ABSTRACT

Thirumoolan, Sudhagar, STI Friendly Clock Recovery Techniques for Bit Banged Communication Protocols (Under the direction of Dr. Alexander Dean)

Nowadays embedded communication networks are used in a number of applications. A majority of these networks use a shared medium for communication. Controllers for these shared medium protocols can be implemented in hardware or software. Protocol implementations in software using standard off the shelf microcontrollers have been found to be faster, easier and most cost effective.

In an embedded communication network, it is essential that the protocol controllers are in synchrony with each other. A slight phase difference between the clocks of the protocol controllers can lead to errors in the data propagated between them. If the sender has a slight error in its clock frequency the receivers that are reading a message from the sender need to adapt to the clock of the sender by recovering the clock information from the data on the shared medium. This paper talks about the various existing techniques in hardware and software and compares a few proposed clock recovery techniques which can be used for software protocol implementations using STI.

A good clock recovery technique must adapt to the sender as quickly as possible and maintain very little phase difference with the sender. It must consume very few computational cycles and the code blocks must remain as close together as possible to avoid fragmenting the idle time of the protocol implementation.

In order to facilitate STI, a good primary thread implementation will need to have large chunks of idle time which can be merged with useful code from a secondary thread. It is also required that the primary thread should have fixed or almost fixed execution times for all its functions, in order to be able to recover as much idle time as possible.

The proposed techniques have large chunks of idle time as well as try to catch up with the sender's clock as quickly as possible. Hence they are good candidates for being used in software protocol implementations using STI. However, the limitation of these techniques is the fixed number of cycles up to which they can recover in every bit interval when the receiver's clock is slower than the sender's.

STI Friendly Clock Recovery Techniques for Bit-Banged Communications Protocols

By
Sudhagar Thirumoolan

A thesis submitted in partial fulfillment of the
requirements for the degree of
Masters in Science

Computer Science

North Carolina State University

2004

Approved By ____
Dr. Alexander Dean (Chairperson of Advisory Committee)

____
Dr. Frank Mueller (Co-chair of Advisory Committee)

____
Dr. Vincent Freeh (Member of Advisory Committee)

Program Authorized
To Offer Degree ____

Date ____

Dedication

I would like to dedicate this thesis to my parents. Without their hard work and effort, I would not be here and this Master's program would not have been possible.

Biography

Sudhagar Thirumoolan graduated with a B.E. Degree in Computer Science and Engineering from College of Engineering, Anna University, Chennai, India in May 2001. He joined the Master of Science in Computer Science Degree program at North Carolina State University, Raleigh, North Carolina, U.S.A, in Fall 2001. He continued his graduate research work on this thesis under the guidance of Dr. Alexander Dean, Assistant Professor, Department of Electrical and Computer Engineering. He is currently a software engineer at Togabi Technologies Inc., San Diego, California, U.S.A.

Acknowledgements

I would like to thank my parents, relatives and friends who played a very important part in the successful completion of this degree. I would like to thank my colleagues at Togabi Technologies Inc. for their valuable support through out the course of my graduate program.

I would like to thank Dr. Alexander Dean for having served as my advisor and guiding me through this work, even as I was working on it from the west coast. I would also like to thank Dr. Frank Mueller and Dr. Vincent Freeh for being part of my thesis committee.

Finally, I would like to thank all my friends who have been with me all through this Degree Program.

Table of Contents

List of Figures

List of Tables

1. Introduction

The number of embedded systems that we come across in our everyday life has increased drastically. Cell-Phones, Vending Machines, Gaming Stations, Global Positioning Systems etc. have become a basic part of human life. Over 98% of the computing products sold today are embedded systems. Most of these embedded systems use low speed and low cost microcontrollers. About 74% of the microcontrollers sold are either 8 bit or 16 bit. The average price of an 8 bit microcontroller is about $1.44. A majority of these 8 bit or 16 bit microcontrollers operate at speeds of less than 20 Mhz.

Most of the embedded systems are used as nodes of embedded networks. These embedded communication networks are used in automobiles (CAN, LIN, J1850, TCN), aircrafts (1553, DATAC, ARINC-429), process control and automation (Profibus, SERCOS, Foundation Fieldbus), building automation (BACNet, LON, DALI) and sensor networks. These embedded protocols need high fault and noise tolerance and good real time performance. Hence conventional protocols like Ethernet are not used for embedded networks. The various embedded network protocols are the result of many protocol optimizations made on existing protocols which make them simple and easy to implement in a constrained environment [3].

A protocol controller for an embedded communication network protocol can be implemented in hardware or in software. In hardware implementations generally an ASIC is designed to implement the protocol, where as software implementations are done as firmware for an off the shelf microcontroller. The hardware method is very expensive because of the time it takes to build an ASIC and the cost involved in the production process (about $500,000). Harsh environments under which the chip is required to operate adds another constraint to the ASIC production process. Software design and development on these microcontrollers is much faster when compared to hardware design. Tuning a protocol in software can be done easier and faster than in hardware. Microcontrollers which operate in various harsh environments are already available for low prices. Hence many protocols for embedded networks are implemented in software [6] [7].

Software implementations can be done in different ways using busy wait, using interrupts, using STI, using context switches across multiple threads etc. It has been proved that STI based implementations are better than busy wait and interrupt based implementations [1]. STI based Software Protocol implementations normally have at least two threads. A primary thread handles the basic protocol implementations. A secondary thread takes care of interfacing this protocol with the actual application. The actual firmware is a result of the temporal overlapping of the code from these two threads using STI. The various stages have been explained in [1].

Even though the embedded network protocols are simple, they need to address some major issues and constraints. Since most networks are shared medium protocols the receiver must be in synchrony with the sender in order for the receiver to scan and interpret the message correctly. Even a slight clock skew between the sender and receiver might lead to errors.

Many embedded protocols are shared medium protocols and operate at speeds of about 10kbps to 1Mbps. The microcontrollers targeted for these software implementations normally operate in frequencies of up to 20 Mhz. A very efficient software implementation of the protocol needs to keep the ratio of CPU clock to bus bit rate between 100 and 1000, i.e., the duration of a single bit on the wire should be between 100 and 1000 cycles of the microcontroller. For example, consider a 20 Mhz microcontroller used to implement a protocol operating at 100 kbps. The ratio of the CPU speed to the bus bit rate in this example is 200. Within these 200 cycles, the receiver needs to sample the data from the wire and resynchronize with the sender as soon as possible in order to let the secondary thread of the STI do some useful work. A very good software implementation of the resynchronization technique on this chip needs to complete the sampling and resynchronization in less than 100 cycles and have large idle time blocks which can be used for executing code from a secondary thread.

This paper presents a comparison of various techniques to recover the clock of the sender at the receiver for use in software protocol implementations using STI. The paper is organized as follows. The use of STI in software implementations of embedded protocols is discussed in section 2. Section 3 explains the need for clock recovery or resynchronization techniques. The various resynchronization techniques that are used today are discussed in Section 4. The various proposed software techniques for resynchronization are discussed in Section 5. Section 6 compares the performance of the various proposed techniques. Conclusion is discussed in Section 7.

2. Background

2.1 Software Thread Integration

Embedded systems have different functions which are handled by different threads. When these multiple threads are allowed to execute on a single microprocessor a lot of CPU cycles are wasted in context switches. Software Thread Integration (STI) [1] is a technique, where multiple such threads having different real-time requirements are merged into a single thread by using various compiler techniques. This resulting thread achieves the same functionality of both the original threads and gets rid of the context switching overheads.

HSM is the process of moving functions from dedicated hardware components to real-time software. This helps to improve system cost, size, weight, power, function availability, time to market and field upgrades. The main targets of HSM are embedded real time systems which cannot afford the luxury of a high performance microprocessor. Hence, HSM is a major area which can benefit largely from using STI.

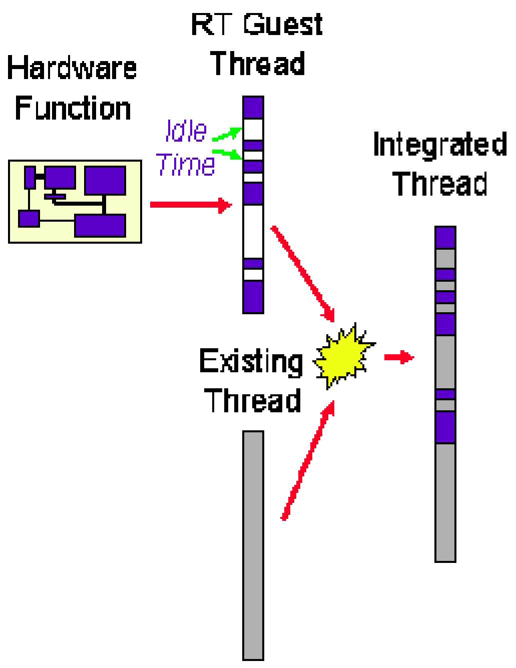

Figure 2.1 STI of Two Threads

Figure 2.1 explains STI of two threads. The real time Guest thread implements a dedicated hardware function and has hard deadlines. It has chunks of idle time between blocks of code which handle the real time constraints. This idle time is normally filled with NOP. The Host thread or Existing thread is a non real time thread. The idle time in the Guest thread can be used to execute code from the Host thread. Using various compiler techniques, blocks of the Host thread code can be inserted into those Guest thread idle time blocks which satisfy the real time constraints of that Host thread block.

STI is performed as follows. At first, the control dependence graphs of the two threads are generated. Then, a Static Timing Analysis (STA) is performed on these control dependence graphs. This analysis gives us the various blocks of code in the two threads and their timing requirements. Using this information, the position of the Guest thread at which the various blocks of code of the Host thread can be inserted is calculated. This however requires register reallocation in order to prevent the two threads from accessing the same register. A special post pass compiler called *thrint* has been developed for achieving STI. *Thrint* reads assembly code, performs control-flow, data-flow and static timing analysis, and integrates threads.

STI implementations have been proven to be more efficient than context switches and interrupt based implementations especially in recovering idle time from real-time guest threads performing periodic low-level (e.g. MAC and data link layer) network communication and video refresh functions. This makes STI a very good candidate for implementing software protocol controllers which replace dedicated hardware chips. An example of such an application is given in [2] in which high temperature (185-225 ˚C) CAN network interface is implemented in software and integrated with buffer management code for the CAN protocol. This system has been built and tested at up to a temperature of 225 ˚C.

2.2 Bit Banged Communication Protocols using STI

STI concepts are employed in [3] for implementing embedded network protocols in software. Examples of embedded networks using such protocols like J1850, CAN, 1553, etc., can be extensively found in cars, conveyor belts etc. These protocols are referred to as Bit-Banged Communication Protocols (BBCP) as the bits are transmitted or received one at a time.

Figure 2.2 explains the generic architecture of software protocol controllers [2][3][5]. Any software implementation of BBCP has two threads. The primary thread or the guest thread implements the main protocol functionality and has critical timing requirements. The secondary thread or host thread interfaces the primary thread with the application. Using STI these two threads can be merged together into a single thread thereby avoiding unwanted context switching overheads.

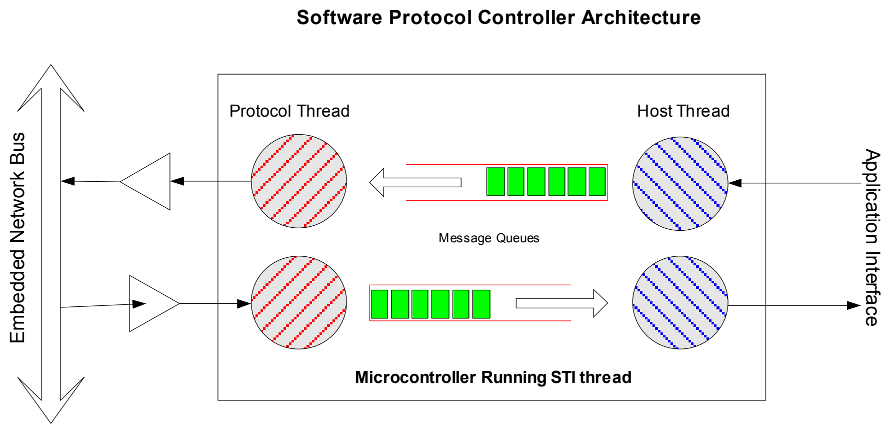

Figure 2.2 Software Protocol Controller Architecture

Any software implementation of a protocol has three layers. The three layers are as shown below in Figure 2.3.

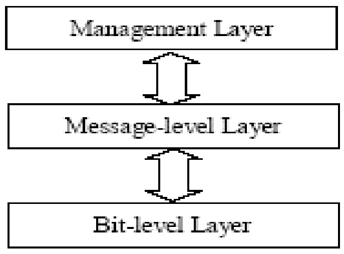

Figure 2.3 Various Layers of a protocol

2.2.1 Executive or management layer:

This layer is responsible to either send to or receive messages from the bus. As shown in Figure 2.3, any request to transmit or receive is passed on to its lower layer, i.e., the messaging layer. This layer is normally implemented as part of the secondary thread or the host thread. This layer is also responsible for interfacing with the application using this protocol controller.

2.2.2 Message layer:

This layer is responsible for all the encoding/decoding schemes of the protocol and passes the messages to the corresponding layers. Additional functions of this layer include CRC calculation and verification and keeping track of resynchronization information across bits. This layer is implemented as part of the primary thread or the guest thread.

2.2.3 Bit layer:

This is the bottom-most layer of the protocol stack and implements the actual physical layer of the protocol. The functions that run at this layer are real time functions, as they write bits to or read bits from the bus at the exact instant. This function is invoked by the message layer for every bit in the message to be transmitted. Additionally this layer also performs functions such as bit stuffing, CRC/parity generation/check and clock resynchronization and recovery at the bit level. This layer is also implemented as part of the primary thread or the guest thread.

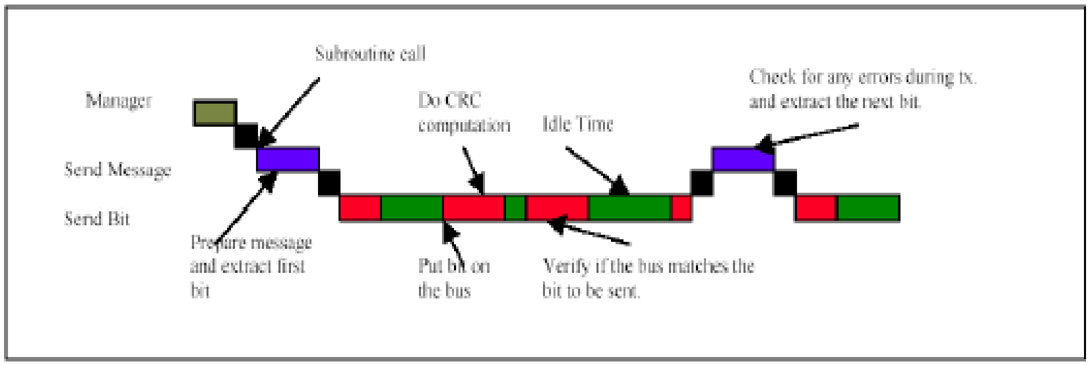
Subroutine call
Manager
Do CRC computation
Idle Time
Check for any errors during tx. and extract the next bit.
Send Message
Send Bit
Prepare message and extract first bit
Put bit on the bus
Verify if the bus matches the bit to be sent.

Figure 2.4 Timeline of the various layers

As it can be seen from figure 2.4, there is coarse grain idle time at the various layers of the protocol implementation. This idle time would be wasted by busy wait or by interrupt processing. The idle times at the bit layer are recovered by integrating with the upper management layer function from the host thread.

A concept called co-call was introduced to switch between these two threads without performing an actual context switch. Co-calls are similar to subroutine calls but it is used to transfer control between two processes. A co-call is effectively a call and return instruction combined into one operation. As far as the calling process is concerned the operation is equivalent to a procedure call. As far as the called process is concerned the co-call operation is equivalent to a return operation. When this called process executes another co-call the first process returns from its co-call and resumes execution from that point. If the two processes execute a sequence of mutual co-calls, control will transfer between the two processes as shown in figure 2.5.

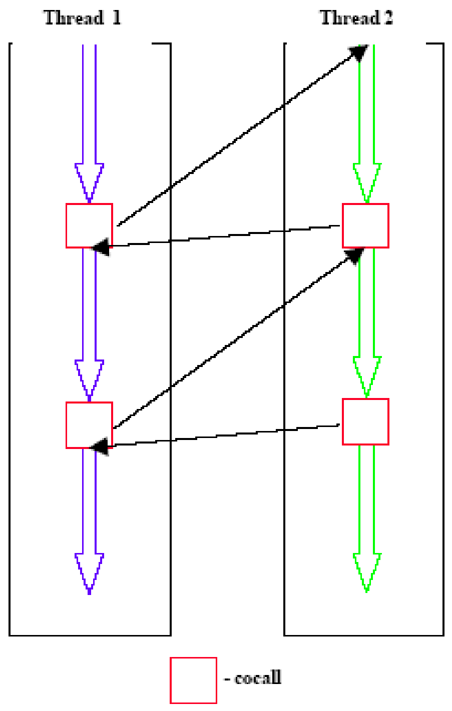

Figure 2.5 Co-calls between two threads

2.3 Asynchronous Software Thread Integration

Co-calls consume considerable number of processor cycles. In order to be able to recover an idle time block, the duration of the block should be large enough to execute two co-calls and accomplish some useful work. If the idle time duration is not large enough the idle block cannot be recovered. In cases where two such idle durations sandwich a small block of real time code, the idle times can be merged and recovered by moving the small block of code to be a part of the host thread. This part of code is introduced into the secondary thread such that it executes at the exact time interval it is supposed to as part of the primary thread. This technique is shown in Figure 2.6 and is called Asynchronous Software

Thread Integration (ASTI) [4]. Thus the idle time duration is made to span larger durations during which the secondary thread can be executed using co-calls.

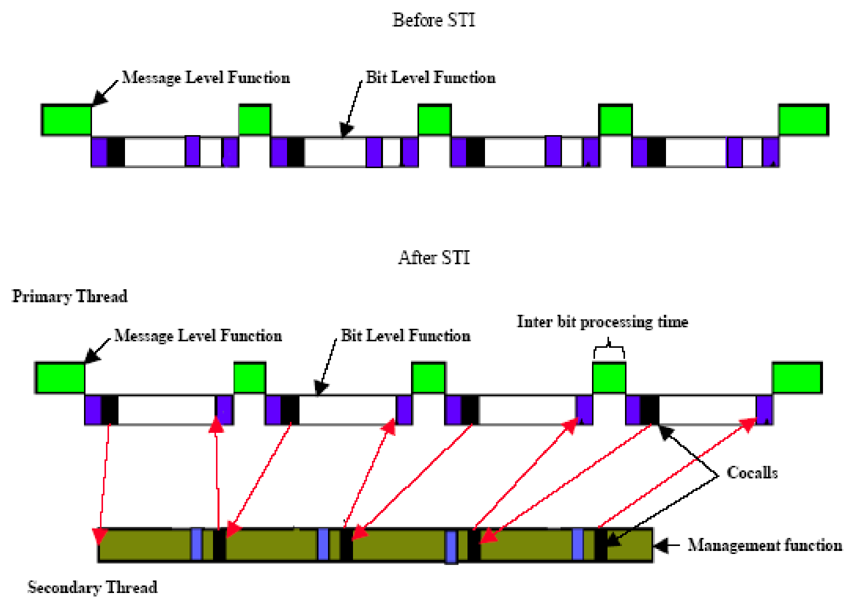

Figure 2.6 ASTI of two threads

While performing STI/ASTI the bit level functions are modified such that they have a fixed execution time. The message level functions are also padded such that they have a fixed duration and that they invoke the bit level functions at fixed intervals. The secondary thread, which contains the management function, is also padded to remove jitter. Thus the resulting threads are as shown in figure 2.6. More details about the various stages of integration are given in [4].

3. Need for Clock Recovery

A fundamental issue in serial data transmission is determining when to sample the data line. An explicitly clocked system provides easily decoded clocking information to control the sampling of the data line. However, this clocking information requires extra wires (e.g. a specific clock line, as in I2C) or extra bandwidth (e.g. Manchester encoding as in Ethernet, Manchester encoding sends a 1 as 10, and a 0 as 01, ensuring a transition in the middle of a bit, which doubles the bandwidth needed).

Many communication systems (whether wired or wireless) cannot afford this overhead and instead use an implicitly timed (or *self-synchronizing*) scheme with NRZ data (non-return to zero where a 1 is sent as a 1, a 0 is sent as a 0). Here, the timing information is defined beforehand (e.g., each data bit lasts 1 microsecond) and is used to determine when to sample the incoming data line to extract bits.

One problem with this implicitly clocked approach is that, frequency errors between the clocks of the transmitter and receiver can lead to sampling the data line too early or too late. Most message-oriented communication protocols use packets that are long enough, that a slight frequency error will result in the wrong bit being sampled by the end of the packet. For example, if the clocks of the sender and the receiver differ by 1% in frequency, then the receiver will sample the wrong bit starting with the 50th in the packet.

A solution to this problem is to recover the clocking information by observing that the incoming data line only changes state between bits. A receiver can sample the data line at some multiple of the bit rate and adjust the frequency (or phase) of its clock to ensure that the data line is sampled halfway between data line transitions.

This is straightforward in hardware implementations unlike software implementations. The duration of 1 bit on the wire in terms of the number of cycles of the microcontroller is in the order of 100 to 1000. This bit duration is the time available for the sampling the bit, resynchronizing the clock and executing a portion of the secondary thread of the protocol implementation.

4. Existing Techniques

4.1 Over-sampling

A number of samples are taken all through a configured bit period. Majority voting is used to calculate the current bit value. Based on the position of the transition among the number of samples, the next bit sampling is done. This technique is not very complex, but it requires frequent sampling of the medium.

The disadvantage to the technique is that it introduces quantization jitter, which is equal to one sampling interval. The advantage is that it can be implemented to include neighboring bit samples also for voting and readjusting the clock. Many protocol implementations use this technique especially in hardware [9].

4.2 Tracking

In this technique generally a Phase Locked Loop or a Delay Locked Loop is used which tries to do the Clock and Data Recovery by adjusting the phase of the receiver clock to match the data that comes on the network [8]. The PLL or DLL basically tries to align the sampling clock for phase shifts, which happen due to propagation delays. Generally, this technique involves sampling the network twice, first at the middle of the data bit and the next at the middle of the actual transition. It tries to synchronize itself so that it does not miss the transition, thus the recovery is achieved.

A known bit pattern is transmitted by the sender at the beginning. The receiver calibrates the delay from the samples received. This calibrated delay is used to avoid jitters and hence to sample the data on the medium correctly. This method avoids quantization jitters completely. However it requires custom hardware circuits.

4.3 Explicit Clock encoding

Some protocols solve the clock problem at a totally different plane by explicitly sending the clock synchronization information [14] [25] [32] [33]. This can be done in one of two ways. A straightforward implementation is to have a clock line separate from the data line to transmit clock signal. The additional clock line makes the sender and receiver use the same clock for writing and reading data. I2C, SPI, TWI, etc., are examples of protocols which use this technique. An alternate implementation is to encode clock information as part of the data by using some mechanism like Manchester encoding to transmit data. This scheme however consumes extra bandwidth. Any protocol which uses these schemes is out of the scope of discussion of this paper.

4.4 Protocol Definition

Some protocols are defined in such a way that, the clocking information is not explicitly encoded [15] [17] [24] [26] [27] [28] [29]. Rather the protocol data format is defined such that, the receiver does not get skewed with respect to the sender's clock. For example, the LIN protocol solves this clock recovery problem at the receiver by specifying that the data header of every message has a fixed pattern 10101010, which is used by the receiver to observe the clock of the sender and adjust accordingly so that the message can be read correctly. CAN is another example of such a protocol. According to the CAN protocol specifications, the bit duration is sub-divided into many smaller units called quanta. The sampling of data has to be done during a particular set of quanta and resynchronization has to be done during another particular set of quanta of the bit interval. If there is no transition in the bus for a long time, then the sender and receiver can go out of synchronization. In order to avoid this, a mechanism called bit-stuffing is built into CAN and many other protocols. Whenever the bus has 5 continuous bit-durations of no transition, then a transition is introduced into the medium for the sixth bit-duration. Normal data transmission is resumed after the transition. This mechanism ensures that there is a transition in the bus at least once every 5 bit-durations. The receivers account for this extra bit and ignore it to avoid errors while receiving the message.

5. Proposed Software Techniques

The proposed techniques assume that the nodes in the network have a certain error in their clock frequency and that this error is constant for the duration during which the message transmission/reception occurs (i.e. the rate of change of error is very small or nil). It is also assumed that the processor executes the instructions at a fixed rate. All techniques are slight variations based on the over-sampling technique. Each technique functionally contains two different threads. The primary thread implements the communication protocol and the secondary thread implements an interface with the rest of the application or system. These two threads can be temporally overlapped into a single thread using STI techniques based on their timing constraints. The primary thread again has two layers as discussed in section 2.2. The resynchronization portion of the code is implemented partly in the message layer and partly in the bit layer depending on the type of technique.

Each software technique in turn contains two parts, the first part deals with the transmission of a bit on the wire and the second part deals with receiving a bit. The transmission part is simple and straight forward. The receiving part is the part where clock recovery and resynchronization has to be done. This part again contains two segments, first is data recovery, the calculation of the bit value on the wire and the second is the clock recovery and resynchronization.

The calculation of the bit value on the wire is done as follows. A number of samples are taken either at the middle of the bit interval or spread over the entire duration of the bit interval. A majority voting is done on these samples that are taken and the resultant value is output as the sampled bit value.

Transition detection and clock recovery is done in two segments. One is detecting the error and the other piece is introducing an appropriate correction. A target window during which a transition can possibly happen is calculated. This transition window normally spans from a few cycles before the end of one bit interval into a few cycles after the start of the next bit interval. The receiver knows from the configured clock value when a transition must happen if at all it happens. The number of cycles between when a transition occurs and when it is supposed to occur gives the amount of error. A correction factor is then calculated and introduced if required based on the amount of error. This correction factor can be a constant correction factor or it can be proportional to the amount of error as in simple control systems.

For STI implementation all blocks of code must have a constant execution time, which can be statically determined. Since the transition window deals with the detection of a skew and resynchronization, the skew adjustment factor is part of this transition window code execution. This transition window $T_{transition}$ has a static execution time as far as control flow is concerned. The time between successive target transitionWindows is called idleWindow T_{idle}. Figure 5.0.1 explains the various segments in a bit interval.

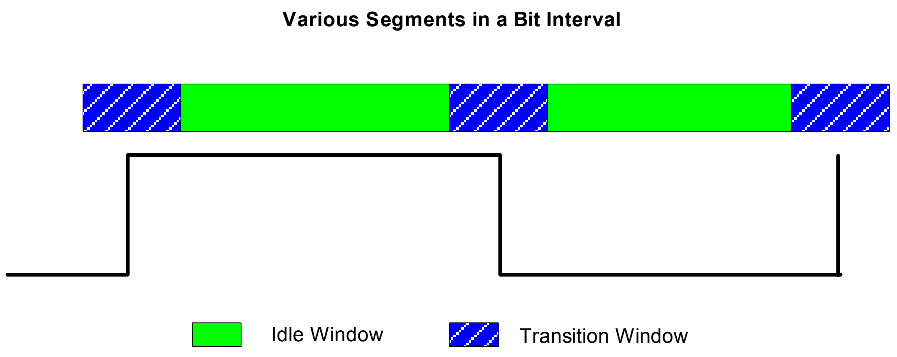

Figure 5.0.1 Various segments of a Bit Interval

The different ways in which the transition detection and resynchronization is done is classified into 4 different clock recovery techniques as follows.

a) Simple Bang Bang Control (Simple BBC) Technique
b) Proportional Control Technique
c) High Frequency Polling Technique
d) Over-sampling Technique

5.1 Simple Bang Bang Control Technique

The idleWindow period is divided into three periods, $T_{pre\text{-}sampling}$ $T_{sampling}$ and $T_{postsampling}$, such that $T_{sampling}$ is at the middle of the idleWindow. During $T_{sampling}$, the medium is sampled multiple (m) times. A majority among these m samples is calculated and output as the sampled bit. The transitionWindow is divided into four sampling zones T_{ts}s and a threshold window called $T_{threshold}$. $T_{threshold}$ is preceeded by 2 T_{ts}s and followed by the other two T_{ts}s. The medium is sampled once during each T_{ts}. $T_{threshold}$ is an idle buffer time. The four transition samples are used to detect the position of a transition and correct the clock by a constant factor if required.

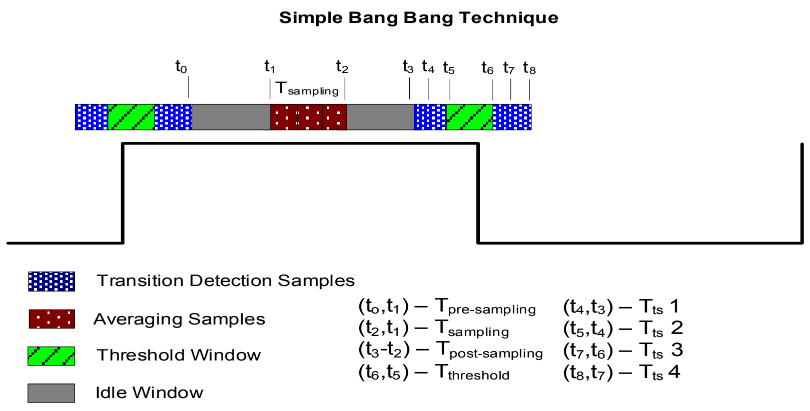

Figure 5.1.1 Simple Bit Bang Technique

Figure 5.1.1 explains the various timing windows in a bit interval. The duration of a bit, Tbit is given by

$$T_{bit} = t_8 - t_0.$$

A perfectly synchronized receiver will have the transition occurring during $T_{threshold}$. If a transition is detected between the first and second transition samples, then the instant at which the second sample is taken is considered to be the end of the $T_{threshold}$ and the transition window is immediately corrected by the constant correction factor. If a transition is detected between the third and fourth transition samples, then the constant correction factor is introduced after the fourth sample. These corrections are idle processor durations when no code is executed. This idle time cannot be used for executing the secondary thread because the occurrence of these durations cannot be statically determined. These corrections approximately correct the skew such that, the next set of averaging samples happen approximately at the middle of the next bit duration.

The idleWindow duration of a bit has some piece of code, which is executed during $T_{sampling}$. This small piece of sampling code is enclosed on either side by large idle times $T_{pre\text{-}sampling}$ and $T_{post\text{-}sampling}$ on the primary thread. These idle times can be used for the execution of the secondary thread. The small piece of code for calculating the bit value can be sandwiched by the secondary thread code using ASTI.

The advantage of this method is that it is simple and does not have very tight real time constraints. The disadvantage is that the correction is only a constant number of cycles. So the recovery does not happen for the exact amount of skew. Hence the resynchronization gets triggered periodically. The periodicity is dependent on the constant correction factor and the amount of error.

5.2 Proportional Control Technique

This technique is almost the same as Simple BBC technique. The idleWindow duration is divided into a $T_{pre-sampling}$, $T_{sampling}$ and $T_{post-sampling}$ windows. During $T_{sampling}$ the medium is sampled multiple (m) times. The sampled bit output is the majority of these m samples. The transition detection is done by taking four samples during the targeted transition window. The first two samples and the last two samples enclose an idle $T_{threshold}$. Based on the position of the transition a recovery is done. This technique differs from Simple BBC only in the way the correction factor is calculated. Simple BBC uses a constant correction factor. Proportional control introduces a correction factor which is proportional to the amount of drift. The receiver keeps track of the number of bus bit times it takes for the transition detection to occur outside of the threshold window after a correction is done or since the beginning of the sampling. A correction factor which is proportional to this number of bus bit times is introduced every time the transition is detected outside the threshold window. The proportionality constant is a configuration parameter. Since this technique introduces a correction factor which is proportional to the actual error, the resynchronization is adaptive to the amount of error.

5.3 High Frequency Polling Technique

The idleWindow T_{idle} is divided into 3 regions $T_{pre-sampling}$, $_{Tsampling}$ and $T_{post-sampling}$. During $T_{sampling}$ the medium is sampled multiple (m) times. A majority among these m samples is calculated and output as the sampled bit. $T_{pre\text{-}sampling}$ and $T_{post\text{-}sampling}$ can be used for the execution of the secondary thread.

For transition detection up to N samples are taken during the transitionWindow $T_{transition}$. The samples are taken until a transition is detected or until the sample count reaches N. Based on the number of samples taken until the transition was detected a correction factor is calculated. This correction factor is such that the next set of averaging samples is taken at the middle of the next bit interval. A perfectly synchronized receiver has the transition occurring between the $(N/2)^{th}$ and $((N/2) + 1)^{th}$ samples.

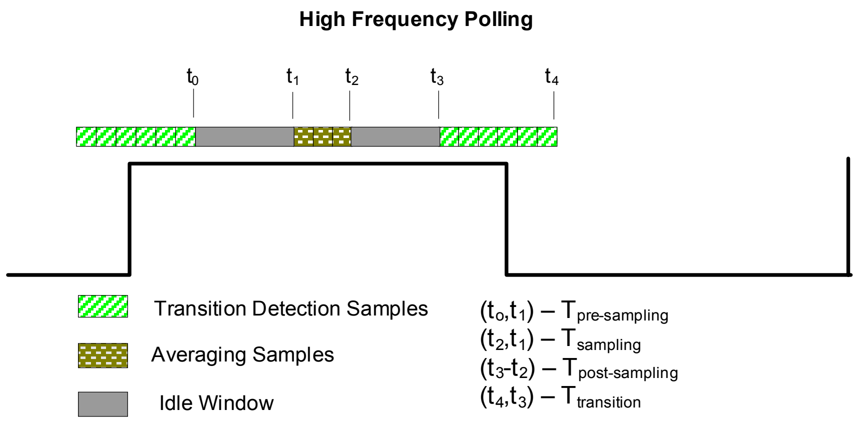

Figure 5.3.1 High Frequency Polling Technique

Figure 5.3.1 shows the various timing windows for the High Frequency Polling technique. During $T_{pre\text{-}sampling}$ and $T_{post\text{-}sampling}$ the secondary thread can be executed using co-calls. STI techniques merge the primary and secondary thread such that the primary thread is executed during $T_{sampling}$. If $T_{pre\text{-}sampling}$ and $T_{post\text{-}sampling}$ are small such that two co-calls cannot be executed during either of these, then the sampling code to be executed during $T_{sampling}$ can be moved into the secondary thread using ASTI. During $T_{transition}$ about N samples (6 are shown in the figure) are taken. If no transitions are detected during $T_{transition}$, N samples are taken. If at any point before Nth sample a transition is detected, the sampling is stopped and the sample before the transition m is used to calculate an inactivity time interval $T_{inactive}$. This inactive interval is calculated so that $T_{transition}$ is increased or decreased to adjust for the drift in clock frequency of the sender causing the next $T_{sampling}$ to occur at the middle of the next bit interval.

Here the duration of a bit T_{bit} is given by

$$T_{\mathbf{bit}} = t_4 - t_0$$

$$T_{\mathbf{inactive}} = N/2 * T_s$$

$$T_{\mathbf{transition}} = (N/2 + m) * T_s$$

where

T_s is the time taken for one sample

This technique is better than the first two techniques in that, it exactly locates the position at which the transition happens. This technique also introduces a correction factor proportional to the amount of drift that has been observed during the present bit interval. Thus this technique ensures immediate recovery from any drift, which is greater than or equal to the time taken to measure a single sample. The disadvantage of this technique is that it is processor intensive.

5.4 Over-sampling Technique

This technique executes the primary thread only during the transition window $T_{transition}$. During the idleWindow T_{idle}, only secondary thread is executed. All samples are taken during $T_{transition}$. These samples are used for calculating the sampled bit value and for detecting transitions. This yields a single block of code, the primary thread, which is executed periodically after having a gap of T_{idle}. This primary thread outputs the sampled bit, detects a transition and does a resynchronization with the sender. The primary and secondary thread can be merged using STI techniques as T_{idle} is generally large enough to accommodate two co-calls.

During $T_{transition}$ N samples are taken, where N is a configured value. A buffer time T_{buffer} at the beginning of the transition window is used to adjust the receiver's clock if a jitter is detected during the previous $T_{transition}$. The position of a transition if present is calculated and the buffer time is adjusted in order to position the next $T_{transition}$ at the correct position.

For a perfectly synchronized sender and receiver T_{buffer} has a configured constant value. If the sender is faster than the receiver, then T_{buffer} is reduced accordingly in order to have the receiver in synchrony with the sender. The correction factor is limited by the constant value configured for T_{buffer}. Similarly, if the receiver is faster than the sender, then this buffer interval is increased accordingly so that the next $T_{transition}$ occurs at the correct position so that the next transition can be detected with in $T_{transition}$. This correction factor does not have a limit as it merely increases T_{buffer}.

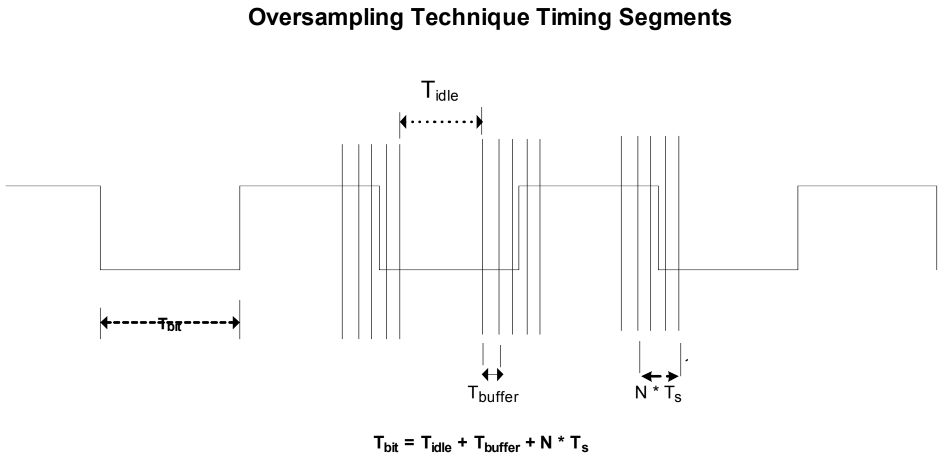
Oversampling Technique Timing Segments
T_{idle}
T_{bit}
T_{buffer}
$N * T_s$
$T_{bit} = T_{idle} + T_{buffer} + N * T_s$

Figure 5.4.1 Oversampling Technique Timing Segments

The time available for the execution of the secondary thread is T_{idle}. The primary thread has tight real time constraints. It has to start executing at the correct position with respect to the data on the medium so that it can detect a transition. Figure 5.4.1 shows the various sampling points during the consecutive bit intervals.

The duration $T_{transition}$ is dependent on 2 factors namely

1) The clock speed of the processor executing the code
2) The number of samples to be taken N.

Assuming that each sample takes T_s cycles, and the medium is sampled N times per bit duration, N/2 in the beginning and N/2 towards the end and the processor takes T_X nanoseconds per cycle,

$$T_{\mathbf{transition}} = 2 * T_s * N/2 * T_X \textit{ nanoseconds}$$

T_{buffer} is a configurable parameter. This determines the amount of recovery that can be made in every bit-duration. Assuming this is configured to be 1 times T_s number of cycles.

Then

$$\mathbf{T_{buffer} = T_s}$$

$$T_{\mathbf{bit}} = T_{\mathbf{idle}} + (N+1) * T_s \textit{ cycles}$$

The restriction however is that, if a jitter is detected, then the maximum possible cycles to which the jitter can be corrected for is T_{buffer} cycles. In comparison with the other techniques $T_{transition}$ might be larger for this technique.

From figure 5.4.2 it can be seen that,

$$T_{\mathbf{idle}} = t_1 - t_0$$

$$T_{\mathbf{transition}} = t_4 - t_1$$

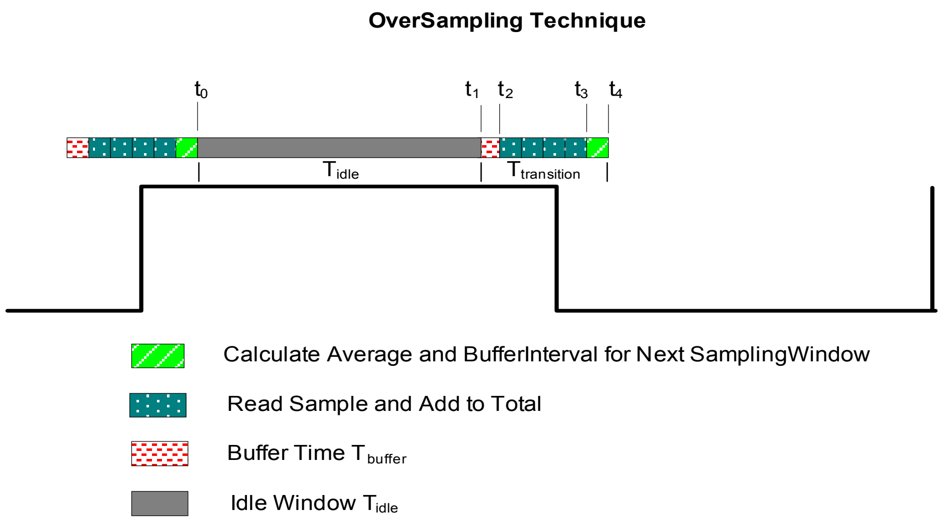

Figure 5.4.2 Oversampling Technique

6. Analysis of Results

6.1 Implementation Details

The implementation has been aimed for ATMEGA103 microcontroller from ATMEL. This is an 8 bit microcontroller which runs at 4 Mhz. It is assumed that the input is available on pin 0 of PORTB (PORTB is addressed as PINB when configured for input). A starting trigger is assumed to be given through pin 0 of PORTD (PORTD is addressed as PIND when configured for input). This trigger is assumed to be a synchronization point where in a bit duration begins on the PORTB. The sampling is done as long as pin 0 of PORTD has a value of 1. The implementations were simulated using AVR Studio 4.0 from Atmel [35].

The simulations were run for a given set of input sequence consisting of alternating 1's and 0's. The duration of a bit interval is configured to be 250 cycles. The sequence starts after leaving 10 bit durations (2500 cycles) for initialization. The implemented techniques serve as receivers. The message is simulated by the generated sequence. The receivers try to resynchronize their clock with the sender and correct any detected skew. The time for a single sample was found to be around 10 cycles for the implementations.

6.2 Results

This section compares the performance of the various techniques. These graphs show the phase difference between the sender and the receiver normalized to a single bit time, where the sender has various clock frequency errors. These graphs show how quickly the receiver recovers the clock and corrects itself.

6.2.1 Simple BBC

For this technique, the threshold window was configured to be 40 cycles. Three samples (m = 3) taken during the middle of the bit duration were used for calculating the bit value. Since the sampling is done at the middle of the bit duration, the output is at a phase difference of about half bit duration with the input sequence. Figure 6.2.1 shows the phase difference between the sender and receiver normalized to a single bit interval for various frequency errors at the sender. It can be seen that the phase difference varies by large amounts as the error increases. This causes periodic resynchronizations. Each fall in the graph denotes a resynchronization.

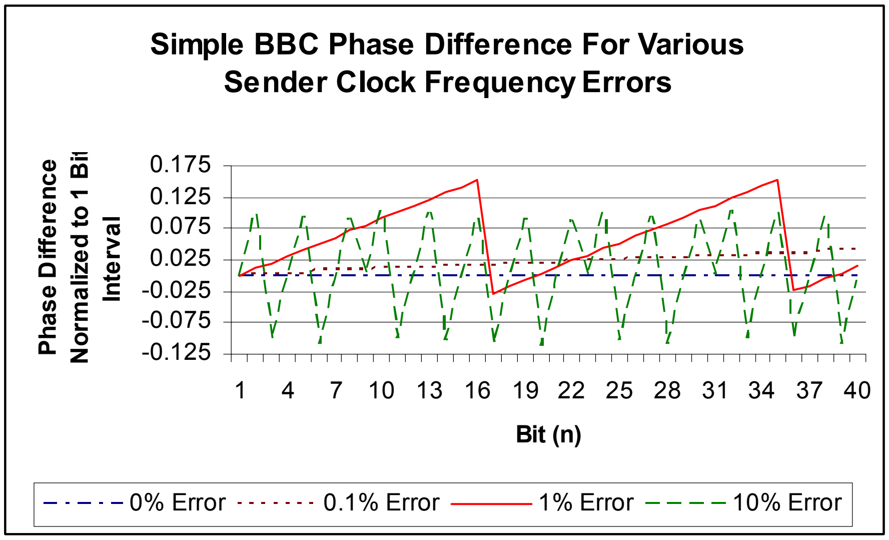

Figure 6.2.1 Simple BBC Phase Difference vs Bit (n)

6.2.2 Proportional Control

For this technique, the threshold window was again configured to be 40 cycles. The proportionality constant was configured to be 4. The number of iterations that it took, for the receiver to exceed the threshold was calculated. A correction factor which is 4 times the number of iterations was applied to the receiver. The phase difference between the receiver and the sender is calculated. This phase difference has been normalized to 1 bit interval. Figure 6.2.2 shows this phase difference as a sequence of bits was being sampled at the receiver for various frequency errors in the sender's clock. The variation in the phase difference has an average value of about 0.075. It can be seen that the phase difference varies by large amounts as the frequency error on the sender increases. This causes frequent resynchronizations as error increases. Each fall in the graph denotes a resynchronization.

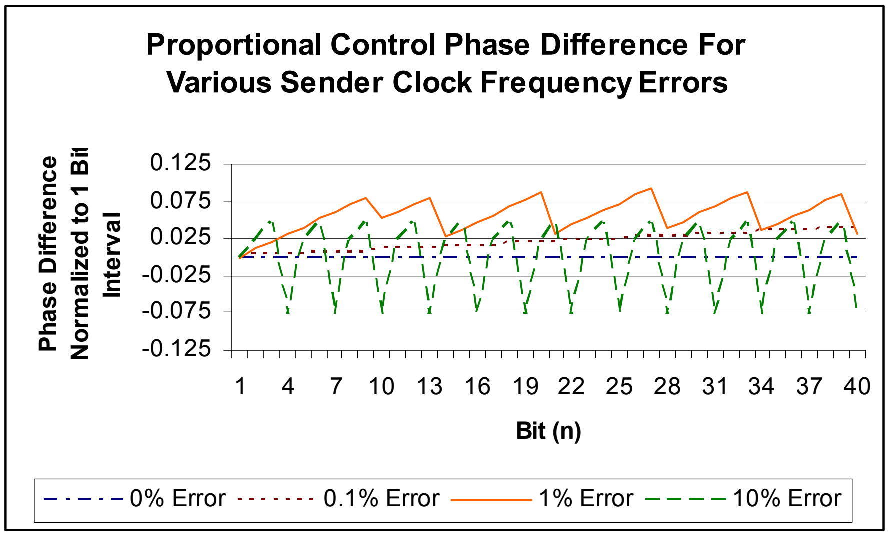

Figure 6.2.2 Proportional Control Phase Difference vs Bit (n)

6.2.3 High Frequency Polling

For this technique, N was assumed to be 8, that is up to 8 samples are taken in the target transition window and the receiver is adjusted based on the position of the transition among these 8 samples. A separate set of three samples in the middle of the bit duration is used for calculating the bit value. The phase difference between the receiver and the sender was calculated and normalized to 1 bit time. This normalized phase difference was plotted against the n Bits for various frequency errors introduced in the sender's clock as shown in Figure 6.2.3. The variation of phase difference averages about 0.01 for low frequency errors and about 0.03 for high frequency errors. The variation of the phase difference increases for high frequency errors. The number of samples used for transition detection decreases as the phase difference between sender and receiver clocks increases.

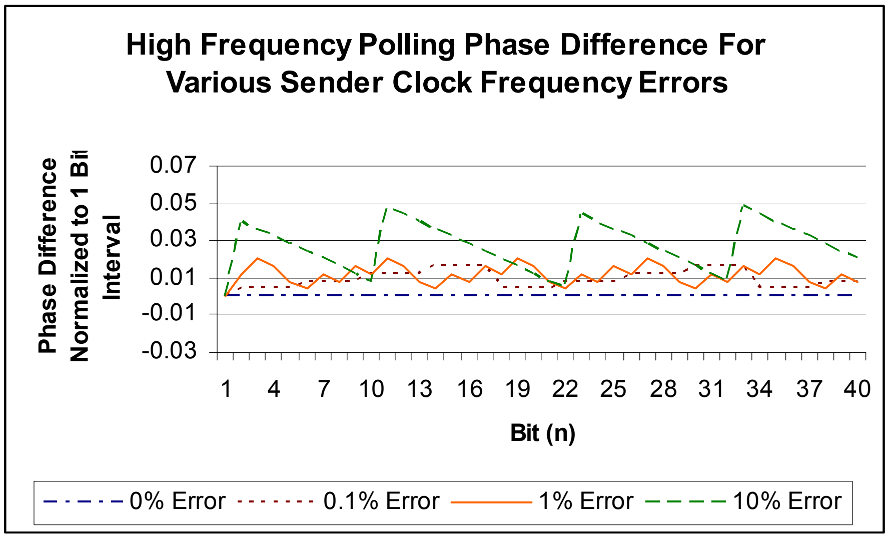

Figure 6.2.3 High Frequency Polling Phase Difference vs Bit (n)

6.2.4 Over-sampling

For this technique, again N was assumed to be 4, i.e. 4 samples are taken in the target transition window. These 4 samples are used to locate the transition and adjust accordingly. The two samples taken at the start of the current bit and the two samples that are taken towards the end of the current bit are used for averaging. The drift between the sender and receiver was calculated and normalized. This normalized drift relative to 0% error, was plotted against the bit index for various frequency errors in the sender's clock, as shown in Figure 6.2.4. Irrespective of the frequency errors, this technique triggers a resynchronization very frequently. This is an inherent nature of this technique, because unlike in other techniques the number of samples taken during transition detection is not reduced even when a large phase difference with the sender is detected.

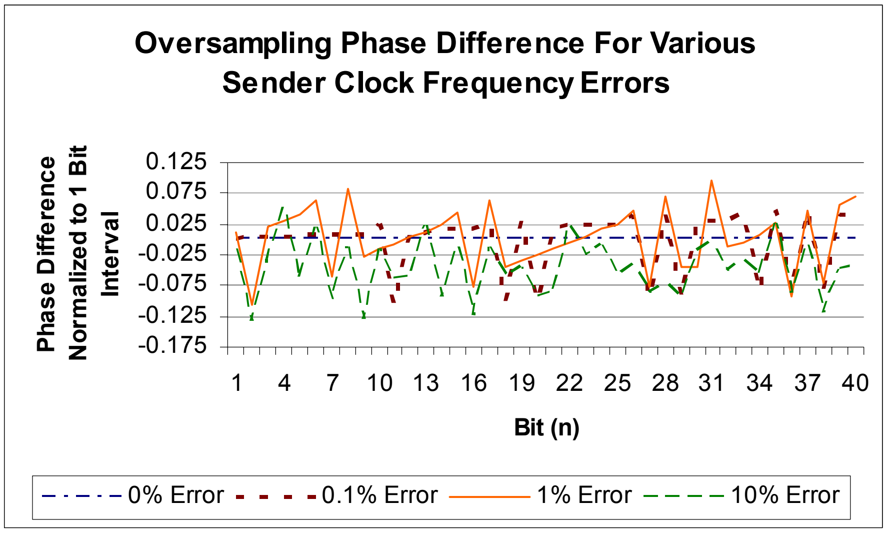

Figure 6.2.4 Oversampling Phase Difference Vs Bit (n)

6.3 Comparison Of the techniques

A comparison of the various techniques is shown below in Table 6.3.1.

Parameter	**SimpleBBC**	**Proportional Control**	**High Freq. Polling**	**Over-sampling**
Code Size (words)	144	177	151	151
Data Segment (bytes)	0	0	Number Of Samples (8)	0
Execution time (cycles) $T_{sampling} + T_{transition}$	136	116	142	123
Initialization time (cycles)	15	15	181 (look up table init)	18
Average Transition Window Size (cycles) $T_{transition}$	108	88	112	123
Maximum Possible Drift per bit interval	Threshold Window Size (40 cycles)	Threshold Window Size (40 cycles)	One Sampling Duration (15 cycles)	One Sampling Duration (15 cycles)

When Correction is applied	End of transition window	End of transition window	End of transition window	Beginning of next transition window
Amount Of Correction	Constant	Proportional to Number of bit-times it takes to exceed the threshold window	Proportional to amount of drift	Proportional to drift up to a maximum correction buffer
Minimum delay between input and output	Half Bit Duration	Half Bit Duration	Half Bit Duration	One Bit Duration
Number Of IdleWindows	2	2	2	1
Idle Window Sizes (cycles) ($T_{bit}=250$)	60,54 ($T_{bit}-T_{sampling}-T_{transition}$) ($T_{sampling}=28$)	82,52 ($T_{bit}-T_{sampling}-T_{transition}$) ($T_{sampling}=28$)	66,42 ($T_{bit}-T_{sampling}-N*T_s$) ($N=8$, $T_s=14$)	127 ($T_{bit}-T_{sampling}-N*T_s-T_{buffer}$) ($N=4$, $T_s=19$, $T_{buffer}=19$))
Parameters	Threshold window, Constant Correction factor	Threshold Window, Proportionality Constant	Number Of Samples	Number Of Samples, Maximum Correction Buffer
Maximum number of bits without resynchronization for a 1% error	17	9	3	3

Table 6.3.1 Comparison of the Various Techniques

A comparison of the phase drift between the sender and the receiver is shown in figure 6.3.1. This difference has been normalized to 1 bit duration. This shows the amount of time taken by each technique as well as how quick they respond to variations in the clock of the sender. This example plot is for a 10% Clock error on the sender. The plot shows that the variation of the phase difference is very less for High Frequency polling technique. Simple BBC and Proportional Control Techniques are found to have high variations in their error rates when compared to Over-sampling and High Frequency Polling techniques.

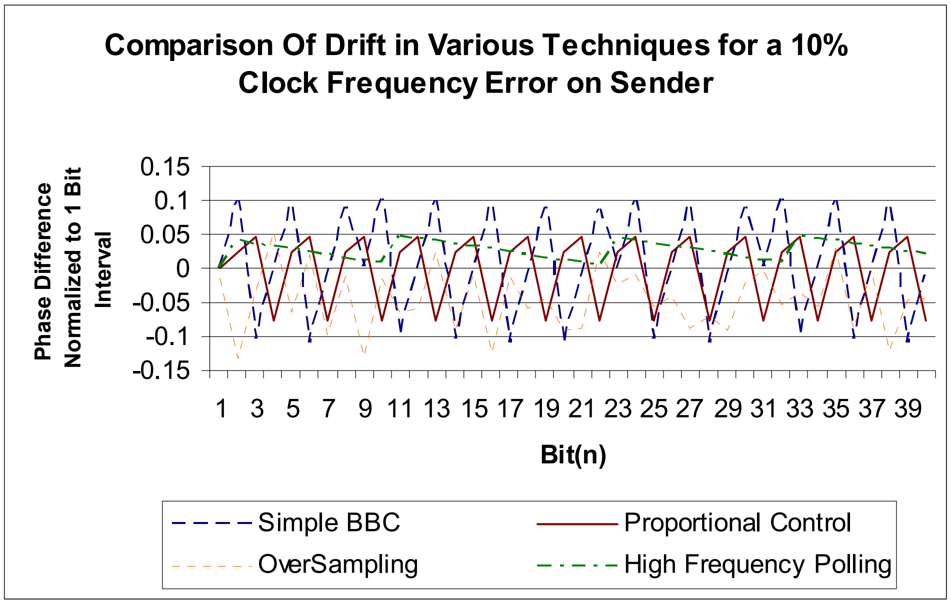

Figure 6.3.1 Comparison Phase Drift Vs Bit (n) for a 10% Error

The maximum number of bits that each technique can sample without a resynchronization was also plotted for the various frequency errors on the sender for the various techniques as shown in Figure 6.3.2. It can be seen that the number of bits that can be sampled without a resynchronization decreases as the frequency error increases for Simple BBC, Proportional control and High frequency polling techniques. However, the Over-Sampling technique has a constant number of bits sampled before each resynchronization. The Over-sampling technique does a resynchronization more often than other techniques even for low frequency errors. This is because $T_{transition}$ does not vary much in the over-sampling technique when compared to the other techniques.

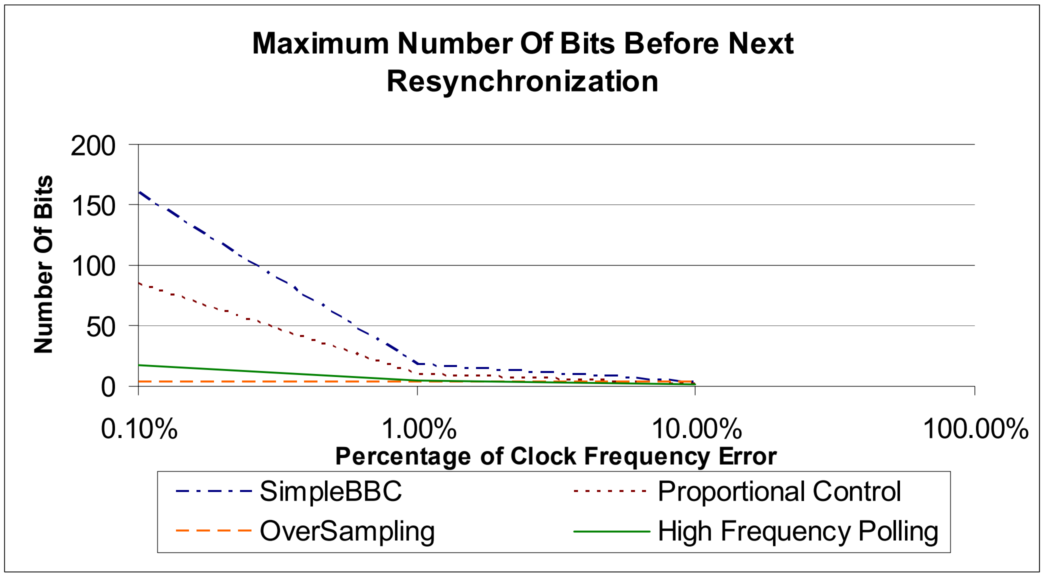

Figure 6.3.2 Maximum Number of Bits before Re-synchronization Vs Error

7. Conclusion and Future Work

7.1 Conclusion

In order to implement a protocol in software using STI, the primary thread, must have large chunks of idle time which can be recovered and used to execute useful code from a secondary thread. The bus interface thread has to implement the various clock recovery techniques. This paper has shown that it is possible to implement software clock recovery techniques with large durations of idle time.

Experiments were performed where the sender had different rates of clock frequency errors when compared with the receiver. A good clock recovery technique must try to keep the phase drift between the sender and receiver as little as possible. The proposed techniques did keep the receiver synchronized with the sender, while still providing constant idle time durations. Simple BBC technique did a constant recovery, and hence had periodic resynchronizations irrespective of amount of error. Proportional Control had recovery proportional to the amount of error and hence had periodic re-synchronizations proportional to the amount of error. High frequency polling technique tried to keep up with the sender and recover clock errors as soon as possible. Over-sampling technique has a delayed recovery where errors detected in the current bit duration are corrected during the next bit duration. Since these recovery and resynchronization functions must have a fixed duration for performing STI, the transition window has padded buffer duration which can be used for error recovery. The amount of error that these techniques can recover from has is limited to this buffer size. Increasing this buffer decreases the amount of idle time available in the bit level function.

When the protocol has large bit durations and there is a lot of noise in the medium, then it could lead to distortion of the value transmitted on the medium near the edges. In such cases over-sampling or high frequency polling techniques should not be used, because these techniques use the samples obtained near the edges. Instead techniques like simple BBC or proportional control must be used to recover the clock, as these techniques permit a threshold window for a transition to happen. In cases where there is constant value on the wire during the entire bit duration, over-sampling and high frequency polling techniques are very effective.

Also, when the amount of expected error in the frequency of the clocks is less or the size of messages transmitted in the protocol is small, simple BBC or proportional control techniques will be effective. Longer messages and higher error percentage in the frequency of the clocks advocate the use of over-sampling or high frequency polling techniques, as these ensure quicker clock recovery on the receiver.

7.2 Future Work

If the input signal fades in the wire due to noise constraints, the over-sampling technique may not be very effective and it is prone to errors. This can be corrected by introducing a possible threshold buffer window at the exact position where a possible transition may happen. This buffer window can be claimed through the use of ASTI to execute secondary thread.

Proportional technique can be improved so that, the proportionality constant can be dynamically calculated instead of configuring it as a static value.

These techniques could be integrated as part of *Thrint* compiler tool, so that they can be seamlessly integrated into any software protocol implementations done using STI.

This paper assumes that the processor executes each instruction at a fixed rate. Another case to consider is a processor which can vary its code execution time is considered. Such a processor could achieve resynchronization with the sender over the entire duration of the message by varying the speed with which it executes the code as the message is received.

8. References

[1] Dean, A. "*Software thread integration for Hardware to Software Migration*". Phd Dissertaion, Carnegie Mellon University, May 2000.

[2] 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.

[3] Nagendra, J., Kumar. "*STI Concepts for Bit-Bang Communication Protocols*". Master's Thesis, North Carolina State University, May 2003.

[4] Nagendra J. Kumar, Siddhartha Shivshankar and Alexander G. Dean. *"Asynchronous Software Threading for Efficient Software Implementations of Embedded Communication Protocol Controllers"*. ACM SIGPLAN / SIGBED 2004 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, June 11-13, 2004.

[5] Sunil Vangara, *"Code Motion Techniques for STI in BBCP"*, Master's Thesis, North Carolina State University, 2003.

[6] Lioupis .D, Papagiannis .A, Psihogiou .D, *"A Systematic approach to Software Peripherals for Embedded Systems"*, Ninth International Symposium on Hardware Software Co-design (CODES 2001), Copenhagen, Denmark, April 25-27, 2001, pp. 140-145.

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

[8] John Poulton, William Dally, Steve Tell, *"A Tracking Clock Recovery Receiver for 4Gb/s Signaling*", Proceedings of Hot Interconnects V, Stanford University, Aug 21-23, 1997, pp. 157-169

[9] Jin Ku Kang, "*Performance Analysis of Over-sampled Data Recovery Circuit"*, IEICE Transactions Fundamentals, Vol. E82-A, NO.6, June 1999, pp. 958-964.

[10] Scott George, *"HC05 Software-Driven Asynchronous Serial Communication Techniques Using the MC68HC705J1A"*, Motorola Semiconductor Application Note AN1240

[11] Greg Goodhue, "*A Software Duplex UART for the 751/752*," Philips Semiconductors Application Note 446

[12] Thomas F. Herbert, "*Integrating a Soft Modem*," Embedded Systems Programming, 12(3), March 1999, pp. 62-74

[13] Stephen Holland, "*Low-Cost Software Bell-202 Modem*," Circuit Cellar, June 1999, #107, pp. 12-19

[14] Naji Naufel, *"Interfacing the 68HC05C5 SIOP to an I2C Peripheral*," Motorola Semiconductor Application Note AN1066

[15] Tony Breslin, "*68HC05K0 Infra-Red Remote Control*," Motorola Semiconductor Application Note AN463, 1997

[16] Peter Topping, "*An RDS Decoder using the MC68HC05E0*," Motorola Semiconductor Application Note AN460

[17] Atmel Corporation, "Software LIN Slave," Application Note AVR308, February 2000

[18] Philips Semiconductors, "*P82C150 CAN Serial Linked I/O Device (SLIO) with Digital and Analog Port Functions Data Sheet,*" June 19, 1996

[19] Microchip Technology, Inc. "*MCP2502X/5X CAN I/O Expander Family Data Sheet*," 2001

[20] Micrel, Inc. "*MIC74 2-Wire Serial I/O Expander and Fan Controller*," August 2000

[21] Philips Semiconductors, "*PCF8574 Remote 8-bit I/O Expander for I2C-bus*," November 22, 2002

[22] Atmel Corp., "*AVR304: Half Duplex Interrupt Driven Software UART*," August 1997

[23] Atmel Corp., "*AVR305: Half Duplex Compact Software UART*," May 2002

[24] Atmel Corp., "*AVR308: Software LIN Slave,*" May 2002

[25] Atmel Corp., "*AVR320: Software SPI Master*," May 2002

[26] Atmel Corp., "*AVR410: RC5 IR Remote Control Receiver*," May 2002

[27] Philips Corp., *"CAN Applications for P8Cx591"* – Application Note http://www.semiconductors.philips.com/acrobat/applicationnotes/AN00043.pdf

[28] Motorola Semiconductors, "*AN 1798D - CAN Bit Requirements"*

[29] Siemens Microelectronics *"CANPRES Specifications"*

[30] David Bursky, "*Speedy 8 bit microcontroller crafts virtual peripherals*", Electronic Design, #185, August 4, 1997, p.36.

[31] Justin Redd, "*Synch and Clock Recovery – An analog guru looks at jitter*", www.planetanalog.com/features/communications/OEG20010827S0037

[32] Infineon Technologies, AP2925 *"CAN Baudrate detection with Infineon CAN devices"*

[33] Ubicom, Application Note 20, *"SPI and Microwire/Plus implementation using SX Communications Controller"*

[34] Ubicom, Application Note 29, "*I2C Virtual Peripheral Implementation"*

[35] Giga, "*2.5 Gbit/s Clock And Data Recovery Circuit",* GD16543.

[36] AVRGCC*, http://www.avrfreaks.net/AVRGCC/index.php*

[37] Atmel Corporation, AVR Studio 4.0 Editor, Assembler & Simulator, *http://www.atmel.com/dyn/resources/prod_documents/AVRStudio4.exe*