2-Wire Time Independent Asynchronous Communications

By

Pj Radcliffe

B.Eng. (Melb), M.Eng. (RMIT)

A dissertation submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

School of Electrical & Computer Engineering

Science, Engineering & Technology Portfolio

RMIT University

Melbourne, Victoria

Australia

Submitted November 2006

Abstract

Communications both to and between low end microprocessors represents a real cost in a number of industrial and consumer products. This thesis starts by examining the properties of protocols that help to minimize these expenses and comes to the conclusion that the derived set of properties define a new category of communications protocol : Time Independent Asynchronous (TIA) communications.

To show the utility of the TIA category we develop a novel TIA protocol that uses only 2-wires and general IO pins on each host. The protocol is analyzed using the Petri net based STG (Signal Transition Graph) which is widely use to model asynchronous logic. It is shown that STGs do not accurately model the behavior of software driven systems and so a modified form called STG-FT (STG For Threads) is developed to better model software systems. A simulator is created to take an STG-FT model and perform a full reachability tree analysis to prove correctness and analyze livelock and deadlock properties. The simulator can also examine the full reachability tree for every possible system state (the cross product of all sub-system states), and analyze deadlock and livelock issues related to unexpected inputs and unusual situations. Reachability pruning algorithms are developed which decrease the search tree by a factor of approximately 250 million.

The 2-wire protocol is implemented between a PC and an Atmel Tiny26 microprocessor, there is also a variant that works between microprocessors. Testing verifies the simulation results including an avoidable livelock condition with data throughput peaking at a useful 50 kilobits/second in both directions.

The first practical application of 2-wire TIA is part of a novel debugger for the Atmel Tiny26 microprocessor. The approach can be extended to any microprocessor with general IO pins.

TIA communications, developed in this thesis, is a serious contender whenever low end microprocessors must communicate with other processors. Consumer and industrial products may be able to achieve cost saving by using this new protocol.

Declarations

Except where due acknowledgment has been made, all work is that of the candidate alone.

This thesis has not been submitted previously, in whole or in part, to qualify for any other award.

The content of this thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Any editorial work, paid or unpaid, carried out by a third party is acknowledged.

signed

(Pj Radcliffe)

Acknowledgments

First I must acknowledge my family for being so supportive and helping me find time to pursue this thesis.

Next I would like to acknowledge the support and encouragement of the best PhD supervisor I have seen in a long time : Dr. Xinghuo Yu.

Table of Contents

List of Illustrations

List of Tables

1 Introduction

Many consumer and industrial products require communications with, or between embedded microprocessors. The addition of a communications function often forces the selection of more expensive microprocessors due to issues such as the need for a communications peripheral, or the need for extra processing power to service both the application and communication activities. Typically the manufacture cost is increased by between US$0.25 to US$1 per microprocessor.

Is it possible to devise a communication method that is lower in cost than existing methods? If such a method is possible then it may well find application in cost sensitive consumer and industrial products. This thesis aims to develop a new protocol that will reduce the cost of microprocessor communications particularly for Low End Microprocessors.

1.1 The Problem Faced

There are a vast variety of existing communication protocols and many such as I2C (Philips 2000) and SPI (Freescale 1994) were created specifically to suit Low End Microprocessors (LEMs). Surely one of these methods should suit any LEM application, yet experience working on many LEM projects has shown specific problems with all existing solutions.

First comes the issue of microprocessor purchase cost introduced above. The addition of a communications link can easily add US$0.25c to US$1 to the cost of a microprocessor (Microchip 2006). On a production run of 100,000 items, using the 25c figure, this becomes at US$25,000 and after a markup from manufacture to retail of perhaps 4 to 1 this becomes US$100,000.

Another problem is caused by the observation that a communication link must be serviced within a given time-frame otherwise data will be lost. This starts to place constraints on the architecture, timing and performance of the application software. The servicing of the communications function may also cause unwanted timing jitter to the application. The net result can be that a more powerful microprocessor must be chosen or the selected microprocessor must be run at a higher clock speed which will increase power drain. Applications such as simple switch sensors will not be troubled by these problems, but other applications such as Digital Signal Processing tasks will find these problems quite important.

The ideal communications method for a LEM will add nothing to the cost of the microprocessor, not effect the timing or software architecture of the application, and make no demands the processing power of the microprocessor. Achieving the goal as stated is clearly impossible but it should be possible to get closer to this goal than existing solutions.

1.2 The State of the Field

There are a large number of excellent communications protocols for Low End Microprocessors. Most are available as built in hardware peripherals on a microprocessor but many could be implemented in software to allow the selection of a cheaper microprocessor. The software solution brings its own costs in the use of timers, interrupts, code space, and the interference afforded the application in terms of timing and software architecture.

Most protocols must be serviced within a fixed time or have timing constraints and so even if they are implemented as hardware peripherals they still negatively effect application timing and software architecture. Section 2.2.1 describes and references a representative range of these protocols including RS232, I2C, SMBus, SPI, I2S, Dallas one-wire, CAN bus, LinBus, USB, Ethernet, Firewire, and MIL-STD-1553.

A small number of protocols do not make timing demands on the application. Section 2.2.2 of this thesis describes and references a range of these protocols including Centronics, GPIB, DEC Unibus, VME bus, and several cross IC buses for VLSI chips. These approaches use a large number of wires and would thus take up an excessive number of IO pins on a Low End Microprocessor. The smallest footprint found in the literature are 3 wire systems but for a Low End Microprocessor three IO pins can be a real resource problem.

The current solutions to Low End Microprocessor communications either add expense, interfere with the application, or demand scarce IO pins.

1.3 Motivation and Objectives

The goal of this thesis is to develop a communications protocol that can be used to reduce communication costs for Low End Microprocessors and to improve on the current state of the field.

This goal raises a number of important research questions that this thesis will attempt to answer-

- What are the properties of a communication protocol that would help reduce the cost of communications with low end microprocessors?
- Do these proper ties describe a protocol category? If so is this an existing category or a new one?
- What existing protocols exist within a suitable category?
- Given insights from the preceding analysis is it possible to create an improved communication protocol that better serves the needs of Low End Microprocessors?
- If such an improved protocol is developed how should it be modeled, simulated, and tested?

These questions provide a structured, academic way to attack a real world problem.

1.4 Overview of Thesis Structure

The chapters in the thesis are driven by the research questions posed earlier in section 1.2.

- **Chapter 1 : Introduction** which outlines the key research questions to be answered. These are based around finding a economic way to communicate with low end microprocessors.
- **Chapter 2 : Literature Survey : Communication Category Identification** : this chapter examines the attributes of the ideal communications system for low end microprocessors and how well existing protocols address this ideal. A gap is discovered and so a new communications category is proposed : Time Independent Asynchronous (TIA) communications. Existing TIA protocols are identified and discussed but none found ideal for low end microprocessors.
- **Chapter 3 : Literature Survey : Protocol Verification and Modelling Issues.** This chapter examines what properties must be verified when a new protocol is defined, and what protocol modelling techniques are appropriate to TIA style protocols. The well known Signal Transition Graph model is shown to have deficiencies and so a new modelling method called STG For Threads (STG-FT) is developed.
- **Chapter 4 : 2-wire TIA Protocol.** The utility of TIA is proven by proposing a novel 2-wire TIA communications protocol. This is modelled using the new STG-FT model and

verified by a custom made simulator. It is shown that 2-wire TIA is guaranteed to work apart from one live-lock which can easily be avoided. The system is implemented and tested and the performance found to match the simulation.

- **Chapter 5 : 2-wire TIA variant.** A variant of the original 2-wire TIA protocol is proposed, simulated, and found to have properties very similar to the original 2-wire TIA. This variant is better suited to microprocessor-microprocessor communications using standard drive electronics.
- **Chapter 6 : Practical Use as an Embedded Debugger.** Existing debugging tools for microprocessors are examined and an unfilled niche is discovered. The 2-wire TIA system is used as the basis for a novel microprocessor debugger that is of very low cost and fills the niche identified.
- **Chapter 7 : Discussion and Conclusions** : discussion and summary of the main findings in relation to the research questions and contributions to the field of knowledge.
- **References** list key information sources discussed in the thesis.
- **Appendix A** lists the data and software contained on the accompanying CDROM.

1.5 Contributions of This Thesis

The thesis structure just presented gives an overview of the achievements but there are particular sections that make a useful contribution to the body of knowledge.

The first contribution comes in section 2.1 which proposes a new physical layer communications category called Time Independent Asynchronous (TIA) communications. This category sits along the communication categories of synchronous, plesiochronous and asynchronous communications. The TIA category is shown to have particular attributes that can make it very attractive for Low End Microprocessors.

Another contribution is found in section 3.2.2 where the well known Signal Transition Graph (STG) modelling technique is found to have problems when attempting to model the behavior of systems implemented using software threads. This is a concern as STGs are an excellent method of analyzing the behavior of communication protocols. Section 3.2.2 proposes a novel extension to STGs called STG For Threads (STG-FT) that overcomes the problems identified.

A study of existing TIA protocols in chapter 2 showed several 3 wire communications protocols. This thesis goes further and chapter 4 develops a new and novel 2-wire TIA protocol.

Section 4.4 outlines a simulator that has been created to rapidly simulate a system described in STG-FT format. Careful selection of language and architecture allow it to work at a useful 4.8 million nodes a second on a 1.6 GHz Celeron processor. Brute force evaluation of the reachability tree proved time consuming but a novel state "tick off" strategy was developed that allowed simulation to be sped up by a factor of nearly 250 million.

Chapter 5 outlines a useful variant of the new 2-wire TIA protocol where both master and slave can use standard digital IO ports. This better suits microprocessor to microprocessor communication.

Finally in chapter 6 the 2-wire TIA protocol has been used as the basis of a novel microprocessor debugger. It helps to overcome two key problems associated with classic monitor ROM debuggers : the need for a dedicated serial data link and code burden of the interference with the application. This has been used on a number of microprocessor projects and found to be extremely useful.

These contributions have been the basis of two published conference papers (Radcliffe and Yu 2006a, Radcliffe and Yu 2006b), and a submitted journal article (Radcliffe and Yu 2006c). The innovation in simulation techniques, and details of the microprocessor debugger, may well be the basis of more papers.

2 Literature Survey : Communications Category Identification

***Overview** : this chapter first considers the ideal attributes of a low cost communications protocol for low end microprocessors. It is argued that these attributes constitute a new category of communications that will be called Time Independent Asynchronous (TIA) communications and that this category has a more general application than just microprocessor communications.*
The literature is then examined to find existing communications protocols and compare them against TIA attributes. While some TIA systems are found none of them are ideally suited to the target application of low end microprocessor communications.

The aim of this thesis is to develop a superior communications protocol for Low End Microprocessors (LEMs) that, for the purposes of this thesis, are typically a single chip device with not more than 128 kB of memory and may have as low as 500 bytes of memory. The first step along this research path is to investigate the current state of the art. This task is made a little difficult by the sheer number of communications protocols that have been used over the years and it would be easy to miss the most suitable LEM option within the vast sea. It is also difficult to detect if the existing options are all deficient in some way and a new method is needed.

This chapter will use a structured problem solving approach to help minimize the problems just identified. Section 2.1 identifies the attributes of the ideal LEM communication method.
It is shown that no existing communication category suits LEM communications and so a new category is proposed. Section 2.2 examines existing communications methods that fall within this new communications category. Section 2.3 will present a summary of the main findings of this chapter.

2.1 Communications Attributes and Categories

The ideal communications system for Low End Microprocessors (LEMs) must take into account the environment where LEMs are used. The main application is in consumer and industrial products which are usually price sensitive. For example, the 2005 7-series BMW and S-Class Mercedes each contain about 100 processors (Turley 2002). According to Garner research in 2002 the average American home had 200 microprocessors (Christ et al 2002). Many products and systems contain multiple microprocessors and inter-processor communications represents a financial cost (Lee et al 2004). Any method that can reduce the cost of the communications link has the potential to reduce the cost of the overall product.

Given such a usage environment the criterion for the ideal LEM communications method should include at least (Radcliffe and Yu 2006b)-

- *Minimal cost* : Communications links do not come for free and can represent a considerable cost in a mass produced product. Clearly the cost of a communications link should be minimized.
  For example the Microchip corporation (2006) product selector guide shows the cheapest microprocessor with a serial link is the PIC16F631 which costs US$0.94. A similar chip without the link is the PIC16F54 that costs US$0.44. If the PIC16F54 microprocessor could be used instead of the PIC16F631 then a cost saving of US$0.50 would be a achieved, a very desirable goal in a mass produced consumer product. The step to two serial communications links is even higher. The cheapest Microchip solution would be the 16F687 which costs $1.14 and contains an EUSART and I2C interface.
  The ideal communications protocol should not require specialized peripherals that increase the cost of the microprocessor.
- *Minimal hardware requirements* : regardless of the use of specialized peripherals the receive side of a communication link needs to be serviced within a certain time frame otherwise data overrun will occur – if a received data element is not read in time then the next received data element will overwrite the the first, which is lost. This places a hard upper limit on the response time of the application.
  This can effect the hardware selected for the application-
  - Faster hardware may be chosen to ensure the software will run faster enough to service communications needs as well as the application.

  - Reception may be handled by interrupt which may require additional hardware in the microprocessor. The receive interrupt may cause timing jitter in any application software and so disturb its operation.

  The ideal communications protocol will not require more powerful or extra hardware from a microprocessor. The only way to avoid this requirement is that the communications protocol should not contain any response time requirements.

- *Minimal software effects* : the problem of a hard upper limit on the receive data response time can also have an effect on the software architecture. The software architecture and even language used may need to be altered to help guarantee the response time.
  The ideal communications protocol should not have any timing requirements thus servicing the data reception can take place in the background and so have minimal interference with the application code and the software architecture.

The foregoing discussion has highlighted two key properties of the ideal communication protocol for low end microprocessors. First, the protocol should not make any demands on hardware ; no specialized peripherals, no interrupts, and no faster processors. Second, the protocol should not have any timing requirements.

The latter point above is of particular interest. What category of communication protocol has no timing requirements? The categories of physical layer communication protocols would appear to be (Freeman 2001)-

- *Synchronou*s : a master clock is used by all hosts to control communications, or the clocking information is included in the bit stream. Representative examples are the read and write signals on a micro-controller bus such as the Intel 8051 family, and manchester encoding (Freeman 2001).
  There are timing requirements so this category does not suit our purpose.
- *Plesiochronous* : communicating parties have their own clocks which are very close but not identical in frequency. An example is the DS1 and E1 families of PCM digital multiplexers which are often called the Plesiochronous Digital Hierarchy (PDH) (Freeman 2001).
  There are timing requirements so this category does not suit our purpose.

- *Asynchronous* : systems have their own independent clocks or timing circuits which drive the communications process. An example is the Centronics standard for the PC parallel printer port (Durda 2004) now defined by the IEEE-1284 standard (IEEE Std 1284-1994), or the RS232 serial communications standard (TIA/EIA-232-F Standard 1997).
  There are timing requirements so this category does not suit our purpose.

New category (TIA). None of the classic communications protocols satisfy the key requirement for low end microprocessors – no timing requirements. This leads to the need of a new category called Time Independent Asynchronous (TIA) communications (Radcliffe 2006a, Radcliffe 2006b). This name indicates that the protocol allows the communicating systems to be independent (asynchronous) and in addition there are no timing requirements in the protocol (Time Independent). There are no timing relationships between signals, or on any one signal, and only signal order is important. Unlike Time Free communications (Le Lann 2003), order of signals is important and gross error conditions such as a host lockup may be corrected with time-outs.

This new category has some interesting properties as already discussed. It can make minimum demands on hardware and software and thus has the potential to reduce the costs of a communication link. Such properties may be useful in areas other than low end microprocessor communication.

A TIA communications system where only signal order is important has some interesting advantages (Radcliffe and Yu 2006b)-

- *Arbitrary speed* : A host may work as fast or as slow as it likes, and change speeds arbitrarily without causing problems. This can be useful when a host can be interrupted with application or operating system tasks, or has low processing power. Another use is when a system is intermittently unavailable as can be found in some power saving strategies. High speed communications may occur when both communicating hosts decide to dedicate CPU time to the communications activity.
- *No response time requirements* : The response time to signal changes does not matter and as a result a host does not have to make any guarantees about response time to a signal change. This enables TIA communications to be run at background level of program execution which often has considerable CPU time available but on an intermittent basis. Time critical

applications or operating system programs are only marginally effected by the TIA communications running in the background.

- *Just IO pins* : TIA can be efficiently implemented with general IO pins which is useful when there is no dedicated communications or special purpose hardware available.
- *Media distortion* : TIA can cope with a communications medium that distorts timing though not to the stage where signals are delivered out of order.
- *No data overrun* of the communications system is possible as a host can simply stop receiving until it is ready to resume.

A TIA communications system may not be appropriate when microprocessor requirements have forced the selection of a microprocessor which has a free serial link. Another situation where TIA may not be appropriate is when high speed communications is required and CPU time is almost totally dedicated to the application.

Having justified a new category of physical layer communication protocol, a range of questions arise-

- Are there any existing protocols that fall within this category?
- If there are any such protocols can they be improved upon, particular to suit our target of low end microprocessors?

These questions will be answered in the following section.

This sub section has proposed a new category of communications for the physical layer and listed its attributes. The usefulness of this activity is immediately evident as the consolidated list of attributes suggests practical applications of TIA protocols. Several ideas have been presented in the form of a concept map in the illustration below. Key TIA attributes are linked to their consequences and eventually an application that needs a communication link. This map is far from comprehensive and many other potential applications exist.

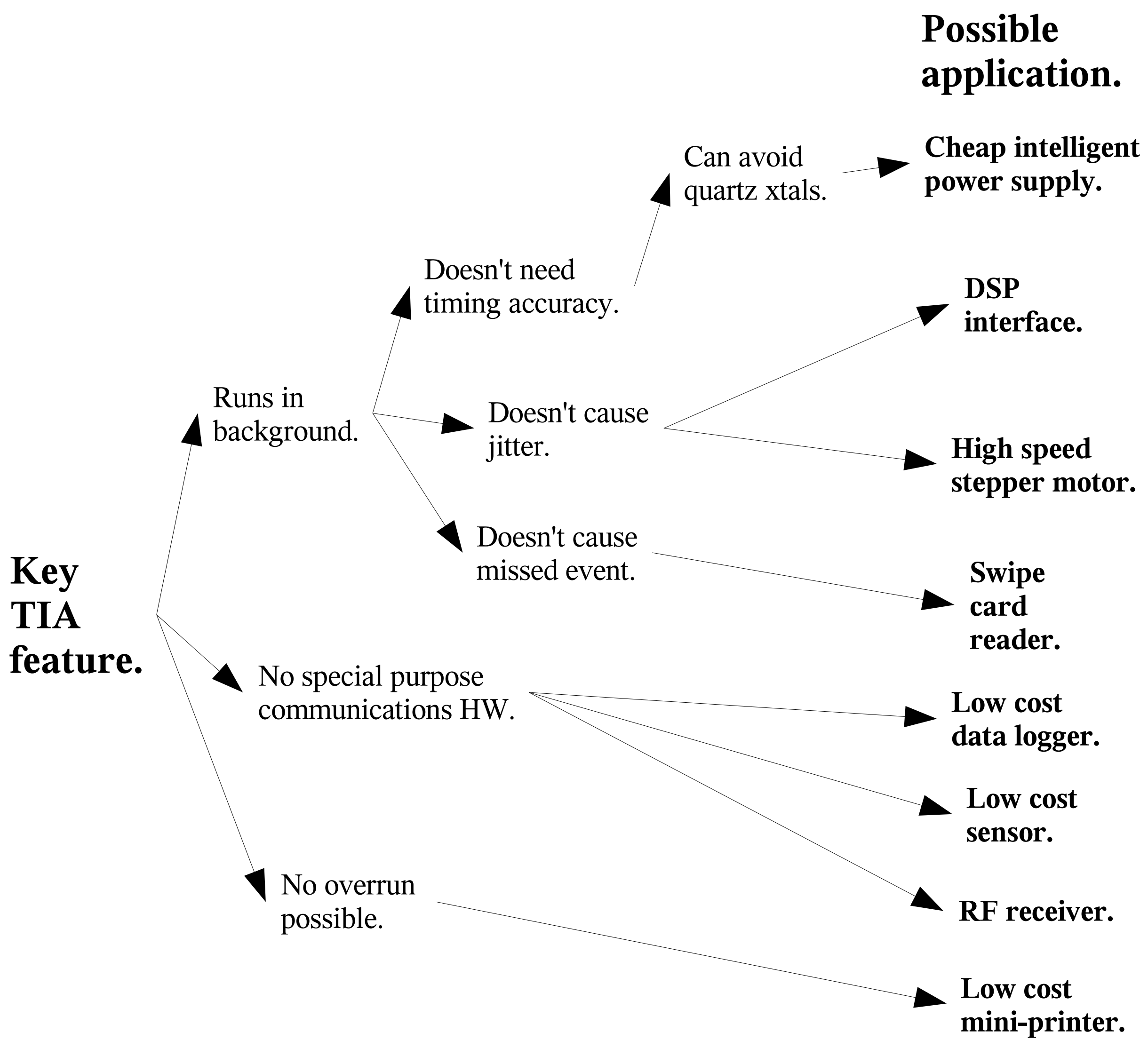


Illustration 2.1: Concept Map of TIA Features to Applications

2.2 Existing Time Independent Asynchronous Protocols

The new Time Independent Asynchronous communications category is most likely to contain protocols that will reduce costs for Low End Microprocessor based systems because the properties of this category match the needs of LEMs. The discovery of the best existing TIA protocol can start by first rejecting protocols which are not TIA in nature, and then investigating in more detail the TIA protocols left.

As a starting point all synchronous and plesiochronous protocols can be ignored as these clearly have timing requirements. The asynchronous category may well contain TIA protocols and must be considered in detail.

2.2.1 Non-TIA Protocols

The communications protocols in this subsection are asynchronous protocols that may appear TIA on cursory inspection, but closer inspection shows timing dependencies that are essential to operation. Such protocols are not TIA in nature and so will be excluded from deeper analysis.

RS232 (TIA/EIA-232-F Standard 1997) is asynchronous but each bit has a fix period, thus there is a time dependency and the protocol cannot be TIA in nature.

I2C was developed by Moelands and Schutte(1982) for Philips corporation as a fast and cheap communications method for embedded systems. The I2C bus specification (2000) shows the basic bit transfer is based around two lines, SCL which provides a clock signal and SDA which handles data. The diagram opposite is a section of figure 6 from the standard. When a bus master sends data the data destination is assumed to be ready and to be able to cope with a fixed clock speed on SCL. Clearly there are timing dependencies and the system is not TIA in nature.

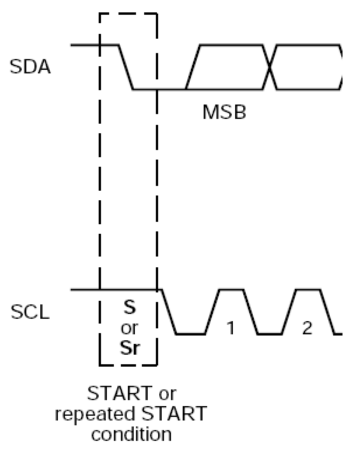


Illustration 2.2: Section of I2C Waveform

SMBus (1995) is a 2 wire bus developed by Intel for the purposes of battery management. It is very close to I2C and so is not TIA in nature.

SPI (Serial Peripheral Interface) was developed by Motorola (now FreeScale) (AN-991 1994), a subset called Microwire was developed by National Semiconductor. SPI and its derivatives tend to be cheap to implement as they take up little silicon area on a chip. There appears to be no accepted standard and many variations can be seen between companies such as Microchip, Atmel, FreeScale, Maxim/Dallas, Zilog, and Texas Instruments.

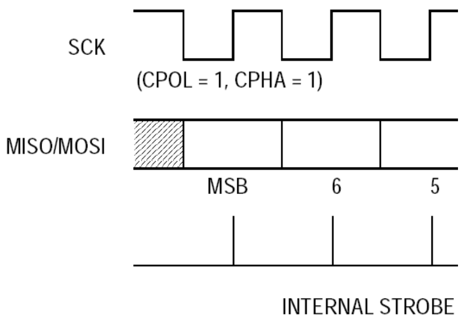


Illustration 2.3: SPI Timing from Freescale AN-991

The physical layer is a 3 wire system with Serial clock, Serial data in and out (MOSI, MISO). A slave device must read or write data within a set time after a clock edge. There is a timing limitation and so the system cannot be TIA in nature.

I2S was developed by Philips Semiconductors (1996) to handle audio data transfer on 3 wires. It is closer to SPI than I2C but like both it has a fixed master clock. The slave must be able to respond within the clock cycle and so has timing limitations, and thus I2S cannot be TIA in nature.

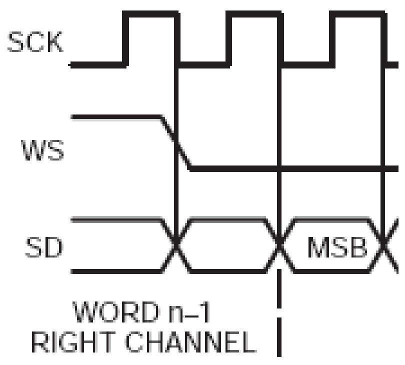


Illustration 2.4: I2S Waveform from Philips Specification

The Dallas one-wire protocol was developed by Dallas corporation (now merged with Maxim) to allow data and power to be transferred over the one wire. The Maxim web site http://www.maxim-ic.com/products/ibutton/ contains an extensive list of documents and specifications.

The one wire protocol is intended for e-commerce applications and includes error detection, encryption, and file transfer. The basic waveforms have many time dependencies and so the system is not TIA in nature.

The CAN bus was developed by Bosch to support automotive applications and has been accepted as an ISO standard ISO-11898. It is protected under patent and developers must seek a license from Bosch (Bosch 2006). While it is single wire in nature the physical layer is implemented as a differential transmission system thus using two physical wires. The bus is a little like RS-232 with a start bit and timing on all bits (Bosch 1991). CAN is thus not TIA in nature.

LinBus (2006) is similar to the CAN bus in nature and is supported by a variety of car manufacturers including Audi, Volvo, and Daimler-Chrysler. Again there are timing requirements and so the system is not TIA in nature.

USB (Universal Serial Bus) is specified by the USB Implementers Forum (http://www.usb.org/home).which that has members such as Intel, NEC, Microsoft, Philips, and Hewlett Packard. The USB 2.0 specification can be downloaded from http://www.usb.org/developers/docs/ . The standard is dated 2000 but there are several errata and extension documents dated up to 2006.

The basic data transmission is differential data transfer using 2 wires and NRZI encoding with bit stuffing. NRZI has timing dependencies and so the system is not TIA in nature.

Ethernet was conceived to work over a single coaxial cable and is defined by the IEEE-802 series of standards. As each new version of Ethernet has been developed the physical layer encoding has changed. The early Ethernet of the 10base family use manchester encoding. The 100base family used PAM-3, PAM-5, MLT-3 and NRZI encoding. The 1000base family have used 8B10B, NRZ and PAM-5 encoding.

All of these modulation techniques have time dependencies and so none are TIA in nature.

Firewire was developed by Apple Computers in the 1990s based on the early work of the IEEE-1394 committee. The standards range from IEEE-1394-1995 to IEEE-1394.1-2004
IEEE-1394 and 1394a use data strobe encoding using two wires and requires clock recovery at the receiver. The faster IEEE-1394b uses 8B/10B encoding which again has timing dependences. All Firewire systems thus have timing dependencies and are not TIA in nature.

MIL-STD-1553 (1989) defines a balanced serial bus. It uses manchester encoding and so is synchronous in nature and has timing dependencies. It is not TIA in nature.

Nasu et al (1989) from NEC patent a two wire system for exchanging data but this has time dependencies and is so not TIA in nature.

2.2.2 Existing TIA Protocols

Subsection 2.2.1 showed that many popular protocols are not TIA in nature and so will not be considered. This subsection identifies asynchronous protocols that either are TIA in nature or can be modified to become TIA in nature. These protocols are examined in more detail as they may be useful for low cost communications to or between Low End Microprocessors. What is found here will also act as a benchmark in that this thesis should later propose a superior TIA protocol. This is achieved in chapter 4 where a 2-wire TIA protocol is proposed.

Centronics interfaces (IEEE Std 1284-1994) defines the parallel printer port seen on many personal computers. There are three modes the SPP mode (the backward compatibility mode), ECP and EPP modes.

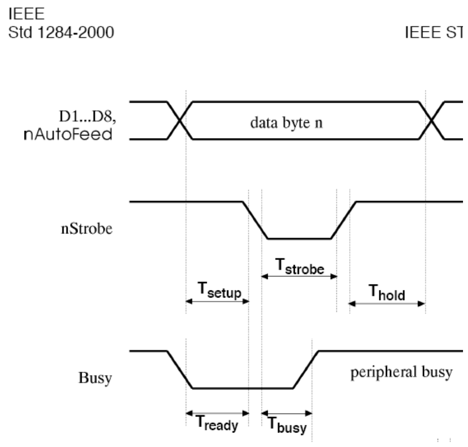


Illustration 2.5: IEEE-1284 Compatibility Mode Waveforms

At first glance the waveforms for the SPP mode appear to be TIA in nature (see illustration opposite from IEEE-1284) but the standard defines the strobe pulse as being 0.5 us wide, and other timing limitations, thus the system is not TIA in nature. SPP mode could be TIA if the timing requirements were removed. The sequence of signals on busy, strobe, and acknowledge could transmit data with no time requirements. Byte wide data transfer could be achieved using some 11 wires but this represents a huge resource demand from a low end microprocessor and so is not a feasible TIA solution. Bit transfer using this approach would require 4 wires for one way transfer.

EPP and ECP modes all have some timing requirements (se table 5 of the IEEE-1284 standard) and are so non-TIA in nature.

GPIB (ANSI/IEEE-488) was developed by Hewlett Packard to provide an asynchronous data bus between its instruments. It was accepted by the IEEE as a standard in 1987 (IEEE-488.1-1987) with updated versions in 1992, 2003, and 2004. It requires 8 wires for the data bus, 3 for byte transfer, and another 5 for general interface management. Annex B of IEEE_488.1-2004 shows the basic data transfer for interlocked handshaking (see the illustration opposite). There are no timing requirements at all on this waveform which does match TIA requirements. Section 5.8 of the standard details timing responses and most of these define settling times for the media rather than any time dependencies intrinsic to the protocol. While the byte transfer has no time dependencies higher functions such as multi-line transfer and non-interlocked states (see section 4.3.2 of the standard) do having timing limitations.

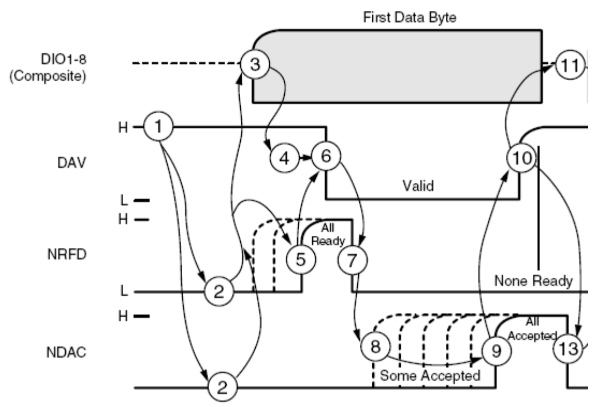


Illustration 2.6: IEEE-488 Byte Transfer

The IEEE-488 protocol is TIA for byte transfer but not some of the higher functions. The system requires 11 wires for bytes transfer though this could be cut to 4 wires for bit transfer. Another feature of IEEE-4888 is that it is a true bus system that, at byte transfer level, can have one master and multiple slaves.

DEC Unibus is an old asynchronous bus that was used on the old DEC computers (DEC Unibus Specification, 1979). It used a very large number of wires to accomplish data transfer as so while it is TIA in nature it is not suitable for low end microprocessors.

The VME bus (VMEbus Standards and Specifications) is define by ANSI/IEEE standard IEEE-1014-1987. There is also a standard from VITA (VMEbus International Trade Association) VITA 1.1-1997 (VME64x; VME64 Extensions). Davis (2005) has an excellent list of the standards applicable to VME and some basic descriptions of VME operation. Kishinevsky's paper (1998) gives an insight into VME bus operation as a controller is synthesize using Petri nets.

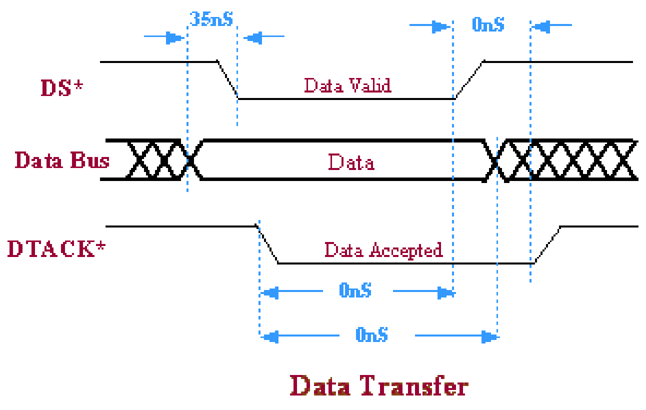


Illustration 2.7: VME Bus according to Davis.

The VME bus is intended to be a scalable backplane bus interface with a bus controller that manages the bus, and master and slave units. There is a minor time dependency in that for basic data transfer – the master must place data on the Data Transfer Bus at least 35nS before bringing one or both of the Data Strobes low. This is really to allow settling and setup time to ensure the first signal is recognized at the destination before the second signal arrives. This timing demand ensures that the basis of TIA, correct signal order, is not violated.

VME requires two strobe wires, plus DTACK, plus data wires making a minimum of 4 wires for basic bidirectional bit transfer. Basic VME data transfer is TIA in nature.

VLSI systems have become so large that asynchronous data transfer is needed to go across a large chip (Kessels 2005). A number of workers have published systems that are TIA in nature. Most of these systems can be reduced to 3 wires for single bit data transfer between a master and a slave-

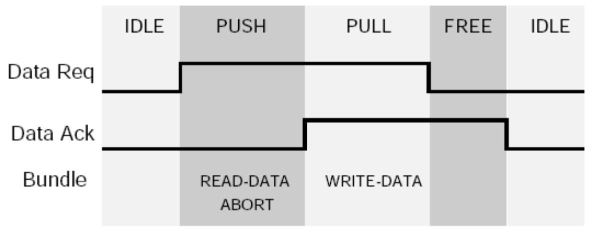


Illustration 2.8: Banbridge & Furber 3-Wire TIA

- Bainbridge and Furber (1998) describe a 3 wire 4 phase bidirectional system that allows bit transfer between a master and a slave.
- Berkel and Bink (1996) describe a 2 wire system but there are timing limitations on pulses and charge must be stored on the line thus the system is not TIA in nature.
- Furber et al (1999) describe a 3 wire system they describe as "Two-phase bundled-data communication." that is TIA in nature and capable of bidirectional bit transfer between a master and slave.
- Jung et al (2003) describe a 3 wire high performance asynchronous bus but it only facilitates unidirectional data transfer.
- Molina et al (1996) describe a bus system that at first looks to be two wire but on closer inspection is a 4 wire bus.
- Peters and Berkel (1996) describe a range of 3 wire 4 phase systems that deliver bidirectional data transfer between a master and slave.
- Takahashi and Hanyu (2004) describe a multi-valued logic solution but as multi-valued logic this is difficult to implement between low end microprocessors it is deemed not to be suitable for the purposes of this thesis.
- Teifel (2003) describe a 3 wire system and model it with a simple state diagram.

- Yakovlev et al (2004) have implemented a 4 wire system that allows bidirectional bit transfer between master and slave. Yakovlev describes this protocol as "self timed" and its lack of time dependencies make it TIA in nature.

The best these VLSI methods can offer is a 3 wire bidirectional bit transfer that is TIA in nature.

Dual rail and N-of-M code logic systems are potentially TIA in nature but such systems increase the number of wires required to send data and so do not represent an economic solution where wires at a premium and delay insensitivity is not a requirement (Lloyd 2001; Nigussi 2006; Yakovlev 2004).

2.3 Summary and Conclusions

Through the literature survey it has been found that no existing physical layer communications category satisfies the ideal needs of Low End Microprocessor communications. This prompts us to propose a new communications category called Time Independent Asynchronous communications to fill this void. The properties of the TIA are sufficiently different from synchronous, plesiochronous and asynchronous communications to justify the creation of this new communications category.

It has been shown that a number of existing asynchronous protocols satisfy the constraints of a TIA system. These range from IEEE standards such as IEEE-488, to data transfer across VLSI chips. Some are point to point communications methods whereas others are master-slave bus systems. The aim of this thesis is to find a TIA system suitable for Low End Microprocessors. The TIA protocol with the lowest resource requirements found in the literature is a three wire system (plus ground) that allows bidirectional bit transfer. This requires 3 pins on a microprocessor, a significant resource constraint on a small microprocessor that may only have 6 IO pins in total or most of its pins already dedicated to application tasks.

The ideal TIA communications system for a Low End Microprocessor would require only one wire but such a system seems improbable. If a TIA system using only two wires (and thus 2 IO pins) were possible then it would be preferred over the methods found in the literature survey. Chapter 4 of this thesis describes exactly this outcome, a new and novel 2-wire TIA system.

The VME standard highlighted an important issue that will effect practical implementation of a TIA system. If signal order is to be preserved at the destination then, given real media, each signal change emanating from a host must be separated in time from the next output by a period that allows for settling time on the real media. This will ensure that signals can be recognized in their correct order at their destination.

3 Literature Survey : Protocol Verification & Modeling Issues

***Overview** : This chapter examines how a TIA protocol should be modelled and verified. First the protocol attributes to be verified are identified. Next a brief survey of modelling methods is offered. The preferred modelling candidate, Signal Transition Graphs, is shown to have problems modelling software based systems and so a new variant STG For Threads is proposed and shown to overcome these problems. Finally the issue of state explosion in the reachability tree is considered and pruning algorithms identified.*

Chapter 2 identified a new communications protocol category named Time Independent Asynchronous communications and that TIA is an attractive, low cost communications method for Low End Microprocessor systems. It foreshadowed that a new 2-wire TIA protocol had been developed but before that can be explored two key things must be decided-

- What attributes of a protocol should be analyzed?
- What modelling method is most suited given the attributes identified?

This chapter performs a literature survey to uncover the answers to these questions.

This chapter is structured as follows. Section 3.1 examines the issues that must be considered when verifying a protocol. Section 3.2 examines modelling methods appropriate to a TIA protocol. It will be shown that Signal Transition Graphs are superior to Finite State Machines. It is further shown that traditional STGs cannot properly model systems implemented in software and so a modification called STG For Threads (STG-FT) is developed and shown to overcome the problems identified.

3.1 Protocol Properties to Verify

The literature is rich with examples of protocol analysis, particularly in the 1970s and 1980s and many of the most interesting papers date from this time. These papers discuss the properties of a protocol that are practically important and must be verified before the protocol is used in practice. Protocol properties can be proven by simulation and by practical implementation.

This subsection starts by considering the confusing terminology of verification and validation and chooses to use the term verification. Key verification issues in the literature are then identified.

Verification or Validation? Three terms appear to capture the issues that arise in the literature : validation, verification, and performance.

The terms validation and verification are often used interchangeably and many authors simply accept them as synonymous, for example (Yuang 1988). The IEEE dictionary of computing terms (IEEE-610 1991) defines the terms as follows.

> **validation.** The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. ***Contrast with:*** **verification.**
>
> **verification.** (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. ***Contrast with:*** **validation.**
>
> (2) Formal proof of program correctness.

Validation would appear to be a subset of verification in that validation is applied at the end of the development cycle on a final product whereas verification can mean the same thing or be applied to an intermediate result.

Dictionary definitions such as the Oxford Collins Pocket Dictionary 1990 match this view and see verification applying to statements and facts whereas validation applies to larger systems such as documents.

The overlap between verification and validation is undesirable but there is little to be gained from trying to separate them and so validation and verification are considered to be interchangeable terms. In this thesis verification will be used as it encompasses testing of both intermediate and final products.

Discussion of verification and performance will be limited by the scope of the 2-wire TIA system proposed by this author. The protocol is concerned with the sending and receiving of bits over the transmission media and as such sits at the physical layer in the OSI model (Freeman 2001). The protocol does not provide error detection and recovery, or manage the establishment, maintenance and release of connections and so fulfills no Data Link Layer responsibilities.

TIA systems provide a form of flow control, which according to the OSI model is a concern of the Transport Layer(level 4). This is a by product of the TIA definition and TIA does not manage the full flow control issue.

Verification issues : Given the author's 2-wire protocol then the literature suggest several important issues that must be verified-

- *Abnormal state recovery* (Merlin 1979) requires that any abnormal system state revert to a working condition in a finite number of steps or time.
- *Correctness* : the implemented protocol will work as per the specification (Merlin 1979; Simpson 1992; Yakovlev 2004).
- *Deadlock* is where the protocol can enter a state where nothing is happening, each party is waiting for another party to act (Merlin 1979; Yuang 1988). A protocol must not exhibit any deadlock.
- *Livelock*, also called dynamic deadlock (Yuang 1988), is where parties are exchanging information but the services the protocol is supposed to deliver are stalled (Bochmann 1978). Typically the exchange is a fixed, unending cycle. A protocol must not exhibit any livelock.
- *Liveness* (Bochmann 1978; Peterson 1981; Kessels 2005) can have a variety of meanings but they all have a commonality in that the liveness refers to additional properties beyond correctness. For example Peterson uses liveness as a synonym for a lack of deadlock. The 2-wire TIA system has no particular liveness requirements other than an absence of deadlock and livelock and so in future sections liveness will not be listed separately.
- *Safeness* as defined in Petri nets (Peterson 1981) exists when the number of tokens in a place can never exceed one. This implies that a place can be implemented in a flip-flop or be a unique state in a state machine. A protocol must be safe to ensure it can be implemented and be correct.

- *Unspecified reception* is where an unexpected input is received (Yuang 1988). Since the design did not consider this input the system may respond in an unacceptable manner. A protocol must cope with any input for all its states.
  At the physical level, to which the 2-wire TIA system belongs, an "unspecified reception" signal can be a simple pulse, which can be created by EMI (Electro-Magnetic Interference). A physical layer protocol should thus examine the effect of an unwanted pulse in every state of the system and ensure that at minimum no livelock or deadlock results. Data error correction may be left to higher levels of the protocol.

The performance of a protocol can be measured in several ways-

- *Throughput of data* given different host configurations.
  The 2-wire TIA method could be used in a vast variety of situations. In this thesis only the performance a few useful applications for 2-wire TIA will be examined.
- *Error rate* for the physical layer is perhaps best measured as BER (Bit Error Rate).
  The TIA 2-wire system should exhibit zero errors due to the handshaking and intrinsic flow control. The occurrence of errors can be examined in normal operation and when one or both hosts are stressed.
- *Speed variance* is the variation in delivery times and can matter for such purposes as audio or video transfer.
  This is not particular relevant to a TIA method where a key goal is to be time insensitive.

The literature identifies two key places in the development cycle to verify a protocol. The first when the protocol specification is complete then a protocol model can be formed and test by methods such as simulation. The second point for verification is when a real system can be created and thoroughly tested.

3.1.1 Specification Level Verification

Foregoing discussion in this section has outlined a variety of protocol issues to be verified and that the verification can take place at specification level and implementation level. Table 3.1 below shows which verification issues may be addressed by modeling and simulation performed during the specification phase of a protocol. This table will provide a guide to section 3.2 which selects modelling methods, and section 4.3 where a simulator is implemented and the simulation results reported.

Verification Issue	Modeling & Simulation Can Verify?
Abnormal Recovery (Including self synchronization.)	Yes : the system could be set to everyone of the possible system states (a cross product of all subsystem states). Every possible execution path could then be investigated to ensure the system recovers. Data errors may occur but the system should recover to a workable condition in every case.
Correctness	Yes : every possible system path from all possible initial system states should result in the correct transfer of data and terminate with slave and master state machines back to an appropriate state.
Deadlock	Yes : every possible system path from all possible initial system states should show no deadlock.
Livelock	Yes : every possible system path from all possible initial states should show no infinite looping. To ensure a finite simulation time the simulation is aborted if the system state trace enters a repetitive state.
Performance	Yes : performance is dependent on the nature of the hosts and the implementation of the hardware and software. This can be modeled before implementation if the hardware and software are well characterized.
Safeness	Yes : Modeling can check for safeness and so ensure the system

Verification Issue	**Modeling & Simulation Can Verify?**
	can be implemented.
Unspecified and invalid signals.	Yes : every system state can be driven with every input from the input alphabet. While data errors may occur the system should return to a useful state and not exhibit livelock or deadlock. If the input alphabet is intrinsic to the system state (as is the case with TIA analysis) then this test becomes identical to abnormal recovery.

Table 3.1: Modeling & Simulation Analysis Capabilities

3.1.2 Implementation Level Verification

Once modeling and simulation have confirmed the protocol is viable then a real physical system can be implemented and verified. This testing may be called conformance testing as it concentrates on checking the system conforms to the specification and simulation results.

The communications and software engineering literature suggests a huge variety of test methods. The ones listed below are particularly relevant to verifying the issues discussed in this section.

- *Combinations and permutations* : all combinations and permutations of initial states and data should be tested.
- *Correctness & assertion checking* : assertions are statements about individual inputs, outputs, internal data records and actions, and any relationship between these items (Chen et al 1978). Communications software could be instrumented with assertions (have code added) and any violations reported. The assertions that matter are that data is transferred properly and that livelock or deadlock do not occur. This corresponds to correctness for the 2-wire system.
- *Destructive testing* also called death testing attempts to apply inputs which will cause the system to fail. The failure and recovery should not violate specification requirements. In the 2-wire application destructive testing is limited to load testing and EMI style testing.

- *Driver testing* : a test system applies test vectors to the application system, observes the result, and ensures the application system is working as per the model and specification. In the 2-wire TIA system the master and slave form a natural driver test system.
- *Forged signals* can test unexpected reception and unspecified reception. Data errors would be expected but livelock and deadlock should not occur. Partial W-Testing (see below) can be achieved with forged signals to the data lines and the reset of subsystem elements.
  EMI (Electromagnetic Interference) is a common source of forged signals at the physical layer as it adds invalid edges and pulses to a system.
- *Load testing* : is simply to drive the protocol with a variety of data rates and check it works as per specifications. This may be achieved by varying the speed of one host and checking the other works correctly. Load testing should examine extremes and attempt to become destructive testing.
- *Observer testing* : simply observe a working system and compare the operation to the model and specification. The master-slave arrangement provides a natural observer testing situation.
- *Platform variation testing* : software products, including protocol implementations are notorious for working with one configuration of software and hardware but not with another. Often this results in the product not working, but sometimes an issue such as deadlock is not exhibited on one platform, but is on another.
  The 2-wire TIA system could be tested with a variety of hosts. Variation is possible even with only one master and one slave by varying key properties such as clock speed and processor availability.
- *Random testing* : applies some form of randomness to a system and then checks key assertions hold. A 2-wire TIA system could have random data, and could have either host stopped and started in a random fashion.
- *T, U, D, W, Wp (partial W), UTS and DS testing* (Chow 1978) (Sidhu and Leung 1989) (Fujiwara et al 1991) : These methods are primarily for testing state machine implementations. They all concentrate on determining an input sequence that causes an implementation of a state machine to enter a known condition and so be verified against the state machine design.

Chanson and Zhu (1993) propose a method they call UTS (Unified Test Sequence) and make the point that when using extended FSMs (EFSMs) the extra information in the data records must also be taken into account.

These methods are excellent for testing that a real implementation will behave as expected but they do nothing to verify the protocol is free of livelock and deadlock and may not uncover livelock or deadlock in a state machine implementation. Table 3.2 below maps the verification issues to the preferred physical testing methods from the list above. These will be applied to a physical implementation of the 2-wire TIA system in chapter 4.

Verification Issue	**Conformance Test Methods**	
Abnormal Recovery (Including self synchronization.)	Forged Signals.	
Correctness	Platform Variation Random Testing	Combinations Load Testing
Deadlock	Platform Variation Random Testing Combinations	Forged Signals Load Testing
Livelock	Platform Variation Random Testing Combinations	Forged Signals Load Testing
Performance	Load Testing	Platform Variation
Safeness	Not relevant as specification level analysis has confirmed the system can be implemented.	
Unspecified and invalid signals.	Random Testing	Forged Signals

Table 3.2: Traceability from Verification Issue to Conformance Tests

This section has been very important because it has identified what a modelling method must be capable of achieving and so will guide the selection of the modelling method and any modelling tool used on 2-wire TIA. The linkage from simulation to implementation and testing is also clarified and will guide the test plans and methods that will be used in that phase.

3.2 Modelling Methods for TIA

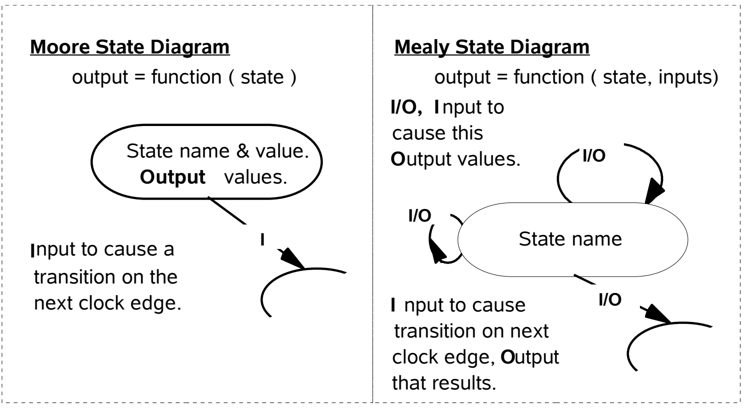
Section 3.1 in this chapter outlined key protocol attributes that must be investigated when ever a new protocol is proposed. Armed with this insight it is now sensible to examine a range of modelling methods and select one appropriate for the new 2-wire TIA protocol described in chapter 4.

The two methods that have most commonly been used to model protocols are Finite State Machines and the Petri net based Signal Transition Graph. These are discussed in some depth in this section and it is concluded that both are suitable but STGs are preferable. Other methods based on CSP (Kishinevsky 1998) have been proposed but FSMs and STG appear to be the preferred methods in the literature.

This section also finds that STGs can mis-model a system implemented in software and can miss livelock and correctness faults. A new extension called STG For Threads is developed to overcome these problems.

3.2.1 Finite State Machines (FSM) & SDL

State Machines have a long history. The first state machine is probably the Jacquard loom invented by Joseph Marie Jacard in 1801. It used holes punched in paste board to control the weaving of patterns in fabric. Each card represented a row and a chained sequence of cards were used to create a full pattern. Since that time state machines have been employed in other mechanical devices, electrical devices, digital electronics and finally software.


Moore State Diagram
output = function (state)
State name & value.
Output values.
I
Input to cause a transition on the next clock edge.
Mealy State Diagram
output = function (state, inputs)
I/O, Input to cause this Output values.
I/O
I/O
State name
I/O
I nput to cause transition on next clock edge, Output that results.
Illustration 3.1: Mealy & Moore State Diagrams

There are large variety of representational variants for state diagrams including the traditional Mealy and Moore state diagrams and Harel's (1987) state charts. State models tend to be preferred when a system can be modelled as a set of waiting conditions (states) with transitions between states that are triggered by events.

Many communication protocols can be modelled as a number of interacting state machines, for example IEEE-488 and IEEE-1284 standards. This fact alone would suggest FSMs are a serious contender as a modelling method for 2-wire TIA.

The Specification and Description Language (1992), ITU specification Z.100, is both a graphical design methodology and a language that can model a system of communicating FSMs. There are a number of advanced tools built around SDL and according to Minea (2002) "Formal verification is exhaustive, covering all possible system behaviors; it is also highly automatable."

If an FSM model of TIA is deemed to be the most suitable then modelling and verification using SDL should be seriously considered.

A number of authors comment on the state explosion issue whereby the number of states of the total system becomes very large (Merlin 1979; Kishinevsky 1998). Bochmann (1978) notes that a state machine may be required for each of the two hosts and the communications medium, particularly when a number of data packets are in flight. The total number of states in any simulation or analysis could be-

#System States = #host_1_states * #medium states * #host_2_states

In many cases state explosion is a serious problem and it is essential to use methods and a simulator that can cope with the problem, or reduce the size of the explosion.

FSMs appear to be entirely suitable to model the individual hosts and the entire 2-wire TIA system. Given the small number of states involved a brute force calculation of the entire reachability tree should be possible and provide the required protocol verification identified earlier in this thesis.

3.2.2 Signal Transition Graphs and Problems

Signal Transition Graphs are a variant of Petri nets particularly suited to asynchronous logic (Yakovlev 1992; Cortadella et al 2003). Vectors represent places and may have a circle to hold a token . Nodes represent transitions which are triggered by a rising (+) or falling (-) edge.

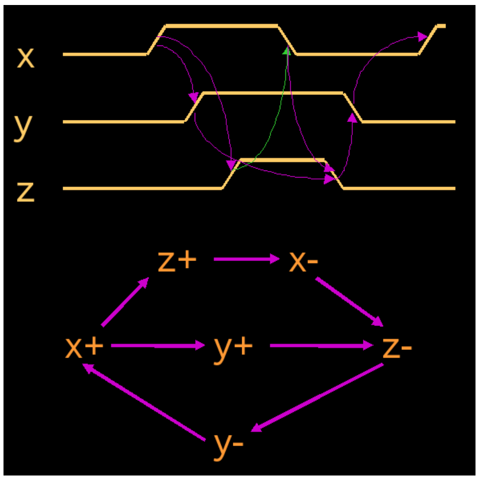


Illustration 3.2: STG Model from Cordatella et al (2003).

STGs that represent speed independent synthesizable circuits are a Petri net with the following constraints (Yakovlev 1992; Kishinevsky 1998; Cortadella et al 1998; Cortadella et al 2003)-

- Consistency where signals alternate between + and - (rising and falling edges) and transitions only fire on these binary triggers.
- Complete state coding : no two different markings (token positions) have identical values for all signals.
- Persistency : to quote Kishinevsky (1998) “ (a) no non-input signal transition can be disabled by another signal transition and (b) no input signal transition can be disabled by a non-input signal transition. The former ensures that no short glitches, known as hazards, can appear at the gate outputs, while the latter ensures that no hazards can occur at inputs of the device.”.

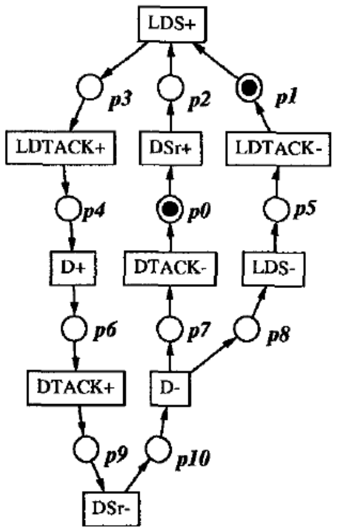


Illustration 3.3: STG for VME Read from Kishinevsky et al 1998.

Verification examples : STGs have been used to model and verify a variety of protocols for example the VME bus (Kishinevsky 1998) and a “Self-Timed Duplex Communication System” from Yakovlev et al (2004). Most of the VLSI TIA protocols discussed in the earlier literature survey on communication categories used some form of STG to verify their protocol.

Implementation : an STG can be mapped onto one or more Finite State Machines (FSMs) or State Graph for the purposes of implementation Cortadella et al (2003). Each State Graph must have Complete State Coding where each state is unique in order for implementation to be possible. The illustration opposite describes how the process by which the tool Petrify creates asynchronous logic.

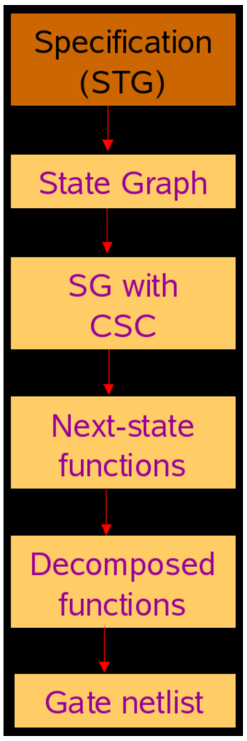


Illustration 3.4: STG Implementation from Cordatella et al (2003).

3.2.3 Signal Transition Graph For Threads

Signal Transition Graphs clearly work very well for asynchronous logic but 2-wire TIA is intended to be implemented in software. Will STG's accurately model software? This section reports on work by this author (Radcliffe and Yu 2006b) that shows the STG model can fail to model software driven implementations. A new form of STG called STG For Threads is proposed and is shown to correctly model the behavior of systems implemented with software.

This subsection will first identify three problems found with STGs when modelling software ; failure to model at an atomic level, state allocation issues, and problems with automatic translation to code. Next a three stage process is proposed to convert an STG into an STG-FT model. Finally an example is used to show how an STG model fails to model a software system but the STG-FT model correctly predicts operation.

Atomic behavior: There are differences at the atomic (indivisible) level of behavior between the asynchronous logic view of STGs and operation of a system implemented with several software threads.

- *Test dead zone* : For a software thread based system a transition that tests an input is polled; it occurs at an instant in time and is then not rechecked until the software thread, at some later time, again executes the test instruction. There is a timing dead zone in which a transition may be true but not acted upon. Other transitions implemented in this FSM, or other FSMs that come true later, may fire before the first transition because of these polling delays.
  Modeling must be at an atomic (indivisible) level of activity to properly capture the range of behaviors of the thread based system.
- *No parallel operation* : Asynchronous logic can respond very quickly to one of several possible transitions firing whereas a thread based system on one CPU can only consider one transition at a time. Multiple test transitions from a single STG place are polled : they are tested sequentially with dead zone periods between each test. The transition to fire may not be the first one to come true due to the polling sequence and delays. Again modeling must describe these tests at an atomic (indivisible) level to properly capture the range of behaviors of the system.
  (Note there is a limited exception to this rule. A processor can read multiple bits in one read, typically a byte but there is still the timing dead zone between these byte wide polled reads.)
- *No edge response* : STG transitions are triggered by rising or falling edges of signals. With the aid of dedicated hardware a software thread may detect edge changes but in general a software thread can only test for high or low.
  A software thread may also implement very complex tests where many lines of code result in a true or false result.

Software driven systems may violate the persistency requirement of STGs.

State allocation: minimizing of the number of states is an important topic in the design of asynchronous logic (Yakovlev 1992; Kishinevsky 1998; Cortadella 2003) and can result in fewer gates. State variables and other data are effectively free when using software threads as memory is inexpensive. A penalty of being too free with states and data comes when simulation is executed. If the reachability tree (signal graph) is to consider all possible system states then state explosion may occur. If the reachability tree is examining all possible operational paths from all initial states then states and variables may produce numerous initial states which can again cause a state explosion.

Automatic translation: Any translation requiring human judgement and expertise provides an opportunity for errors to creep in. Ideally the translation from one form to another should follow a strict set of rules that can be automated, or at least followed in an automated manner by a human being.

An STG may be mapped onto FSMs and this has been automated in tools such as Petrify (Cortadella 2006). Once an FSM is available it can be translated automatically into the code structure of a programming language. State tables are a convenient solution as the driver code, the state table, and the empty functions for action and test routines can be automatically generated. The user must add the code inside each action and test routine.

Standard STGs face several problems when applied to a system that must be implemented using several FSMs run by software threads-

- The splitting of an STG in to several FSMs is difficult to automate.
- The traditional labeling does not differentiate input and output signals which can cause readability problems and so encourages errors.
- The traditional poor naming conventions inhibits readability and also encourages errors.

A new and novel form of STGs - "STG For Threads" (STG-FT) is proposed here that allows STGs to accurately model a system implemented using software threads. The following three step process will convert an STG model into an STG-FT model which accurately models software behavior and creates an STG model that is isomorphic to an FSM implementation.

A Petri net may be modelled as a quadruple $N = (P, T, F, m_0)$, where P is a finite set of places, T is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is the set of flow relations that define a token moving from a place through transitions to another place, and m_0 is the initial marking (Cordatella 1998). For the purposes of this section individual transitions are $t \in T$, and individual places $p \in P$. A transition s may be part of T or not, $s : (s \in T) \vee (s \notin T)$, while *q* is a new place not in *P*, $q : q \notin P$. A transition *t* may be an input or output transition from a place, *to* is an output transition and *ti* is an input transition.

STG-FT Step 1 : Atomic modelling. The first set of constraints to apply to an STG model aim to enforce atomic level modelling so as to properly capture the behavior of a software thread based system as just discussed.

- *Atomic transition*: each transition $t \neq (s_1 ; q_1 ; s_2)$ must operate at one instant in time given the software implementation, or at least sufficiently fast that no part of the system can see any intermediate results. In other words all transitions must be decomposed into a triplet of transition-place-transition until no further decomposition is possible. Given that (;;;) denotes a sequence then $t \neq (s_1 ; q_1 ; s_2)$. Given that the system will be implemented in software this means that a transition can fire on the result of a single read (be a test transition) of an IO port, or always fire and perform a single write (be an action transition) to an IO port. The terms test and action have been chosen to avoid confusion with the terms input and output which define relationships between places and transitions.

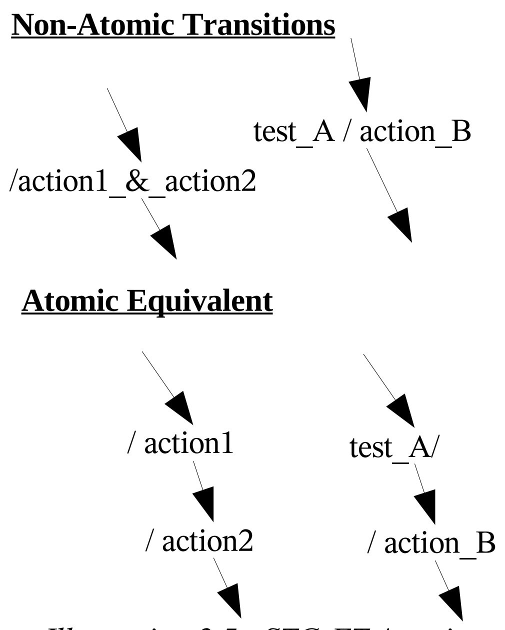


Illustration 3.5: STG-FT Atomic Modelling.

- *Atomic output transitions*: Consider the output transitions of a place. A place may have only one action output transition, or two output test transitions where the second output test transition has a firing rule the exact inverse of the first transition.

Given *toa* is an output action transition and *tot* is an output test transition for a place *p* then $p:(\exists! toa) \vee (tot, \backslash tot)$ where $\backslash tot$ is a transition with the inverse firing rule to transition tot. A place which violates this rule must be decomposed into a place-transition-place structure until the rule is satisfied. This again ensures the STG model captures the behavior of software at an atomic level.

- *An atomic output action transition*: this is a transition that causes an output or a change to data. An action transition always fires if there is a token in its input place. It is labeled as follows-

  / action_name

- *An atomic output test transition*: this is a transition that samples an input (or inputs) at one instant in time from the point of view of the system. An output test transition fires if the test is true and the input place contains a token. An atomic test is labeled as follows-

  atomic_test_name /

  A single test transition labeled test_A / action_B would not be atomic. The test must be broken into two atomic tests as per the illustration above.

- *Polled test transitions* : all test transitions implemented in software must properly model the timing dead zone (place C below) that occurs if the test is polled by software rather than being truly edge sensitive. The illustration below shows how this is modelled and a short hand notation for representing a polled test. Given the flow relation set *F* defined above where $f \in F$ then $f = \{p1, t1, t2, p2\} : p1 \neq p2$.

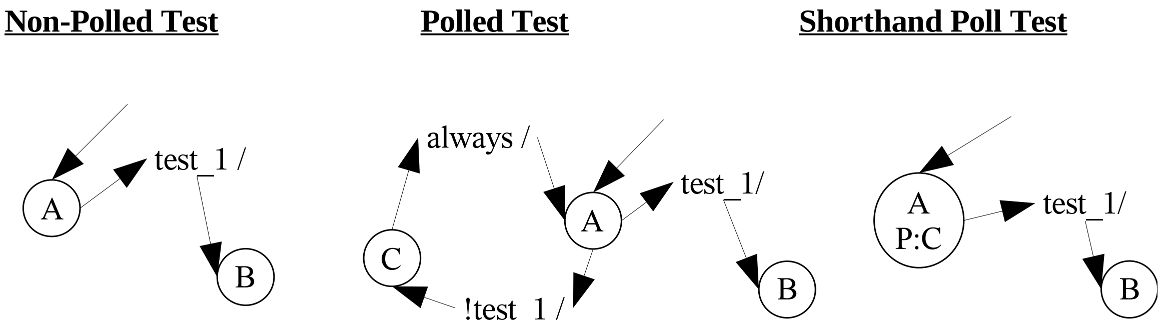


Illustration 3.6: Polled and Non-Polled Transition Tests

STG-FT Step 2 : FSM translation. The next set of constraints to apply to an STG model aim to make it easy to translate the STG-FT model into multiple FSMs.

- *FSM object*: an STG-FT representation may be implemented as several FSMs and so places and transitions are grouped into FSM objects each of which will implement one FSM. FSM objects, as with software objects, should have good encapsulation. Each FSM object should have a clear purpose and have a minimal and well defined interface to the rest of the system. A dotted ring may be used to denote an FSM object.
  Given the definitions above each FSM object *Ox* or *Oy* has a set of places $P_x, P_y \in P$ and a set of transitions $T_x, T_y \in T$. Given $\emptyset$ = null set and $x : x \neq y$ then $\forall P_x \cap P_y = \emptyset$ and $\forall T_x \cap T_y = \emptyset$.
- *Place = state*: a place (a vector under STGs which may contain a circle, and may show an initial marking) represents a single state in an FSM. Only one place in an FSM object may contain a token.
- *Place naming*: each FSM object has its own name. Each place has its own number and may have a descriptive name as well.
  For example X2 : wait_for_busy
  X is the FSM name, 2 is the place number within the FSM object, and wait_for_busy is an optional descriptive name.

STG-FT Step 3 : Inter-FSM communication. Once places and transitions are arranged into FSM objects there will be some places and transitions that link the FSM objects. In reality these communications between FSMs represent an IO pin, or information sent via a media. Such a link is eliminated by placing an action transition in the source that effects a variable which may be as simple as a boolean IO pin, or a complete data structure. There will be one or more test transitions in the destination FSM object that can test the resulting variable. None of these variables have any state transition behavior of their own and depend on the FSMs to change values. Variables may depend on each other and be modelled with boolean logic.

The eliminated linkage place and transition may be represented as a dotted arrow from the source action transition to the destination test transition and this represents a cause-effect relationship. The dotted arrow is an optional readability aid rather than an artefact to guide automatic implementation.

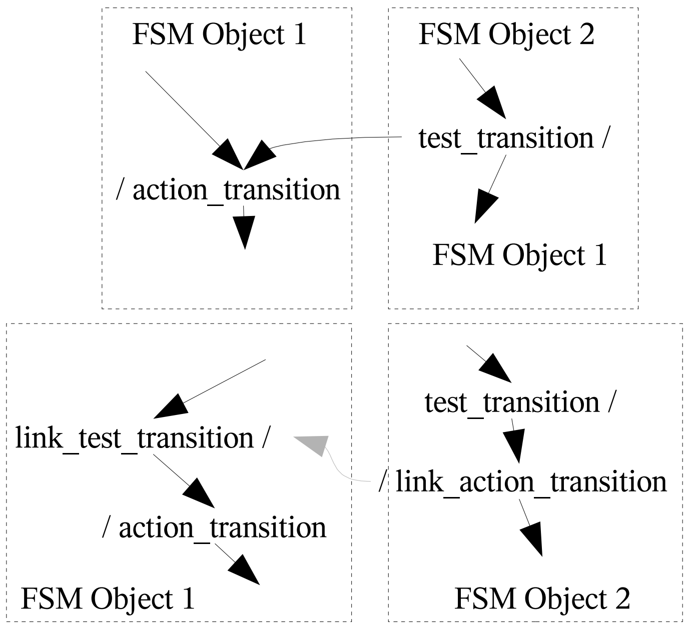


Illustration 3.7: STG-FT Link Transition Replacement

All variables must have a known value at the start of a simulation. If they can have different values then the simulation must be run for every possible data combination of all variables. The number of variables and the range of values they can take should be minimized to avoid state explosion.

FSM objects may be able to directly test the state variable of another FSM and so avoid the need for an extra variable. A dotted arrow may still be used to indicate a cause-effect relationship.

The STG-FT diagram that results after the three STG-FT conversion steps will accurately model a software system. Each FSM object iso-morphic to an FSM which means that implementation can proceed relatively automatically from FSM to code.

STG-FT example. This example will show how an STG which is not fully in STG-FT format, will not accurately model a software implemented system. The procedure for creating STG-FT is applied and the resulting model does correctly predict behavior.

Consider the waveforms opposite. The unit to be designed must read a clock and a wait signal and generate a data bit from some internal source. If the wait signal is high when the clock goes high then the data must stay stable until after the next rising clock edge where wait is low.

The first pass STG-FT is shown at the top of the illustration. The STG-FT with Sx labelling is the data source that must be created. The diagrams labelled with Cx (clock generator) and Wx (wait generator) implement signal generators that are only used in simulation and as such they can break the atomic operation rule. They allow for the generation and testing of one cycle of wait.

Section 4.3 details an STG-FT simulator which has been used to simulate this problem. The simulator examines the full reachability tree from every initial condition through every possible valid path across all state machines. This results in full detection of livelock, deadlock and correct or incorrect operation. The translation from STG-FT to software code in the simulator is automatic in nature except that the user must create the initial conditions, the code that implements each action and test transition, and the conditions which terminate the simulation.

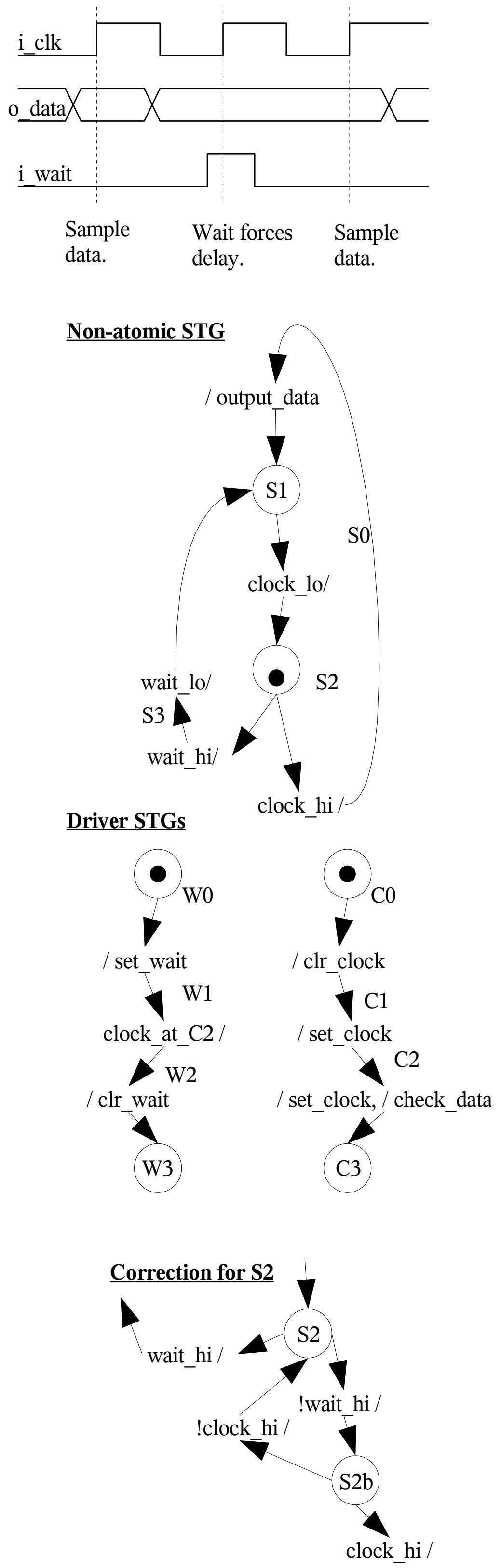


Illustration 3.8: STG-FT Example

The simulation will show no problems and indicate correct operation with no livelock and deadlock for all possible sequences of operation. When implemented using software on microprocessors the system will sometimes fail. Problem scenarios include-

- wait just before clock : the CPU polls wait and finds it low, shortly after wait goes high but the CPU does not see this due to the polling sequence. The CPU tests for clock high and finds this true and then asserts new data. Clearly a wait request has been missed and a data bit will be lost.
- wait just after clock : the CPU samples clock just before it goes high and finds it low, clock then goes high but this is not detected by the CPU due to the polling sequence. The signal wait goes high and the CPU tests wait and finds it high and so delays the sending of new data. Clearly the wait request has been applied one cycle too early.

If the STG-FT process introduced in this section is applied to the non-atomic STG just introduced, it can be seen that it has a non-atomic test transition at S2. Note how this place has been redrawn at the bottom of the illustration to satisfy the atomic modelling rules. Simulation of the corrected STG-FT will detect the problem scenarios listed above. This example shows the necessity of the STG-FT approach when a system is implemented using software threads rather than asynchronous logic.

The data generator example as implemented with software threads cannot be made to work as correct operation requires the detection of the rising edge of clock and simultaneously reading the state of wait. The sampling delays of a thread mean the exact point of the clock transition is missed, and even if it is detected then it is not possible to sample wait at the same time. Some additional hardware is required, for example wait could be sampled by hardware – perhaps a D flipflop clocked by clock with the D input connected to wait.

The original STG-FT, with non-atomic elements, could be made to work with asynchronous logic though meta-stability (Johnston 1993) is an issue as wait could be changing just as the clock is rising.

3.2.4 Reachability

This chapter has been working toward specifying simulation and testing methods for the new 2-wire TIA protocol that will be detailed in chapter 4. Thus far the protocol attributes have been identified and the preferred modelling method (STG-FT) identified. STG-FT has been chosen as like its parent (basic STGs), it can be used to identify behavior such as livelock and deadlock and prove the protocol will always be correct. The literature explains that these significant conclusions are achieved by using the system model to create a reachability tree (also called a reachability graph) which details the system trajectory given some initial state or states. This subsection investigates reachability tree issues that are relevant to the construction of the simulator created and used in chapter 4.

Regrettably the term reachability does not have a single meaning in the literature. The various meanings need to examined and a meaning suitable for this thesis must be chosen. The range of meanings in the literature are as follows-

Peterson (1981) describes a reachability tree which captures not just the unique states but all possible sequences of global states give some initial global state. He showed that even a simple Petri net can have an infinite reachability tree given this approach. He then goes on to propose an algorithm that only captures the states executed which is guaranteed to give a finite reachability tree.
A reachability tree that captures every possible path is not the same as a reachability tree that just captures every possible state. From Peterson's own reasoning the former will detect livelock (a never ending loop) whereas the later may not.

Yuang (1988) describes a simple “perturbation” technique to create a tree that uncovers all global states given a number of cooperating state machines each with their own state. A global state is a unique combination of the individual state machine states. An initial global state is used as a starting point and all transitions are exercised to create a new set of global states. Each new global state has all its transitions exercised thus forming and expanding tree structure. This approach can find deadlock and unspecified receptions but is not guaranteed to find all livelock. Yuang refers to many other techniques to help prune the reachability tree.

Dutta (1997) comments that the content of a reachability tree is dependent on the algorithm used to create it and so different protocol properties may be discoverable from different algorithms. The nature of the algorithm will also effect how long a complete reachability analysis will take and how much memory it will consume.

Cortadella (2003), when describing the tool Petrify, describes a reachability tree called a state graph that captures all possible sequences of states.

The ideal verification technique for 2-wire TIA will detect *all* livelock, deadlock and confirm correctness. The verification should guarantee these things not just offer a high probability that everything is OK. This means that the preferred reachability method for 2-wire TIA must be that which examines every possible path from some initial state and determines whether it ends in correct operation or some error including livelock or deadlock. The warning from the literature is that such an approach can take large amounts of time, memory, and may lock up with infinite looping. This problem is exacerbated by the state explosion issue discussed in section 3.2.1 on FSMs.
Clearly any simulator must be designed to cope with such eventualities and be designed to be as fast as possible.

The literature also warns that the calculation of the reachability tree can require significant amounts of processor time and possibly memory. Given that this thesis implements a reachability tree simulator in chapter 4 it is important to consider methods by which the computational effort can be minimized. The literature offers a variety of methods-

Chu and Liu (1989) propose several techniques to limit state explosion in Extended Finite state Machines (EFSM).
Dead variable sets : The first technique analyses variables associated with an FSM and notes when they no longer have any effect on the system. Since the system state space is the cross product of all subsystem state spaces eliminating a variable will reduce the system state space.
Undefined variable sets : The second technique is similar and eliminates variables when they are undefined, assuming a variable is not referenced when it is undefined. Chu & Liu note that it may be necessary to compute the total global state space before states can be

eliminated thus making this approach less than useful.
These techniques reduce the global state space of an Alternating Bit Protocol (ABP) from 158 states to at best 56 states. Chu and Liu use this approach for calculating the global state space which is necessary for some forms of deadlock analysis, which cannot check for livelock. Unfortunately it does not help to reduce the size of the reachability tree and so is not useful.

Corbett (1996) surveys a number of state explosion reduction techniques and trials 3 of them. The survey contains several techniques that may be of use for modelling TIA systems.
State space reduction reduces the number of states and thus the size of the global state space. Techniques include virtual coarsening (merging of internal actions with adjacent external actions), partial order techniques (which model interleaved FSMs rather than full concurrency), and taking advantage of symmetries.
Compositional techniques divide a system into smaller elements which are individually analyzed. A simplified model of each subsystem is used to model the total system thus reducing the global state space.
Abstraction ignores some state or variable information.
Most of the relevant techniques surveyed by Corbett rely on the reduction in the fidelity of the simulation in order to reduce the global state space. This runs the danger of missing behavior that may result in deadlock and livelock. The human designer must be able to look at each space reduction proposal and be able to say “I am 100% certain that this will not change the behavior of the system”. If only one mistake is made then a livelock or deadlock may be missed.
Corbett also mentions simultaneous versus interleaved simulation, the former being where multiple events can happen at one time and the later where only one event happens at any time. It is noted that simultaneous event simulation takes more time than interleaved simulation.

D'Argenio and Niebert (1996) examine partial order methods and note that “They are based on the observation that execution order of concurrent operations does not usually change the validity of the operation.” Given this observation it would seem that adopting any form of partial order analysis ruins the 100% certainty of property verification that is inherent in a full analysis of the transition based reachability tree.

Kishinevsky et al (1998) comments on methods to reduce state explosion in reference to Petri nets and delivers conclusions very similar to Corbett.

Khomenko (2002) discusses u*nfoldings* are essentially partial order reductions which will have the same problems with fidelity as discussed above.

Paster et al (2001) uses symbolic Binary Decision Diagrams (BDD) to investigate the reachability tree for livelock and deadlock and trials this approach on a number of problems. He concludes that the number of BDD nodes may exceed the reachability set if the Petri net is not highly concurrent.
Gunther et al (2001) consider several efficiencies that may be made to some BDD problem types.
The TIA problem has master and slave continually waiting for each other. This suggests a lack of concurrency and so the symbolic BDD approach may not be useful.

TIA simulation based on STG-FT, which is described in section 4.4, has several lessons to learn from this survey which are-

1. Clearly the number of states and the number of state machine or variables must be minimized. As commented any reduction in model fidelity must be totally invisible to all parts of the system otherwise key behavior may be missed.

2. Partial order reductions may limit the ability to detect livelock and deadlock, and so these methods will not be used. The reachability tree must capture the full universe of states and transitions between those states in order to be guaranteed of detection livelock and deadlock.

3. Enumeration of the global state space and investigation of that space will detect deadlock but may miss livelock. Full traversal of the transition based reachability tree is necessary for livelock detection.

4. The simulation must choose between concurrent or interleaved operation. If a reachability tree is being traversed then in effect an interleaved search is being performed. If however every possible path is being checked then one of those paths will correspond to a concurrent combination where several event generators fire before an event receiver is triggered. The STG-FT simulator will use this approach.

3.3 Summary and Conclusions

This chapter dealt with a number of key issues that must be decided before the 2-wire TIA protocol is described, modelled, simulated, implemented and tested in chapter 4.

The first issue in section 3.2.1 was to detail what properties of a protocol should be verified by modelling, and then tested on a real implementation. These included livelock, deadlock, correct operation, unspecified input and abnormal state/input.

Section 3.2 examines several different modelling methods and discovered, surprisingly, that Signal Transition Graphs did not properly model a system implemented with software. A variant of STG called STG For Threads was developed to overcome these problems.
The choice of simulation method is between STGs and FSMs . This is not an easy decision with several conflicting indicators-

> *STGs model concurrency* : as Cortadella et al (1995) explain "(FSMs) cannot explicitly express the notions of concurrency, causality and conflict. Petri nets can naturally capture these notions." Peterson (1981) shows how a Petri net based system can easily model a system where an FSM would not.
> These advantages are also commented on by other authors such as Merlin (1979) who also offers examples.
>
> *Simpler* : as Peterson (1981) comments "the state machine description is more understandable than the Petri net description".
>
> *STG converts to FSM* : STGs are usually converted to FSMs for the purposes of simulation so why not simply start with FSMs?
>
> *Software implementation* : STG-FT is well suited to modelling systems implemented in software.

On balance STG-FT model developed in section 3.2.3 would appear to be a the better modelling method as the superior concurrency modelling power of STGs seems worthwhile and STG-FT can model the behavior of software.

This chapter has completed a range of important tasks as described above and so the next chapter (chapter 4) is able to propose, model, simulate, implement, and test a new 2-wire TIA protocol.

Looking forward, the work performed in chapter 4 has validated the decisions made in this chapter. As expected the STG-FT model worked very well and the simulation matched the implementation. The STG-FT model was particularly useful and fulfilled its minor promises as well-

> *Error detection* : 2-wire TIA was modelled using FSMs and STG-FT. Using STG-FT forces the human to carefully check the sequence of operations that make the protocol work thus helping eliminate errors. Drawing the STG-FT model for 2-wire TIA highlighted two errors made in the original FSM model.
>
> *FSM interaction is explicit* : the interaction of FSMs can be unclear given the FSM definition. The STGs show the interactions in a very explicit way which allows error checking and eases the translation to state tables.
>
> *Close to state tables* : the reachability tree is generated by a simulator, and the heart of the simulator can be a state table that holds the entire STG-FT model. It was found that the translation from STG-FT to state table was very easy and that most of the simulator code could be kept generic. Perhaps the simulator created by this author could be turned into a general purpose toolkit.

4 2-Wire Time Independent Asynchronous Protocol

***Overview** : this chapter outlines a new and novel 2-wire TIA protocol. The 2-wire TIA proposed is modelled using STG-FT, simulated using a custom developed simulator, implemented and tested. The simulation matches the implementation and the protocol is found to work correctly.*

This thesis seeks to discover a more economic method of communications for Low End Microprocessors. Chapter 2 started the process by identifying a new category of communications called Time Independent Asynchronous communications which offers hope of achieving this goal. The literature survey identified several existing 3-wire TIA systems but noted that 3 wires is a significant resource for a LEM. This chapter shows a new and novel 2-wire TIA system that uses one less pin than anything found in the literature survey.

The new 2-wire TIA protocol is described, verified, and implemented as follows. Section 4.1 explains initial design problems that frustrated the search for 2-wire TIA and how these were solved. Section 4.2 explains the 2-wire TIA protocol using timing waveforms. Section 4.3 shows the STG-FT modelling method (developed in section 3.2.3) modelling 2-wire TIA. Section 4.4 introduces a purpose built STG-FT simulator and shows how the 2-wire TIA protocol is verified as per the protocol attributes identified in section 3.1 of the literature survey. Section 4.5 shows how the protocol is implemented between a personal computer and a Tiny26 microprocessor and how the protocol attributes are tested. The attributes from real measurement are found to match those found by the simulator.

4.1 Initial Design Problems

Initial design of a 2-wire TIA system using simple digital gates proved fruitless. No combination of normal inputs and outputs could deliver bidirectional data transfer using only two wires. Something novel was required to make it feasible and so the rules and assumptions behind the interface logic design were examined and questioned.

The questioning that paid most benefit was to consider alternative drive technologies. The IEEE-488 (GPIB) bus uses open collector drive which allows outputs to be connected together and achieves a wired OR function, if any one output is low the resulting logic level is low else the logic level is high. Microprocessor buses use tristate outputs which can drive high, low, or go into a high impedance state and act as an input. On a tristate bus only one tristate output may be active at a time otherwise damage may result. These two extra output drive options still did not appear to allow a 2-wire TIA system.

There is another output type called a weak drive where the digital output has a notable output impedance but can drive a digital input high or low quite successfully. A normal digital tristate drive when driving actively high or low (not in high impedance mode) can overdrive the weak drive without fear of damage. A weak drive can be implemented with a normal digital output connected in series with a resistor. Weak drives are not commonly used as the extra impedance slows the speed at which a bus can be driven as the resistance of the output forms an RC low pass filter with the capacitance of the wire being driven and any attached devices.

There are now four drive types available for use – normal, open collector, tristate, and weak drive. Consider the simplest communications situation with just two hosts, there are 4^2 or 16 combinations of possible output drives to consider for each line. One of those combinations : weak drive and tristate showed most promise and with some innovative protocol design provided a 2-wire TIA solution as shown in the next section.

4.2 Waveforms and Protocol

A new and novel 2-wire TIA protocol is explained by the waveforms below (Radcliffe 2006b). It is a physical layer protocol which is only concerned with the bidirectional transmission of data bits between a master and a slave. Issues such as byte transfer, error correction are left to high layers. Unlike most other protocols transmit, receive, and any synchronization is achieved in one cycle and as such the entire physical layer protocol is defined by this one cycle.

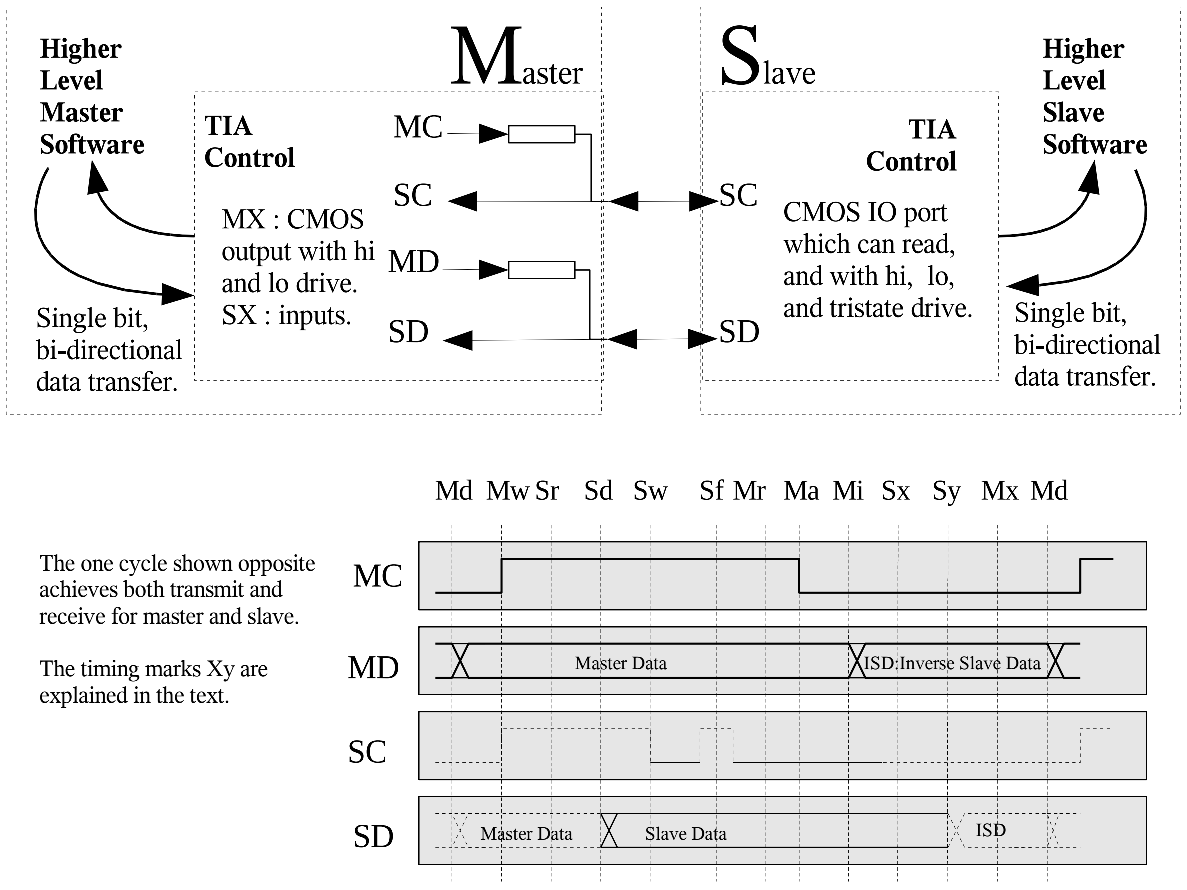

Master
Slave
Higher Level Master Software
TIA Control
MC
SC
MD
SD
MX : CMOS output with hi and lo drive.
SX : inputs.
Single bit, bi-directional data transfer.
TIA Control
CMOS IO port which can read, and with hi, lo, and tristate drive.
Higher Level Slave Software
Single bit, bi-directional data transfer.
Md Mw Sr Sd Sw Sf Mr Ma Mi Sx Sy Mx Md
The one cycle shown opposite achieves both transmit and receive for master and slave.
The timing marks Xy are explained in the text.
MC
MD
Master Data
ISD:Inverse Slave Data
SC
SD
Master Data
Slave Data
ISD

Illustration 4.1: Proposed Digital 2-Wire TIA Waveforms

The waveforms show communications between a master M and a slave S. The master has a weak drive (as shown with a series resistor) which means that a slave can either be tristate and leave the master data value on a wire, or choose to be an active output and overwrite any master data value without causing damage to the master outputs. At every stage of the data transfer only the sequence of signals controls the link. There is no time dependency on a signal, or time relationship between signals, only signal order is important.

The dotted waveforms for SC and SD indicate the slave is tri-stated and any logic level is due to the master drive. A solid line indicates the slave is driving the wire.

The following points explain in detail what is happening at each important point in the waveforms-

- Initially SC (Slave Clock) and SD (Slave Data) are tristate, MC (Master Clock) is low, MD (Master Data) is high or low.
- Md : M initiates a cycle by first setting MD to the bit value to send to S.
- Mw : M sets MC high to tell S that data is available to read.
- Sr : S sees MC as high and reads the data bit.
- Sd : S now asserts the data bit it wishes to send to M.
- Sw : S tells M the data bit is available by forcing SC low.
- Sf : S checks if M has read its data bit by tristating SC, because SC goes high M has not read yet.
  At this point if M reads SC to check if a data bit from S is available it sees a high, and concludes the data bit is not ready.
- Mr : M sees SC is low and so reads the data bit from SD.
- Ma : M acknowledges it has read the data bit from SD by setting MC low.
- Mi : M puts an inverse of the bit from S on MD.
- Sx : S attempts to see if M has read its data bit by tristating SC. The resulting low means M has acknowledged reading the data bit from S.
- Sy : S tristates SD resulting in an inversion due to the inverse value M has placed on MD.
- Mx : M sees the inversion on SD (now matching MD) and concludes that S has tristated both SC and SD and is ready for another cycle.
- Md : the cycle starts again.

The solution proposed is only possible because of the combination of several ideas. Firstly the master is designed to have a weak drive and the slave a standard IO pin drive of high, low, and tristate. The slave output can overdrive the master output without causing device damage. Secondly at timing point Sf the slave poll is designed to look like Sd (the slave is not finished writing its data yet). If the master scan Mr coincides with the slave scan Sf then there is not an error, the master thinks the slave is not ready and scans again. Finally the slave's release of the data wire is designed to be detected by the master putting an inverse version of the slave data on MD, which is only seen on SD after the slave has tristated its drive.

These ideas may well be useful in creating other forms of digital communications.

The literature survey in section 2.2.2 briefly examined the VME bus and showed that even when a system is TIA in nature the reality of the media can create timing problems. The VME bus requires that a host put data on the data wire 35 ns before a Data Strobe was pulled active low.

Consider a situation where this 35 ns rule was not satisfied. The data would be asserted at the same time as the Data Strobe. Settling time on the bus (a transmission line in effect) and differential delays may cause the strobe signal to be recognized before the data is valid thus the receiving end may receive corrupted data. Any TIA system must ensure that signal order is preserved and this will mean a settling time delay between signals sent from one host to another. The settling time allowed depends on the nature of the media and only needs to be large enough to guarantee the order of signals.

For a microprocessor where software is driving the IO pins the minimum time period between events could be as short as one instruction cycle. For a fast low end microprocessor such as a 20 MHz Atmel Tiny26 this would be 50 ns which is quite adequate for settling times on a digital logic connection. If TIA were implemented using fast logic then a settling time delay, much as that in the VME bus, would need to be built into the logic.

This section has successfully described a new 2-wire TIA protocol but the properties of that protocol are not verified. This will be pursued in the following sections.

4.3 STG-FT Model

The STF-FT modelling method developed in this thesis in section 3.2.3 has been used to model the 2-wire TIA protocol described in section 4.2. This model will be used to implement the simulation in following subsections.

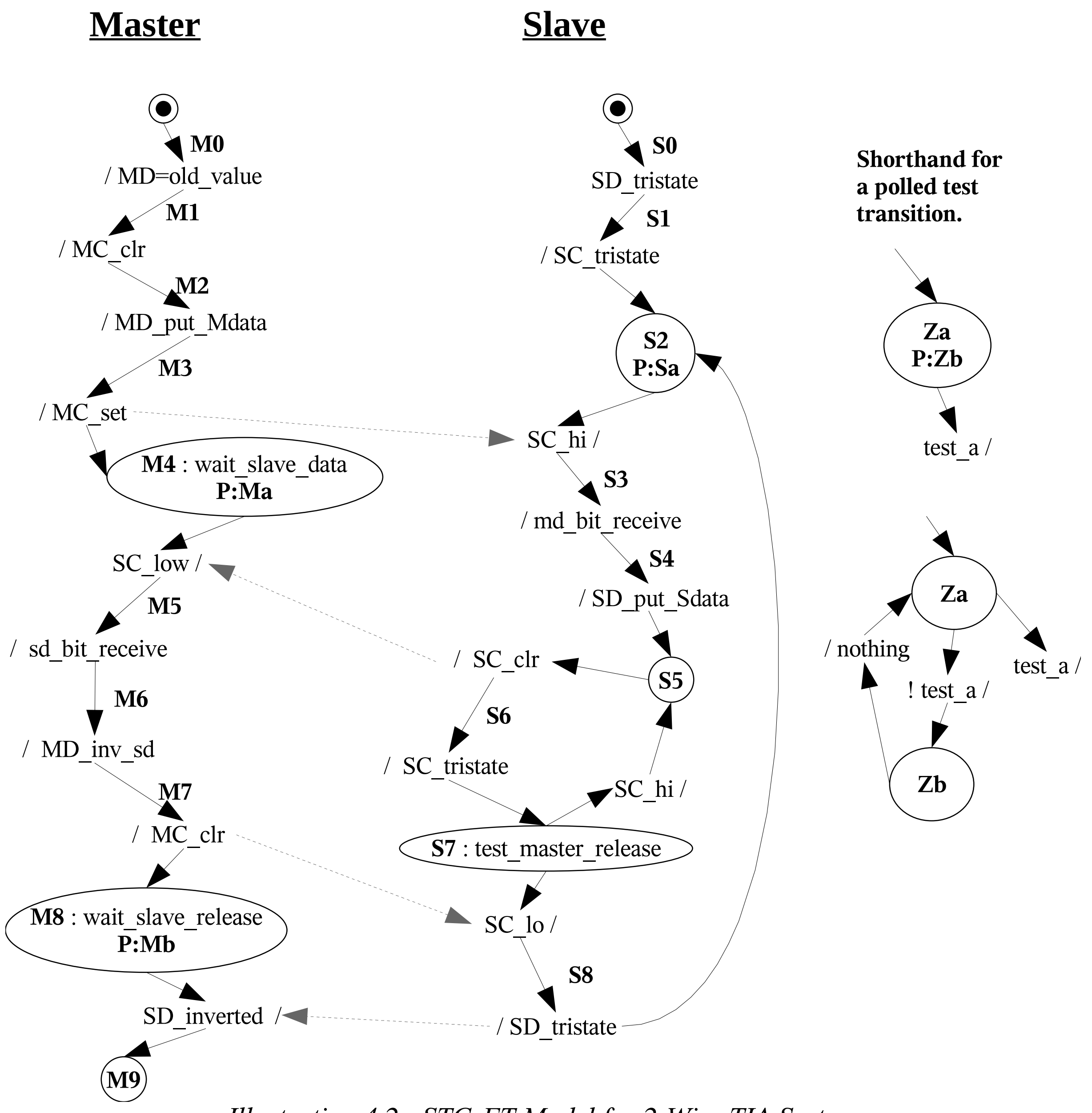


Illustration 4.2: STG-FT Model for 2-Wire TIA System

Illustration 4.2 has several points worthy of note.

Wait states where an FSM is waiting for an input are modelling with extra polling states to model the dead zone inherent in software implemented systems. These include states M2, M8, and S2. These states may well cause problems as per the example in section 3.2.3.

The slave has a loop between states S5, S6, and S7 which might interact with polled state M8 and cause problems.

Any problems will be discovered by simulation in the next section.

4.4 STG-FT Simulator and Results

Section 4.3 has developed an STG-FT model of 2-wire TIA. The next step is to create a simulator to test the properties of the modelled protocol. This section starts by describing the key design decisions made in designing the simulator and finally reports on the results from simulating the 2-wire TIA.

Section 3.1 investigated what properties of a protocol should be verified. The key conclusions were that a full reachability tree of all possible paths between states must be constructed and this should be investigated to check for-

- correct operation,
- livelock and deadlock,
- recovery from any initial state which includes all input signals.

In practice this can only be achieved with a simulator as it is usually impossible to drive a real system through all such states, and even if this was possible it would take far too long to complete the verification.

This section will show that 2-wire TIA in normal operation is always correct and free of livelock and deadlock with one exception. There is a livelock that can only be sustained if a master and slave loop have exactly the same timing. In practice this is impossible but it would be prudent to ensure the loops have different periods to ensure no short lived livelock that can reduce data throughput.

Starting from any initial system state with any input combination did show deadlock but this can be eliminated by applying a timeout on the master which returns the master to its idle state. This timeout can be quite long and so not interfere with the TIA operation.

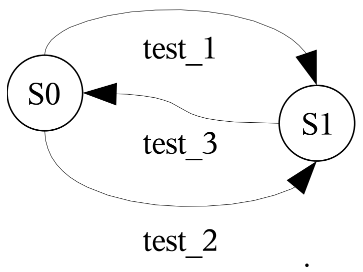
4.4.1 Simulator Implementation

This subsection outlines a number of important decisions that had to be made before a simulator could be selected or created.

It is important to note that an STG-FT model defines a number of communicating FSMs that defines the total system to be simulated. The simulator needs to run multiple FSMs and check the protocol properties. Several key decisions were made that resulted in a custom simulator that this author designed and coded-

> *Use existing simulator?* There are many excellent simulators available from the web, a Google search on "protocol simulators" will show a large number of excellent products. Classic simulators include Opnet (www.opnet.com) which due to competition from tools such as Omnet++ (www.omnetpp.org) is now essentially free for research students. STG-FT is a new modelling method and it is not certain that existing simulators will implement everything that is needed. Given the author has good programming skills it was decided to write a simulator from scratch to ensure the simulator would be able to do all that STG-FT modelling requires.
>
> *Use C* : the literature survey in section 3.2.4 found that state explosion could be a severe problem even with relatively small systems and so any simulator should be very fast. Given that a simulator will be programmed from scratch which language should be used?
> Java is an excellent language but given that it is an interpreted language its execution speed is relative slow.
> The object oriented features of C++ allow the use of clever object structures and patterns. Dingle and Hildebrandt (1998) found that C++ structure that use dynamic memory tend to be very slow. This author's personal experience is that complex number functions implemented using C++ operator overloading run 3 times slower than equivalent functions written in C.
> Given that speed is essential the simulator should be written in C.
>
> *Use state tables* : There are many ways to implement FSMs in code. State tables have been selected as they allow a fair degree of automation. Given a FSM diagram the code for the state table can be created in a mechanical manner as can the function headers for all transitions, as shown in illustration 4.3 that follows. The programmer need only create the

code to implement the transition functions. All state machines have been placed in one table thus making it easy to have an arbitrary number of state machines in the simulation.



```
typedef struct { byte current_state ;       // state variable.
                 bool (*exit_test_ptr)() ;  // addr of function.
                 byte next_state ;          // if exit returns true
                 void (*exit_action)() ;    // addr of function.
               } state_line ;

state_line state_table[] =
 {// current state    exit test     next state   exit action.
        0,            &test_1,         1,         &action_1
       ,0,            &test_2,         1,         &action_2
       ,1,            &test_3,         0,         &action_3
 } ;
```

Illustration 4.3: State Table Implementation

Given the decisions made above the simulator was implemented as a command line program written in C. The development environment was KDevelop running on the KDE desktop. The operating systems was Fedora Core 3 which has Linux kernel 2.6. The code can be found on the CDROM that accompanies this thesis. Key architecture features are discussed below-

Generic : the simulator code has been kept generic apart from two header files called problem_part_1.h and problem_part_2.h. Much of these files could be automatically generated from the STG-FT model. The programmer must still add code for several items-

- Code for each test or action transition.
- Define all possible initial states.
- Define when the system has achieved a correct outcome and when it is finished.
- Define what livelock and deadlock conditions to ignore (for example polling loops within one FSM object).

The same simulator has since been used on other problems and the generic structure has shown that only the problem_part_?.h files need be changed. The generic modules include-

- sim_ioio.c is a main file that selects which simulation in sim_run.c to run given command line parameters.
- fsm_defn.c : includes the user definitions of the state machines to fully define the state machines. It also handles the marking of system states (cross product of individual state machines) for short cut evaluation.
- simulator.c : checks user supplied state data is consistent and runs a state machine one step. Because all state machines are in the one table the one routine can service any state machine.
- sim_run.h uses the state machine routines from fsm_defn.h to do various simulations. This includes a step by step analysis of what happens from one initial state, to exhaustive analysis of all paths from all valid initial states, to analysis of all paths from every possible initial system state.

Key data structures include-

- struct_fsm_state in problem_part_1.h defines a node and contains all the state and data that will completely define the system state. A pointer called "now" is used to point to this system state and is used extensively in the simulation.
- struct_fsm_init_data in problem_part_1.h defines the initial state of the system.
- fsm_list in problem_part_1.h names all the state machines, variables, and constants that together define the system.

Recursive implementation : The reachability tree must examine every possible path between all states in order to be certain about livelock and deadlock. A recursive algorithm proved a simple and fast way to traverse the tree and investigate its properties. The memory requirement was only the depth n of the tree rather than the full fan out of every possible node which is proportion to m^n where m is the number of branches from each node.

4.4.2 Simulation of Normal Operation

This subsection details the simulation results for normal operation of the 2-wire TIA protocol. It also proposes a method by which simulation can be dramatically sped up. This method was not discovered during the literature survey on reachability analysis in section 3.2.4.

The simulator has a simple mode which shows operation from one initial state and this may be used to check basic operation. Normal operation is checked by running from every possible initial state and investigating every possible path : a full path based reachability analysis. The results for a full reachability analysis from one initial state are shown below-

```
  Atomic STG Simulator  version 1.4
   Problem : 4 Port - 2 IO 2-wire TIA Communications, polled tests version
1.1

   Reachability tree check for every possible valid path.
      - one initial state.
      - checks for correct data transfer ( correctness), livelock,
        and deadlock.
      - allowable livelocks eliminated ( eliminate_OK_livelock()
        in problem_part_2.h).



   Finished reachability tree analysis from initial state-
     ms=M0, ss=S0, mc=mc0, md=md0, sc=sct, sd=sdt, mt=md0, st=sd1
     Search depth before livelock/deadlock error listed = 200
     Number of nodes examined    = 31977049015
     Number of correct  finishes = 1042888704
     Number of faulty   finishes = 0
     Number of livelock finishes = 0
     Number of deadlock finishes = 0
   One reachability tree finished.
   In total 31977049015 nodes examined.
   The activity took 6546.853521 seconds.

Press Enter to continue!
```

Illustration 4.4: Simulator Results for Slow Evaluation

There are several conclusions of interest which can be drawn from this simulation result.

Correct : For normal operation the protocol is correct and all possible variations in sequencing (and thus timing) will always result in correct transmission of the data bits.

No deadlock : there is no deadlock possible in the protocol.

False livelock : the routine `eliminate_OK_livelock()` in code file problem_part_2.h allows the user to abort specified livelock conditions. Some livelocks are a product of the simulation. For example with reference to the STG-FT model in section 4.3 all polling states are a loop where the simulator will be able to find the possibility of livelock. These states include S2/Sa, M4/Ma, and M8/Mb. The slave loop consisting of S5,S6,S7 is also falls into this category. These livelocks result from only one state machine being operational, are thus an artifact of the simulation, and can safely be ignored.

Real livelock : the simulator did find one real livelock between states M4/Ma and the slave loop S5,S6,S7. It is notable that this livelock would not be found with a normal STG model, only the STG-FT enhancements will detect it. On close inspection this livelock can continue only if the timing of the master and slave loops are identical. Even a small timing difference will eventually cause the loops to drop out of synchrony and so the livelock is of a transitory nature and is not permanent. The implementation of the 2-wire TIA should consider making sure these loops do not have the same period to forestall any temporary livelock behavior.

Long simulation time : analyzing one initial state required the traversing of nearly 32 billion nodes which took about 109 minutes on a 1.6 GHz Celeron running Linux kernel 2.6 and the KDE desktop. Testing all eight possible valid initial states took 8 times longer : about 14.5 hours.
This could be a real problem for the next stage of analysis, starting from every possible initial system state not just the 8 valid initial system states.
The literature survey on protocol verification in section 3.1 found that the ability to cope with any input and any system state is very important to gauging the robustness of a protocol. Data errors would be expected but the protocol should not exhibit livelock or deadlock. A full analysis from one initial state took 109 minutes, there are some 17280 possible system states leading to a simulation time of approximately 3.5 years. Clearly a faster form of simulation is required.

Fast simulator: The simulator achieved a respectable search rate of approximately 4.8 million nodes per second.

One conclusion above was a particular problem. Checking robustness by brute force and running from any initial system state was impossible as a single simulation would take 3.5 years. The literature survey of reachability analysis methods in section 3.4.2 did not offer any speed up techniques that were applicable to the 2-wire TIA problem.

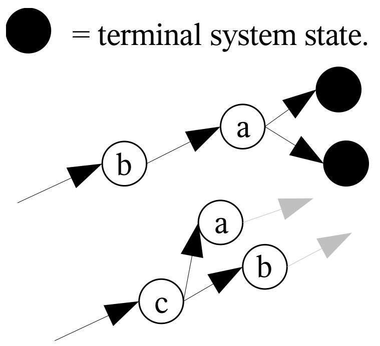
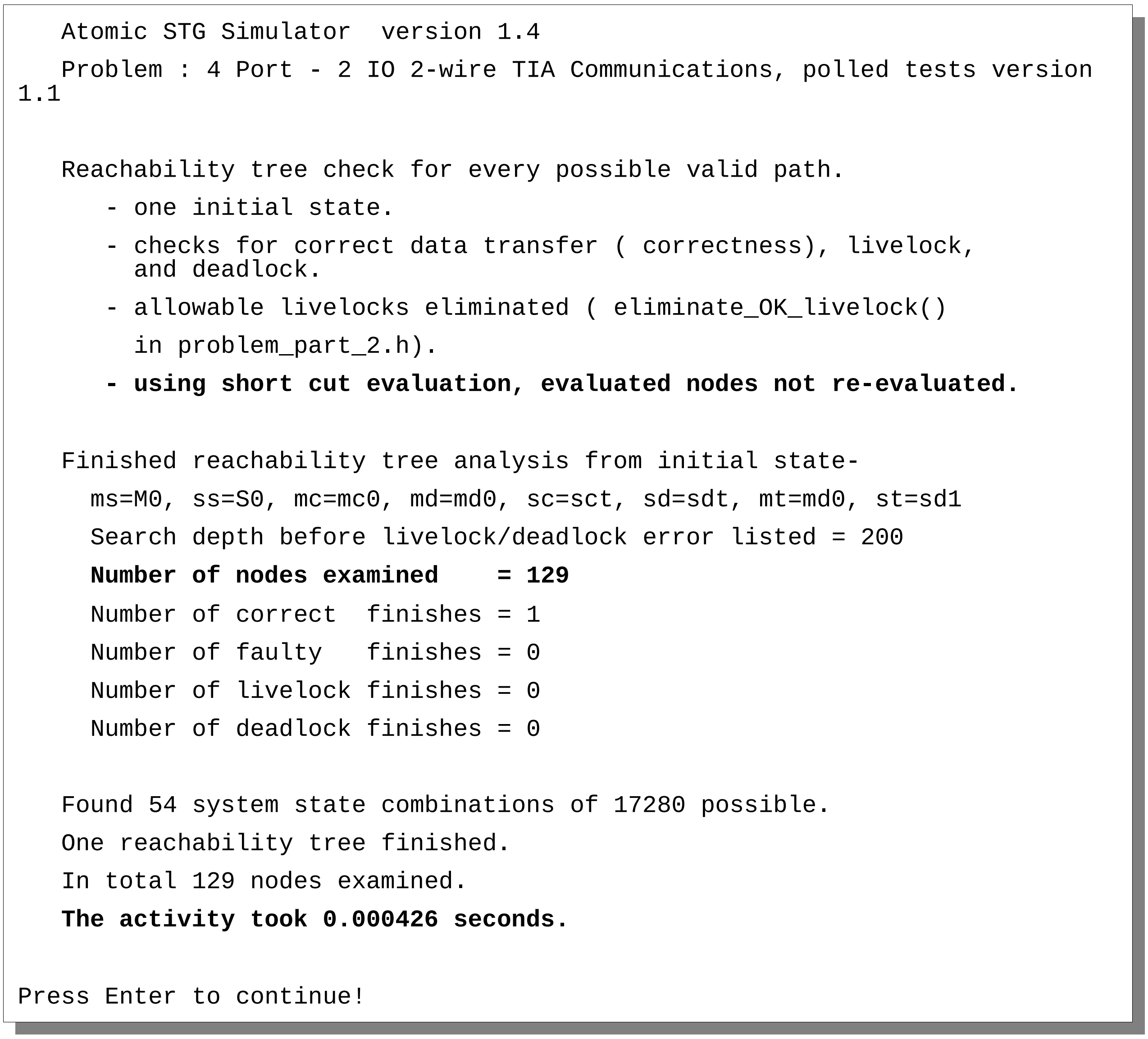


Illustration 4.5: State Tick Off Strategy

A speed up solution came from observation of the simulation trajectory. The same loops were being executed again and again but each coming from a slightly different system state. Consider the illustration above. Solid circles represent a system state where the simulation terminates. From the top flowchart it can be seen that system state “a” will always terminate with known results. State “b” can only result in state “a”. System states “a” and “b” could be “ticked off” as being fully simulated. Now consider the bottom flowchart. Given that state “c” can only result in state “b” or state “a” then it too can be ticked off and several simulation steps are avoided. The effect or ticking off states is quickly cumulative resulting in a dramatic reduction in the nodes visited. The tick off result can be as simple as a boolean flag indicating no livelock or deadlock, or more complex such as set of outcomes found. The tick off strategy will only work if the system state vector contains *everything* that can effect the performance of the system. It should include not only all states and variables but data such as the master and slave bits to be sent. On reflection the tick off strategy is a way to truncate a depth first tree search in order to reduce the search space.
The results of applying this strategy are shown in the following illustration; the result is staggering, a reduction from 32 billion nodes to 125 nodes, speedup factor of nearly 250 million! This new strategy was stressed by adding a variety of faults that should induce livelock and deadlock, the strategy worked perfectly each time and properly predicted the same outcome found by exhaustive simulation.



```
   Atomic STG Simulator  version 1.4
   Problem : 4 Port - 2 IO 2-wire TIA Communications, polled tests version 
1.1

   Reachability tree check for every possible valid path.
      - one initial state.
      - checks for correct data transfer ( correctness), livelock,
        and deadlock.
      - allowable livelocks eliminated ( eliminate_OK_livelock()
        in problem_part_2.h).
      - using short cut evaluation, evaluated nodes not re-evaluated.

   Finished reachability tree analysis from initial state-
     ms=M0, ss=S0, mc=mc0, md=md0, sc=sct, sd=sdt, mt=md0, st=sd1
     Search depth before livelock/deadlock error listed = 200
     Number of nodes examined    = 129
     Number of correct  finishes = 1
     Number of faulty   finishes = 0
     Number of livelock finishes = 0
     Number of deadlock finishes = 0

   Found 54 system state combinations of 17280 possible.
   One reachability tree finished.
   In total 129 nodes examined.
   The activity took 0.000426 seconds.

Press Enter to continue!
```

Illustration 4.6: Simulation Result for Fast State Tick Off Evaluation

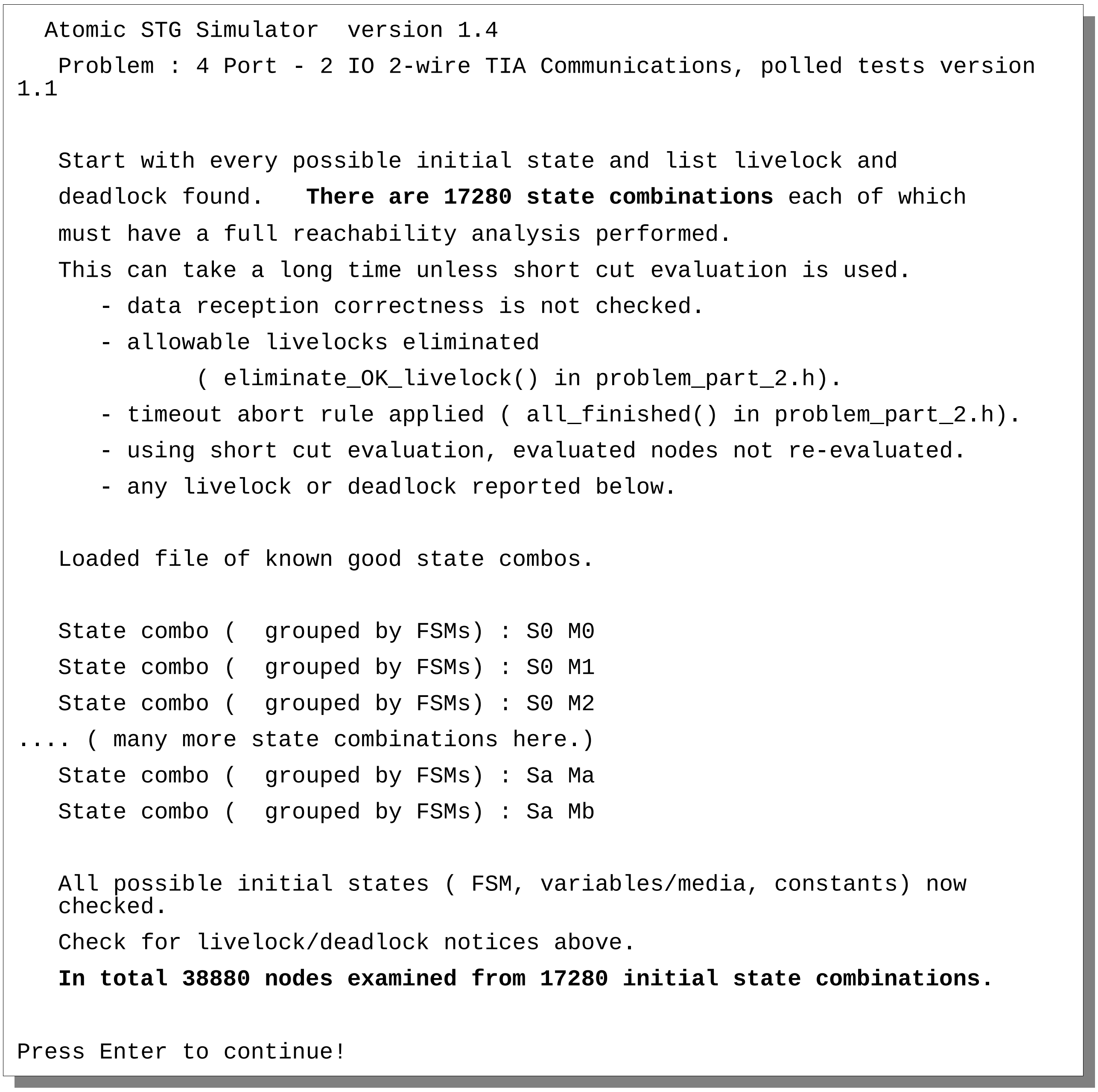
Normal operation of 2-wire TIA has been shown to be correct, free of deadlock, but having one livelock which should be transitory. With new speed up it is possible to complete the final phase of simulation, running from every possible initial state.

4.4.3 Simulation of Any Initial State

The STG-FT simulation encapsulates all active and passive actors in the simulation into a system state vector, including all possible inputs which are defined by variables. By starting the simulation from every possible system state then every possible input for every possible FSM state is examined. As already discussed this would have proved impossible with the initial brute force simulation method but the "state tick off" strategy allows a more reasonable simulation time. Data transfer cannot be expected to be correct but no livelock or deadlock should exist.

The simulation did find a deadlock situation which can be solved by allowing the master to have a long term timeout back to idle state. This timeout could be quite long, as long as the application can tolerate, and so not interfere with the TIA signalling for normal operation.

In the results below only the initial FSM state combinations are printed out but every system initial state is analyzed, all 17280.



```
  Atomic STG Simulator  version 1.4
   Problem : 4 Port - 2 IO 2-wire TIA Communications, polled tests version
1.1

   Start with every possible initial state and list livelock and
   deadlock found.   There are 17280 state combinations each of which
   must have a full reachability analysis performed.
   This can take a long time unless short cut evaluation is used.
      - data reception correctness is not checked.
      - allowable livelocks eliminated
             ( eliminate_OK_livelock() in problem_part_2.h).
      - timeout abort rule applied ( all_finished() in problem_part_2.h).
      - using short cut evaluation, evaluated nodes not re-evaluated.
      - any livelock or deadlock reported below.

   Loaded file of known good state combos.

   State combo (  grouped by FSMs) : S0 M0
   State combo (  grouped by FSMs) : S0 M1
   State combo (  grouped by FSMs) : S0 M2
.... ( many more state combinations here.)
   State combo (  grouped by FSMs) : Sa Ma
   State combo (  grouped by FSMs) : Sa Mb

   All possible initial states ( FSM, variables/media, constants) now
   checked.
   Check for livelock/deadlock notices above.
   In total 38880 nodes examined from 17280 initial state combinations.

Press Enter to continue!
```

Illustration 4.7: Simulation of Every System State

This subsection has shown that 2-wire TIA will always recover from any system state combination and any set of inputs. A timeout on the master to idle state is required to avoid a deadlock. The timeout can be as long as the application can tolerate and so not interfere with the basic TIA behavior.

4.5 2-Wire TIA Implementation and Conformance Testing

Section 4.2 in this chapter has proposed a 2-wire TIA protocol that suits Low End Microprocessors. Section 4.3 has modelled the new protocol using STG-FT, and section 4.4 simulated the protocol to determine its properties. This section starts by showing how the 2-wire TIA system was implemented between an IBM-PC and a Tiny26 microprocessor. This physical system is then tested and is shown to exactly match the theory and simulation right down to the transitory livelock predicted by the simulation. The livelock is transitory in nature and can be eliminated by ensuring a wait loop in the master, and a matching wait loop in the slave, have different delays.

This section proves that 2-wire TIA does successfully implement a cost effective communications protocol for LEMs, which is the main goal of this thesis.

The diagram below illustrates the testbed used to implement and test 2-wire TIA. The IBM-PC is an Acer Travel Mate 4600 with a 1.6 GHz Celeron processor. The parallel port is an Acer EZ4 EzDock docking station. The PC ran Fedora Core 3 and the KDE desktop.

The Tiny26 is a Low End Microprocessor with 1000 instructions and 20 pins. It is programmed in C using the gcc-avr compiler and the avrdude device programmer. The 2-wire TIA communications has been designed to use the same cable as the avrdude programmer to make programming and testing much easier.

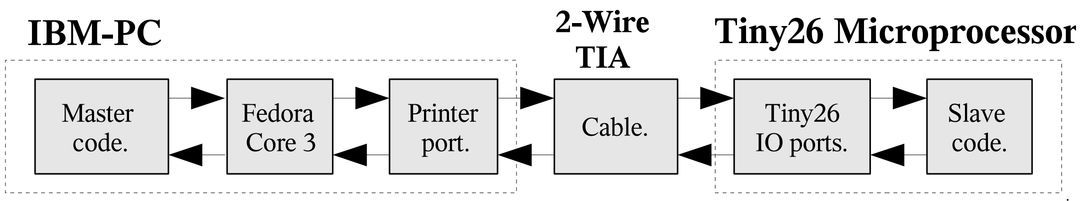


Illustration 4.8: Block Diagram of 2-Wire TIA Implementation

4.5.1 Implementation Details

The 2-wire TIA implementation has been deliberately overlapped with the programming pins of the Tiny26. This saves pin usage as either one or the other may be active never both. The user can use one cable for both programming the Tiny26 and any 2-wire TIA communications.

The circuit of the cable and Tiny26 are below.

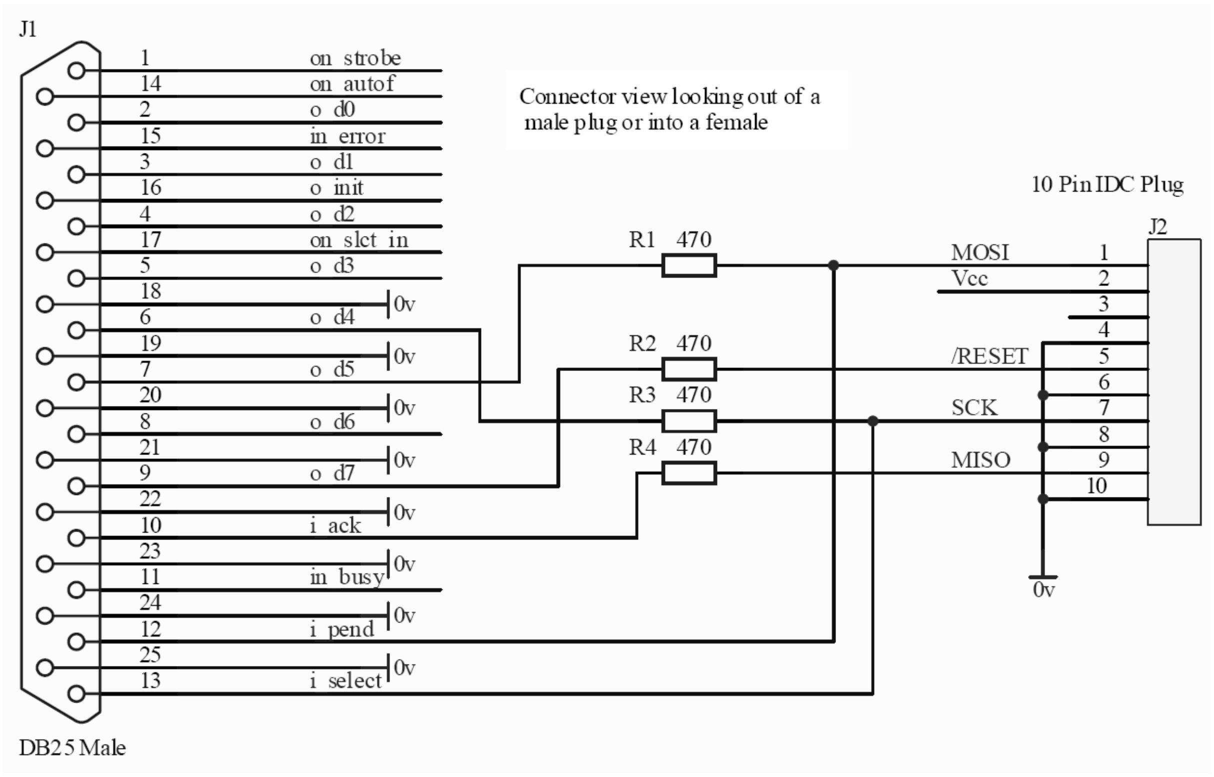


Illustration 4.9: 2-Wire TIA Cable for Tiny26 & IBM-PC

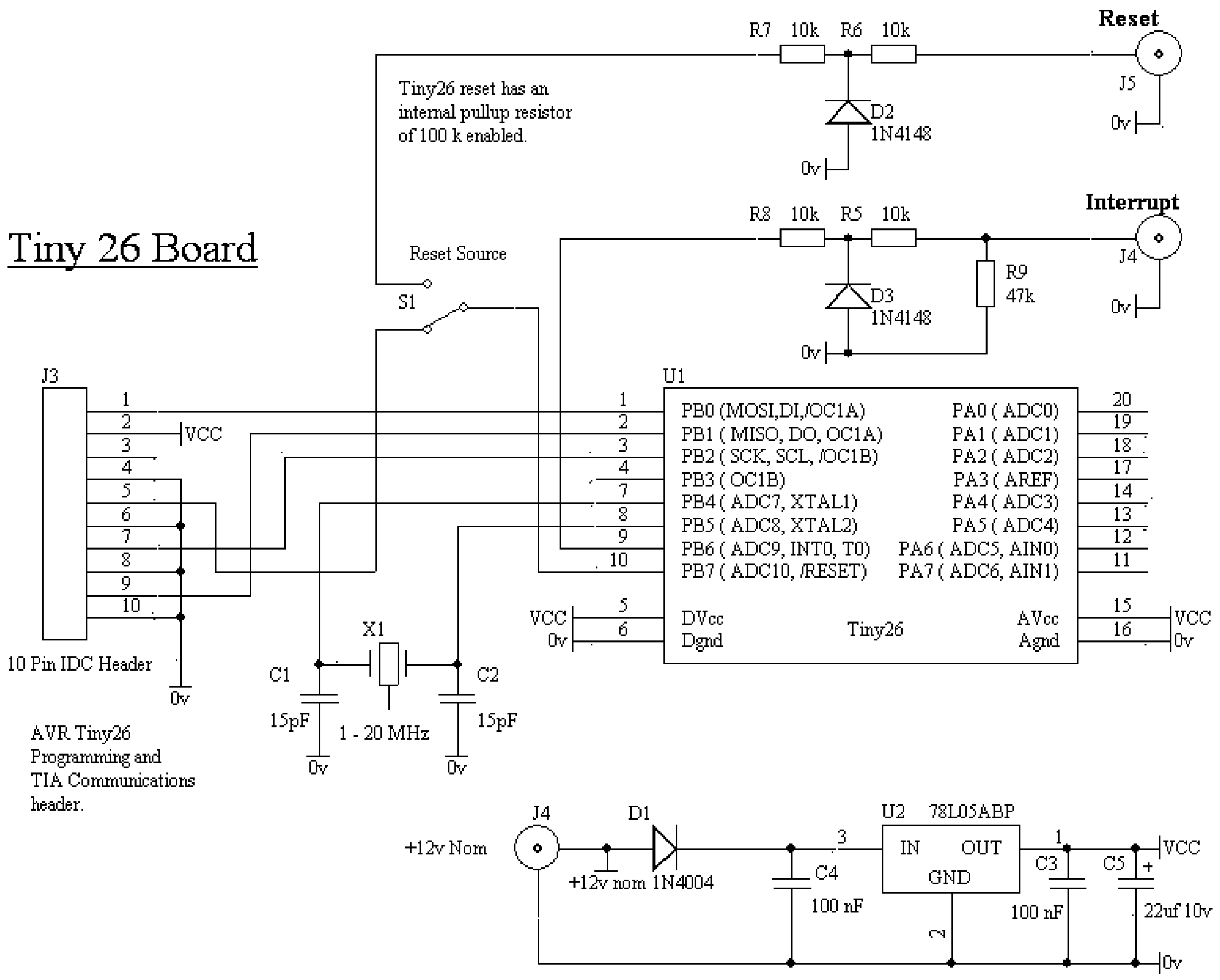


Illustration 4.10: 2-Wire TIA Circuit for Tiny26 Implementation

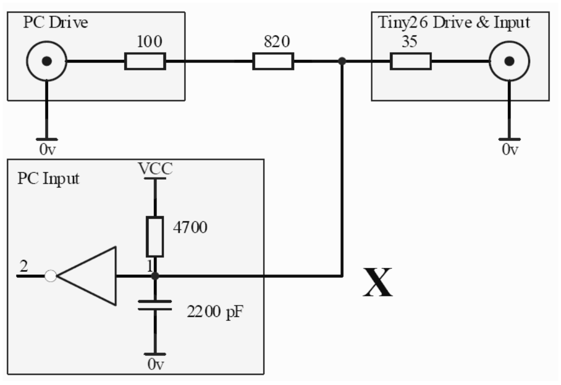
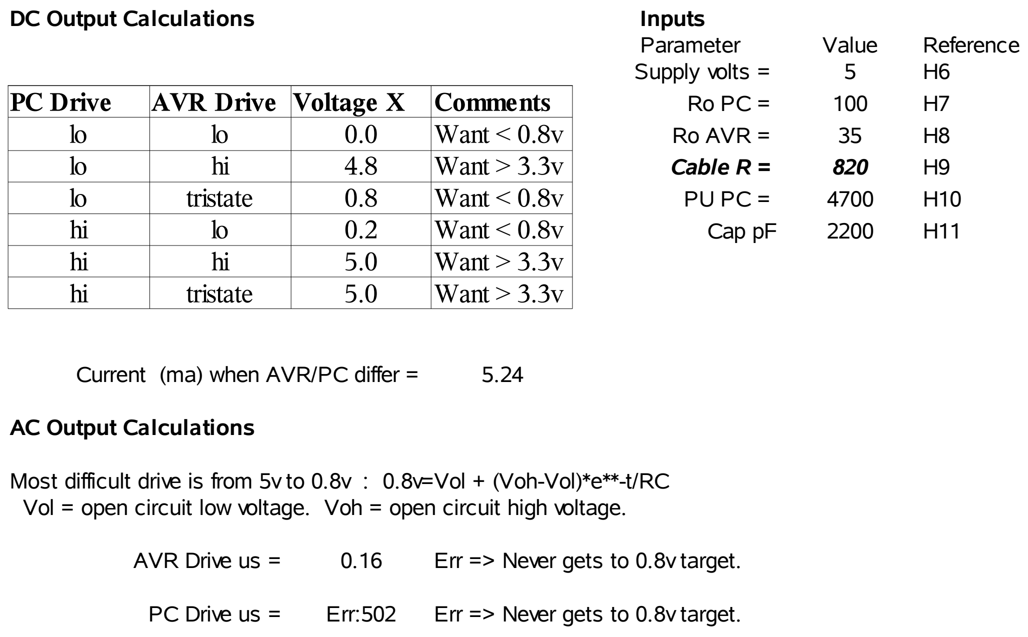


Illustration 4.11: Weak Drive Calculations

Logic loading calculations : the weak drive nature of each wire of the “2 wires” requires careful analysis. The illustration opposite models what happens when the PC and the Tiny26 both attempt to drive a wire.

The spreadsheet calculations below show that an 820 ohm line resistor allows all DC voltages to be satisfactory with the most marginal situation of PC driving low, and Tiny26 tristate, just giving 0.8v as a legitimate low on the line. An AC analysis shows an 820 ohm resistor is not acceptable because the line voltage never quite makes it to 0.8 volts. In reality most logic gates switch at around 1.1 volts so the circuit will just work but it is marginal. It was found that TIA worked well but the programmer which uses the same pins did not work reliably.



DC Output Calculations

PC Drive	AVR Drive	Voltage X	Comments
lo	lo	0.0	Want < 0.8v
lo	hi	4.8	Want > 3.3v
lo	tristate	0.8	Want < 0.8v
hi	lo	0.2	Want < 0.8v
hi	hi	5.0	Want > 3.3v
hi	tristate	5.0	Want > 3.3v

Inputs

Parameter	Value	Reference
Supply volts =	5	H6
Ro PC =	100	H7
Ro AVR =	35	H8
Cable R =	***820***	H9
PU PC =	4700	H10
Cap pF	2200	H11

Current (ma) when AVR/PC differ = 5.24

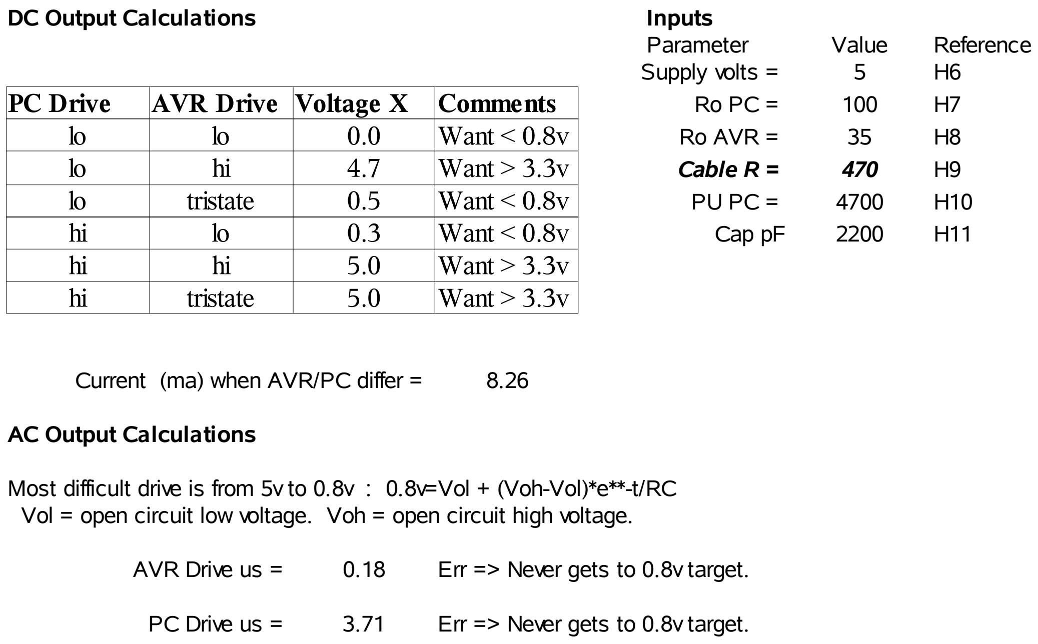
AC Output Calculations

Most difficult drive is from 5v to 0.8v : 0.8v=Vol + (Voh-Vol)*e**-t/RC
Vol = open circuit low voltage. Voh = open circuit high voltage.

AVR Drive us =	0.16	Err => Never gets to 0.8v target.
PC Drive us =	Err:502	Err => Never gets to 0.8v target.

Illustration 4.12: TIA AC and DC Calculations #1 for Tiny26 and PC

Using a line resistor of 470 ohms gave a better result for AC and DC analysis and the programming function works properly.



DC Output Calculations

PC Drive	AVR Drive	Voltage X	Comments
lo	lo	0.0	Want < 0.8v
lo	hi	4.7	Want > 3.3v
lo	tristate	0.5	Want < 0.8v
hi	lo	0.3	Want < 0.8v
hi	hi	5.0	Want > 3.3v
hi	tristate	5.0	Want > 3.3v

Inputs

Parameter	Value	Reference
Supply volts =	5	H6
Ro PC =	100	H7
Ro AVR =	35	H8
Cable R =	***470***	H9
PU PC =	4700	H10
Cap pF	2200	H11

Current (ma) when AVR/PC differ = 8.26

AC Output Calculations

Most difficult drive is from 5v to 0.8v : 0.8v=Vol + (Voh-Vol)*e**-t/RC
Vol = open circuit low voltage. Voh = open circuit high voltage.

AVR Drive us =	0.18	Err => Never gets to 0.8v target.
PC Drive us =	3.71	Err => Never gets to 0.8v target.

Illustration 4.13: TIA AC and DC Calculations #2 for Tiny26 and PC

4.5.2 Test Setup

The literature survey on protocol verification in section 3.1 derived a number of practical tests that could be performed on a protocol test bed. This section shows how the tests can be applied to the circuits given in this section. The test setup is defined by the following illustration which has the PC as a 2-wire TIA master on the left and the Tiny26 as a 2-wire TIA slave on the right. The tests are aimed at stressing normal operation (bottom set of illustration comments), and separately inducing a variety of faults and initial system states (top set of illustration comments).

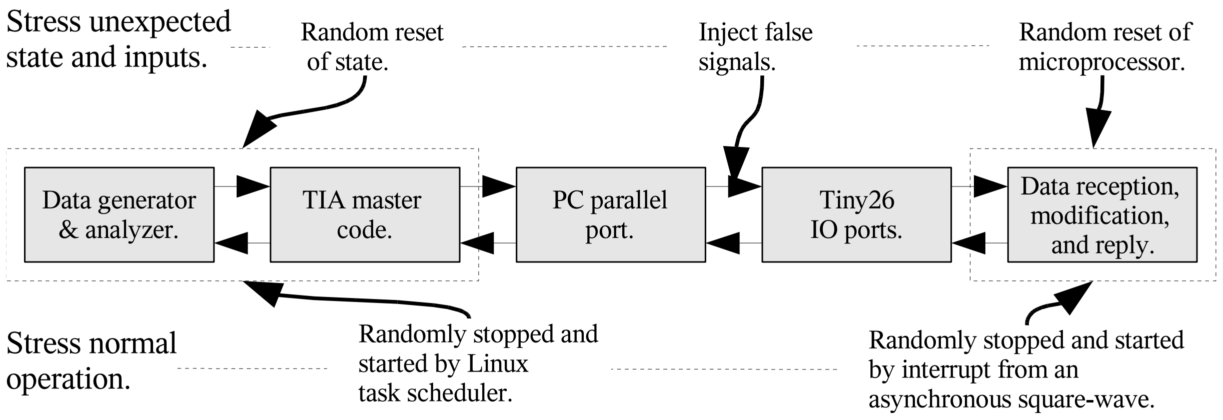

Stress unexpected state and inputs.
Random reset of state.
Inject false signals.
Random reset of microprocessor.
Data generator & analyzer.
TIA master code.
PC parallel port.
Tiny26 IO ports.
Data reception, modification, and reply.
Stress normal operation.
Randomly stopped and started by Linux task scheduler.
Randomly stopped and started by interrupt from an asynchronous square-wave.

Illustration 4.14: Block diagram of the 2-wire implementation & stress tests applied.

Normal operation stressing should never result in faulty data transfer, livelock or deadlock. Tests include-

Combination Testing : various combinations of data and initial conditions must be tested. In the PC-Tiny26 system this can be checked using an echo program whereby the master sends data to the slave and expects a modified return of the data. Timing variations may be added by combining with other tests such as platform variations and random testing.

Loading testing : the IBM-PC and Tiny26 can each have a variety of other loads applied.

- The master control program on the PC can be run at a variety of priorities relative to other application programs. This will effected timing in a random manner.
- The Tiny26 may be interrupted by an external source and so stop and start the slave control program in a random manner.

Platform variation : testing will be confined to IBM-PC running Fedora Core 3 as a master and the Tiny26 as a slave. The 2-wire TIA system will be transported to other systems but this is beyond the scope of this thesis.
Platform variations of several types can still be tested-

- The clock frequency of the Tiny26 can be easily varied from 1 MHz to 20 MHz.
- The master process or other processes on the PC can be given lower or higher priority under Linux.
- The Tiny26 can be programmed to respond to an external rising interrupt and return to the background code (which drives the slave communication routines) only when the input signal returns low. A variety of signals with different periods and duty cycles can be applied to the Tiny26 interrupt and so cause significant timing variation.
- On the PC other applications, at a variety of priorities can be run to vary the time available to the 2-wire TIA system.

Random Testing : the timing of the 2-wire TIA system is randomized to a significant degree in several ways and as a result no separate random testing is required.

- The process scheduling of Linux will start and stop the master code at random points.
- The proposed interrupt to the Tiny26 can be random relative to the Tiny26 and PC. The Tiny26 clock is independent to the PC and so the slave is random with reference to the master.

Fault induction generates false signals and forces the system state to a variety of random states. Data transfer errors are expected but no deadlock or livelock should occur. Tests include-

Forged signals causes random data corruption. Unlike most other tests data errors are not at issue. The aim is to test if the system ever exhibits livelock or deadlock. External signal generators can be used to apply corruption to the 2 wires.

Random resets cause the PC or Tiny26 to be reset at some random part of the TIA cycle.

4.5.3 Testing Results

The 2-wire TIA circuit was tested, as described in subsection 4.5.2, and the key result is that there are no surprises. The real unit worked exactly the same as the simulation suggested it would. The only new information derived was the data speeds achievable between the Tiny26 and PC. The test results matched the simulation in all aspects-

> *Normal operation* is always correct with no livelock or deadlock with the exception of the identified transient livelock.
>
> *Any initial state and false signals* could result in a deadlock that could be resolved by a timeout on the master back to idle.

The data transfer capability of the system is shown in the table below.

Table 4.1: 2-wire TIA Performance between a PC and an Atmel Tiny26

Master (PC) condition (1.6 Ghz Celeron, Fedora Core 3, KDE desktop)	**Data Transfer Rate kb/sec both ways, for Atmel Tiny26 with Clock in MHz.**					
	1 MHz	4 MHz	8 MHz	12 MHz	16 MHz	20 MHz
Linux priority -20 (highest) only KDE desktop running.	11	32	44	46	53	41
Linux priority 0 (normal) only KDE desktop running.	11	32	41	46	53	40
Linux priority +20 (lowest) only KDE desktop running.	11	30	41	44	50	37
Linux priority -20 (highest) KDE and hang-up running.	10	28	40	42	48	35
Linux priority 0 (normal) KDE and hang-up running.	5.7	16	22	23	25	20
Linux priority +20 (lowest) KDE and hang-up running.	0.52	1.5	2.0	2.1	2.2	1.7

Note an unexpected drop in throughput when going from a clock speed of 16 MHz to 20 MHz. This is due to the presence of live lock which will be discussed in detail later in this section. Livelock was predicted by the simulation in section 4.4.2, "Simulation of Normal Operation".

In the preceding table the slave is 100% available for TIA data transfer, but the master is stressed in two different ways. The first is that the master is run at priorities of -20 (maximum priority under Linux), 0 (normal user level priority), and +20 (lowest priority). These three combinations are also run with a hang-up program running at user level (priority 0). A hang-up program is a simple forever loop that chews up as much time as the operating system will allow it.

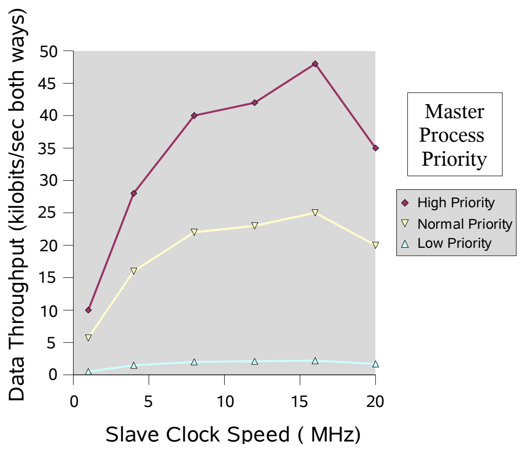


Illustration 4.15: Data Throughput with Stressed Master

The illustration opposite shows the data throughput for the master stressed with the hang-up program which corresponds to the bottom half of table 4.1 above.

The slave could was also stressed with a 50% duty cycle square-wave which froze the slave while the waveform was high. When using frequencies from 10 Hz to 100 kHz all data speeds presented in table 4.1 above were exactly halved as would be expected with a 50% duty cycle square-wave.

Livelock : the only surprise in the data throughput measurements is the drop in throughput going from 16 MHz to 20 MHz as shown in table 4.15 above. This oddity was investigated using a 20 MHz slave clock but changing the slave scanning loop that produces livelock. A variable delay was added that could extend the scan period.

The code that implements these tests can be seen on the CDROM accompanying this thesis-

Master code : byte_2_wire.c version 1.2

Slave code : avr_byte_echo V1.2

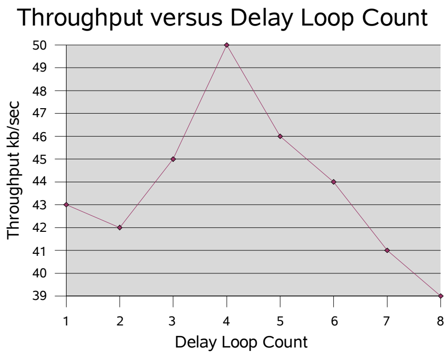
The results can be seen in the table opposite and graphed in the illustration below. One would expect the data throughput to drop as to delay loop counter increased, instead from count 2 to 4 the throughput increases. The existence of livelock could explain this anomaly.

Scan high period loop count.	**Transfer kb/sec**	**Scan period high/ total us.**
1	43	0.6 / 1.8
2	42	1.0 / 2.2
3	45	1.2 / 2.4
4	50	1.5 / 2.7
5	46	1.7 / 3.0
6	44	2.0 / 3.2
7	41	2.2 / 3.5
8	39	2.5 / 3.7

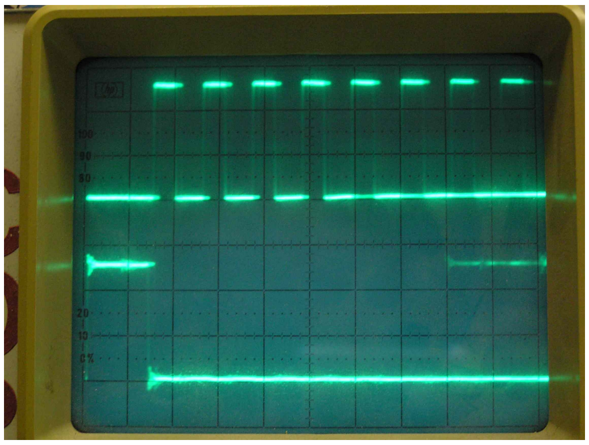
Table 4.2: Slave Scan Delays versus Data Throughput

The livelock can also be seen on the CRO pictures in the illustration below. Each picture is an accumulation of many data transfer cycles thus the brighter the trace the more common that behavior.

Illustration 4.16: Livelock Throughput Versus Delays



When the delay count is one there are 4 slave scans which always occur as shown by the pulses being high or low but not both. After 4 cycles however the slave scanning sometimes finishes (as shown by a scan being both high and low) but it usually continues – the livelock is quite long lived. When the delay count is 4 there are three cycles of slave count that always occur and then a very few after that indicating that the livelock either does not occur or is relatively short lived.


Delay Count = 1

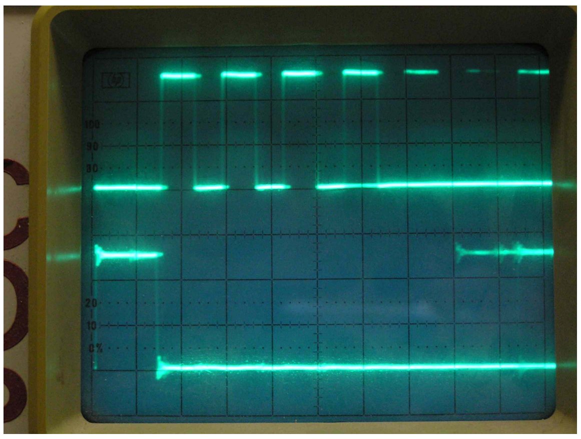

Delay Count = 4

Illustration 4.17: Livelock CRO Traces

The livelock predicted by the simulation definitely exists as can be demonstrated by data throughput anomalies and direct observation with a CRO. The livelock is easy to observe and can be largely eliminated by changing the delay in the slave scanning loop, or indeed the master scanning loop, that is responsible for the livelock.

4.6 Summary and Conclusions

This chapter is the culmination of the previous chapters. Chapter 2 proposed TIA as a communications category that may be useful for Low End Microprocessors and discovered that the literature describes 3 wire TIA protocols. It also posed a challenge to create a 2-wire bidirectional TIA protocol. Chapter 3 surveyed the literature to find what properties of a protocol should be verified and what modelling technique should be used. This chapter used these discoveries to propose a 2-wire TIA protocol which was modelled, simulated, implement and tested.

Section 4.3 modelled 2-wire TIA model using the STG-FT approached developed in section 3.2.3.

Section 4.4 described a custom simulator written to suit the STG-FT modelling and optimized for speed given the warning about state explosion found in chapter 3. Reachability tree pruning methods were examined in section 3.2.4 and appropriate lessons have been integrated into the simulator.

In section 4.4.2 the simulator was shown to achieved a respectable throughput of 4.8 million nodes per second but this was barely adequate for analyzing normal operation which took 14.5 hours and had to examine some 32 billion nodes. This approach would not work for the “any state/input” test which was estimated to take 3.5 years. The solution was to develop a novel reachability pruning technique called “state tick off” that dramatically reduced simulation time by a factor of approximately 250 million. Section 4.4.3 described the now feasible test for any state/input and showed a potential deadlock state that could be eliminated by having a timeout on the master which forced the master state to idle.

Section 4.4 on simulation has proved the 2-wire TIA protocol in normal operation is correct, contains no deadlock, and contains one livelock. If the system is put into any state with any input then a timeout on the master is required to avoid deadlock and livelock.

In section 4.5 2-wire TIA was implemented using an IBM-PC running Linux and an Atmel Tiny26 microprocessor. This was thoroughly tested as planned in section 3.1.2 and showed the real system worked exactly as the simulation predicted. Data throughput peaked at approximately 50 kilobits per second in both directions. The livelock was observed as an anomaly in data throughput when a master and slave loop period happened to match. The livelock predicted by simulation can be made largely irrelevant by ensuring the period of these two loops is different.

2-wire TIA has been proved to be a realistic communications protocol and ideally suited to Low End Microprocessors.

5 2B-2B Variant of 2-wire Time Independent Asynchronous Protocol

***Overview** : the 2-wire TIA protocol proposed previously in this thesis requires the master to have two weak drive pins, or use 4 pins and two resistors to support 2-wire TIA. This chapter proposes and verifies a 2-wire TIA variant where the master can use two standard IO pins. The new 2-wire TIA has been labelled 2B-2B TIA to signify that master and slave each require two general purpose bidirectional IO pins*

The original 2-wire TIA described so far in this thesis relied on a weak drive from the master. Practical implementation of a single weak drive requires either a specially constructed weak drive IO pin, or two general purpose IO pins and a resistor. The former solution requires changes at the silicon level, the latter solution requires 4 IO pins on the master to support a 2-wire system. Neither of these solutions is ideal. This chapter proposes a 2-wire TIA variant that allows the master to use two standard IO pins thus avoiding the implementation difficulties identified.

The original 2-wire TIA would be better suited to communications from a microprocessor to a device that has only standard digital inputs and outputs, for example the parallel printer port of a standard personal computer. The TIA variant presented here would better suit microprocessor to microprocessor communications.

The two TIA variants need to be labelled to avoid confusion. The original TIA will be labelled 2I2O-2B to indicate that the master requires 2 Inputs and 2 Outputs whereas the slave requires 2 general purpose Bidirectional pins. The variant examined in this chapter will be labelled 2B-2B to indicate that both master and slave require 2 general purpose Bidirectional pins.

It will be shown that the 2B-2B TIA has the same properties as the original 4P-2B TIA : it is correct apart from an avoidable livelock, and can cope with any initial state and unexpected input if the master has a timeout.

This chapter is structured as follows. Section 5.1 describes the new protocol using a timing waveform. Section 5.2 describes the protocol as an STG-FT model and simulates that model to determine the protocol properties. Section 5.3 will compare the results from the two TIA protocols.

5.1 2B-2B Waveforms

Apart from the drive changes the 2B-2B TIA variant differs from the original in the final stages of the bit transfer. The slave, not the master, is responsible for asserting the inverse of the original slave data (note timing mark s6).

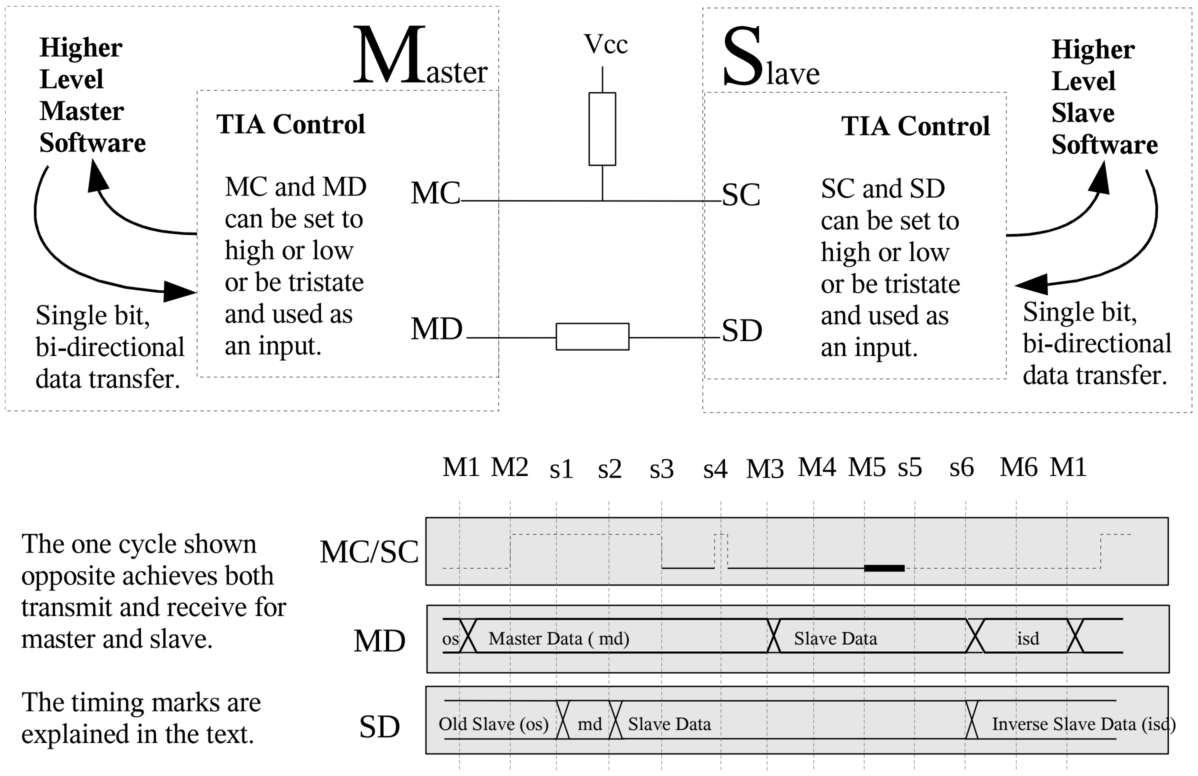

Higher Level Master Software
Master
TIA Control
MC and MD can be set to high or low or be tristate and used as an input.
MC
MD
Single bit, bi-directional data transfer.
Vcc
Slave
TIA Control
SC and SD can be set to high or low or be tristate and used as an input.
SC
SD
Higher Level Slave Software
Single bit, bi-directional data transfer.
M1 M2 s1 s2 s3 s4 M3 M4 M5 s5 s6 M6 M1
The one cycle shown opposite achieves both transmit and receive for master and slave.
The timing marks are explained in the text.
MC/SC
MD
SD
os
Master Data (md)
Slave Data
isd
Old Slave (os)
md
Slave Data
Inverse Slave Data (isd)

Illustration 5.1: Timing Waveforms for 2B-2B TIA

One cycle again transmits one bit from master to slave, and another bit from the slave to the master. The MC/MD lines work as an open collector (or drain) arrangement. A dotted line on MC/SC means the master is driving low, or master and slave are tristated. A thin solid line means the slave is driving MC/SC low. A thick single line means master and slave are driving MC/SC low.

The waveforms work as follows-

- M1 : SD is still outputting old data from the slave.
  MC is driving low, SC is tristate.
  MD asserts the new data from the master to the slave.
- M2 : the master tells the slave new MD data is ready by tristating MC thus causing MC/SC to go high.
- s1: the slave notes MC/SC has gone high to start a new cycle and tristates SD so the master data can be seen on SD.
- s2 : the slave reads in the master data and asserts its own data on SD.
- s3 : the slave sets SC low to show it has read the master data and has asserted its own data.
- s4 : the slave goes tristate to see if the master has read the slave data yet, because MC/SC goes high the master has not read the slave data.
- M3 : the master reads MC/SC and notes the slave is driving it low, and tristates MD so the slave data can be read from MD.
- M4 : the master reads the slave data.
- M5 : the master drives MC low to show it has read the slave data.
- s5 : the slave again polls the master on MC by tristating SC but it finds SC held low by the master. SC is left in tristate.
- s6 : the slave puts an inverted version of its data on SD to signify it is finished.
- M6 : the master reads the newly inverted slave data on MD and notes the slave has finished.
- M1 : the cycle starts again.

5.2 STG-FT Model & Simulation

Work on the initial 2-wire TIA system in section 3.2 and chapter 4 showed how modelling of the system is best performed using STG-FT. The STG-FT model for the 2B-2B TIA system in the illustration below has much in common with the original 2-wire model.

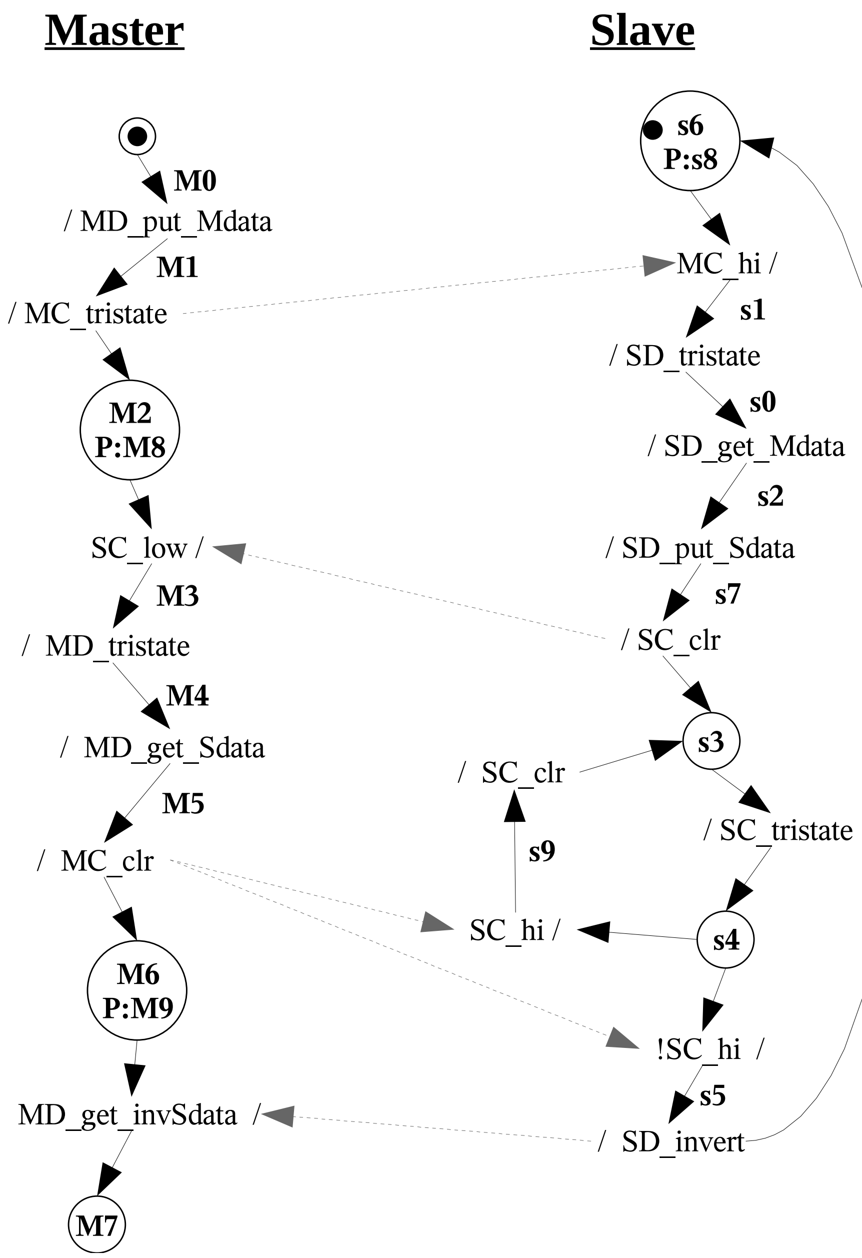


Illustration 5.2: 2IO-2IO TIA STG-FT Model

The original TIA was simulated using the STG-FT simulator developed by this author. The same simulator has been used to validate the new 2B-2B TIA protocol. The careful structure of the simulator has proved its value as only the problem definition header files needed to be modified and the generic code modules remain unchanged.

The simulation results are very similar to the original TIA protocol explained in chapter 4.

Brute force & pruning : full reachability form one initial state requires the analysis of 16,000,000,000 nodes which took 51 minutes on an Acer Travelmate 4600 (1.6 Ghz Celeron) running Fedora Core 3 . The pruning algorithm developed and described earlier in this thesis reduced the search to 115 nodes which took 1.5 milliseconds.
Evaluation of all possible system states (the cross product of all subsystem states) requires the analysis of 14400 initial states and took some 10 seconds using pruning. It would have taken approximately 500 days using brute force.

Livelock : as with 2I2O-2B TIA there is a potential livelock, this can be seen on the STG-FT diagram between master states M2/M8 and the slave loop s3,s4,s9. This livelock can only exist if the master and slave loops have identical timing and stay in sync. The livelock can be avoided by ensuring the loops do not have identical delays.

Correct : the system will work in all circumstances (apart from the livelock) given a correct initial state.

Any initial state / unexpected inputs : given all possible initial states and inputs there must be a timeout on the master to avoid deadlock.

<u>5.3 Summary and Conclusions</u>

A variant of 2-wire TIA has been created that better suits microprocessor to microprocessor communications as the master (and slave) require only two standard IO pins. The original TIA is now labelled 2I2O-2B TIA (the master has 2 inputs, 2 outputs, and the slave 2 bidirectional pins). The new TIA is labelled 2B-2B (master and slave require 2 bidirectional pins).

In this chapter the new 2B-2B TIA has been described using timing waveforms, modelled using STG-FT, and simulated using the STG-FT simulator described in section 4.4. The new variant is very similar to the original and simulation shows the same result. The protocol has one avoidable livelock, no deadlocks, and is always correct. Given any initial state and input there is a deadlock which can be eliminated by applying a timeout on the master which forces the master back to the idle state.

The 2B-2B TIA has only been simulated and has not been physically tested for performance as yet. There is little doubt that the simulation results will be observed in the real implementation. It will be interesting to discover the data throughput and the exact nature of the livelock.

Two variants of 2-wire TIA have been discovered and there may well be others. There are other challenges to investigate as well. Is it possible to create a one wire TIA? Can a 2-wire TIA be developed that is a bus not just a point to point protocol? These are interesting research challenges which may have practical application.

6 Practical Use as an Embedded Debugger

Overview *: the debugging of software takes considerable labor time and many methods of debugging have been developed. This chapter examines the families of debugging tools applicable to low end microprocessors. It is found that the monitor ROM debugger is potentially useful but limitations preclude its use in most circumstances. The 2-wire TIA protocol is found to solve most of the problems and a TIA based ROM debugger is implemented and described for the Tiny26 microprocessor. This solution should be very useful where ever the superior but more expensive solutions such as In Circuit Emulators (ICE) cannot be afforded.*

The testing and fixing (debugging) of software code can consume huge amounts of labor time. Atif Memon (2002) states that testing complex software can consume 50%-60% of a labor budget, and that 30%-40% is the lower limit for any type of project. Many other respected authors such as Beizer (1990) echo this view.

The respected software estimation tool COCOMO 1 from Barry Boehm (1981) offers a quantitate insight into how various factors effect the labor costs of a project. COCOMO 1 uses multiplicative cost drivers to derive labor times. With reference to table 6.1 that follows, people factors are clearly very important. For example moving from a very highly skilled analyst (factor 0.71) to a very poorly skilled analyst (factor 1.46) can effectively double the total labor cost of a project. Other people factors such as application experience and programmer skills are likewise very important.

The role of tools can be seen in the multiplicative constant TOOL. Moving from a suite of poor tools (factor 1.24) to very good tools (factor 0.83) can reduce labor costs by one third. Debugging tools are an essential link in the development tool set and so using or improving debuggers can result in significant labor savings.

Cost Driver (see Boehm for exact interpretation.)	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY - Software reliability required.	0.75	0.88	1.00	1.15	1.40	
DATA - Database size		0.94	1.00	1.08	1.16	
CPLX - Product complexity.	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME - Execution time constraints.			1.00	1.11	1.30	1.66
STOR - Main storage constraints			1.00	1.06	1.21	1.56
VIRT - Virtual machine volatility.		0.87	1.00	1.15	1.30	
TURN - Computer turnaround time.		0.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP - Analyst capability.	1.46	1.19	1.00	0.86	0.71	
AEXP - Applications experience.	1.29	1.13	1.00	0.91	0.82	
PCAP - Programmer capability.	1.42	1.17	1.00	0.86	0.70	
VEXP - Virtual machine experience.	1.21	1.10	1.00	0.90		
LEXP - Language programming experience	1.14	1.07	1.00	0.95		
Project Tools						
MODP - Modern programming practices.	1.24	1.10	1.00	0.91	0.82	
TOOL - Software tools used.	1.24	1.10	1.00	0.91	0.83	
Schedule						
SCED - Required development schedule	1.23	1.08	1.00	1.04	1.10	
(% of nominal schedule for SCED)	(75%)	(85%)	(100%)	(130%)	(160%)	
Base Environment Factor *						
TBEF - Team Base Environment Factor			see	later		

Table 6.1: COCOMO 1 Cost Drivers

This chapter will examine a range of debugging tools available for Low End Microprocessors and show how 2-wire TIA can be used as the basis of a novel debugger.

The chapter is structured as follows. Section 6.1 surveys existing debugging tools and comments on how applicable each tool is Low End Microprocessors. Section 6.2 looks at the relevance of monitor ROM style debuggers and how 2-wire TIA can overcome many of the problems associated with this form of debugging. Section 6.3 reports on the implementation of a monitor ROM debugger using 2-wire TIA. Section 6.4 provides a summary and conclusions for the chapter.

6.1 Existing Microprocessor Debugging Tools

In 1971 Intel corporation developed what is generally accepted as the the world's first microprocessor, the 4004. There is some debate about who were the developers, the key patents for a microprocessor (US patents 3,821,715 and 3,753,011) are signed by Ted Hoff, Stan Mazer and Federico Faggin. Intel initially removed Faggin's name (Faggin 2001) but has now restored it. In 1973 Intel delivered the Intellec 4, a complete development system that included an assembler, a linker, and debugger, in short a full Microprocessor Development System (MDS). See www.old-computers.com for pictures and feature descriptions of these and later Intel MDS systems.

In 1979 this author, as a very young engineer, developed 8080 applications on the next generation of Intel MDS systems the MDS-800. These systems enabled developers to create products much more quickly and so boosted the sales of Intel's chips and helped it become a premier semiconductor company. The author has been involved with debugging tools in industry and education up to the present time.

Debugger families : there are several families of debugging tools that can be used on microprocessors-

Ad-hoc approaches use side effects and writing to IO pins to help understand what the microprocessor is doing.

Monitor ROM based programs add code to the system, and use hardware resources such as serial ports to talk to display and command devices.

In Circuit Emulators (ICE) replace the microprocessor with a unit which emulates the microprocessor.

Boundary scan systems use serial scan paths to capture or insert key signals and data.

Microprocessor simulators run on another computer and simulate the microprocessor hardware and in some cases other hardware as well.

Debugger programs such as the UNIX gdb can be used on larger microprocessor based systems.

Built in debug : some microprocessors have built in hardware that aids debugging such as special purpose debug registers.

6.1.1 Ad-hoc Approaches

Ad-hoc microprocessor debugging uses visible outputs to gauge what the microprocessor is doing. These outputs may be natural outputs of the microprocessor or specially programmed into the code specifically for the purposes of debugging. For example a spare IO pin may be set or cleared if a certain line of code is reached. Another common options is to add an LCD display to indicate the state of the software.

Advantages of the ad-hoc approach include that they are cheap as little or no extra hardware is required. Real time performance can be observed if a CRO is attached to the IO pin.

Disadvantages of the ad-hoc approach include-

- *Code changes* : All debugging must be designed and then code added to the application. This code must then be reliably removed to avoid interference with the application or later debugging code.
- *Frustrating* : From personal experience, and that of many students, if unexpected things are observed then the user if left with no idea of what is happening.
- *Slow* : The above problems slow down the debugging activity and so increase labor costs.
- *Hardware used* : Hardware such as port pins must be dedicated to debugging and are not available for the application.
- *Data oriented debugging* such as viewing, writing or break-pointing registers and data is not possible.

The biggest problem with this approach is the slow pace of debugging and the high labor costs. From observing many student projects, the ad-hoc debugging approach is much worse than other debugging approaches except for trivial applications.

6.1.2 Monitor ROM Debuggers

As discussed at the start of this chapter the Intel Intellect 4 provided a complete development system for the Intel 4004 microprocessor. It contained input switches and displays and a ROM who's program content enabled basic control of the Intel 4004 microprocessor. This monitor ROM approach has been widely used since, for example Vallejo (1992) and the well known Buffalo Monitor for the Motorola 68HC11. Like many other monitors the Buffalo monitor uses code in ROM and the 68HC11 UART serial port on the microprocessor to talk to a display and command device. Simple commands allow the user to read and write RAM, set and observe breakpoints, and call functions.

This author, starting in 1979, has created monitors for microprocessors including the 8048, 8085, 8051, Atmel AVR series, Microchip 16C series. Illustration 6.1 that follows is derived from a 1982 development by this author for Ericsson Australia, an MFC signalling controller in the

Ericsson ASDP-162 telephone queuing system. It shows a monitor ROM debugger called RADBUG that also added a system level debugging capability to the standard monitor functions. System level debugging is one level above symbolic level debugging, it works with application level events such as "telephone line released" and "exchange timeout" not source code events such as "line 632 executed". In reference to illustration 6.1 the application reported system level events to RADBUG, that after interpretation would be displayed on a dumb terminal. The ability to see system and application level interpretation of events in a compact manner was essential to the development and debugging of the total system. It was not possible to get this information unless the debugger was intimately tied to the source code of the application. Telecom Australia (now Telstra) purchased the software in 1984 order to debug their own telephone exchanges.

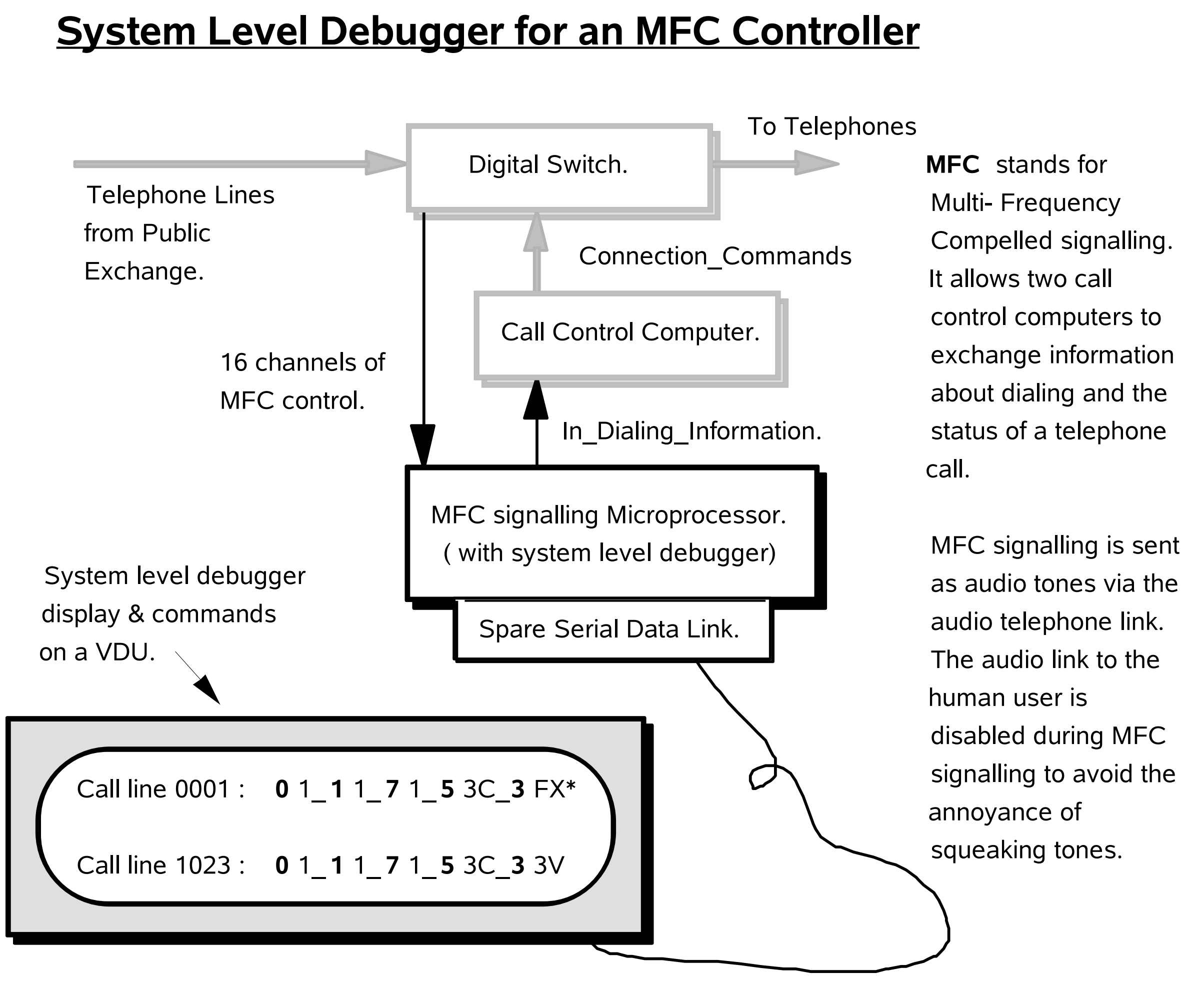

System Level Debugger for an MFC Controller
To Telephones
Digital Switch.
Telephone Lines from Public Exchange.
Connection_Commands
Call Control Computer.
16 channels of MFC control.
In_Dialing_Information.
MFC signalling Microprocessor. (with system level debugger)
Spare Serial Data Link.
System level debugger display & commands on a VDU.
Call line 0001 : 0 1_1 1_7 1_5 3C_3 FX*
Call line 1023 : 0 1_1 1_7 1_5 3C_3 3V
MFC stands for Multi- Frequency Compelled signalling. It allows two call control computers to exchange information about dialing and the status of a telephone call.
MFC signalling is sent as audio tones via the audio telephone link. The audio link to the human user is disabled during MFC signalling to avoid the annoyance of squeaking tones.

Illustration 6.1: Monitor with System Level Debugger

Many monitor ROM debuggers such as the Buffalo monitor and RADBUG work with a dumb display. An alternative is to use a personal computer (PC) to provide intelligent display an so minimize the size of the debugger on the target machine. The microprocessor debugger can work in binary not ASCII and need provide minimal services. Such an approach is most desirable for low end microprocessors where code size is frequently an issue.

A intelligent display can provide many features such as a GUI interface, symbolic display instead of binary display, and call stack display (if the stack can be read). The ease of use and quality of the displayed information can be significantly better than that provided by a monitor ROM alone.

From the author's experience and that of other practitioners such Beach (1999) the advantages and disadvantages of monitors are reasonably obvious-

Advantages of Monitor ROM systems include-

- *Cheap* as the monitor code can reside in a cheap ROM, perhaps even the same ROM as the application.
- *Field testing* : Since the monitor is embedded with the application it can be available at every site and may facilitate field testing.
- *Easy to create* : A monitor program is not difficult to create.
- *System level debugging* : The application code can call the monitor when a system level decision or event occurs. This level of observation or debugging is a level above symbolic debugging and can be very powerful for system testing.
  The Ericsson ASDP-162 example has already been discussed, another example is the power train tester developed by Ford (Toeppe, Ranville, and Butts 1998).

Disadvantages of Monitor ROM systems include-

- *Target resources used* : The monitor program may use target system resources including RAM and ROM memory, timers, a serial port to talk to a display system, and CPU time. This is a major problem in low end microprocessors where resources are minimal.

- *Application effected* : the monitor activity, particularly the reception and transmission of information on the serial link, may interfere with the application. This is particularly true if the application makes heavy use of CPU capacity or can impede response to interrupts.
- *Extra resources* may be required that would not normally be added to a product. For example if the application requires a serial link then a second serial link must be added for the monitor program.
  If the extra serial link is implemented in hardware then a more expensive microprocessor must be chosen. If a serial link is created using only software and several IO pins then significant CPU time and code space must be stolen from the application along with a timer or interrupt.
- *Fewer advanced features* are available compared to In Circuit Emulators (see later in this section). For example the ability to back trace the execution path may be impossible or at best significantly slow down execution and use up significant RAM.

The main advantage of a monitor debugger for Low End Microprocessors is its low cost and the ability to build in system level debug capacity. The main disadvantages are the usage of limited resources including memory and serial link, and the interference with the application. If any of these disadvantages can be eliminated then monitor based debuggers may find greater usage in low end microprocessor systems.

6.1.3 In Circuit Emulators

An In Circuit Emulator requires that the microprocessor be removed from its socket, and be replaced with a unit that emulates the microprocessor. Most emulators are a combination of a powerful logic analyzer and control system coupled with the target microprocessor as a bonded die with access to test points. The logic analyzer observes all bus transactions and works out what line of code is being executed and what data is being written or read. The control system can stop the microprocessor or force it to execute a particular instruction.

The first powerful emulator was the Intel MDS-800 that was launched in 1975. These were very expensive, from this author's knowledge in 1978 Ericsson Australia got a special deal and bought an MDS-800 for AU$25,000 which was nearly twice the salary of a graduate engineer. The features were truly grounding breaking for the time and radically reduced development and debugging time. See www.old-computers.com for pictures and feature description of these and later Intel MDS systems.

Features of ICE systems include-

- Memory : Able to simulate non-existent memory of either ROM or RAM nature.
- Read and write to memory in binary or symbolic manner.
- Execution can be started and any point (with parameters for functions) and stopped when desired. Single stepping of source or assembler code is possible.
- Breakpoints allow execution to be stopped given some condition related to the program counter, and data read or write.
- Pass points are like breakpoints except the emulator may capture a snapshot of desired information and then continue execution. After some condition the passpoint may disappear or become a breakpoint.
- State analysis allows breakpoints based on the current state plus some historical information.
- Back-trace allows a history of execution to be examined to see how the program got to the current point. This may be in binary form, the programming language, or bus waveforms.
- Loading and verification of program code and data can be performed.

Disadvantages of ICE systems include-

- *Cost* : The cost of an MDS back in 1978 was immense as already discussed.
  Today good quality ICE systems can be bought for as little as US$1600, for example those from Metalink (www.metaice.com). There are many cheaper "ICE" system advertised but these appear to be inferior JTAG based systems which are discussed in the next section.
  Each microprocessor variant may require additional hardware and software which adds to

the cost. Different microprocessor within the same family may require a completely different ICE.

- *System level debugging* , as discussed previous, is very difficult to achieve.
- *Power drain* of a system with an ICE is different to that with a real microprocessor. This makes it difficult to debug power sensitive applications or those within high voltage environments such as the 48v of a standard telephone line.
- *Availability* can be a problem as the ICE is special expensive hardware that cannot be available at every installation.

ICE systems are the pinnacle of debugger sophistication but their high cost and weakness at system level debugging can preclude their use in many circumstances. Commercial microprocessor development tends to prefer ICE systems as the higher purchase cost is more than offset by reduced labor costs.

6.1.4 Boundary Scan Systems

Boundary scan systems require that a microprocessor has special purpose hardware added at the design stage. This hardware has serial scan path registers that allow a serial link to capture data from key points in the microprocessor hardware, and also allow data to be inserted into key points. For example Winters (1994) used a IEEE-1149.1 JTAG serial link to debug the AMD 29K processor. While JTAG is often used to test

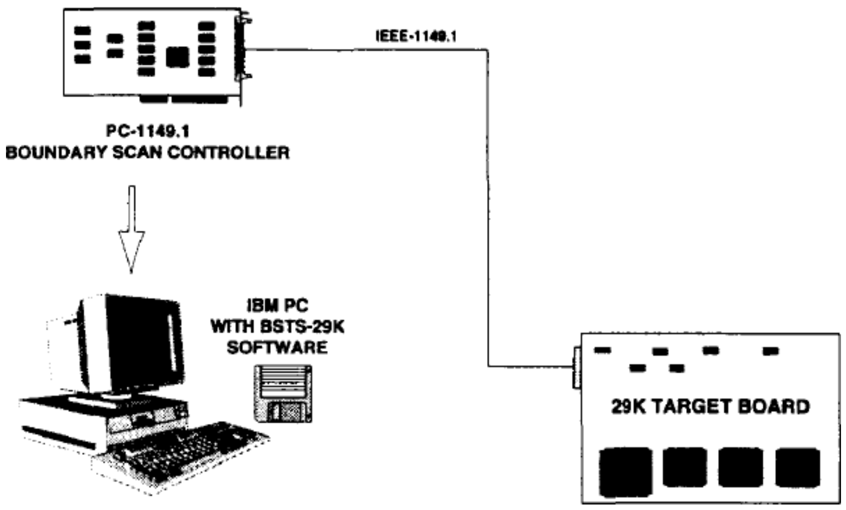


Illustration 6.2: Boundary Scan Debugger from Winters (1994)

hardware it can also be used to debug software as key processor states such as the address bus can be accessed through the serial interface.

A variety of other microprocessors such as the ARM, Coldfire, MIPS, Power PC, and AMD-K7, and higher end Atmel microprocessors have built in JTAG interfaces which can be used for software debugging.

Advantages : The target microprocessor can be debugged without addition of code or hardware changes.

Disadvantages are considerable for the Low End Microprocessor (LEM)-

- A serial scan path will add to the cost of a microprocessor, a noticeable sum for LEMs.
- Most serial scan paths are not fast enough to capture full speed operation of the microprocessor.
- A JTAG debug interface requires 6 IO pins, a considerable resource for an LEM.
- The scan path based debuggers can be expensive, though there are exceptions such as the Atmel JTAG debugger for under US$40 from ERE (www.ere.com.th).

Many JTAG, serial scan based systems are advertised as In Circuit Emulators. This is misleading as key features of an ICE, such as full speed operation, cannot be matched by serial scan based systems. Such systems may be better described as Background Mode Emulators (Hector 2002) or BMEs.

Serial scan path debuggers are generally not preferred for LEM debugging.

6.1.5 Microprocessor Simulators

Microprocessor simulators recreate the microprocessor in a software environment, usually on a PC. Software is used to simulate every aspect of the microprocessor, may account for external inputs and outputs (usually from file), and may simulate peripherals and other hardware.

Example products include-

- University developed simulators such as those from Smith (1996) and Fadul (1993).
- Open source simulators such as gpsim from www.dattalo.com/gnupic/gpsim.html and misim from www.feertech.com/misim/homepage.html
- Some simulators allow connection to real hardware via PC ports such as the parallel port. For example this author's project students created a package called picmicrosim at http://sourceforge.net/projects/picmicrosim.
- Commercial simulators from companies such as Oshonsoft www.oshonsoft.com and Microchips Mplab product www.microchipc.com.

The advantages of simulators include-

- *Full control and observability* : Every element of the microprocessor is simulated in software so every element can be displayed and controlled in a very powerful and complete manner. Very powerful breakpoint conditions can also be implemented.
- *Easy* : Simulators tend to be more approachable for students who have not used a microprocessor before. Real world issues such as power supplies and crystals can be ignored.

The disadvantages of simulators include-

- *Timing wrong* : Simulators never seem to match the exact timing of the microprocessor and so the behavior of non-trivial programs on the simulator seldom matches behavior on the real hardware.
- *Hardware limited* : The simulator can only cope with a very narrow range of other hardware such as switches for input and LEDs for output. It is usually difficult or impossible to add

to the simulator code to simulator application hardware. The simulator picmicrosim mentioned above stands out as an exception.

- *Microprocessor mismatch* : Few simulators can match the full range of microprocessor variants in a given family so one may be forced to use a microprocessor that is different to the one that will be used in the target.
- *No help* : If the real hardware does not work then the simulator cannot offer any help.

Microprocessor simulators do not appear to be widely used except in education where their ease of use and excellent display capabilities are valuable for students new to microprocessors. In most other situations the mismatch between the real microprocessor plus its real world interfaces and the simulator becomes a significant disadvantage which leads most developers to avoid simulators.

6.1.6 Debugger Programs

Some microprocessors have such a large memory that it is possible to add a sophisticated debugger program such as the UNIX gdb program by itself or its remote debug feature (see the gdb man pages). Some microprocessor systems will even take a full operating system such as QNX or an embedded Linux. In these cases debugging becomes much more like debugging on a PC.

The advantages of full debugger programs include-

- *Full debugger capability* is available via a sophisticated debugging program.

The disadvantages of full debugger programs include-

- *Memory requirements* can be very substantial, in the order of hundreds of kilobytes or more.
- *RAM needed* : many debugger programs require code to reside in RAM rather than ROM, EEPROM, or Flash RAM

The debugger program approach to debugging microprocessors is only relevant to very large microprocessor systems that have considerable memory capacity and come near to personal computer power in their own right. This approach makes little sense for Low End Microprocessors.

6.1.7 Built in Debugging

Built in debug (BID) refers to hardware or software that is added to the microprocessor by the manufacturer specifically for the purpose of debugging. Commonly there is a hardware link to a personal computer which handles the display and command aspect of the debugger. The Boundary Scan approach discussed in section 6.1.4 is one example of built in debugging. This section lists other BID approaches.

The x86 family of microprocessors, starting with the 80386, added breakpoint registers that could be accessible from application code. These registers allow an interrupt to be asserted if a condition about the program counter or data read and write becomes true. Application development programs such as the Microsoft and Borland compilers, and the UNIX gdb program make use of this hardware to provide powerful debuggers that run on the same machine as the application program.

Low end microprocessors such as the P51 from Cybernetic Micro Systems (www.controlchips.com) contain BID but the cost is usually rather more than an equivalent chip with no debugging. For example the P51 is US$12 in 10,000 quantities whereas equivalent chips with no debugging cost about $2.50, for example those from Ramtron (www.ramtron.com).

Advantages of BID include-

- *Cheap* compared to an ICE.

- *Powerful features* can be delivered using this approach.

Disadvantages of built in debugging include-

- *Expensive* : For low end microprocessors the BID becomes a non-trivial cost and will raise the price of the microprocessor.
- *Slow* : some approaches such as serial scan path slow down the target microprocessor or require it to run at a very low speed.
- *Availability* is limited to processors that the facility built it at manufacture.

Built in debug (BID) is a powerful concept that can provide powerful debugging. BID is uncommon in low end microprocessors because it adds costs to a cost sensitive product.

6.2 Low End Microprocessor Debuggers and TIA

The foregoing discussion has examined several families of debugging tools that are used on microprocessor systems. Two of these approaches stand out for Low End Microprocessors-

- In Circuit Emulators (ICE) systems can deliver more features and make minimal demands on microprocessor resources but the purchase cost can be significant.
- Monitor ROM based debuggers may have less features but come at minimal cost and can be quite adequate for debugging. Demands on microprocessor resources make it impractical for most low end microprocessor systems.

Can the limitations of the Monitor ROM debugger be minimized to make it more attractive? The table below details the main problems and potential solutions.

Monitor ROM Debugger Problem	**Solution**
Target RAM/ROM used.	Use minimal debugger code on the microprocessor and move everything possible to a PC.
Timer, serial port, interrupt resources used.	Use TIA communications so these resources are not required.
IO pin usage.	Use TIA communications as this will only require 2 general purpose IO pins not any specialized pins.
CPU time.	Use TIA communications as it can be run in background when CPU time is available.
Timing interference.	Use TIA communications as it makes no timing demands and will not cause any timing jitter in interrupt or high priority tasks.

Table 6.2: Solutions to Monitor ROM Debugger Problems

Clearly the use of TIA can make monitor ROM debugging much more feasible particularly for low end microprocessors.

The utility of TIA in monitor ROM debugging can be examined by proposing several case studies-

Case study 1 : the Atmel Tiny26 is a very useful and cheap microprocessor. It includes 1k instructions, 128 bytes RAM, 128 bytes EEPROM, two PWMs, ADC, two timers, and an SPI serial link. The PWM is particularly useful as it includes an internal 64 MHz clock and so the Tiny26 is often used in switching regulators. Unfortunately the serial link is not available if any PWM is used and so any application that uses the PWM cannot use monitor ROM based debugging.

The TIA communications approach will require only two general IO pins and a little ROM and RAM space. It will not interfere with the PWM or any other special hardware resource. Monitor ROM based debugging becomes a feasible option.

Case study 2 : the Microchip Corporation (2006) provides and excellent microprocessor comparison guide.

The PIC16F54 is low end microprocessor with 12 IO ports, no serial port and a timer that costs an amazing low US$0.44. For the purposes of monitor ROM debugging a serial link could be fabricated in software but this would use the timer or an interrupt, plus IO pins. This would appear to rule out monitor ROM debugging for most applications. The only solution for a monitor ROM debugger would be to chose a more expensive chip such as the PIC 16F631 which costs US$0.94.

The cheaper PIC16F54 could use a TIA based monitor ROM debugging at the cost of two IO pins and some program space. TIA makes a monitor ROM debugger a much more feasible option for the PIC16F54.

In all these cases the TIA communications makes a monitor ROM based debugger a feasible option. It is worth implementing and testing this approach to see if it is indeed as viable as the above reasoning would suggest.

6.3 TIA Based Debugger Implementation

Earlier in this thesis the Atmel Tiny26 microprocessor was used as a test-bed to prove the 2-wire TIA protocol would work. In this section the TIA communications link is extended to provide a debugger facility for the Tiny26. This approach overcomes the problems with monitor ROM based debuggers identified in the previous section.

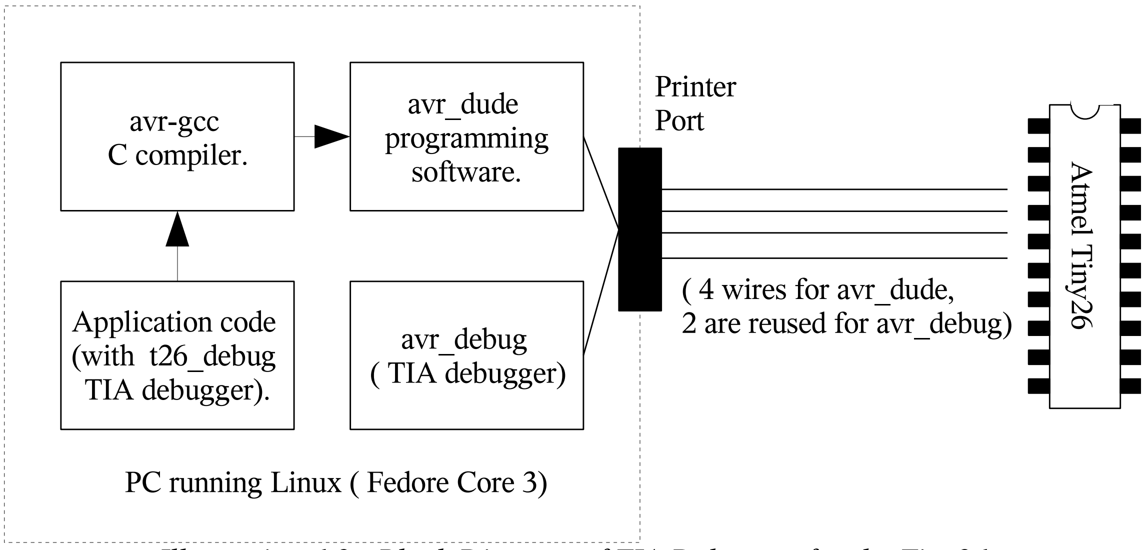


Illustration 6.3: Block Diagram of TIA Debugger for the Tiny26

The illustration above shows the solution path chosen for creating a Tiny26 development environment. Source code in C is compiled using the avr-gcc compiler (www.avrfreaks.net) and the Tiny26 programmed using avr_dude (www.nongnu.org/avrdude). The programmer avr_dude uses 4 pins on the Tiny26 to program the device.

The TIA debugging approach adds several elements which have all been created for this thesis. First the source code must have a minimal debugger and TIA code added, the small t26_debug module. The TIA debugger avr_debug runs on the PC, after programming of the Tiny26 has finished, and uses two pins of the four in the same programming cable required by avr_dude. This provides maximum convenience to the user as only one cable is required for both programming and debugging.

Software is added to the application software that runs on the Tiny26 to provide a TIA slave interface from the Tiny26 to the Linux box, and to provide basic debug facilities. The communication format is simple binary that results in minimal code size but is not human readable. The CDROM (in the directory t26_debug) shows a minimal software application that that consists of two modules-

- t26_debug.c : contains all the TIA code and the debugger code. It contains two items of interest to the application program that can be seen in t26_debug.h -
  - A function "breakpoint(int integer) ;" that can be called by the application code whenever a breakpoint is reached. The application source code must have a statement added at the desired breakpoint site-

    `breakpoint( id_byte) ;` // the id_byte identifies the breakpoint
  - A variable called event_byte can be treated as a byte or set of bits to report system level events. Whenever avr_debug on the PC performs any debugger function it reads and the clears event_byte. The source code addition is simply-

    `event_byte |= 0x01 ;`// use bit 0 to report an event to the PC.
- globals.h provides key constants required by the debugger and may be extended with application constants.
- main.c : a simple program that flashes and LED and has places where an event may be reported and a breakpoint set.

avr_debug is a command line program running under Linux (full GUI version is being developed). It communicates with t26_debug via 2 wire TIA and provides the following facilities as shown from the help printout-

```
============ AVR Debugger : 2 wire Async TI Communications =============

  PURPOSE : debug AVR Tiny26 via the modified STK200 cable.

  RESET AVR : lp_tty_start ./avr_debug R
  READ RAM  : lp_tty_start ./avr_debug r start_address length
  WRITE RAM : lp_tty_start ./avr_debug w address value
  CALL      : lp_tty_start ./avr_debug c address
  BKP GO    : lp_tty_start ./avr_debug b

  NOTE - the call address is the byte address of a label from the
         *.map file, using 'make list', or avr-nm.
         The actual avr program counter is byte address/2.
       - if this program hangs the AVR is not responding, hit ^C.
       - all registers appear in the memory map BUT with an offset.
```

Illustration 6.4: Printout of avr-debug help information.

The following printouts show these features at work.

Data read and write : The printout below from avr_debug shows a data read (./avr_debug r a0), then write (./av_debug w a4 cd), then a read to confirm the write worked (./avr_debug r a0 11). The number of locations printed is adjusted so a full line (or lines) are displayed. Note the event byte is always zero indicating that no system level events have been reported, the breakpoint is also zero indicating no breakpoint.

```
[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  62  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug w a4 cd
  Write 0xA4 = 0xCD,  event_byte= 0x0,  breakpoint= 0x0

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0 11
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  CD  31  F6  BD  BE  81  98  FD  72  68   8  AD
  (0xB0)=  AD  99  F5  13   0  65  70  78  FD  4B  DB  30  64  FC  56  AF
```

Illustration 6.5: TIA Debugger Read - Write - Read Sequence

The Atmel AVR microprocessor can view all data and IO ports in the memory map so everything can be examined and changed using the red and write commands as above.

Event reporting : the application can use the variable event_byte as a way to report system level events as either a byte or a set of 8 bits. The illustration below shows how the event_byte is clear, then set to 2 by the calling of a function that has the statement “event_byte = 2;”. The function is called using the “c” command (./avr_debug c 4c). The new event is then display on the next read (event_byte= 0x02), and is cleared by that read, as shown by the next read.

```
[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  62  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug c 4c
  Call pc 0x26,  event_byte= 0x0,  breakpoint= 0x0

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x2,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  62  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  62  31  F6  BD  BE  81  98  FD  72  68   8  AD
```

Illustration 6.6: TIA Debugger Event Reporting

Breakpoints : breakpoints are inserted manually into the source code with a statement such as-

```
breakpoint(3) ;
```

In the command listing below a read first shows no breakpoint (breakpoint = 0x0). A call is made to a routine with a breakpoint statement as above (./avr_debug c 54). The next two reads show that the system is halted at "breakpoint = 0x3". The command to continue from breakpoint is given (./avr_debug b), and the next read shows the breakpoint is cleared and the application is running again.

```
[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  CD  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug c 54
  Call pc 0x2A,  event_byte= 0x0,  breakpoint= 0x0

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x3,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  CD  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x3,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  CD  31  F6  BD  BE  81  98  FD  72  68   8  AD

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug b
  Continued from breakpoint 0x3,  event_byte= 0x0

[root@acer debug_led_flash_1MHz_internal_RC]# lp_tty_start ./avr_debug r a0
 event_byte= 0x0,  breakpoint= 0x0,  Read starting at 0xA0
 Hex Addr  A0  A1  A2  A3  A4  A5  A6  A7  A8  A9  AA  AB  AC  AD  AE  AF
  (0xA0)=  E6  98  3B  10  CD  31  F6  BD  BE  81  98  FD  72  68   8  AD
```

Illustration 6.7: TIA Debugger Breakpoint Demonstration

Any monitor ROM debugger make demands on resources within the target microprocessor. The TIA approach and t26_debug make the following resource demands-

- *Code size* : t26_debug is written in C and consumes 151 instructions of the 1024 available. This was found from adding t26_debug to a PWM based switching regulator application of some 1705 bytes. This is much smaller than say the 2000 bytes taken by the 68HC11 Buffalo monitor. The code size may be reduced by carefully rewriting it in assembler.
- *Hardware use* : t26_debug consumes two general purpose IO pins. Unlike other monitor ROM debuggers it makes no use of timers, serial ports, interrupts and does not interfere with time critical application events.

The development environment described, including the TIA based debugger, has been used by three student project groups, and on two switching regulator projects by this author. It has also been ported to the Atmel ATMega 8535. The system has worked flawlessly and has enabled fast development of the applications. From the author's experience there are only a few times when an ICE would have been significantly more useful-

- *System lockup* : a monitor ROM debugger cannot easily tell how a program got into a lockup state. A slow but effective way is to add many breakpoints until the cause is found or to baseline the system (return to a previous known version that did not lockup).
- *Data corruption* : a monitor ROM debugger cannot breakpoint on a data write from an unexpected code location. A slow but effective solution is to comment out code fragments until the culprit is eliminated.
- *Symbolic views missed* : the current version of avr_debug does not allow symbolic views of variables and functions. The symbolic addresses are available using the avr-nm utility and could be built into avr_debug in the future.

6.4 Summary and Conclusions

This chapter started by identifying some seven different categories of debugging tools that are feasible for Low End Microprocessors. Two of these stood out, first the In Circuit Emulator (ICE) which is very powerful but also very expensive. The second approach is a monitor ROM based debugger which useful but has several significant problems. Section 6.2 showed that the use of 2-wire TIA plus some other innovations eliminated most of the problems associated with monitor ROM based debuggers.

Finally in section 6.3, the 2-wire TIA – monitor ROM debugger concept was implemented between an Atmel Tiny26 and an IBM-PC running Linux. This was shown to be very successful and allowed several projects to be successfully debugged. The debugger has been ported to the Atmel ATMega 8535 microprocessor and has also worked very well.

There are several future developments for the TIA based monitor ROM.

Other microprocessors : the debugger monitor can be rewritten to suit other microprocessors.

Optimization : the current debugger code on the Tiny26 is written in C with some optimization. Perhaps a careful rewrite in assembler would reduce the size of the code.

GUI symbolic interface : the command line debugger can be rewritten to allow symbolic views of code and data, not just a binary view. This work is currently in progress.

2-wire TIA based debuggers should find wide application where ever labor costs are not at a premium. This includes the university and education environment, small to medium industry, and hobbyists. A more polished version the debugger should be made public and may well deserve publication in the appropriate journals.

7 Discussion and Conclusions

Overview: *this chapter discusses key outcomes, and the theoretical and practical implications of the work in this thesis. While the research questions have been answered many other questions have been raised and will provide interesting follow-on research.*

7.1 Summary of Conclusions

This thesis is driven by a structured set of research questions introduced in chapter 1. These have now been answered-

- What are the properties of a communication protocol that would help reduce the cost of communications with low end microprocessors?
  Conclusion: Section 2.1 outlined a set of properties that clearly reduce the cost of communications.
- Do these properties describe a protocol category?
  If so is this an existing category or a new one?
  Conclusion: Section 2.2 shows that the properties do indeed define a new category of communications that can stand along side the traditional synchronous, plesiochronous, and asynchronous categories. The new category has been called Time Independent Asynchronous (TIA) communications.
- What existing protocols exist within a suitable category?
  Conclusion: Section 2.2 identified a number of existing protocols that are TIA in nature.
- Given the insights from preceding analysis is it possible to create an improved communication protocol that better serves the needs of low end microprocessors?
  Conclusion: Chapter 4 and 5 details the development of novel new 2-wire TIA protocols.
- If such an improved protocol is developed how should it be modeled, simulated, and tested?
  Conclusion: Chapter 3 examined the literature to uncover what properties of a protocol should be verified. It also discovered problems with a modelling tool of choice called Signal Transition Graphs and proposed the STG-FT solution. Chapter 4 detailed a new

STG-FT simulator that verified the 2-wire TIA protocol and went on to implement and test the protocol and showed the simulation results match the real performance.

Practical application? While 2-wire TIA clearly works just where will it find practical application? Key relevant attributes uncovered in this thesis include-

Cost saving: The cost saving from TIA is real but relatively small, in the order of US$0.25 to US$1 per microprocessor as detailed in chapter 2. This makes 2-wire TIA attractive for mass manufactured items where the multiplying effect of 100,000 items translates to a saving of at least US$25,000. The markup from manufacture to retail cost is an additional multiplying factor.

Speed: the peak data speeds outlined in section 4.4.3 are 50 kilobits/sec in both directions. This is adequate for low quality voice or low speed data transfer and commands. Higher speeds may be possible with the 2B-2B version of 2-wire TIA communications.

No peripherals: 2-wire TIA requires no special peripherals. This can be very useful in niche applications such as the microprocessor debugger developed in chapter 6. This debugger has been embraced by a number of real projects and found to be very useful. The author is currently under pressure to develop it further.
Any application that lacks enough communications peripherals may find 2-wire TIA an attractive solution.

The end of section 2.1 presented a concept map of example applications of 2-wire TIA. Applications such as ultra-low cost data loggers and low cost interfaces for RF receiver modules are worthy of serious investigation.

STG limitations: Section 3.2 discussed how the Petri net based Signal Transition Graphs appeared to be a modelling tool of choice for TIA based systems. It was surprising to uncover fundamental problems with the STG model that meant software implemented systems could not be modelled reliably. In particular an STG model may miss livelock and deadlock properties of a protocol.

Section 3.2.3 proposes a solution in the form of an extension to the STG model called STG For Threads (STG-FT) (Radcliffe and Yu 2006b). This appears to solve the inherent problems in the STG model. Any system implemented in software may fall foul of the same STG modelling problems and will need the STG-FT extensions to properly model behavior. The STG-FT simulator described in section 4.3 has a very extensible structure and may warrant further work to make it an open source simulator specifically for STG-FT modelling.

7.2 Future Research Directions

This thesis has answered all the research questions posed in the chapter 1 but the work has raised yet more questions. The pursuit of these new questions will be interesting and may well develop useful solutions of practical and theoretical importance.

One wire TIA : This may seem impossible but then again so did 2-wire TIA. What allowed 2-wire TIA to work was the use of novel digital driver arrangements. Perhaps there are other media or driver innovations that will allow one wire TIA.

2B-2B Implementation : This 2-wire TIA variant has been proposed and simulated but not yet reported in the literature nor has its data throughput or livelock been measured.

Speed comparisons : It would be interesting to compare the speeds of the two 2-wire TIA variants developed in this thesis, the 3 wire TIA variants discovered in the literature survey, and any 1-wire system developed.

Speed improvement : 2-wire TIA as implemented in this thesis is not very fast. Can it be made faster? Are there TIA variants which perform better than those described in this thesis?

Bus systems : With the exception of the IEEE-488 (GPIB) bus all TIA systems are point to point protocols not a bus system. Is a 2-wire TIA bus system possible?

More implementations : Section 2.1 presented a concept map that proposed a variety of practical uses for 2-wire TIA. Interesting lessons may be learnt from actually implementing these applications and comparing the result to existing solutions.

Microprocessor debugger application : This has real potential as a tool for small industries and for educational use. The implementation in this thesis is very useful but really is just a proof of concept. It should be possible to move the concept much closer to an ICE in terms

of symbolic rather than hexadecimal displays and a much better GUI.
A 1-wire TIA system would be of even more use than a 2-wire TIA system.

Simulation speed up: The simulation of 2-wire TIA by brute force full tree evaluation required a huge number of system state nodes to be visited. The state tick off strategy described in section 4.4 reduced this by a factor of 250 million. Further investigation of this method and its application to other problems may prove fruitful.

Higher level protocols: the work in this thesis concentrates on physical layer issues on communications. It would be interesting to examine the higher layer issues of a communications system based on 2-wire TIA.

And on a personal note ... the work described in this thesis literally started with some carefully posed research questions related to a problem faced by this author. It has led to a fascinating tour of some highways and side streets of communications theory and technology. This has been an enjoyable activity in itself, even more satisfying it appears to have resulted in outcomes of theoretical and practical value, as evidenced by the two accepted IEEE papers (Radcliffe and Yu 2006a) and (Radcliffe and Yu 2006b), one more submitted (Radcliffe and Yu 2006c), and some practical applications. With luck the “future directions” outline above will provide similar levels of interest and satisfaction.

8 References

Bainbridge, W.J.; Furber, S.B.; 1998, Asynchronous macrocell interconnect using MARBLE, *Proceedings of the Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 30 March-2 April 1998, pg 122 – 132.

(Describes the MARBLE asynchronous bus for cross VLSI communications. Notable for a 2 wire control of a bidirectional bus which is TIA in nature, see figure 6.)

Behcet, B. ; Bochmann, G.V.; 1984, Synchronization and Specification Issues in Protocol Testing, *IEEE Transactions in Communications*, Vol. COM-32, No.4 April 1984, pp 389-395.

(Good review of methods to test protocols, many practical hints.)

Berkel, K.; Bink, A.; 1996, Single-track handshake signaling with application to micropipelines and handshake circuits, *Proceedings of the Second International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 18-21 March 1996 Page(s):122 – 133.

(Single wire control system but not TIA in nature.)

Beizer, B.; *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold, 1990.

Bochman, G.V.; 1978, Finite State description of Communication Protocols, *Computer Networks* Vol 2, 1978, pp 361-372, North Holland Publishing Company.

(Good description of how to apply FSM to protocol analysis and its limitations.)

Bosch, 1991, *CAN bus 2.0 Specification*, retrieved from http://www.semiconductors.bosch.de/en/20/can/index.asp

Bosch, 2006, Bosch CAN bus home page, retrieved from http://www.semiconductors.bosch.de/en/20/can/index.asp

Chen, W.T.; Ho, J.P.; Wen, C.H.; 1978, Dynamic validation of programs using assertion checking facilities, *IEEE Conference on Computer Software and Applications* , COMPSAC '78. Nov. 13-16 1978, Pages:533 – 538.

Chow, T.S.; 1978, Testing design modeled by finite-state machines, *IEEE Transactions on Software Engineering*, Vol. 4, pp. 178-186, March 1978.

(Describes the W-Method for testing FSMs.)

Christ, O.; Fleisch, E.; Mattern, F.; 2002, *M-Lab : The Mobile and Ubiquitous Computing Lab Phase II*, ETH Zurich & University of St. Gallen, 2002. Retrieved from www.m-lab.ch/about/MLabIIProjectPlan_e.pdf

Chu, P.-Y.M.; Liu, M.T.; 1989, Global state graph reduction techniques for protocol validation in the EFSM model, Conference Proceedings of the Eighth Annual International Phoenix Conference on Computers and Communications, 22-24 March 1989, pg 371 – 377.

(Good discussion of ways to limit state explosion.)

Corbett, J.C.; 1996, Evaluating deadlock detection methods for concurrent software, IEEE Transactions on Software Engineering, Volume 22, Issue 3, March 1996 pg. 161 – 180.

(Good survey of state explosion reduction techniques, and trial of three several.)

Cortadella, J.; Kishinevsky, M.; Lavagno, L.; Yakovlev, A.; 1995, Synthesizing Petri nets from state-based models, *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, Dec. 1995, pp. 164-171.

Cortadella, J.; Kishinevsky, M.; Lavagno, L.; Yakovlev, A. ; Deriving Petri Nets from Finite Transition Systems, IEEE Transactions on Computers, Vol. 47, Number 8, pages 859-882, Aug. 1998.

Cortadella, J.; Kishinevsky, M.; Kondratyev, A.; Lavagno, L.; Yakovlev, A.; 2003, *"Petrify: Method and Tool for Synthesis of Asynchronous Controllers and Interfaces*".
A tutorial from ASYNC2003 retrieved from
http://www.staff.ncl.ac.uk/alex.yakovlev/home.formal/async03-tut-demo.ppt

Cortadella, J.; 2006, *Petrify : a tutorial for the designer of asynchronous circuits*, (no date or revision number given). Retrieved from
http://www.cs.technion.ac.il/~cs234305/petrify/docs/tutorial.ps

D'Argenio, P.R.; Niebert, P.; 1996, Partial order reduction on concurrent probabilistic programs Proceedings of the First International Conference on the Quantitative Evaluation of Systems, QEST, 27-30 Sept. 2004, pg 240 – 249.

(Good discussion of partial order problems.)

Davis, L.; 2005, *VME Bus*, retrieved from
http://www.interfacebus.com/Design_Connector_VME.html

Dingle, A.; Hildebrandt, T.H.; 1998, Improving C++ Performance Using Temporaries, *IEEE Computer*, March 1998, pg. 31-41.

(Show how dynamic memory use slows down C++.)

DEC Unibus Specification, 1979, retrieved from-
http://www.bitsavers.org/pdf/dec/unibus/UnibusSpec1979.pdf

Durda IV, F.; 2004, *Centronics and IBM Compatible Parallel Printer Interface Pin Assignment Reference,* 2004. Retrieved from-
http://nemesis.lonestar.org/reference/computers/interfaces/centronics.html

Dutta, S.K.; Saha, D.; Das, P.K.; 1997, Design of a parallel protocol verification system, *Proceedings of the IEEE Conference on Speech and Image Technologies for Computing and Telecommunications* (TENCON '97), Volume 2, 2-4 Dec. 1997 Page(s):425 – 428.

(General discussion of reachability trees and FSM.)

Fadul, F.; 1993, Advanced simulation topics on 8-bit microprocessors, Proceedings of the Twenty-Third Annual Conference of Frontiers in Education, 6-9 Nov. 1993 pg. 179 – 181.

(Educational use of microprocessor simulator.)

Faggin, F.; 2001, *The Intel 4004*, Retrieved from www.intel4004.com

Freeman, R.L.; *Practical Data communications*, 2nd edition, 2001, Wiley-Interscience Publication.

Freescale, 1994, *AN-991 Serial Peripheral Interface*, retrieved from http://www.freescale.com/files/microcontrollers/doc/app_note/AN991.pdf

(SPI description from Motorola, now Freescale.)

Fujiwara, S.; Bochmann, G.V.; Khendek, F.; Amalou, M.; Ghedamsi, A.; 1991, Test Selection Based on Finite State Models, *IEEE Transaction on Software Engineering*, Vol 17, No. 6, June 1991, pp. 591-603.

(Good review of W and Wp methods for testing protocols.)

Furber, S.; Garside, J.D.; Riocreux, P.; Temple S.; Day, P., Liu J.;Paver, N.; (1999), AMULET2e: An Asynchronous Embedded Controller, *Proceeding of the IEEE,* Vol. 87, No.2 February 1999.

(Contains a reference to a 3 wire TIA protocol.)

Gunther, W.; Hett, A.; Becker, B.; 2001, Application of linearly transformed BDDs in sequential verification, Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC 2001, 30 Jan.-2 Feb. 2001, pg. 91 – 96.

(Good discussion of BDDs and possible efficiencies when analyzing reachability tree.)

Harel, D.; 1987, Statecharts : a visual formalism for complex systems, *Scientific Computer Programming*, vol. 8, pp 231-274.

Hauck, S.; 1995, Asynchronous Design Methodologies: An Overview, *Proceedings of the IEEE* 83(1):69-93, January 1995.

Hector, J.; 2002, How to choose an in-circuit emulator, *embedded.com*, retrieved from http://www.embedded.com/showArticle.jhtml?articleID=23901694

Hills, H.; Beach, M.; 1999, *Microprocessor Debuggers*, retrieved from http://www.hitex.com/iuk/general/whyICE2.pdf

(White paper from Hitex corporation on microprocessor debuggers.)

IEEE Std 1014-1987, *IEEE Standard for A Versatile Backplane Bus: VMEbus -Description.*

IEEE Std 1284-1994, *IEEE standard signaling method for a bidirectional parallel peripheral interface for personal computers*.

IEEE Std 488.1-1987, *IEEE standard digital interface for programmable instrumentation*.

IEEE-610 1990, *IEEE Standard Computer Dictionary*, IEEE, New York.

Intel Corporation, 1988, *80C31BH/80C51BH/87C51 MCS(R) 51 CHMOS Single-Chip 8-Bit Microcontroller*
Datasheet retrieved from http://www.intel.com/design/mcs51/datashts/270419.htm

Johnston, H.; Graham, M.; 1993, *High Speed Digital Design*, Prentice-Hall, Englewood Cliffs NJ.

(An extremely good book on the black art of high speed digital design.)

Jung, E.-G.; Choi, B.-S.; Won, Y.-G.; Lee, D.-I.; 2003, Handshake protocol using return-to-zero data encoding for high performance asynchronous bus, *IEE Proceedings on*

Computers and Digital Techniques, Volume 150, Issue 4, 18 July 2003 Page(s):245 – 251

Kessels, J.; 2005, Register-communication between mutually asynchronous domains, *11th IEEE International Symposium on Asynchronous Circuits and Systems* (ASYNC 2005), 14-16 March 2005 Page(s):66 – 75.

Kishinevsky, M.; Cortadella, J.; Kondratyev, A.; 1998, Asynchronous interface specification, analysis and synthesis, *Design Automation Conference,* 15-19 Jun 1998 Page(s):2 – 7.

(Uses STGs to model VME.)

Khomenko, V.; Koutny, M.; Yakovlev, A.; 2002, Detecting state coding conflicts in STGs using integer programming, *Proceedings of the Conference on Design, Automation and Test in Europe*, 4-8 March 2002, pg 338 – 345.

(Discusses ways to reduce state explosion in Petri nets.)

Le Lann, G.; 2003, Asynchrony and real-time dependable computing, *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems*, 2003. (WORDS 2003), 15-17 Jan. 2003. pp18 – 25.

Lee, G.; Lee, M.; Hui H.; Shao, H.; Zhao, X.; 2004, Networked intelligent controller based on embedded system, *30th Annual Conference of IEEE Industrial Electronics Society*, 2004. IECON 2004. Volume 3, 2-6 Nov. 2004, pp. 2942 – 2945.

LinBus, 2006, LIN bus home page, retrieved from http://www.lin-subbus.org/

Lloyd, D.W.; Garside, J.D.; 2001, A practical comparison of asynchronous design styles, Seventh *International Symposium on Asynchronous Circuits and Systems*, ASYNC 2001, 11-14 March 2001 Page(s):36 – 45.

Memon, A.; 2002, GUI Testing ; Pitfalls and Processes, *IEEE Computer*, August 2002, pg 87-88.

Merlin, P.; 1979, Specification and Validation of Protocols, *IEEE Transactions on Communications*, Volume 27, Issue 11, Nov 1979 Page(s):1671 – 1680.

(Old but interesting paper on protocol validation with references to yet older works.)

Microchip Corporation, Microprocessor comparison chart. Retrieved on 12/7/2006 from- http://www.microchip.com/ParamChartSearch/chart.aspx?branchID=1031&mid=10&lang=en&pageId=74

MIL-STD-1553, 1989, *Specification of MIL-Standard 1553 Bus Protocol and Application to EA-6B Communications Countermeasures.* , retrieved from http://stinet.dtic.mil/

Minea, M.; Izbasa, C.; Jebelean, C.; 2002 Experience with Formal Verification of SDL Protocols, retrieved from http://www.cs.utt.ro/%7Emarius/papers/cipc03.pdf

(Excellent list of SDL verification papers and case studies.)

Moelands, A.; Schutte H.; 1982, *Two-wire bus-system comprising a clock wire and a data wire for interconnecting a number of stations*, US patent 4,689,740.

Molina, P.A.; Cheung, P.Y.K.; Bormann, D.S.; 1996, Quasi delay-insensitive bus for fully asynchronous systems, *1996 IEEE International Symposium on Circuits and Systems*, ISCAS '96 'Connecting the World'., Volume 4, 12-15 May 1996 Page(s):189 – 192.

Nasu, M.; Katori, S.; Maehashi, Y.; Yoshizawa, K.; *Serial bus interface system for data communication using two-wire line as clock bus and data bus*, US patent 4,847,867, July 11 1989.

Nigussie, E.; Plosila, J.; Isoaho,J.; 2006, Delay-insensitive on-chip communication link using low-swing simultaneous bidirectional signaling, *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, Volume 00, 2-3 March 2006 Page(s):6 pp.

Nystrom, M.; Martin, A.; 2002, *Crossing the Synchronous-Asynchronous Divide*, Retrieved from http://www.ece.rochester.edu/~albonesi/wced02/papers/nystrom.pdf

Old Computers, 2006, Retrieved from www.old-computers.com

(Excellent web site describing a huge range of old computers including ICE systems.)

Pastor, E.; Cortadella, J.; Roig, O.; 2001, Symbolic analysis of bounded Petri nets, *IEEE Transactions on Computers*, Volume 50, Issue 5, May 2001 pg 432 – 448.

(Shows how symbolic Binary Decision Diagrams can analyze reachability. Only useful for highly concurrent Petri nets.)

Peeters, A.; van Berkel, K.; 1995, Single-rail handshake circuits, *Proceedings of the Second Working Conference on Asynchronous Design Methodologies*, 30-31 May 1995 Page(s):53 – 62.

Peterson, J.L.; 1981, *Petri Net Theory and Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ.

(Key reference to Petri Net theory and analysis.)

Philips Semiconductors, 1996, *I2S Bus Specification*, retrieved from www.semiconductors.philips.com/acrobat_download/various/I2SBUS.pdf

Philips Corporation*, The I2C-Bus Specification Version 2.1,* 2000,. Retrieved from http://www.semiconductors.philips.com/acrobat/literature/9398/39340011.pdf

Radcliffe, P.; 1985, *System Debugging of the MFC subsystem in the Ericsson ASDP-162*, internal Ericsson report.

Radcliffe, P.; Yu, X.; 2006a, Microprocessor Communications for Cost Sensitive Products, *IEEE International Conference on Industrial Informatics*, Singapore, 16-18 August 2006.

(Describes work done in this thesis - the TIA category, waveforms for the new 2-wire TIA protocol, and reports on testing of the protocol.)

Radcliffe, P.; Yu, X.; 2006b, A Novel Time Independent Asynchronous Communication Protocol & Application. *Proceedings of the 32nd Annual Conference of the IEEE Industrial Electronics Society*, Paris, November 7 - 10 2006.

(Describes work done in this thesis – TIA justification, STG problems leading to STG-FT extensions, further test results of 2-wire TIA.)

Radcliffe, P.; Yu, X.; 2006c, A New Time Independent Asynchronous Protocol & its Applications, *IEEE Transactions on Industrial Infomatics*.

(Submitted and undergoing revision as requested by the reviewers.
Encapsulates the two conference papers and adds further results.)

Reisig, W. ; 1991, *A Primer in Petri Net Design*, Springer-Verlag.

(Another variant of Petri nets, most authors appear to favor the Peterson format.)

Sidhu, D.P.; Leung, T.-K.; 1989, Formal methods for protocol testing: a detailed study, *IEEE Transactions on Software Engineering*, Volume 15, Issue 4, April 1989 Page(s):413 – 426.

Sietz, C.L.; 1980, System Timing. *Introduction to VLSI Systems*, chapter 7. Addison Wesley, 1980.

Simpson, H.R.; 1992, Correctness analysis for class of asynchronous communication mechanisms, *IEE Proceedings on Computers and Digital Techniques*, Volume 139, Issue 1, Jan 1992 Page(s):35 – 49.

SMBus, 1995, *System Management Bus Specification*, retrieved from http://www.smbus.org/specs/

(An I2C like bus from Intel developed for battery management.)

Smith, M.R.; Cheng, M.; 1996, Use of "virtual" (simulated) hardware devices in microprocessor laboratories and tutorials, Proceedings of 26th Annual Conference on Frontiers in Education Conference, FIE '96, Volume 3, 6-9 Nov. 1996 pg. 1181 – 1185.

(Use of microprocessor simulator for education.)

International Telecommunication Union, *Specification and Description Language* (SDL), ITU-T Standard Z.100. , 1992

Takahashi, T.; Hanyu, T.; 2004, Multiple-valued multiple-rail encoding scheme for low-power asynchronous communication, *Proceedings of the 34th International Symposium on Multiple-Valued Logic*, 19-22 May 2004 Page(s):20 – 25.

Teifel, J.; Manohar, R.; 2003, A high-speed clockless serial link transceiver, *Proceedings of the Ninth International Symposium on Asynchronous Circuits and Systems*, 12-15 May 2003 Page(s):151 – 161

Telles, M.; Hsieh, Y.; 2001, *The Science of Debugging*, Coriolis, Arizona.

(Debugging mostly at source level but some good system level testing.)

Toeppe, S.; Ranville, S.; and Butts K.; 1998, Specification and Testing of Automotive Powertrain Control System Software using CACSD Tools, *Digital Avionics Systems Conference Proceedings*, 17th DASC, the AIAA/IEEE/SAE Vol 2, 31 Oct.-7 Nov. 1998, pg I13 1-9.

TIA/EIA-232-F Standard 1997, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*, Electronics Industries Association Engineering Department.

Turley, J., 2002, Motoring with Microprocessors, *Embedded Systems Programming*, Dec 2002; retrieved from www.embedded.com/showArticle.jhtml?articleID=13000166.

(Discusses the large number of microprocessors within cars.)

Vallejo, F.; Harbour, M.G.; Gregorio, J.A.; 1992, A laboratory for microprocessor teaching at different levels, *IEEE Transactions on Education*, Volume 35, Issue 3, Aug. 1992 pg. 199 - 203.

VMEbus Standards and Specifications, retrieved from- http://www.interfacebus.com/Design_Connector_VME.html

Winters, M.; 1994, Using IEEE-1149.1 nfor In Circuit Emulation, IEEE Conference on Ideas/Microelectronics, Anaheim CA, 27-29 September 1994, pg 525-528.

(Describes using JTAG boundary scan as an in circuit emulator.)

Yakovlev, A.V.; 1992, On limitations and extensions of STG model for designing asynchronous control circuits, *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, ICCD '92 11-14 Oct. 1992 pg 396 – 400.

(Mainly concerned with the STG variant of Petri nets.)

Yakovlev, A.; Furber, S.; Krenz, R.; Bystrov, A.; 2004, Design and analysis of a self-timed duplex communication system, *IEEE Transactions on Computers*, Volume 53, Issue 7, July 2004, pp 798 – 814.

Yuang, M.C.; 1988, Survey of protocol verification techniques based on finite state machine models, *Proceedings of the IEEE Computer Networking Symposium*,, 11-13 April 1988 Page(s):164 – 172.

(Good list of problems a protocol may exhibit, good study of FSM based protocol analysis methods.)

9 Appendix A

This thesis is accompanied by a CDROM which contains much of the software code created by the author.

The following Tiny26 code can be inspected but will not run without a Tiny26 connected to the parallel port as shown in section 4.5. The IBM-PC partner code can be found in the directory pc_demo.

- Directory avr_bit_echo : the IBM-PC sends bits to the Tiny26 using 2-wire TIA and checks the echoed return bits.
- Directory avr_bit_echo_corrupt : as per avr_bit_echo but an interrupt can be used to corrupt the slave state. This used to try and cause deadlock by examining the slave part of the any state/input universe.
- Directory avr_byte_echo : the IBM-PC sends bytes to the Tiny26 using 2-wire TIA and checks the echoed and modified byte.

The directories debug_proformas ... contain minimal code for the Tiny26 and ATMega 8535 that implement the debugger described in chapter 6. The partner IBM-PC code can be found in the directory pc_demo.

The directory sim contains 3 versions of the simulator which can be run on most Linux distributions and does not need any Tiny26 connected to the parallel port. Each simulator subdirectory contains the full KDevelop project where the executables can be found in the subdirectory called debug. The simulator in subdirectory dff implements the STG-FT example used in section 3.2.3. The directory sim_4p2io simulates the original TIA protocol. The directory sim_ioio_fs simulates the 2B-2B TIA variant.