
            Duration
            If((!Analysis) AND (Guests_Duration[Guest_Index] > False
            Path Duration))
            Begin
               If(!Guests_Comp_For_CocallPeriod)
               Begin
                  Guests_Duration[Guest_Index] -= 2
                  Do_Host_Pred_Analysis(TV_F,CocallStepping)
               End
         End
      End
      If (True_remain > False_remain)
      Begin
         Pad = True_remain – False_remain
         Make List of leaf nodes for the True child
         For each leaf node
            Insert padding = pad
      End
      Else
         Pad = False_remain – True_remain
         Make List of leaf nodes for the False child
         For each leaf node
            Insert padding = pad
      End
            CocallStepping = Min(True_remain,False_remain)
   End
   Else
   Begin    // If the predicate is a loop closing predicate
            CocallStepping -= Node Duration
   End
End
Else
```

```
Begin     // If CocallStepping exceeds the node duration
   If(Analysis And !Loop Closing Predicate)
   Begin
      Pad the Jitter in the predicate node
   End
   CocallStepping -= Node Duration
   Temp_CocallStepping = CocallStepping

// Though CocallStepping is greater than the predicate node duration,
//Guests_Duration[] may be less. Hence a check is done and if
//Guests_Duration[] is less, then a further check is made as to whether it is less
//than the true and false path durations. If so, the true and/or false paths are
//traversed and intervening guest code is inserted.

   If(!Analysis)
   Begin
         If (Predicate Node Duration >    Guests_Duration[Guest_Index])
         Begin
            Save Intervening Guest State
            If(Guests_Duration[Guest_Index] < True Path Duration)
            Begin
               If(!Guests_Comp_For_CocallPeriod)
               Begin
                     Guests_Durations[Guest_Index] -= 2
                     Do_Host_Pred_Analysis_For_Guest(TV_T, CocallStepping)
               End
         End
         Restore Intervening Guest Code
         If(Guests_Duration[Guest_Index] < False Path Duration)
         Begin
               If(!Guests_Comp_For_CocallPeriod)
               Begin
                  Guests_Durations[Guest_Index] -= 2
                  Do_Host_Pred_Analysis_For_Guest(TV_F, CocallStepping)
               End
         End
         CocallStepping = Temp_CocallStepping
```

```
                End    // End of If(Predicate Node Duration >
                       //Guests_Duration[Guest_Index])
                Else
                Begin
                   Guests_Duration[Guest_Index] -= Predicate Node Duration
                End


            End
        Break
        Return CocallStepping
    END PROCEDURE
```

// Following are supporting functions required for the loop and predicate nodes. These functions in turn call //Do_Host_Analysis_And_Insert_Cocall for further analysis of the loop or predicate nodes.

```
    PROCEDURE Do_Host_Loop_Analysis (CocallStepping)
    Begin
        For each child of the node
        Begin
            Repeat_Node_For_InterGuest = FALSE
            Repeat_Node_For_HostNode = FALSE
            CocallStepping = Do_Host_Analysis_And_Insert_Cocall(CocallStepping)
            If((!Repeat_Node_For_InterGuest) AND (Repeat_Node_For_HostNode))
            Begin
               Not_Exec_InterGuest_Code = 1
            End
            Else
            Begin
               Not_Exec_InterGuest_Code =0
            End
            If((Repeat_Node_For_InterGuest) AND (!Repeat_Node_For_HostNode))
            Begin
               Not_Exec_HostNode_Code = 1
            End
            Else
            Begin
               Not_Exec_HostNode_Code =0
```

```
                End
                If((!Repeat_Node_For_InterGuest) AND (!Repeat_Node_For_HostNode))
                Begin
                  Advance child pointer
                End

        End

END PROCEDURE

PROCEDURE Do_Host_Pred_Analysis(Truth Value tv, CocallStepping)
Begin
        For each child of node with TV = tv
        Begin
                Repeat_Node_For_InterGuest  = FALSE
                Repeat_Node_For_HostNode = FALSE
                CocallStepping = Do_Host_Analysis_And_Insert_Cocall(CocallStepping)
                If((!Repeat_Node_For_InterGuest) AND (Repeat_Node_For_HostNode))
                Begin
                  Not_Exec_InterGuest_Code = 1
                End
                Else
                Begin
                  Not_Exec_InterGuest_Code =0
                End
                If((Repeat_Node_For_InterGuest) AND (!Repeat_Node_For_HostNode))
                Begin
                  Not_Exec_HostNode_Code = 1
                End
                Else
                Begin
                  Not_Exec_HostNode_Code =0
                End
                If((!Repeat_Node_For_InterGuest) AND (!Repeat_Node_For_HostNode))
                Begin
                  Advance child pointer
                End
```

```
        End
    END PROCEDURE

PROCEDURE Do_Host_Pred_Analysis_For_Guest(Truth Value tv, CocallStepping)
    Begin
        Not_Exec_HostNode_Code = 1
        For each child of node with TV = tv
        Begin
          Repeat_Node_For_InterGuest = FALSE
          Do_Host_Analysis_And_Insert_Cocall(CocallStepping)
          If(!Repeat_Node_For_InterGuest)
          Begin
            Advance Child Pointer
          End
        End
    END PROCEDURE
```

APPENDIX II

Following is the assembly code used for using interrupts as the context switching mechanism. Timer 2 of Atmega103 generates interrupts. The interrupts are caused during the idle time slots of the bus interface thread.

```
< 10 cycles to get into ISR >

/* Save all the registers of the Bus Interface thread */

push r4
push r5
push r6
push r7
push r8
push r9
push r10
push r11
push r12
push r13
push r14
push r15
push r16
push r17
push r18
push r19
push r20
push r21
push r22
push r23
push r24
push r25
push r26
push r27
push r28
push r29
push r30
push r31

/* Save the status register and Stack Pointer */

in r2,__SREG__
sts _T1_SREG_,r2
in r2,__SP_L__
sts _T1_SPL_,r2
in r2,__SP_H__
sts _T1_SPH_,r2

/* Load the values for the other context */

lds r2,_T2_SPL_
out __SP_L__,r2
```

```
        lds r2,_T2_SPH_
        out __SP_H__,r2
        lds r2,_T2_SREG_
        out __SREG__,r2


        pop r31
        pop r30
        pop r29
        pop r28
        pop r27
        pop r26
        pop r25
        pop r25
        pop r23
        pop r22
        pop r21
        pop r20
        pop r19
        pop r18
        pop r17
        pop r16
        pop r15
        pop r14
        pop r13
        pop r12
        pop r11
        pop r10
        pop r9
        pop r8
        pop r7
        pop r6
        pop r5
        pop r4


 /*
 Host Interface thread is structured as a FSM. Switching code is required to decide on the state to jump to
 */
Switching:
        lds r30, state
        clr r31
        mov r16, r30
        inc r16
        sts state, r16
        subi r30, lo8(-(jmp_table))
        subi r31, hi8(-(jmp_table))
        lpm

        < Host Interface code here >

return_path:
        ldi r16,(1<<CTC2)|(1<<CS22)|(1<<CS21)|(1<<CS20)
        out TCCR2,r16 ;                 // Timer clock = system clock/1024
        ldi r16,1<<OCF2
        out TIFR,r16 ; Clear OCF2       //clear pending interrupts
```

```
        ldi r16,1<<OCIE2
        out TIMSK,r16 ;                 //Enable timer output compare interrupt
        ldi r16,X
        out OCR2,r16 ;                  // Set output compare value to X
/* Following code restores context before switching control back to the bus interface thread */

        push r4
        push r5
        push r6
        push r7
        push r8
        push r9
        push r10
        push r11
        push r12
        push r13
        push r14
        push r15
        push r16
        push r17
        push r18
        push r19
        push r20
        push r21
        push r22
        push r23
        push r24
        push r25
        push r26
        push r27
        push r28
        push r29
        push r30
        push r31
        in r2,__SREG__
        sts _T1_SREG_,r2
        in r2,__SP_L__
        sts _T1_SPL_,r2
        in r2,__SP_H__
        sts _T1_SPH_,r2

        lds r2,_T2_SPL_
        out __SP_L__,r2
        lds r2,_T2_SPH_
        out __SP_H__,r2
        lds r2,_T2_SREG_
        out __SREG__,r2


        pop r31
        pop r30
        pop r29
        pop r28
        pop r27
        pop r26
        pop r25
```

```
pop r25
pop r23
pop r22
pop r21
pop r20
pop r19
pop r18
pop r17
pop r16
pop r15
pop r14
pop r13
pop r12
pop r11
pop r10
pop r9
pop r8
pop r7
pop r6
pop r5
pop r4
 reti
```