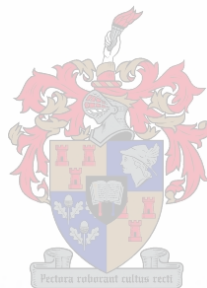


Design of a GPS Based Time Stamping and Scheduling System for Power System Applications

M.T.S. van As



Thesis presented in partial fulfilment of the requirements for the degree of

Master of Science in Engineering

at the

University of Stellenbosch

Supervisor: Dr. H.J. Vermeulen

December 2003

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work, and that I have not previously in its entirety or in part submitted it at any other university for a degree.

M.T.S. van As

Date

Abstract

This thesis describes the development of a GPS Based Time Stamping and Scheduling System for power system applications. These applications include Wide Area Measurements (WAMs) of electrical power system quantities and high-voltage transmission line fault location.

The developed system employs a microcontroller and a GPS receiver to synchronise an on-board microsecond-accurate clock to a global time standard. The system is therefore able to provide an accurate GPS-synchronised time stamp of a received trigger signal for use in high-voltage transmission line fault location. The system is also able to generate a trigger signal at a pre-programmed time for initiation of data acquisition runs on electrical power systems.

The system was constructed and tested in a laboratory environment. Although the system is designed to operate in stand-alone mode, a host computer software program was also developed for system control and data downloading. The software program was used to time stamp a number of trigger signals and data was downloaded to a host computer. Trigger signals were also generated at predefined times. The acquired data was validated and presented.

In conclusion, the low system cost, relative to existing commercial systems, accuracy and programmability of the developed system makes it suitable for a wide variety of time-critical data acquisition applications.

Opsomming

Hierdie tesis beskryf die ontwikkeling van 'n GPS gebaseerde tyd stempel en skedulerings sisteem met die oog op kragstelsel toepassings. Ingesluit by hierdie toepassings is wye area metings op elektriese kragstelsels, asook foutopsporing op hoogspanning transmissielyste.

Die ontwikkelde sisteem gebruik 'n mikrobeheerder en 'n GPS ontvanger om 'n aanboord mikrosekonde-akkurate horlosie te sinkroniseer met 'n internasionale tyd standaard. Dus kan die ontwikkelde sisteem 'n akkurate GPS gesinkroniseerde tyd stempel aan 'n snellersein heg. Hierdie tyd stempel kan gebruik word in hoogspanning transmissielyste foutopsporing. Die sisteem kan ook 'n snellersein genereer op 'n vooraf geprogrammeerde tyd. Hierdie snellersein kan gebruik word om belangrike data van elektriese kragstelsels te versamel, deur gebruik te maak van bestaande dataversamelingstelsels.

Die sisteem was ontwerp en getoets in laboratorium toestande. Alhoewel die stelsel ontwerp is om alleenstaande te funksioneer, is 'n beheer rekenaar gebruik om, met die hulp van ontwikkelde sagteware, die sisteem te beheer en data af te laai. 'n Tyd stempel is aan 'n aantal snellerseine geheg en hierdie data is afgelaai na 'n beheer rekenaar. Die sisteem is ook geprogrammeer om 'n aantal snellerseine te genereer op vooraf gedefinieerde tye. Die data wat uit hierdie toetse versamel is, is bespreek.

In vergelyking met bestaande kommersiële stelsels is die ontwikkelde stelsel se lae koste, akkuraatheid en programmeerbaarheid eienskappe wat die stelsel geskik maak vir 'n wye verskeidenheid van tyd-kritieke dataversameling toepassings.

Acknowledgements

I would like to thank the following people from the bottom of my heart:

My Lord and Saviour, Jesus Christ, for His Life, unsurpassing Love and for giving me reason to live.

Dr. Johan Vermeulen, for his vision, patience, guidance, encouragement, caring, friendship and time.

Francois for his friendship, help and encouragement through tough times, his brilliant mind and sound effects.

Andrew, Johan Strauss, Mev. Susan Locke, Charlene, William, Alfie and John for help and friendship.

My ouers, vir ondersteuning en baie liefde. Ek waardeer julle meer as wat woorde kan sê, sonder julle insae in my lewe sou ek nooit kon vrugte dra nie.

Jane, for standing by me, being my friend, hours of prayer and tons of encouragement. You are amazing.

The van Nieuwholtz family, for hospitality and friendship.

My vriende: Robert, Bernard, Pierre Tredoux, Jan-Willem, Pierre Theron, Johan en Danel Steenkamp, Wynand, Phillip, Chris, Barry en JF vir julle gebede en vriendskap.

Table of Contents

1	Introduction	1
1.1	Project Motivation	1
1.2	Project Description	2
1.3	Thesis Overview	3
2	Literature study.....	5
2.1	Introduction	5
2.2	Synchronised Sampling	5
2.3	Applications of Synchronised Sampling in Power Systems.....	6
2.3.1	Transmission Line Fault Location.....	6
2.3.2	Wide Area Measurements	7
2.3.3	Protective Relay Testing.....	8
2.4	Fault Location Algorithms	8
2.5	Wide Area Measurements	12
2.5.1	Phasor Measurement Units.....	12
2.5.2	Phasor Measurement Process	13
2.6	Sources of Timekeeping.....	14
2.6.1	Time Transfer Techniques.....	14
2.6.2	Oscillator Technology	16
2.6.2.1	Crystal Oscillator.....	16
2.6.2.2	Oven Controlled Crystal Oscillator	17

2.6.2.3	Atomic Oscillator Technology	17
2.7	GPS Based Time Stamping and Scheduling System Overview	19
3	Introduction to GPS	22
3.1	GPS Segments	22
3.2	GPS Operation	23
3.3	GPS Receiver Operation (“User Segment”)	24
3.4	Motorola Oncore™ GT+ GPS Receiver	27
3.4.1	Functional Block Diagram	28
3.4.2	Terminal Connections	29
3.4.3	One-Pulse-Per-Second Signal	30
3.5	Serial Interface Protocol	31
3.5.1	Interface Protocol	31
3.5.1.1	Motorola Binary Format	31
3.5.1.2	NMEA 0183 Format	33
3.5.2	Receiver Antenna Placement	33
3.6	GPS Accuracy	34
3.6.1	Timing Accuracy	35
3.6.2	RF Jamming Immunity	36
3.7	Conclusion	36
4	Real Time Clock and Comparator	37
4.1	Introduction	37

4.2	Data Flow Operations.....	38
4.2.1	Address Decoder	39
4.2.2	Input Latches	39
4.2.3	Output Latches.....	40
4.3	Real Time Clock.....	40
4.3.1	Description	41
4.3.2	Simulation Waveforms	45
4.3.2.1	Normal RTC Operation	45
4.3.2.2	Set RTC Time.....	45
4.3.2.3	Read RTC Time.....	46
4.4	Real Time Clock Comparator.....	47
4.4.1	Block Diagram.....	47
4.4.2	Implementation.....	47
4.4.3	Simulation Waveforms	49
4.5	Conclusion.....	49
5	Microsecond Counter and Comparator	51
5.1	Introduction	51
5.2	Block Diagram.....	52
5.3	Counter System Block.....	53
5.3.1	System Clock.....	55
5.3.2	1PPS Signal Conditioning	55

5.3.3	Programmable Clock Prescaler	56
5.3.4	24-bit Counter.....	60
5.3.5	Trigger Signal Conditioning.....	61
5.3.5.1	Signal <i>SyncTrig</i>	62
5.3.5.2	Signal <i>IntRequest</i>	62
5.3.6	Counter Output Latch	64
5.3.7	Data Bus Output Latches.....	65
5.4	Microsecond Counter Comparator	66
5.5	Address Decoder	68
5.5.1	Indicator LEDs	68
5.5.2	FIFO Memory Control Signals.....	69
5.6	FIFO Memory Operation.....	69
5.6.1	FIFO Detail.....	70
5.6.2	Reset Operation	71
5.6.3	Write Operation	72
5.6.4	Read Operation.....	72
5.7	Conclusion.....	73
6	System Microcontroller	74
6.1	Introduction	74
6.2	System Control Program High-Level Discussion	76
6.3	Data Write and Read Routines	77

6.4	Initialisation Procedures	78
6.4.1	Microcontroller I/O Port Setup.....	78
6.4.2	Variable Initialisation	79
6.4.3	UART Setup	79
6.5	Communication Routines	80
6.5.1	UART Control Registers	80
6.5.2	Timer/Counter1	81
6.5.3	Software Data Buffer.....	82
6.5.4	Communication Error Checking.....	83
6.5.4.1	GPS Receiver Communication Error Checking.....	83
6.5.4.2	Host Computer Communication Error Checking.....	85
6.5.5	GPS Communication Operation.....	87
6.5.5.1	Procedure <i>SendCommand</i>	88
6.5.5.2	Procedure <i>GPS_RxInt</i>	89
6.5.5.3	Function <i>ByteInGPSRxBuf</i>	90
6.5.5.4	Procedure <i>WaitForGPSReply</i>	90
6.5.5.5	Procedure <i>ProcessMessage</i>	91
6.5.5.6	Procedure <i>StoreTime</i>	92
6.5.5.7	Procedure <i>StoreDate</i>	93
6.5.6	Host Computer Communication Operation.....	93
6.5.6.1	Procedure <i>PC_RxInt</i>	95

6.5.6.2	Procedure <i>PC_UDREEmpty</i>	95
6.5.6.3	Procedure <i>TxByteToPC</i>	96
6.5.6.4	Procedure <i>StartTransmit</i>	96
6.5.7	Software UART Routines.....	97
6.6	GPS Receiver Setup	98
6.6.1	Procedure <i>StartGPS</i>	99
6.6.2	Procedure <i>GPSReceiverSetup</i>	100
6.6.3	Procedure <i>GetTime</i>	100
6.6.4	Procedure <i>GetDate</i>	101
6.7	Interrupt Service Routines	102
6.7.1	1PPS Interrupt	102
6.7.2	PC Command Interrupt.....	103
6.7.2.1	Command <i>SendTriggerTime</i>	104
6.7.2.2	Command <i>SetCompare</i>	105
6.7.2.3	Command <i>ReadCompare</i>	106
6.7.2.4	Command <i>GPS_RTC_Status</i>	107
6.7.2.5	Command <i>SetRTC</i>	108
6.7.2.6	Command <i>ClearFIFO</i>	109
6.7.2.7	Command <i>SendGenTrigData</i>	110
6.7.2.8	Command <i>ClearGenTrigData</i>	110
6.7.3	Trigger Received Interrupt	111

6.7.3.1	Procedure <i>StoreTrigData</i>	111
6.7.3.2	Procedure <i>TrigEvent</i>	111
6.8	Conclusions	112
7	LabVIEW™ Control Software.....	113
7.1	Introduction	113
7.2	LabVIEW™ Programming Environment.....	113
7.3	Example: The CRC-16 VI.....	115
7.4	Host Computer Control Program High-Level Discussion.....	116
7.5	Main Control Panel.....	118
7.6	Configuration Panel.....	119
7.7	Conclusion.....	121
8	Results and Recommendations.....	122
8.1	Introduction	122
8.2	Test Setup and Procedures.....	122
8.3	Logic Analyser Signal Captures.....	122
8.3.1	RTC Set operation	123
8.3.2	RTC and Microsecond Counter Read operation	123
8.3.3	Trigger Interrupt Operation	125
8.3.4	FIFO Write and Read Operation	125
8.4	LabVIEW™ PC Control Program.....	127
8.5	Oscilloscope Measurements	127

8.6	Conclusions and Recommendations.....	129
8.6.1	Results and Measurements	129
8.6.2	Final System Overview	129
8.6.3	Recommendations	130
8.6.3.1	Local System Clock.....	130
8.6.3.2	System Memory.....	131
8.6.3.3	System Communications Interface.....	131
9	References	133
A	WinOncore™ GPS Control Software.....	138
B	Oncore™ GT+ Control Commands	140
C	Address Decoder: Code and simulation	142
C.1	Data Register Addresses.....	142
C.2	VHDL Code for RTC Address Decoder	143
C.3	VHDL Code for Microsecond Counter Address Decoder	144
C.4	Altera® MAX+PLUS® Simulation Waveforms	145
D	RTC and Comparator: Implementation and simulation	147
E	Microsecond Counter and Comparator: Implementation and simulation	156
F	Software UART Theory	164
F.1	Introduction	164
F.2	Theory of operation	164
F.3	Implementation.....	166

F.4	UART Initialisation.....	167
F.5	Byte Transmission.....	167
F.6	UART Status Byte.....	168
F.7	Interrupt Service Routines.....	169
F.8	Conclusion.....	171
G	Microcontroller Code Listings	173
G.1	Software UART Program Code.....	173
G.2	System Test Program Code	179
G.3	Main Control Program Code	181
H	LabVIEW™ Program Diagrams	200
I	Schematic Diagrams.....	226

Chapter 1

Introduction

1.1 Project Motivation

Transmission systems represent a vital component of the electrical utility infrastructure of a country. This infrastructure is aimed at supplying power to a variety of users. Power systems consist of a number of different components, which include generators, power transformers, transmission lines, and loads. Design of the system components and overall systems is implemented under a stringent reliability requirement with a strong emphasis on continuity of the power supply [2,3].

As in any other technical system, there are circumstances under which failures in the system operation occurs [2,12,19,23]. When faults are not identified quickly and corrected, the power system's security, stability and reliability may be compromised, thereby influencing the utility's ability to deliver high quality un-interrupted electrical energy to its customers. In the past, power system engineers have depended heavily on maintenance crews to locate transmission line faults after protective relays have isolated a faulted line [29,30]. Also, many engineers have used and continue to use fault analysis programs to calculate the locations of such faults. These practices are satisfactory in some cases. However, they can be time consuming, complicated, difficult to implement and inaccurate [6,14,17].

As a result of these complicated and inaccurate practices, the need arose for properly measured data of a power system to prevent, or at least control fault conditions of a system more efficiently [15,16,20]. Such measurement data typically include voltage and current measurements, taken at different geographical locations, over the same time window [8,19,24,25]. Apart from fault location applications, simultaneous measurements (or synchronised data acquisition) also have a very special role in network analysis as it can be used to form a consistent picture of the network, which is the basis for all power system monitoring, protection and control functions [18,28,30].

The research presented in this thesis is aimed at developing a relatively inexpensive system for the following applications:

- Accurate transmission line fault location systems.
- Wide area synchronised data acquisition.

Synchronised sampling and measurement can also be implemented in other monitoring and control devices and systems, such as data acquisition systems, digital fault recorders and sequence-of-event recorders [4,7]. Some of these applications will be discussed in chapter 2.

1.2 Project Description

Several techniques of fault location and power system state measurements have been developed in the past [5,6,8,9,11,15,20,25]. Each technique makes use of different power system data to perform calculations. The calculations are done with algorithms that belong to essentially three groups: Phasor-based algorithms, partial differential equation algorithms and advanced algorithms such as travelling wave analysis [9,25]. Some of these algorithms use data recorded at one end of the faulted transmission line only, while others use data recorded at both ends of the line. Algorithms that use data from both ends of a faulted line demand that data be collected at exactly the same time, which means that data samples have to be synchronised to the same time base. In addition, Wide Area Measurements (WAMs) of power systems also rely on the fact that measurement data is taken at synchronised time intervals.

For the algorithms mentioned above to be effective and accurate, synchronisation of sampling clocks at different geographic locations is essential [8,9,10]. Simultaneous measurements across the power system can be obtained by synchronising the sampling clocks at each measurement site. The synchronisation must be achieved over long distances (hundreds of kilometres), and a high degree of precision in synchronisation must be maintained [2,12].

Different methods of synchronising sampling clocks exist [1,7]. These methods will be discussed in detail in chapter 2. The synchronising methodology used in this thesis relies on Global Positioning System (GPS) timing features, which has proven to be very accurate [1,6,9,10].

The developed GPS based time stamping and scheduling system uses microprocessor based technology, Programmable Logic Array (PLA) technology and a commercial GPS receiver to provide accurate timing data, which may be used in fault location methods and other application algorithms. The system also provides accurate pre-programmed trigger signals used for the initiation of pre-programmed data acquisition runs. The measurements acquired by data acquisition hardware during these runs produce wide area synchronised measurement data.

1.3 Thesis Overview

Chapter 2 presents an overview of applications and literature applicable to the work done in this thesis. Applications such as WAMs and transmission line fault location are discussed with reference to where the developed *GPS Based Time Stamping and Scheduling System* will fit in. In conclusion, the literature review is used to compile a list of specifications for the system, which will in turn lead to a system overview.

Chapter 3 discusses the use of GPS in synchronising sampling clocks at different geographic locations. This chapter gives a summary of GPS technology, its operation and the GPS receiver used in this design.

Chapter 4 presents the Detail design and implementation of the *GPS Based Time Stamping and Scheduling System*. One of the system blocks implemented in PLA technology is discussed, namely the Real Time Clock (RTC).

Chapter 5 presents another system clock implemented with the help of PLA technology. This clock is a microsecond counter, used to keep time accurate to a microsecond.

A microcontroller is used to control the system. Chapter 6 gives detail flow diagrams representing the developed system control software.

A feature of the developed system is that it can be controlled remotely by a host computer (or a Personal Computer¹). The software used to implement the host computer control software, i.e. LabVIEWTM virtual instrumentation software, is discussed in chapter 7.

¹ PC

Chapter 8 presents results and measurements of the developed system. Estimations and comments on the accuracy of the system is given. In conclusion it states final comments and recommendations for improving the system. Points to remember for further research will also be given here.

Chapter 2

Literature study

2.1 Introduction

Much research and work has been done in the field of time synchronisation, transmission line fault location and WAMs (Wide Area Measurements) [5,6,8,9,11,15,20,25]. The aim of this chapter is to present an overview of some of the relevant work that has been done on these topics. Some applications and the mathematical theory behind them will be presented briefly. Some sources of timekeeping and time-synchronisation, as well as their advantages and disadvantages with regards to time synchronisation will be discussed in section 2.6. Finally a list of specifications will be used to compile a high-level system block diagram.

2.2 Synchronised Sampling

IEEE 1344-1995 [31] defines a synchronised phasor as follows:

“ A phasor calculated from data samples using a standard time signal as reference for the sampling process. Thus, phasors from remote sites have a defined common phase relationship.”

A phasor sample, like the one in the above definition, is derived from several samples which is associated with the data-window that spans a sample set. It is intriguing to consider the possibility of making measurements of several voltages and currents over the same data window, which would produce a simultaneous measurement set. These measurement sets have a very special role in network analysis as they can be used to form a consistent picture of a power system, which is the basis for all network monitoring, protection and control functions [28].

As mentioned in chapter 1, simultaneous measurements across the power system can be obtained by synchronising the sampling clocks at each measurement site, and the

synchronisation must be achieved over long distances. Section 2.3 shows how synchronised phasors fit into the context of fault location and other applications.

2.3 Applications of Synchronised Sampling in Power Systems

A few applications mentioned in chapter 1 will be discussed in this section. Later sections will deal with the mathematical aspects of these applications.

2.3.1 Transmission Line Fault Location

Most transmission line faults occur during severe weather conditions when lightning strikes towers or conductors, producing stresses on the insulation between the conductors and supporting structures. In addition, some natural environmental conditions such as a tree growing or a bird flying into a transmission line can cause a fault [1]. Fault location may be based on phasor measurements, or on travelling wave concepts. Figure 2-1 shows the travelling wave method principle.

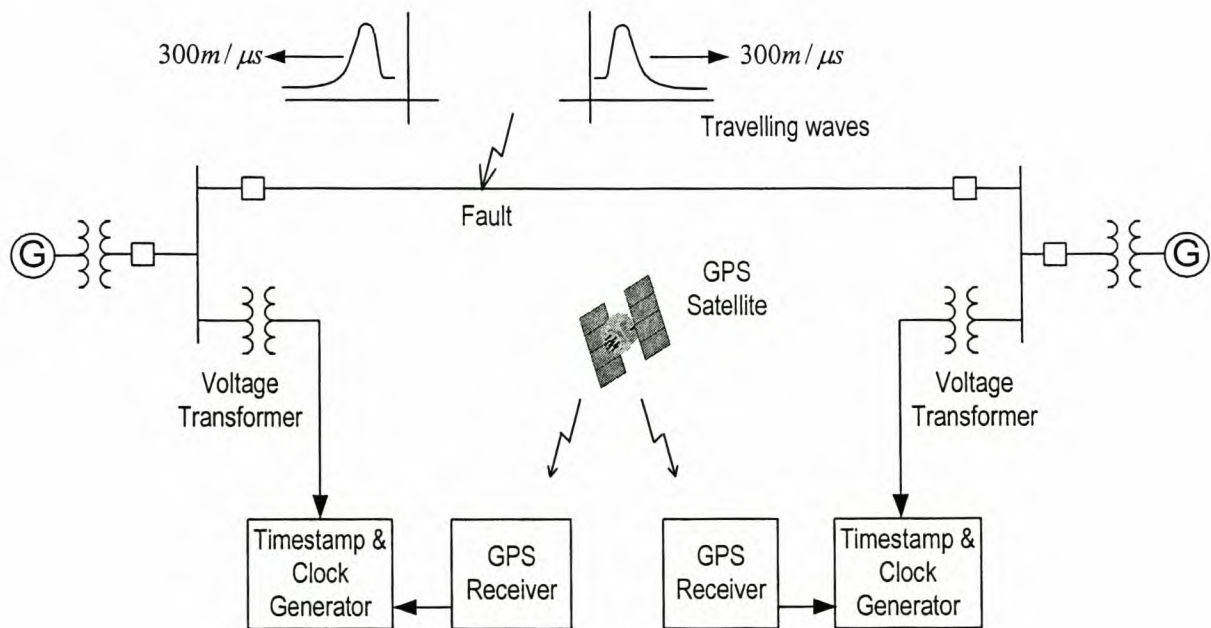


Figure 2-1 Transmission line fault location system

Figure 2-1 shows a fault occurring, be it either line-to-ground or line-to-line. This fault typically causes a travelling wave, moving at approximately the speed of light (300 000 km/s), which translates to 300 m in one microsecond [7]. If such a travelling wave can be time-stamped when it arrives at sampling equipment at both ends of such a line and the

sampling clocks are synchronised to the same time base (GPS), it can easily be calculated exactly where the fault occurred [6,8,9]. Section 2.4 will discuss these concepts.

2.3.2 Wide Area Measurements

In a power system the complex voltages of substation buses are the state vectors of the power system and hence represent the key in the application of proper control of a power system. The idea of Wide Area Measurements (WAMs) is to obtain these voltage vectors in real time. The information is obtained by initiating accurate pre-programmed data acquisition runs of data acquisition hardware at exactly the same time. Once the voltage phasors of the power system are measured and available at the substation or a central location, many applications in system monitoring and control are possible. This information is very useful for the calibration of system state estimation predictions, determining system stability margins, improved system out-of-step response and protection relay testing [7,15,16,17]. Figure 2-2 shows a conceptual explanation. A more detailed discussion of WAMs can be found in section 2.5.

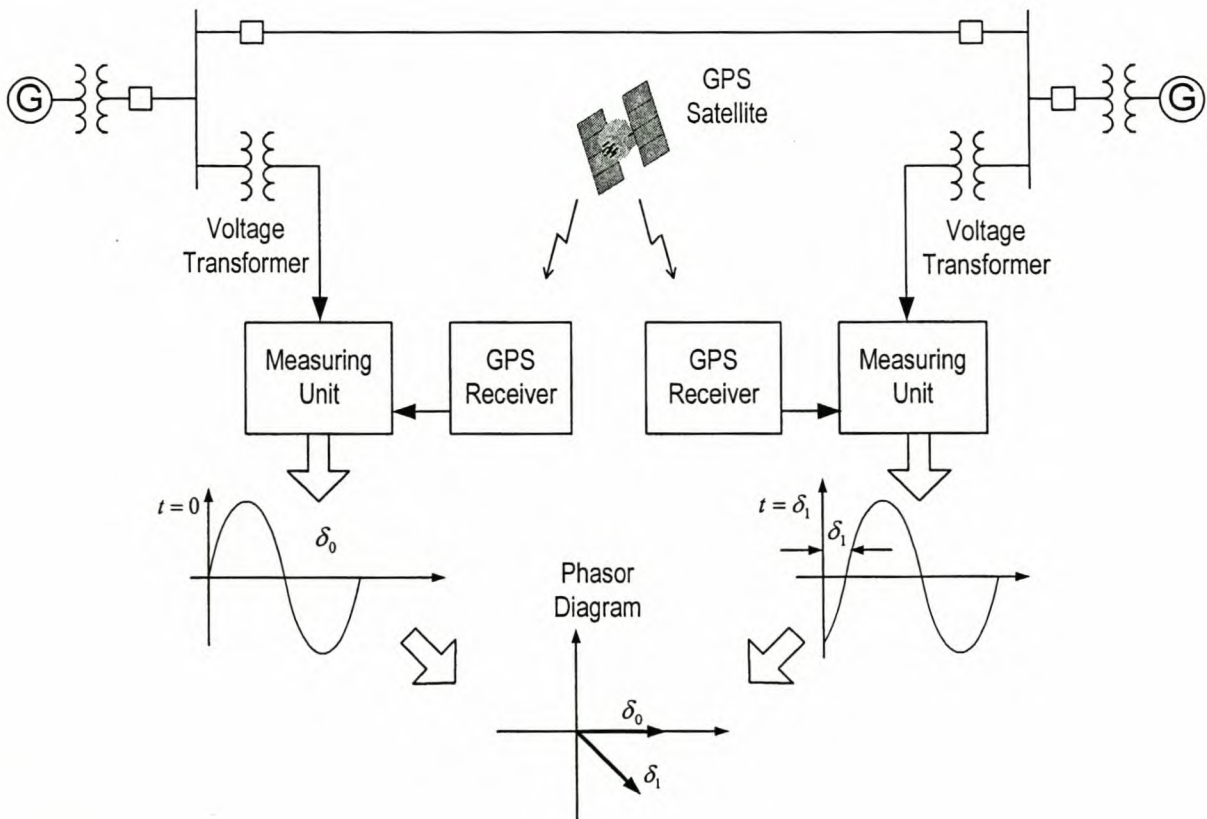


Figure 2-2 GPS Synchronised WAMs of a power system

2.3.3 Protective Relay Testing

It has long been the desire of a protective relay engineer to test protection systems under conditions that are as close to actual conditions as possible [40]. When a critical transmission line trips falsely, the reasons for this need to be discovered. A very good method of analysis would be to retest the whole protection scheme (including communications circuits) with a recording of the fault disturbance that produced the problem. This application ties in very closely to Wide Area Measurements in the sense that digital recorders need to be started at a synchronised time to be able to effectively compare measurement results, as systems that test protective relays would need to be triggered at the same time as other testing equipment on the same system.

2.4 Fault Location Algorithms

A fault location algorithm defines the steps needed to obtain the fault location by using the measurements from one or both ends of the line. A set of equations representing the mathematical model of the faulted transmission line is needed to define the algorithm. The quantities that appear in the equations are voltages, currents, transmission line parameters and fault parameters.

Two types of transmission line mathematical models are in use for fault location algorithms: the distributed parameter model and the lumped parameter model. The distributed parameter model is mostly suitable for long transmission lines. The lumped parameter model is a simplification of the distributed parameter model and is used for shorter lines only [1]. Because of this, only the distributed model will be discussed here.

In the distributed parameter model, the voltages and currents are functions of time (t) and position (x). The model consists of two linear partial differential equations of the first order. We consider the equations for the case of a single-phase transmission line [1]:

$$-v_x(x,t) = li_t(x,t) + ri(x,t) \quad (2-1)$$

and

$$-i_x(x,t) = cv_t(x,t) + gv(x,t) \quad (2-2)$$

In these equations, line parameters l , r , c and g are inductance, resistance, capacitance and conductance per unit length respectively. $v(x,t)$ presents voltage and $i(x,t)$ current. The subscripts x and t denote partial derivatives with respect to position and time. The solution of equations (2-1) and (2-2) belong to three main groups, namely phasor-based algorithms, partial differential equation-based algorithms and lastly, the travelling wave algorithm. Only the travelling wave-based algorithm will be discussed here.

Travelling wave methods do not require the solution of partial differential equations. In this approach, the line resistance r and line conductance c is neglected. Such a line is known as a lossless line, and the describing equation is known as the telegrapher's equation. A simplification of this kind is appropriate for long and high voltage transmission lines. Figure 2-3 defines the circuit of a faulted transmission line. S , F and R are the positions for sending end, fault and receiving end respectively. x is the distance to the fault, Z the line impedance and d the transmission line length. V_S , V_F and V_R are the voltages at sending end, fault and receiving end respectively. I_S , I_F and I_R are the currents at sending end, fault and receiving end respectively. Z_{ES} and Z_{ER} are the Thevenin equivalent impedances. V_{ES} and V_{ER} are the Thevenin equivalent voltages [1].

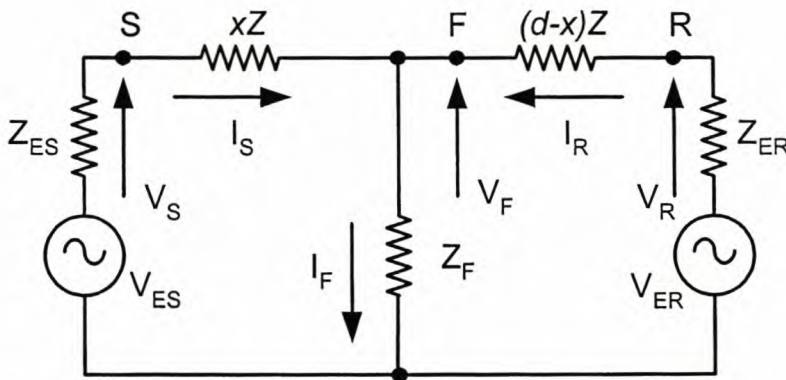


Figure 2-3 Circuit model of a faulted transmission line [1]

The solution of the two equations then has a rather simple form given by equations (2-3) and (2-4) for voltage and current respectively. The voltage and the current are linear combinations of two components known as forward and backward travelling waves denoted by S_F and S_B respectively [1]:

$$v(x,t) = [S_F(t - \gamma x) + S_B(t + \gamma x)]/2 \quad (2-3)$$

and

$$i(x,t) = [S_F(t - \gamma x) - S_B(t + \gamma x)]/2Z_0 \quad (2-4)$$

where $Z_0 = \sqrt{l/c}$ is the surge impedance of the line, and $(t - \gamma x)$ and $(t + \gamma x)$ define parallel lines in the position/time plane, as shown in Figure 2-4.

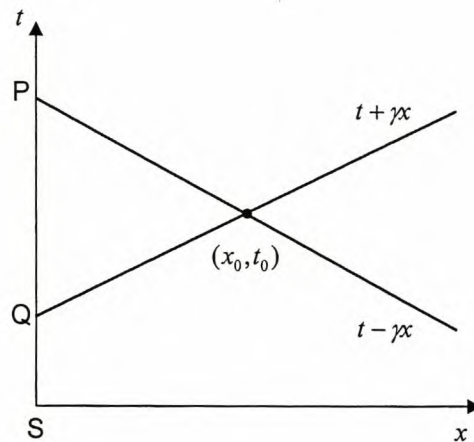


Figure 2-4 Characteristics in the position-time plane

The forward and backward travelling waves may be calculated from the sending-end voltage $v(0,t) = v_s(t)$ and the sending-end current $i(0,t) = i_s(t)$ as follows [1]:

$$S_F(t) = v_s(t) + Z_0 i_s(t) \quad (2-5)$$

and

$$S_B(t) = v_s(t) - Z_0 i_s(t) \quad (2-6)$$

Fault location uses the transient component of the travelling waves only. The transient travelling waves appear in the transmission line after any abrupt change in voltage and current. When a fault occurs, the voltage at the fault point F drops. This generates a backward and forward travelling wave at the location of the fault [1]. These travelling waves do not change shape until they reach some discontinuity in the transmission line. The discontinuities are the sending end, the receiving end, and the fault itself. When a travelling wave arrives at a discontinuity, it ceases to exist in its original form, and two new waves emerge at the discontinuity. The first is a reflection of the original wave which has the shape

of an attenuated original wave and continues motion in the same direction as the original wave. The coefficients determining the magnitudes of both new waves depend on the type of fault. Low impedance faults have high coefficients of reflection, and high impedance faults have low coefficients of reflection. The motion of the travelling waves along the transmission line and the generation of new waves at points of discontinuity are represented by the lattice diagram in Figure 2-5.

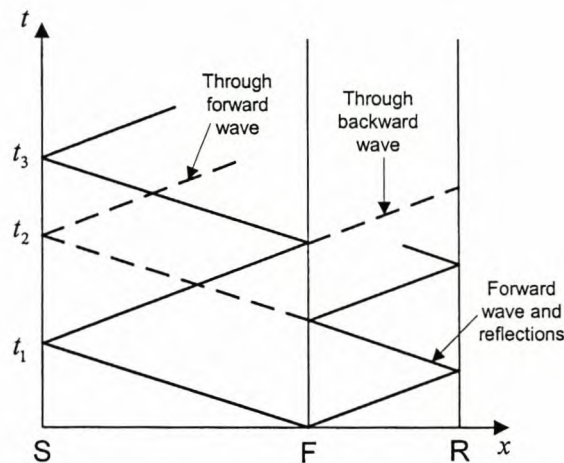


Figure 2-5 Lattice diagram

The idea to use reflections to estimate the fault location appeared in 1930 for the fault location of underground cables. A cable is energised with a short voltage impulse. The impulse and its reflection are recorded, and the travel time is found. Later, similar devices were used to measure the fault location for transmission lines. These methods are called active methods. The calculation of the elapsed time is easy if the inserted pulse and its reflection have sufficient power. However, travelling waves caused by a fault may have low power, especially if the fault occurs when the instantaneous voltage at the point of the fault is close to zero. In that case the calculation of this time requires special signal processing. One of the signal processing methods most commonly used is the correlation technique, which is covered by Ancell *et al* [43]. In the other case where the travelling wave has sufficient power, the determining of the fault location lies in these two simple relations:

$$D_S = \frac{l + (T_{1S} - T_{1R})v}{2} \quad (2-7)$$

and

$$D_R = \frac{l - (T_{1S} - T_{1R})v}{2} \quad (2-8)$$

where D_S and D_R represent the distance from the sending end S to the fault F and the distance from the receiving end R to the fault F respectively. l denotes the transmission line length and T_{1S} and T_{1R} are the times when the fault-generated transients first arrive at the sending (S) and receiving (R) end bus bars. v is the propagation speed of a travelling wave. This velocity is very close to the speed of light (300 km/ms) for transmission line cables.

The concepts in this section were only briefly discussed and more detail and examples on the travelling wave concept and the *wavelet* analysis method can be found in references [1,2,5,6,8,9,10,11,12].

2.5 Wide Area Measurements

A WAMs system collects satellite-synchronised data to control a power system reliably while operating the power system closer to its capacity limits [19,20,21]. Measurements are based on GPS synchronised Phasor Measurement Units. These devices are briefly discussed in section 2.5.1. The aim of WAMs is to detect system dynamic disturbances and prevent propagation of such instabilities, which, if not localised and stopped, might lead to widespread system outages and ultimately regional blackouts.

2.5.1 Phasor Measurement Units

Phasor Measurement Units (PMUs) using synchronisation signals from the GPS satellites have evolved into advanced systems and are being manufactured commercially [21,22,26]. Figure 2-6 shows a functional block diagram of a typical PMU.

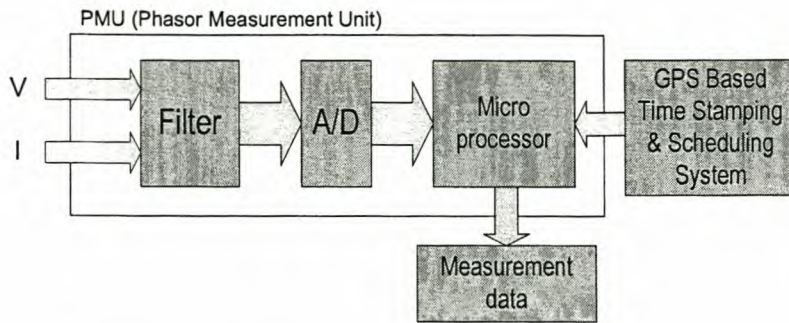


Figure 2-6 Phasor Measurement Unit (PMU) [20]

2.5.2 Phasor Measurement Process

The basic phasor measurement process is that of deriving positive sequence, fundamental frequency phasors from voltage and current waveforms captured by a PMU shown in Figure 2-6. There are a variety of methods for implementing this process [19,21,22] with many possible hardware implementations. PMUs currently in use [19] digitise three phase power line waveforms at 720 Hz or 2880 Hz (for a 60Hz system). They compute phasors from these digital samples using a Discrete Fourier Transform (DFT). The DFT computation is referenced to UTC (French for Co-ordinated Universal Time), which will be defined in section 2.6. As mentioned earlier, the precise time reference allows comparison of phase angles between stations. The phasor measurement process as implemented in a typical PMU is shown in Figure 2-7 [19].

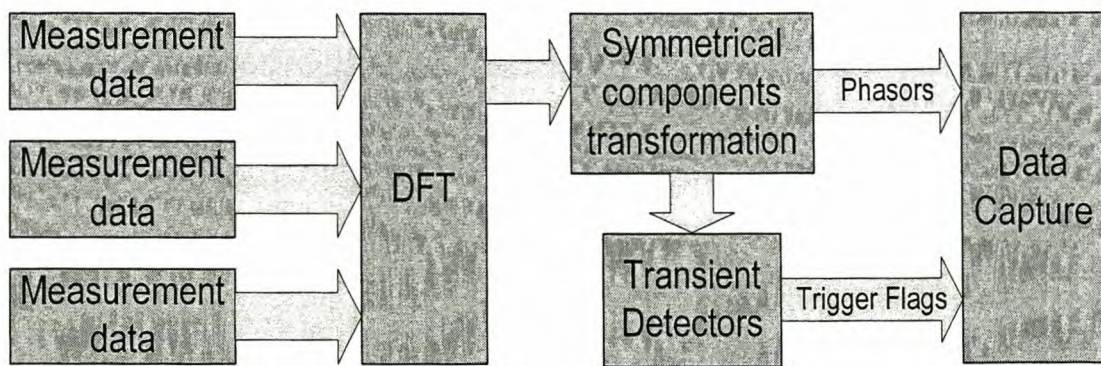


Figure 2-7 Phasor measurement process [19]

After the phasors have been stored, it is brought together as a single measurement set [19,21,28].

2.6 Sources of Timekeeping

Only an introduction can be given to the complex subject of timekeeping sources. References [32-36] can be consulted for a more detailed discussion on different methods only briefly mentioned here. By the early 1950's scientists had developed atomic clocks (oscillators) with a high degree of accuracy and stability. A time scale called Atomic Time (AT), based on Cesium and Rubidium oscillators [13], was developed. With atomic clocks scientists could accurately measure changes in the earths' motions. A time scale called Earth Time (ET) was developed from astronomical observations. These two time scales, i.e. AT and ET, could be out of step and any country could maintain its own ensemble of atomic oscillators for its own version of AT [7].

In an attempt to create a world wide time scale, over 70 nations now contribute astronomical, stability and accuracy information to the Bureau International des Poids et Mesures (BIPM) in Paris, France. UTC, which is French for Co-ordinated Universal Time, is co-ordinated by BIPM. Each of these 70 nations has a different version of UTC and they report their version to BIPM. The differences between versions of UTC was found to be not more than 5 μ s [7]. Because of BIPM, the overall difference between different versions of UTC now vary between 0.2 and 0.5 μ s. With respect to applications discussed earlier, these differences are tolerable, because most Wide Area synchronised Measurements will be done on a national scale, so the same UTC would be used to synchronise sampling clocks.

2.6.1 Time Transfer Techniques

Time transfer techniques refer to methods of getting the timing signals to the user. These timing signals will then be used to synchronise sampling clocks, as applications demand.

An ideal transfer technique would have the following characteristics [7]:

- A very stable and predictable propagation delay.
- A wide bandwidth to permit very high rise times of timekeeping pulses.
- Noise free.
- World wide coverage.
- Immunity to substation interference.
- Perfect reliability.

- Reasonably priced receivers and moderately sized antennae.
- Complete time and date information.

There are many ways of transferring time and frequency information. The practical methods encountered by the author were the following:

- Medium frequency and high frequency radio broadcasts (2.5 to 20 MHz).
- Low frequency and very low frequency broadcasts (10-15 kHz).
- Ultra high frequency or microwave broadcasts from navigational satellites.
- High accuracy portable Cesium, Rubidium or quartz clocks.

Each method mentioned above has various degrees of accuracy, ease-of-use, reliability and cost. Many factors influence the cost of a time transfer system, such as complexity, portability and availability of receiver modules, size and power of transmission equipment and immunity to substation noise.

Table 2-1 gives a comparative summary of the different methods of time transfer in terms of broadcast frequency, accuracy and some comments on the service. The reader is referred to references [7,13] for more information on these services.

Table 2-1 Time transfer techniques

Service	Frequency	Accuracy	Comments
Low/Medium frequency broadcasts	3 – 60 MHz	1-5 ms	Intermittent reception. Reception difficult at substations.
Very Low frequency broadcasts	60 kHz	1 ms	System not widely used, thus not maintained.
Loran-C (navigational service)	100 kHz	1-3 μ s	Substation reception difficult. Not a global system. Only in Western hemisphere.
GOES (Geostationary Operational)	468 MHz	100 μ s	Occasionally inaccurate because of age. Broadcast frequency shared with some land mobile

Environmental Satellites)			radios. Not a global system (Western hemisphere), receivers bulky and expensive.
GPS (Global Positioning system)	1.5 GHz	< 1 μ s	Proven μ s service by US Department of Defence and it is a global system, used for hundreds of applications. Receivers are cheap and readily available.

When inspecting Table 2-1, it is noted that the Global Positioning System (GPS) is an attractive option of time transfer because of its accuracy and availability. These are only a few of the advantages of GPS. GPS operation, as well as the GPS receiver used in this development will be discussed in chapter 3.

2.6.2 Oscillator Technology

It is a well-known fact that a chain is only as strong as its weakest link. When time transfer techniques are as accurate as pointed out in the previous section, accurate on-board timing is required in the receiving system to keep overall system accuracy to a maximum. The system only uses the GPS receiver to synchronise an on-board clock and to download the current time on start-up. An on-board clock should thus be as stable as possible. A Crystal oscillator's accuracy is measured in time lost in a million units of time. An oscillator with an accuracy of 100 parts per million (PPM) will typically lose or gain 100 microseconds every second. Because these inaccuracies can go both ways, it is commonly referred to as "clock drift". Quartz crystals typically drift due to thermal, mechanical, and aging effects. In this section some precision timing methods will be discussed [42].

2.6.2.1 Crystal Oscillator

A conventional Crystal Oscillator (XO) is an oscillator circuit that bases its oscillating characteristics on a quartz crystal. The crystal's characteristics change with changes in temperature and humidity, in other words its oscillating frequency drifts from the specified frequency with such changes. A typical oscillator circuit and crystal are enclosed in a single

metal casing and it is therefore not isolated from temperature fluctuations. A typical crystal oscillator has an accuracy of 100 PPM, which means that the oscillator could be inaccurate up to 100 microseconds over a period of second. For the applications described earlier, this type of oscillator would be inadequate. Even with GPS receiver synchronising such a crystal oscillator, it would still not be accurate enough, because 1 μs (1 PPM) accuracy is desirable.

2.6.2.2 Oven Controlled Crystal Oscillator

All the abovementioned problems are overcome with an Oven Controlled Crystal Oscillator (OCXO). Figure 2-8 shows a diagram of such an oscillator.

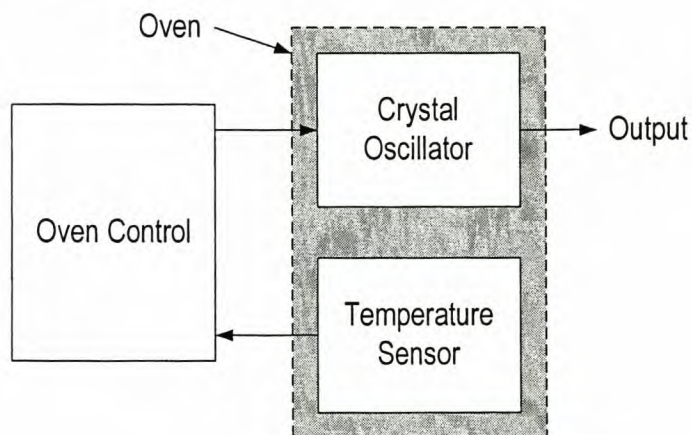


Figure 2-8 Oven controlled crystal oscillator

The “oven” keeps the crystal oscillator at a constant temperature, and thus minimises fluctuations in frequency stability. Oscillators such as these can achieve accuracies of up to 0.1 PPM, which is desired for this application.

2.6.2.3 Atomic Oscillator Technology

In these sections, atomic oscillator technology will be briefly discussed. Because of their complexity, price and size, it is not feasible to be used as on-board oscillators. It is, however, of interest because such oscillators are used in atomic clocks around the world. The Cesium atomic clock is described in more detail, as it is the atomic clock used by GPS. References [42] may be consulted for in depth details on other atomic clocks.

- *Cesium atomic clocks* are the most commonly used atomic clock. It employs a beam of cesium atoms. To turn the cesium atomic resonance into an atomic clock, it is

necessary to measure one of the cesium atoms' resonant frequencies accurately. Locking a crystal oscillator to the principal microwave resonance of the cesium atom normally does this. This signal is in the microwave range of the radio spectrum. To create a clock, cesium is first heated so that atoms boil off and pass down a tube maintained at a high vacuum. The atoms pass through a magnetic field that selects atoms of the right energy state, which are then passed through an intense microwave field. The frequency of the microwave energy sweeps backward and forward within a narrow range of frequencies, so that at some point in each cycle it assumes a frequency of exactly 9,192,631,770 Hz. The range of the microwave generator is already close to this exact frequency, as it is generated by an accurate crystal oscillator. When a cesium atom receives microwave energy at exactly the right frequency, it changes its energy state. At the other end of the tube, a magnetic field separates the atoms that have changed their energy state if the microwave field was at exactly the correct frequency. A detector at the end of the tube gives an output proportional to the number of cesium atoms striking it, and the output peaks when the microwave frequency is exactly correct. This peak is then used to make the slight correction necessary to keep the crystal oscillator, and thus the microwave field, exactly at the correct frequency. This locked frequency is then divided by 9,192,631,770 to give the familiar one pulse per second required for timing applications.

- *Hydrogen atomic clocks* maintain hydrogen atoms at a required energy level in a container constructed with walls of insulating material to ensure that the atoms do not lose their high-energy state.
- *Rubidium atomic clocks* are, compared to other atomic clocks, rather simple and compact. Rubidium gas changes its absorption of light at the optical rubidium frequency when the frequency of an applied microwave beam is at the desired value.

2.7 GPS Based Time Stamping and Scheduling System Overview

In this section, the hardware topology for the *GPS Based Time Stamping and Scheduling System* is discussed. Out of the information gathered and presented in this chapter, the following specifications would be needed in the proposed system:

- For fault-location applications, a time stamping accuracy of $1 \mu\text{s}$ is necessary. A travelling wave caused by a transmission line fault would travel 300 meters in $1 \mu\text{s}$ at approximately the speed of light. With such accuracy, it would be possible to trace the location of a fault down to approximately one transmission line tower span. An external data acquisition system [53] is required to provide a trigger signal (named Trigger IN), signalling the need for time information. The developed system must then be able to provide time information accurate to $1 \mu\text{s}$.
- As mentioned in chapter 1, the developed system must be able to generate a trigger signal at a pre-programmed time, accurate to $1 \mu\text{s}$, in order to start synchronised data acquisition runs for WAMs, using compatible data acquisition hardware [53]. A signal Trigger OUT is defined with this purpose in mind. This signal can be generated at a time accurate to $1 \mu\text{s}$, although this kind of accuracy would not be necessary as the frequency of the South African national power system is 50 Hz. An accuracy of $1 \mu\text{s}$ would translate into 0.018 degrees accuracy [41] on a phasor measurement for a 50 Hz power system, which would be sufficient.
- To increase the response of the system to a trigger signal, an on-board Real Time Clock (RTC) is needed. This implies that the GPS receiver need not be prompted for time information every time a trigger signal is received. This also facilitates the generation of a trigger signal at a pre-programmed time, as synchronised time information is always available on-board.
- It would be convenient to store received trigger data on-board for analysis at any time. For this purpose on-board memory, which could be read by compatible hardware or a personal computer, would be used.
- A microcontroller is used to control the different blocks of the developed system. The microcontroller needs two Universal Asynchronous Receiver Transmitters (UARTs) to communicate with a personal computer and the GPS receiver. It also needs enough

input/output pins to set-up an 8-bit data bus, an 8-bit address bus and a 10-bit control bus.

With the above specifications in mind, an outline on the implementation thereof can be compiled.

- To achieve an on-board timing accuracy of 1 μ s, an accurate binary counter is needed. The counter has to be used in conjunction with the on-board RTC, and must be able to count up to 1 000 000 (the amount of microseconds in a second). In order to implement such a counter with enough precision, and to be able to read its value through an 8-bit data bus, the counter value must consist of three bytes. Thus, a counter with maximum value of 2^{24} is needed. These two blocks, the RTC and the microsecond counter, is implemented using Erasable Programmable Logic Device (EPLD) technology [47].
- The RTC and microsecond counter (with a 1MHz clock signal) has to be synchronised by a highly accurate (± 500 ns) pulse from the GPS receiver. This signal is known as the 1 pulse-per-second. Chapter 3 will describe this pulse in detail.
- For storage of trigger data a FIFO memory is used. This will assist in downloading large amounts trigger data for analysis to a compatible device or a personal computer.
- The microcontroller used is the ATmega161L from Atmel [46]. Design details will be provided in chapter 6. The AT90S8515 was used in a previous version of this system, but was found to be inadequate in terms of communications peripherals and on-board memory. The routines for implementing software communications will however be presented in chapter 6 and Appendix F.
- Figure 2-9 presents the specifications mentioned above in block diagram format.

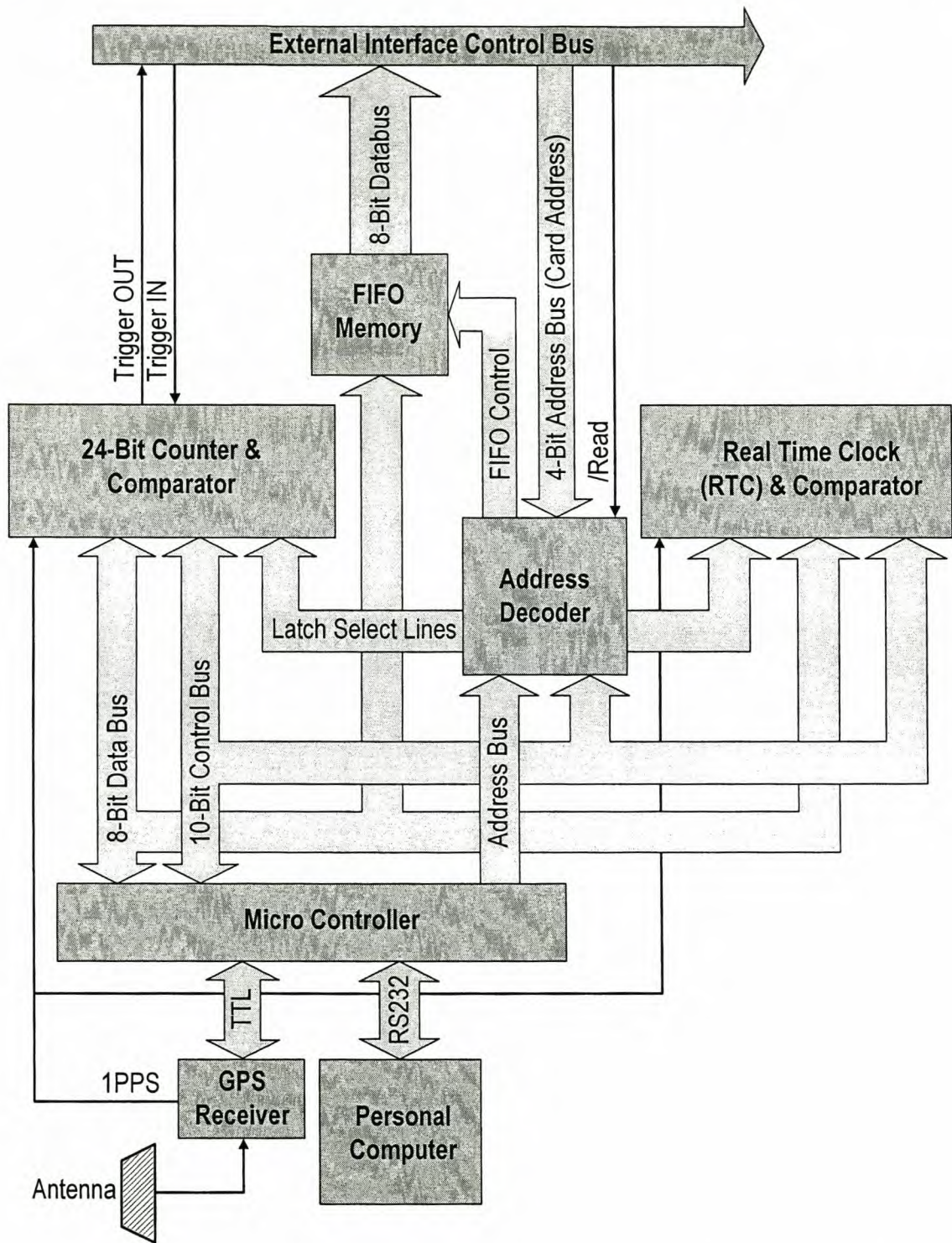


Figure 2-9 GPS Based Time Stamping and Scheduling System block diagram

Chapter 3

Introduction to GPS

3.1 GPS Segments

The Global Positioning System (also called NAVSTAR by US Department of Defence) is an array of military navigational satellites. GPS timekeeping is tied to UTC, which, as discussed earlier, is the global time standard. The civilian user is able to obtain navigational and geographical positions in three dimensions, velocity and highly accurate time from GPS satellites [37,38,39]. GPS is divided into three segments namely the space segment, the control segment and the user segment, as can be seen in Figure 3-1 [38].

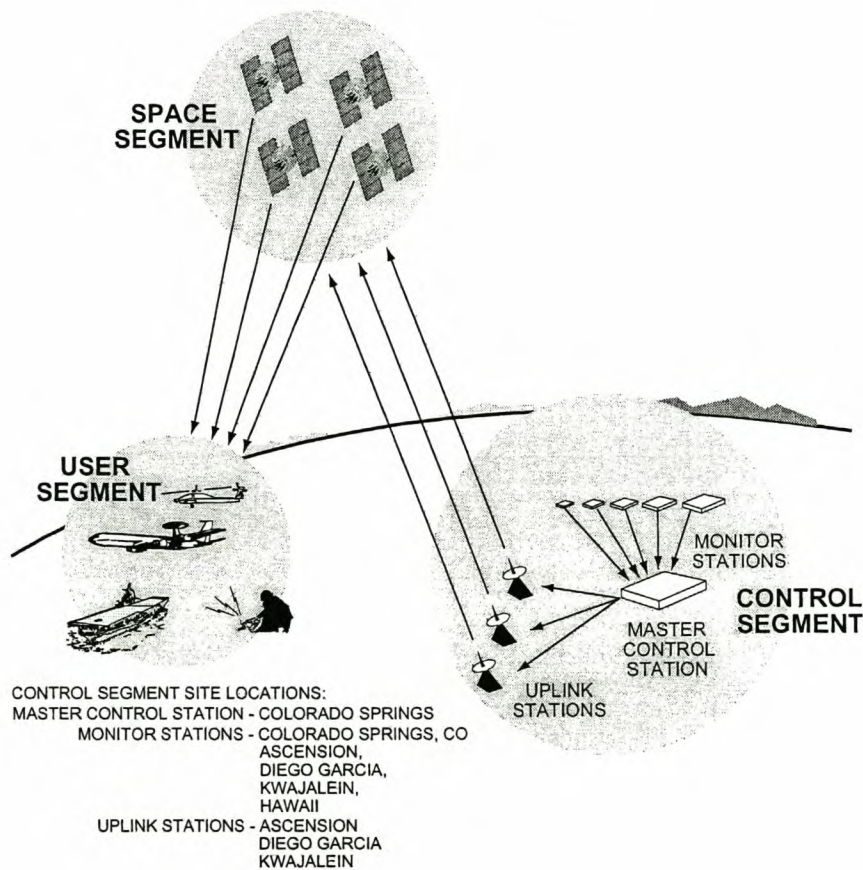


Figure 3-1 GPS system segments [38]

3.2 GPS Operation

The space segment of GPS consists of four satellites in each of six orbital planes for a total of 24 operational satellites, as well as 3 in orbit spare satellites. By observing a minimum of four satellites simultaneously, a GPS receiver can automatically determine its own location by triangulation. The satellites orbit at 19650 kilometres above the earth [38,39]. This high altitude allows signals to be broadcast over a much wider area than would be possible with lower orbit satellites. The satellites travel at about 11200 km/h, circling the earth once every 12 hours [37,38,39].

Navigational and time information are transmitted at two levels of accuracy on two different L-band microwave frequencies, namely L1 (1.57543 GHz) and L2 (1.2276 GHz). L1 contains two “pseudo-random” signals, i.e. the protected (P) and Coarse/Acquisition (C/A) code. Each satellite transmits a unique code, allowing the GPS receiver to uniquely identify the satellites. The main purpose of the coded signals is to allow for calculating the travel time from the satellite to the GPS receiver on earth. The travel time is multiplied by the speed of light to produce the satellite range (distance from the satellite to the GPS receiver on earth). The Navigation message, which in effect is information the satellites transmit to a receiver, contains satellite orbital and clock information, and general system status messages. Every satellite carries a cesium beam frequency standard. It is essential to the accuracy of the navigational solution that the time and frequency standards of these satellites be controlled to the maximum extent possible [37,38,39].

The control segment of GPS does exactly what the name implies. It controls the entire system by tracking the satellites and providing them with corrected orbital and clock (timing) information. There are five control stations located around the world, namely four unmanned stations and one master control station. The four unmanned receiving stations constantly receive data from the satellites and then send that information to the master control station. The master control station corrects the satellite data and, together with two other transmitting stations, sends the information to the GPS satellites [37,39].

The user segment consists of GPS receivers that are available freely today for a fraction of the cost it was 10 years ago. The US Department of Defence began a policy of selective availability (SA) that inserts an intentional error in the C/A code. This hides encoded data from the civilian user and weakens the positioning capabilities of the GPS receiver. This is

because GPS was initially a military service, which would be most handy to an opposing military entity. Even though SA was introduced, GPS accuracy still exceeded the accuracy demands of electrical utility applications, such as WAMs and fault location. In March of 2000 the US Department of Defence deactivated SA. That decision meant that public users could utilise GPS at its full accuracy ability [37,39].

3.3 GPS Receiver Operation (“User Segment”)

While the internal operation of a GPS receiver is extremely complex, the user will find it to be the simplest system for time and frequency purposes. The reason is that the GPS system is completely self-contained and makes all necessary corrections automatically. As mentioned in chapter 2, GPS receivers are relatively cheap and are readily available.

For GPS to function correctly, the GPS receiver needs two crucial pieces of information. It has to know the location of every satellite and the distance to those satellites. The GPS receiver picks up coded information, which is called the Almanac data. This data contains approximate positions of all the satellites and it is continually transmitted and stored in the memory of the GPS receiver. Thus it knows the orbits of the satellites and where each satellite is supposed to be. The almanac data is periodically updated with new information as the satellites move around.

Any satellite can travel slightly out of orbit, and because of that, the ground monitor stations keep track of the satellite orbits, altitude, location and speed. The ground stations send the orbital data to the master control station, which in turn sends the corrected data to the satellites. This corrected and exact position data is valid for about six hours, and is transmitted in coded form to the GPS receiver. This allows the GPS receiver to determine the location of GPS satellites at all times.

At this point an interesting situation arises. Figure 3-2 displays it diagrammatically.

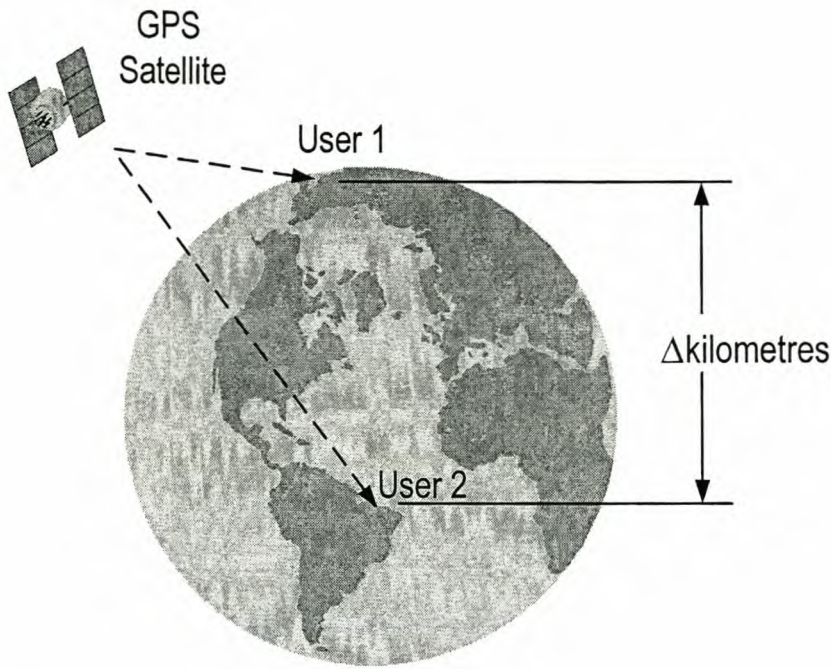


Figure 3-2 Two users receiving the same navigation data at different times

Two users, Δ kilometres apart, receive data from the same satellite. Because these two users are far apart, how is it possible that one user can calculate exactly the same time (UTC) information from data that reaches him at a different time? The answer is that in order to calculate sufficiently accurate data, a user must be able to communicate with *at least* four satellites to be able to calculate all of the unknowns in such a system.

Even though the receiver knows the location of every satellite in space, it still needs to determine the distance to these satellites so it can determine its position on earth. As mentioned earlier, the distance is calculated by multiplying the speed of light, which is the propagation speed of a radio wave, with the time it takes the radio wave to reach the receiver. Equation (3-1) [39],

$$R_1 = C(\Delta t_1 + \Delta T - \tau_1), \quad (3-1)$$

calculates R_1 , which is the distance from a user to satellite 1. C is the speed of light, t_1 is the signal travel time, T is the GPS receiver clock error and τ is the satellite clock error. Four satellites will give four equations of the form of equation (3-1). By using these equations, a GPS receiver will then be able to calculate all of the unknowns in equation (3-1). The correct time and position data can then be calculated even though different users receive data at different times from the same satellite.

The time it takes the radio wave to reach the receiver is determined by comparing the “pseudo-random code” generated by the GPS satellite to the code generated by the receiver itself. The receiver then determines how much this code needs to be delayed to match up exactly with the satellite code. This delay is the time it takes the code to reach the receiver, and so the distance to a satellite can be calculated.

Once both satellite location and distance have been calculated, the receiver can compute position. Figure 3-3 gives a good summary of the operation of GPS [38]. References [33,37,38,39] contain more information on the operation and calculations of GPS.

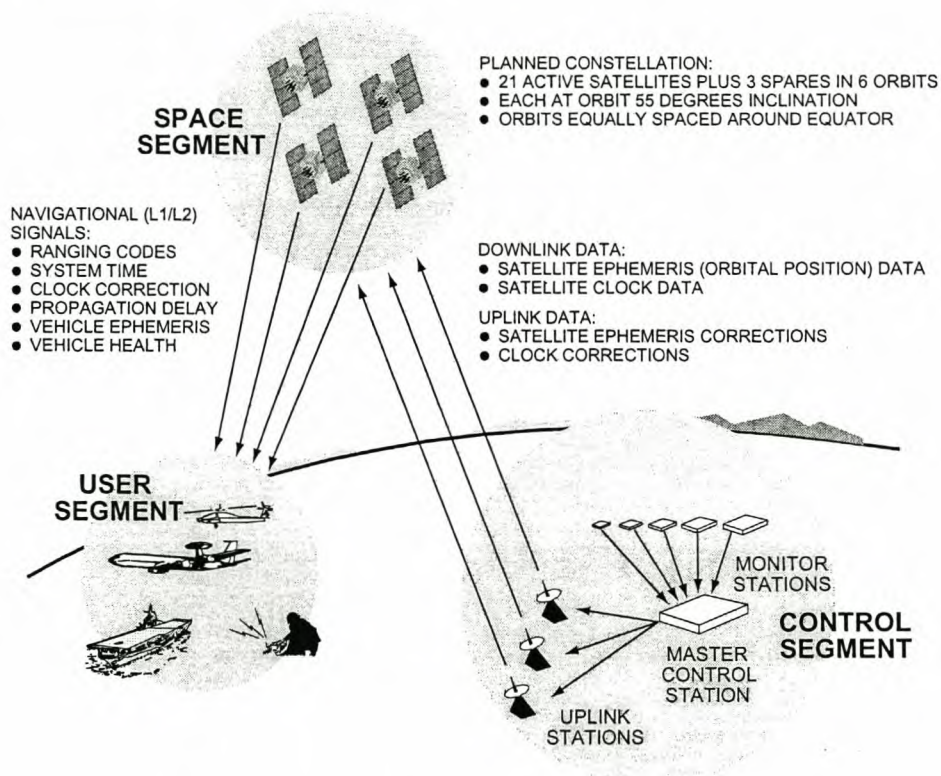


Figure 3-3 GPS system data signals [38]

It is important at this stage to note that the most important feature of the GPS receiver used in this design is the timing pulse that it outputs every second. This pulse is called the 1 Pulse Per Second (PPS) and it is used to synchronise the *GPS Based Time Stamping and Scheduling System* on-board clock to UTC. Manufacturers of GPS receivers have also designed GPS receivers with 10 PPS and 100 PPS capabilities [39]. The 1PPS signal will be discussed in section 3.4.3.

Most modern GPS receivers are a parallel multi-channel design. Older single channel designs were once popular, but were limited in their ability to continuously receive signals in the toughest environments, such as under heavy tree cover.

Parallel receivers typically have between five and twelve receiver circuits, each devoted to one particular satellite signal, so strong locks can be maintained on all the satellites at all times. Parallel-channel receivers are quick to lock onto satellites when first turned on and perform the required calculations. For the purpose of the design done in this thesis, a Motorola Oncore™ GT+ GPS receiver was used.

3.4 Motorola Oncore™ GT+ GPS Receiver

The Motorola Oncore™ GT+ GPS receiver is an 8 channel parallel receiver, specially designed for OEM (Original Equipment Manufacturers) applications.

The most important features of this GPS receiver are the following [38]:

- Operates on 5V dc regulated power.
- TTL interface to host equipment.
- Latitude, longitude, height, velocity, heading, time and satellite status information output either once every second, or polled.
- NMEA (National Marine Electronics Association) 0183 output.
- 1 PPS output (± 500 ns accuracy).

3.4.1 Functional Block Diagram

Figure 3-4 presents a functional block diagram of the GT+ receiver [38].

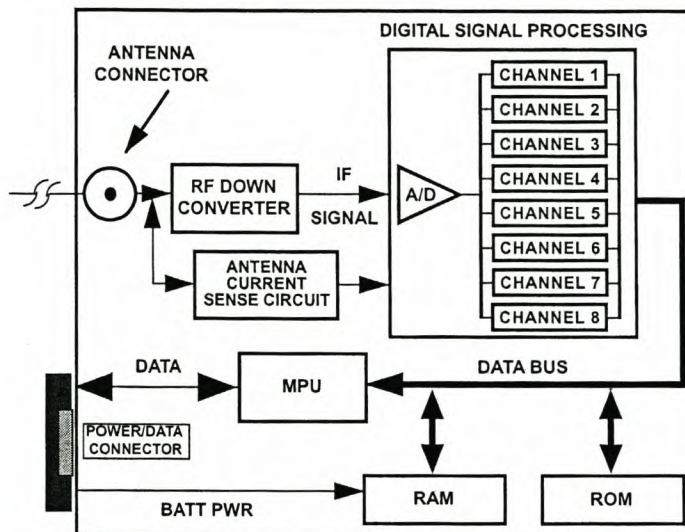


Figure 3-4 GPS receiver block diagram [38]

Because the GT+ receiver is designed for OEM applications, the application designer has to design the interface to the receiver, which also means connections to the receiver have to be made. Table 3-1 shows the receiver power requirements.

Table 3-1 GPS receiver power requirements

	Main power	Backup power
Voltage	4.75 V to 5.25 V, 50mV _{pp} ripple	2.5 V to 5.25 V
Current	0.9 W at 5 V	5 μA @ 2.5V, 100 μA @ 5 V
Comments	Active antenna draws 20 mA	Used only for memory backup

3.4.2 Terminal Connections

Figure 3-5 shows the connector to the receiver, and Table 3-2 gives the connector terminal pin assignments.

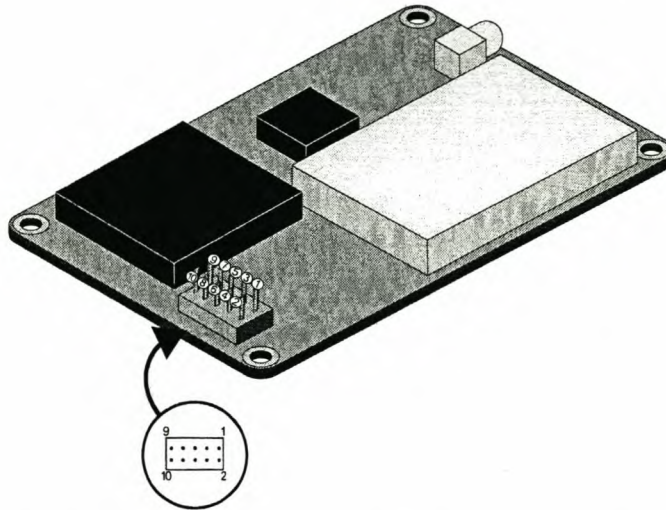


Figure 3-5 Connector terminal of the GPS receiver [38]

Table 3-2 GPS receiver connector terminals

Pin #	Signal name	Description
1	BATTERY	Externally applied backup power
2	+ 5 V POWER	Regulated main power
3	GROUND	Ground (receiver)
4	VPP	Flash memory programming voltage
5	RTCM in	RTCM input (not used)
6	1 PPS	One pulse per second output
7	1 PPS RTN	One pulse per second return
8	TTL TXD	Transmit 5 V logic
9	TTL RXD	Receive 5 V logic
10	TTL RTN	Transmit/Receive return

3.4.3 One-Pulse-Per-Second Signal

The 1 PPS signal is used to synchronise the on-board clock of the developed system in a manner that will be discussed in later chapters. The 1 PPS signal is defined as follows [38]:

- 0 to 5 V pulse.
- 1 PPS time mark is synchronous with the midpoint of the rising edge of the pulse rising from 0 to 5 V.
- Rise time is approximately 20 to 30 ns.
- Pulse width is approximately 200 ms \pm 1 ms.
- Accurate to < 500 ns in stand alone mode (with SA on).
- Requested serial data is output 0 to 50 ms after 1 PPS output.

Figure 3-6 gives a graphical representation of the definition given above.

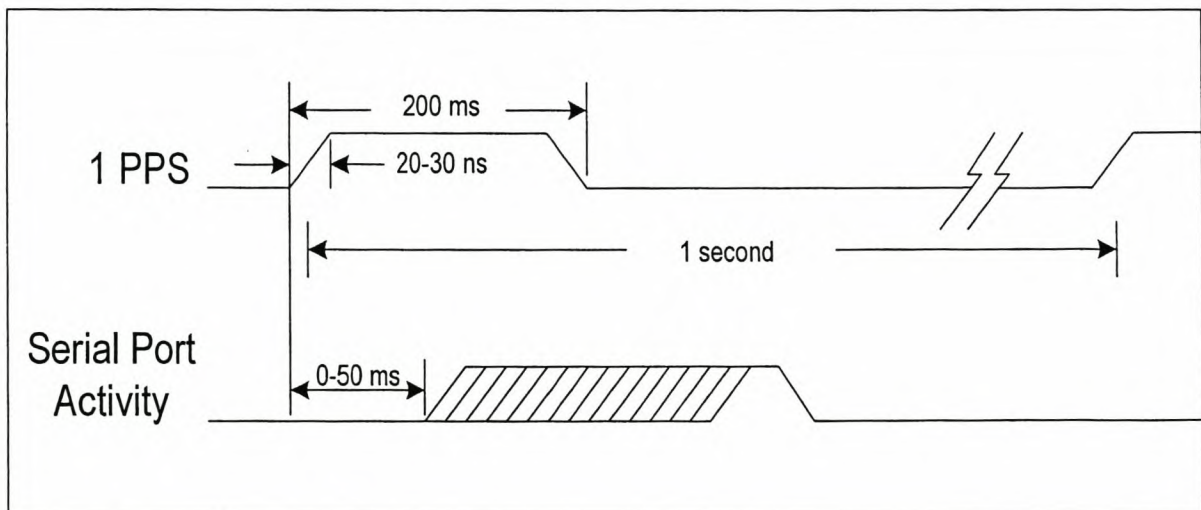


Figure 3-6 GPS receiver 1 pulse per second (1PPS) definition

3.5 Serial Interface Protocol

The receiver is an intelligent GPS sensor intended to be used as a component of precision positioning, navigation or timing system. The receiver is capable of providing autonomous position, velocity and time information over a serial TTL port. These messages may be polled or sent at a programmable rate per second.

3.5.1 Interface Protocol

To connect the GPS receiver to a system using a RS-232 interface, circuitry must be provided to convert TTL to RS-232. When connected to a personal computer, the GPS receiver may be controlled with Motorola WinOncore™ OEM GPS receiver control software (Appendix A and B).

Table 3-3 defines the interface protocol of the GPS receiver for OEM applications:

Table 3-3 GPS receiver interface protocol

Format	Motorola Binary	NMEA 0183
Type	Binary	ASCII
Direction	In/Out	In/Out
Port	1	1
Baud Rate	9600	4800
Parity	None	None
Data Bits	8	8
Start/Stop bits	1/1	1/1

The two communication formats supported by the GPS receiver is the Motorola Binary format, and the NMEA 0183 format. The user is able to select via software which format to use.

3.5.1.1 Motorola Binary Format

A message in the Motorola Binary Format will have the following components:

- *Message start:*
“@@” denotes the start of a binary message.
- *Message ID:*
(A..Z)(a..z,A..Z), which is an ASCII uppercase letter, followed by an ASCII lowercase or uppercase letter. These two characters together identify the message type and imply the correct message length and format.
- *Binary data sequence:*
Variable number of bytes of binary data dependent on the command type.
- *Checksum:*
The exclusive OR of all bytes after the “@@” and prior to the checksum.
- *Message terminator:*
<CR><LF> that represents a carriage return and line feed, denoting the end of a binary message.

Every receiver command has a corresponding response message so it can be determined whether the command has been processed successfully. An input command will be rejected completely if one of the input parameters is out of range, or if the checksum is invalid. The GPS receiver will operate on all full messages received during the previous one second interval and will process them in the order they are received. The Motorola Binary Format for command messages will be used in the design because it is less complex when compared to the NMEA format. The input commands used are given in Table 3-4. The reader is referred to Appendix B for a complete listing and details of all the Motorola Binary commands.

Table 3-4 GPS receiver commands used

Command description	Binary command	Parameters
Time of day request	@@Aa	Three out of range bytes for hours, minutes and seconds.
Set GMT offset	@@Ab	Three bytes, containing the sign, hours and minutes GMT offset – +02:00 for South Africa.

Date request	@@Ac	Four out of range bytes for day, month, and year (two bytes).
Set Time Mode	@@Aw	One byte selects between UTC and GMT time mode.
Receiver ID request	@@Cj	None. This command requests the receiver to output its ID message.
Power on fail message	@@Sz	Receiver needs to be repaired.

3.5.1.2 NMEA 0183 Format

The NMEA format allows direct interfacing via serial port to an electronic navigation instrument that support specific output messages. This format will not be discussed in much detail, because it is not used in this design.

All NMEA command messages are formatted in sentences that begin with the ASCII “\$” and end with ASCII <CR> and <LF>. A five character address occurs after the “\$”. A typical NMEA command message (e.g. time of day) will have the form

$$\$PMOTG,ZDA,yyyyCC<CR><LF>,$$

where yyyy is the update rate, and CC is the checksum, the exclusive OR of all the bytes before the checksum, and “ZDA” the time-of-day command. The first two characters of the reply message are the talker ID (which is “G” for GPS-equipment), and the last three characters are the sentence formatter or message ID, “ZDA” (time-of-day) in this example.

3.5.2 Receiver Antenna Placement

When mounting the antenna module (pictured in Figure 3-7) it is important to remember that GPS positioning performance will be more optimal when the antenna “patch plane” is level with the local geographic horizon and the antenna has full view of the sky, ensuring direct line-of-sight to all visible satellites. Figure 3-8 shows proper antenna placement [38].

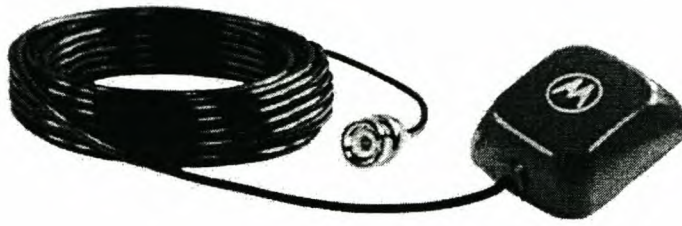


Figure 3-7 GPS receiver antenna [38]

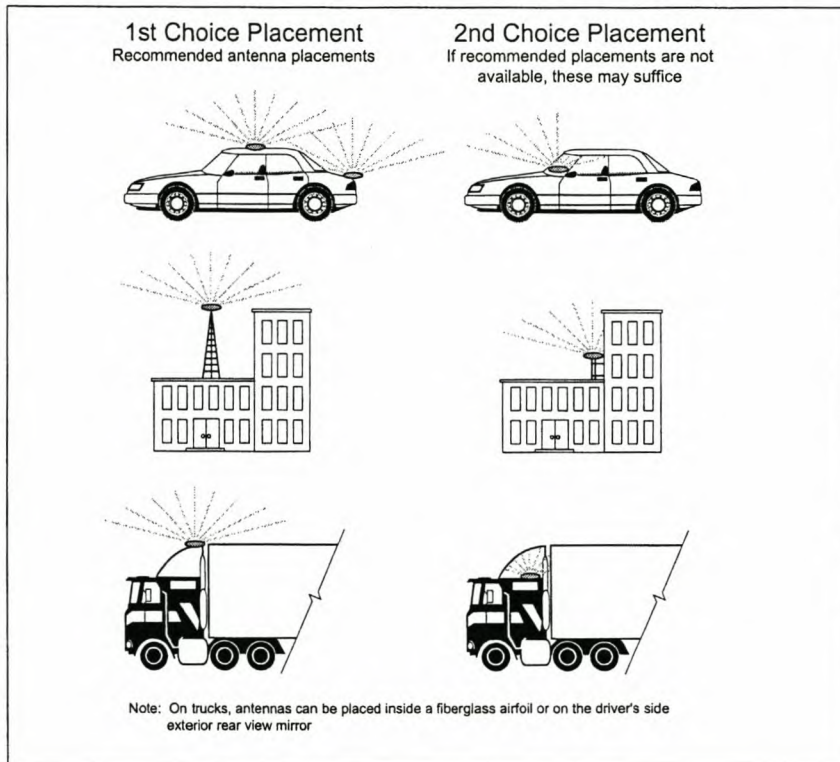


Figure 3-8 GPS receiver antenna placement [38]

3.6 GPS Accuracy

In this project the precise 1PPS pulse from a GPS receiver is used to synchronise a real time clock (RTC). The pulse stability from the GPS receiver is thus very important in this application. This pulse is not only used to synchronise the RTC, it is also used to synchronise the RTC to UTC, which will be the time base for a national phasor measurement or fault location system. In this section, the accuracy of two Motorola GPS receivers (GT+ and UT) versus the Cesium-beam standard will be presented, as well as antenna RF jamming immunity.

3.6.1 Timing Accuracy

Over the years, Motorola GPS receiver output has been compared against various atomic time standards [13]. Figure 3-9 shows a comparison of the GT+ Oncore receiver 1PPS stability to the Cesium beam standard described in section 2.6.2.3. Reference [13] and the GT+ receiver user manual [38] gives the accuracy of the 1 PPS signal to be ± 500 ns.

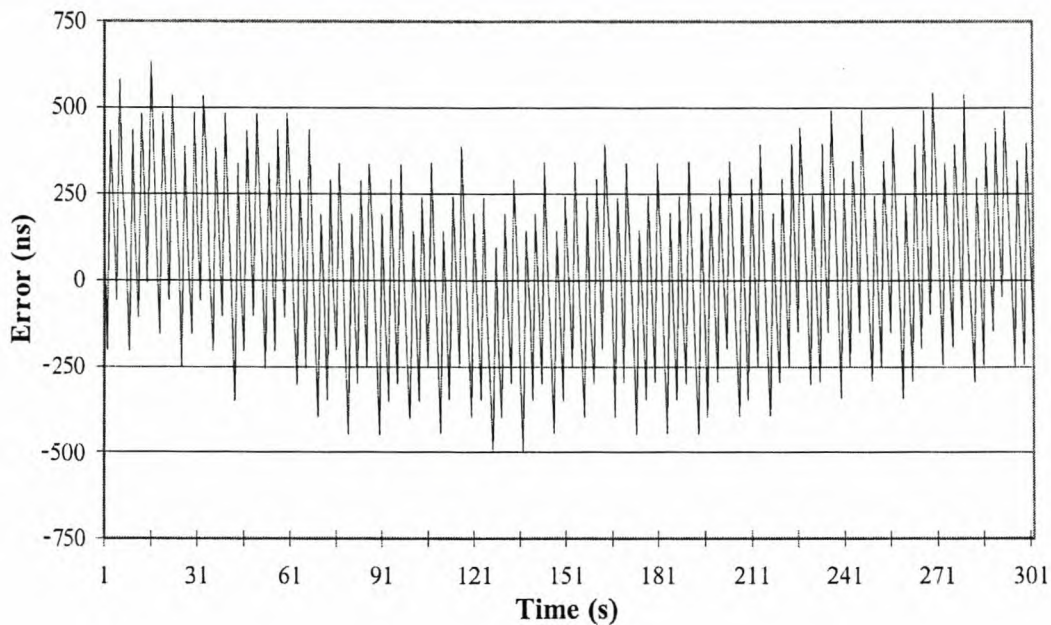


Figure 3-9 GPS receiver stability error compared to the Cesium beam standard [13]

Many precise timing GPS installations require locating the GPS antenna at close range to radiating antennas such as cellular telephone, paging, or other wireless communication systems that operate at frequencies close to L1 and L2. Some of these transmitters may cause the GPS receivers to lose their lock on tracked satellites. This can be very disconcerting to the timing community since a system relying on accurate clock disciplining will have to rely on “clock-coasting” until the satellite signals are reacquired. Long “coasting” times will require more expensive on-board oscillators for the timing electronics in the developed system. Fortunately Motorola Oncore™ GPS receivers are designed with RF Jamming Immunity.

3.6.2 RF Jamming Immunity

Experience [13] has shown that the ability of a GPS receiver to select only the GPS band of information and reject all other signals is an important feature for GPS receivers. Figure 3-10 shows the RF Jamming Immunity characteristics of different Motorola GPS receivers.

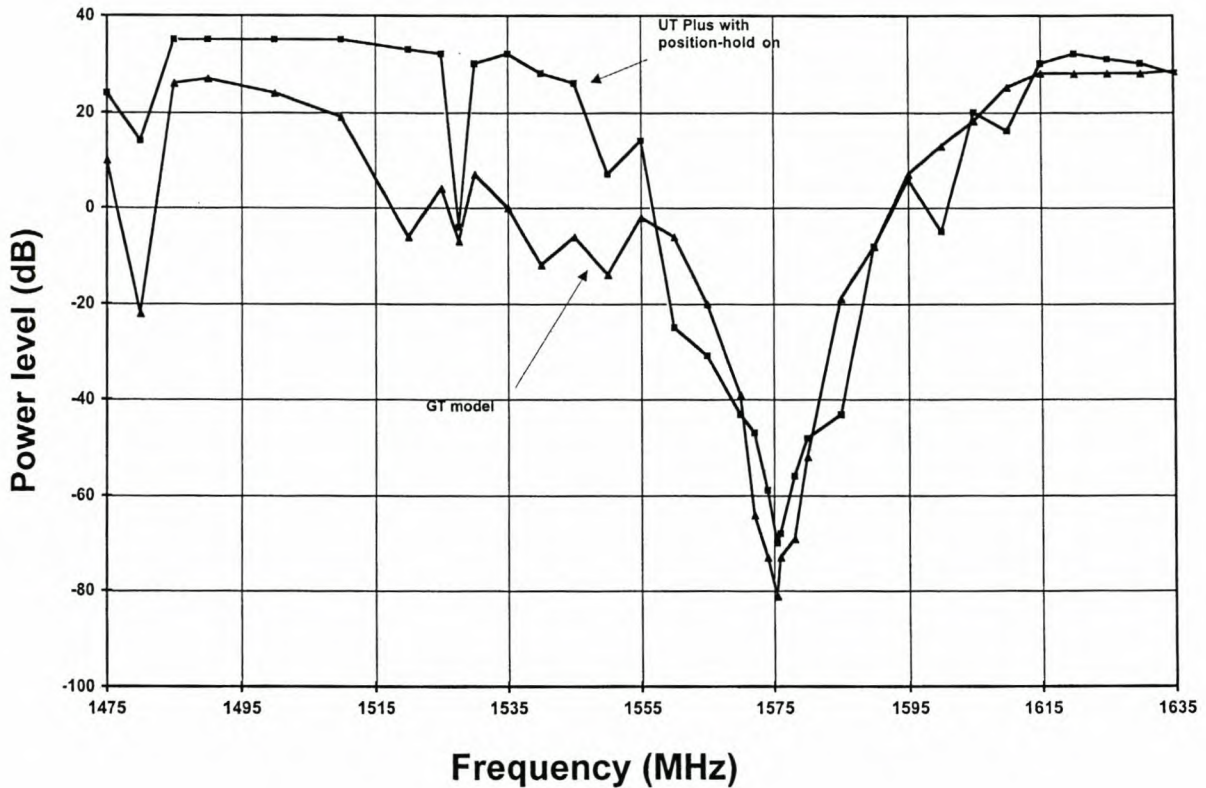


Figure 3-10 GPS receiver RF Jamming Immunity [38]

Figure 3-10 shows that the GPS receiver sufficiently blocks RF signals at frequencies other than its operating frequency, 1575 MHz. It is therefore sufficiently isolated against noise at its operating frequencies.

3.7 Conclusion

Using GPS is a cost effective method for precise timing applications. In this chapter, an overview of GPS operation was given, the communication interface was discussed and some comments on GPS accuracy for timing applications were made. The most useful GPS receiver feature for this project, i.e. the 1 PPS, was also defined and discussed.

Chapter 4

Real Time Clock and Comparator

4.1 Introduction

This chapter describes the on-board Real Time Clock (RTC) and comparator. Figure 4-1 shows a functional block diagram of the Real Time Clock and comparator.

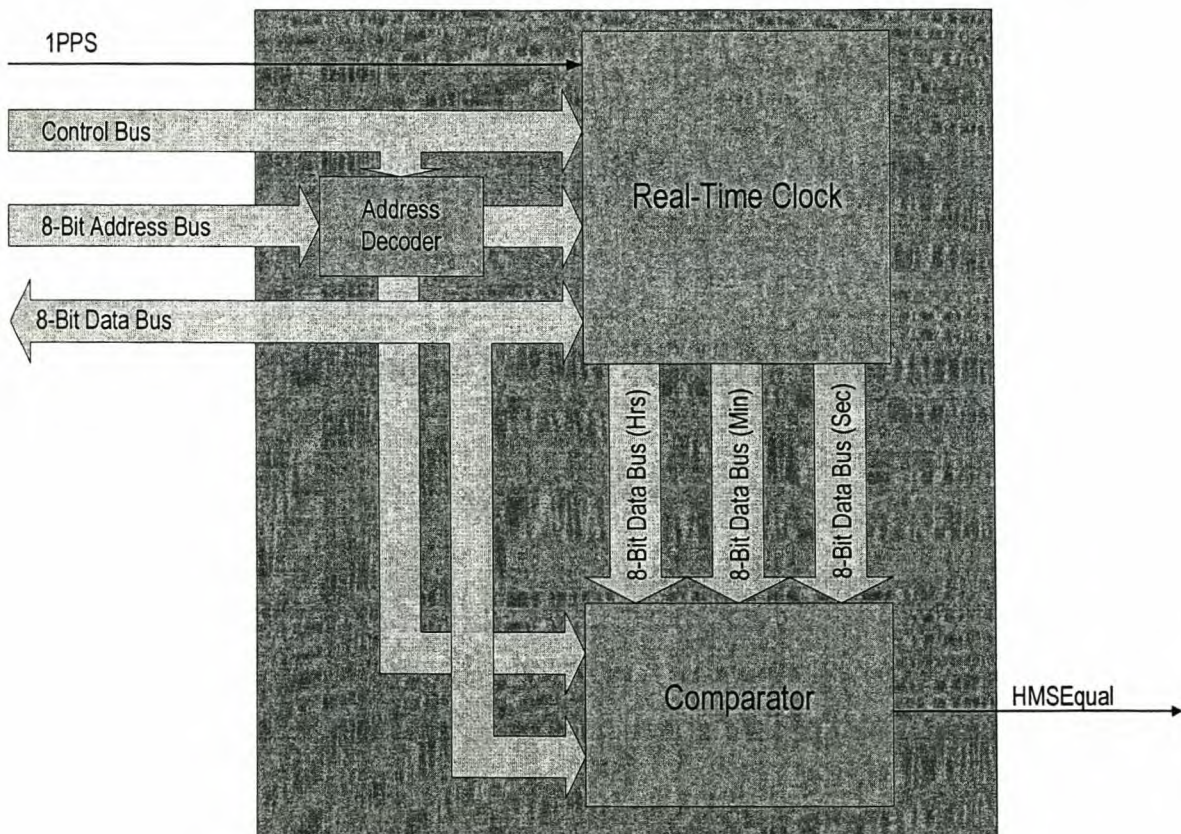


Figure 4-1 Real Time Clock and Comparator block diagram

As an overview, some features are listed here.

- At start-up, the system will read time from the GPS receiver and set the RTC. The GPS takes up to a second to reply on a time-request. When the time is received, the RTC is set.

- After the RTC has been set with GPS time, the time in hours, minutes and seconds is available through the 8-bit data bus at any time, without interrogating the GPS receiver for time information.
- The GPS receiver 1PPS will be the seconds “clock signal” for the RTC.
- Data latches on the RTC output ensures that time information is available instantly on the 8-bit data bus when requested.
- The RTC comparator is connected to the RTC via three 8-bit data buses, one each for hours, minutes and seconds. The RTC comparator is programmed (via the 8-bit data bus) with a certain time (hours, minutes and seconds). The comparator then compares these two times and when the programmed time is equal to the RTC time, HMSEqual is asserted to signal a time match. The comparator detail is discussed in section 4.4.

4.2 Data Flow Operations

Figure 4-2 illustrates data transfer operations in the developed system. This transfer system consists of an 8-bit data and address bus, an address decoder, input/output latches and several different subsystems that need or output data, such as RTC hours output, RTC minutes output, etc.

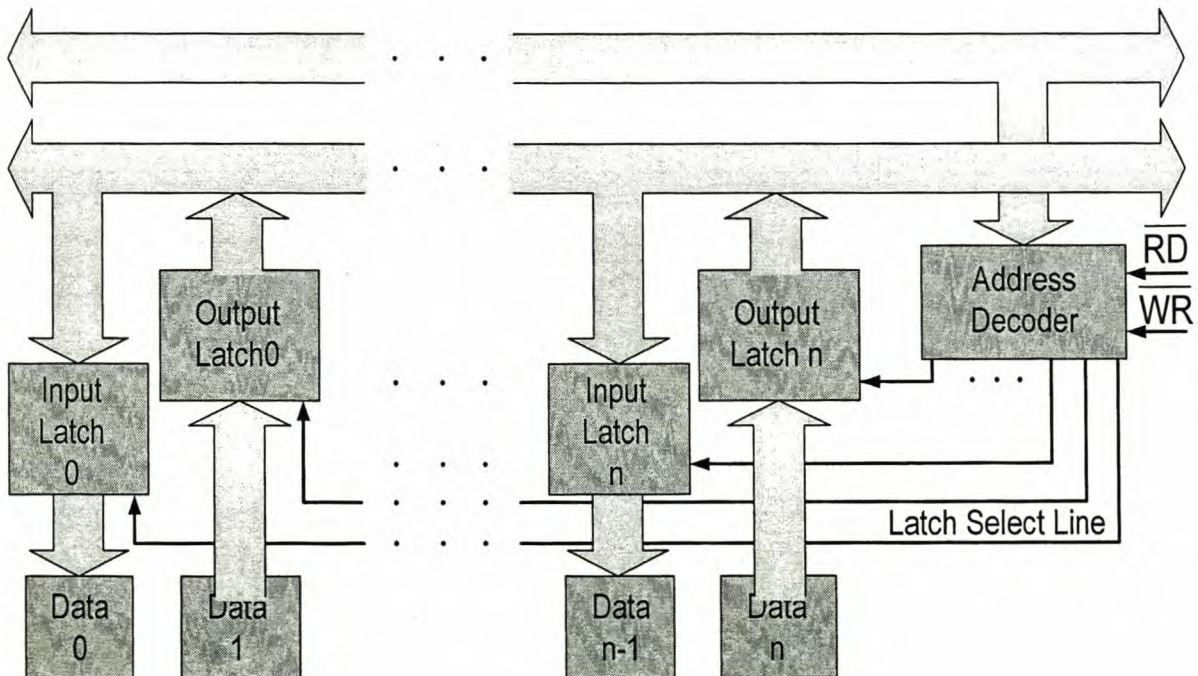


Figure 4-2 Data access structure

4.2.1 Address Decoder

Numerous subsystems, that either read or write data, exist in this system. Therefore many latches need access to the 8-bit data bus. Every latch has a unique address linked to it. When a latch has to be accessed, its 8-bit address is written to the address bus. Depending on whether a read from the data bus or a write to the data bus has to be done, the \overline{RD}^1 or the \overline{WR}^2 signals are asserted. The data can then be read from or written to the data bus. The developed system implements two address decoders. One address decoder services latches in the RTC EPLD, and another decoder services latches in the Microsecond Counter EPLD. These address decoders are identical in implementation. The latch addresses are different for every latch, however. Figure 4-3 shows address decoder control signals and latch select line operation.

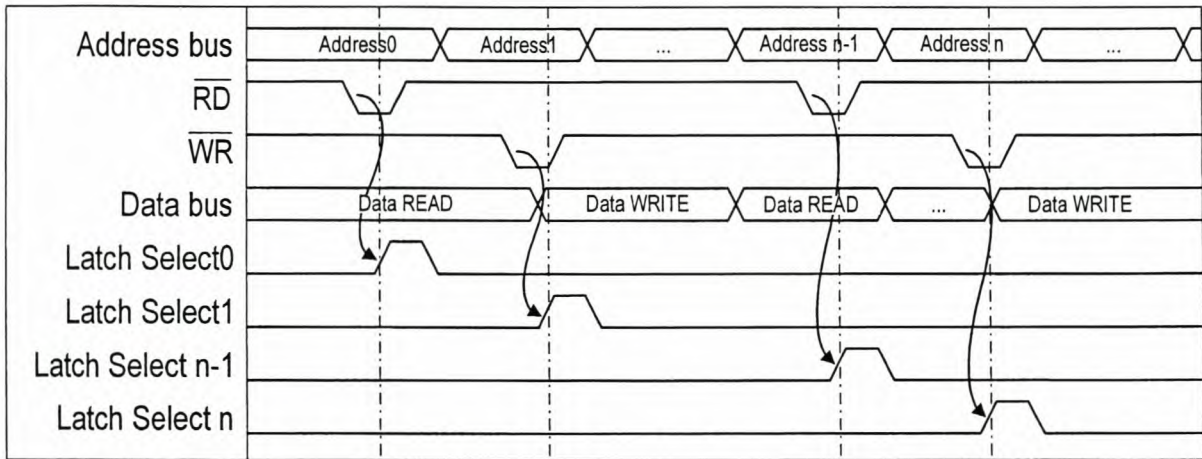


Figure 4-3 Address decoder operation

The VHDL (Verilog Hardware Description Language) program listing and a list of input and output latch addresses can be found in Appendix C. Simulations using the *Altera*® *MAX+PLUS*® software are presented in Appendix D.

4.2.2 Input Latches

An input latch facilitates data transfer from the data bus to a register that uses the data. As described in section 4.2.1, time data is written to the RTC for example, by putting the data on

¹ NOT-Read
² NOT-Write

the data bus and switching on the corresponding data latches that feed data to appropriate registers in the RTC EPLD. Data is thus written from the data bus through the latch to its destination. Figure 4-4 shows a block diagram of a latch. When *Latch Select* is asserted, data is allowed to flow through. Otherwise, data is blocked.

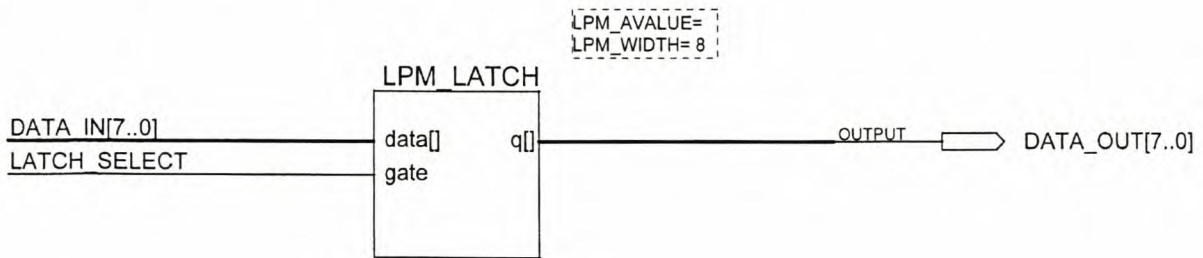


Figure 4-4 Latch block diagram

4.2.3 Output Latches

Output latches used are very similar to input latches, in the sense that they facilitate data flow from data registers to the data bus. The difference is that an output latch needs to change its output to the data bus to high-impedance mode when the latch is not asserted. In that way the data bus is released and bus contention, i.e. different latches driving the data bus at the same time, does not occur. Figure 4-5 shows waveforms for operation of an output latch.

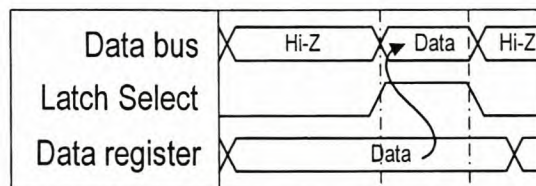


Figure 4-5 Output latch operation

The input- and output latches discussed here are widely used in the design of the system. For this reason they will only be discussed here.

4.3 Real Time Clock

As mentioned previously, the RTC (Real Time Clock) is implemented in order to have time information on-board, so that the GPS receiver does not need to be prompted when time information is needed. Several commercial RTCs exist, for example STMicroelectronics

M41T94, Intersil CDP68HC68T1 and ICM7170. These devices all use crystal oscillators to keep time, and they are fully programmable. Unfortunately there are some shortcomings which make them unsuitable for this application. They do not offer an option of using an external clock signal other than that of the crystal oscillator. Thus, the 1PPS signal from the GPS receiver cannot be used to drive a commercial RTC device. Because of this limitation, a RTC was created and implemented in an *Altera® MAX7128-SLC84-15* EPLD [47].

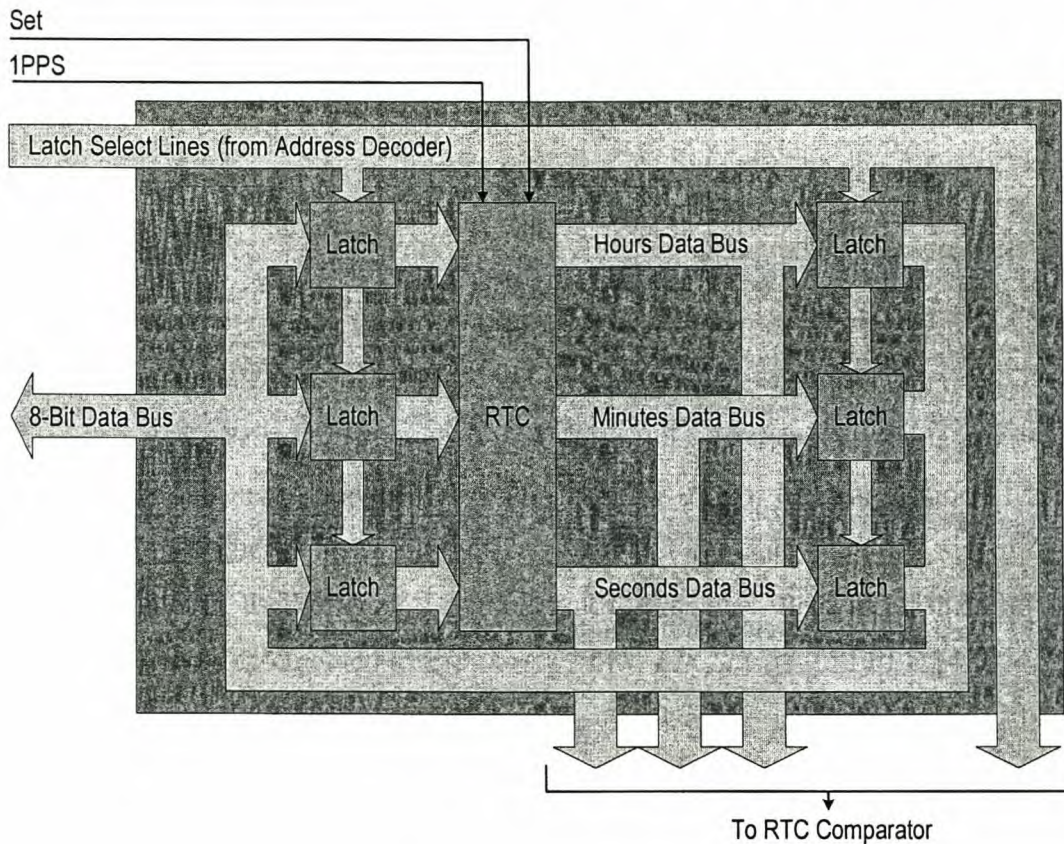


Figure 4-6 RTC block diagram

Figure 4-6 shows an overview of the RTC with latches (as described in section 4.2) and 1PPS clock signal included.

4.3.1 Description

The RTC employs three counters (*LPM_Counter* in *Altera® MAX+PLUS®* software) that serve as hour, minute and second counters. Such a counter is configured as an ‘up’ counter with a maximum value that corresponds with the maximum value that a second, minute and hour can reach. *Carry_Out* is asserted when the counter reaches its maximum value.

Carry_Out serves as clock signal for the next counter stage, which is minutes or hours. The *Data* input specifies the value the counter assumes when *Load* is asserted. In this way the RTC may be programmed with a specified time obtained from the GPS receiver. Figure 4-7 shows a diagram of the counter function used in *Altera® MAX+PLUS®* software.

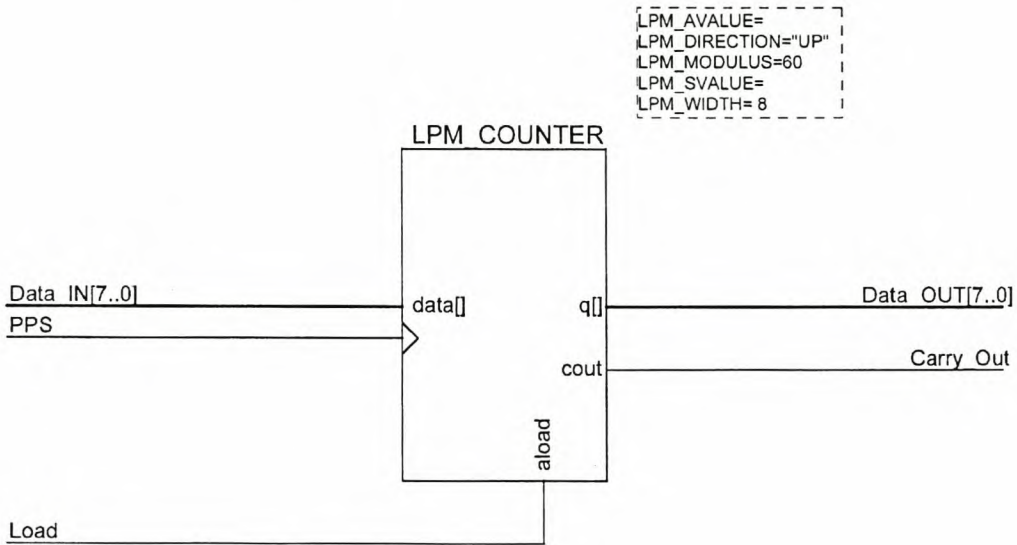


Figure 4-7 LPM_Counter function

Figure 4-8 presents the timing signals of the *LPM_Counter* function. It shows a counter being loaded with a value of 15 and then set by asserting *Load*. It also shows a counter set up to count up to 60, reset and then start at zero again. The *LPM_Counter* function is designed such that *Carry_Out* is asserted at *maximum_count-1* (in this case, 59).

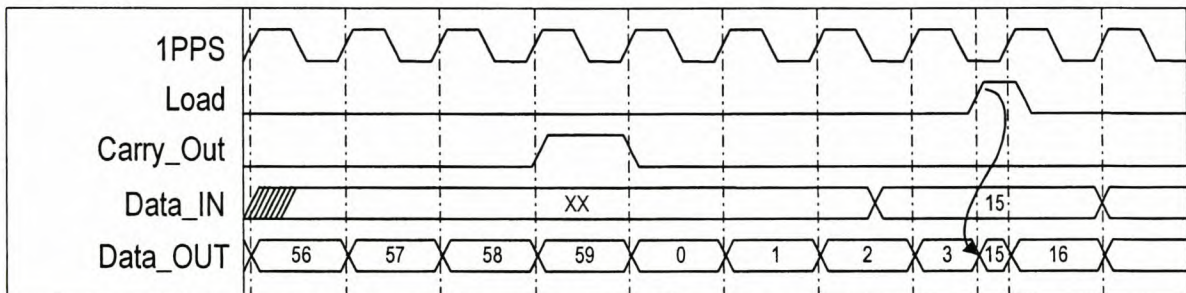


Figure 4-8 LPM_Counter operation

As mentioned previously, the idea is to use *Carry_Out* as a clock pulse for minutes and hours counters, as shown in Figure 4-9.

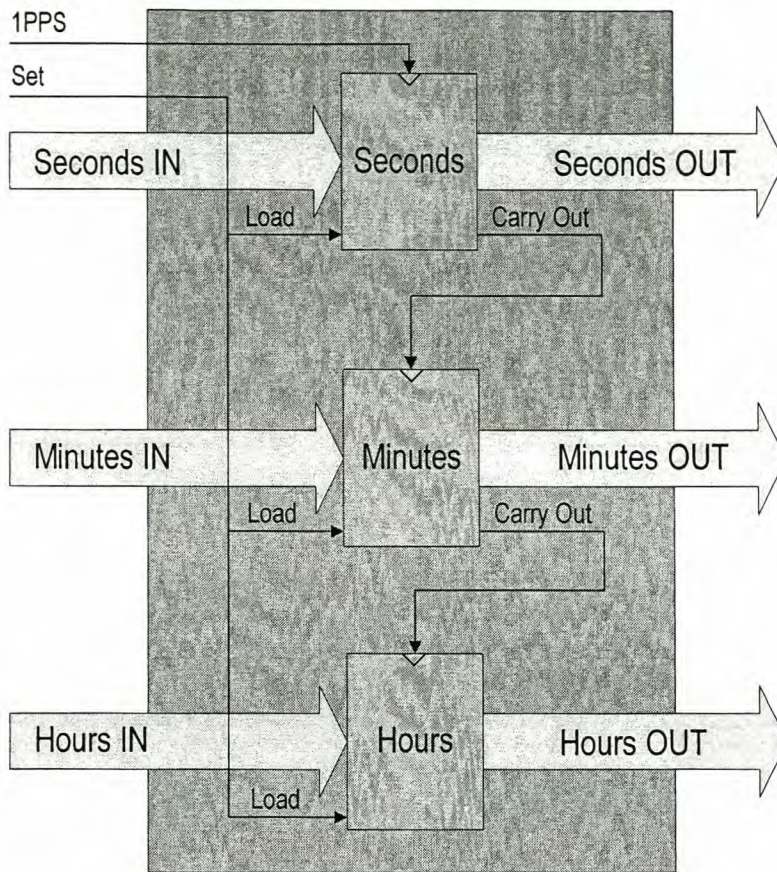


Figure 4-9 Using *Carry_Out* as clock signal for minutes and hours

At this stage a problem arises if *Carry_Out* is to be used as the next stage clock pulse because it goes high at *maximum_count-1*, and not at *maximum_count*, which is desired. It means that the minutes would change count at, in this case, 59, and not zero as it should be. Figure 4-10 shows minutes changing to 0 when seconds changed to 59, which is not the desired operation.

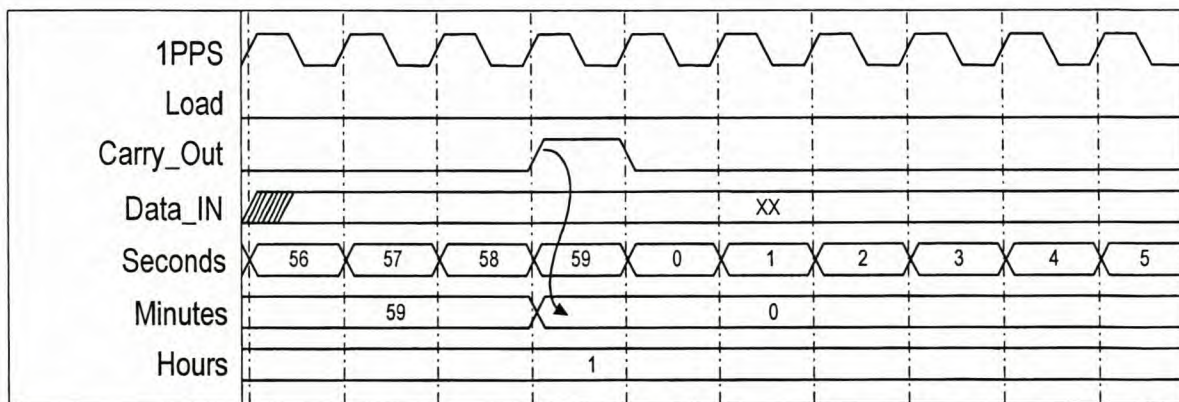


Figure 4-10 *Carry_Out* used as clock signal for next stage counters

This problem is fortunately easily dealt with by using a *D-type* flip-flop [44]. This flip-flop clocks the *Carry_Out* (connected at *D*) signal through on the next clock pulse (connected to the clock input of the flip-flop) it receives.

Table 4-1 *D-type* flip-flop [44]

Clock	D (Input)	Q (Output)
1	0	0
1	1	1
0	X	Last Q

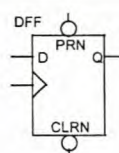


Figure 4-11 is a block diagram showing the implementation of the *LPM_Counter* function in the RTC, including *D-type* flip-flops as discussed previously. This is implemented in an *Altera® MAX7128-SLC84-15 EPLD* [47].

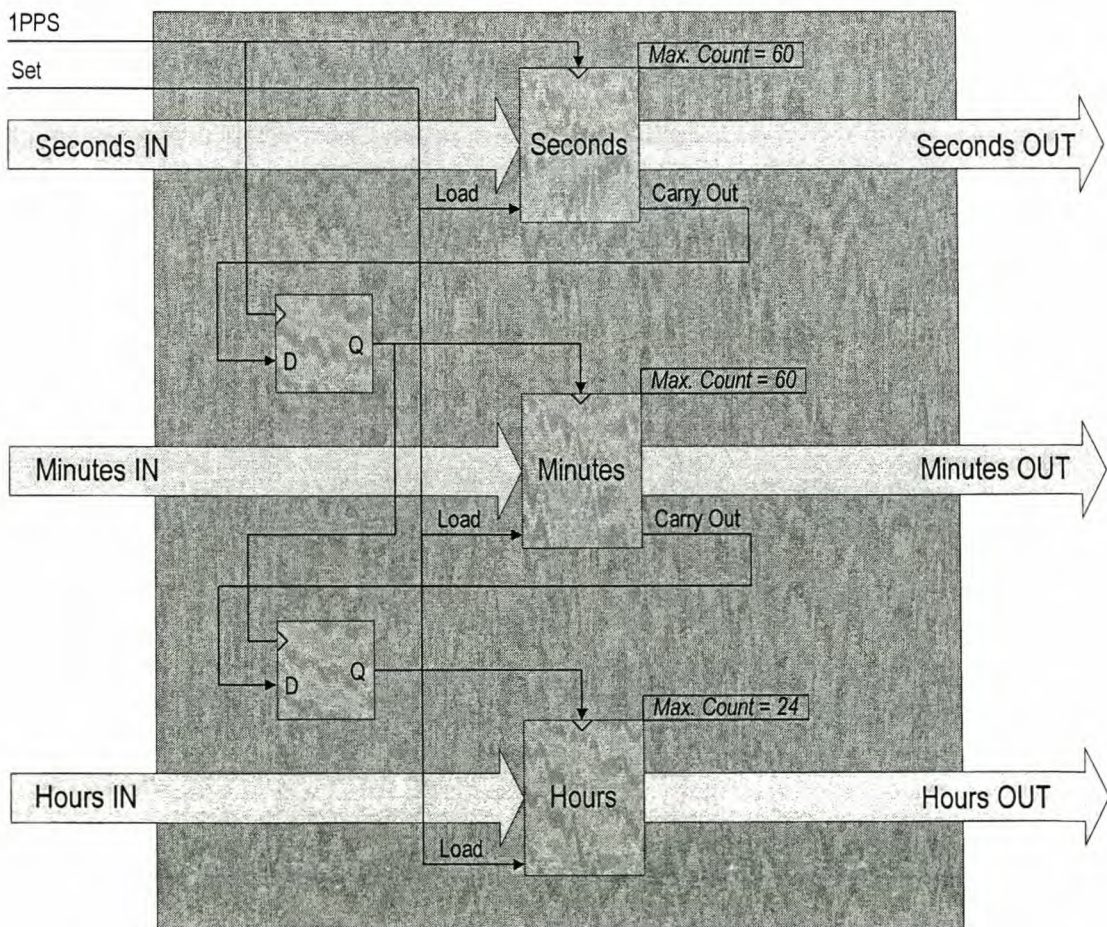


Figure 4-11 RTC block diagram

4.3.2 Simulation Waveforms

The design presented in Figure 4-11 was implemented in *Altera® MAX+PLUS®* software and simulated. Only the most important simulation results are presented here. Complete simulation waveforms and implementation diagrams can be found in Appendix D.

4.3.2.1 Normal RTC Operation

Normal RTC operation occurs when the system is idle, i.e. no data is transferred on the buses. The RTC behaves just like an ordinary clock, with seconds ticking over with low-to-high transitions of 1PPS. These values of hours, minutes and seconds are ready to be read at any time. Figure 4-12 shows normal RTC operation.

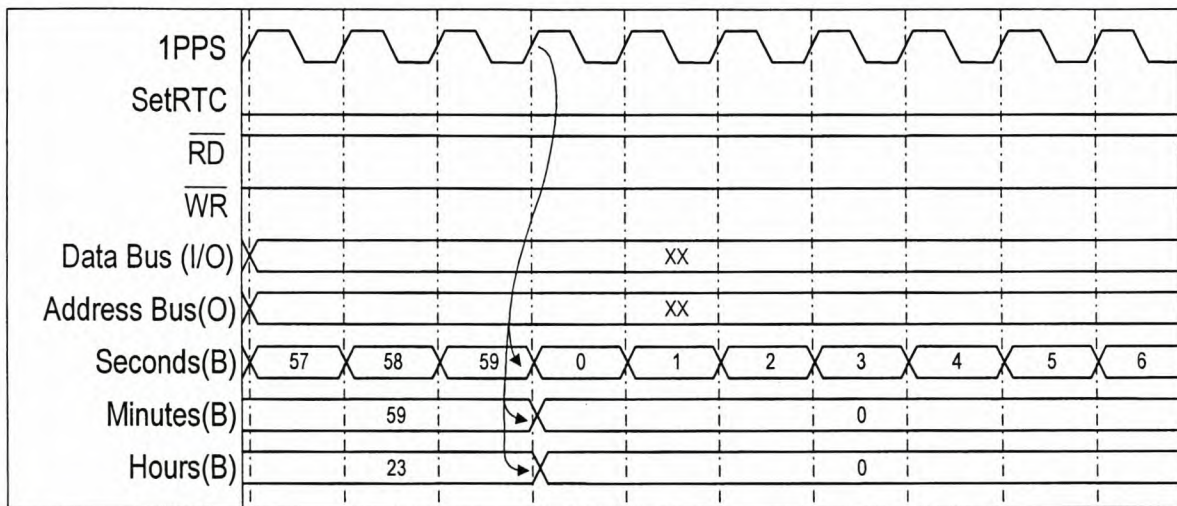


Figure 4-12 Normal RTC operation (midnight turnover)

4.3.2.2 Set RTC Time

At system start-up, or when the user so wishes, UTC (+GMT offset) time is read from the GPS receiver, and the RTC is set accordingly. This procedure was discussed in section 4.2. Timing data is put on the data bus, the addresses of the corresponding registers are put on the address bus, \overline{WR} is asserted and *SetRTC* is pulsed high to write data to the RTC. Figure 4-13 shows this procedure.

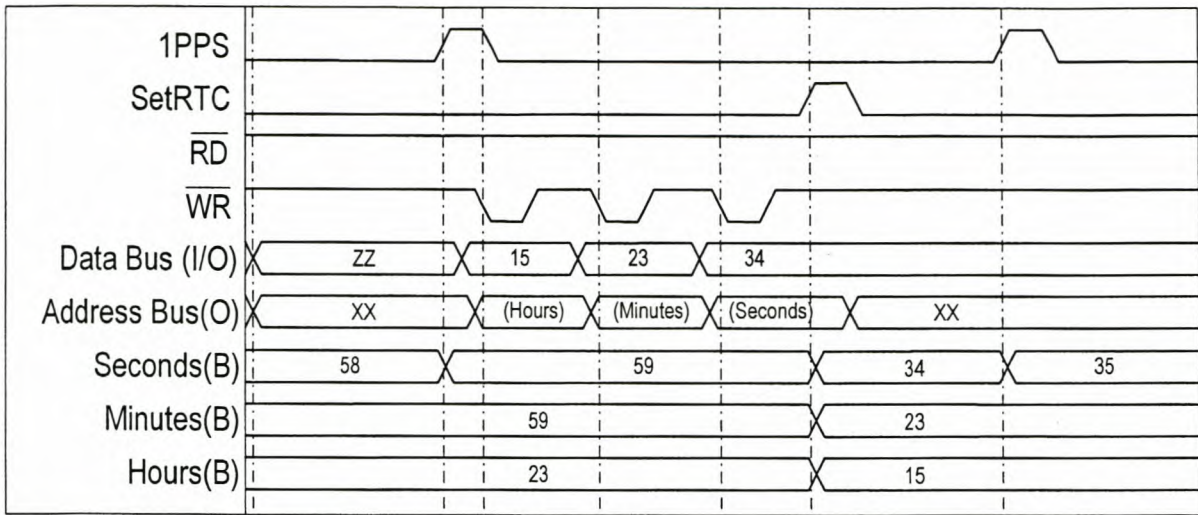


Figure 4-13 Set RTC to 15:23:34

4.3.2.3 Read RTC Time

As discussed previously, time data is always available on request. Figure 4-14 shows buried (B) nodes containing hours, minutes and seconds. When the addresses of these buried nodes are written to the address bus and \overline{RD} is asserted, the latches to these nodes put the relevant data onto the data bus. The data may then be read by the system microcontroller (chapter 6). A complete listing of data register addresses may be found in Appendix C.

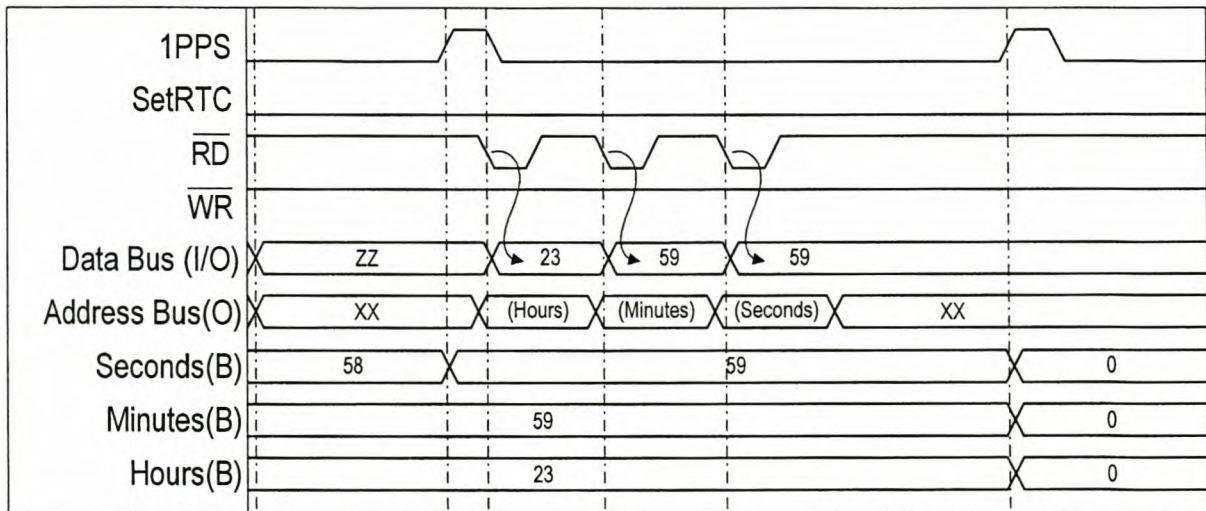


Figure 4-14 Read RTC time (23:59:59)

4.4 Real Time Clock Comparator

The function of the RTC comparator as shown in Figure 4-1 is to compare a pre-programmed time with the current RTC time and when these two times are equal, it should assert *HMSEqual*. This signal is routed to the microsecond counter (chapter 5) and it is used in conjunction with the microsecond count to generate a μ s-wide trigger pulse.

4.4.1 Block Diagram

Figure 4-15 presents a functional block diagram of the RTC comparator. The buried (B) nodes of time (hours, minutes and seconds) of the RTC are connected to the comparator via three 8-bit data buses. The comparator is programmed by writing hours, minutes and seconds via the data bus to the corresponding latches (section 4.2). When the RTC time is equal to the pre-programmed time, *HMSEqual* is asserted.

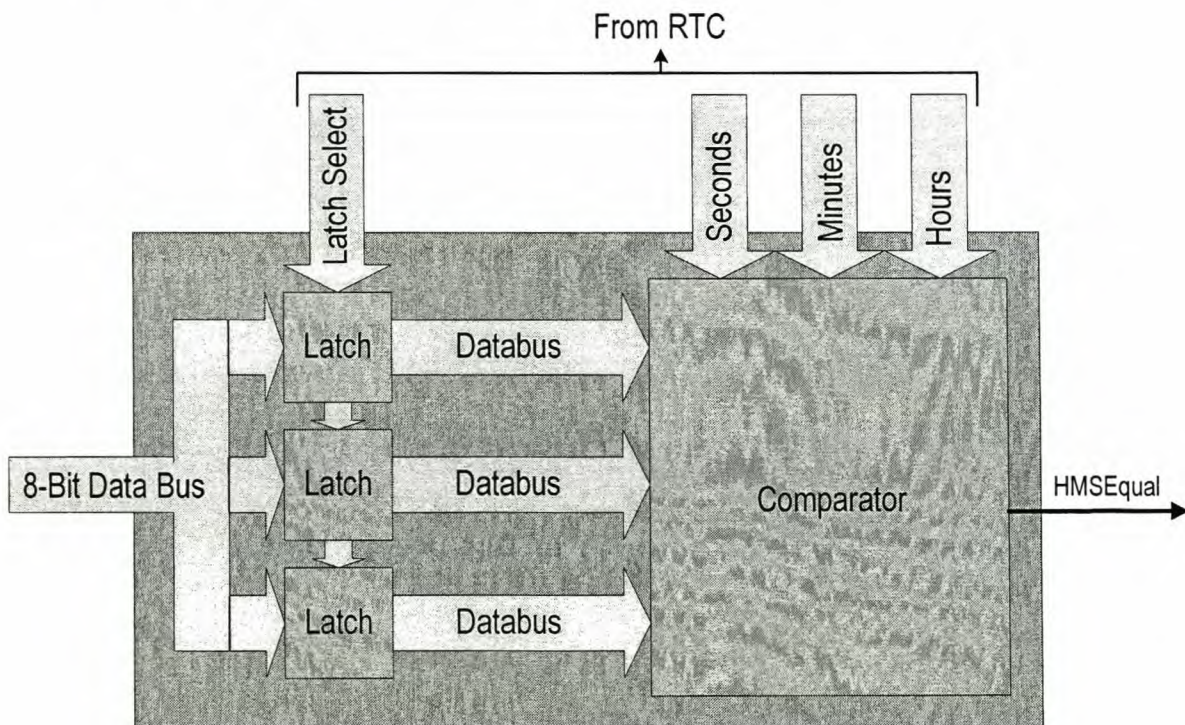


Figure 4-15 RTC Comparator block diagram

4.4.2 Implementation

The comparator is implemented by using the *LPM_Compare* function in *Altera® MAX+PLUS®* software. Figure 4-16 presents this function.

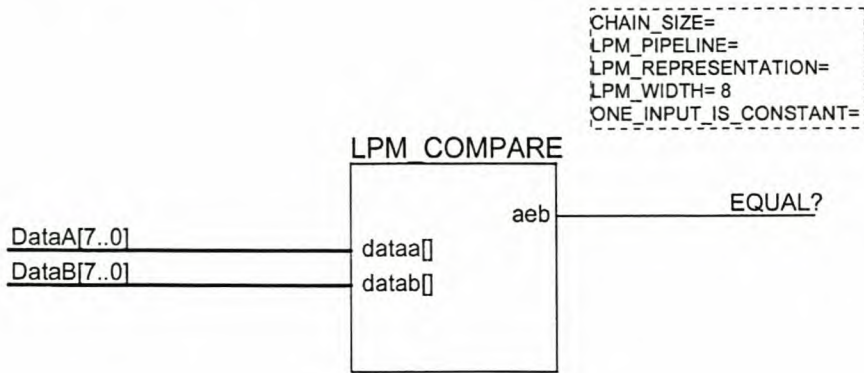


Figure 4-16 LPM_Compare function

aeb is asserted when *DataA* and *DataB* is equal. Three of these functions are used in the comparator, i.e. one each for hours, minutes and seconds. The output of all the *LPM_Compare* functions are connected to an AND-gate to form *HMSEqual*. Figure 4-17 shows the final comparator block diagram.

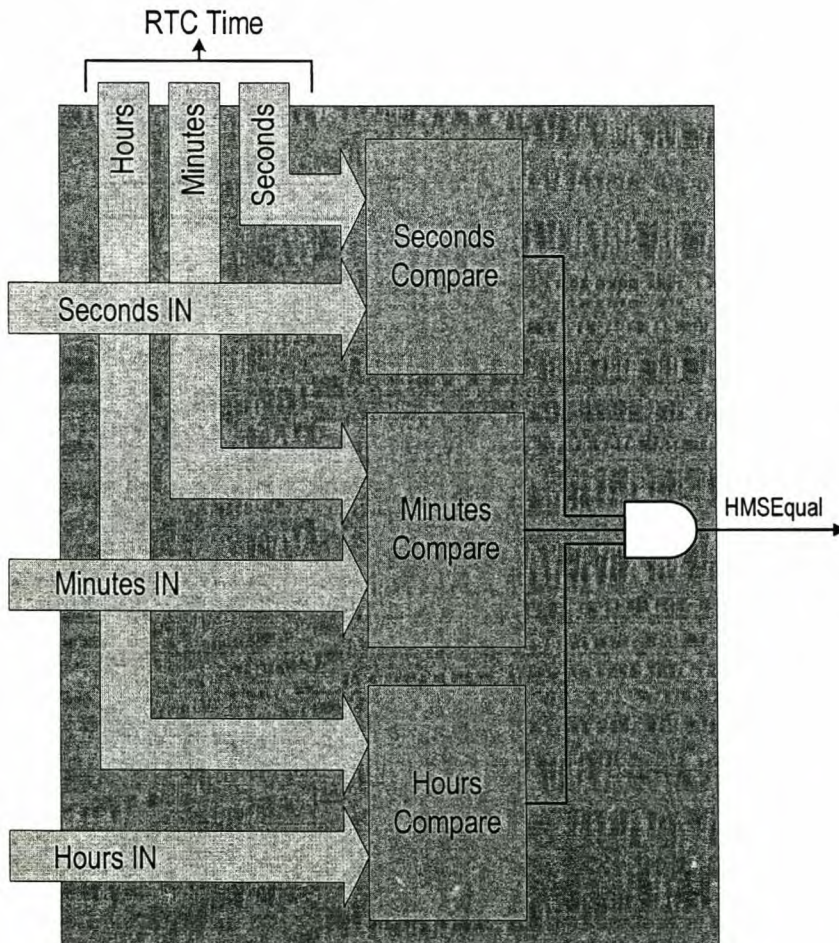


Figure 4-17 RTC Comparator block diagram showing *HMSEqual*

4.4.3 Simulation Waveforms

The comparator was implemented using *Altera® MAX+PLUS®* software. Complete printouts of the simulations are presented in Appendix D. Only the most important results are shown here. Firstly, a time value is programmed into the comparator as shown in Figure 4-18. Secondly, Figure 4-19 shows *HMSEqual* being asserted when RTC time reaches the pre-programmed time.

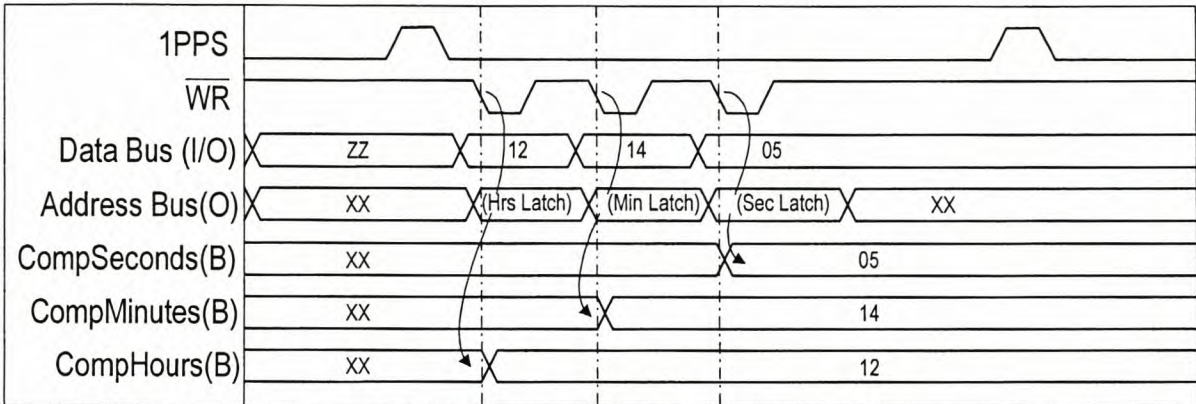


Figure 4-18 Programming the RTC comparator

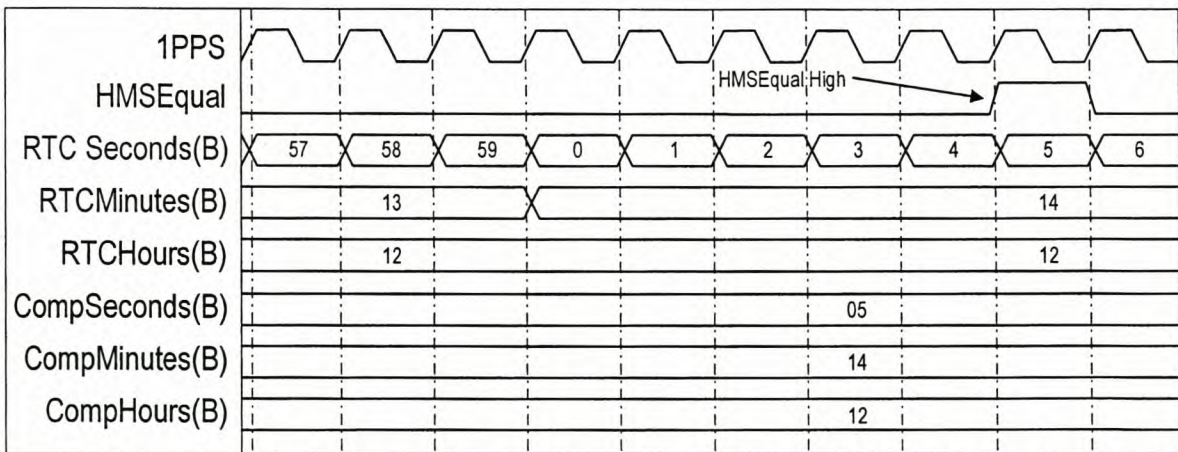


Figure 4-19 *HMSEqual* asserted when RTC and Comparator times match

4.5 Conclusion

This chapter discussed the detail design of the RTC and RTC comparator. The RTC is used to keep time information on-board when it is needed for time stamping purposes. The RTC uses the *1PPS* signal from a GPS receiver as a clock pulse. The RTC comparator is

connected to the RTC and generates a one-second-length pulse (*HMSEqual*) when a pre-programmed time is reached. This feature, in conjunction with the microsecond counter which will be discussed in the next chapter, is used to generate a trigger signal for data acquisition applications. System simulations using *Altera® MAX+PLUS®* software was presented and excellent results were obtained. For full implementation and simulation details, consult Appendix D.

Chapter 5

Microsecond Counter and Comparator

5.1 Introduction

As mentioned in chapters 1 and 2, the aim of the system is to supply a timestamp and be able to generate a trigger signal for data acquisition applications, all to an accuracy of 1 μ s. The purpose of the RTC is to keep time accurate to one second, and this time could be obtained from the GPS receiver at system startup. The microsecond counter, however, has to be implemented on-board.

The design criteria of the Microsecond Counter can be summarised as follows:

- To achieve an on-board timing accuracy of 1 μ s, an accurate binary counter is needed. The counter has to be used in conjunction with the on-board RTC, and must be able to count up to 1 000 000 (microseconds in a second). To implement such a counter with enough precision, and to read its value through an 8-bit data bus, the counter value must consist of three bytes. Thus, a counter with maximum value of 2^{24} is used, although a value of 2^{20} would be sufficient (but not as easily read). As with the RTC, the counter is implemented using EPLD [47] technology.
- The counter can provide the microsecond count when a trigger signal, Trigger IN, is received. This signal opens the output latch of the counter so the value may be read. A microcontroller interrupt signal is generated to alert the system microcontroller that a trigger has been received, and that data must now be downloaded and stored.
- The counter clock (1MHz) is synchronised by a highly accurate (± 500 ns) pulse from the GPS receiver, the 1PPS (section 3.4.3).
- To generate a trigger signal on a microsecond scale, another comparator is implemented. This comparator compares the actual microsecond count with a pre-programmed value and when HMSEqual (section 4.4) is asserted and the microsecond count corresponds with the pre-programmed value, a trigger signal is generated. Details on this comparator can be found in section 5.4.

5.2 Block Diagram

Figure 5-1 shows a functional block diagram of the counter system and comparator. The address decoder is identical to that in section 4.2.1, except for a few differences. Apart from data latch select lines, this address decoder drives indicator LEDs and control signals of the FIFO memory. The external read signal *RDVME* is also connected here. External hardware reads FIFO data by asserting this line. These components will be discussed later in this chapter. The comparator is presented in section 5.4 and is programmed in exactly the same manner as in 4.4. When *HMSEqual* is asserted and the microsecond count is equal to a pre-programmed value, a microsecond-long pulse *Trigger OUT* is generated. This signal is used in data acquisition applications.

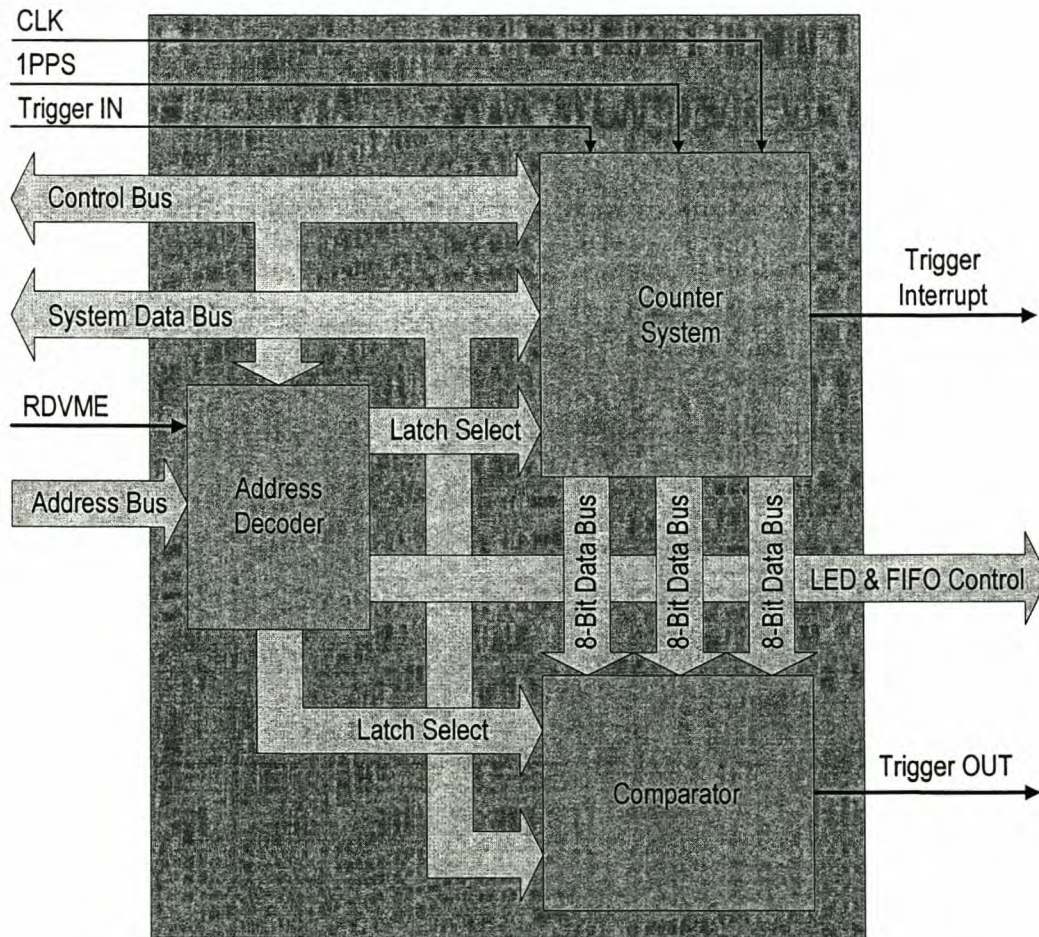


Figure 5-1 Microsecond Counter System and Comparator overview

5.3 Counter System Block

The counter system consists of the following components:

- *The programmable clock signal-prescaler:*
This component scales the clock signal obtained from the system hardware crystal oscillator package to a 1MHz signal, which serves as a clock signal for the 24-bit counter. See section 5.3.3.
- *1PPS conditioning:*
1PPS conditioning ensures that the counter is only reset on the rising edge of *1PPS*. *LPM_Counter* functions (section 4.3.1) are used in the implementation of the Microsecond Counter. *1PPS* resets the *LPM_Counter* functions. *1PPS* signal length (200ms) is more than one clock pulse width (1 μ s) and that would keep the counters reset for 200ms which this is not desirable, as the counter needs to keep on counting microseconds even though *1PPS* is still in high-state. See section 5.3.2.
- *The 24-bit counter:*
This component counts every clock pulse of the 1MHz signal mentioned above. On top of every second (signalled by *1PPS*) this counter is reset. If sufficient accuracy is obtained, the maximum count of this counter should reach 999 999 before being reset by *1PPS*. This count value is read as three bytes (a 20-bit counter would be sufficient, but 20 bits are not easily divided into bytes). See section 5.3.4.
- *Trigger signal conditioning:*
Trigger signal conditioning ensures that if *Trigger IN* is longer than one microsecond, the value of only the microsecond count at the rise of *Trigger IN* is returned. When *Trigger IN* is received, an interrupt signal for the system microcontroller must be generated. This interrupt signal should stay high until the system microcontroller has finished reading and storing time information. No interrupts should be generated until this task is finished. See section 5.3.5.
- *Counter Output latches:*
The output latches are selected via the address decoder, ensuring that every byte of the three-byte counter value can be output sequentially on the system data bus. See section 5.3.6.

Figure 5-2 shows the above components in block diagram form.

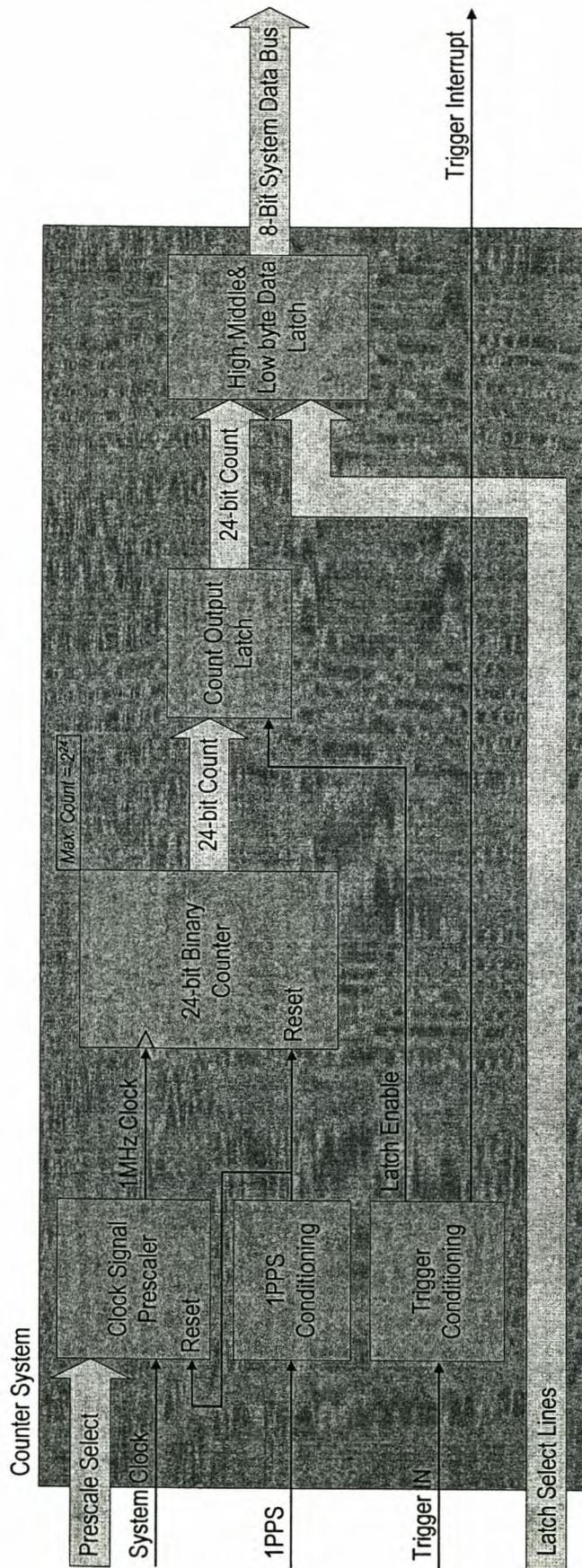


Figure 5-2 Counter System block diagram

5.3.1 System Clock

Before any more design details are discussed, a few comments on the system clock are required. A 20MHz crystal oscillator package is used as a clock signal for the EPLDs. This oscillator is a package that generates a clock pulse at the rated frequency when a 5V supply voltage is applied. Chapter 2 (section 2.6.2) concluded that an oven-controlled oscillator package is the optimum oscillator for this design. In the prototype development of this project, a normal 100PPM oscillator package was used, because oven-controlled packages are extremely expensive and have to be manufactured to user specifications and ordered in bulk quantities.

5.3.2 1PPS Signal Conditioning

LPM_Counter functions are used in this counter design. This function was discussed in section 4.3.1. This function is a synchronous function, which means that every operation performed must be synchronised with its input clock signal. *1PPS* is used to reset the microsecond counter as well as the counter prescaler. As mentioned, *1PPS* is about 200 ms long (section 3.4.3) which means that the counter will be reset for the entire duration of the *1PPS* signal – which would be an error, and not desirable. In order to reset the *LPM_Counter* functions exactly on the rising edge of *1PPS*, and only for one system clock pulse length, some signal conditioning on *1PPS* must be done. A *D-type* flip-flop (Table 4-1) is used to create a signal that is essentially *1PPS*, but synchronised with the flip-flop input clock signal. The *D-type* flip-flop clocks *1PPS* through on its own clock signal. A NOT-gate is used to invert *1PPS* and finally a NOR-gate is used to create a short reset signal for the counters. Figure 5-3 shows the circuit used for conditioning *1PPS*.

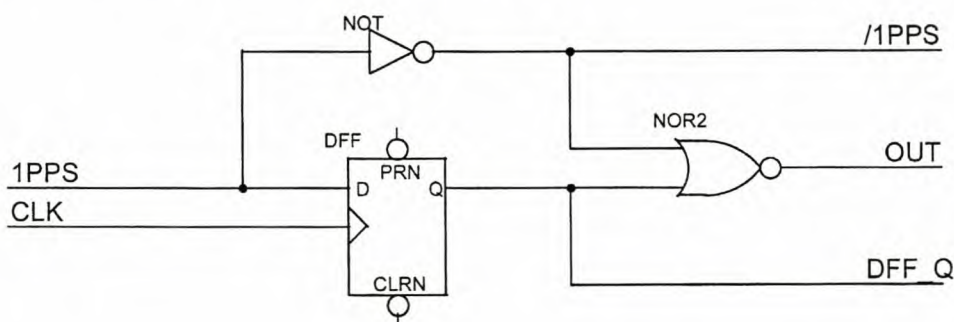


Figure 5-3 1PPS conditioning circuit

Signals *1PPS* and *DFF_Q* are NOR'd to obtain signal *OUT*, which is used to reset the *LPM_Counter* functions of the Clock Prescaler (section 5.3.3) and the 24-bit counter (section 5.3.4). *1PPS* conditioning guarantees that signal *OUT* is not longer than two clock pulse widths, as can be seen in the simulation waveform in Figure 5-4.

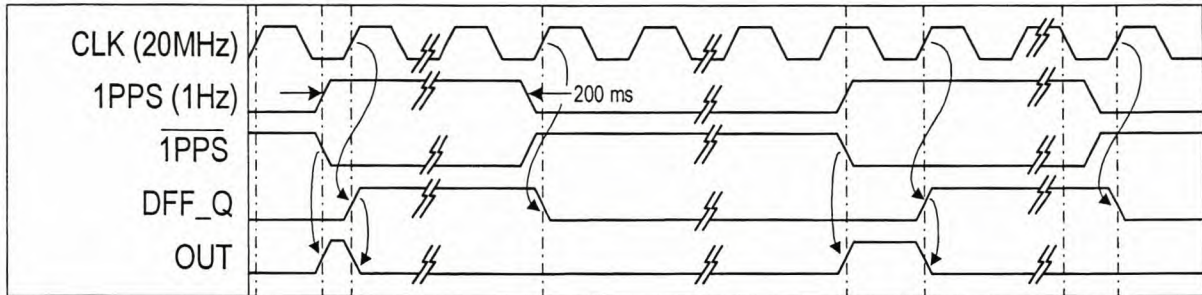


Figure 5-4 1PPS conditioning simulation

Complete *Altera® MAX+PLUS®* waveform simulations can be found in Appendix E.

5.3.3 Programmable Clock Prescaler

The 24-bit counter needs a clock signal of 1MHz in order to increment the counter value for every microsecond that passes. The on-board crystal oscillator package is, however, not 1MHz. This is because a faster clock speed is needed as a general clock for system operations other than the Microsecond Counter. Thus there exists a need for a clock divider, or *clock prescaler*, to divide the available clock signal so that a 1MHz clock signal can be obtained. A very simple state machine is used for this [44]. The machine needs to change its current state from 0 to 1, or vice versa, every *x* amount of clock pulses. For example, if a 6MHz clock is available, and a 1MHz clock signal is needed, the state machine needs to change states after 3 pulses of the 6MHz clock. Figure 5-5 presents this example, and shows clearly that for every 6 clocks, the state machine produces a state transition, which translates to a 1MHz clock signal.

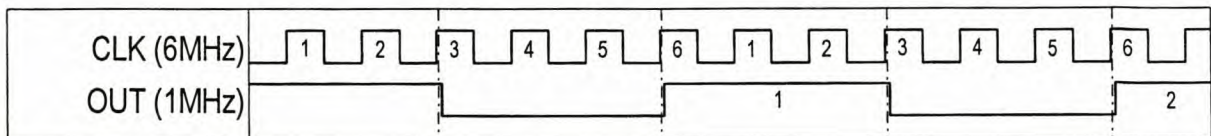


Figure 5-5 State machine example

The clock divider acts according to equation (5-1), where A is the number of clock cycles in the high state, and B is the number of clock cycles in the low state.

$$\text{Output Clock} = \text{System Clock} \frac{1}{A + B} \tag{5-1}$$

It should be noted that $A = B$ because a clock signal with 50% duty cycle is needed. A is therefore the clock divider or *prescaler* value, which assumes the value of $SWDATA+1$, for reasons that will be discussed later. The state diagram for the state machine is shown in Figure 5-6. The state machine will now be implemented.

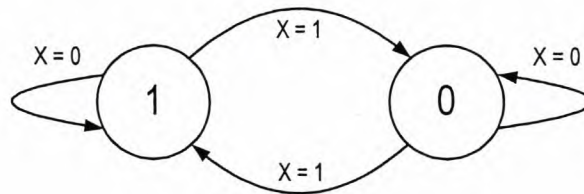


Figure 5-6 State machine diagram

The above diagram shows a system that changes states only when $X = 1$, and it stays in the same state when $X = 0$. If X changes state every clock cycle, the system will spend 2 cycles in state 1 (high), and two cycles in state 0 (low). If, for instance, X alternates state every two clock cycles, the system will spend 4 cycles in each state.

When a XOR-gate is examined, it is noticed that if one input of the gate stays constant, and the other input toggles between 0 and 1, the output also toggles between 0 and 1, which is the desired operation. The state machine used here is an *Altera® MAX+PLUS®* function *LPM_DFF*, which is a *D-type* flip-flop. The flip-flop clocks *data* through with every 0 to 1 transition of *CLK*. A XOR-gate truth table is presented in Table 5-1.

Table 5-1 XOR-gate truth table

X	CLK	XOR(X,CLK)
0	0	0
0	1	1
1	0	1
1	1	0

The state machine and XOR-gate is shown in Figure 5-7.

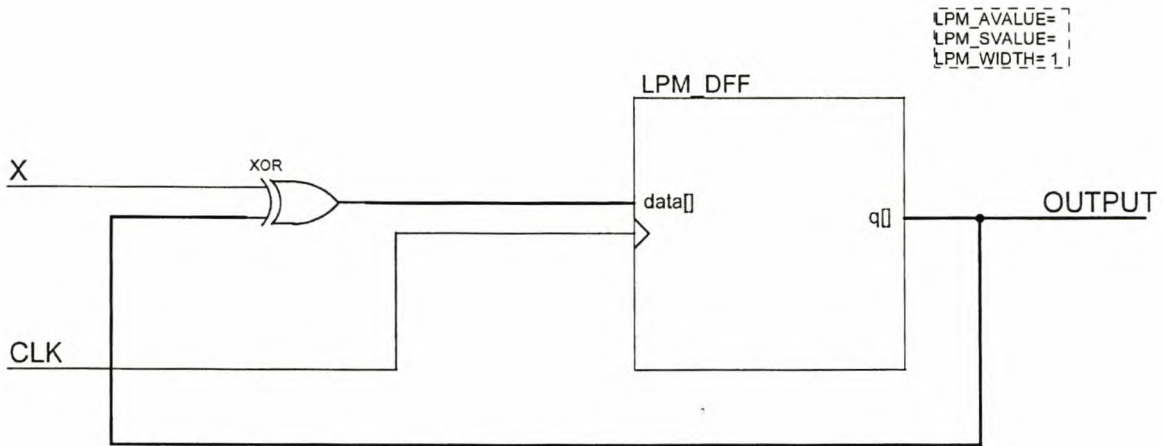


Figure 5-7 State machine circuit

All that remains to be done is to get *X* to change state when the desired amount of clock cycles have passed, for example, in Figure 5-5, *X* has to change state every 3 clock cycles. A *LPM_Counter* is used to count every clock pulse and it asserts a *Carry_Out* signal when a certain amount of clock pulses have passed. This counter counts down from a value programmed via a DIP-switch and *Carry_Out* is asserted when zero is reached. *Carry_Out* is fed back to *SLoad*, which loads the counter. Figure 5-8 presents the complete Counter Prescaler circuit. Figure 5-8 shows the 1PPS reset pulse (section 5.3.2) clearing the *LPM_Counter* and *LPM_DFF*. It also shows the *LPM_Counter* used to count down from a value defined by *SWDATA* with every clock pulse, when zero is reached, the *LPM_Counter* is once again loaded with *SWDATA*. It should also be noted that the clock divide value is equal to $SWDATA+1$ because counting stops at zero, the count on which *Carry_Out* is generated. In this prototype design, a 20MHz clock was used – and to get a 1MHz clock signal (*1MHzClk*), the clock divider value must be 10, that is, *SWDATA* must be equal to 9.

At this point it is important to be certain that *1MHzClk* indeed has a frequency of 1MHz. Although it is an internal signal, *1MHzClk* was routed to an output pin, and captured with a storage oscilloscope. The waveform obtained has a period of $1\mu\text{s}$, which translates to a frequency of 1MHz. Figure 5-11 presents this waveform.

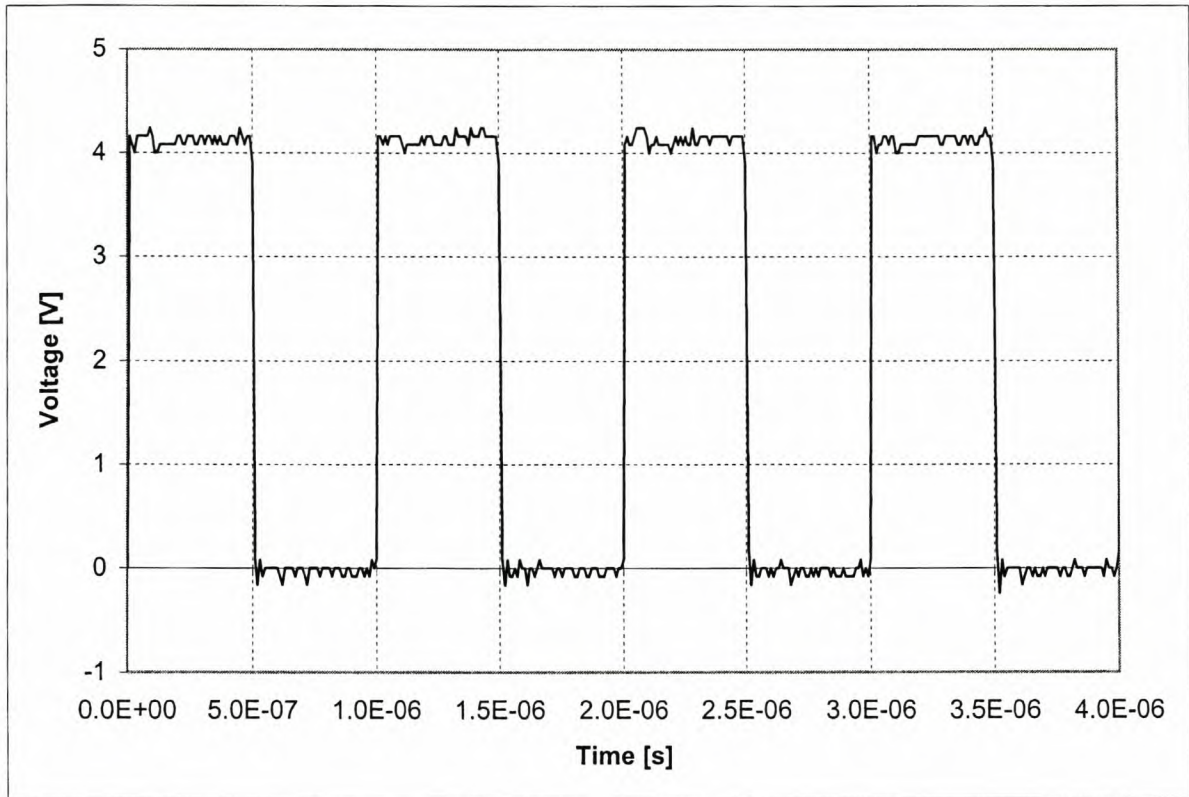


Figure 5-11 *1MHzClk* oscilloscope measurement

5.3.4 24-bit Counter

After the signal conditioning and clock prescaling has been done, the microsecond value needs to be obtained. A 1MHz clock signal is now available and another *LPM_Counter* is implemented. As mentioned, this counter has to count up to a minimum value of 1 000 000, which would be a 20-bit value. The system data bus is 8-bits wide, and therefore a 24-bit counter is used so that the microsecond count can be easily read as three bytes. The *LPM_Counter* function was discussed in 4.3.1. Figure 5-12 shows the configuration of *LPM_Counter* for the 24-bit counter application.

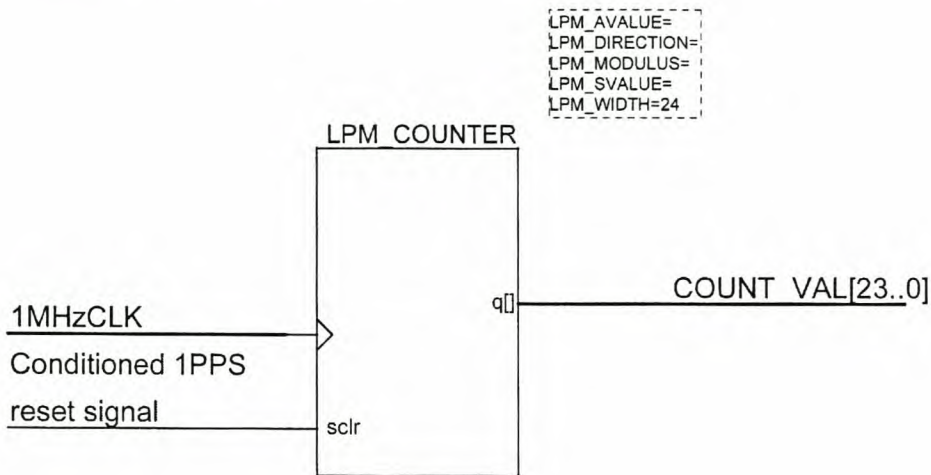


Figure 5-12 24-bit Microsecond Counter

Before it is discussed how the microsecond value will be placed on the 8-bit data bus, the handling of *Trigger IN* has to be discussed, as this will have an effect on how the microsecond output latch will function, as shown in Figure 5-2.

5.3.5 Trigger Signal Conditioning

When the system receives an incoming trigger signal, the following must be kept in mind:

- *Trigger IN* is a signal that will be received from external data acquisition hardware, so it is necessary to condition *Trigger IN* signal length. This conditioned trigger signal is called *SyncTrig*. The signal length is crucial in the sense that if the signal is wider than $1\mu\text{s}$, the data latch providing the microsecond count to the 8-bit system data bus must present the microsecond count corresponding to the rising edge of *Trigger IN*, not of the count corresponding to the falling edge.
- When *Trigger IN* is received, the system microcontroller has to be notified that data transfer and storage may begin. For this purpose, a signal *IntRequest* is created. This signal needs to be long enough so that the microcontroller does not miss the interrupt signal. The system microcontroller is discussed in chapter 6.
- The system must not accept *Trigger IN* signals that appear when the system is busy with data transfer and storage. When the microcontroller receives a trigger interrupt, a signal */Ready* is asserted and that ensures that no triggers will be accepted during that time.

5.3.5.1 Signal *SyncTrig*

As mentioned earlier, it is important that the microsecond count corresponding to the rising edge of *Trigger IN* is obtained. If *Trigger IN* is received in the ‘middle’ of a microsecond and it stays high over the transition from one microsecond to another, some difficulties arise. This problem is overcome by assuring that *Trigger IN* is conditioned in such a way that it is not longer than a system clock pulse, i.e. 62,5ns. This process was implemented in *Altera® MAX+PLUS® VHDL*. Figure 5-13 shows a flow diagram of program *TrigProcess*. The VHDL program can be found in Appendix E.

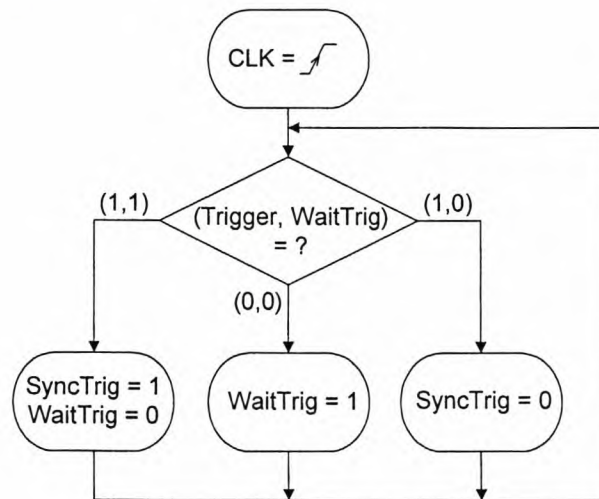


Figure 5-13 *TrigProcess* program flow diagram

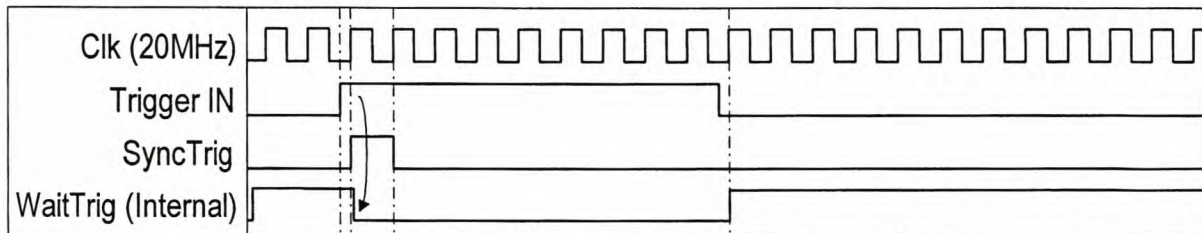


Figure 5-14 *TrigProcess* operation simulation

5.3.5.2 Signal *IntRequest*

The same difficulties as with *SyncTrig* arise when signal *IntRequest* is created. As will be discussed in chapter 6, signal *IntRequest* needs to be a certain length in order to be a sufficient microcontroller interrupt signal. This is done as follows: signal *SyncTrig* is connected to a monostable multivibrator. *IntRequest* (output of the monostable multivibrator) goes high until

the multivibrator is reset. The reset signal is generated by the *Carry_Out* signal of an *Altera® MAX+PLUS® LPM_Counter* function. *SyncTrig* loads the counter with a binary value of 1111 and the counter thus counts 15 clock pulses and generates a *Carry_Out* signal, which resets the monostable multivibrator. In this way a sufficient microcontroller interrupt request signal, which is 15 clock pulses wide, is generated. Figure 5-15 shows a *D-type* flip-flop configured as a monostable multivibrator and Figure 5-16 presents the circuit for generating signal *IntRequest*, including the block *TrigProcess* discussed in the previous section.

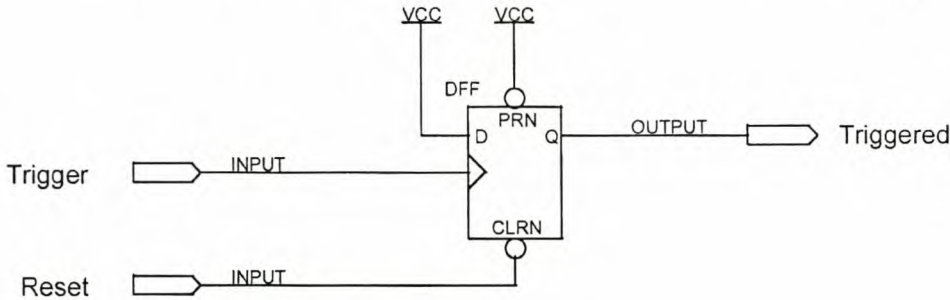


Figure 5-15 Monostable Multivibrator circuit

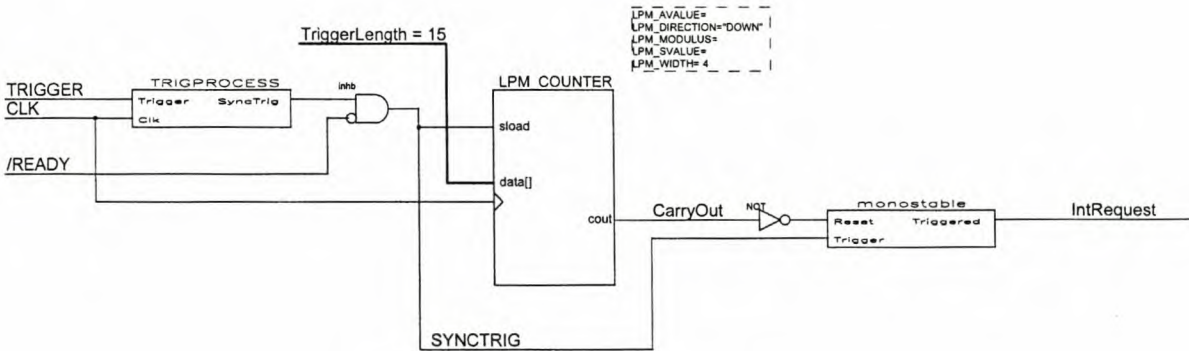


Figure 5-16 *IntRequest* generation circuit

Figure 5-17 is a simulation of this circuit and shows *Trigger IN* as a wide signal. *SyncTrig* is a one clock pulse length trigger signal – this signal will be connected to the microsecond count output latch that will make the microsecond count available on the 8-bit data bus (section 5.3.6). *SyncTrig* generates a 15 clock-cycle-length signal *IntRequest* to notify the system microcontroller of the received trigger. Once the microcontroller receives the interrupt request, signal */Ready* is asserted. If another trigger signal arrives before the microcontroller has performed all the needed operations, i.e. */Ready* is still asserted, that particular trigger signal is ignored.

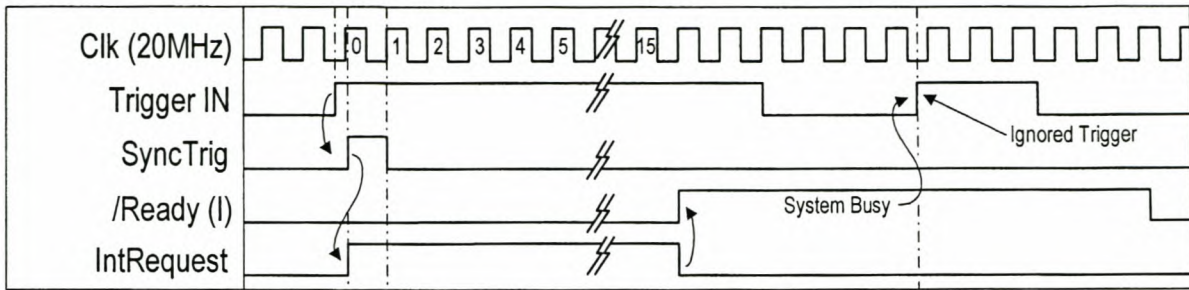


Figure 5-17 Trigger IN signal conditioning operation

5.3.6 Counter Output Latch

The 24-bit counter has a real-time output of the microsecond count. When *Trigger IN* arrives, signal *SyncTrig* enables a latch-configured *D-type* flip-flop. This latch makes the applicable microsecond count available to be output by a separate data latch (section 4.2.3) on the 8-bit system data bus. Figure 5-18 shows the *LPM_DFF* function and Figure 5-19 shows operation of the counter output latch.

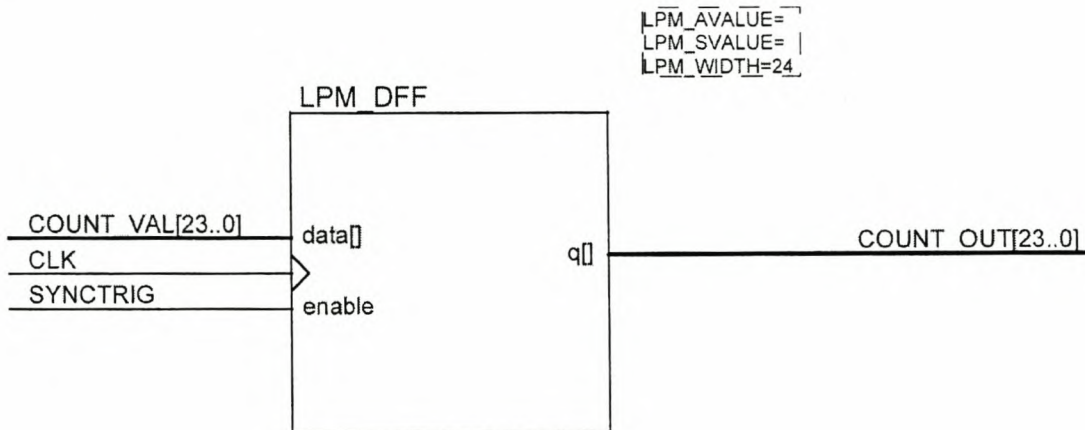


Figure 5-18 LPM_DFF function used as a latch

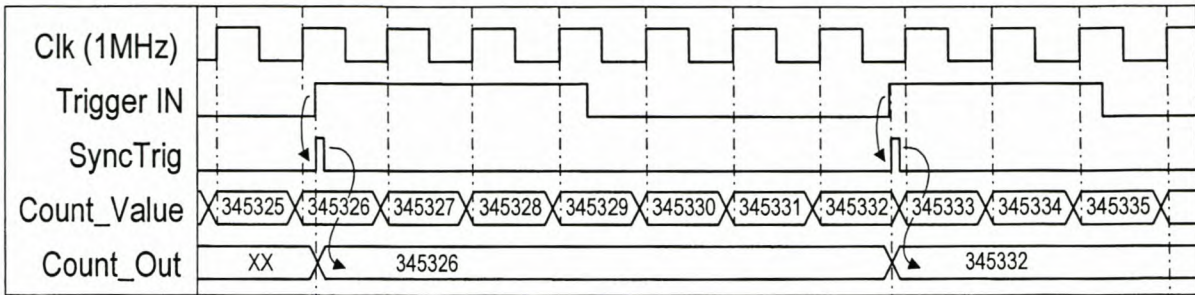


Figure 5-19 Counter output latch operation

5.3.7 Data Bus Output Latches

The 24-bit counter value is very easily broken up into three bytes, as can be seen in Figure 5-20. The address decoder selects the appropriate latch and so the high, middle and low bytes of the microsecond count is output on the system data bus.

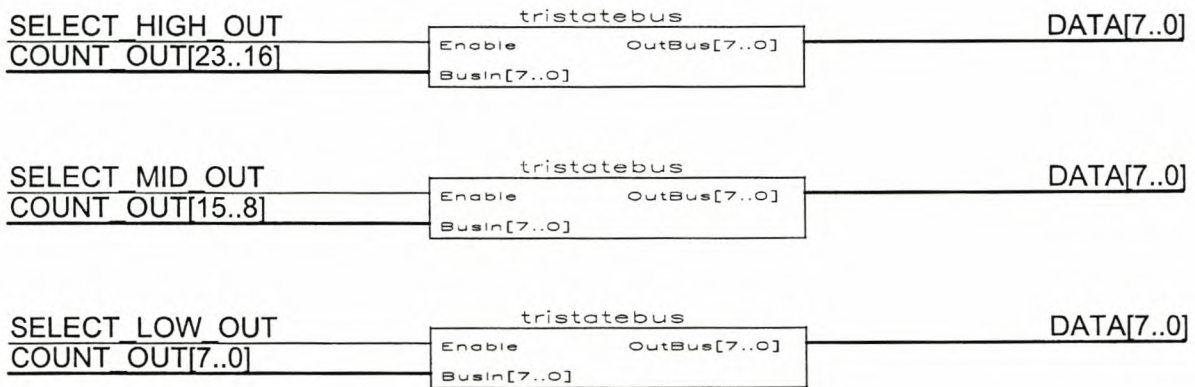


Figure 5-20 High, middle and low microsecond count bytes connected to data bus

Figure 5-21 presents a final simulation of the microsecond counter operation.

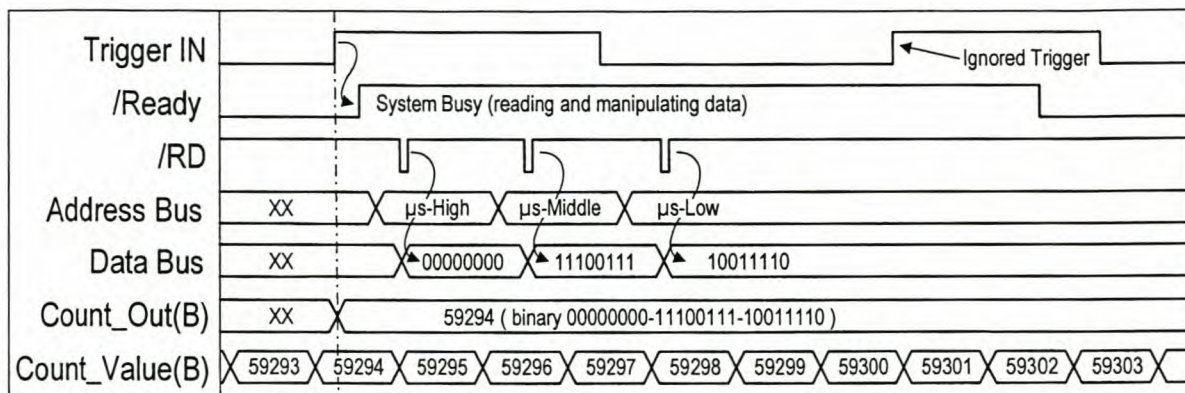


Figure 5-21 Microsecond Counter operation

5.4 Microsecond Counter Comparator

The microsecond counter comparator operates in exactly the same way as the RTC Comparator (discussed in section 4.4). The function of the microsecond comparator is to compare a pre-programmed value for the microsecond count with the current microsecond count. When these two values are the same, and when *HMSEqual* is asserted (hours, minutes and seconds are equal to a pre-programmed time), a trigger signal *Trigger OUT* is generated. This signal is 1 μ s wide. Figure 5-22 shows the overview block diagram.

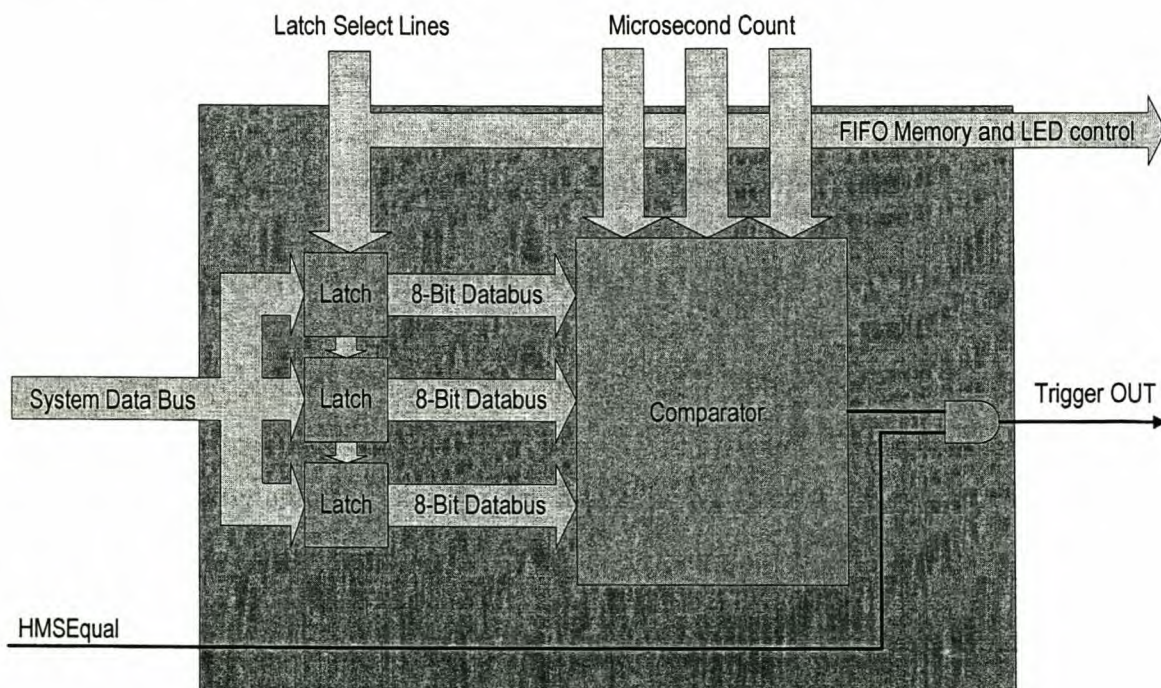


Figure 5-22 Microsecond Counter Comparator system block diagram

The comparator is programmed in the same way as the RTC comparator (discussed in chapter 4). The address decoder selects the appropriate input latch and the corresponding value for microsecond high, middle and low bytes are programmed into the comparator. The input latches were discussed in section 4.2.2, and will not be discussed here. The comparator is implemented using *LPM_Compare* functions, as in the RTC comparator (section 4.4.2). Figure 5-23 shows the detail microsecond comparator block diagram.

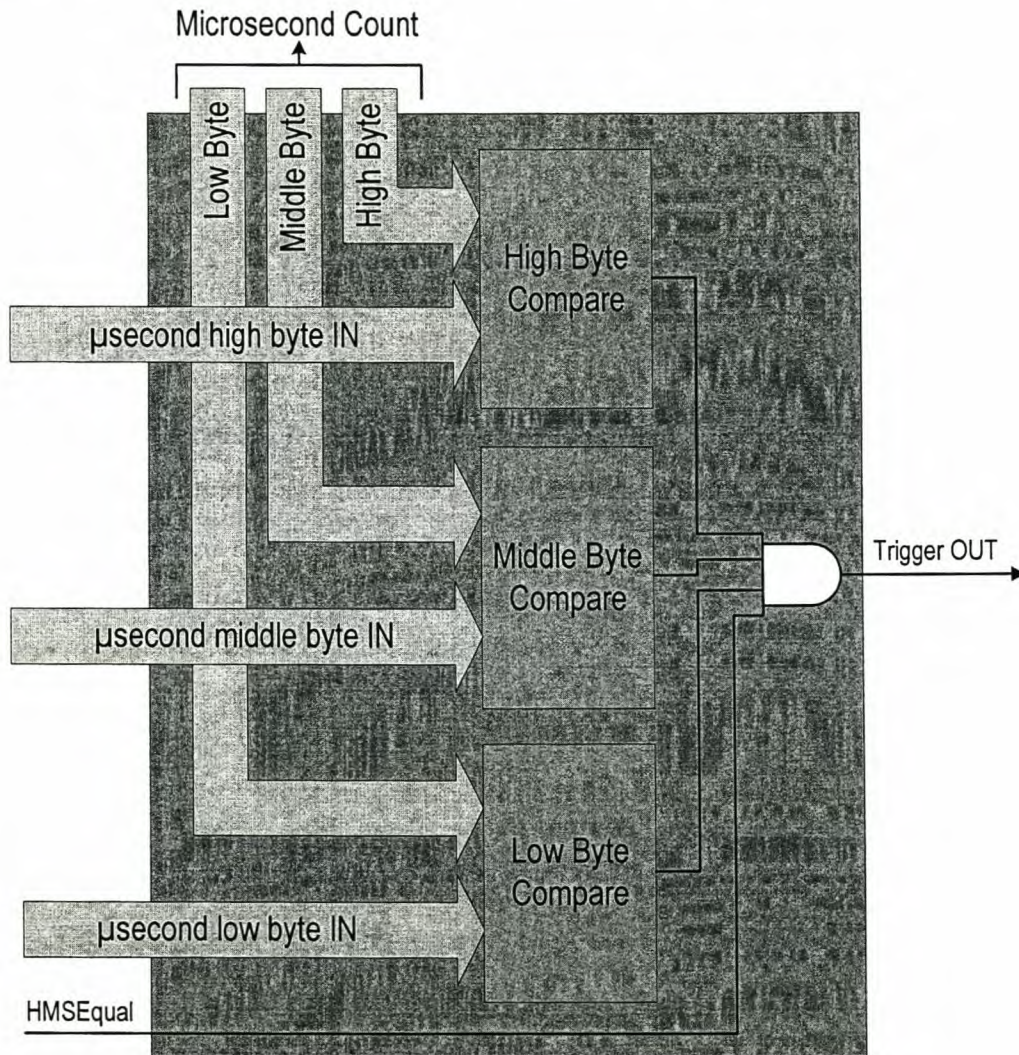


Figure 5-23 Microsecond Counter Comparator detail block diagram

Figure 5-24 shows the most important simulation waveforms. Appendix E gives complete *Altera® MAX+PLUS®* simulations.

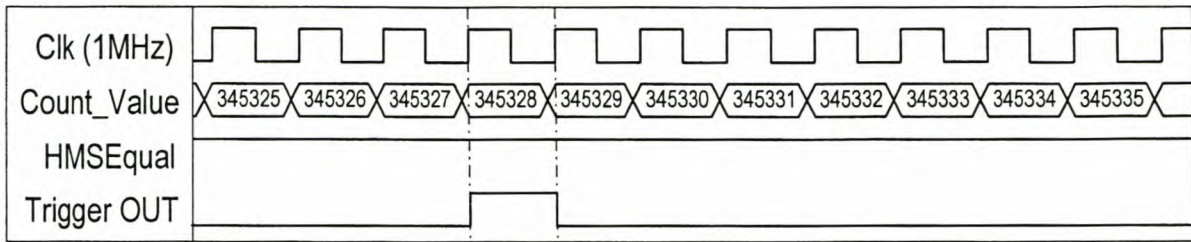


Figure 5-24 Microsecond Comparator programmed to generate a trigger on the 345328th microsecond (*HMSEqual* is asserted)

5.5 Address Decoder

As mentioned earlier, the microsecond counter address decoder works in exactly the same way as the RTC address decoder, except for the latch addresses, and a couple of other changes, which will be discussed here.

5.5.1 Indicator LEDs

There are four indicator LEDs used in the system. They are:

- *GPS Ready:*
 This indicator LED lights when communication with the GPS receiver has been established, and no errors occurred.
- *IPPS:*
 This indicator LED lights with every one pulse per second received from the GPS receiver.
- *System Ready:*
 This indicator LED lights when the system is ready. It switches off when the system is busy with operations like sending and receiving data from a PC or the GPS receiver, or when a trigger signal is received.
- *Trigger:*
 This indicator will light when a trigger signal (*Trigger IN*) is received.

Every indicator LED has a unique address, and the LEDs are switched on and off by writing to their individual addresses. The *LED_ON* addresses switch a monostable multivibrator (section 5.3.5.2) on and the *LED_OFF* addresses reset the monostable multivibrator. In order

not to draw too much current from the microsecond counter EPLD, the LEDs are switched on via a MOSFET¹ transistor. Complete schematic diagrams may be found in Appendix I.

5.5.2 FIFO Memory Control Signals

Section 5.6 will discuss the FIFO memory in detail. It is important however that the addressing of the control signals be discussed here. The FIFO memory has three control signals, namely */RD*, */WR* and */RS*. They are read, write and reset signals. Every signal has its unique address and are asserted by writing the corresponding address on the address bus and then asserting the microcontroller */RD* or */WR* signals. These control signals are active low, and thus need to go low for a read, write or reset operation to be carried out.

5.6 FIFO Memory Operation

As mentioned in chapter 2, the FIFO² memory is used as storage for received trigger time-stamp information. The information may be read with compatible external hardware. Figure 5-25 shows how the FIFO memory is implemented.

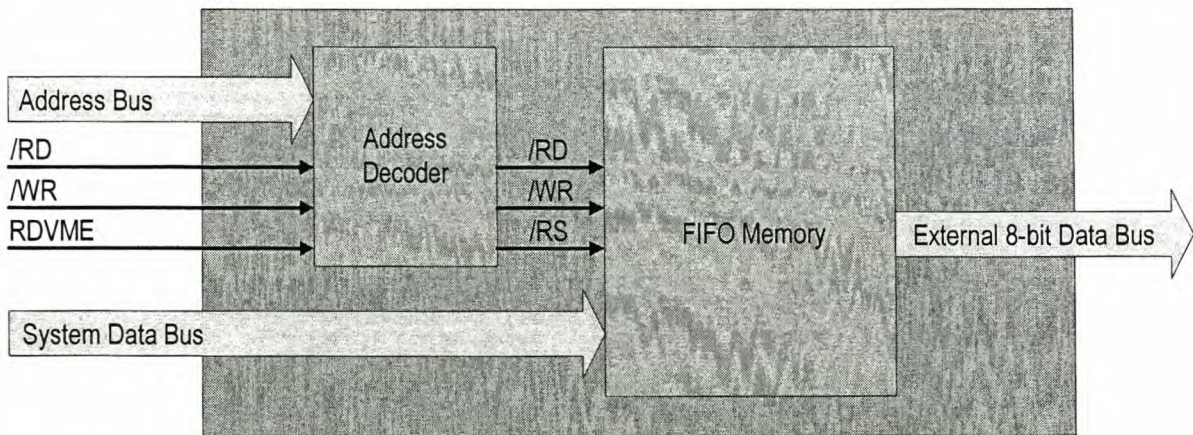


Figure 5-25 FIFO memory implementation

The external 8-bit data bus and signal *RDVME* is connected to the external system interface control bus, as can be seen in Figure 2-9. The FIFO memory can be cleared by asserting */RS*

¹ Metal Oxide Semiconductor Field Effect Transistor
² First In, First Out

and memory can also be cleared by reading all data contained in the FIFO memory. Detail on the FIFO memory operation follows.

5.6.1 FIFO Detail

The FIFO chosen for this application is the IDT 7201 [45]. It has 512 bytes of storage space and two separate data buses, one for data input and one for data output. Figure 5-26 shows a functional block diagram. The block diagram shows these signals of importance:

- */W*:
Write data into the FIFO memory.
- */R*:
Read data from the FIFO memory.
- */RS*:
Reset, and thus clear, the FIFO memory.
- */XI*, */XO* or */HF*:
Expansion input and output is used when FIFO memories are cascaded. */HF* is a signal that is asserted when the memory is half-full. These signals are not used in this application.
- */EF* and */FF*:
Empty and Full flags. These signals are asserted when the FIFO is empty or full, respectively. No write operations are allowed when the memory is full, and no read operations are allowed when memory is empty.

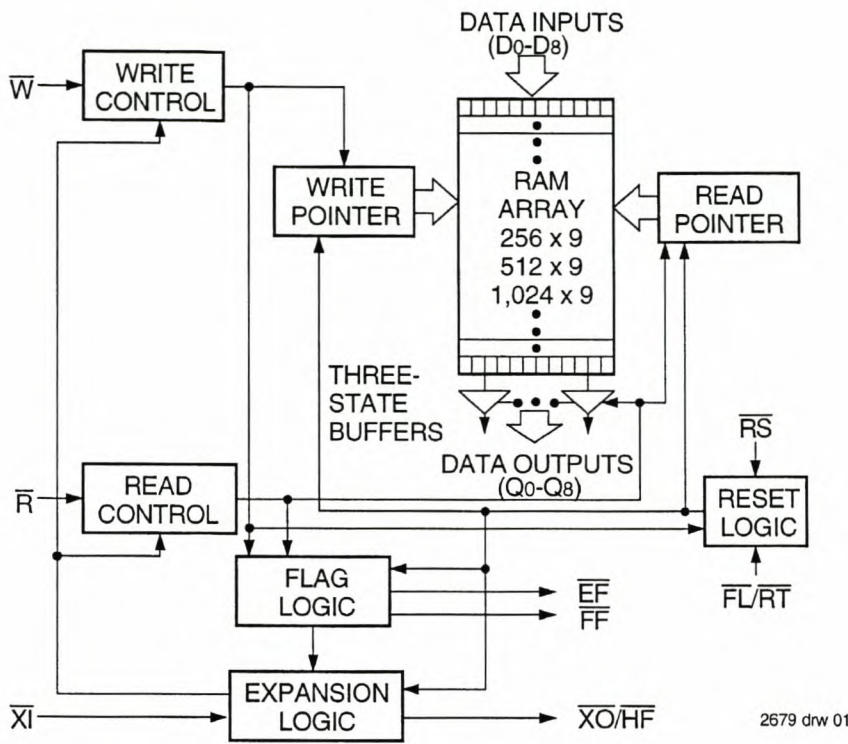
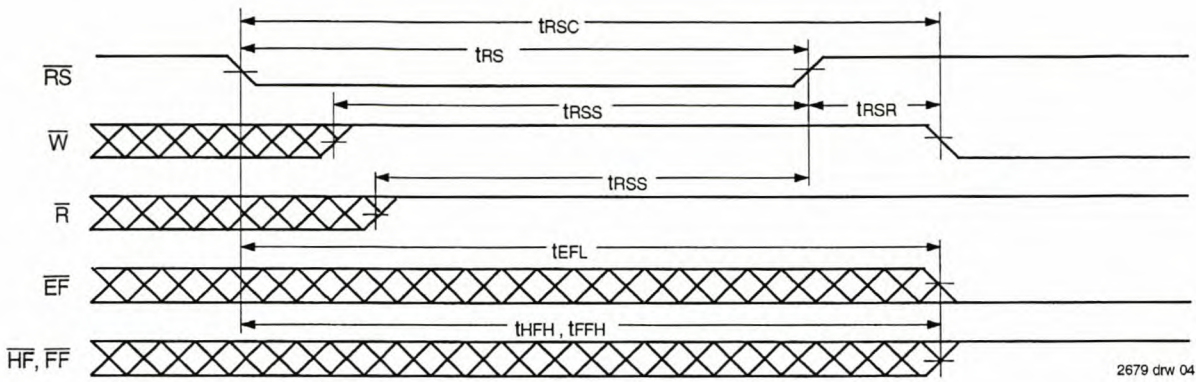


Figure 5-26 FIFO memory functional block diagram [45]

For more detail on the FIFO memory, consult the manufacturer datasheet [45]. The software procedures for controlling the FIFO memory in this system will be presented in chapter 6.

5.6.2 Reset Operation

Reset is accomplished whenever $\bar{R}S$ is taken to low state. During reset, both internal read and write pointers are set to first location. A reset is required after power up before a write operation can take place. Both \bar{R} and \bar{W} signals must be in high state during reset operation, as shown in Figure 5-27. All timing constraints shown in Figure 5-27, Figure 5-28, Figure 5-29 and Figure 5-30 are adhered to. The timing values shown can be found in references [45].



- NOTES:
 1. $\overline{EF}, \overline{FF}, \overline{HF}$ may change status during Reset, but flags will be valid at t_{rSC} .
 2. \overline{W} and $\overline{R} = V_{IH}$ around the rising edge of \overline{RS} .

Figure 5-27 FIFO reset operation timing diagram [45]

5.6.3 Write Operation

A write cycle is initiated on the falling edge of \overline{W} and if the Full Flag \overline{FF} is not set. To prevent data overflow, the Full Flag \overline{FF} will go low, inhibiting further write operations. When the FIFO is full, \overline{FF} will ensure that all subsequent write operations are blocked and thus ignored. Figure 5-28 shows FIFO write and read operations [45].

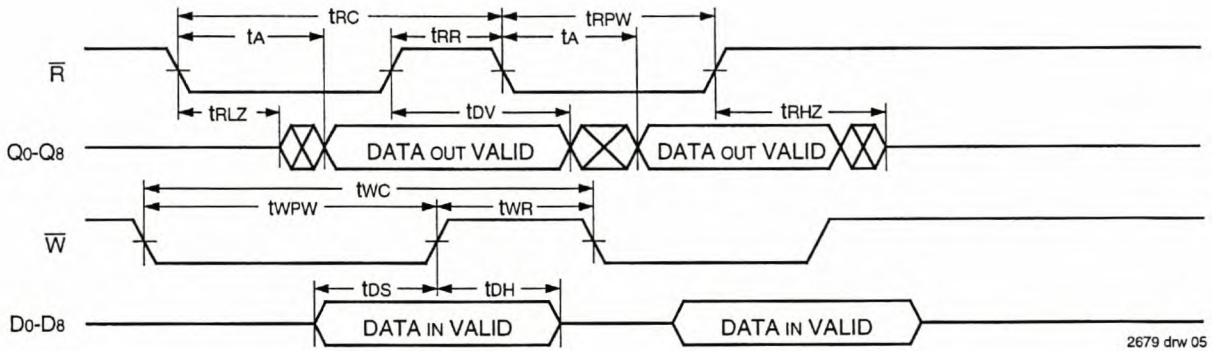


Figure 5-28 FIFO asynchronous write and read operations [45]

5.6.4 Read Operation

A read cycle is initiated on the falling edge of \overline{R} provided the Empty Flag \overline{EF} is not set. Data is accessed on a First-In, First-Out basis, independent of any ongoing write operations. After \overline{R} goes high, the data output bus will return to high-impedance condition until the next read operation. When all data has been read from the FIFO, the Empty Flag \overline{EF} will go low, thus inhibiting further read operations with the data outputs remaining in the high impedance

state. Once a valid write operation has been completed, \overline{EF} will go high and a read operation may once again begin. Figure 5-28 shows FIFO write and read operations [45]. Figure 5-29 and Figure 5-30 shows Full Flag (\overline{FF}) and Empty Flag (\overline{EF}) operation respectively.

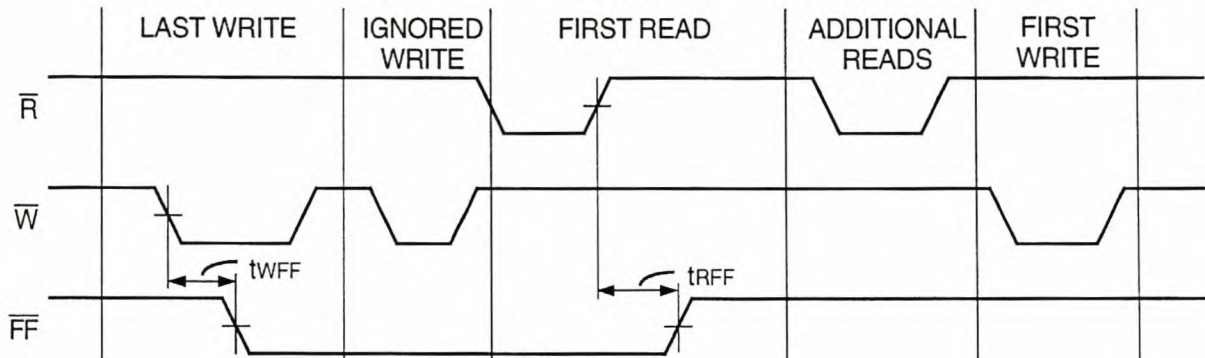


Figure 5-29 FIFO full flag Operation [45]

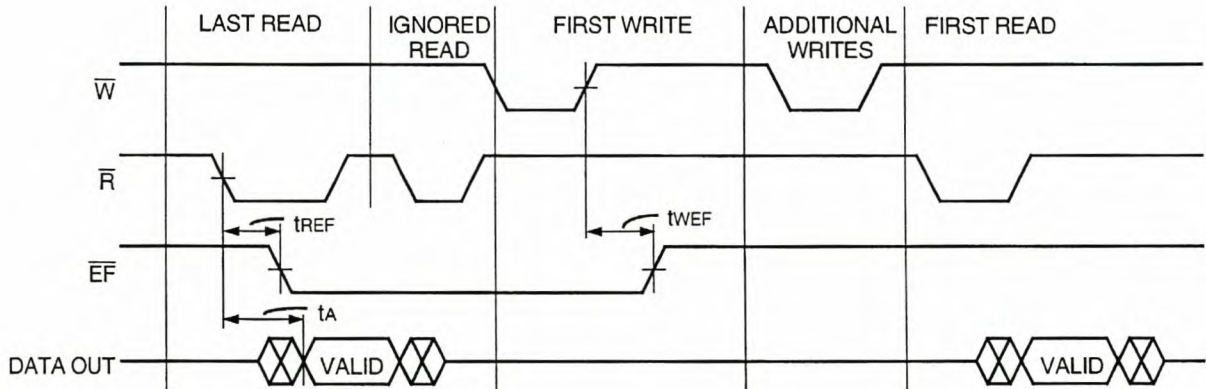


Figure 5-30 FIFO empty flag Operation [45]

5.7 Conclusion

In this chapter the Microsecond Counter and Comparator were discussed. The Microsecond Counter is used to count 1MHz clock pulses in order to obtain a microsecond count. This count is reset every second by the GPS receiver *IPPS*. The Microsecond Counter Comparator is used to compare a pre-programmed value for the microsecond count with the actual microsecond count and, when they are equal, generate a $1\mu\text{s}$ -wide *Trigger OUT* signal if *HMSEqual* from the RTC Comparator is asserted. The *Trigger OUT* signal will be used to start pre-programmed data acquisition runs for analysis of a power system.

Chapter 6

System Microcontroller

6.1 Introduction

The system microcontroller's function is to facilitate data transfer between different system blocks shown in Figure 2-9. The GPS receiver, host computer commands, RTC, Microsecond Counter and FIFO memory need to be controlled effectively and efficiently.

An *Atmel ATmega161L* microcontroller was chosen for this application [46], although an *AT90s8515* was used in an earlier prototype. The *AT90s8515* proved to be inadequate for this application because two UART¹ peripherals are required and because of insufficient program code memory space. The most notable feature of the *ATmega161L* is that it has two programmable serial UARTs, for communicating with a personal computer and the GPS receiver. More details on the *ATmega161L* can be found in references [46]. Figure 6-1 is a schematic diagram of the microcontroller showing pin connections of the data, address and control buses.

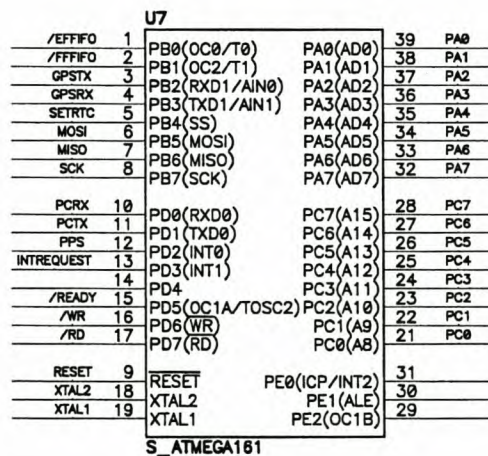


Figure 6-1 Microcontroller diagram

¹ Universal Asynchronous Receiver Transmitter

Figure 6-2 is a block diagram of the microcontroller that shows peripherals and different buses.

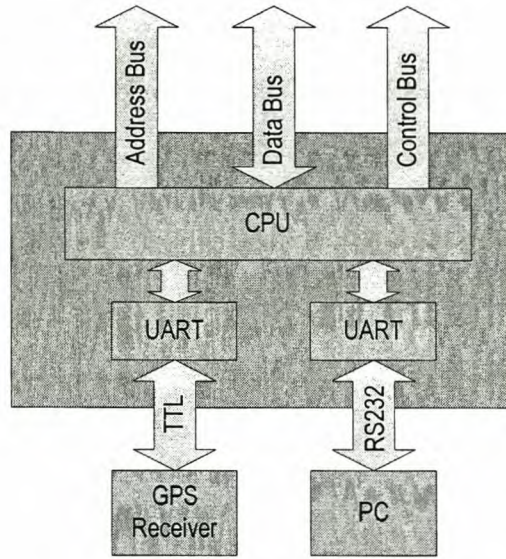


Figure 6-2 Microcontroller functional block diagram

There are three main components of Figure 6-2. They are communications, internal buses namely address, data and control buses, and the CPU running the hardware system control program. Port A is the data bus, Port C is the address bus and Port B and D are the control and communication buses respectively. Figure 6-1 shows some familiar control signals, namely *SetRTC*, *PPS*, *IntRequest*, */RD*, */WR* and */Ready*. Pins 3 and 4 are connected to the GPS communication lines, and Pins 10 and 11 are connected to a PC RS232 port via a *MAXIM Max232CPE* chip. For complete schematic diagrams, consult Appendix I.

Table 6-1 summarises the features of the embedded microcontroller system control program.

Table 6-1 System control program features

Feature	Description	Comment
Communication with GPS receiver	Send initialisation routines, read and store time information in RTC	Data received from GPS receiver is stored in a software data buffer and processed
Communication with PC	Receive and process commands from the PC	System responds to commands by sending

	control program (chapter 7)	requested data to the PC
Process received trigger data	Timestamp of <i>Trigger IN</i> signal is stored for downloading and processing	Trigger data is stored in software data buffer and FIFO memory
Set RTC and Microsecond Comparators to generate <i>Trigger OUT</i> at a pre-programmed time	Read 'generate trigger at hh:mm:ss.µs'-time from PC and store in comparators	"Generate trigger at"-time is stored in software data buffer for transmitting to PC when requested
Report system status	Transmit GPS Time, RTC-time and error, triggers received and triggers generated for display in PC control program	Calculate RTC error by subtracting RTC time from actual GPS time.

6.2 System Control Program High-Level Discussion

A list of features were presented in Table 6-1. These features are implemented in the system control program. Figure 6-3 shows a high-level flow diagram of this program. When the system starts up, initialisation procedures are executed. These initialisation procedures include setup of the GPS receiver, reading time information from it and programming the RTC with the correct time. The program then enters an "endless-loop". This is a safe place where interrupts can execute their service routines. There are three main interrupt events that may occur:

- *1PPS Interrupt:*
Every 1PPS signal generates an interrupt. The interrupt service routine sets the RTC with the correct time once every hour. This is a precautionary measure in the unlikely event that the RTC time should be incorrect.
- *PC Command Interrupt:*
The system can be connected to a PC for downloading and control purposes (chapter 7). When the PC sends a command, an interrupt is generated. The interrupt service routine identifies which command has been received and acts accordingly.
- *Trigger Received:*
When a trigger signal, *Trigger IN*, is received, an interrupt is generated. The interrupt service routine reads time information, i.e. date, hours, minutes, seconds and

microseconds, of the trigger signal from the RTC and Microsecond Counter. The trigger signal is numbered and stored in a software data buffer (section 6.5.3) for later retrieval by the PC control program.

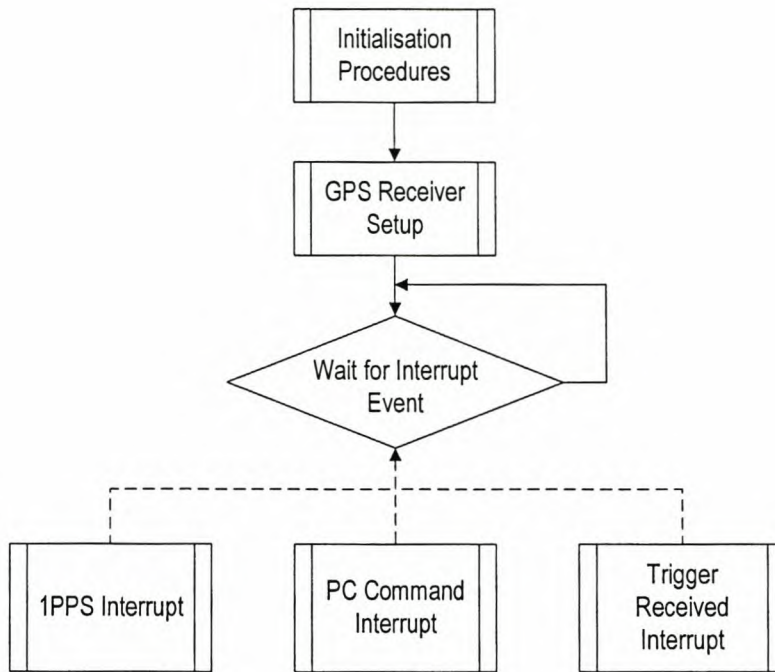
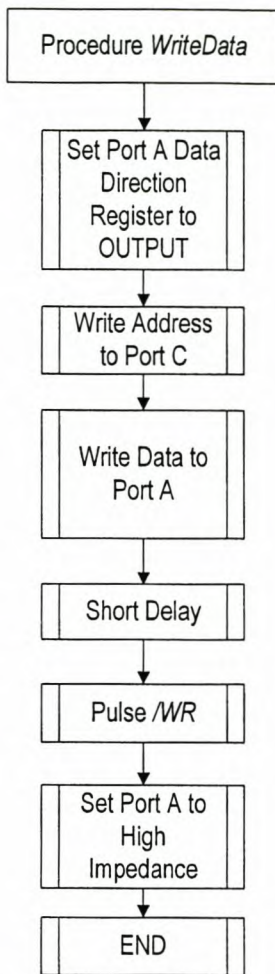
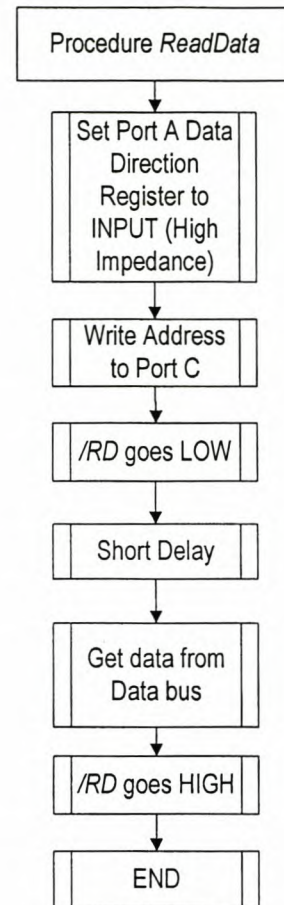


Figure 6-3 System Control Program high-level flow diagram

6.3 Data Write and Read Routines

The address decoder and latch operation was discussed in chapter 4. The specific data input or output register address is written to the address bus, data is written to the data bus and $/WR$ is asserted in order to send data to the correct register. For a read operation the process is as follows: the address of the register to be read is written to the address bus, $/RD$ is asserted and the data bus is read after a short time is given to the data register for the data to be output on the data bus. It should also be noted that the data write routine must configure the data bus port as an output, and that the data read routine must configure the data bus port as an input. The data bus should also be released, i.e. put into high impedance mode, after any operation, in order to avoid data bus contention. Figure 6-4 and Figure 6-5 describes the data write and read procedures respectively. A full list of data register addresses can be found in Appendix C.1.

Figure 6-4 Procedure *WriteData*Figure 6-5 Procedure *ReadData*

6.4 Initialisation Procedures

When the system powers up, some initialisations need to be done in order to configure different microcontroller subsystems, like UARTs, I/O ports, software data buffers and variable initialisation. These initialisation procedures are now discussed.

6.4.1 Microcontroller I/O Port Setup

The *ATmega161L* has five I/O ports [46] which can be configured as outputs or inputs by writing to the applicable Data Direction Registers (*DDRs*) [46]. Detail about this procedure can be found in reference [46] and in Appendix G. Table 6-2 shows how every I/O port is configured.

Table 6-2 Port Initialisation values

Port	Initialise Value	Comments
Port A – Data bus	Configured as input when data is read and configured as output when data is written. Initial value is high-impedance to avoid bus contention.	Data write procedure configures Port A as output, and data read procedure configures Port A as input.
Port B – Control bus	Control signals: <i>/EF</i> and <i>/FF</i> (FIFO control signals) are input signals and <i>SetRTC</i> is an output signal.	Port B2 and B3 are GPS UART pins – setting of the data direction registers have no effect.
Port C – Address bus	Port configured as output, data register addresses are written to this bus.	Addresses of data registers are written to address decoders
Port D – Control bus	Inputs: <i>PPSInterrupt</i> , <i>IntRequest</i> . Outputs: <i>/Ready</i> , <i>/RD</i> and <i>/WR</i> .	Control signals that control the RTC and Microsecond Counter EPLDs
Port E – Test Indicator LEDs	Set as outputs that are connected to indicator LEDs.	These LEDs are for testing purposes and are not the same as system indicator LEDs (section 5.5.1)

6.4.2 Variable Initialisation

Every procedure in the system control program uses different types of variables for data storage, loop counting and temporary storage. Each of these variables need to be initialised to a certain value at system start-up by either the procedure that uses the variable or the start-up routine.

6.4.3 UART Setup

As mentioned earlier, the UART is a hardware peripheral that facilitates communication to and from the GPS receiver and PC. The *Atmel ATmega161L* has two programmable serial UARTs. The UART functional block diagram can be found in reference [46]. Table 6-3 shows the UART configuration at system start-up.

Table 6-3 UART configuration

UART	Connected to	Direction	Data Rate	Comments
UART 0	Personal Computer	Receive and Transmit	9600 bps ¹	Connected via <i>MAX232</i> RS232 TTL-to-RS232 converter
UART 1	GPS Receiver	Receive and Transmit	9600 bps	Connected directly to GPS receiver (normal TTL levels ²)

6.5 Communication Routines

As mentioned in the previous section, two programmable UARTs are used to facilitate communication to and from the GPS receiver and a PC. Details of the UARTs and the software routines that control them are now discussed.

6.5.1 UART Control Registers

The UARTs are controlled by setting or clearing certain bits in their respective control registers (*UCSRx*). The UARTs are identical and operate in the same way. There are two control registers, namely *UCSRxA* and *UCSRxB* where *x* is 0 or 1, depending on which UART is controlled. The control registers for UART0 are shown in Figure 6-6 and Figure 6-7 [46]. Bits that will be used to control the UARTs in this system are the following:

- *RXENx*:
Setting of this bit enables the UART receiver
- *TXENx*:
Setting of this bit enables the UART transmitter
- *RXCIEx*:
Setting of this bit (in conjunction with global interrupt enable [46]) enables the receive complete interrupt. This means that an interrupt will be generated once a complete byte

¹ bits per second

² 0V to 5V

is received by the UART. The interrupt service routines (PC_RxInt and GPS_RxInt) will be discussed in section 6.5.5.2.

- **UDRIEx:**

Setting of this bit enables the ‘UART data register empty’-interrupt. When there is no data present in the UART data register (UDR), an interrupt is generated. The interrupt service routine (PC_UDREmptyInt) will be discussed in section 6.5.6.2.

Bit	7	6	5	4	3	2	1	0	
\$0B (\$2B)	RXC0	TXC0	UDRE0	FE0	OR0	–	U2X0	MPCM0	UCSR0A
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Figure 6-6 UART0 control register A [46]

Bit	7	6	5	4	3	2	1	0	
\$0A (\$2A)	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	CHR90	RXB80	TXB80	UCSR0B
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	1	0	

Figure 6-7 UART0 control register B [46]

For more detail on the operation of the UART, consult references [46].

6.5.2 Timer/Counter1

When data is received from the GPS receiver or a PC, it is not known how many bytes will be received. It is assumed that a complete message has been received when the RS232 receive procedure times out after receiving the last data byte. A timer is used for this time out routine. The time out count register is loaded with a pre-determined value and it counts down from this value. The value ensures that the counter will take approximately 2 byte lengths to count from the pre-determined value to zero. The time out count register is reloaded with this value every time a byte is received. When a byte is not received, the counter will be allowed to count down to zero without being reset and an overflow flag (*TOV1*) will be set. For GPS receiver data reception, this flag is polled. When it is set, a complete message was received from the GPS receiver. For PC data reception, this flag generates an interrupt when *TOIE1* has been set to enable this interrupt. The interrupt service routine will then perform tasks on

the received message as well as disabling the timer/counter interrupt until it is needed again for data reception. Figure 6-8 presents this operation.

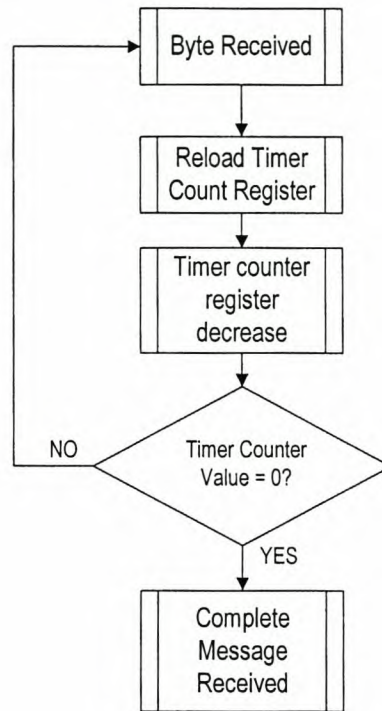


Figure 6-8 Message received timeout procedure

The *ATmega161* provides three general purpose Timer/Counters, namely two 8-bit and one 16-bit Timer/Counter. Detail on Timer/Counter1 can be found in reference [46].

6.5.3 Software Data Buffer

Data received from either PC or GPS receiver must be stored on-board for subsequent processing. A software data buffer is used for this purpose. This buffer is as an array of bytes that store all received bytes. Figure 6-9 and Figure 6-10 show write and read operation of the software data buffer. Software data buffer head and tail pointers are initialised to 1. When data is put into the buffer, the head pointer increments as data fills the buffer. When data is read, the tail pointer increments until it reaches the head pointer which means that all data has been read. These buffers have a pre-defined size and when number of bytes received exceeds the buffer length, the buffer is cleared and data is written from the beginning of the buffer. It is important to ensure that the buffer size is adequate.

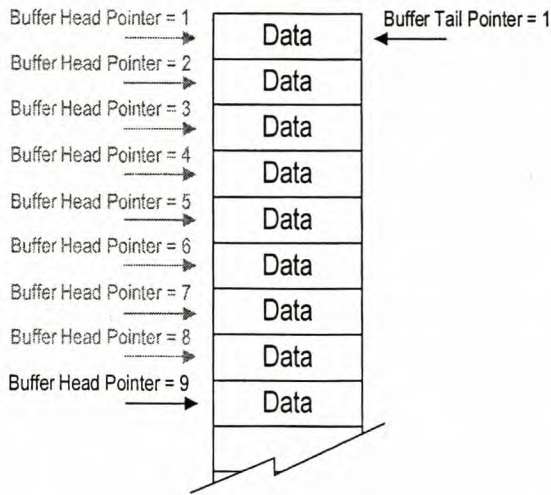


Figure 6-9 Software data buffer write

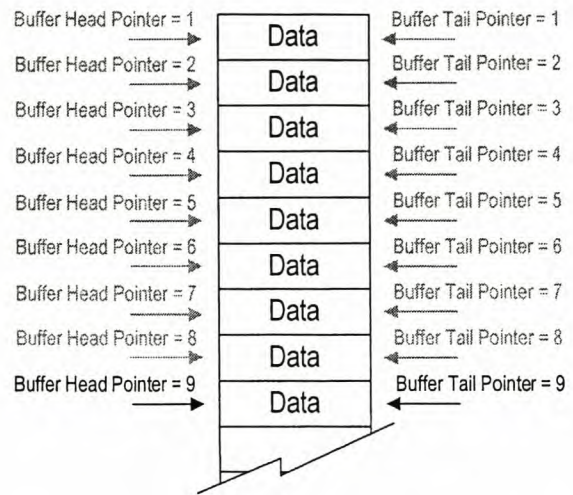


Figure 6-10 Software data buffer read

When a procedure needs to determine if data is present in the data buffer, the head and tail pointers can be compared. If they are equal, all data has been read and no new data is present in the buffer. The buffer may be cleared by setting the head and tail pointers equal to zero.

6.5.4 Communication Error Checking

When data is transmitted or received from the GPS receiver or PC, errors in communication can occur, especially if long communication cables are used. For this reason, two different methods of communication error checking are implemented, namely the XOR-checksum method used for GPS receiver and CRC-16 error checking used for the PC communication.

6.5.4.1 GPS Receiver Communication Error Checking

Section 3.5.1.1 described the GPS receiver communication message. The byte before $\langle CR \rangle \langle LF \rangle$ is a checksum byte which is the XOR of all the bytes in the message before the checksum. A received message is written to a software data buffer (section 6.5.3). The buffer head pointer will point to the last message byte received, which will be a $\langle LF \rangle$ character. It is thus known that the checksum byte will be at position *Buffer Head Pointer* - 2. Figure 6-11 gives a graphical representation of this.

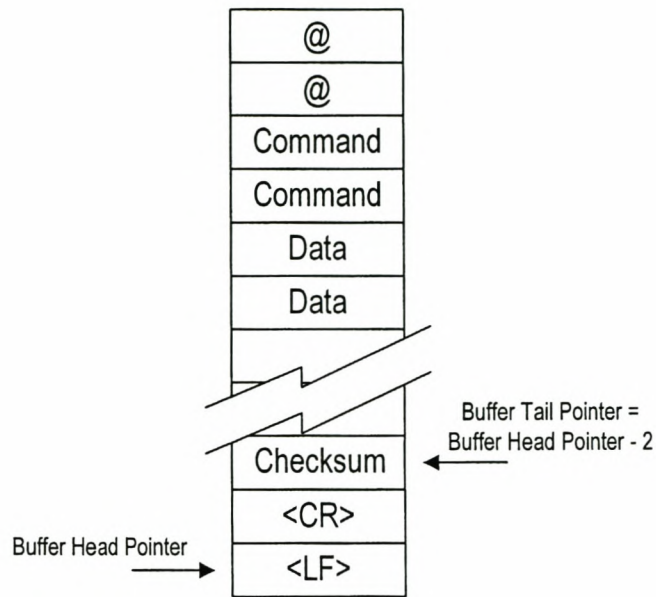


Figure 6-11 GPS message in software data buffer

The software data buffer is stepped through from the beginning byte, the XOR-checksum is calculated byte-by-byte and it is compared with the received checksum byte. If the calculated checksum is not equal to the received checksum byte, an invalid message was received. If however, a correct checksum was received, a valid message was received. Figure Figure 6-12 describes this process.

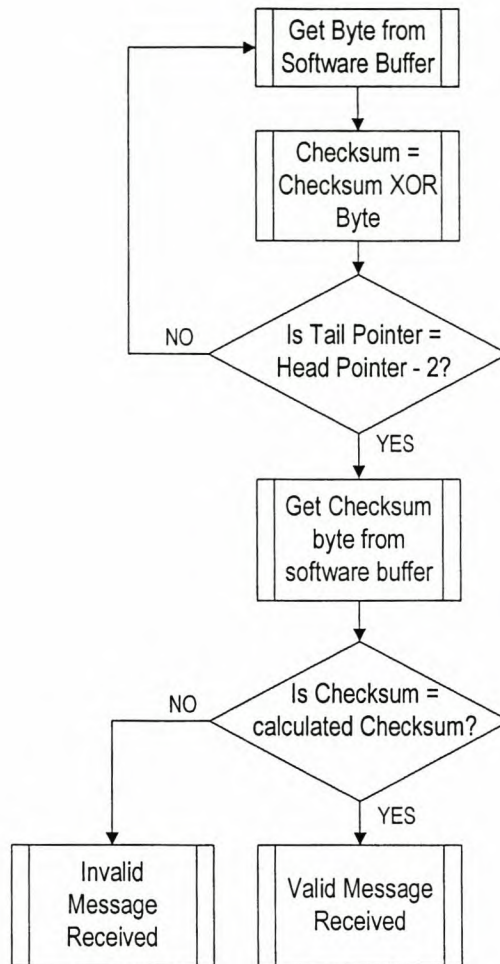


Figure 6-12 GPS message validation procedure

6.5.4.2 Host Computer Communication Error Checking

CRC16¹ is a common 16-bit method for detecting errors in transmitted messages or stored data. The CRC16 algorithm is a very powerful, but an easily implemented technique to obtain data reliability [50]. Due to its many advantages, CRC-16 error checking is used for detecting errors in PC communication messages.

Every byte in a data message is divided by a binary number called the polynomial. The rest of the division is the CRC16 checksum, which is then appended to the transmitted message. The receiver of the data divides the message, including the CRC16, by the same polynomial the transmitter has used. If the result of the division is zero, it means that transmission was successful. If the result is not zero, an error occurred during data transmission. Figure 6-13

¹ Cyclic Redundancy Check

shows the CRC16 calculation routine. It is somewhat more complex than the simple description given above, because of byte division that is not as easily implemented in software routines.

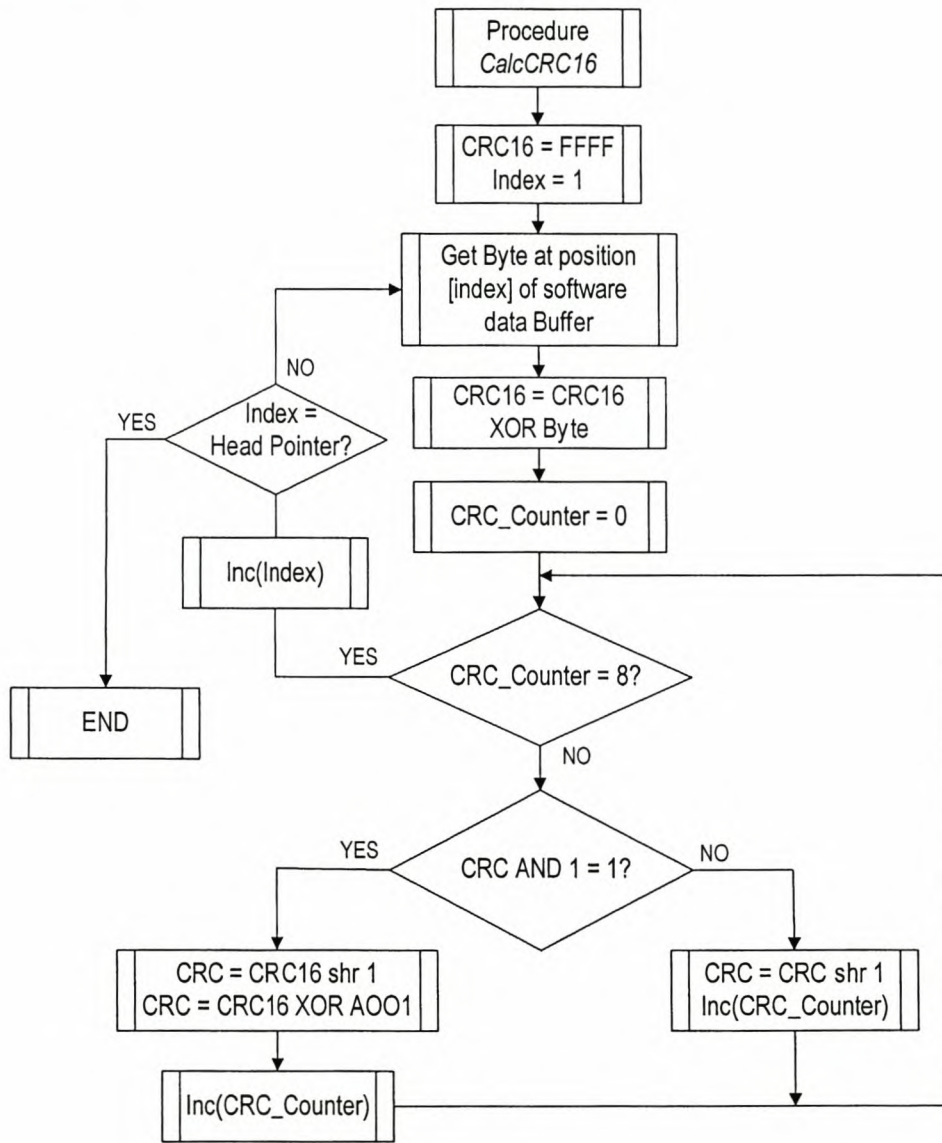


Figure 6-13 Procedure *CalcCRC16*

The 16-bit value *CRC16*, calculated in the above procedure, is split into two bytes and added to the message to be transmitted. If this routine is applied on a message that was received, the value of *CRC16* should be zero if a valid message was received. For more detail on the CRC algorithm and other implementations of it (CRC-CCITT and CRC-32), reference [50] may be consulted.

6.5.5 GPS Communication Operation

GPS receiver communication is started by sending a command, including parameters, checksum and control characters to the GPS receiver. After the command has been sent, the program waits for the GPS receiver to respond to the command. When the GPS receiver replies with data, it is put into a software data buffer for processing. When the received message is validated (section 6.5.4.1), applicable processing of the data is done. This operation is shown in Figure 6-14.

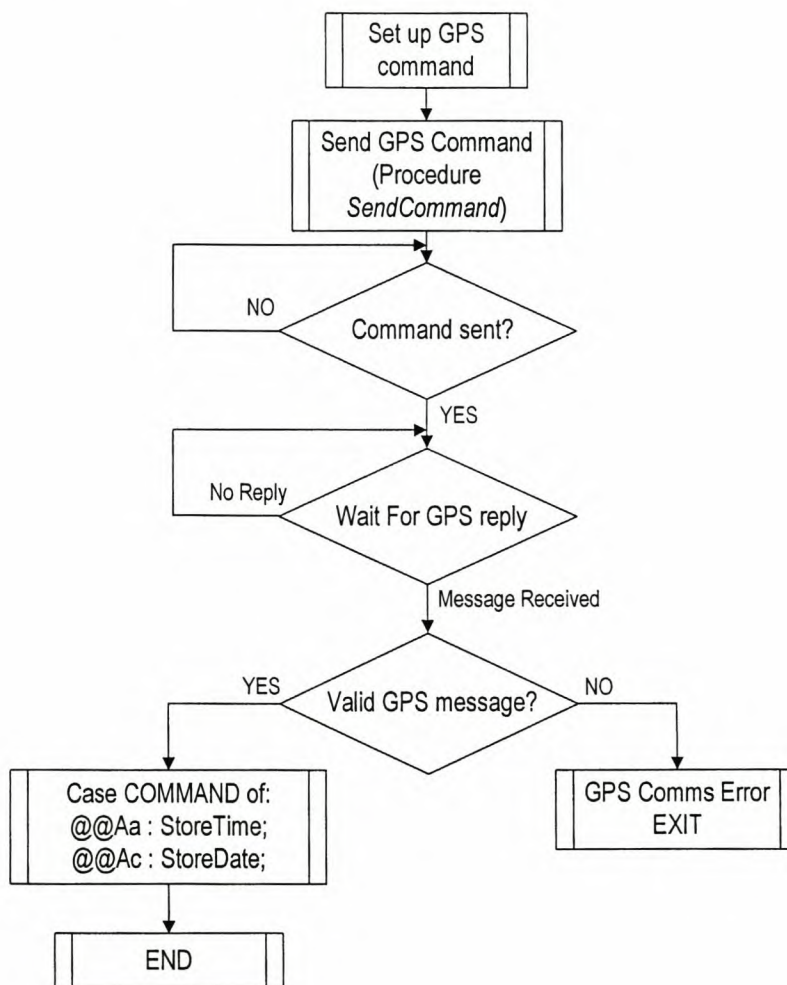


Figure 6-14 GPS communication operation

Figure 6-14 shows the main communication procedures, namely procedure *SendCommand*, procedure *WaitForGPSReply*, procedure *ProcessMessage* which validates GPS messages and processes the received data, procedure *StoreTime* and procedure *StoreDate*. The communication validation part of procedure *ProcessMessage* has already been discussed in section 6.5.4.1.

6.5.5.1 Procedure *SendCommand*

The command to be sent is placed in a variable called *Command*, before procedure *SendCommand* is called. Procedure *SendCommand* then shifts the command out to the UART byte-by-byte. This is done by simply writing the byte to the UART Data Register (*UDR*), waiting for *UDR* to empty and writing the next byte. After the whole command has been sent, procedure *WaitForGPSReply* (section 6.5.5.4) is called. When a reply from the GPS receiver has been received, procedure *ProcessMessage* is called to process the received data. Figure 6-15 describes procedure *SendCommand*.

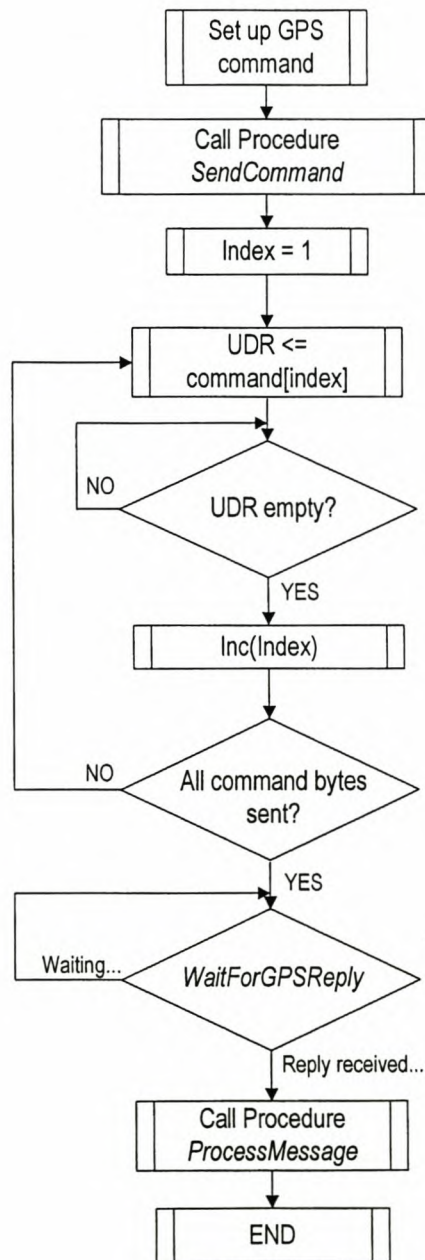


Figure 6-15 Procedure *SendCommand*

Procedure *StoreTime* and *StoreDate* is called within procedure *ProcessMessage* depending on whether time or date information has been requested from the GPS receiver. Next, procedure *GPS_RxInt* will be discussed. This procedure receives data from the GPS receiver.

6.5.5.2 Procedure *GPS_RxInt*

When the *RXCIE1* bit is set, an interrupt will be generated when data is received from the GPS receiver. When an interrupt is generated, interrupt service routine *GPS_RxInt* will be called when the GPS receiver has sent data. Procedure *GPS_RxInt* functions as described by Figure 6-16.

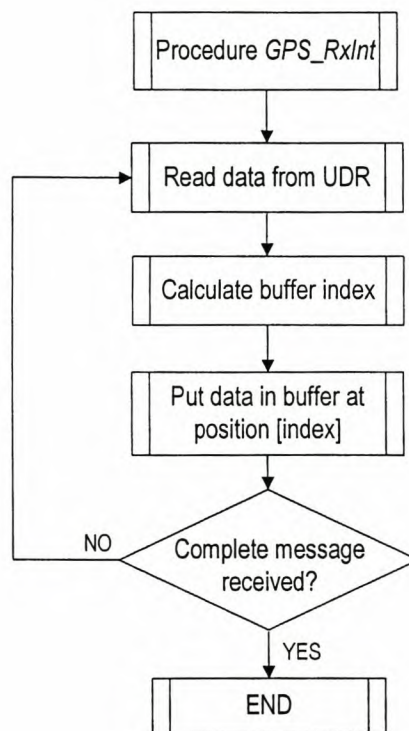


Figure 6-16 Procedure *GPS_RxInt*

Data is read directly from the UART1 data register (*UDR1*) and data is put into the software data buffer (section 6.5.3). By using the serial communications time out in section 6.5.2, it can be determined whether a complete GPS receiver data message has been received.

6.5.5.3 Function *ByteInGPSRxBuf*

This function reads data from the software data buffer. The read data byte is returned in the function name. This function automatically handles the tail pointer of the software data buffer while data is read from it (section 6.5.3). Figure 6-17 describes this function.

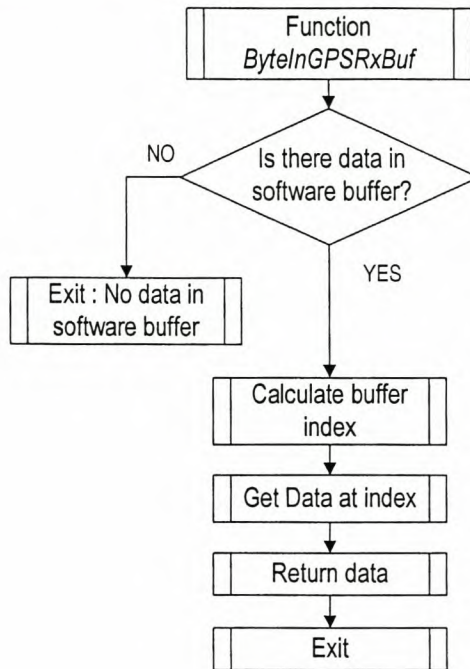


Figure 6-17 Function *ByteInGPSRxBuf*

6.5.5.4 Procedure *WaitForGPSReply*

Procedure *WaitForGPSReply* enables the 'UART1 data received'-interrupt by setting the *RXCIE1*-bit (section 6.5.1) in the UART1 control register. When data is received from the GPS receiver it is put into the software data buffer. By using the Timer/Counter1 overflow flag (*TOV1*) as described in section 6.5.2, procedure *WaitForGPSReply* can determine when a complete GPS received data message has been received (procedure *GPS_RxInt*, section 6.5.5.2). Figure 6-18 describes procedure *WaitForGPSReply*.

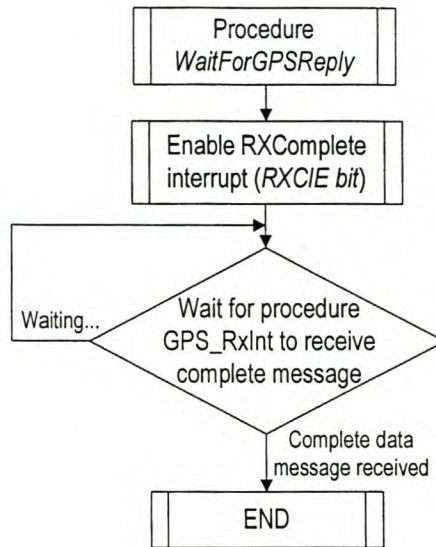


Figure 6-18 Procedure *WaitForGPSReply*

After a complete GPS receiver data message has been received, procedure *ProcessMessage* can process the received data.

6.5.5.5 Procedure *ProcessMessage*

The function of procedure *ProcessMessage* is to validate the message received from the GPS receiver. The detailed validation procedure was discussed in section 6.5.4.1. A case statement decides what needs to be done with the received data based on what command was sent to the GPS receiver. If time information was requested, procedure *StoreTime* will be called. If date information was requested, procedure *StoreDate* is called. Figure 6-19 shows procedure *ProcessMessage*.

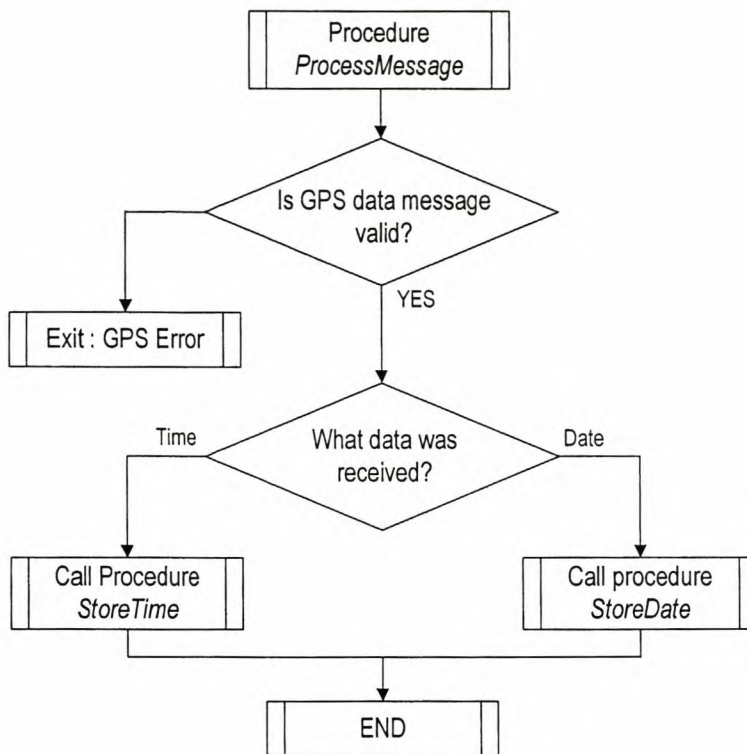


Figure 6-19 Procedure *ProcessMessage*

Procedures *StoreTime* and *StoreDate* are now discussed.

6.5.5.6 Procedure *StoreTime*

A typical GPS receiver reply was shown in chapter 3. Procedure *StoreTime* sets the GPS data receive buffer tail pointer to the beginning of the received time information. Data is stored in the applicable registers by first reading it from the software data buffer using function *ByteInGPSRxBuf* (section 6.5.5.3) and then storing it using procedure *WriteData* (section 6.3). The data is read until all information has been processed. Procedure *StoreTime* has a parameter called *DoStore* that controls whether the time data is stored to the RTC (by procedure *WriteData*) or transmitted to the PC control program. More on this parameter can be found in section 6.6.3. Procedure *StoreTime* is described by Figure 6-20.

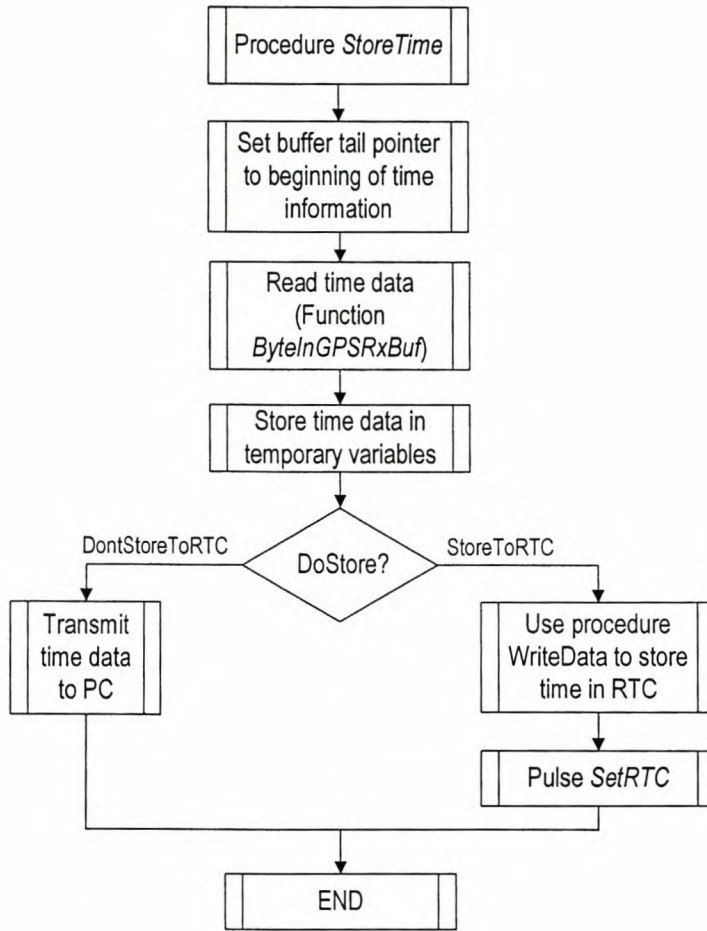


Figure 6-20 Procedure *StoreTime*

6.5.5.7 Procedure *StoreDate*

Procedure *StoreTime* functions the same way as procedure *StoreTime*, except that there is no option *DoStore*. The software data buffer tail pointer is set to point at the beginning of date data, and with the help of function *ByteInGPSRxBuf* data is stored in temporary variables for use in trigger time stamping.

6.5.6 Host Computer Communication Operation

Communication with the host computer functions much in the same way as GPS receiver communication. PC communication uses the software data buffer for transmission and reception of data, whereas GPS receiver communication only uses this type of buffer for reception of data. Another difference between PC and GPS receiver communication is that transmission and reception of data are not at pre-determined times. Data can be received at any time from the PC and the response must be immediate. It is also important to note that

the system will never initiate communication with the PC control program. The PC control program will always send a command on which the system must respond.

Data is transmitted by using the 'UART0 Data Register Empty'-interrupt (enabled by setting bit *UDRIE0* in the UART0 control register) – when the UART0 Data Register (*UDR0*) is empty and interrupt service routine ensures that data is read from the software data buffer until all data has been transmitted.

PC data transmit and receive flow diagrams are presented by Figure 6-21 and Figure 6-22.

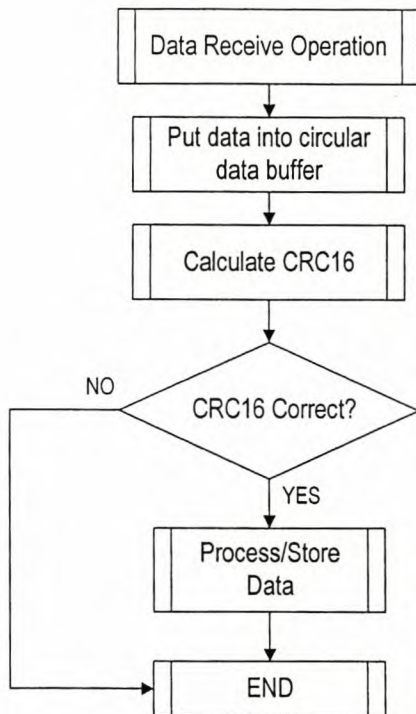


Figure 6-21 PC data receive operation

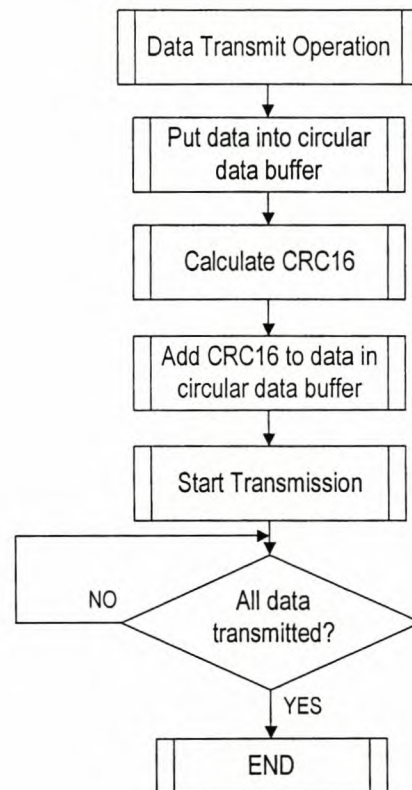


Figure 6-22 PC data transmit operation

Procedure *PC_RxInt* (section 6.5.6.1) receives data from the PC and places it into the software data buffer. When no more bytes are being received, a timer overflow interrupt is generated to signal the end of the data message. The timer overflow interrupt service routine calls procedure *ServiceComms* (discussed in section 6.7.2) that generates a response to received commands.

For data transmission, data is put into a software data transmit buffer by procedure *TxByteToPC* (section 6.5.6.3). Procedure *StartTransmit* (section 6.5.6.4) calls procedure

CalcCRC16 (section 6.5.4.2) which calculates CRC-16 on the data. The ‘UART0 data register empty’-interrupt is enabled by setting bit *UDRIE0*. When the UART data register (*UDR0*) is empty, a new byte will be read from the software transmit data buffer by procedure *PC_UDREmpty* (section 6.5.6.2).

6.5.6.1 Procedure *PC_RxInt*

When the *RXCIE0* bit is set, an interrupt will be generated when data is received from the PC. When an interrupt is generated, interrupt service routine *PC_RxInt* will be called when the PC has sent data. Procedure *Rx_Int* is identical to procedure *GPS_RxInt* which is described by Figure 6-16. Data is read directly from the UART0 data register (*UDR0*) and put into the software data buffer (section 6.5.3). By using Timer/Counter1 as described in section 6.5.2, it can be determined whether a complete PC data message has been received.

6.5.6.2 Procedure *PC_UDREmpty*

Procedure *PC_UDREmpty* handles transmission of data to the PC control program. This procedure gets data out of the software data transmission buffer and transfers it to the UART0 data register (*UDR0*) for transmission. Procedure *PC_UDREmpty* is called by the ‘UART0 data register empty’-interrupt – which is enabled by setting bit *UDRIE0* in the UART0 control register. This is done in procedure *StartTransmit* (section 6.5.6.4). When transmission of data has completed (thus no more data in software data transmission buffer), the UART data register empty interrupt is disabled and procedure *PC_UDREmpty* is not called again. Figure 6-23 shows the operation of procedure *PC_UDREmpty*.

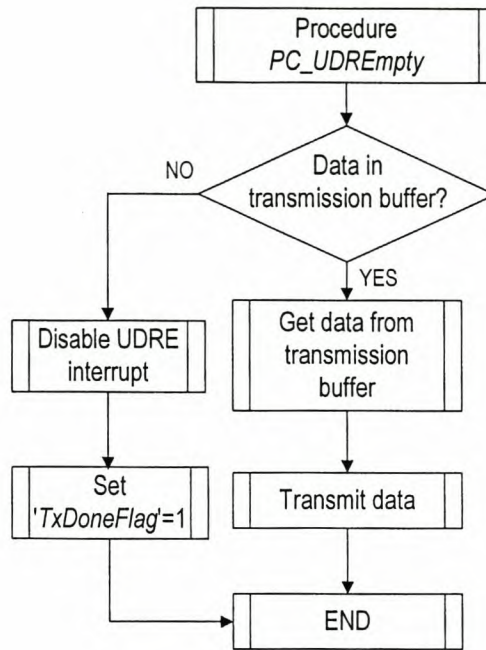


Figure 6-23 Interrupt Procedure *PC_UDREEmpty*

6.5.6.3 Procedure *TxByteToPC*

Procedure *TxByteToPC* puts data in the software data transmission buffer. It calculates the correct position index for the data and stores it at that position. For more information on the software data buffer see section 6.5.3.

6.5.6.4 Procedure *StartTransmit*

After data to be transmitted has been stored in the software data transmission buffer by procedure *TxByteToPC*, procedure *StartTransmit* enables the ‘UART0 data register empty’-interrupt. This procedure also clears the *TxDone* flag, which is polled in order to determine whether transmission has finished or not. This polling loop gives interrupt service routine *PC_UDREEmpty* chance to transmit data to the PC. Flag *TxDone* is set in procedure *PC_UDREEmpty* when transmission of all data is done, which terminates the polling loop. It should also be noted that CRC-16 calculation procedure is called out of procedure *StartTransmit*. Procedure *StartTransmit* is presented by Figure 6-24.

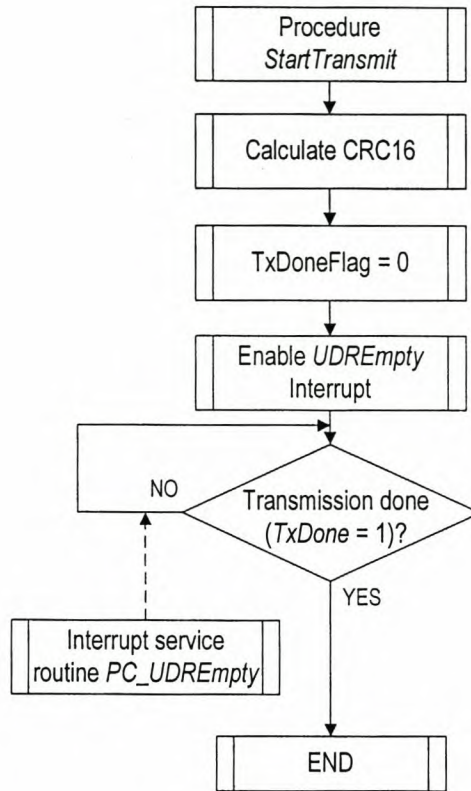


Figure 6-24 Procedure *StartTransmit*

6.5.7 Software UART Routines

The software described here can be used to implement a serial port on a microcontroller device with no available hardware UARTs. A typical device would be the *Atmel AT90S8515* used in an early prototype of this system. However, the software UART routines were not used in the final prototype of the developed system. It is presented here as additional software.

The theory of operation of this half-duplex (serial communication in one direction at a time) software UART was obtained from references [51]. This implementation needs most microcontroller hardware resources when it is running. In addition, the routines are not generic in terms of timing characteristics and implementation in other programming compilers such as Embedded Pascal, the compiler used in this project. These routines have however been implemented successfully in Embedded Pascal and detail on the implementation of the software UART can be found in Appendix F.

6.6 GPS Receiver Setup

When the GPS receiver is first powered-up after a long time, no navigational or time information is present in its RAM. This type of GPS receiver startup is called a cold start. The GPS receiver therefore needs time during which satellites are tracked, an almanac is downloaded and geographical position is calculated. Appendix A presents the WinOncore™ GPS Control Software program from Motorola. This software package is used to first configure the GPS receiver and monitor its status in terms of satellites tracked and position calculation. This package is then used to switch the GPS receiver to “polled only” mode, i.e. a mode where the GPS receiver only responds with data when it is requested. The default setting is to transmit complete navigational data once every second. This data, however, is not needed for this application. After the GPS receiver has acquired enough satellites and position and time is calculated, it is disconnected from the PC using the WinOncore™ GPS Control Software and connected to the developed system. Table 6-4 shows startup commands that the *GPS Based Time Stamping and Scheduling System* sends to the GPS receiver once it is connected to it. A full list of GPS receiver commands can be found in Appendix B.

Table 6-4 GPS startup commands

Binary Command	Function	Comment
@@ Aw	Set Time mode	GPS receiver responds on time request with UTC+GMT offset
@@ Ab	Set GMT offset	+02:00 for South Africa
@@Aa	Request time	GPS receiver responds with current time – system stores it to RTC
@@Ac	Request date	GPS receiver responds with current date

The following procedures implement the above mentioned commands.

6.6.1 Procedure *StartGPS*

Procedure *StartGPS* is described by Figure 6-25. Firstly the 1PPS signal interrupt is enabled (section 6.7.1) and the system waits for 5 seconds. This wait period is necessary when the GPS receiver is not powered up for the first time. This start-up is called a warm start, i.e. satellite navigation data and time information can be retrieved out of GPS Receiver RAM.

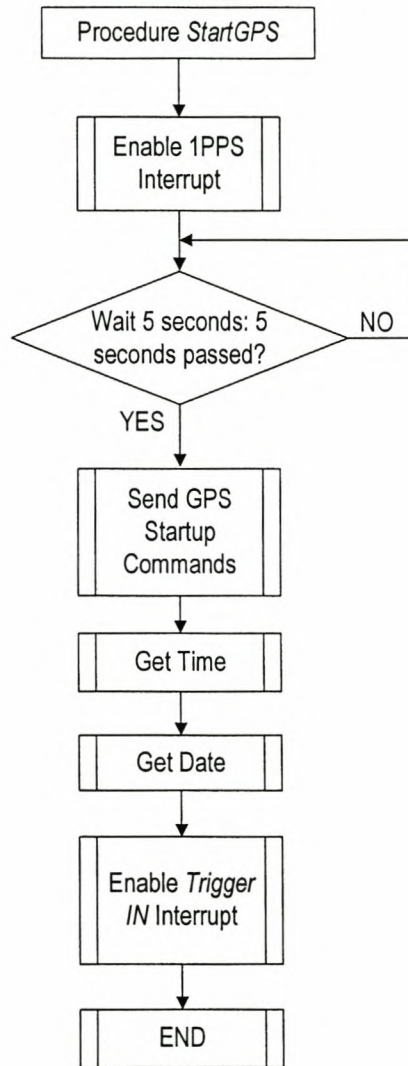


Figure 6-25 Procedure *StartGPS*

GPS receiver startup commands shown in Table 6-4 are sent, time and date are acquired and the *Trigger IN* interrupt is enabled. The system is now ready to receive, timestamp and store trigger signals.

6.6.2 Procedure *GPSReceiverSetup*

Figure 6-26 describes procedure *GPSReceiverSetup*. The GPS Receiver setup commands shown here was described in Table 6-4. All communication to the GPS receiver is handled by specific communications routines, including procedure *SendCommand*, described in section 6.5. The GPS receiver responds to these commands by echoing the command and the setting. This was discussed in section 6.5.

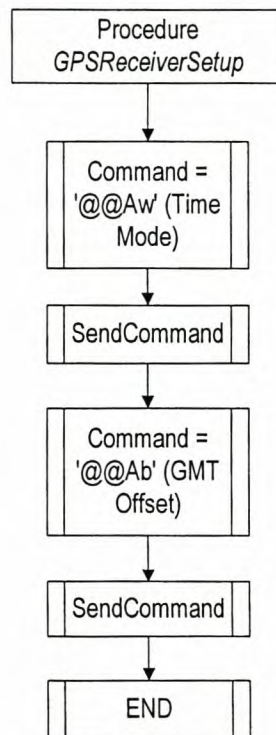


Figure 6-26 Procedure *GPSReceiverSetup*

6.6.3 Procedure *GetTime*

After the GPS receiver has been setup properly, the time is read from the receiver and the RTC is programmed with the time as described in chapter 4. Section 6.5.5.6 described that procedure *GetTime* has two options, *StoreToRTC* and *DontStoreToRTC*, which decided whether procedure *StoreTime* (section 6.5.5.6) will set the RTC or just transmit time data to the PC. When the PC control program (chapter 7) requests GPS time information, the system should not store that received time to the RTC. This command is for comparing the RTC time with the specific GPS receiver time. At the PC control program's or procedure *GPSReceiverSetup*'s specific request, procedure *GetTime* should store the GPS receiver time

to the RTC. Figure 6-27 describes procedure *GetTime*. This procedure writes time information to the RTC by using procedure *WriteData* as was described in sections 4.2 and 6.3.

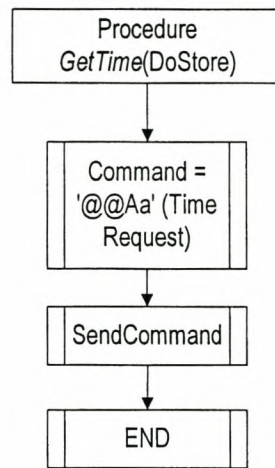


Figure 6-27 Procedure *GetTime*

6.6.4 Procedure *GetDate*

After time has been acquired from the GPS receiver, the date is read and stored in variables in the system control program. The date is included with every timestamp. Figure 6-28 shows this procedure.

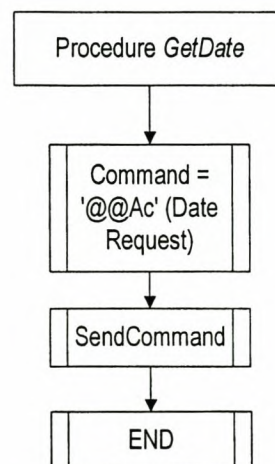


Figure 6-28 Procedure *GetDate*

6.7 Interrupt Service Routines

As shown in Figure 6-3, three possible interrupts might occur. They are the 1PPS interrupt, the PC command interrupt and the trigger received interrupt. An interrupt is an event that occurs outside of normal program flow and these events usually need to be attended to immediately by using Interrupt Service Routines.

6.7.1 1PPS Interrupt

The 1PPS interrupt service routine is used to make sure that the RTC is correct at all times, by resetting the RTC with GPS time once every hour. It is also used at system start up to ensure a warm-start (section 6.6.1) of the GPS receiver. Figure 6-29 shows the service routine operation.

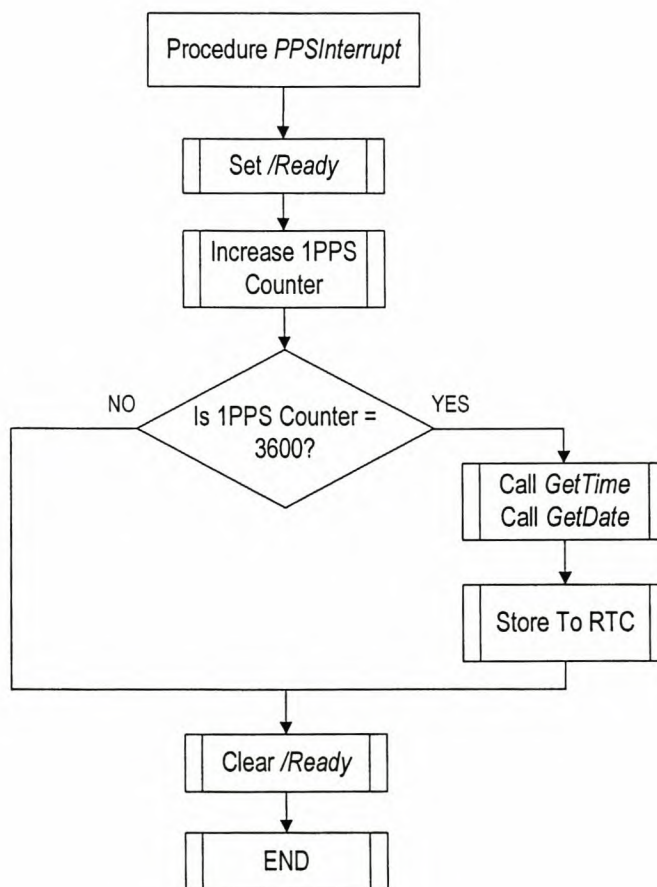


Figure 6-29 Interrupt service routine *PPSInterrupt*

When the service routine is called once every second, the system is set in 'not-ready' mode. The system is not ready to accept any triggers when the RTC is updated once every hour

because communications with the GPS receiver is being carried out. The system is ready to accept triggers when this routine has finished.

6.7.2 PC Command Interrupt

The *GPS Based Time Stamping and Scheduling System* may be controlled by a PC control software program as described by chapter 7. The PC control program transmits a command number, data (if any), and the system control program responds on these commands by transmitting the command number echo and applicable data (if any). A typical PC control message and system response is shown in Figure 6-30.

PC Control Command:

Command Number	Data Bytes (if any)	CRC Low Byte	CRC High Byte
----------------	---------------------	--------------	---------------

GPS System Response:

Command Number	Data Bytes (if any)	CRC Low Byte	CRC High Byte
----------------	---------------------	--------------	---------------

Figure 6-30 PC Control command and response message

The PC command interrupt handler routine is shown in Figure 6-31. When a command is received from the PC control program an interrupt is generated and procedure *ServiceComms* is called when a complete command has been received. If the CRC on the data is correct, the command byte is read from the PC UART receive buffer and a case statement decides what procedure needs to be called to service the particular command. When the command has been carried out, the PC UART0 data receive buffer is cleared to ensure correct reception of the next control command.

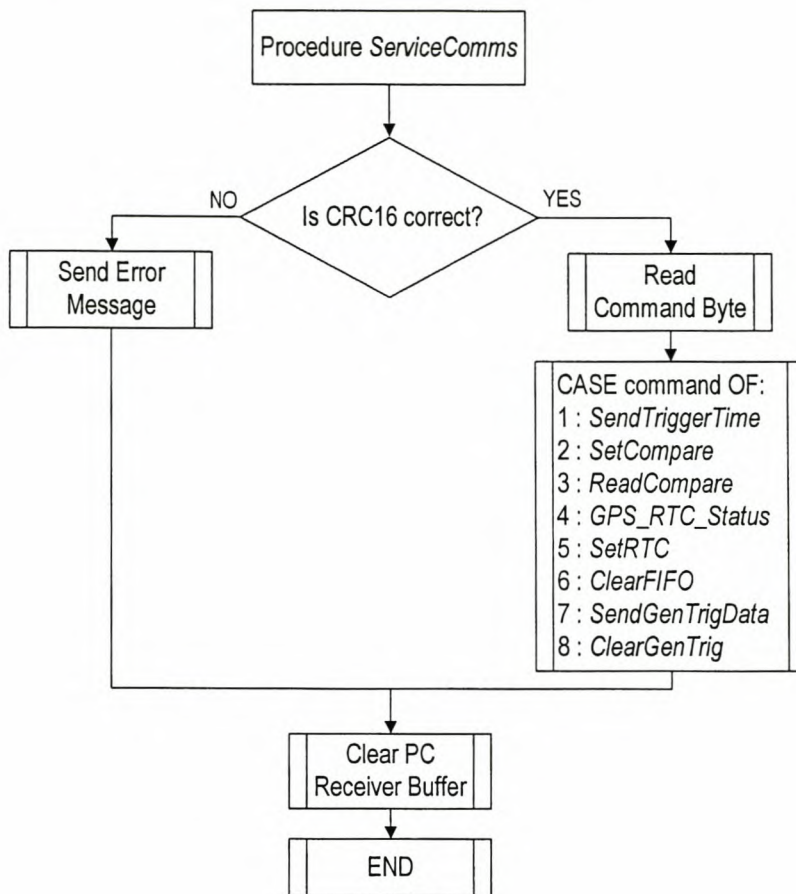


Figure 6-31 PC Command interrupt handler routine

All the different control commands in the above case statement are now discussed.

6.7.2.1 Command *SendTriggerTime*

Command *SendTriggerTime* sends all trigger-received data to the PC control program for display. All trigger information is stored in the FIFO memory, as well as in a software data buffer (discussed in section 6.5.3). Figure 6-32 shows a *repeat*-loop filling up the transmit buffer with trigger data. When trigger data is ready to be transmitted, transmission starts, as discussed in section 6.5.6, and data is sent to the PC. The trigger data buffer tail pointer is reset so that all trigger data can be retransmitted if needed.

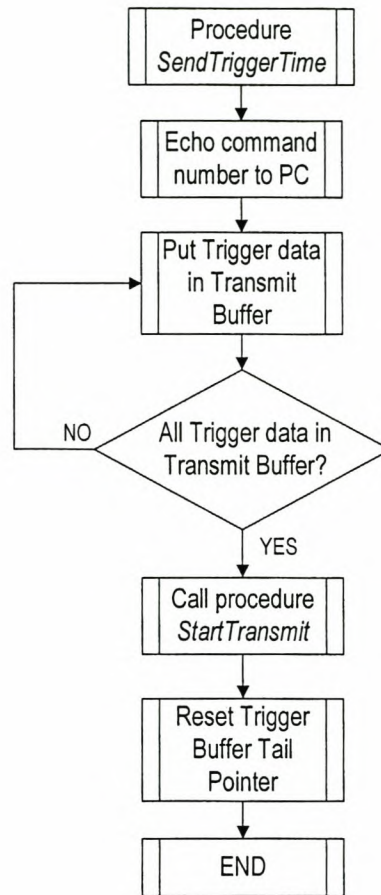


Figure 6-32 Procedure *SendTriggerTime*

6.7.2.2 Command *SetCompare*

Command *SetCompare* sets the RTC and Microsecond Counter compare registers with the desired value. A $1\mu\text{s}$ -long trigger pulse will then be generated when the pre-programmed time comes. As mentioned, this signal can be used to start pre-programmed data acquisition runs on power systems. This procedure also validates the programmed time. If the time is invalid, i.e. already past or out-of-range, an error signal is returned to the PC control program. Figure 6-33 shows procedure *SetCompare*.

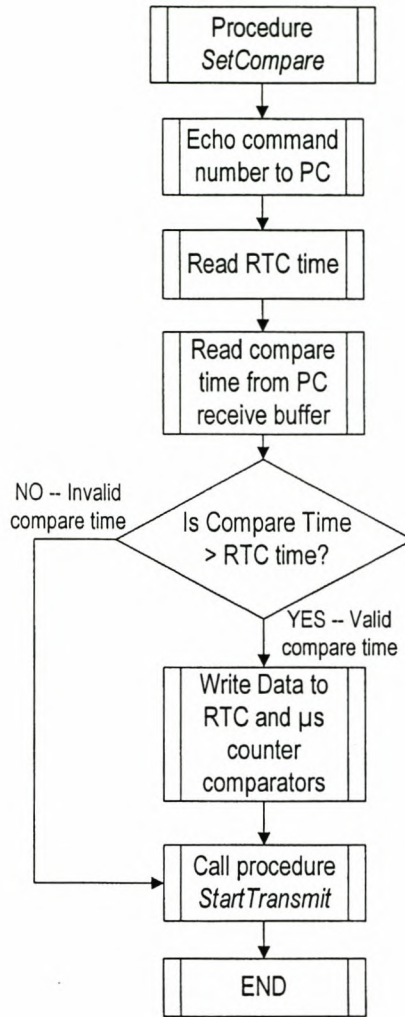


Figure 6-33 Procedure *SetCompare*

6.7.2.3 Command *ReadCompare*

Command *ReadCompare* enables the PC control program user to view the current comparator pre-programmed time, which was programmed by procedure *SetCompare*. This procedure reads the variables of hours, minutes, seconds and microseconds that was stored with the last time program of the comparators and transmits it to the PC control program. Figure 6-34 presents this procedure.

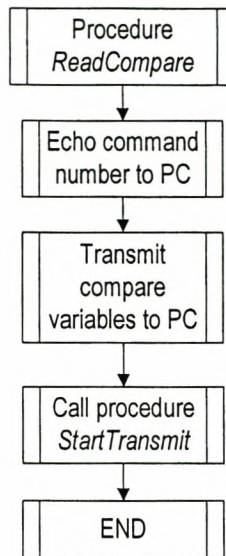


Figure 6-34 Procedure *ReadCompare*

6.7.2.4 Command *GPS_RTC_Status*

Command *GPS_RTC_Status* enables the user to check on the status of the system. The RTC time, GPS time and RTC error which is RTC time subtracted from the GPS time, are status elements that need to be monitored. The system microcontroller reads these values, computes the RTC error and transmits it to the PC control program as shown in Figure 6-35.

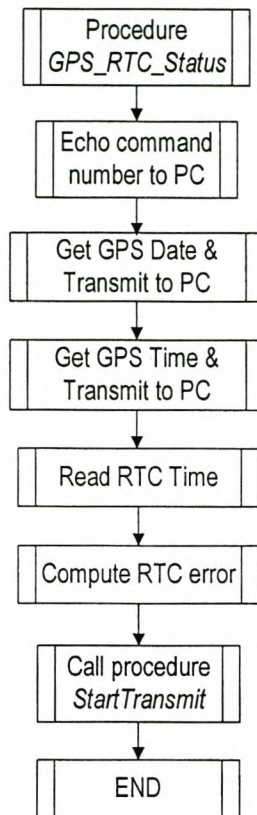


Figure 6-35 Procedure *GPS_RTC_Status*

6.7.2.5 Command *SetRTC*

When the PC control program reports that the RTC has lost or gained time, the RTC needs to be reprogrammed with the correct GPS time. Command *SetRTC* reprograms the RTC with the GPS time on command. Procedure *SetRTC* is called to perform this operation and is presented by Figure 6-36.

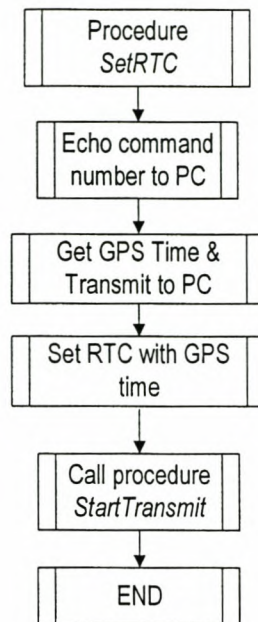


Figure 6-36 Procedure *SetRTC*

6.7.2.6 Command *ClearFIFO*

When all trigger data has filled up the FIFO memory, it has to be cleared. Command *ClearFIFO* clears the FIFO memory by reading the memory until no data is left in it. The FIFO memory is discussed in section 5.6. Procedure *ClearFIFO* is presented by Figure 6-37.

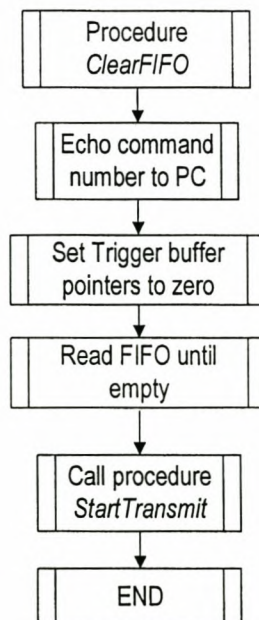


Figure 6-37 Procedure *ClearFIFO*

6.7.2.7 Command *SendGenTrigData*

Every time a pre-programmed trigger signal was generated, the time is stored in a software data buffer. This data may be read by the PC control program. When command *SendGenTrigData* is received, procedure *SendGenTrigData* sends the contents of this software data buffer to the PC control program. Procedure *SendGenTrigData* is presented by Figure 6-38.

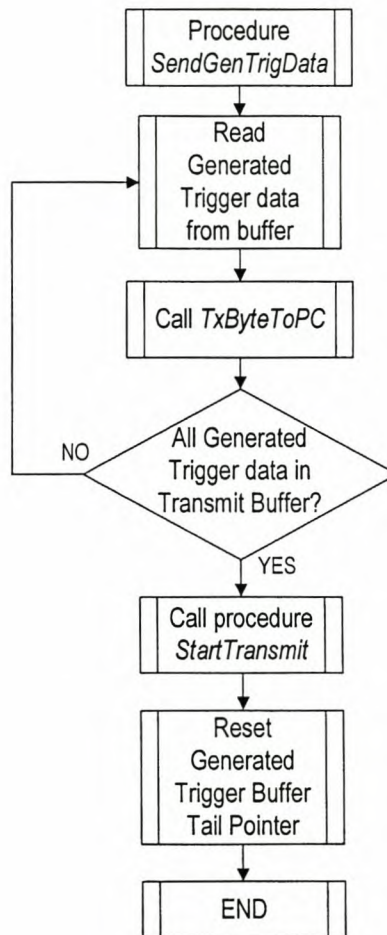


Figure 6-38 Procedure *SendGenTrigData*

6.7.2.8 Command *ClearGenTrigData*

When all generated trigger data has been viewed or downloaded by the PC control program, it might be necessary or desired by the user that the software data buffer containing this data is cleared. Procedure *ClearGenTrigData* does this. These software data buffers are cleared by resetting the head and tail pointers to zero. Figure 6-39 shows the operation of procedure *ClearGenTrigData*.

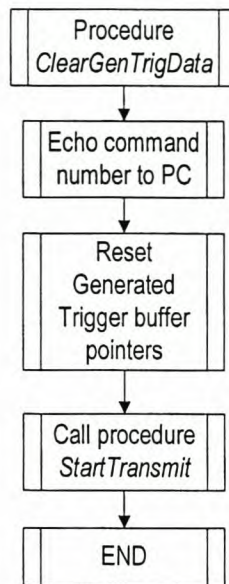


Figure 6-39 Procedure *ClearGenTrigData*

6.7.3 Trigger Received Interrupt

When a trigger signal is received as shown in section 5.3.5, an external interrupt is generated in the microcontroller. The interrupt service routine *TrigEvent* reads the trigger timestamp from the RTC and Microsecond Counter and stores it in a software data buffer. A stored trigger message contains the trigger number and time, i.e. hours, minutes, seconds and microseconds. Procedure *StoreTrigData* will now be discussed and thereafter procedure *TrigEvent*.

6.7.3.1 Procedure *StoreTrigData*

Procedure *StoreTrigData* stores trigger received data in a software data buffer every time a trigger interrupt is generated and timestamp data is available for storage. As mentioned in the previous section, a stored trigger message contains the trigger number and time (hours, minutes, seconds and microseconds). This data is all stored in a software data buffer. For more information on the operation of the software data buffer, consult section 6.5.3.

6.7.3.2 Procedure *TrigEvent*

Procedure *TrigEvent* is called when a trigger interrupt signal (section 5.3.5.2) is received. The system is set to 'not-ready'-mode, to block any triggers that might be received during downloading of trigger timestamp data. A trigger counter is kept to number all incoming

trigger timestamp data. By using procedure *ReadData* (section 6.3) the timestamp data is read from the RTC and Microsecond Counter EPLDs. This data is then stored in a FIFO memory and software data buffer (by procedure *StoreTrigData*) for later retrieval by the PC control program, or external compatible hardware. After this storing of trigger timestamp data, the system is once again ready for the time stamping of new trigger signals.

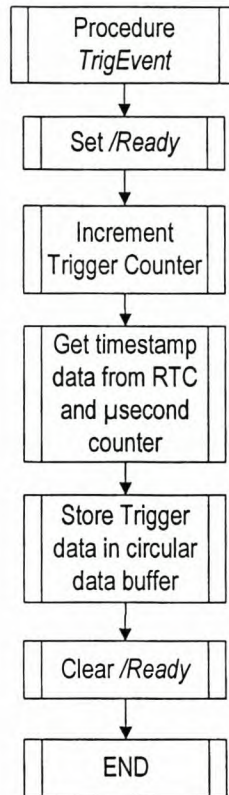


Figure 6-40 Procedure *TrigEvent*

6.8 Conclusions

This chapter discussed the hardware embedded system microcontroller main control program in detail. The software discussed here was implemented in *Embedded Pascal for the AVR* [48], which is a high-level Pascal [49] compiler for the *Atmel AVR* series microcontrollers. The system communications interface and system response to commands received by the host computer control program (chapter 7) was also discussed. The program has proved to be an effective controller of the different blocks, namely RTC, Microsecond Counter and communication peripherals, used in this system. Some of the most important control and data bus signals were captured by a PC based logic analyser and they are presented in chapter 8.

Chapter 7

LabVIEW™ Control Software

7.1 Introduction

A host computer¹ may be used to control the *GPS Based Time Stamping and Scheduling System*. The host computer control program is implemented using National Instruments' LabVIEW™ virtual instrument software. Most commonly used software routines, communication peripherals and functions exist in the LabVIEW™ programming toolbox and it is up to the user to connect these blocks together to create the desired software. Firstly, section 7.2 will discuss the LabVIEW™ programming environment. Section 7.3 will present the CRC-16 calculation routine (also called a virtual instrument or VI). Finally, sections 7.4, 7.5 and 7.6 will present the *GPS Based Time Stamping and Scheduling System* host computer control program. Appendix H presents complete host computer control software diagrams.

7.2 LabVIEW™ Programming Environment

The LabVIEW™ programming environment consists of a control panel window and a program diagram window. The control panel window contains controls such as buttons, LED indicators and string/number inputs. The program diagram window contains “behind-the-scenes” graphical programming that uses the controls contained in the control window to perform certain user defined functions. For example, the process that has to be carried out when a certain button is pressed is defined in the program diagram window. Figure 7-1 shows the control panel window with the LabVIEW™ control palette, with an OK-button control inserted. Different controls can be dragged and dropped into this window to create the program user interface. The LabVIEW™ programming environment will be used in section 7.3 to create a program that calculates the CRC-16 of a text string. This program is used

¹ Or a Personal Computer (PC)

extensively in the *GPS Based Time Stamping and Scheduling System* host computer control program communication routines.

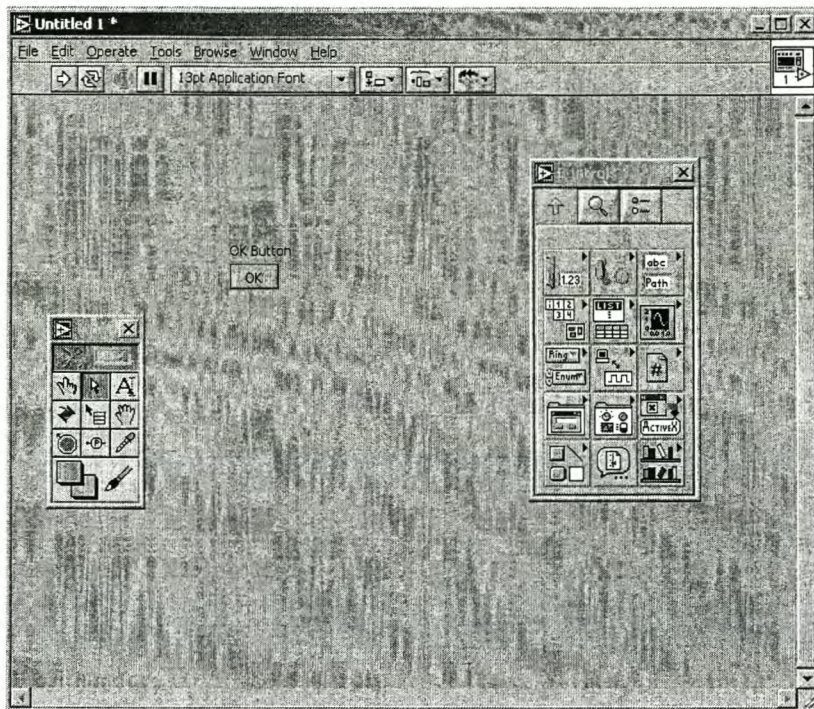


Figure 7-1 LabVIEW™ control window, tools and control palette

Figure 7-2 shows the program diagram window with the LabVIEW™ functions palette.

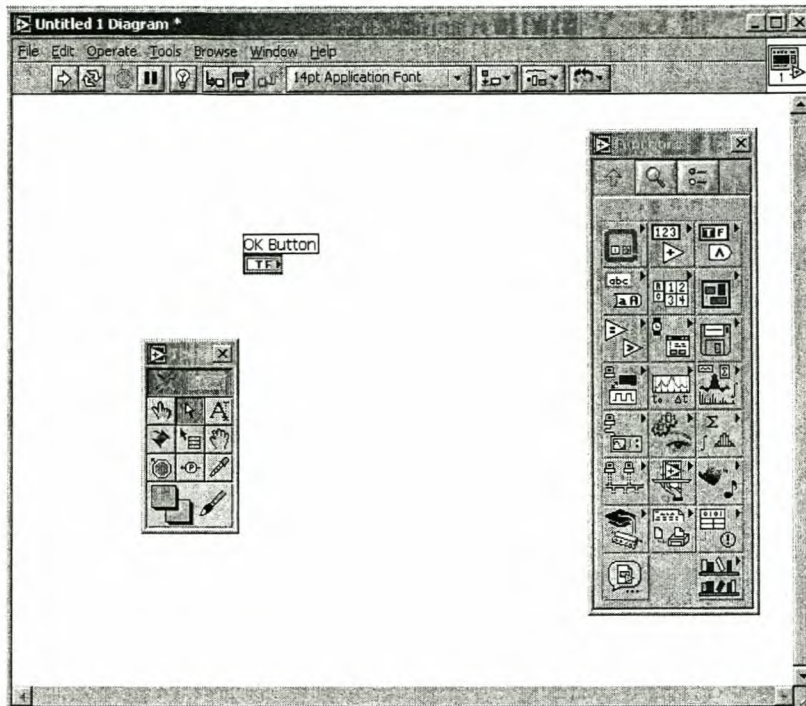


Figure 7-2 LabVIEW™ diagram window, tools and functions palette

7.3 Example: The CRC-16 VI

The CRC-16 VI was created in LabVIEW™ as an implementation of the CRC-16 routine described in section 6.5.4.2 by Figure 6-13 (page 86). Figure 7-3 shows the CRC-16 VI control window. The CRC-16 of *Test String* is calculated and shown in *CRC16*, *CRC Low* and *CRC High*.

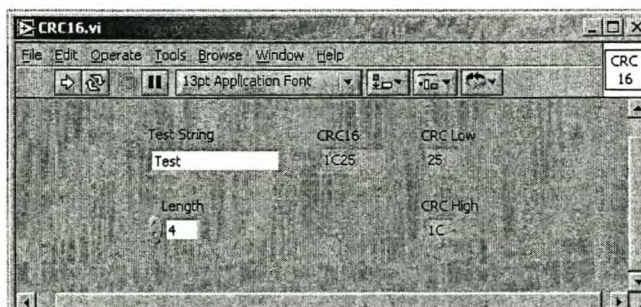


Figure 7-3 CRC-16 VI user interface

Figure 7-4 shows the CRC-16 program diagram. Program flow is from left to right in this VI. The controls can be seen on the left of the diagram and the outputs (CRC16, CRC High and CRC Low) on the right.

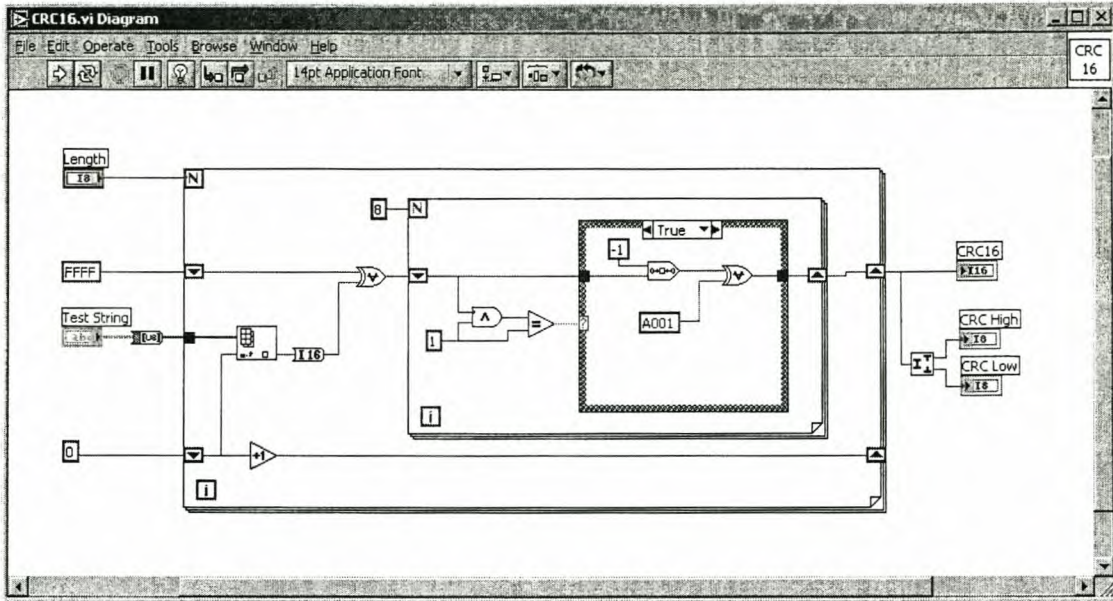


Figure 7-4 CRC-16 program diagram

This VI can be compiled and used as a sub-VI in another LabVIEW™ VI. The CRC-16 VI is used as a sub-VI in the *GPS Based Time Stamping and Scheduling System* main control program as part of the communications interface. When data is sent or received, the CRC-16 is calculated to check for transmission errors.

7.4 Host Computer Control Program High-Level Discussion

As mentioned in chapter 6, the host computer control program has the following functions:

- Read trigger received (*Trigger IN*) data from the GPS Based Time Stamping and Scheduling System and display it in a drop-down list. This information can then be used in applications such as described in chapter 2.
- Read information on triggers previously generated (*Trigger OUT*) and display it in a drop down list.
- Set up time when pre-programmed trigger signal (*Trigger OUT*) must be generated.
- Obtain GPS receiver and RTC status and display it.
- Set the RTC.

- Clear FIFO memory and on-board software data buffers.

Figure 7-5 shows a high-level flow diagram of the host computer control program operation.

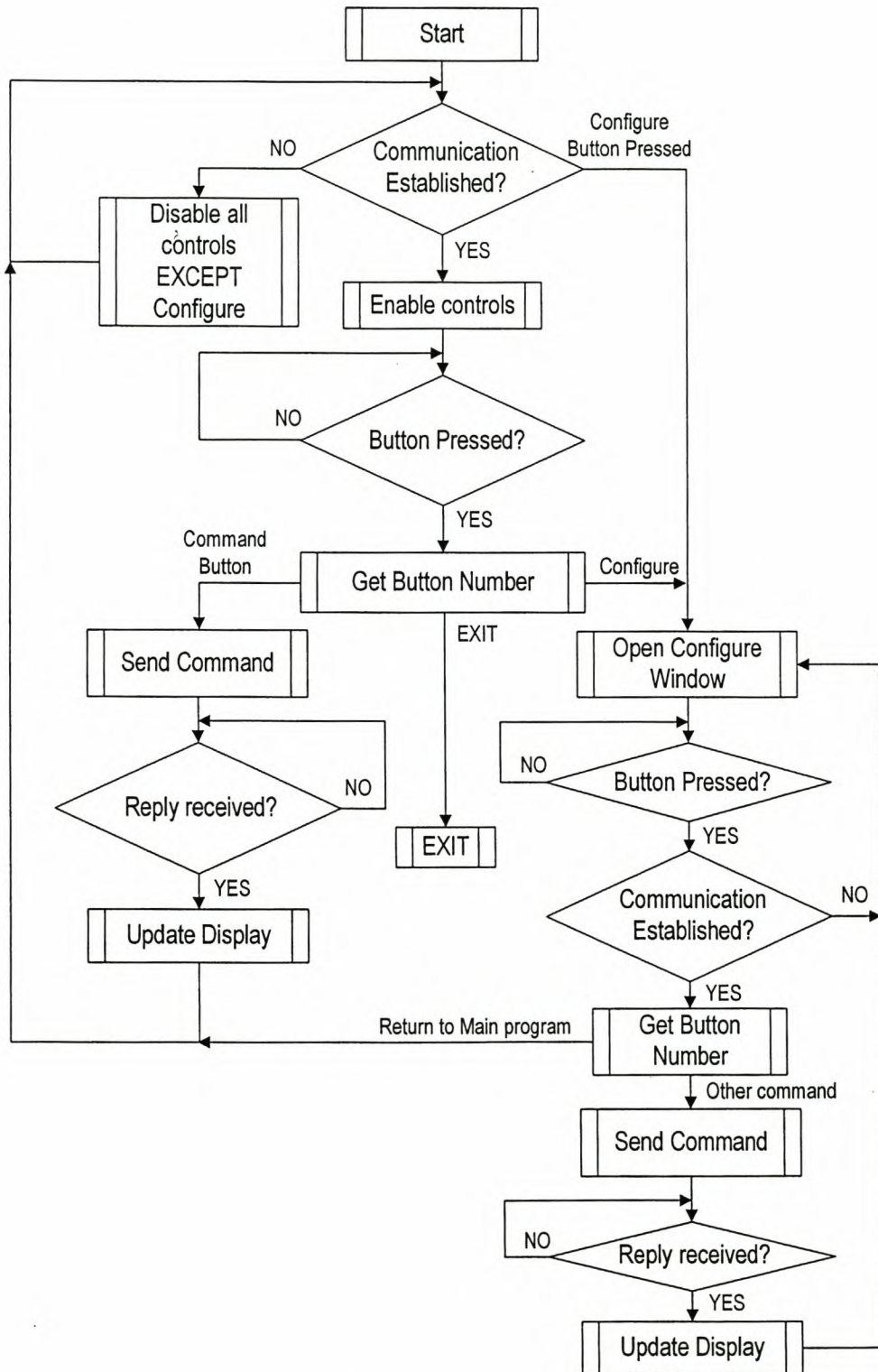


Figure 7-5 Host Computer control program flow diagram

Figure 7-5 will become more clear when the main control panel is discussed.

7.5 Main Control Panel

The main control panel is the GUI¹ of the system. When communication has not been established with the *GPS Based Time Stamping and Scheduling System*, the control buttons are disabled, as shown in Figure 7-6.

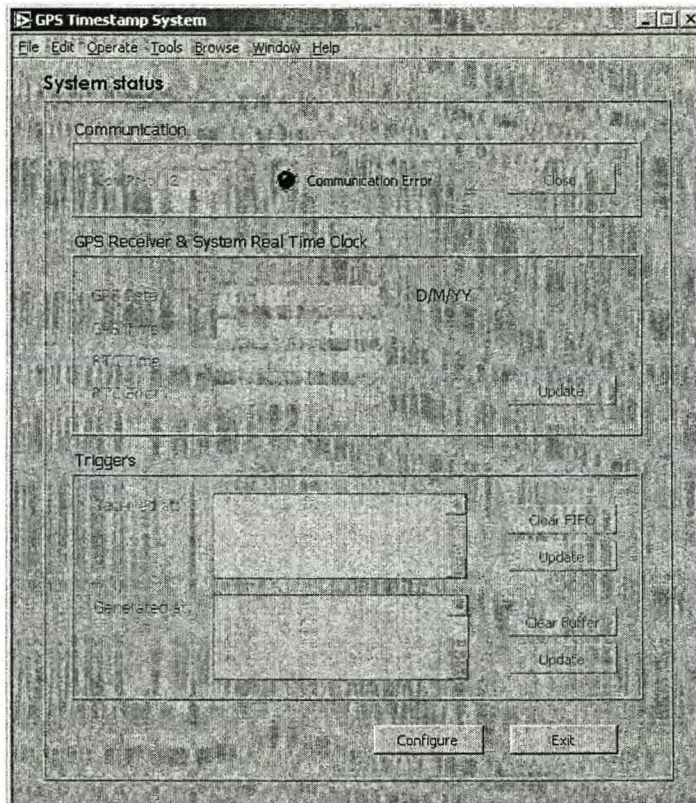


Figure 7-6 Main control program controls disabled

If a valid connection was established, the main control program is ready to send commands and display relevant data. This is presented in Figure 7-7. This figure shows the communications port number to which the system is connected, the GPS/RTC status message, received trigger information and previously generated trigger information. When a communications error occurs, the LED in the communication box will light, notifying the user that an error occurred.

¹ Graphical User Interface

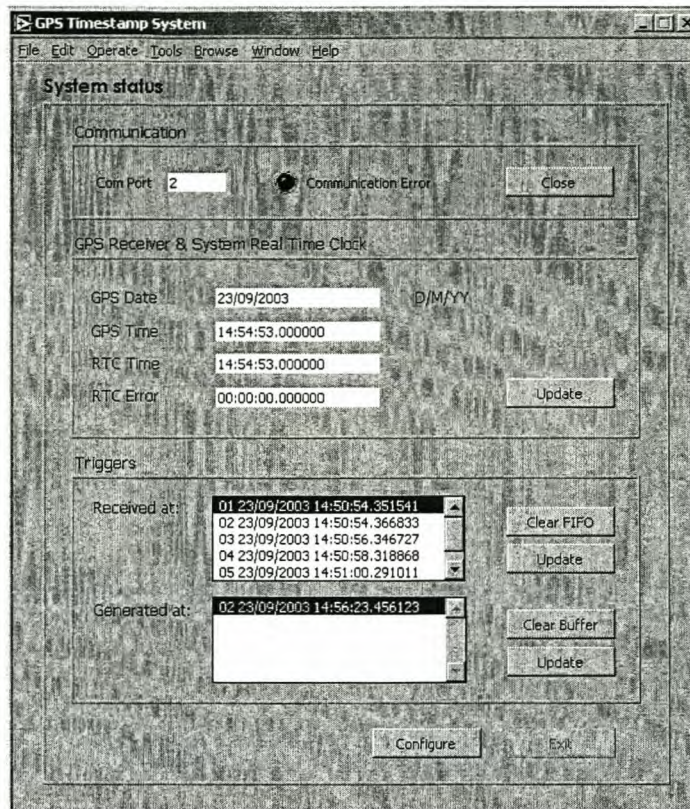


Figure 7-7 Main control program normal operation

The configuration panel that opens with a click of *Configure*, is discussed next.

7.6 Configuration Panel

In the configuration panel, it is possible for the user to select a valid communications port and establish a connection. When a connection could not be established, an error LED will light. It is also possible for the user to set the RTC with GPS time when the main control panel GPS/RTC status message reported a RTC error. The time when a trigger signal must be generated is programmed here. When an invalid time, i.e. already past, or out of range, was typed, an error LED will light. Figure 7-8 shows the configuration panel with disabled controls, meaning that a connection was not established with the system. When a valid connection to the system has been established, the controls are enabled and the user may proceed as shown in Figure 7-9. The window returns the user to the main control panel when *OK* is clicked.

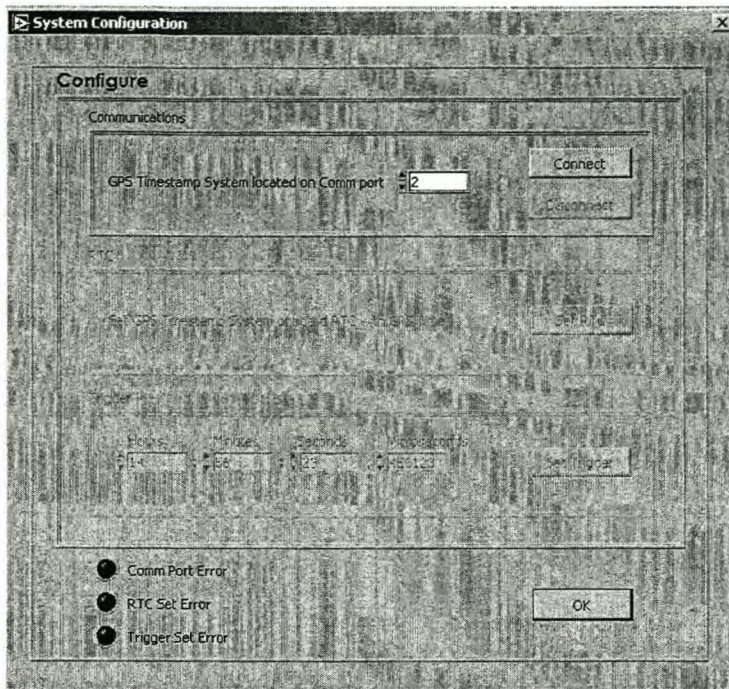


Figure 7-8 Configuration panel with disabled controls

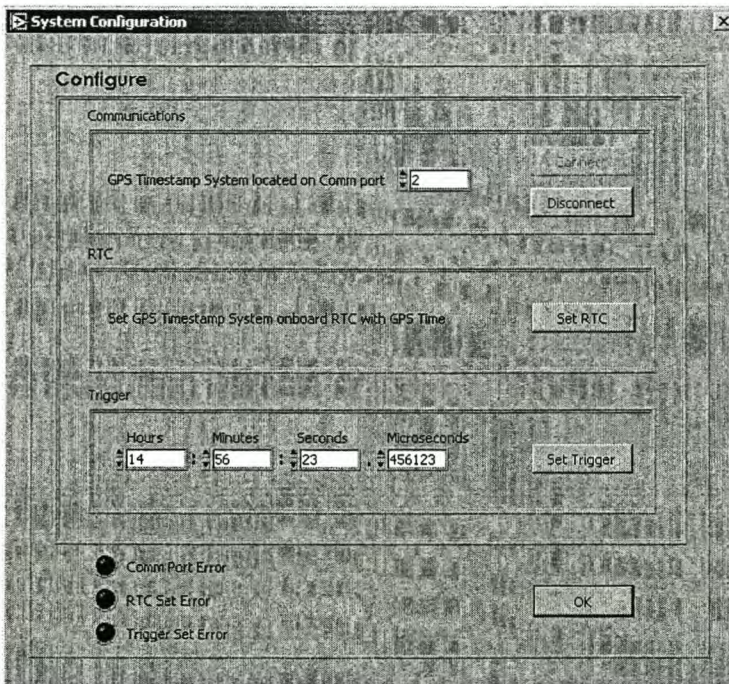


Figure 7-9 Configuration panel with enabled controls

The complete LabVIEW™ implementation diagrams can be found in Appendix H.

7.7 Conclusion

In this chapter the host computer control program was discussed. National Instruments LabVIEW™ was an effective programming tool because peripheral functions and other procedures are already available and ready to be used in user defined software. The host computer control program proved to be an integral and effective part of the *GPS Based Time Stamping and Scheduling System*.

Chapter 8

Results and Recommendations

8.1 Introduction

The previous chapters presented detail design of the *GPS Based Time Stamping and Scheduling System*. These chapters also presented simulation waveforms of the respective systems. More simulation waveforms (*Altera MAX+PLUS* software) can be found in Appendices. This chapter presents actual results obtained by measurements and practical laboratory implementation of the developed system.

8.2 Test Setup and Procedures

The developed system was tested extensively with the help of a PC based logic analyser and the PC control program developed in chapter 7. As can be seen in Figure 2-9, the FIFO output data bus can only be read from the external interface control bus. A test circuit was created to emulate an external data acquisition system and to supply power to the *GPS Based Time Stamping and Scheduling System*. This test circuit is called the *Backplane* because of the way it is physically connected to the system. This backplane was used to read trigger information out of the FIFO memory and send it to a PC for testing purposes. The backplane was also used to connect *Trigger IN* to *Trigger OUT* (section 8.4) for testing purposes. Complete program code and circuit diagrams can be found in Appendix G.2 and I.

8.3 Logic Analyser Signal Captures

After the laboratory setup was done, system control signals were captured using a PC-based logic analyser. The logic analyser GUI is shown in Figure 8-1. This logic analyser is manufactured by *Jobmatch* and it can sample up to 16 channels at 200MHz, with 128 kilosamples per channel. It transfers sampled data to the PC via a parallel port where it can be saved to disk or viewed in the Logic Analyser software GUI.

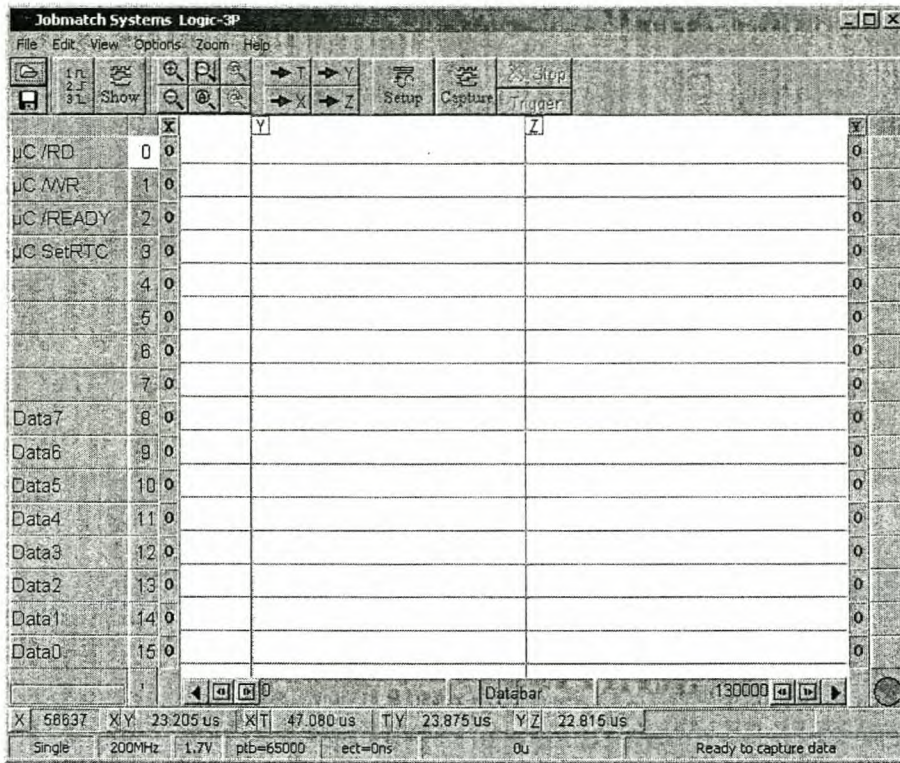


Figure 8-1 Logic Analyser GUI

Some captured signals of different system operations will now be presented.

8.3.1 RTC Set operation

RTC set operation was discussed in section 4.3.2.2. The logic analyser was used to capture waveforms during actual system operation. Figure 8-2 shows three $/WR$ assertions, one for hours (X), minutes (Y) and seconds (Z) respectively, and then $SetRTC$ is pulsed to set the RTC with the correct time. The system data bus is presented by the eight $Data$ signal captures.

8.3.2 RTC and Microsecond Counter Read operation

RTC and Microsecond Counter read operation was discussed in section 4.3.2.3 and section 5.3.6 respectively. Figure 8-3 shows six $/RD$ assertions to read hours, minutes, seconds and microseconds, which consists of three bytes, from the RTC and Microsecond Counter respectively.

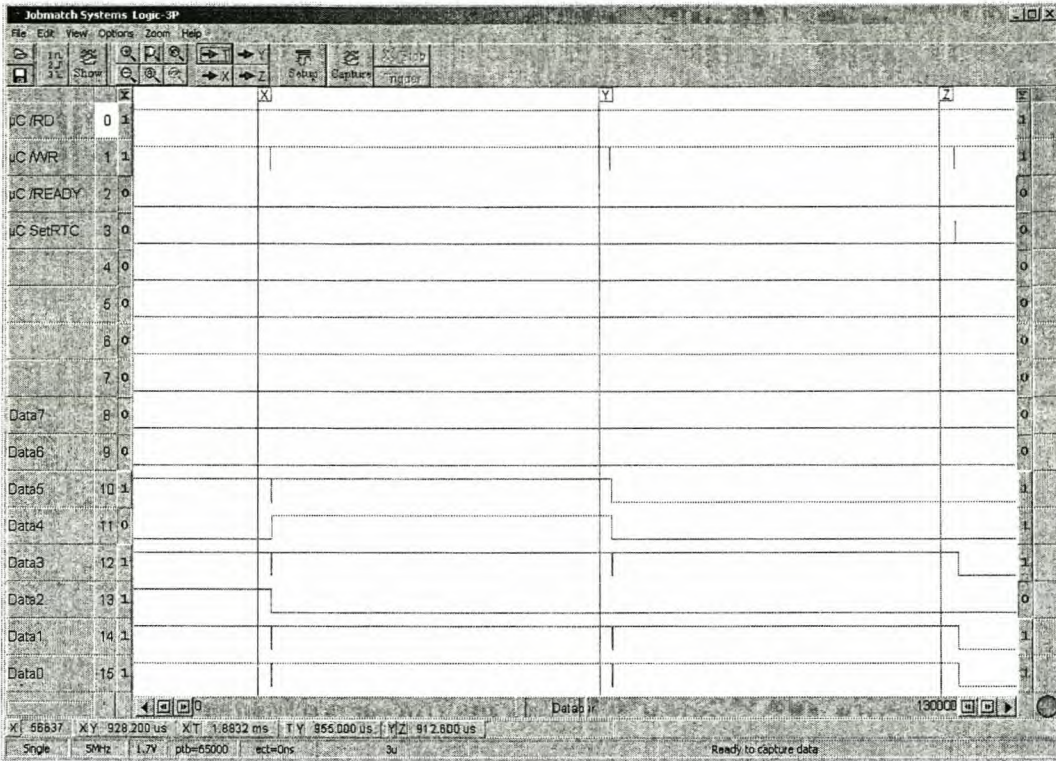


Figure 8-2 Signal capture during RTC set operation

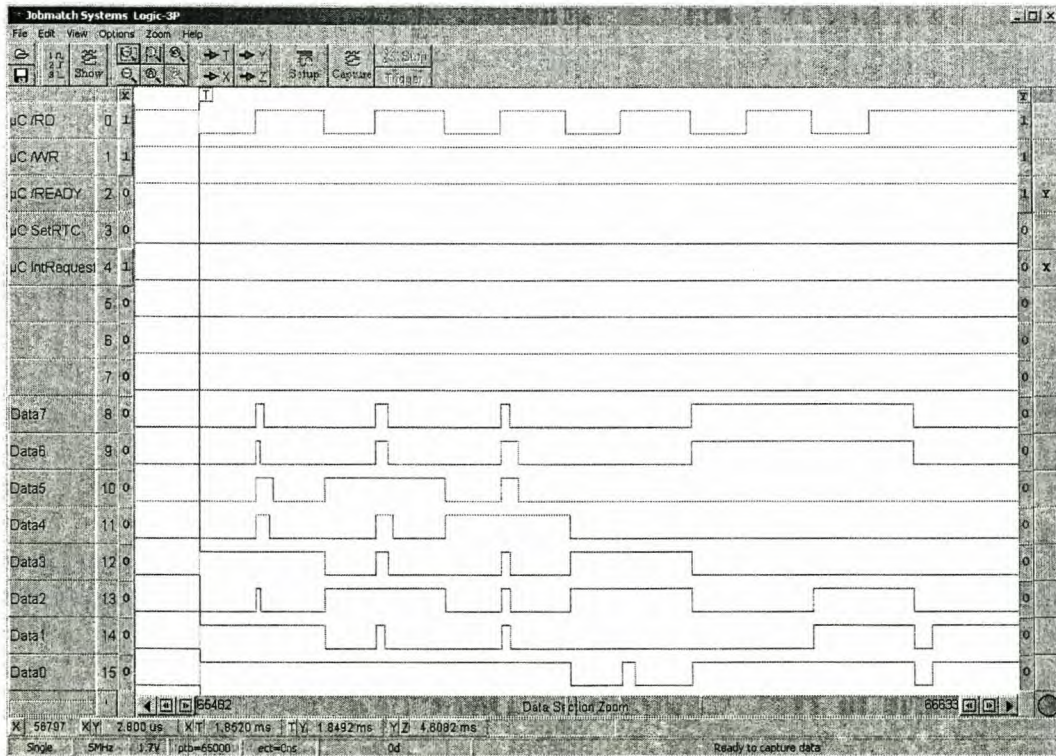


Figure 8-3 RTC and Microsecond Counter read operation

8.3.3 Trigger Interrupt Operation

The trigger interrupt signal *IntRequest* was discussed in section 5.3.5.2. Figure 8-4 shows signal *IntRequest* being generated and the system microcontroller asserting */Ready* to signal that no more triggers may be accepted until the current trigger data is downloaded into the system microcontroller and stored in the FIFO memory.

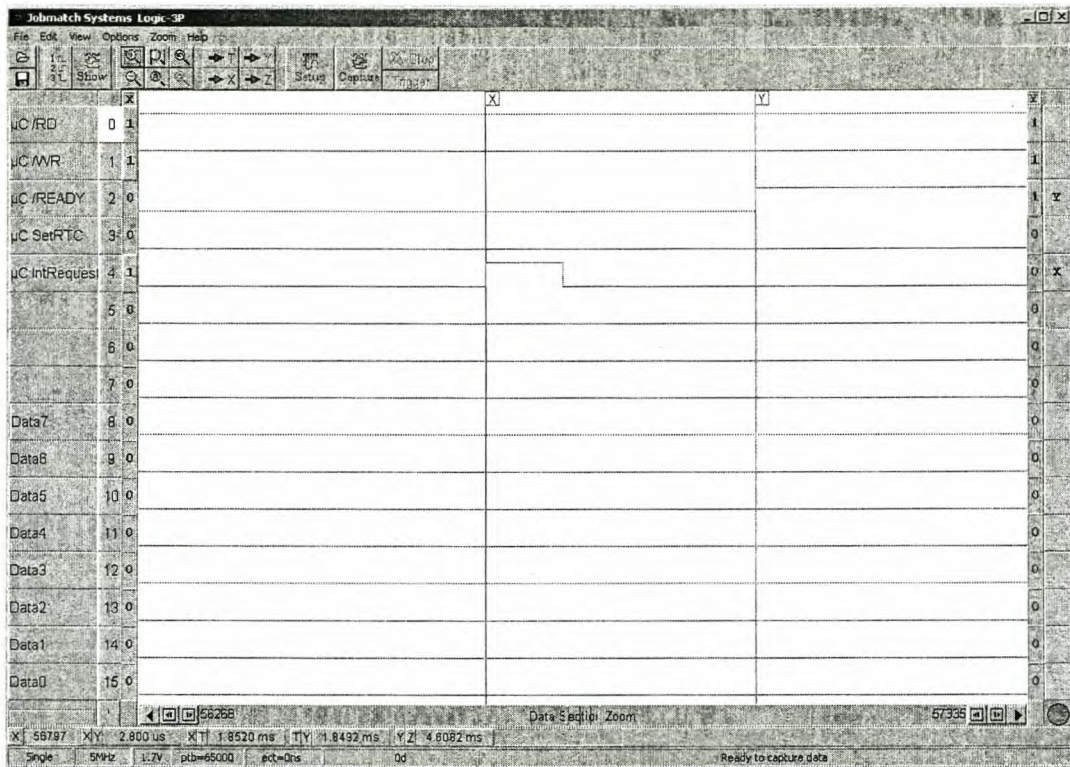


Figure 8-4 Signals *IntRequest* and */Ready*

8.3.4 FIFO Write and Read Operation

FIFO memory operation was discussed in section 5.6. Figure 8-5 and Figure 8-6 shows FIFO memory write and read operation respectively. The Empty and Full Flag, i.e. */EF* and */FF*, operation is clear. The Empty Flag is high when FIFO memory is not empty and low when no data is present in the FIFO memory. The Full Flag stays high until the FIFO memory is full.

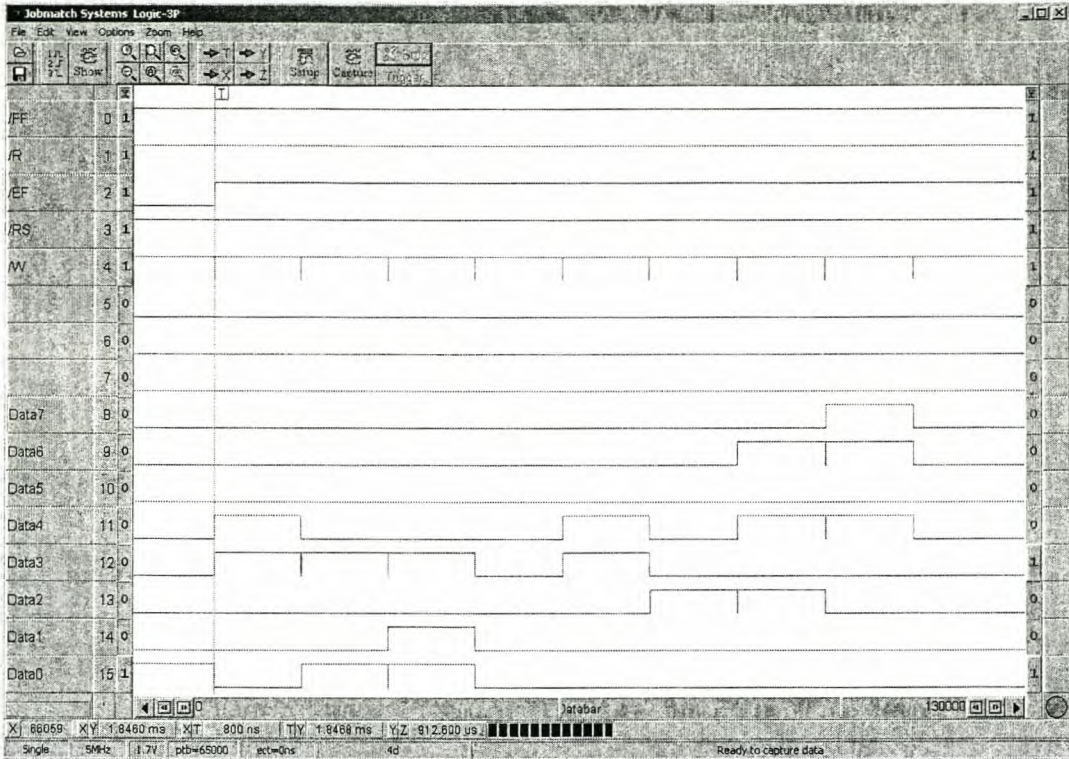


Figure 8-5 FIFO memory write operation

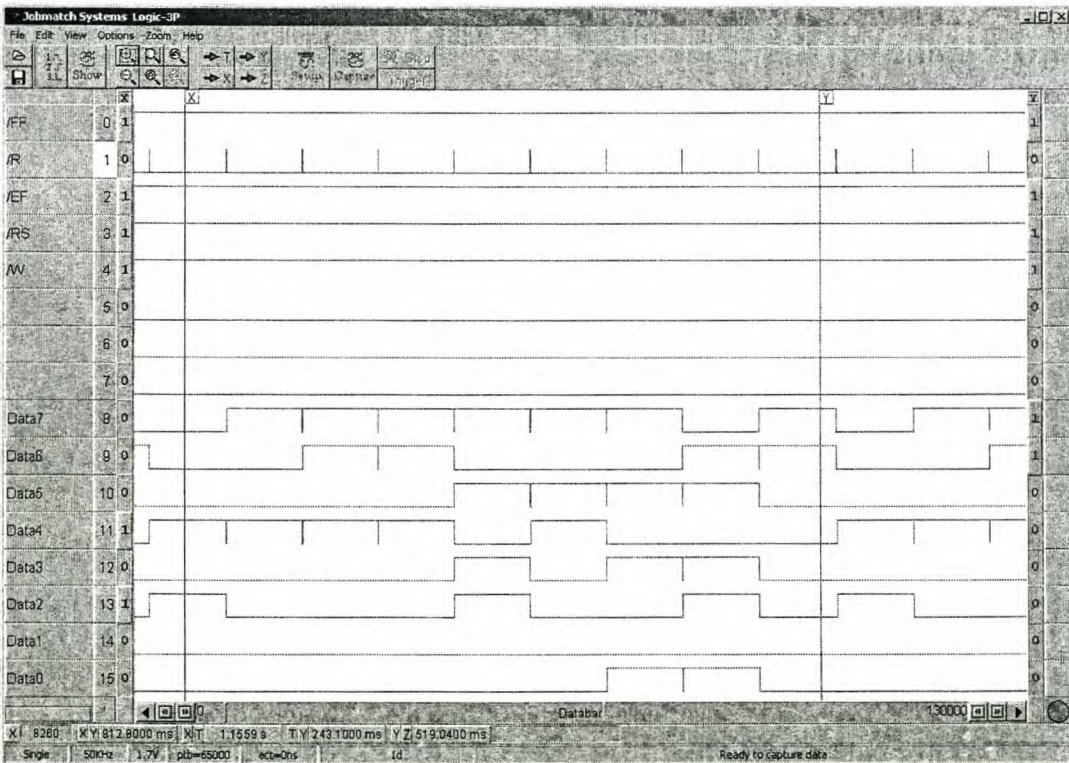


Figure 8-6 FIFO memory read operation

8.4 LabVIEW™ PC Control Program

The PC control program was discussed in chapter 7. One test has however not been presented in that chapter. This test consists of connecting the *Trigger IN* signal to the *Trigger OUT* signal. A trigger signal is then generated at a pre-programmed time and the *Trigger IN* data is then viewed. If the system operates correctly, the time of the received trigger should be the same as the time of the generated trigger signal. Figure 8-7 shows that this is indeed the case. It can thus be concluded that the system operation is exactly as planned.

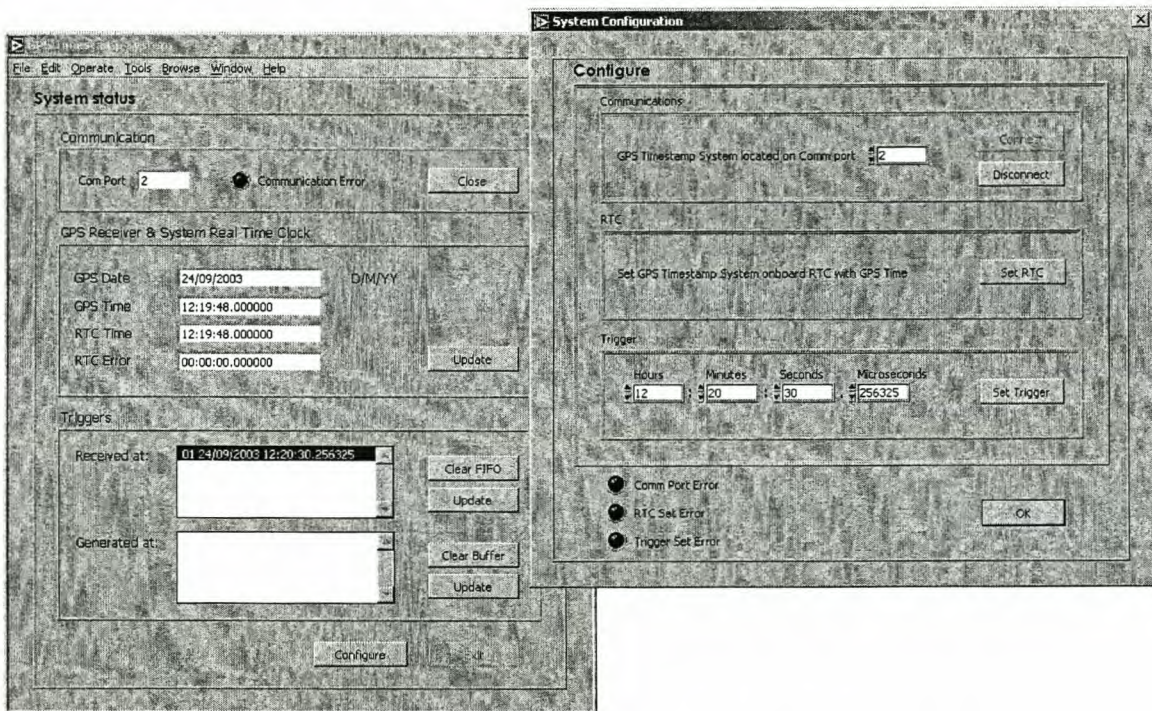


Figure 8-7 *Trigger IN* time is exactly that of *Trigger OUT*

For more details on the LabVIEW™ PC control program, consult chapter 7, and Appendix H.

8.5 Oscilloscope Measurements

An oscilloscope was connected to a test output that represented the 1MHz signal, i.e. *1MHzClk*, that serves as a clock signal for the Microsecond Counter. The waveform obtained is presented in section 5.3.3 and by Figure 8-8.

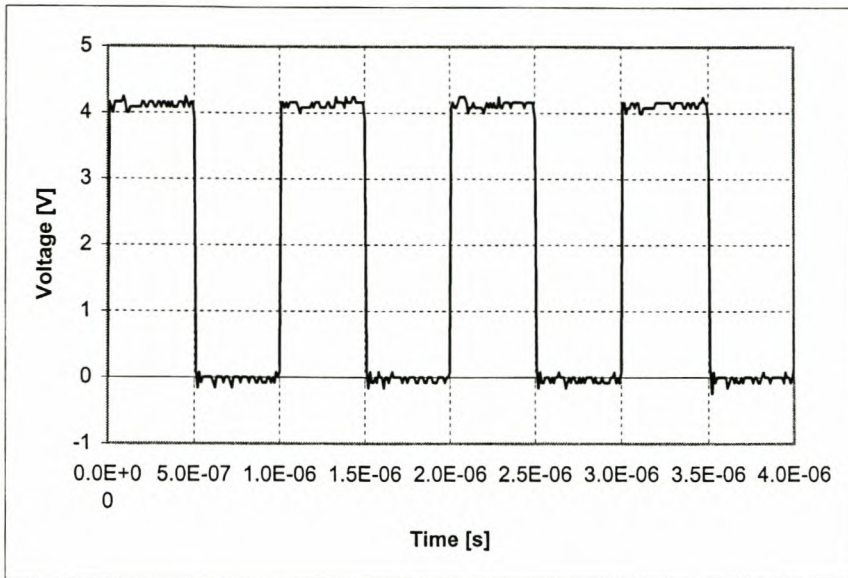


Figure 8-8 1MHzClk Microsecond Counter clock signal

Another waveform was captured by use of an oscilloscope, namely the *Trigger OUT* signal generated at a pre-programmed time, accurate to $1\mu\text{s}$. Consequently, this signal (Figure 8-9) is $1\mu\text{s}$ wide and can be used to start pre-programmed data acquisition runs on power systems.

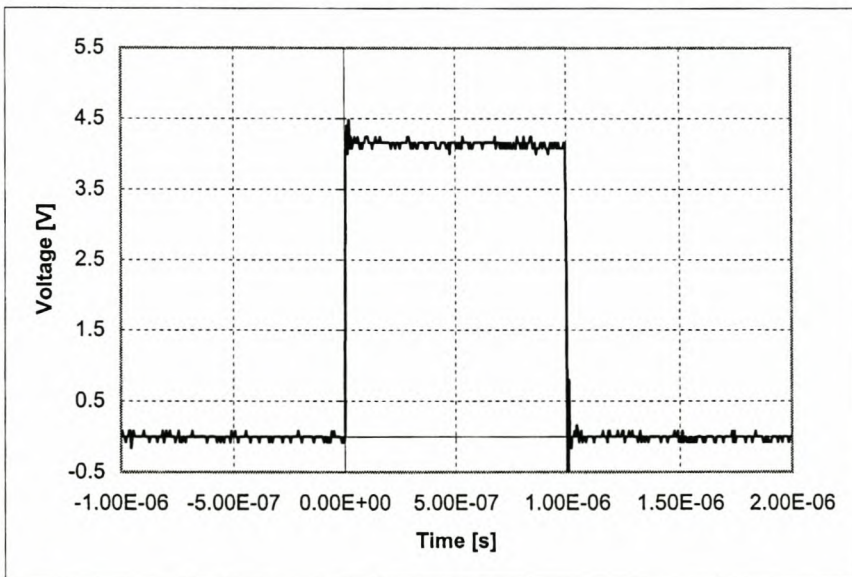


Figure 8-9 Generated trigger (*Trigger OUT*) signal

8.6 Conclusions and Recommendations

This section will discuss conclusions of the tests and measurements presented in this chapter, as well as an overall overview. Finally, some recommendations are made.

8.6.1 Results and Measurements

This chapter presented some measurements and practical test results. These results have shown that the *GPS Based Time Stamping and Scheduling System* achieved the design goals laid out in chapter 1 and 2. This chapter showed that an accurate 1MHz signal could be generated and used as a clock signal for a Microsecond Counter (section 8.5). Other results have shown that a given trigger signal can easily be time stamped and a 1 μ s-wide trigger signal can just as easily be generated at any time (sections 8.4 and 8.5). Chapter 7 and section 8.4 concluded that the PC control program proved to be an effective system control interface.

8.6.2 Final System Overview

Chapter 1 and 2 recognised a need for a stable clock signal that may be used to synchronise data time tagging information to a common time standard for a variety of applications in power systems. On these power systems it has always been recognised that the value of the information gathered by disturbance recorders, sequence-of-event recorders and SCADA¹ terminals would be greatly enhanced if they could ‘time-tag’ their measurements to a common time standard.

Chapter 3 showed that the launch of GPS satellites made these extremely accurate timing signals available all around the world, for a variety of purposes. Many commercial users, who require accurate time or position information for navigation or geographical mapping, make use of the GPS service because it is recognised as being extremely reliable and secure, since military and other vital systems depend on its availability and accuracy. By installing low-cost GPS receivers in power system substations, synchronised pulsing is available for applications such as time tagging of data collected by any system monitor or initiation of pre-programmed data acquisition runs [52].

¹ Supervisory Control And Data Acquisition

Chapter 4,5 and 6 provided design details of the developed *GPS Based Time Stamping and Scheduling System*. Simulations proved that the chosen topology is effective and accurate. Design goals presented in chapter 2 have thus been achieved. Chapter 7 discussed design of a host computer control interface and results presented in chapter 7 and 8 were satisfactory. A system topology has thus been created that is relatively cheap compared to other existing systems. The created prototype system provides an accurate (to $1\mu\text{s}$) time-tag of a received trigger signal. A $1\mu\text{s}$ wide trigger pulse can also be generated at a pre-programmed time.

8.6.3 Recommendations

Some recommendations can be made in light of the results and measurements obtained.

8.6.3.1 Local System Clock

The fundamental time stability of the local system clock must be consistent with accuracy specifications. The accuracy specifications for the system clock drift demand that the clock should not lose or gain more than $1\mu\text{s}$ every second (1PPM). These high accuracy specifications for local clocks can be difficult to meet. The trade-off will be between cost and stability. Local oscillators will typically be quartz crystals. As mentioned in chapter 2, quartz crystals typically drift due to thermal, mechanical, and aging effects. Of these, thermal effects are the most difficult to handle effectively. For example, a typical thermal specification for uncompensated crystals is 1PPM per degree Celsius. Without synchronisation, a one-degree temperature rise over an interval of 2 seconds will produce an error of roughly two microseconds. The thermal environment of the crystal would need to be controlled to this level. Accuracies in the tens of nanosecond range therefore imply that some combination of better thermal specifications on the crystal, reduced synchronisation interval, or better thermal management combine to reduce the thermal drift by two orders of magnitude [41]. A reduced synchronisation interval can be obtained by implementing a GPS receiver similar to the Motorola UT+ Oncore™. This GPS receiver provides a software selectable 100PPS signal. This means that an on-board system clock may be synchronised 100 times a second, instead of only once every second (as in this prototype system).

Crystals become increasingly expensive below a specification of 1PPM/degree. Control of the thermal environment must be carefully managed particularly in high accuracy implementations. Very long clock-coasting times typically require oven-controlled crystals or

the use of more stable oscillators, as is recommended for this project. Thermal drift during short clock-coasting intervals (1 second) can often be managed by attention to heat dissipation in surrounding devices, cooling patterns within the node, increasing the thermal mass of the oscillator or similar techniques [41].

8.6.3.2 System Memory

In this prototype, a FIFO memory was used to store trigger information. This memory can only be read from the external interface control bus by compatible hardware. It is proposed that the FIFO memory output bus also be connected to the 8-bit system data bus via a hardware buffer or latch. This will enable the system microcontroller to read the data out of FIFO memory and transmit it to a PC for analysis or storage. In this prototype system, trigger information is only available for transmission to a PC because of a software data buffer that is implemented in the system microcontroller. This data will be lost when a system power failure occurs. After such a power failure, the FIFO can only be read by external compatible hardware, and not by the system microcontroller.

8.6.3.3 System Communications Interface

The developed system communicates with a host computer (or PC) via a RS-232 serial data link. It would also be possible to integrate Universal Serial Bus (USB) technology with the developed system. The system topology can be changed in such a way that a PC would see the developed system as a memory device containing data. This data can then be easily downloaded and used in PC system control software. The USB technology would also enable the system to be connected to a universal bus containing other similar equipment, like compatible data acquisition systems. These interconnected systems can then easily be identified by each other and applicable data may be exchanged.

The system topology lends itself to be a stand-alone system installed at remote substations. It would be impractical to visit every such substation to download acquired data. Consequently a need exists to contact the *GPS Based Time Stamping and Scheduling System* from a remote location and download data. A couple of options are available, including Wireless Application Protocol (WAP) and modem technology. A user would typically dial the modem connected to the system at a remote substation and download data. WAP technology would enable a user to dial a cellphone modem connected to the developed system and visit the

system website to view or download data. The system might also be configured to dial a central server to automatically download newly acquired data.

References

- [1] John. G. Webster, "Fault location", Wiley Encyclopaedia of Electrical and Electronics Engineering, Dept of Electrical and Computer engineering, University of Wisconsin-Madison, Wiley Interscience, John Wiley & Sons, Inc.
- [2] J.G. Kappenman, M.E. Gordon and T.W. Guttormson, "High precision location of lightning-caused distribution faults", Proceedings of the 2001 IEEE/PES Transmission and Distribution Conference and Exposition, Atlanta, Georgia.
- [3] Working Group H-7 of the Relaying Channels, subcommittee of the IEEE Power System Relaying Committee, "Synchronized sampling and phasor measurements for relaying and control", IEEE Transactions on Power Delivery, Vol. 9, No.1, January 1994, pp. 442-449.
- [4] Fault Location Working Group of the IEEE, "IEEE Guide for Determining Fault location on AC Transmission and Distribution Lines", Draft 3, 7 May 1999.
- [5] P.A. Crossley and P.E. McLaren, "Distance protection based on travelling waves", University of Cambridge, IEEE Transactions on Power Apparatus and systems, Vol. PAS-102, No.9, September 1983, pp. 2971-2983.
- [6] Thompson Adu, "A new transmission line fault locating system", IEEE Transactions on Power Delivery, Vol. 16, No.4, October 2001, pp. 498-503.
- [7] R.E. Wilson, "Methods and uses of precise time in power systems", IEEE Transactions on Power Delivery, Vol.7, No.1, January 1992, pp.126-132.
- [8] M. Kezunovic and B. Perunicic, "Automated transmission line fault analysis using synchronized sampling at two ends", IEEE transactions on Power Delivery, Vol. 11, No.1, February 1996, pp. 441-447.
- [9] W. Zhao, Y.H. Song and W.R. Chen, "Improved GPS travelling wave fault locator for power cables by using wavelet analysis", Electrical Power and Energy Systems Research, Elsevier Science, 23 (2001), pp. 403-411.

- [10] M.B. Dewe, S. Sankar and J. Arrilaga, "The application of satellite time references to HVDC fault location", *IEEE Transactions on Power Delivery*, Vol.8, No.3, July 1993, pp. 1295-1302.
- [11] M. Kezunovic, J. Mrkic and B. Perunicic, "An accurate fault location algorithm using synchronized sampling", *Electric Power Systems Research*, Elsevier Science, 29 (1994), pp. 161-169.
- [12] A. Mousa and H. Lee, "GPS travelling wave fault locator systems: Investigation into the anomalous measurements related to lightning strikes", *IEEE Transactions on Power Delivery*, Vol. 11, No.3, July 1995, pp. 1214-1223.
- [13] J. Bullock, T. King, L. Kennedy, E. Berry and G. Zanfino, "Test results of a low cost GPS receiver for time transfer applications", *IEEE Frequency Control Symposium*, Orlando, Florida, 1997.
- [14] G.B. Gilcrest, G.D. Rockefeller and E.A. Udren, "High speed distance relaying using a digital computer. Part I system description." *IEEE Transactions on Power Apparatus and Systems*, Vol. PAS-91, no.3, May/June 1972, pp.1235-1243.
- [15] J.S. Thorp, A.G. Phadke, S.H. Horowitz and M.M. Begovic, "Some applications of phasor measurements to adaptive protection", *IEEE Transactions on Power Systems*, Vol. 3, No.2, May 1988, pp. 791-798.
- [16] A.G. Phadke, J.S. Thorp and N.J. Karimi, "Real time voltage measurements for static state estimation", *IEEE Transactions on PAS*, Vol. PAS-104, No.11, November 1985, pp. 3098-3104.
- [17] P. Bonanomi, "Phase angle measurements with synchronised clocks – Principle and applications", *IEEE Transactions on Power Apparatus and Systems*, Vol. PAS-100, No.12, December 1981, pp. 5036-5043.
- [18] C. Rehtanz and J. Bertsch, "Wide area measurement and protection system for emergency voltage stability control", *IEEE PES Winter Meeting*, Panel Session on "Emergency Voltage Stability Control", New York, Jan. 2002

- [19] K.E. Martin, "Phasor measurements at the Bonneville Power Administration", Power systems and communications infrastructure for the future, Beijing, China, September 2002.
- [20] C. Rehtanz, M. Zima, M. Kaba and J. Bertsch, "System for wide area protection, control and optimisation based on phasor measurements", Power systems and communications infrastructure for the future, Beijing, China, September 2002.
- [21] A.R. Katancevic, "Electromechanical oscillations and advanced protection applications – summary", S-18.156 Power systems engineering special assignment, Helsinki University of Technology, Espoo, Finland, April 2002.
- [22] Hewlett-Packard Company, "Accurate Transmission line fault location using synchronised sampling", Application note 1276-1, 1996.
- [23] S. Vasilic and M. Kezunovic, "An improved neural network algorithm for classifying the transmission line faults", IEEE-PES winter meeting, New York, January 2002.
- [24] M. Kezunovic, S. Vasilic and F. Gul-Bagriyanik, "Advanced approaches for detecting and diagnosing transients and faults", MED Power 2002, Athens, Greece, November 2002.
- [25] X. Xu, M. Kezunovic, "Automated Feature Extraction from Power System Transients Using Wavelet Transform", PowerCon 2002, Kunming, China, October 2002.
- [26] M. Kezunovic, B. Perunicic, "Synchronized Sampling Improves Fault Location". IEEE Computer Applications in Power Vol. 8, No. 2, April 1995.
- [27] M. Kezunovic, "Power System Monitoring Using Intelligent Techniques and Synchronized Sampling", IFAC Symposium on Control of Power Plants and Power Systems, Cancun, Mexico, December 1995.
- [28] G. Benmouyal, E. Schweitzer and A. Guzman, "Synchronized phasor measurement in protective relays for protection, control, and analysis of electric power systems", 29th annual Western Protective Relay Conference, Spokane, Washington, 24 October 2002.
- [29] C. T. Nguyen, M. Kezunovic and T. E. Dy-Liacco, "Application of Microprocessors in Power System Control and Analysis", IFAC Proceedings Series, No. 4, Pergamon Press, 1985, pp. 2293-2295.

- [30] M. Kezunovic and B.D. Russell, "Microprocessor Applications to Substation Control and Protection", IEEE Computer Applications in Power Vol. 1, No. 4, October 1988, pp. 16-20.
- [31] IEEE Std. 1344-1995 (R2001), "IEEE Standard for Synchrophasors for Power Systems", 12 December 1995
- [32] J. Jespensen and J. Fitz-Randolph, "From sundails to atomic clocks: understanding time and frequency", New York, NY, Dover publications, 1982.
- [33] P. Daly, I.D. Kitching and D.W. Allen, "Frequency and time-stability of GPS and GLONASS clocks", Forty-fourth annual symposium on Frequency control, Baltimore, MD, May 1990.
- [34] M. King, M. Miranian and D. Busch, "Test Results and analysis of a low cost core GPS receiver for time transfer applications", Proceedings of the ION National Technical Meeting, 1994.
- [35] J. Geier, M. King, H. Kennedy and R. Thomas, "Prediction of time accuracy and integrity of GPS timing", IEEE Proceedings on Frequency and Control, 1995.
- [36] U.S. Department of defence, "Global positioning service general specification", 2nd Edition, June 2, 1995.
- [37] B.G. Blazer, "GPS receiver operation", Global Positioning System, Washington, DC, The Institute of Navigation, 1980.
- [38] GARMIN Corporation, "GPS Guide for beginners", December 2000. Website: www.garmin.com.
- [39] Motorola Inc., Oncore GPS user's guide; Revision 3.2; June 1998.
- [40] J.A. Jodice, "Time synchronous end-to-end relay testing", Second conference on Precise Time and Frequency in power systems, Fort Collins, CO, September 28-29, 1987.
- [41] IEEE Std. 1588-2002, "IEEE Standard For a Precision Clock Synchronization Protocol For Networked Measurement and Control Systems", 8 November 2002.
- [42] Hugo Fruehauf, "Precision Oscillator Overview", Zyfer Inc, December 2001.

- [43] G.B. Ancell and N.C. Pahalawatha, "Maximum likelihood estimation of fault location on transmission lines using travelling waves", IEEE Transactions on power delivery, No. 9, pp. 680-689, 1994.
- [44] J.F. Wakerly, "Digital Design: Principles and practices", Second Edition, Prentice-Hall Inc, 1994.
- [45] Integrated Device Technology, Inc, "IDT7200/7201A/7202A CMOS Asynchronous FIFO", www.idt.com, December 2002.
- [46] Atmel Corporation, "8-bit AVR Microcontroller with 16k-bytes of In-System Programmable Flash – Atmega161 and Atmega161L", www.atmel.com, 2002.
- [47] Altera Corporation, "MAX7000 Programmable Logic Device Family", www.altera.com, Revision 6.01, July 1999.
- [48] Rainier Lamers, "Embedded pascal: A structured high-level language with a low level soul", User Manual, July 2002.
- [49] Michael Yester, "Using Turbo Pascal", Que Corporation, 1989.
- [50] Ross N. Williams, "A painless guide to CRC error detection algorithms", Ver. 3, Rocksoft Pty Ltd, ross@guest.adelaide.edu.au, 19 August 1993.
- [51] Atmel Corporation, "Half duplex interrupt driven software UART", Application note 304, www.atmel.com, August 1997.
- [52] B.J. Cory and P.F. Gale, "Satellites for power system applications", IEEE Power Engineering Journal, pp. 201-206, October 1993.
- [53] F.B. Siebrits, "Field Implementation of a Transient Voltage Measurement Facility Using High-Voltage Current Transformers", M.Sc.Eng Thesis, University of Stellenbosch, December 2003.

Appendix A

WinOncore™ GPS Control Software

Motorola have provided software to control the GPS receiver from a personal computer. This software was used to discover how the GPS receiver responded to certain commands (Appendix B) and to give the GPS receiver time to do satellite tracking and position calculations before it was connected to the developed system. The software may be downloaded from Motorola's website, www.oncore.motorola.com.

The purpose of this appendix is to introduce the reader to the software, by presenting a screenshot of it. For more information, the reader is advised to visit the GPS product website.

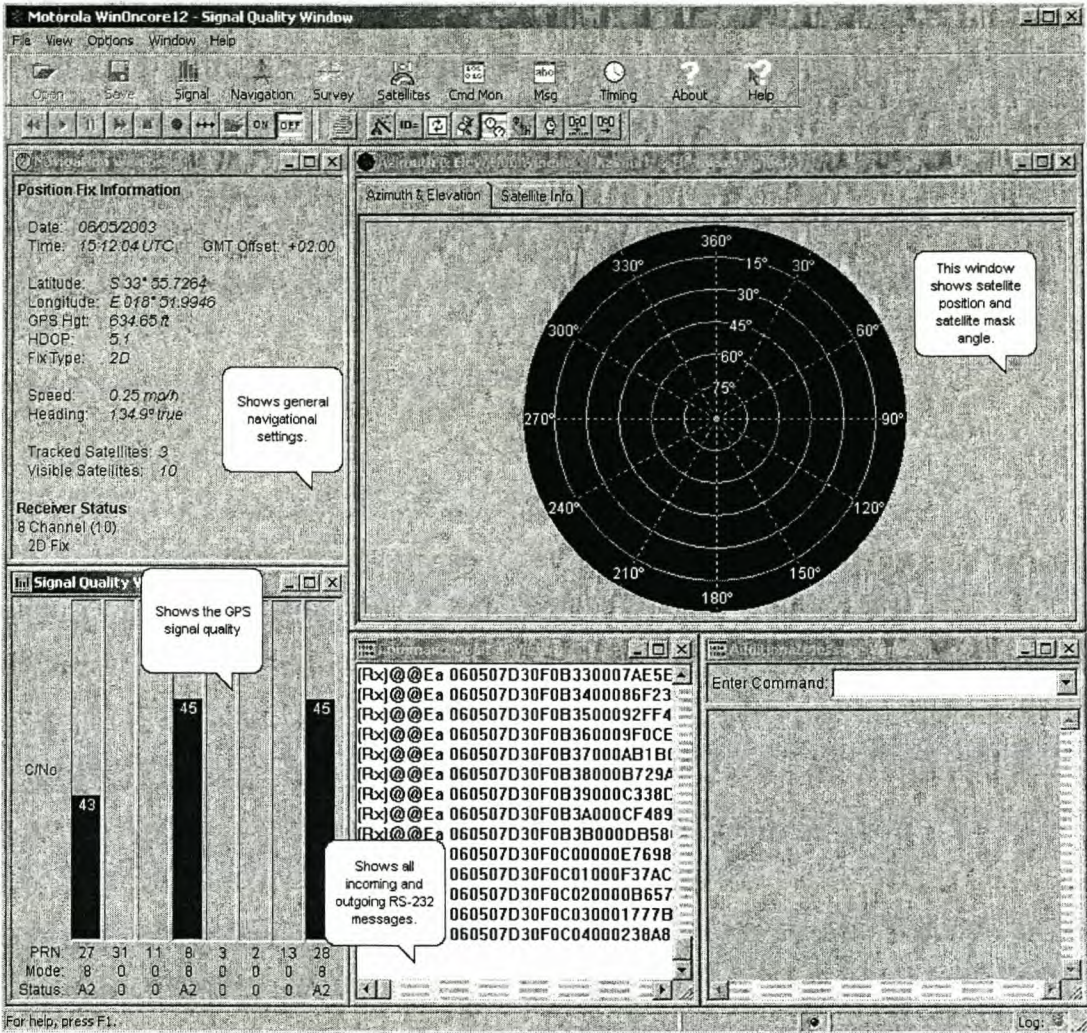


Figure A-1 WinOncore™ GPS control software screen capture

Appendix B

Oncore™ GT+ Control Commands

A list of GPS receiver control commands is presented here [39]. For more information on commands, refer to the GPS receiver User Guide [39] pages.

GT & UT Oncore

Function	Description	Binary Command	Controller Command	User Guide Page #
Time	GMT Offset	@@Ab	gnt	6.4
Time	Time Mode	@@Aw	utc	6.6
Time	Date	@@Ac	date	6.8
Time	Time of Day	@@Aa	time	6.10
Position	Latitude	@@Ad	lat	6.12
Position	Longitude	@@Ae	lon	6.14
Position	Height	@@Af	hgt	6.16
Position	Position/Status/Data Message	@@Ea	ps8	6.18
Satellite	Set Mask Angle	@@Ag	mask	6.22
Satellite	Visible Satellite Status Message	@@Bb	vis	6.24
Setup	Leap Second Pending Status	@@Bj	leapsec	6.26
Setup	Atmospheric Correction Mode	@@Aq	ion	6.28
Setup	Set User Datum	@@Ap	udatum	6.30
Setup	Select Datum	@@Ao	datum	6.32
Almanac	Almanac Data Input	@@Cb	almin	6.34
Almanac	Almanac Data Output	@@Be	almout	6.36
Receiver	System Power-on Failure Message	@@Sz	n/a	6.38
Receiver	Receiver ID	@@Cj	id	6.40
Receiver	Self-Test	@@Fa	selftest8	6.42
Receiver	Set-to-Defaults	@@Cf	default	6.44

GT Oncore Only

Function	Description	Binary Command	Controller Command	User Guide Page #
Position	ASCII Position Message	@@Eq	as8	6.46
Setup	Altitude-Hold Height	@@Au	ahp	6.51
Setup	Altitude-Hold Mode	@@Av	ah	6.52
Setup	Velocity Filter	@@AN	filter	6.54
Setup	RTCM Port Mode	@@AO	p2baud	6.56
Ephemeris	Ephemeris Data Input	@@Bf	ephin	6.58
DGPS	Pseudorange Correction Input	@@Ce	n/a	6.60
NMEA	Switch to NMEA	@@Ci	ioformat	6.62
NMEA	8 NMEA Messages	n/a	n/a	6.64

GT & UT Oncore

Function	Description	Binary Command	Controller Command	User Guide Page #
Time	Time of Day	@@Aa	time	6.10
Time	GMT Offset	@@Ab	gmt	6.4
Time	Date	@@Ac	date	6.8
Position	Latitude	@@Ad	lat	6.12
Position	Longitude	@@Ae	lon	6.14
Position	Height	@@Af	hgt	6.16
Satellite	Set Mask Angle	@@Ag	mask	6.22
Setup	Velocity Filter	@@AN	filter	6.54
Setup	Select Datum	@@Ao	datum	6.32
Setup	RTCM Port Mode	@@AO	p2baud	6.56
Setup	Set User Datum	@@Ap	udatum	6.30
1PPS	Pulse Mode	@@AP	n/a	6.86
Setup	Atmospheric Correction Mode	@@Aq	ion	6.28
1PPS	Position-Hold Position	@@As	ph	6.88
1PPS	Position-Hold Mode	@@At	php	6.90
Setup	Altitude-Hold Height	@@Au	ahp	6.51
Setup	Altitude-Hold Mode	@@Av	ah	6.52
Time	Time Mode	@@Aw	utc	6.6
1PPS	1PPS Offset	@@Ay	ppssoff	6.84
1PPS	1PPS Cable Delay	@@Az	ppsdelay	6.82
Satellite	Visible Satellite Status Message	@@Bb	vis	6.24
Almanac	Almanac Data Output	@@Be	almout	6.36
Ephemeris	Ephemeris Data Input	@@Bf	eptin	6.58
Time	Leap Second Pending Status	@@Bj	leapsec	6.26
Time	UTC Offset Status Message	@@Bo	utcoff	6.80
Almanac	Almanac Data Input	@@Cb	almin	6.34
DGPS	Pseudorange Correction Input	@@Ce	n/a	6.60
Receiver	Set-to-Defaults	@@Cf	default	6.44
NMEA	Switch to NMEA	@@Ci	ioformat	6.62
NMEA	8 NMEA Messages	n/a	n/a	6.64
Receiver	Receiver ID	@@Cj	id	6.40
Position	Position/Status/Data Message	@@Ea	ps8	6.18
1PPS	Time RAIM Setup and Status Message	@@En	trstat8	6.92
Position	ASCII Position Message	@@Eq	n/a	6.46
Receiver	Self-Test	@@Fa	selftest8	6.42
Receiver	System Power-on Failure Message	@@Sz	n/a	6.38

Appendix C

Address Decoder: Code and simulation

C.1 Data Register Addresses

R/W	μ s/RTC	Register	Pin Name	Address	HEX
W	RTC	Set RTC (seconds)	SSEC	0001 0001	\$11
W	RTC	Set RTC (minutes)	SMIN	0001 0010	\$12
W	RTC	Set RTC (hours)	SHRS	0001 0011	\$13
W	RTC	Set compare register (seconds)	SECCOMPL	0001 0100	\$14
W	RTC	Set compare register (minutes)	MINCOMPL	0001 0101	\$15
W	RTC	Set compare register (hours)	HRSCOMPL	0001 0110	\$16
R	RTC	RTC output (seconds)	SECO	0001 0111	\$17
R	RTC	RTC output (minutes)	MINO	0001 1000	\$18
R	RTC	RTC output (hours)	HRSO	0001 1001	\$19
R	μ s	μ second out (high byte)	CS_HIGH_OUT	1000 0001	\$81
R	μ s	μ second out (middle byte)	CS_MID_OUT	1000 0010	\$82
R	μ s	μ second out (low byte)	CS_LOW_OUT	1000 0011	\$83
W	μ s	μ second compare (high byte)	CS_HIGHC	1000 0100	\$84
W	μ s	μ second compare (middle byte)	CS_MIDC	1000 0101	\$85
W	μ s	μ second compare (low byte)	CS_LOWC	1000 0110	\$86
W	μ s	FIFO write	FIFOWR	1000 0111	\$87
W	μ s	FIFO read	FIFORD	1000 1000	\$88
W	μ s	FIFO reset	FIFORS	1000 1001	\$89
W	μ s	'GPS_OK' LED ON		1000 1010	\$8A
W	μ s	'READY' LED ON		1000 1011	\$8B
W	μ s	'TRIGGER' LED ON		1000 1100	\$8C
W	μ s	'GPS_OK' LED OFF		1000 1101	\$8D
W	μ s	'READY' LED OFF		1000 1110	\$8E
W	μ s	'TRIGGER' LED OFF		1000 1111	\$8F

C.2 VHDL Code for RTC Address Decoder

```
-- addr_rtc.vhd (used in rtc_system.gdf)
--
-- This VHD block implements an address decoder for input and output
-- data registers used in the RTC system.
```

```
Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;
```

Entity ADDR_RTC is

```
Port
(
  ADDR      : in std_logic_vector(7 downto 0); -- ADDRESSBUS
  NOTWR     : in std_logic;                  -- Microcontroller /WR line
  NOTRD     : in std_logic;                  -- Microcontroller /RD line
  SSEC      : out std_logic;                 -- RTC sec, min & hrs
  SSEC      : out std_logic;                 -- 0001 0001 ($11)
  SMIN      : out std_logic;                 -- 0001 0010 ($12)
  SHRS      : out std_logic;                 -- 0001 0011 ($13)
  SECCOMPL  : out std_logic;                 -- Compare sec, min & hrs
  SECCOMPL  : out std_logic;                 -- 0001 0100 ($14)
  MINCOMPL  : out std_logic;                 -- 0001 0101 ($15)
  HRSCOMPL  : out std_logic;                 -- 0001 0110 ($16)
  SECO      : out std_logic;                 -- RTC output registers
  SECO      : out std_logic;                 -- 0001 0111 ($17)
  MINO      : out std_logic;                 -- 0001 1000 ($18)
  HRSO      : out std_logic );               -- 0001 1001 ($19)
```

End ADDR_RTC;

Architecture IMPLEMENTATION of ADDR_RTC is

```
Begin
  SSEC <= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and not(ADDR(1))
    and ADDR(0);
  SMIN <= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and ADDR(1)
    and not(ADDR(0));
  SHRS <= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and ADDR(1)
    and ADDR(0);
  SECCOMPL<= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and ADDR(2) and not(ADDR(1))
    and not(ADDR(0));
  MINCOMPL<= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and ADDR(2) and not(ADDR(1))
    and ADDR(0);
  HRSCOMPL<= not(NOTWR) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and ADDR(2) and ADDR(1)
    and not(ADDR(0));
  SECO <= not(NOTRD) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
    and (ADDR(4)) and not(ADDR(3)) and ADDR(2) and ADDR(1)
```

```

        and ADDR(0);
MINO <= not(NOTRD) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
        and (ADDR(4)) and ADDR(3) and not(ADDR(2)) and not(ADDR(1))
        and not(ADDR(0));
HRSO <= not(NOTRD) and not(ADDR(7)) and not(ADDR(6)) and not(ADDR(5))
        and (ADDR(4)) and ADDR(3) and not(ADDR(2)) and not(ADDR(1))
        and (ADDR(0));
End IMPLEMENTATION;

```

C.3 VHDL Code for Microsecond Counter Address Decoder

```

Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;

```

Entity ADDR_DEC_US is

Port

```

(
  ADDR          : in std_logic_vector(7 downto 0); -- Address bus
  NOTWR         : in std_logic;                  -- Micro /WR
  NOTRD         : in std_logic;                  -- Micro /RD LINE
  RDVME         : in std_logic;                  -- VMEBUS read signal (FIFORD)
                                                    -- Microsecond count output
  CS_HIGH_OUT   : out std_logic;                 -- 1000 0001 ($81)
  CS_MID_OUT    : out std_logic;                 -- 1000 0010 ($82)
  CS_LOW_OUT    : out std_logic;                 -- 1000 0011 ($83)
                                                    -- Microsecond compare registers
  CS_HIGHC     : out std_logic;                 -- 1000 0100 ($84)
  CS_MIDC      : out std_logic;                 -- 1000 0101 ($85)
  CS_LOWC      : out std_logic;                 -- 1000 0110 ($86)
                                                    -- FIFO control signals
  FIFOWR       : out std_logic;                 -- FIFO Memory /WRITE 1000 0111 ($87)
  FIFORD       : out std_logic;                 -- FIFO Memory /READ  1000 1000 ($88)
  FIFORS       : out std_logic;                 -- FIFO Memory /RESET 1000 1001 ($89)
  GPS_OK_ON    : out std_logic;                 -- GPS OK LED         1000 1010 ($8A)
  GPS_OK_OFF   : out std_logic;                 -- GPS OK LED OFF     1000 1101 ($8D)
  READY_ON     : out std_logic;                 -- READY LED          1000 1011 ($8B)
  READY_OFF    : out std_logic;                 -- GPS OK LED OFF     1000 1110 ($8E)
  TRIGGER_ON   : out std_logic;                 -- TRIGGER LED        1000 1100 ($8C)
  TRIGGER_OFF  : out std_logic;                 -- GPS OK LED OFF     1000 1111 ($8F)
);

```

End ADDR_DEC_US;

Architecture IMPLEMENTATION of ADDR_DEC_US is

Begin

```

  CS_HIGH_OUT<=not(NOTRD) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and not(ADDR(1))
        and ADDR(0);
  CS_MID_OUT<=not(NOTRD) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and ADDR(1)
        and not(ADDR(0));
  CS_LOW_OUT<= not(NOTRD) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))

```

```

        and not(ADDR(4)) and not(ADDR(3)) and not(ADDR(2)) and ADDR(1)
        and ADDR(0);
CS_HIGHC <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and ADDR(2) and not(ADDR(1))
        and not(ADDR(0));
CS_MIDC <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and ADDR(2) and not(ADDR(1))
        and ADDR(0);
CS_LOWC <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and ADDR(2) and ADDR(1)
        and not(ADDR(0));
FIFOWR <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and not(ADDR(3)) and ADDR(2) and ADDR(1)
        and ADDR(0);
FIFORD <= RDVME or (not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and not(ADDR(2)) and not(ADDR(1))
        and not(ADDR(0)));
FIFORS <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and not(ADDR(2)) and not(ADDR(1))
        and ADDR(0);
GPS_OK_ON <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and not(ADDR(2)) and ADDR(1)
        and not(ADDR(0));
READY_ON <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and not(ADDR(2)) and ADDR(1)
        and ADDR(0);
TRIGGER_ON <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and ADDR(2) and not(ADDR(1))
        and not(ADDR(0));
GPS_OK_OFF <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and ADDR(2) and not(ADDR(1))
        and ADDR(0);
READY_OFF <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and ADDR(2) and ADDR(1)
        and not(ADDR(0));
TRIGGER_OFF <= not(NOTWR) and ADDR(7) and not(ADDR(6)) and not(ADDR(5))
        and not(ADDR(4)) and ADDR(3) and ADDR(2) and ADDR(1)
        and ADDR(0);

```

End IMPLEMENTATION;

C.4 Altera® MAX+PLUS® Simulation Waveforms

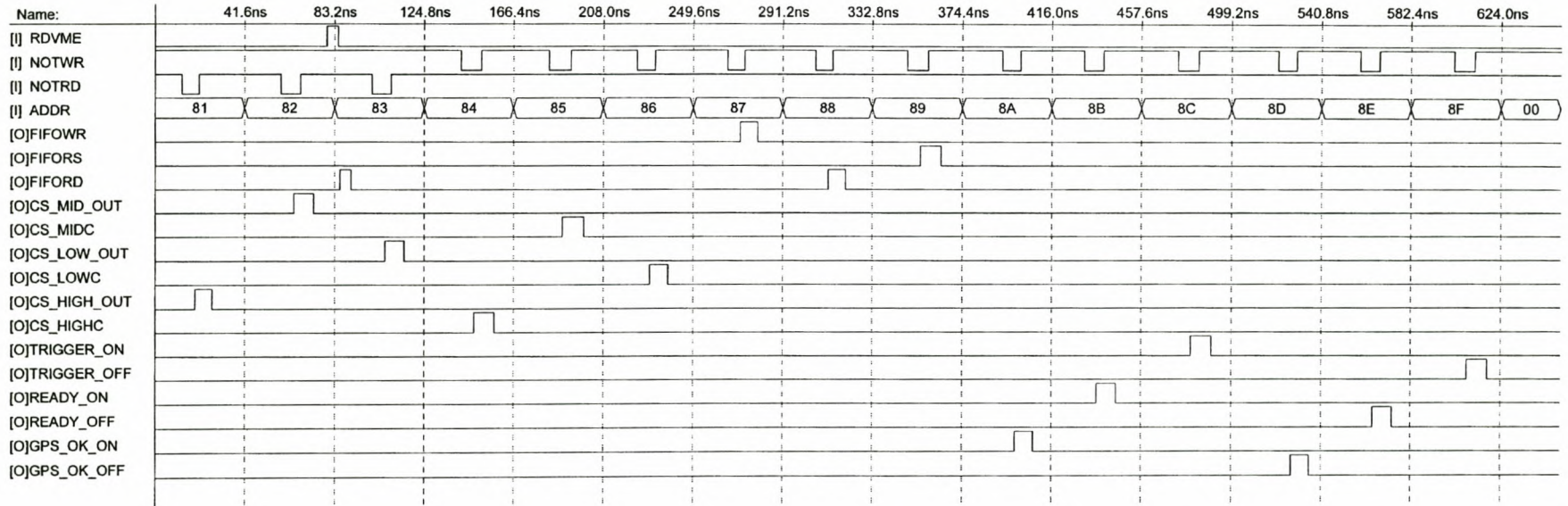


Figure C-1 Address decoder simulation waveforms

Appendix D

RTC and Comparator: Implementation and simulation

The RTC *Altera*® *MAX+PLUS*® implementations and simulations are presented in this appendix. An implementation diagram will be followed by its simulation waveform.

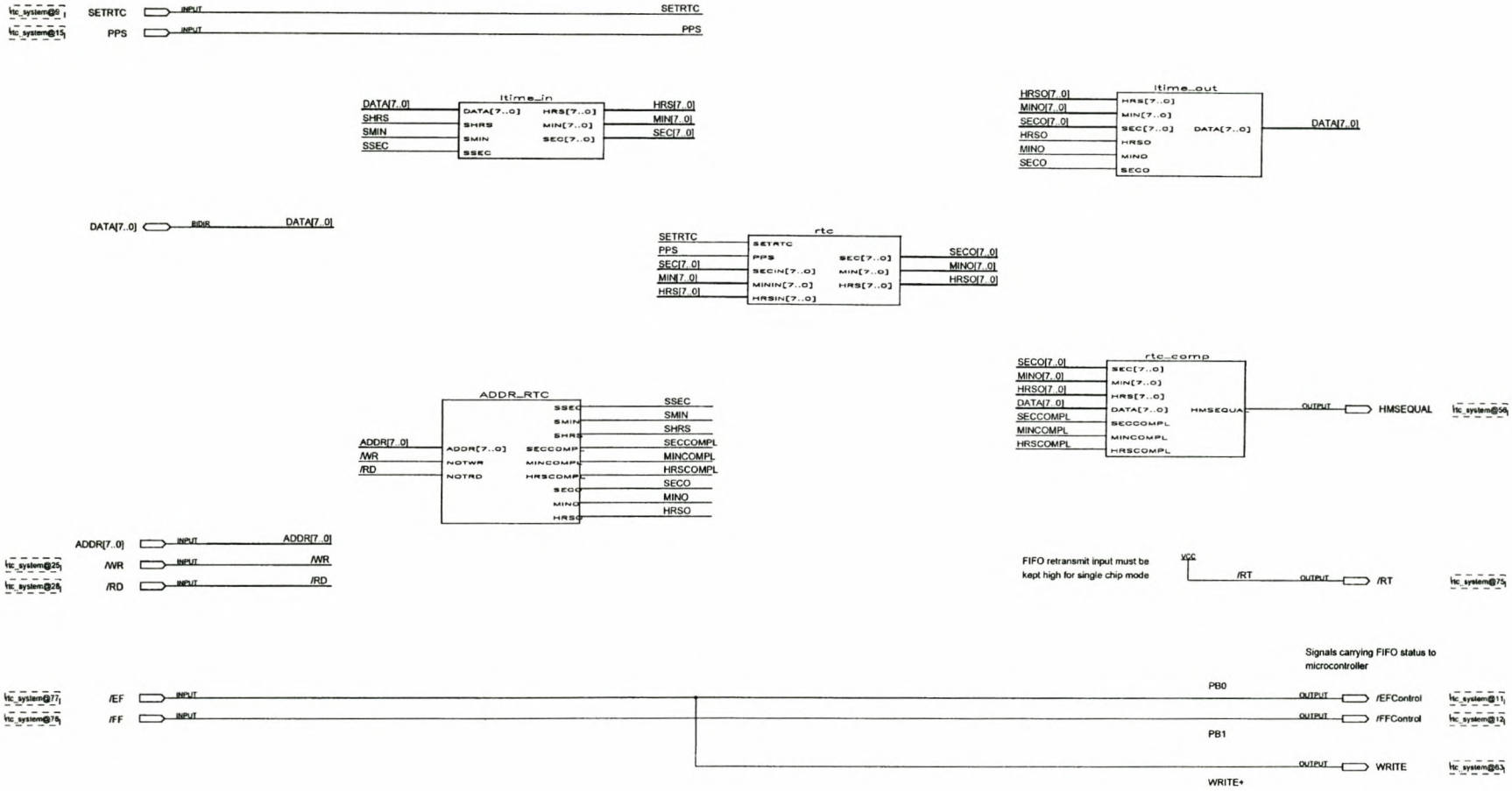


Figure D-1 Complete RTC system implementation diagram

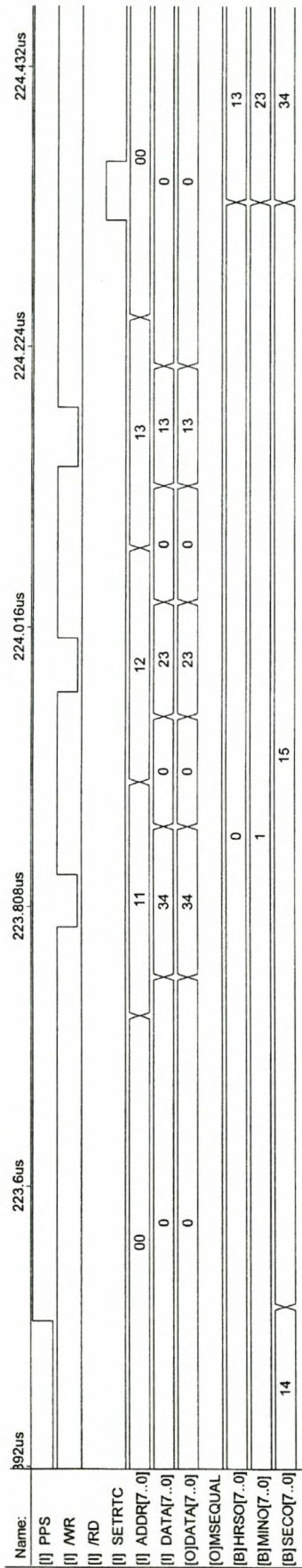


Figure D-2 Complete RTC system simulation waveforms (SetRTC)

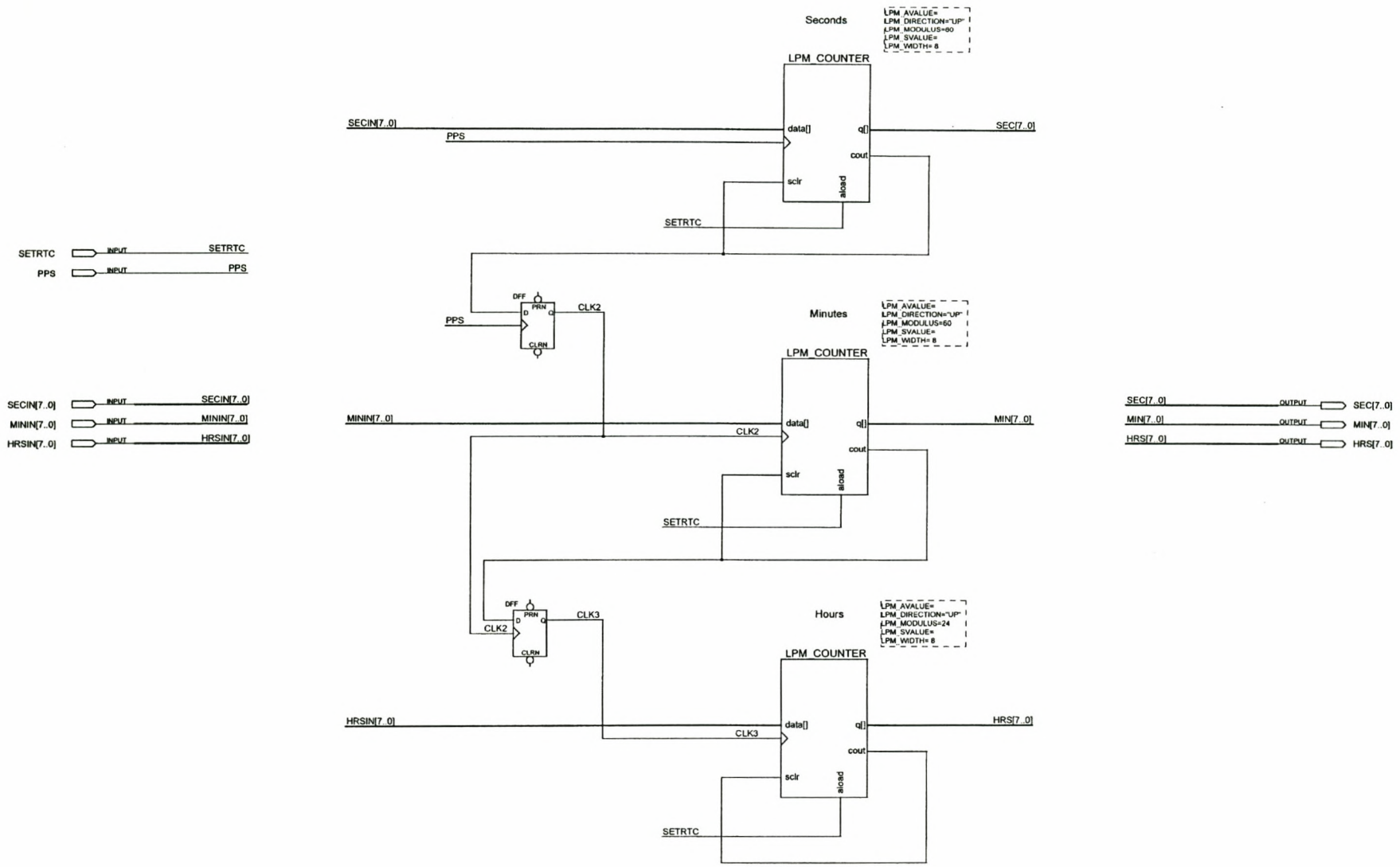


Figure D-3 RTC implementation diagram

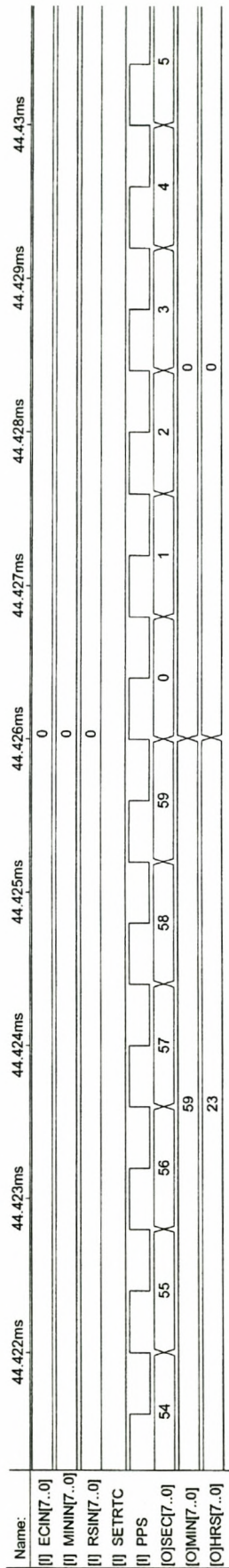


Figure D-4 RTC normal operation simulation

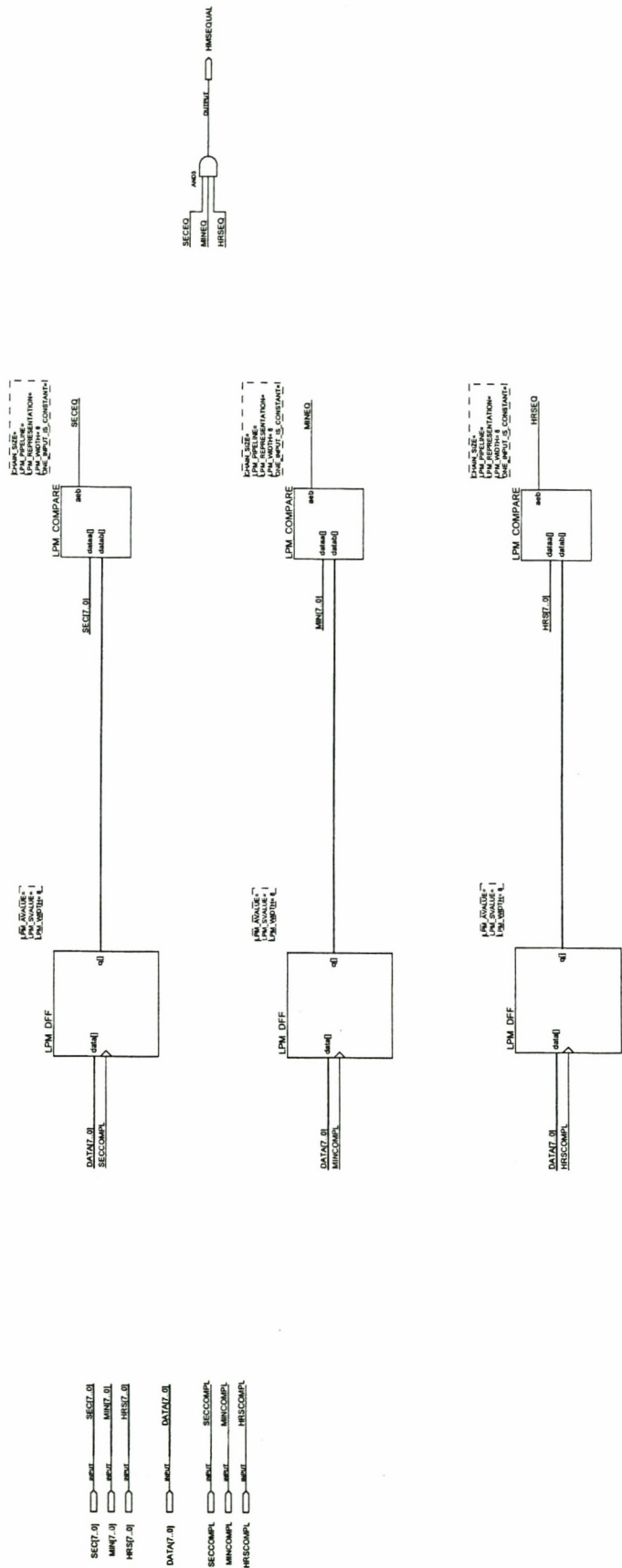


Figure D-5 RTC Comparator implementation diagram

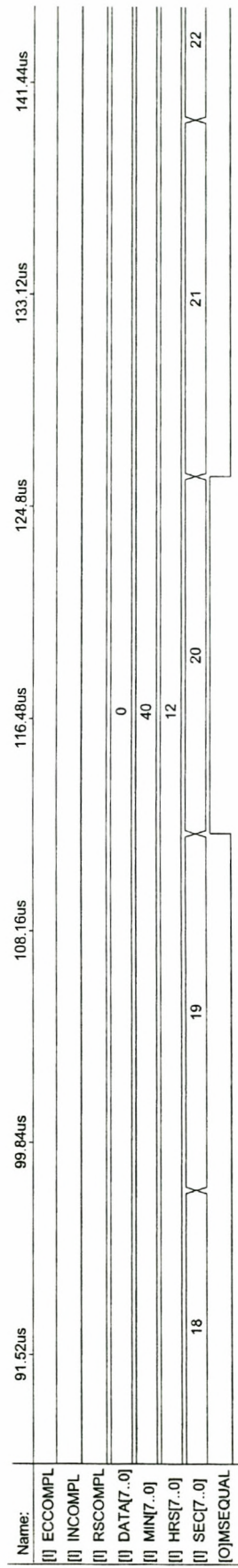


Figure D-6 RTC Comparator simulation waveforms (HMSEqualI)

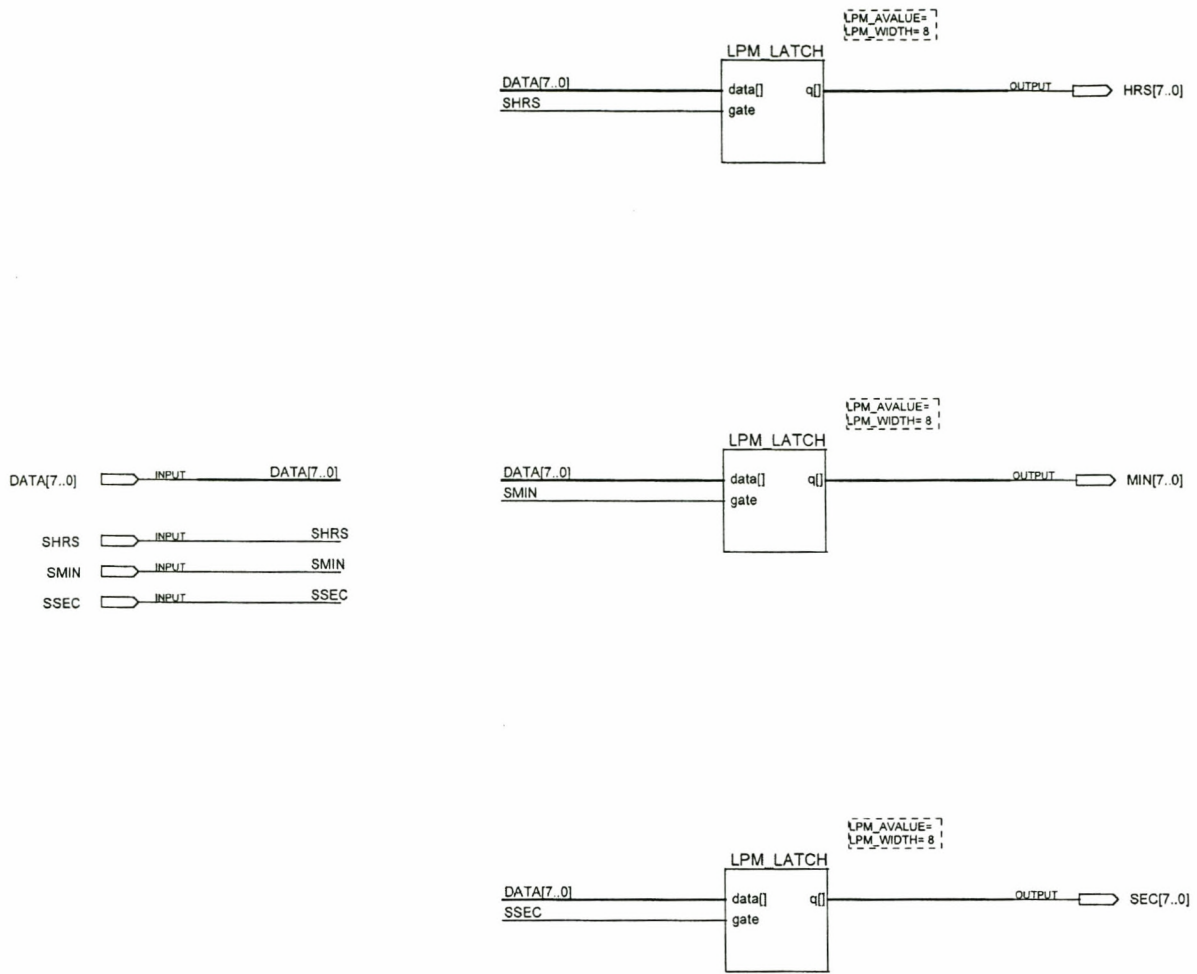


Figure D-7 RTC Data input latch implementation diagram

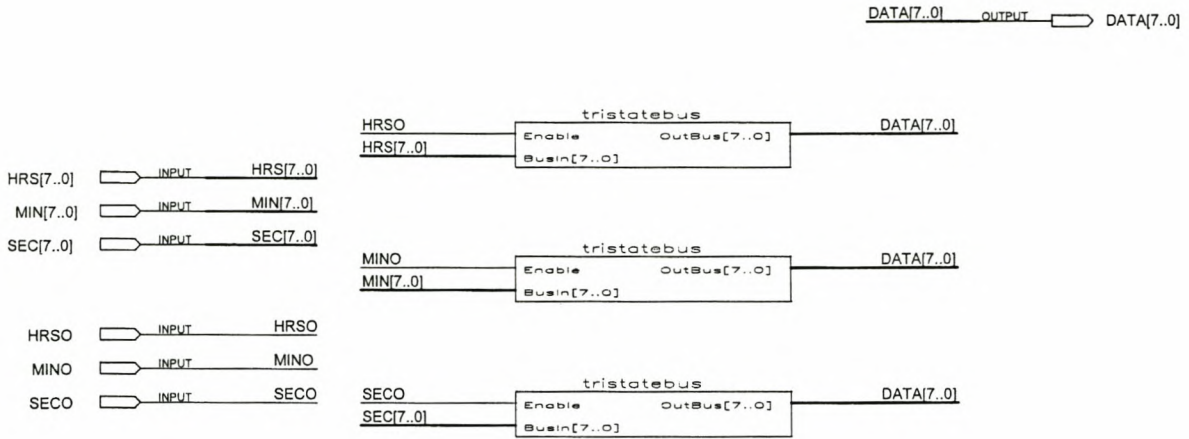


Figure D-8 RTC Data output latch implementation diagram

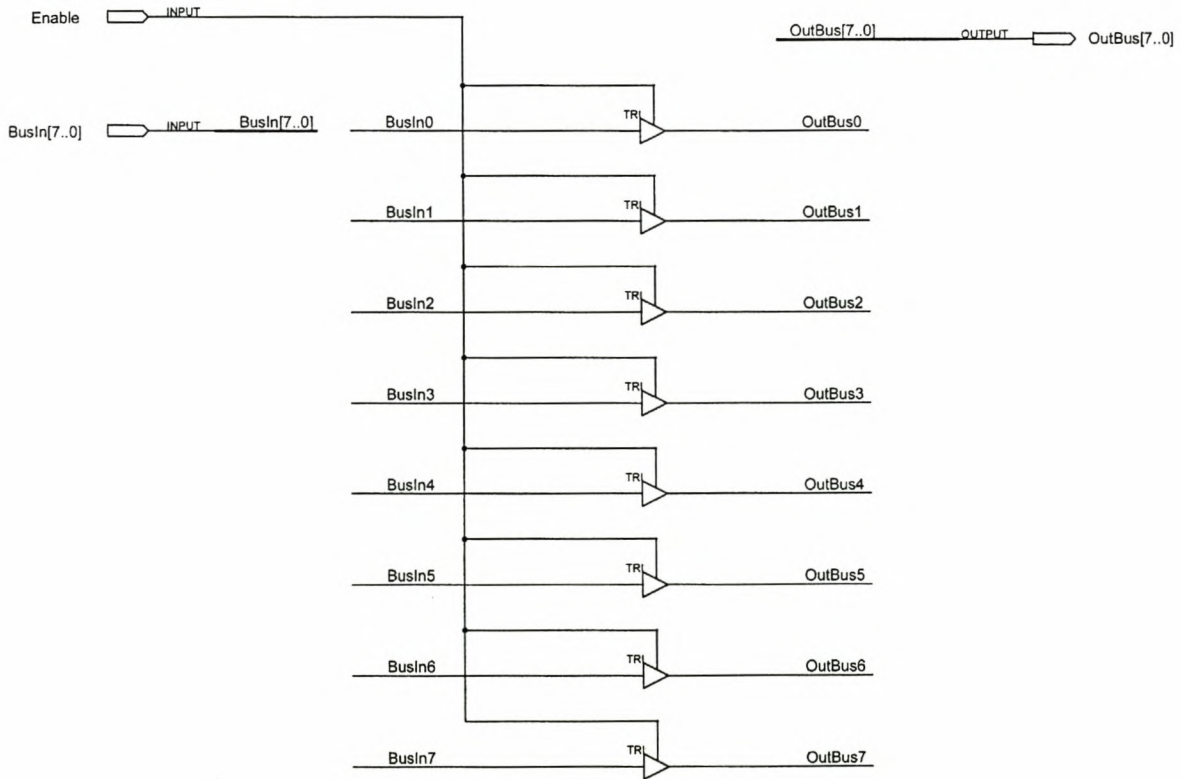


Figure D-9 RTC Data output latch (tri-state bus) implementation diagram

Appendix E

Microsecond Counter and Comparator: Implementation and simulation

The Microsecond Counter *Altera*® *MAX+PLUS*® implementations and simulations are presented in this appendix. An implementation diagram will be followed by its simulation waveform. VHDL program code for the *Monostable Multivibrator* (section 5.3.5.2) and *TrigProcess* (section 5.3.5.1) will also be presented.

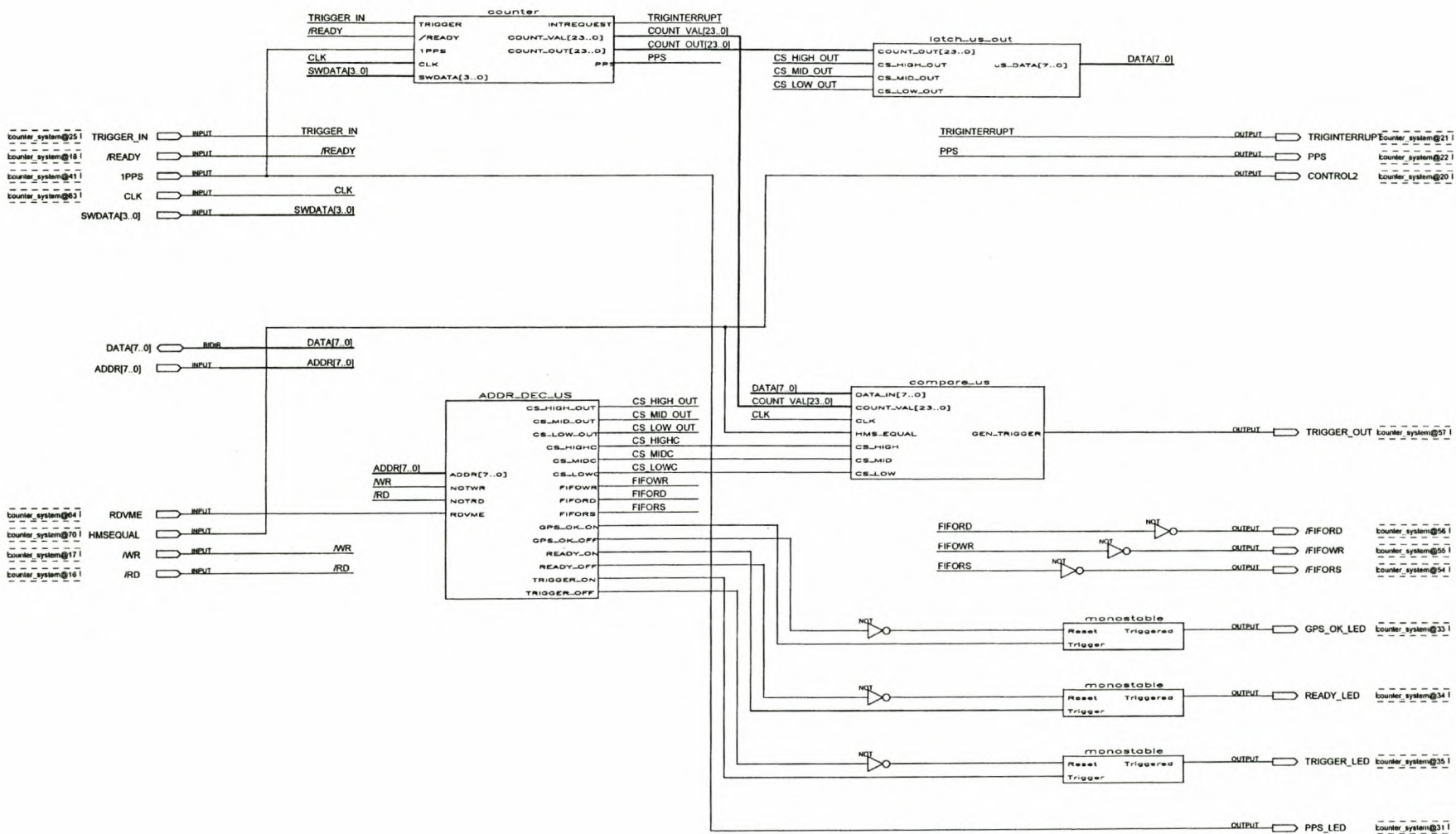


Figure E-1 Microsecond Counter complete system implementation diagram

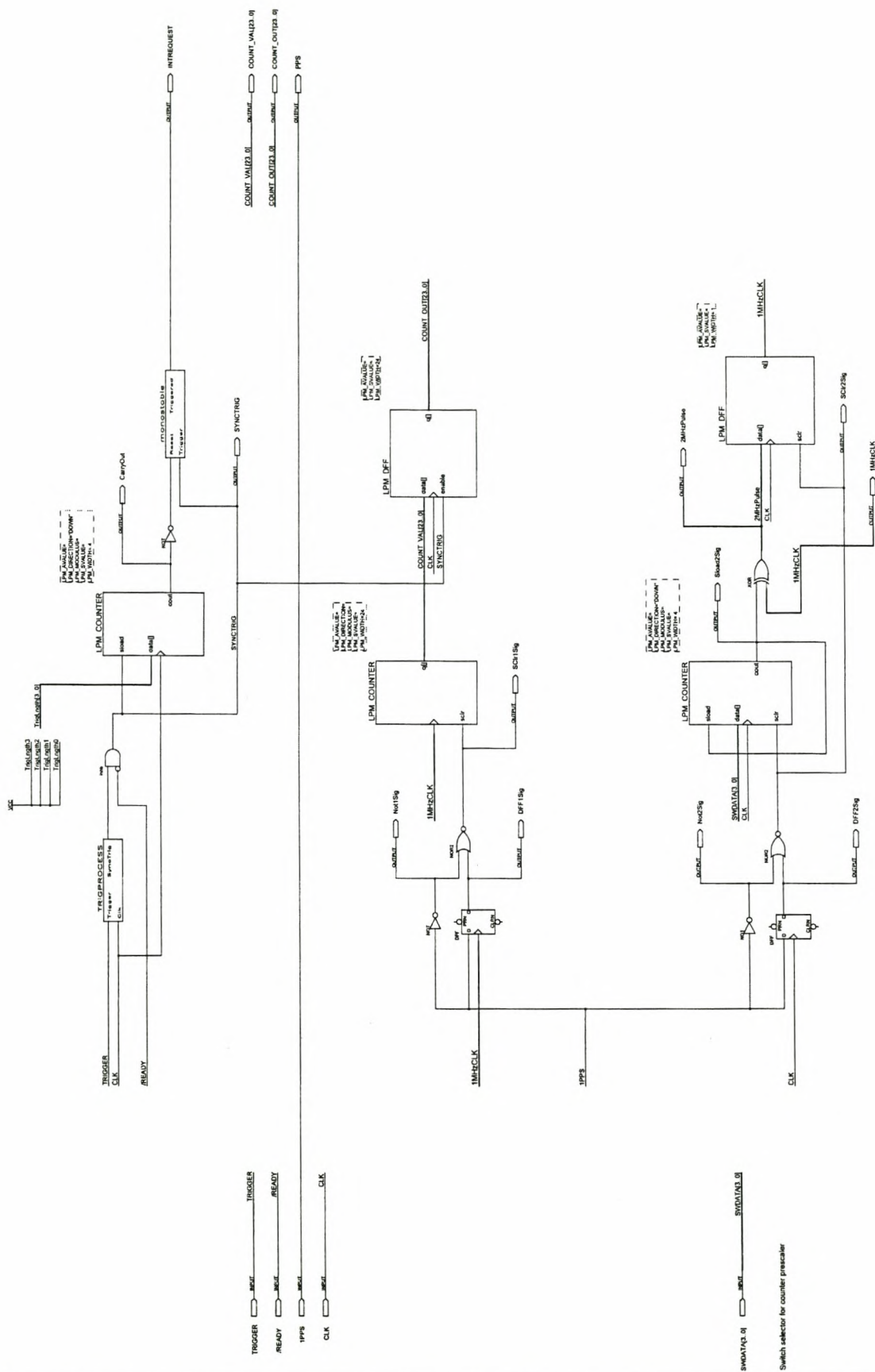


Figure E-2 Microsecond Counter and 1MHz Clock generator implementation diagram

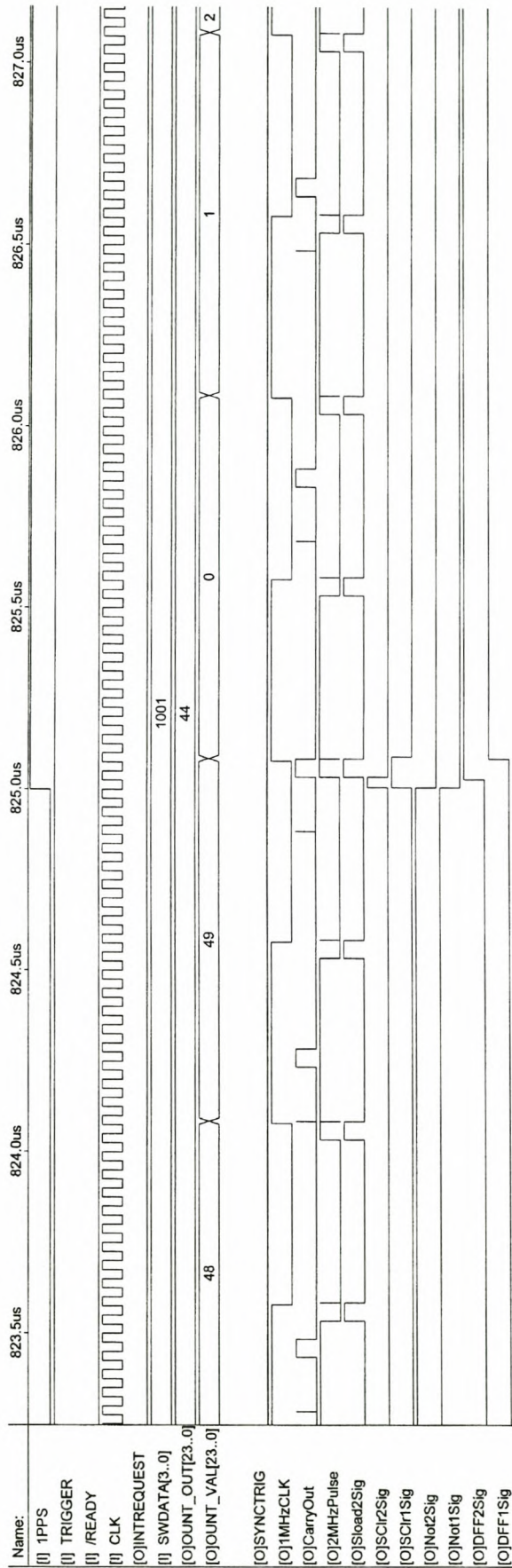


Figure E-3 Microsecond Counter operation waveform (1PPS reset)

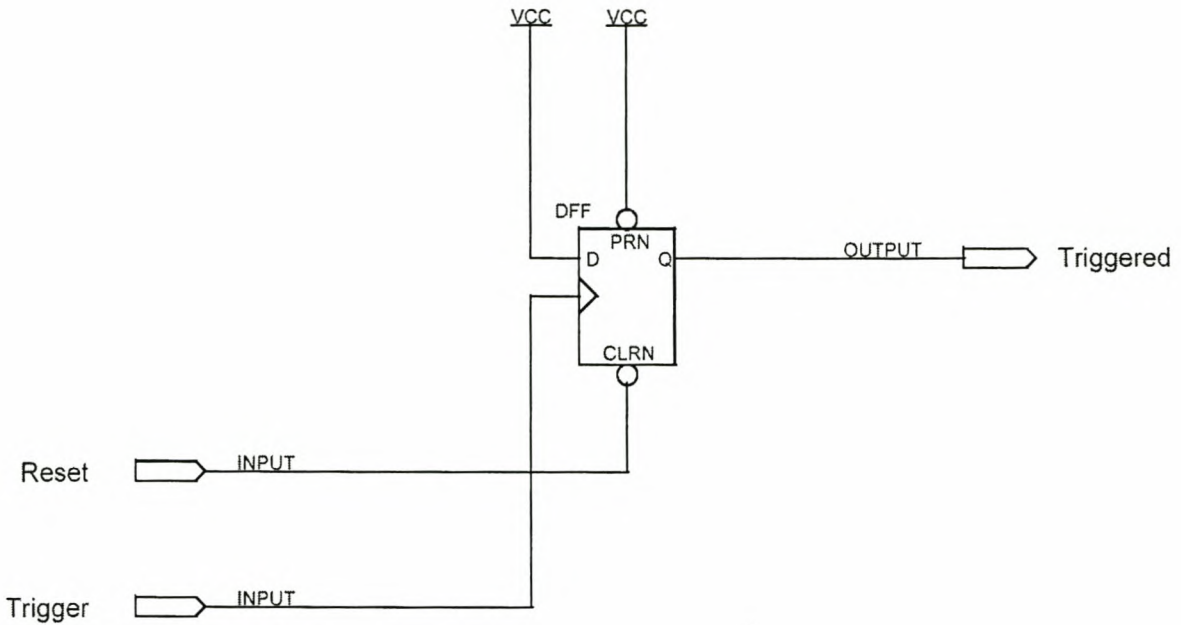


Figure E-5 Monostable Multivibrator implementation diagram

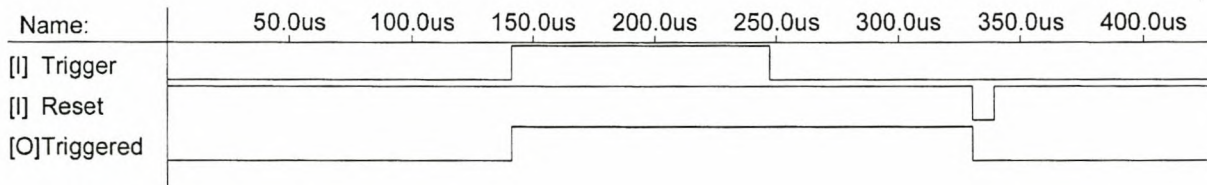


Figure E-6 Monostable Multivibrator simulation waveform

VHDL code for the *TrigProcess* block will be presented now.

```
-- trigprocess.vhd (used in counter.gdf)
--
-- This VHDL block generates a 6 clock cycle wide pulse from any length input
-- See trigprocess.scf for details

Library ieee;
Use ieee.std_logic_1164.all;
Use ieee.std_logic_arith.all;

Entity TRIGPROCESS is

Port
(
    Trigger      : in std_logic;           -- External trigger signal
    Clk          : in std_logic;         -- Global clock signal

    SyncTrig: out std_logic      );       -- Pulse synchronised with CLK

End TRIGPROCESS;
```

Architecture IMPLEMENTATION of TRIGPROCESS is

```
Signal WaitTrig : std_logic := '1';
Begin
  Process(Clk)
  Begin
    If (Clk'EVENT) and (Clk = '1') then
      If (Trigger = '1') and (WaitTrig = '1') then
        SyncTrig <= '1';
        WaitTrig <= '0';
      Elsif Trigger = '1' and (WaitTrig = '0') then
        SyncTrig <= '0';
      Elsif Trigger = '0' and (WaitTrig = '0') then
        WaitTrig <= '1';
      End If;
    End If;
  End Process;
End IMPLEMENTATION;
```

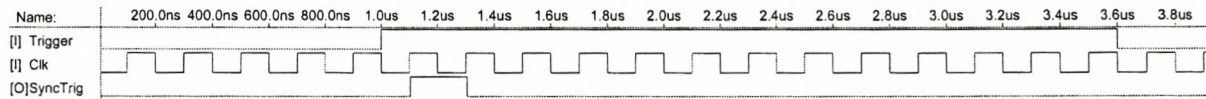


Figure E-7 TrigProcess implementation simulation waveform

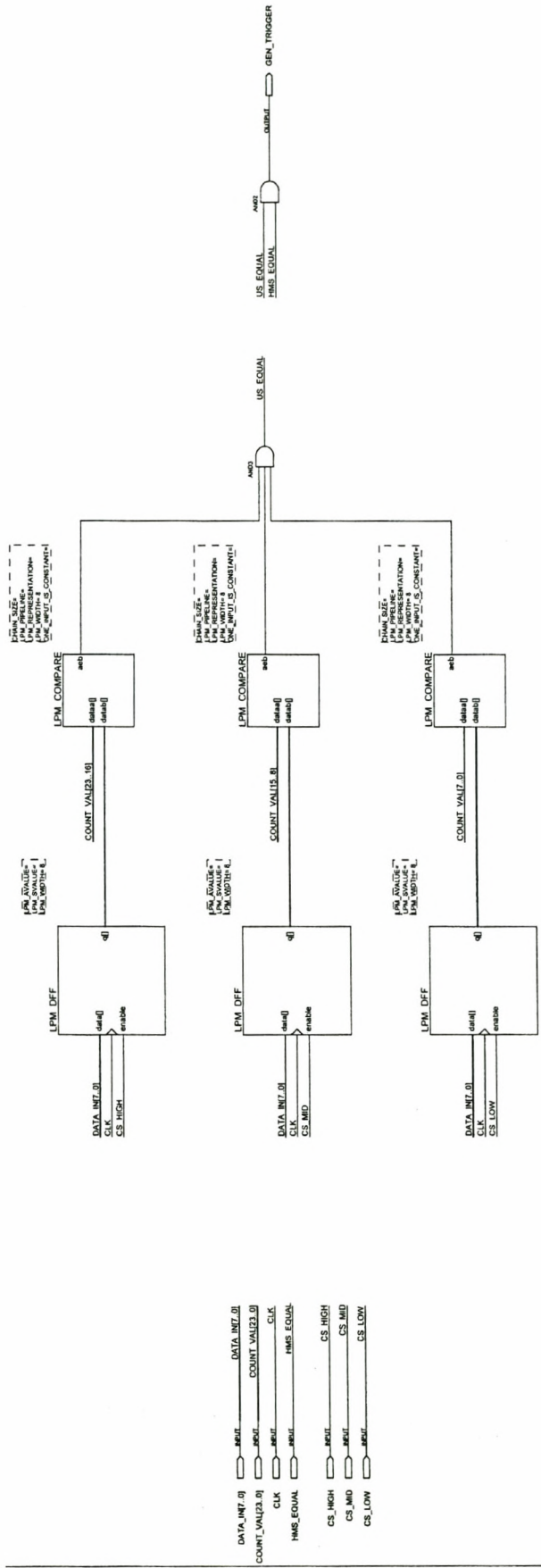


Figure E-8 Microsecond Counter Comparator implementation diagram

Appendix F

Software UART Theory

F.1 Introduction

Many control applications communicate serially in one direction at a time only (half-duplex). This appendix describes how the creation of a half-duplex UART on any *Atmel AVR* device using an 8-bit Timer/Counter and an external interrupt. The software described here may be used to implement a second serial port on devices with only one hardware UART. The operation of the software UART was deduced from references [51]. A typical transmission byte is presented in Figure F-1.

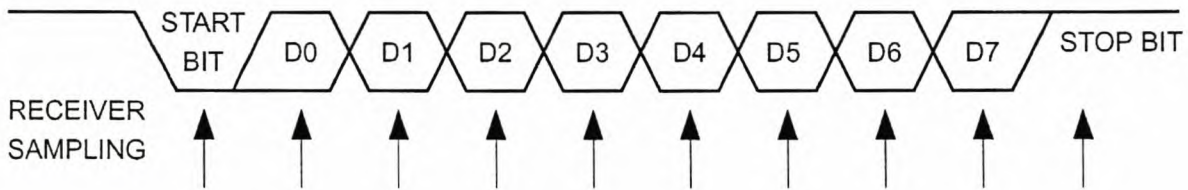


Figure F-1 Transmission byte format [51]

F.2 Theory of operation

Asynchronous serial data communication follow some simple rules on data transfer. Data is transmitted sequentially, one bit at a time. To inform the receiver that a new byte is arriving, each byte is placed between start and a stop bits as shown above. This construction is called a data frame. The frame has a start-bit, 8 data bits, and a stop-bit. The frame format can be extended and might also include parity bits and more stop bits.

An idle line is signaled by holding the communications line at logical one. The start bit is always zero and the UART receiver will detect the start of a frame by the first falling edge. Data bits follow the start bit and the byte ends with a stop bit which is always a logical one. This stop bit is held stable at one until the next start bit is sent.

In asynchronous transmissions, no separate clock is provided to the receiver. Correct reception of data is guaranteed by keeping all bit lengths equal. The receiver will synchronise from the first falling edge of the start bit and it determines the next sampling time with its own timer.

The bit length is determined by the baud rate of communication. In a UART, the baud rate is equal to the number of bits transmitted per second. The transmitter and receiver has to be set up using the same baud rate for correct reception. The falling edge of the start bit generates an initial interrupt (using an external interrupt). The interrupt starts Timer/Counter0 and presets it to time out in exactly 1.5 bit lengths. This 1.5 bit length delay is required to generate the next sampling event at the bit-center of the first data bit. The next eight interrupts are generated by a predefined delay (1 bit length), using Timer/Counter0. Figure F-2 shows the flow diagram for receiving serial data.

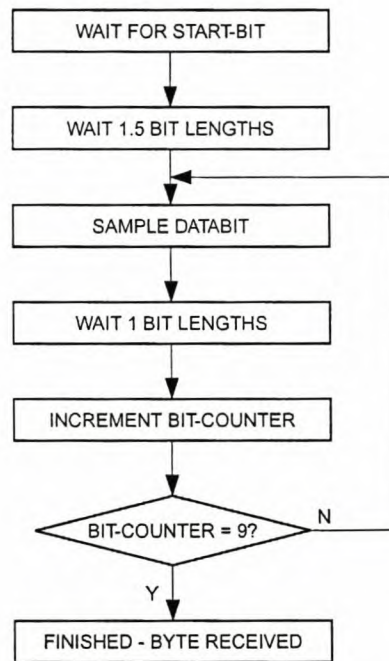


Figure F-2 Flow diagram of serial data reception [51]

Transmitting data is less complex, as all bits have equal length and the timer can be preset at a constant delay (1 bit-length). The first bit is the start-bit and this is always a logical zero. The data bits can then be shifted out, LSB (least significant bit) first and MSB (most significant bit) last. Finally, the last bit is a stop bit. This is always a logical one. Figure F-3 shows the flow diagram for transmitting serial data.

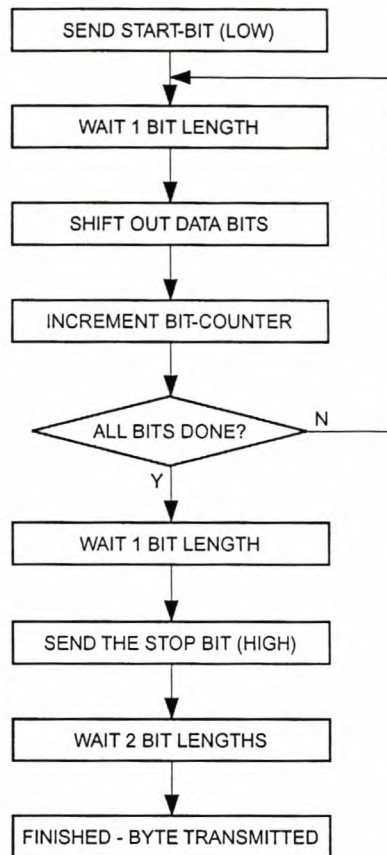


Figure F-3 Flow diagram of serial data transmission [51]

F.3 Implementation

These software UART routines use Timer/Counter0 and one external interrupt. The clock provided to the MCU will limit the maximum baud rate obtainable. This software UART is capable of handling baud rates up to 38.400 kbit/s, at 1MHz clock frequency. At this speed nearly all computing power is used, but the MCU is still available for other tasks between each byte being transmitted. The bit length is determined by the number of cycles (C times N) required to generate another overflow. with the Timer/Counter. 256-N is the value pre stored in Timer/Counter0 and C is the Timer/Counter0 prescaling factor, as described in the T/C Prescaler section in the AVR datasheet [46]. N can be calculated using the following equation, where $Xtal$ is the frequency of the system:

$$N = Xtal \frac{1}{Baudrate \times C}$$

Note that the prescaling factor C should be either 1, 8, 64, 256, or 1024. The minimum value of N times C is 17. If N times C is set to be smaller, the overflow will occur even before the Timer/Counter interrupt handler has finished. The maximum value of N is $(170+20/C)$, as the receiver has to generate a delay of 1.5 bit periods when receiving the start bit. Thus,

$$\frac{17}{C} \leq N \leq 170 + \frac{20}{C}$$

F.4 UART Initialisation

Before data can be transferred using the UART, it has to be initialized by calling subroutine *uart_init*. This subroutine will set up the Timer/Counter0 prescaler and enable the Timer/Counter0 and external interrupt needed for communication. Upon return from the subroutine, a 'sei' instruction should follow to enable global interrupts. This will enable the UART. By issuing a 'cli' instruction at a later time, the UART can be disabled.

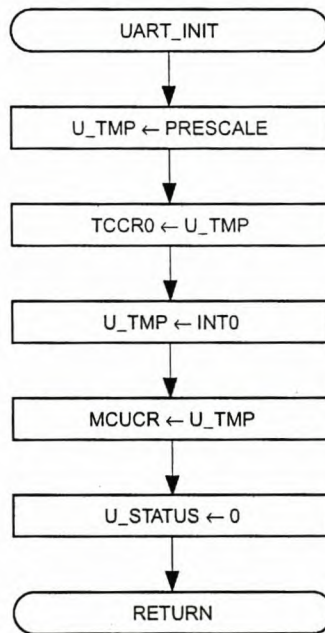


Figure F-4 UART initialisation flow diagram [51]

F.5 Byte Transmission

Routine *uart_transmit* is used to transmit data. It sets the transmit- and busy-flags, disables the external interrupt (this also disables reception), sets the correct baud rate and sets the start bit.

This routine cancels all other pending activities. Calling *uart_transmit* will clear the UART shift register *u_buffer*, clearing any data currently stored in this register. If called while receiving data, the transmitted data may be corrupted if a Timer/Counter overflow interrupt occurs while executing this subroutine. By waiting until the BUSY flag is cleared in *u_status*, safe transmissions are guaranteed.

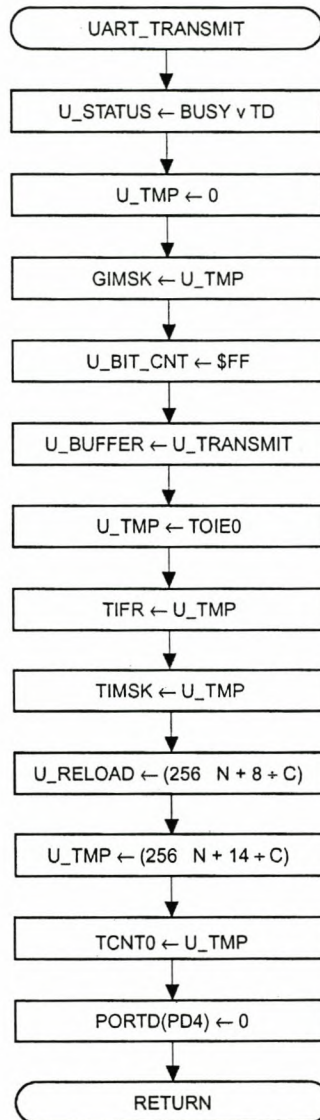


Figure F-5 UART transmission flow diagram [51]

F.6 UART Status Byte

The *u_status* byte is used by all functions. It implements three status bytes:

- **BUSY:** This bit indicates whether the UART is busy. If only the busy bit is set, the UART is in data receive mode.
- **TD:** This bit is used in conjunction with the Busy-flag. If this bit is set, the UART is currently transmitting data. The bit is set by routine `uart_transmit`, and is automatically cleared after the stop bit has been sent in routine `tim0_ovf`.
- **RDR:** This bit is set whenever `u_buffer` contains valid received data (Receive Data Ready). When the UART begins to transmit or receive, this bit will be cleared. The bit can also be cleared by software after received data has been read from `u_buffer`.

Software will detect whether incoming data is present by reading the RDR-bit. Whenever this bit goes high, new data has arrived. If the software is waiting for available time to send data, it can read the BUSY flag, and call `uart_transmit` whenever this flag is cleared. The `u_status` byte is read-only and should not be altered by user software. The RDR bit is cleared by software using a `cbr` (Clear Bit in Register) instruction, operating directly on the `u_status` register.

F.7 Interrupt Service Routines

Timer Overflow Interrupt Service Routine

This routine takes care of sending and receiving each bit in the transmission. The routine is called automatically on Timer/Counter overflow, to send or receive the next bit. The Timer/Counter overflow interrupt is enabled by `uart_transmit` or `ext0_int` when transmitting or receiving data. Upon entering this ISR¹, the Timer/Counter is preset to give the next overflow in one bit length. Then the next bit is handled, before this routine exits. If the received bit was the stop bit, the Timer/Counter overflow interrupt is cleared, and the external interrupt is enabled.

¹ Interrupt Service Routine

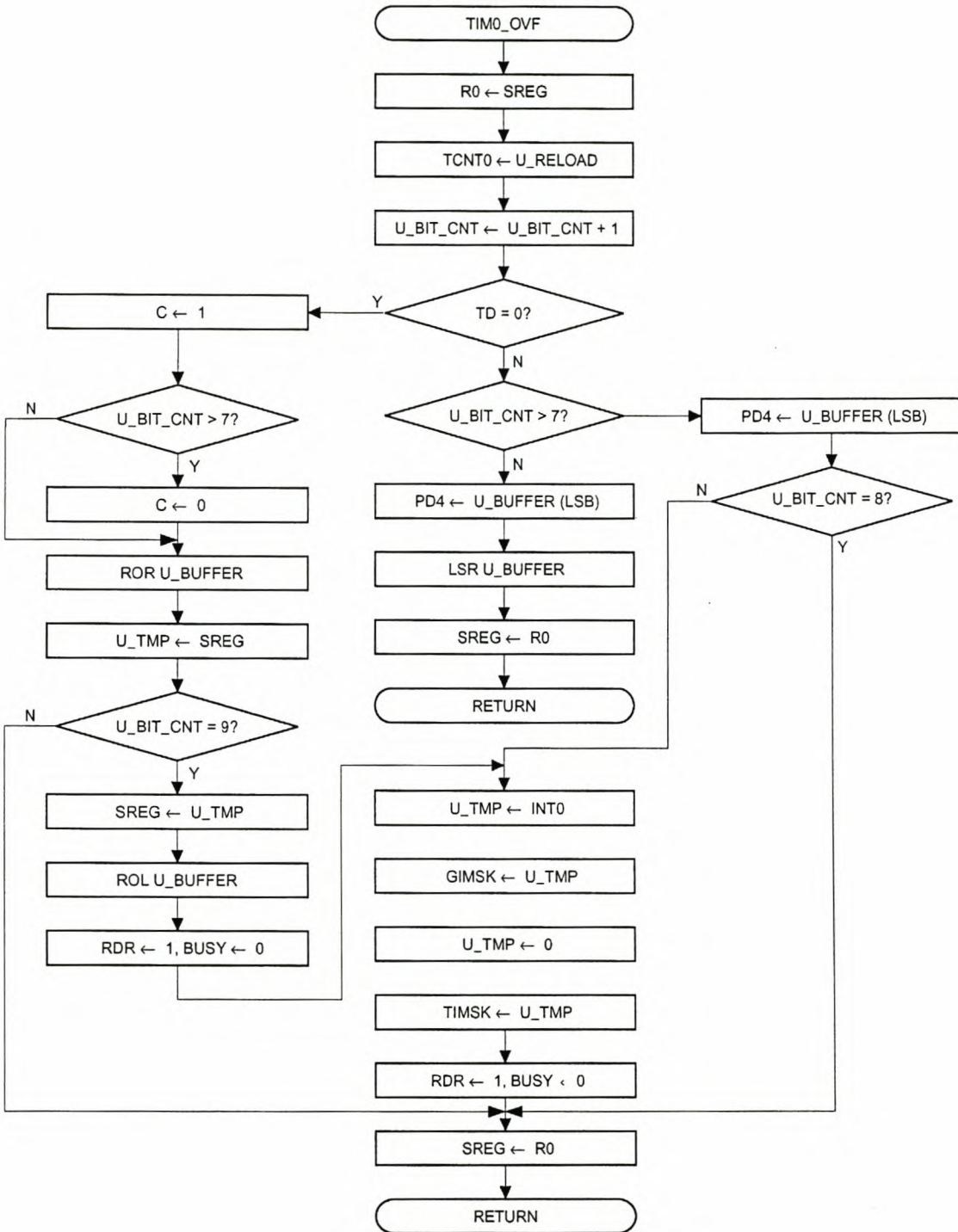


Figure F-6 Flow diagram for the Timer Overflow interrupt (*tim0_ovf*) [51]

External Interrupt Service Routine

The external interrupt 0 is active whenever the UART is idle. Upon an external interrupt, the *ext_int0* routine is called. This routine initiates the reception of serial data. An external interrupt occurs on a falling edge on the *INT0* pin (a falling edge marks the beginning of the

start-bit). This activates the Timer/Counter overflow interrupt and generates a 1.5 bit delay for the first start bit. Before exiting, the external interrupt is disabled to prevent falling edges in the incoming byte from reinitializing the receiver.

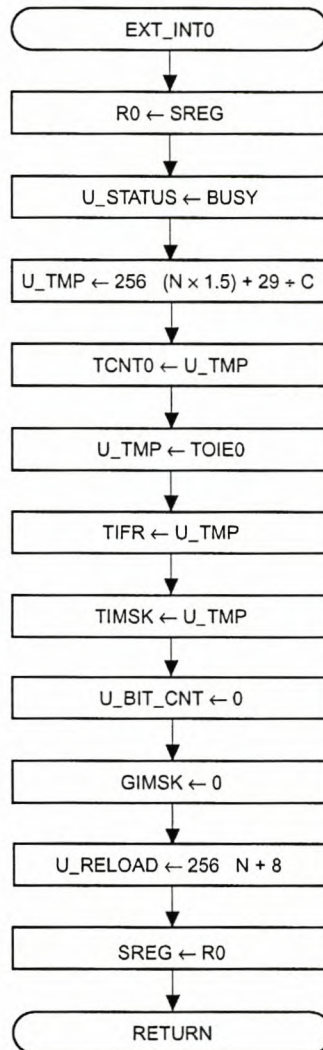


Figure F-7 External interrupt service routine (*ext_int0*) [51]

F.8 Conclusion

In the implementation, some IO-registers are manipulated without preserving current register settings. In the case of *GIMSK*, *GIFR*, *TIMSK*, and *TIFR* [46], altering these registers might also affect the operation of other peripherals. Software should normally manipulate the bits needed, preserving the rest, but to speed up the UART, the routines sets these registers by brute force. Any other bits that were in use, will be cleared. If other peripherals are being used, all UART routines must be extended to preserve all other flags.

In this appendix, the theory for a software UART has been discussed. The microcontroller is capable of using 38400 baud at 1MHz clock frequency. The UART is initialized by calling *uart_init* and enabling global interrupts. If the UART is idle, it will automatically receive incoming data. To transmit data, subroutine *uart_transmit* is called with the transmission data stored in *u_transmit*.

Appendix G

Microcontroller Code Listings

G.1 Software UART Program Code

```
// Interrupt driven software UART
```

```
link 0,$60
```

```
Program Test;
uses Intlib;
```

```
{#p abs 0}
```

```
Procedure Startup;
```

```
// Dummy procedure with startup code
```

```
Begin
```

```
  Asm
```

```
    .jmp Reset
```

```
    .jmp @Test|SUART_Rx
```

```
    .jmp INT1
```

```
    .jmp ICP1
```

```
    .jmp OCA
```

```
    .jmp OCB
```

```
    .jmp @Test|Tim1_overflow_Int
```

```
    .jmp @Test|Tim0_overflow_Int
```

```
    ;.jmp SPICmpl
```

```
    ;.jmp RXCmpl
```

```
    ;.jmp UDREmpt
```

```
    ;.jmp TxCmpl
```

```
    ;.jmp AnaComp
```

```
INT1:
```

```
ICP1:
```

```
OCA:
```

```
OCB:
```

```
  reti
```

```
Reset:
```

```
  .jmp @Test
```

```
end;
```

```
// End of assembler entry
```

```
end;
```

```
// End of dummy procedure
```

```
{#p code}
```

```
Const
```

```
SUART_BufSize = 16;
```

```
// Software UART Constants
```

```
SUART_BufMask = 15;
```

```
// 1,2,4,8,16,32,64,128 or 256 bytes
```

```
SUART_RDR = 0;
```

```
// must be one less than the buffer size
```

```
// Flags in software UART statusbyte
```

```
SUART_MSGDone = 1;
SUART_TD = 6;
SUART_BUSY = 7;
```

```
Var
```

```

// Software UART variables
Status, Bitcount, Reload, Buffer : Byte; // working registers
SUART_RxData : Byte; // Data received
SUART_BufData : Byte; // Data to be transmitted
SUART_Buf : array[0..SUART_BufSize-1] of Byte; // Buffer to store SUART incoming data
SUART_Head, SUART_Tail, Tmphead, Tmptail : Byte; // Buffer head and tail pointers
// Other variables
DelCount : Word; // Count value for delay generator
Counter : Byte; // General purpose count variable
```

```
//----- Software UART Section -----
```

```

Procedure SUART_Init; // Init Software UART
Begin
  Counter := 0;
  SUART_Tail := 0; // Initialise buffer pointers
  SUART_Head := 0;
  Repeat // Clear SUART buffer
    SUART_Buf[Counter] := 0;
    inc(Counter);
  until Counter = SUART_BufSize;
  Asm
    PreScScltTim0 equ 2 ; CK/8
    PreScScltTim1 equ 1 ; CK
    Prescale equ 8 ; C Baudrate 9600 bps
    CountVal equ 104 ; N
    ldi R16, PreScScltTim0 ; R=2 -- prescaler select
    out TCCR0, R16 ; Start timer0 and set clock source
    ldi R16, 01000000B ; Set INTO bit in GIMSK
    out GIMSK, R16 ; Enable external interrupt 0
    ldi R16, 00000010B ; enable ISC01 bit to set interrupt...
    out MCUCR, R16 ; ...on falling edges
    clr R16 ; Erase status-byte
    sts Test|Status, R16 ; Clear Statusbuffer
    sts Test|Bitcount, R16 ; Clear Bitcounter
  end;
end;

Procedure SUART_Rx; // Ext int handler for software UART RX
Begin
  Asm
    push R20
    push R16
    in R16, SREG
    push R16 ; Store Status Register
    in R16, GIMSK ; Mask bit INTO
    andi R16, 10111111B ; Disable external interrupt
    out GIMSK, R16
    ldi R16, 01000000B ; Set busy-flag (clear all others)...
    sts Test|Status, R16 ; ...in status register
```

```

ldi R16,(256-(CountVal+CountVal/2)+(29/Prescale)); Set timer reload-value (to 1.5
out TCNT0,R16 ; bit len)for start bit. 29 = time delay that
; have already been used in this
; interrupt plus the time
; that will be used by the time
; delay between timer interrupt request
; and the actual sampling of the first
; data bit.

ldi R16,00000010B ; Set bit TOV0 and TOV1...
out TIFR,R16 ; ...to clear Timer0&Timer1 overflow flags
out TIMSK,R16 ; Set bit TOIE0 to enable Tim0 ovrf int

Start_Timer1:
ldi R19,PreScSlctTim1 ; Load prescaler select, start timer1
out TCCR1B,R19 ; Start Timer1 for message timeout(CS10=1)
ldi R19,$D9 ; High byte of 11011001 01111100 (55 676)
ldi R20,$7C ; Low byte of the above
out TCNT1H,R19 ; Timer1 timeout reload value
out TCNT1L,R20
clr R16
sts Test|Bitcount,R16 ; Clear bit counter
ldi R16,(256-CountVal+(8/Prescale)) ; Set reload-value (constant)=10011001
sts Test|Reload,R16

Wait:
sei ; Enable interrupts again (for timer int)
; lds R17,Test|Status
; Test if RDR bit is set in status(bit0 in R17)
; If not, wait for timer ints to finish

sbrs R17,0
jmp Wait
clr R17
sts Test|Status,R17 ; Clear Status byte
ldi R19,01000000B ; Clear INT0 flag and...
out GIFR,R19 ; ...enable external Int0
out GIMSK,R19
ldi R19,10000000B ; Clear timer overflow flags
out TIFR,R19 ; Disable Timer0 int, enable Timer1 ovf int
out TIMSK,R19 ; Store registers used in Add_SUARTBuffer
push R21
push R20
push R17
push R16
end; // Pascal procedure Add_SUARTBuffer

Tmphead := ( SUART_Head + 1 ) and SUART_BufMask;// Calculate new buffer index
SUART_Head := Tmphead; // store new index
If Tmphead > SUART_BufSize then // buffer rollover
    Begin
        SUART_Head := 0; // reset buffer pointers
        Tmphead := 0;
    end;
SUART_Buf[Tmphead] := SUART_RxData; // store received data in buffer
_PORTA := SUART_Buf[Tmphead]; // TEST

Asm
pop R16 ; Restore regs used in Add_SUARTBuffer
pop R17
pop R20
    
```

```

    pop R21
    pop R16
    out SREG,R16                ; Restore SREG
    pop R16
    pop R20
    reti
end;
end;

Procedure Tim0_overflow_Int;    // Timer0 overflow interrupt
Begin
    Asm
        in R19,SREG
        push R19                ; Store statusregister
        lds R19,Test|Reload
        out TCNT0,R19          ; Reload timer
        lds R16,Test|Bitcount
        inc R16                 ; Increment bit counter
        lds R17,Test|Status
        sbrs R17,7              ; if transmit-bit (TD) set
        .jmp tim0_receive      ; goto receive
        sbrc R16,3              ; if bit 3 in bitcount (>7) is set...
        .jmp tim0_stopb        ; ...then jump to stop-bit routine
        lds R18,Test|Buffer
        sbrc R18,0              ; if LSB in buffer is 1
        sbi PORTB,PB4          ; Set transmit to 1
        sbrs R18,0              ; if LSB in buffer is 0
        cbi PORTB,PB4          ; Set transmit to 0
        lsr R18                 ; Shift buffer right
        sts Test|Buffer,R18
        sts Test|Bitcount,R16
        pop R19
        out SREG,R19           ; Restore SREG
        reti

tim0_stopb:
    sbi PORTB,PB4              ; Generate stop-bit
    sbrs R16,0                 ; if bitcount==8 (stop-bit) then do another...
    .jmp tim0_ret              ; ...stop bit --> jump to exit
    clr R19
    sts Test|Status,R19        ; clear status flags
    out TIFR,R19               ; Clear timer overflow flag
    out TIMSK,R19              ; Disable timer interrupt
    ldi R19,01000000B
    out GIMSK,R19              ; ...enable external Int0
tim0_complete:                 ; bitcount = 9
tim0_ret:
    sbi PORTB,3                ; Test
    cbi PORTB,3
    sts Test|BitCount,R16      ; Store bit counter
    pop R19
    out SREG,R19               ; Restore status register
    reti

tim0_receive:

```

```

sec ; Set carry
sbis PIND,2 ; if PD2=LOW <=== SAMPLE HERE
clc ; clear carry
ror R15 ; Shift carry into data
in R19,SREG ; Store SREG
cpi R16,9 ; if bitcount!=9 (must sample stop-bit)
brne tim0_ret ; exit interrupt
out SREG,R19 ; Get old SREG
rol R15 ; Rotate back data (get rid of stop-bit)
ldi R19,10000000B ; Set TOV1 in TIFR to...
out TIFR,R19 ; ...clear timer 1 overflow flag
ldi R19,$D9 ; High byte of
ldi R20,$7C ; Low byte of the above
out TCNT1H,R19 ; Timer1 timeout reload value
out TCNT1L,R20
sts Test\SUART_RxData,R15 ; Store rx byte to SRAM(SUART_RxData)
ldi R17,00000001B ; Clear busy,set Data Receive Ready(RDR)
sts Test>Status,R17 ; Store Statusbyte for external usage
jmp tim0_complete
end;
end;

```

```

Procedure Tim1_overflow_Int; // Returns true if GPS message timeout

```

```

Begin
  Asm
    push R16 ; Store R16
    in R16,SREG ; Store SREG
    push R16
    ldi R16,00000010B ; Set SUART_MSGDone bit in Status
    sts Test>Status,R16 ; Store status register
    sbi PORTB,0 ; Test
    cbi PORTB,0
    in R16,TIFR
    andi R16,11111111B
    out TIFR,R16 ; Clear TOV1 flag
    in R16,TIMSK
    andi R16,01111111B ; Clear TOIE1 bit in TIMSK
    out TIMSK,R16 ; Disable timer1 overflow interrupt
    pop R16 ; Restore R16
    out SREG,R16 ; Restore SREG
    pop R16
    reti
  end;
end;

```

```

Procedure Read_SUARTBuffer; // Procedure to read from Software UART buffer

```

```

Begin
  Tmptail := ( SUART_Tail + 1 ) and SUART_BufMask; // calculate buffer index
  SUART_Tail := Tmptail; // store new index
  SUART_BufData := SUART_Buf[Tmptail]; // return data
  _PORTA := SUART_BufData; // TEST
end;

```

```

Procedure SUART_Tx( Data : byte ); // Software UART data send routine
Begin

```


//----- END -----

G.2 System Test Program Code

// System Test (Backplane) program

link 0,\$60

Program Backplane; Vector Shortjmp 0;
 Uses Intlib;

// Changed JMP to .JMP in original
 // "intlib.asm" library

Const

GreenLED = 6; // PortD pins
 ReadFIFO = 5;
 TrigOUT = 4; // Trigger generated FOR GPS board
 EN_FIFODATA = 3; // Enable FIFO data read
 NotEmpty = 2;

TrigCnt = 10;

PCBaud = 51; // 9600 baud UART connected to PC

Var

Delcount : word;
 TrigCount : byte;
 Flag : byte;
 Count : byte;
 PCInData : byte;
 Data: byte;

Procedure Delay(DelayValue : Word); // Delay value

Begin
 Delcount := 0;
 while DelCount < DelayValue do
 Inc(DelCount);
end;

Procedure Generate_Trigger;

Begin
 SetIOBit(_PORTD,TrigOUT);
 delay(7000);
 ClrIOBit(_PORTD,TrigOUT);
end;

Procedure Gen_TrigInt; Vector Shortjmp INT1addr; // Switch on LED when Gen_trigger is received

begin
 Interrupt_entry;
 ClrIOBit(_PortD,GreenLED);
 Delay(20000);
 SetIOBit(_PortD,GreenLED);
 Interrupt_exit;

```

end;

Procedure InitPCUART; // initialize PC UART0
begin
  _UBRR := PCBaud; // set Baud rate for 8MHz crystal
  _UCR := __RXCIE or __RXEN or __TXEN; // enable Rx, Tx & Rx complete int for PC
end;

Procedure ReadFIFOData;
Var
  FIFODATA : byte;
begin
  while IOBitHigh(_PinD,NotEmpty) do
    begin // Until FIFO is empty (/EF = low)
      SetIOBit(_PortD,5); // Initiate read from FIFO
      SetIOBit(_PortD,EN_FIFODATA); // Open latch
      Delay(500);
      Data := _PinB;
      ClrIOBit(_PortD,5); // Stop read
      Delay(100);
      ClrIOBit(_PortD,EN_FIFODATA); // Open latch
      _UDR := Data;
      while IOBitLow(_USR,_UDRE) do; // wait for Tx to finish
    end;
  end;

Procedure PC_RxInt; Vector ShortJmp URXCaddr; // Rx from PC complete interrupt routine
begin
  Interrupt_Entry;
  PCInData := _UDR; // read received data
  if PCInData = $52 then Flag := 1 else Flag := 0;
  Interrupt_Exit;
end;

Procedure WaitForData;
begin
  Repeat until Flag = 1;
  Flag := 0;
  ReadFIFOData;
end;

Procedure InitPorts;
begin
  _DDRB := $00; // Inputs to read FIFO
  _PORTB := $00;
  _DDRD := %01111000; // PD6,PD5,PD4 set as outputs
  _PORTD := %01000000; // LED off
  _MCUCR := __ISC11 or __ISC10; // Rising edge of trigger calls interrupt
  _GIMSK := __INT1; // Enable interrupt 0 (trigger interrupt)
  TrigCount := 0;
  Count := 0;
  Flag := 0; // enable global interrupts
  InitPCUART;
  EI;
  SetIOBit(_PORTD,TrigOUT);

```



```

Not_Ready    = 5;           // System ready signal (O)
Not_Write    = 6;           // Write data signal (O)
Not_Read     = 7;           // Read data signal (O)
                                   // PortE pins
RedLED       = 1;           // PortE red LED(O)
GreenLED     = 2;           // PortE green LED(O)

GPSBaud      = 62;           // 9600 baud UART1 connected to GPS
GPS_RxBuf_Size = 32;        // 1,2,4,8,16,32,64,128 or 256 bytes
GPS_RxBuf_Mask = 31;

PCBaud       = 62;           // 9600 baud UART0 connected to PC
PC_RxBuf_Size = 64;         // 1,2,4,8,16,32,64,128 or 256 bytes
PC_TxBuf_Size = 256;        // 1,2,4,8,16,32,64,128 or 256 bytes
PC_RxBuf_Mask = 63;
PC_TxBuf_Mask = 255;

GenTrigDataSize= 64;        // Trigger buffers sizes
GenTrigDataMask= 63;
TrigDataSize  = 256;
TrigSizeMask  = 255;

SetRTCSec     = $11;        // RTC register addresses
SetRTCMin    = $12;
SetRTCHrs    = $13;
SetCompSec   = $14;
SetCompMin   = $15;
SetCompHrs   = $16;
RTCSecOut    = $17;
RTCMinOut    = $18;
RTCHrsOut    = $19;

UsCountHigh  = $81;        // Microsecond counter register addresses
UsCountMid   = $82;
UsCountLow   = $83;
UsCompHigh   = $84;
UsCompMid    = $85;
UsCompLow    = $86;

FIFOWrite    = $87;        // FIFO control addresses
FIFORead     = $88;
FIFOReset    = $89;

GPS_OK_LED_ON    = $8A;    // Indicator LEDs ON
Ready_LED_ON     = $8B;
Trigger_LED_ON   = $8C;
GPS_OK_LED_OFF  = $8D;    // Indicator LEDs OFF
Ready_LED_OFF    = $8E;
Trigger_LED_OFF  = $8F;

PPSCountValue = 5;         // Amount of seconds to warm start GPS

StoreToRTC     = True;     // Store to RTC commands
DontStoreToRTC = False;

var
GPS_RxBuf      : array[0..GPS_RxBuf_Size-1] of byte; // GPS UART Rx buffer

```

```

GPS_RxHead  : byte;
GPS_RxTail  : byte; // GPS Rx buffer head & tail pointers
Command     : CmdString; // Storage space for message to be sent to GPS
GPSErr      : byte; // GPS Error status register
PPSCount    : word; // Count variable for PPS counter

PC_RxBuf    : array[0..PC_RxBuf_Size-1] of byte; // PC UART Rx buffer
PC_TxBuf    : array[0..PC_TxBuf_Size-1] of byte; // PC UART Tx buffer
PC_RxHead   : byte;
PC_RxTail   : byte; // PC Rx buffer head & tail pointers
PC_TxHead   : byte;
PC_TxTail   : byte; // PC Tx buffer head & tail pointers
CRC16       : word; // CRC16 word variable
TxDoneFlag  : byte; // Is transmit done?

GenTrigData : array[0..GenTrigDataSize-1] of byte; // Buffer for generated trigger data
GenTrigHead : byte;
GenTrigTail : byte;
GenTrigCounter: byte; // Count generated triggers

TriggerData : array[0..TrigDataSize-1] of byte; // Copy of FIFO memory
TrigHead    : byte; // Trigger data buffer pointers
TrigTail: byte;

RegData     : byte; // Temp register for data from databus
Temp        : byte;
Store       : boolean; // Should GPS time be stored to RTC?

Seconds     : byte; // Storage registers for GPS time
Minutes     : byte;
Hours       : byte;
Month       : byte; // Storage for date (to be stored in FIFO)
Day         : byte;
YearHigh    : byte; // Century
YearLow     : byte; // Decade

TrigSec     : byte; // Time when trigger was received
TrigMin     : byte; // This is also stored in FIFO
TrigHr      : byte;
TrigUsHigh  : byte;
TrigUsMid   : byte;
TrigUsLow   : byte;
TrigCounter : byte; // Count amount of triggers received

CompareHrs  : byte; // Temporary storage for compare registers
CompareMin  : byte;
CompareSec  : byte;
CompareUsHi : byte;
CompareUsMid : byte;
CompareUsLow : byte;

```

//----- EPLD Routines -----

```

Procedure Delay(DelayValue : Word); // Delay generation procedure
Var Delcount : word;

```



```

begin
  GPSErr := 1;
  SetIOBit(_PortE,RedLED);
  WriteData(0,GPS_OK_LED_OFF);
end;

Procedure GPS_OK;
begin
  If GPSErr = 0 then WriteData(0,GPS_OK_LED_ON); // switch on 'GPS OK' LED
end;

Procedure System_Ready; // Switch on 'System Ready' LED
begin
  If GPSErr = 1 then exit; // If other_error then exit; System NOT ready
  WriteData(0,Ready_LED_ON);
end;

Procedure System_Busy;
begin
  WriteData(0,Ready_LED_OFF); // System busy
end;

//----- PC (UART0) Communcation routines -----

Procedure ClearPCRxBuffer; // clear PC buffers by clearing pointers
begin
  PC_RxTail := 0;
  PC_RxHead := 0;
end;

Procedure ClearPCTxBuffer;
begin
  PC_TxTail := 0;
  PC_TxHead := 0;
end;

Procedure InitPCUART; // initialize PC UART0
begin
  _UBRRHI := 0; // Hi byte of UBRR0
  _UBRR0 := PCBaud; // set Baud rate for 4MHz crystal
  _UCSR0B := __RXCIE0 or __RXEN0 or __TXEN0; // enable Rx, Tx & Rx complete int for PC
  ClearPCRxBuffer;
  ClearPCTxBuffer;
end;

Function DataInPCRxBuf : boolean; // return 0 (FALSE) if the receive buffer is empty
begin
  DataInPCRxBuf := PC_RxHead <> PC_RxTail;
end;

Function RxByteFromPC : byte; // Read data from PC Rx buffer
var // ...used by 'Service PC command' routine
  Tmptail : byte;
begin
  if not(DataInPCRxBuf) then exit; // No data from PC --> exit

```

```

    Tmptail := (PC_RxTail + 1) and PC_RxBuf_Mask;           // calculate buffer index
    PC_RxTail := Tmptail;                                   // store new index
    RxByteFromPC := PC_RxBuf[Tmptail];                     // return data
end;

Procedure TxByteToPC(Data : byte);                          // Enables UART data reg empty interrupt
var
    Tmphead : byte;
begin
    System_busy;
    Tmphead := (PC_TxHead + 1) and PC_TxBuf_Mask;         // calculate buffer index
    while Tmphead = PC_TxTail do;                          // wait for free space in buffer
    PC_TxBuf[Tmphead] := Data;                             // store data in buffer
    PC_TxHead := Tmphead;                                  // store new index
end;

Procedure PC_RxInt; Vector LongJmp URXC0addr;              // Rx from PC complete interrupt routine
var
    Data, Tmphead : byte;
begin
    Interrupt_Entry;
    System_Busy;
    Data := _UDR0;                                         // read received data
    Tmphead := (PC_RxHead + 1) and PC_RxBuf_Mask;         // calculate buffer index
    PC_RxHead := Tmphead;                                  // store new index
    if Tmphead = PC_RxTail then ClearPCRxBuffer;          // if buffer overflow, do something here
    PC_RxBuf[Tmphead] := Data;                             // store received data in buffer
    _TCCR1B := $02;                                       // Start timer for PC comms timeout (CK/8)
    _TIFR := __TOV1;                                      // Clear TOV1 flag in TIFR
    _TIMSK := __TOIE1;                                    // Timer1 overflow interrupt enable
    _TCNT1H := $FC;                                       // Reload timer for 2.13ms timeout
    _TCNT1L := $00;
    Interrupt_Exit;
end;

Function CRC16Correct : Boolean;                            // Validate Rx message CRC16
var
    CRC16_Ctr : byte;
    Index : byte;
begin
    CRC16Correct := false;
    CRC16 := $FFFF;
    Index := 1;
    For Index := 1 to (PC_RxHead) do
        begin
            CRC16 := CRC16 xor PC_RxBuf[Index];
            CRC16_Ctr := 0;
            while CRC16_Ctr < 8 do
                begin
                    if (CRC16 and 1) = 1 then
                        begin
                            CRC16 := CRC16 shr 1 ;
                            CRC16 := CRC16 xor 40961;
                        end else CRC16 := CRC16 shr 1 ;
                    inc(CRC16_Ctr);
                end
            end
        end
    end
end;

```



```

    end;
  end;
  If CRC16 = 0 then CRC16Correct := True
    else CRC16Correct := False;           // if CRC(Data + CRC) = 0 then data is valid
end;

Procedure CalcCRC16;                       // Calculate CRC-16 on data to be transmitted
var
  CRC16_Ctr : byte;
  Index      : byte;
begin
  CRC16 := $FFFF;
  Index := 1;
  For Index := 1 to (PC_TxHead) do
    begin
      CRC16 := CRC16 xor PC_TxBuf[Index];
      CRC16_Ctr := 0;
      while CRC16_Ctr < 8 do
        begin
          if (CRC16 and 1) = 1 then
            begin
              CRC16 := CRC16 shr 1 ;
              CRC16 := CRC16 xor 40961;
            end else CRC16 := CRC16 shr 1 ;
          inc(CRC16_Ctr);
        end;
      end;
    TxByteToPC(lo(CRC16));                 // Add CRC16 bytes to message
    TxByteToPC(hi(CRC16));
  end;

Procedure StartTransmit;                   // Transmit data in Tx buffer
begin
  CalcCRC16;                               // Calculate CRC16 of Tx message
  TxDoneFlag := 0;
  SetIOBit(_UCSR0B,_UDRIE0);              // enable UDRE0 interrupt--> start Tx
  while TxDoneFlag = 0 do;                 // wait for transmit to finish
    System_Ready;
  end;

Procedure PCUDREEmptyInt; Vector Long.Jmp UDRE0addr; // Interrupt procedure that handles PC Tx
var
  Tmptail : byte;
begin
  Interrupt_Entry;
  TxDoneFlag := 0;
  if PC_TxHead <> PC_TxTail then           // All data in TXbuffer transmitted?
    begin
      Tmptail := (PC_TxTail + 1) and PC_TxBuf_Mask; // calculate buffer index
      PC_TxTail := Tmptail;                  // store new index
      _UDR0 := PC_TxBuf[Tmptail];          // start transmission
    end else
      Begin
        ClrIOBit(_UCSR0B,_UDRIE0);         // disable UDRE interrupt
        TxDoneFlag := 1;                   // Transmit is done
      end;

```



```

begin
  TrigHead := 0;
  TrigTail := 0;
end;

Function TrigDataInBuf : boolean;           // return false if buffer empty
begin
  TrigDataInBuf := TrigHead <> TrigTail;
end;

Function TrigBufData : byte;               // Get trigger data from software data buffer
var
  Tmptail : byte;
begin
  if not(TrigDataInBuf) then exit;         // Exit subroutine if trig buffer is empty
  Tmptail := (TrigTail + 1) and TrigSizeMask; // calculate buffer index
  TrigTail := Tmptail;                    // store new index
  TrigBufData := TriggerData[Tmptail];    // return data
end;

Procedure StoreTrigDataBuf(Data : byte);   // Store trigger data in Software FIFO
Var
  Tmphead : byte;
begin
  Tmphead := (TrigHead + 1) and TrigSizeMask; // calculate buffer index
  TrigHead := Tmphead;                       // Store new index
  if Tmphead = TrigTail then ClearTrigBuffer; // software data buffer overflow
  TriggerData[Tmphead] := Data;              // store data in buffer
end;

Procedure StoreTrigData;                   // Store data in FIFO memory & software data
buffer
begin
  WriteData(TrigCounter,FIFOWrite);         // '#dmhmsuuu<CR>'
  WriteData(day,FIFOWrite);                 // FIFO message constructed and stored in FIFO
  WriteData(month,FIFOWrite);
  WriteData(TrigHr,FIFOWrite);
  WriteData(TrigMin,FIFOWrite);
  WriteData(TrigSec,FIFOWrite);
  WriteData(TrigUsHigh,FIFOWrite);
  WriteData(TrigUsMid,FIFOWrite);
  WriteData(TrigUsLow,FIFOWrite);
  //WriteData($0A,FIFOWrite);               // Message ended with a <CR>
  StoreTrigDataBuf(TrigCounter);           // Store data in software FIFO
  StoreTrigDataBuf(day);
  StoreTrigDataBuf(month);
  StoreTrigDataBuf(TrigHr);
  StoreTrigDataBuf(TrigMin);
  StoreTrigDataBuf(TrigSec);
  StoreTrigDataBuf(TrigUsHigh);
  StoreTrigDataBuf(TrigUsMid);
  StoreTrigDataBuf(TrigUsLow);
  //StoreTrigDataBuf($0A);                 // Trigger message ended with a <CR>
end;

```

```

Procedure TrigEvent; Vector LongJmp INT1addr;           // Trigger interrupt handler
begin
  Interrupt_Entry;
  SetIOBit(_PortD,Not_Ready);                          // System NOT ready for further triggers
  inc(TrigCounter);                                    // Number of this received trigger signal
  WriteData(0,Trigger_LED_ON);
  WriteData(0,Ready_LED_OFF);
  GetData(TrigHr,RTCHrsOut);
  GetData(TrigMin,RTCMinOut);
  GetData(TrigSec,RTCSecOut);
  GetData(TrigUsHigh,UsCountHigh);                    // Get microsecond count
  GetData(TrigUsMid,UsCountMid);
  GetData(TrigUsLow,UsCountLow);
  StoreTrigData;                                       // Store data in FIFO & Software data buffer
  Delay(700);
  WriteData(0,Trigger_LED_OFF);
  WriteData(0,Ready_LED_ON);
  ClrIOBit(_PortD,Not_Ready);                          // System READY for trigger signals
  Interrupt_Exit;
end;

```

//----- PC command handler section -----

```

Procedure SendTriggerTime;                             // (1) Send trigger and compare data to PC
begin
  //TxByteToPC($31);                                   // Echo command number
  Repeat
    TxByteToPC(TrigBufData);                          // Send trigger data
  until TrigTail = Trighead;
  //if IOBitLow(_PinB,FIFO_FF) then TxByteToPC($46)
  // else TxByteToPC(0);                               // Transmit "F" to PC if FIFO is full
  TxDoneFlag := 0;
  SetIOBit(_UCSR0B,_UDRIE0);                          // enable UDRE0 interrupt--> start Tx
  while TxDoneFlag = 0 do;                             // wait for transmit to finish
  System_Ready;
  TrigTail := 0;                                       // So that data can be retransmitted
end;

```

```

Procedure SetCompare;                                 // (2) Set time of next generated trigger
Var
  Second,Minute,Hour : byte;
  CompSec,CompMin,CompHour : byte;
  RTCTime : Triplet absolute Second;
  CompareTime : Triplet absolute Compsec;
begin
  TxByteToPC($32);                                    // Echo command number
  GetData(Hour,RTCHrsOut);                             // Read EPLD RTC
  GetData(Minute,RTCMinOut);
  GetData(Second,RTCSecOut);
  CompSec := CompareSec;                              // Read compare registers
  CompMin := CompareMin;
  CompHour := CompareHrs;
  if RTCTime>CompareTime then
    begin
      LogGenTrig;                                     // Last comparetime is past, log it
    end;
end;

```

```

end;
CompareHrs := RxByteFromPC;           // Read compare time from PC
CompareMin := RxByteFromPC;
CompareSec := RxByteFromPC;
CompareUsHi := RxByteFromPC;
CompareUsMid := RxByteFromPC;
CompareUsLow := RxByteFromPC;
CompSec := CompareSec;               // Read compare registers
CompMin := CompareMin;
CompHour := CompareHrs;
if CompareTime>RTCTime then         // Received VALID compare time
begin
  TxByteToPC($32);
  WriteData(CompareHrs,SetCompHrs);  // Set Compare registers
  WriteData(CompareMin,SetCompMin);
  WriteData(CompareSec,SetCompSec);
  WriteData(CompareUsHi,UsCompHigh);
  WriteData(CompareUsMid,UsCompMid);
  WriteData(CompareUsLow,UsCompLow);
end
else TxByteToPC($33);               // Received INVALID compare time
StartTransmit;
end;

Procedure ReadCompare;               // (3) Get time of previous generated trigger
begin
  TxByteToPC($33);                  // Echo command number
  TxByteToPC(CompareHrs);
  TxByteToPC(CompareMin);
  TxByteToPC(CompareSec);
  TxByteToPC(CompareUsHi);
  TxByteToPC(CompareUsMid);
  TxByteToPC(CompareUsLow);
  StartTransmit;
end;

Procedure GetTime(DoStore : boolean); Forward; // Forward declarations
Procedure GetDate; Forward;

Procedure ReadRTC;                   // Read RTC and send to PC
Var
  Secs,Mins,Hrs : byte;
begin
  GetData(Hrs,RTCHrsOut);           // Read EPLD RTC
  GetData(Mins,RTCMinOut);
  GetData(Secs,RTCSecOut);
  TxByteToPC(Hrs);
  TxByteToPC(Mins);
  TxByteToPC(Secs);
end;

Procedure CompareTimes;              // (part of 4) Compare GPS&RTC time,send error
Var
  TimeError : byte;
  Secs,Mins,Hrs : byte;

```

```

begin
  GetData(Hrs,RTCHrsOut);           // Read EPLD RTC
  GetData(Mins,RTCMinOut);
  GetData(Secs,RTCSecOut);
  TimeError := Hours - Hrs;
  TxByteToPC(TimeError);
  TimeError := Minutes - Mins;
  TxByteToPC(TimeError);
  TimeError := Seconds - Secs;
  TxByteToPC(TimeError);
end;

Procedure GPS_RTC_Status;          // (4) Get GPS time/date from GPS
Var
  Secs,Mins,Hrs : byte;
begin
  TxByteToPC($34);                // Echo Command number
  GetDate;
  TxByteToPC(day);
  TxByteToPC(month);
  TxByteToPC(YearHigh);
  TxByteToPC(YearLow);
  GetTime(DontStoreToRTC);        // Get GPS Time and send to PC
  ReadRTC;
  CompareTimes;
  StartTransmit;
end;

Procedure SetRTC;                  // (5) Read GPS time and store to RTC
begin
  TxByteToPC($35);                // Echo command number
  GetTime(StoreToRTC);            // Carry out command
  StartTransmit;                  // Reply to PC
end;

Procedure ClearFIFO;               // (6) Clear FIFO memory (test procedure)
begin
  TxByteToPC($36);                // Echo command number
  ClearTrigBuffer;
  Repeat
    WriteData(0,FIFORead);        // Read FIFO until empty
  until IOBitLow(_PinB,FIFO_EF);  // Thus clearing FIFO memory
  StartTransmit;
end;

Procedure TestFIFO;                // (7) Fill FIFO memory (test procedure)
var
  Text : string[20];
  index : byte;
  Data : char;
  Store : byte;
begin
  TxByteToPC($37);                // Echo command number
  Text := 'Katzenellenbogen';
  Repeat

```

```

For index := 1 to length(Text) do
begin
  Data := Text[index];
  Store := byte(Data);
  WriteData(Store,FIFOWRITE);
end;
until IOBitLow(_PinB,FIFO_FF);
StartTransmit;
end;

// Fill FIFO memory with "Katzenellenbogen"

Procedure SendGenTrigData;
begin
  // (8) Send generated trigger data to PC
  TxByteToPC($38);
  // Echo command number
  Repeat
  TxByteToPC(GenTrigBufData);
  // Send generated trigger data
until GenTrigTail = GenTrighead;
TxDoneFlag := 0;
SetIOBit(_UCSR0B,_UDRIE0);
// enable UDRE0 interrupt--> start Tx
// wait for transmit to finish
while TxDoneFlag = 0 do;
System_Ready;
GenTrigTail := 0;
// So that data can be retransmitted
end;

Procedure ClearGenTrig;
begin
  // (9) Clear generated trigger buffer
  TxByteToPC($39);
  // Echo command number
  ClearGenTrigBuffer;
  StartTransmit;
end;

Procedure SendErrorMsg;
begin
  // Error in communications
  TxByteToPC($45('E'));
  // E = 'Error'
  StartTransmit;
  ClearPCTxBuffer;
  ClearPCRxBuffer;
end;

Procedure ServiceComms;
Var
  CommandByte : byte;
begin
  // Respond to command received from PC
  If CRC16Correct = False then
  begin
  // CRC16 on rx message is invalid
  SendErrorMsg;
  // Send a error message to PC
  exit;
end;
CommandByte := RxByteFromPC;
// Read command byte
ClearPCTxBuffer;
// Clear buffer for data to be sent
Case CommandByte of
  $31 : SendTriggerTime;
  // (1) Send trigger data
  $32 : SetCompare;
  // (2) Set time of trigger to be generated next
  $33 : ReadCompare;
  // (3) Send time of previous generated trigger
  $34 : GPS_RTC_Status;
  // (4) Send GPS,RTC and RTC error status to PC
  $35 : SetRTC;
  // (5) Set RTC with GPS time
end;

```

```

$36 : ClearFIFO; // (6) Clear FIFO memory
$37 : TestFIFO; // (7) Fill FIFO with test data
$38 : SendGenTrigData; // (8) Send generated trigger data to PC
$39 : ClearGenTrig; // (9) Clear generated trigger buffer
end;
ClearPCRxBuffer; // Get ready for next command
end;

Procedure Tim1_Ovflnt; Vector LongJump OVF1addr; // PC command service interrupt routine
begin
Interrupt_Entry;
_TIFR := __TOV1; // Clear timer1 overflow flag
_TIMSK := 0; // Disable timer1 overflow interrupt
_TCCR1B := 0; // Stop timer1
EI;
ServiceComms; // Process data received from PC
System_Ready;
Interrupt_Exit;
end;

//----- GPS (UART1) communication routines -----

Procedure ClearGPSRxBuffer;
begin
GPS_RxTail := 0;
GPS_RxHead := 0; // Clear receive buffer
end;

Procedure InitGPSUART; // initialize GPS UART1
begin
_UBRRHI := 0; // Hi byte of UBRR1
_UBRR1 := GPSBaud; // set Baud rate for 4MHz crystal
SetIOBit(_UCSR1B,_TXEN1); // switch on UART1 transmitter
ClearGPSRxBuffer;
end;

Procedure WaitForGPSReply; // Receive message string from GPS
begin
SetIOBit(_UCSR1B,_RXCIE1);
SetIOBit(_UCSR1B,_RXEN1); // Enable receiver & Rx interrupt (UART1)
while IOBitLow(_TIFR,_TOV1) do; // Wait here for a GPS message to be received
_TIFR := __TOV1; // Clear timer overflow flag
_TCCR1B := 0; // Stop the rx 'timeout' timer1
System_Ready;
end;

Function DataInGPSRxBuf : boolean; // return FALSE if the GPS rx buffer is empty
begin
DataInGPSRxBuf := GPS_RxHead <> GPS_RxTail;
end;

Function ByteInGPSRxBuf : byte; // Get byte from rx buffer
var
Tmptail : byte;
begin

```



```

if not(DataInGPSRxBuf) then exit;           // Exit subroutine if rx buffer is empty
TmpTail := (GPS_RxTail + 1) and GPS_RxBuf_Mask; // calculate buffer index
GPS_RxTail := TmpTail;                     // store new index
ByteInGPSRxBuf := GPS_RxBuf[TmpTail];     // return data
end;

Procedure GPS_RxInt; Vector LongJmp URXC1addr; // 'Rx from GPS complete' interrupt routine
var
  Data, Tmphead : byte;
begin
  Interrupt_Entry;
  Data := _UDR1;                             // read received data
  System_Busy;
  Tmphead := (GPS_RxHead + 1) and GPS_RxBuf_Mask; // calculate buffer index
  GPS_RxHead := Tmphead;                     // store new index
  if Tmphead = GPS_RxTail then ClearGPSRxBuffer; // Receive buffer overflow
  GPS_RxBuf[Tmphead] := Data;               // store received data in buffer
  _TCCR1B := $02;                           // Start timer for GPS message timeout (CK/8)
  _TIFR := __TOV1;                           // Clear TOV1 flag in TIFR
  _TIMSK := 0;                               // Overflow interrupt disable
  _TCNT1H := $FC;                            // Reload timer for 2.13ms timeout
  _TCNT1L := $00;
  Interrupt_Exit;
end;

Procedure StoreTime;                       // Save time(received from GPS) to RTC
begin
  GPS_RxTail := 4;                          // Reset pointer to start of data
  Hours := ByteInGPSRxBuf;                  // Get GPS time from GPS Rx buffer
  Minutes := ByteInGPSRxBuf;
  Seconds := ByteInGPSRxBuf;
  If Store = StoreToRTC then
    begin
      WriteData(Seconds, SetRTCSec);
      WriteData(Minutes, SetRTCMin);
      WriteData(Hours, SetRTCHrs);
      SetIOBit(_PortB, RTCSet);             // Set RTC with GPS time
      ClrIOBit(_PortB, RTCSet);
    end else
    begin
      TxByteToPC(Hours);                   // Send GPS time to PC
      TxByteToPC(Minutes);
      TxByteToPC(Seconds);
    end;
end;

Procedure StoreDate;                       // Keep date when needed for FIFO store
begin
  GPS_RxTail := 4;
  Month := ByteInGPSRxBuf;
  Day := ByteInGPSRxBuf;
  YearHigh := ByteInGPSRxBuf;
  YearLow := ByteInGPSRxBuf;
end;

```



```

Procedure GetTime(DoStore : boolean);           // Get time from GPS and store it in RTC
begin
  If DoStore = StoreToRTC then Store := StoreToRTC;
  If DoStore = DontStoreToRTC then Store := DontStoreToRTC;
  Command := '@@Aa'+#$FF+$FF+$FF+$SDF+$0D+$0A;
  SendCommand;                                 // Get time from GPS receiver
end;

Procedure GetDate;
begin
  Command := '@@Ac'+#$FF+$FF+$FF+$FF+$22+$0D+$0A;
  SendCommand;                                 // Get date from GPS receiver
end;

Procedure GPSReceiverSetup;
begin
  Command := '@@Aw'+#$01+$37+$0D+$0A;         // Time mode -->
  SendCommand;                                 // Set GPS to respond with UTC + GMT Offset
  Command := '@@Ab'+#$00+$02+$00+$21+$0D+$0A; // GMT offset -->
  SendCommand;                                 // +02:00
end;

Procedure StartGPS;                            // Procedure to startup GPS receiver
begin
  GPSErr := 0;
  PPSCount := 0;
  _MCUCR := __ISC01 or __ISC00;               // Rising edge of PPS calls interrupt
  _GIMSK := __INT0;                          // Enable Int0 (PPS)
  EI;                                         // enable global interrupts
  repeat until PPSCount = PPSCountValue;     // wait for pps interrupts
  GPSReceiverSetup;                          // Send GPS setup commands
  GetTime(StoreToRTC);                       // Set RTC from GPS time
  GetDate;                                   // Get Date from GPS
  System_Ready;
  _GIMSK := 0;
  _MCUCR := 0;
  _MCUCR := __ISC11 or __ISC10 or __ISC01 or __ISC00; // Rising edge of TrigInterrupt calls interrupt
  _GIMSK := __INT0 or __INT1;               // Enable Int0 (PPS)& Int1 (TrigInterrupt)
end;

//----- Init Routines -----

Procedure InitPorts;                           // Setup controller I/O ports
begin                                          // and port initial values
  _DDRA := 0;                                // Data bus -- set as input OR output
  _PortA := 0;                               // Tri-state data-bus for startup
  _DDRBB := %00011000;                       // Control signals
  _PortB := %00001000;                       // Init port values
  _DDRC := $FF;                              // Address bus -- always set as outputs
  _PortC := 0;                               // Write Zero out to address bus
  _DDRD := %11100010;                       // Control signals
  _PortD := %11110010;                       // Init port values
  _DDRE := $FF;                              // Outputs
  _PortE := 0;                               // Init port values

```

```

end;

Procedure ClearGeneratedTime;                               // Init Compare registers
Var
  Secs,Mins,Hrs : byte;
begin
  GetData(Hrs,RTCHrsOut);                                   // Read RTC time
  GetData(Mins,RTCMinOut);
  GetData(Secs,RTCSecOut);
  WriteData(Hrs,SetCompHrs);                               // Store Startup time
  WriteData(Mins,SetCompMin);
  WriteData(Secs,SetCompSec);
  WriteData(0,UsCompHigh);
  WriteData(0,UsCompMid);
  WriteData(0,UsCompLow);
  CompareHrs := Hrs;
  CompareMin := Mins;
  CompareSec := Secs;
  CompareUsHi := 0;
  CompareUsMid := 0;
  CompareUsLow := 0;
end;

Procedure Init;
begin
  InitPorts;
  WriteData(0,GPS_OK_LED_OFF);
  WriteData(0,Ready_LED_OFF);
  WriteData(0,Trigger_LED_OFF);
  InitGPSUART;                                           // Init UART connected to GPS receiver
  InitPCUART;                                           // Init UART connected to PC
  StartGPS;                                             // Init GPS Receiver
  TrigCounter := 0;
  GenTrigCounter := 0;
  ClrIOBit(_PortD,Not_Ready);                           // Ready to receive trigger signals
  ClearTrigBuffer;                                       // Clear Trigger Data
  Repeat
    WriteData(0,FIFORead);                               // Read FIFO until empty
  until IOBitLow(_PinB,FIFO_EF);                          // --> clearing FIFO memory
  WriteData(0,FIFOReset);                               // Reset FIFO
  ClearGenTrigBuffer;
  ClearGeneratedTime;
end;

Procedure PPSInterrupt; Vector LongJmp INTOaddr;         // Handles 1PPS interrupt from GPS at startup
begin
  interrupt_entry;
  SetIOBit(_PortD,Not_Ready);                            // Trigger signals blocked
  inc(PPSCount);
  if PPSCount = 3600 then
    Begin
      EI;                                                // Update RTC every 3600 seconds,
      GetTime(StoreToRTC);                               // or 1 hour (60 minutes)
      GetDate;                                           // Enable ints for trigger & GPS Comms
      PPSCount := 0;                                     // Store current GPS time to RTC
      PPSCount := 0;                                     // Get Date
      PPSCount := 0;                                     // Reset PPS counter
    End;
  End;
end;

```

```
    end;
    CclrIOBit(_PortD,Not_Ready);           // Ready to receive trigger signal again
    interrupt_exit;
end;

//----- Main program section -----

begin
    _SPL:=lo(_RAMEND);
    _SPH:=hi(_RAMEND);
    Init;
    Repeat until False;
end.

//----- END -----
```

Appendix H

LabVIEW™ Program Diagrams

This Appendix presents complete LabVIEW™ program diagrams, describing the *GPS Based Time Stamping and Scheduling System* PC control software. The program flow diagram is presented in chapter 7.

Front Panel

System status

PortStatus

ValidPort PortNr

Communication

Com Port: ● Communication Error Close

GPS Receiver & System Real Time Clock

GPS Date:

GPS Time:

RTC Time:

RTC Error: Update

Triggers

Received at: Clear FIFO

Update

Generated at: Clear Buffer

Update

Configure
Exit

PortStatus 2

ValidPort PortNr

Triggers

▼ 0



Index	Day	Month	Hours	Minutes	Seconds	Microsecs
0	1	1	0	0	0	0

Generated Triggers

▼ 0

Index	Day	Month	Hours	Minutes	Seconds	Microsecs
0	1	1	0	0	0	0

Controls and Indicators

-  **Configure**
-  **Exit**


 **Close Port**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **Update GPS&RTC**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **Update RXTriggers**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **Update Gen Trigger**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **PortStatus**

 **ValidPort**

 **PortNr**

 **Clear FIFO**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **Clear GenTrigBuf**
Click here to set GPS Timestamp system onboard RTC to GPS time

 **Generated at:**

 **Communication Error**

 **PortStatus 2**

 **ValidPort**

 **PortNr**

 **Com Port**

 **GPS Date**

 **GPS Time**

 **RTC Time**

 **RTC Error**

 **Received at:**

 **Triggers**

 **Cluster**

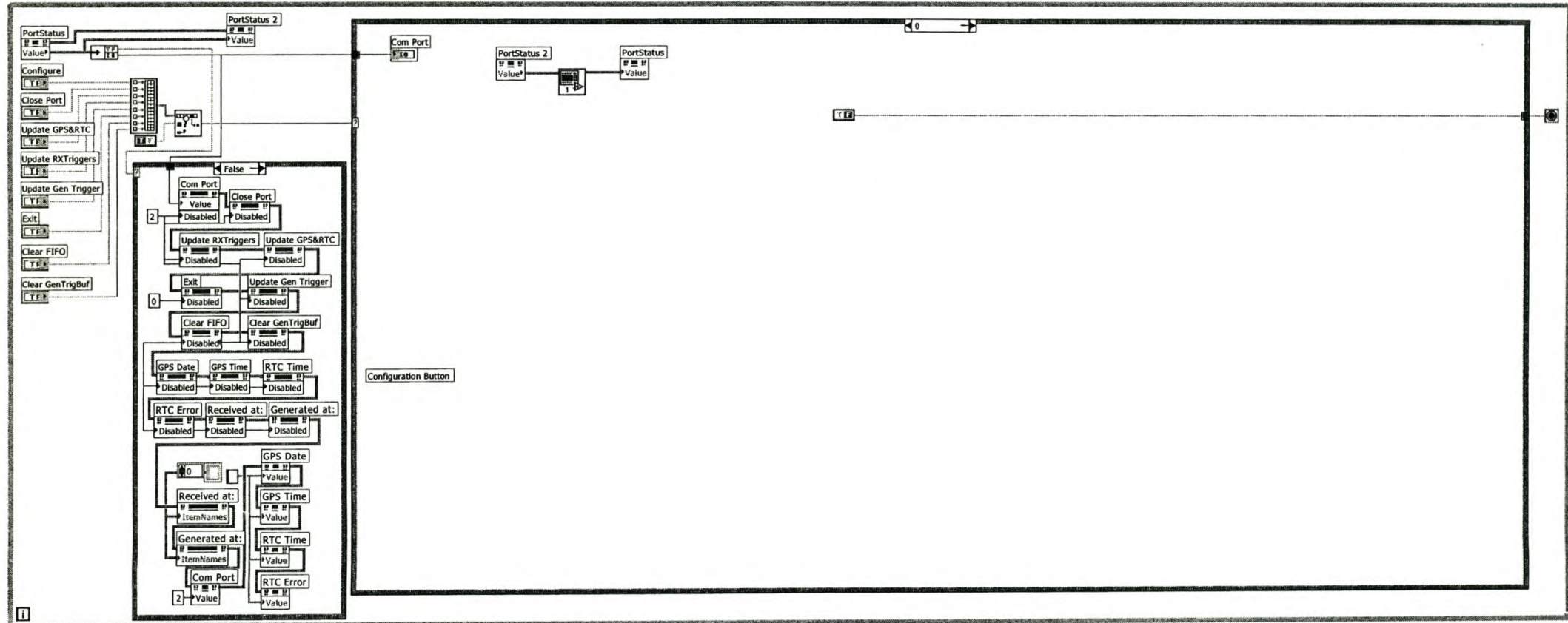
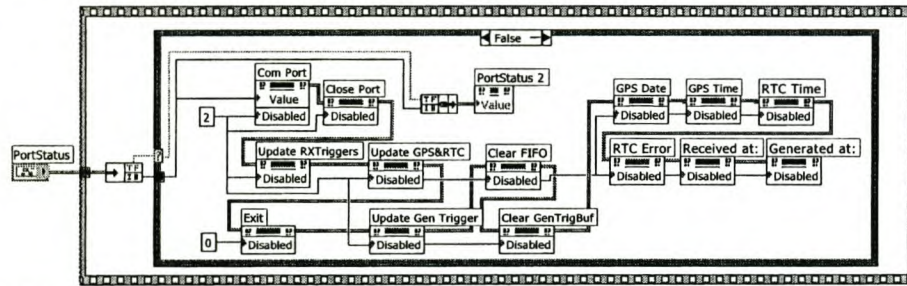
 **Index**

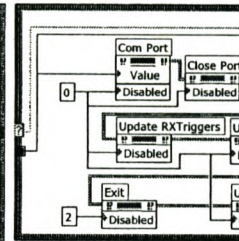
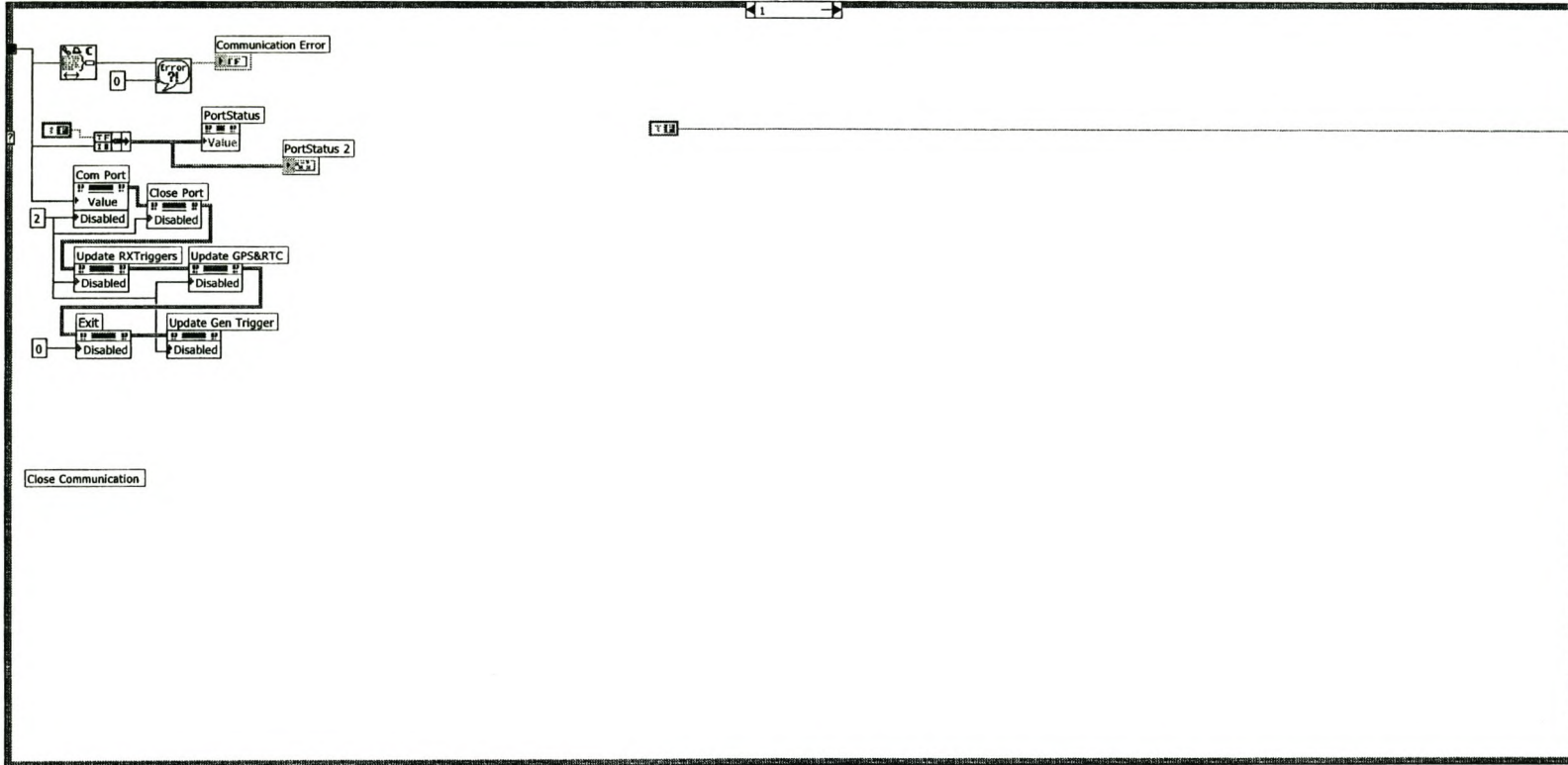
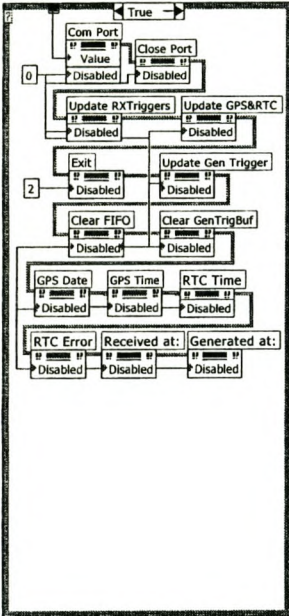
- U8 Day
- U8 Month
- U8 Hours
- U8 Minutes
- U8 Seconds
- U32 Microsecs

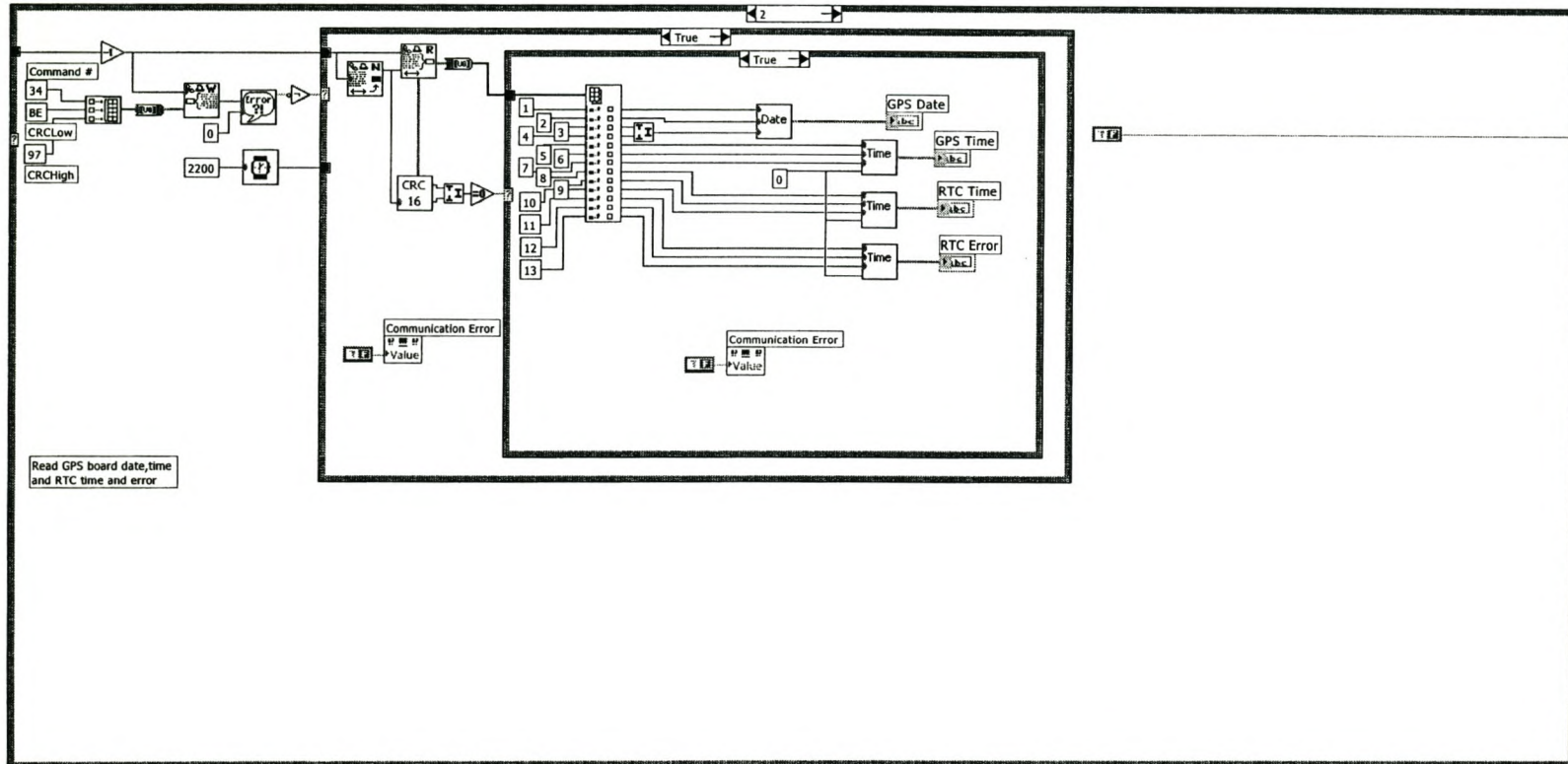
2006 **Generated Triggers**

- 2006 **Cluster**
 - U8 Index
 - U8 Day
 - U8 Month
 - U8 Hours
 - U8 Minutes
 - U8 Seconds
 - U32 Microsecs

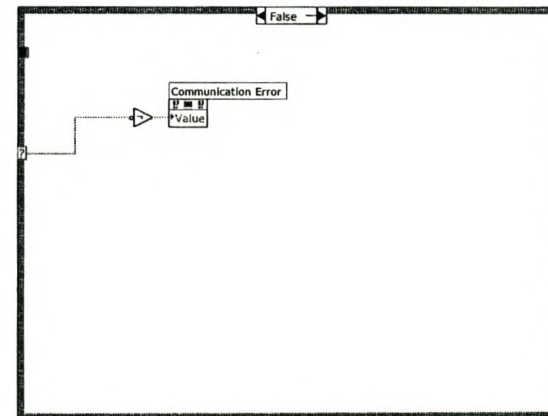
Block Diagram

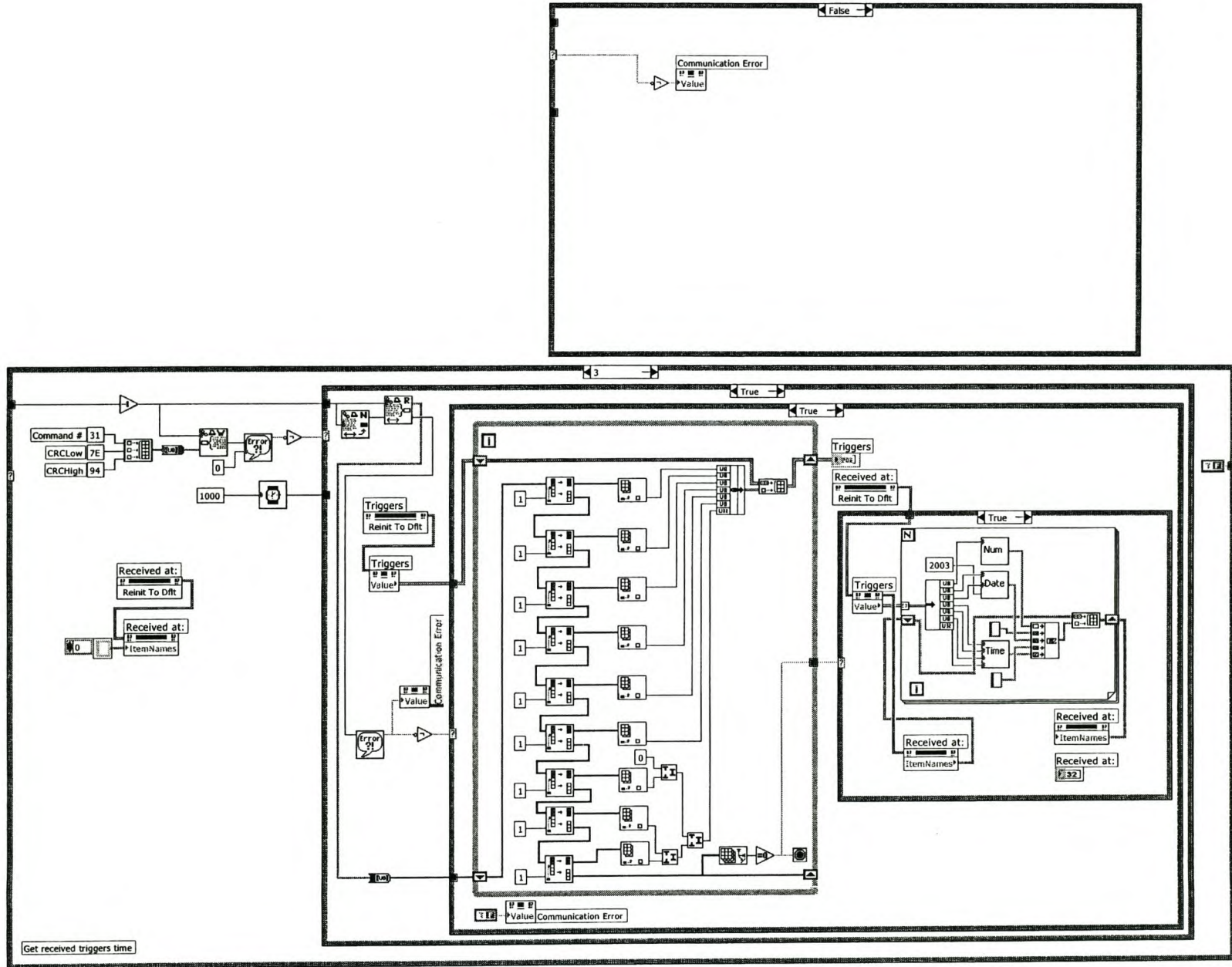


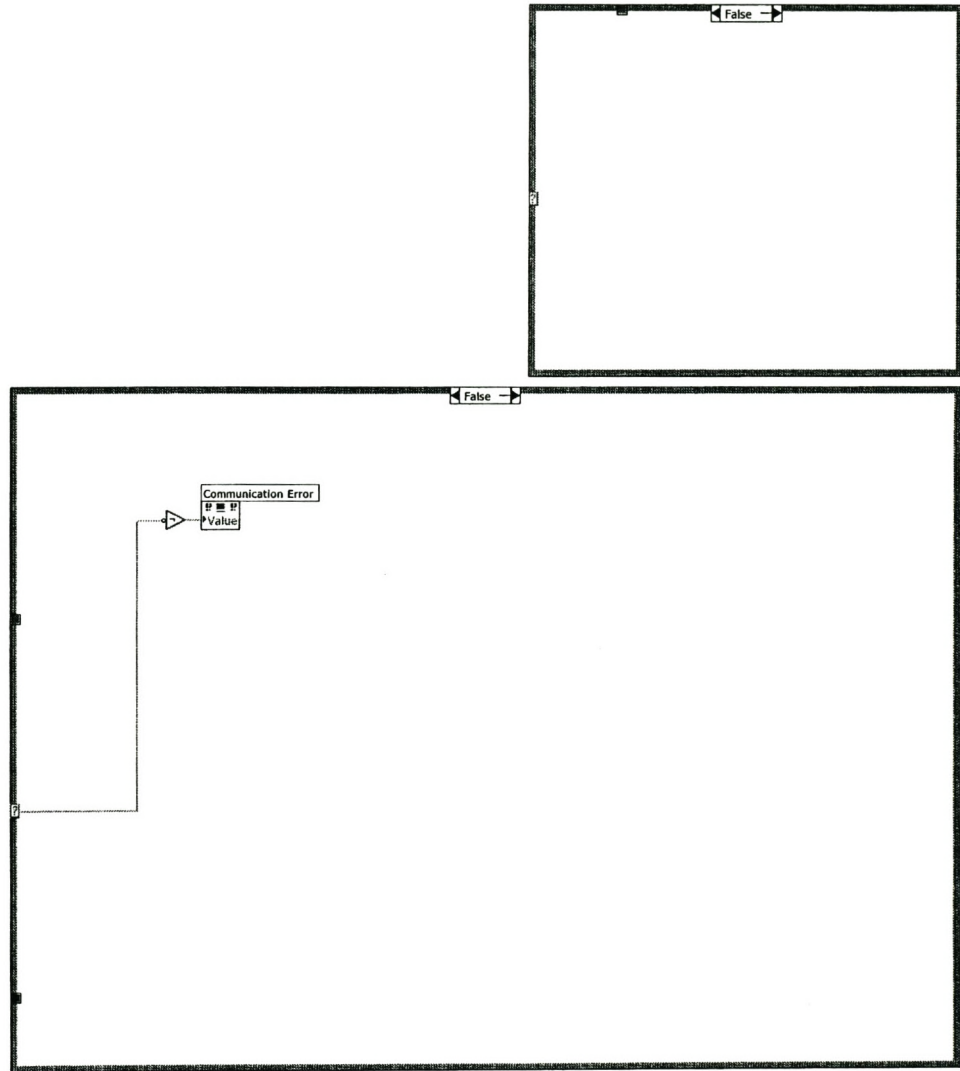


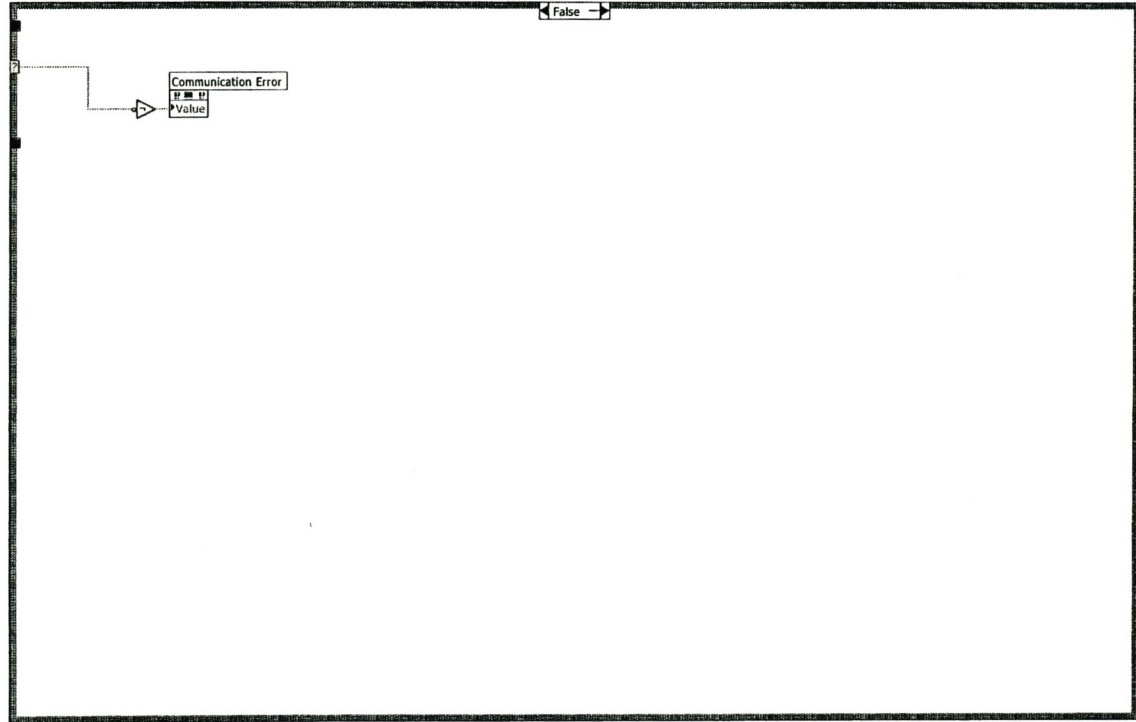


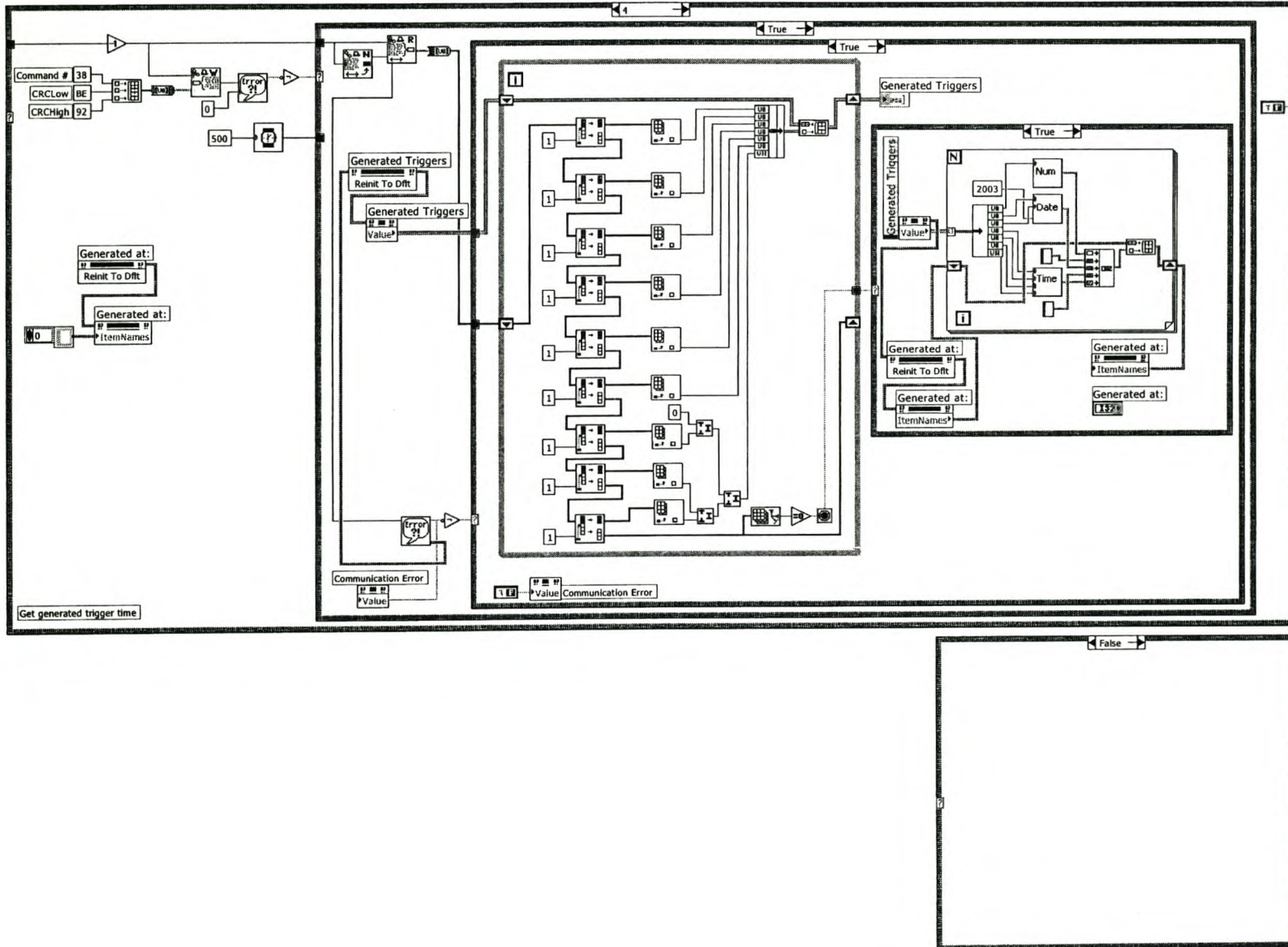
Read GPS board date,time and RTC time and error

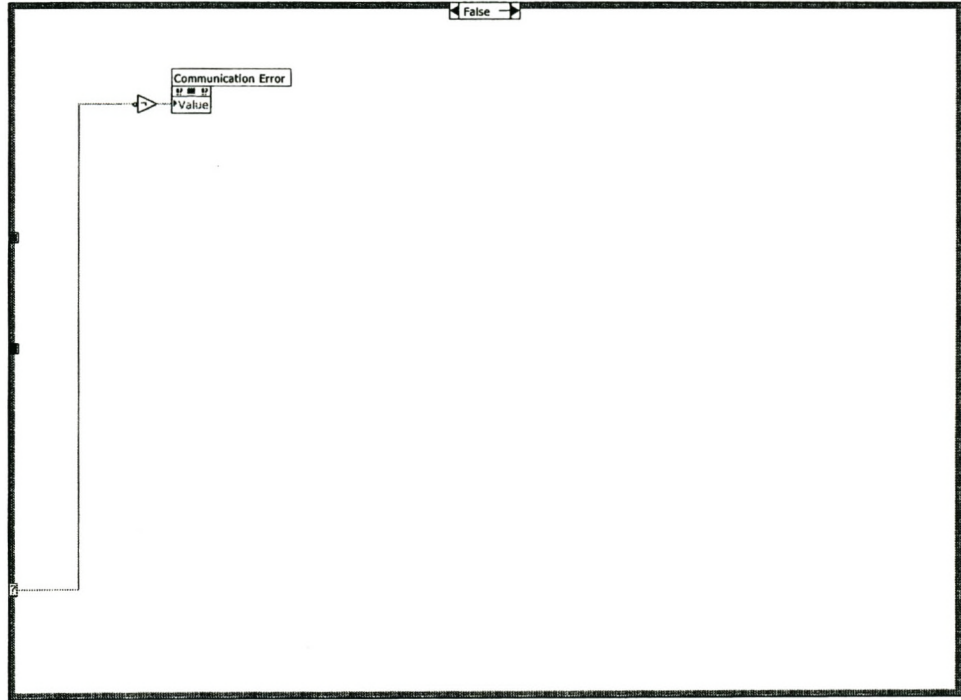


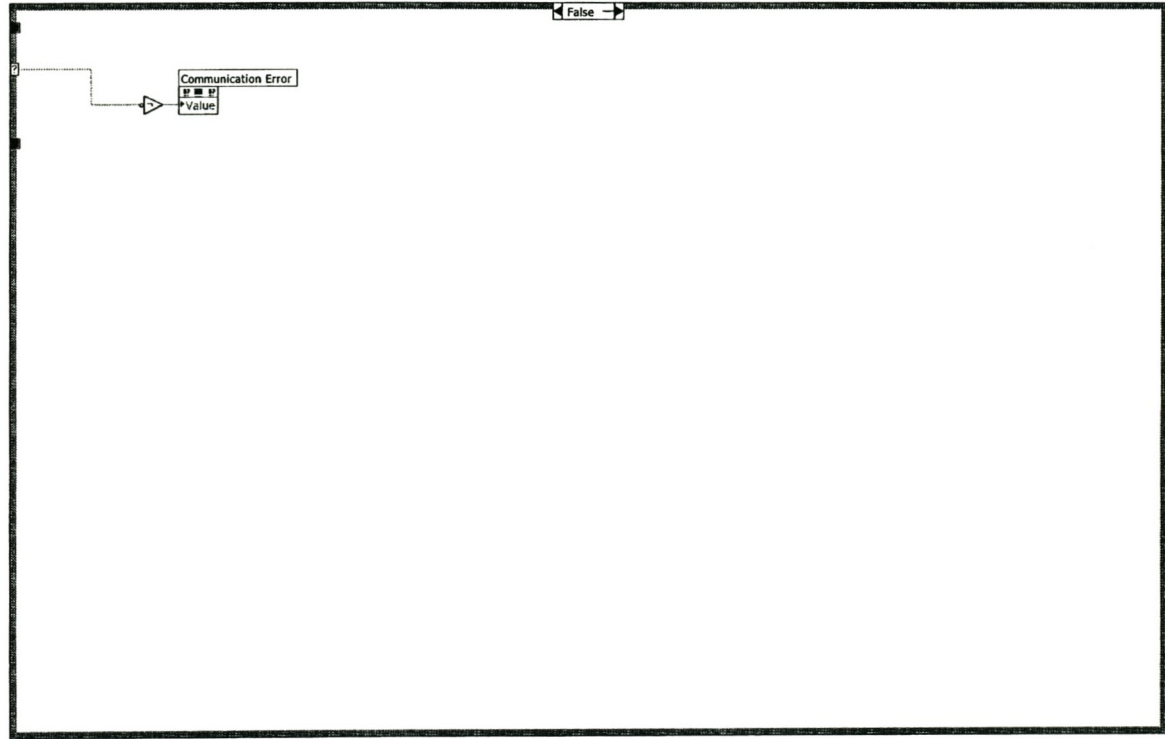


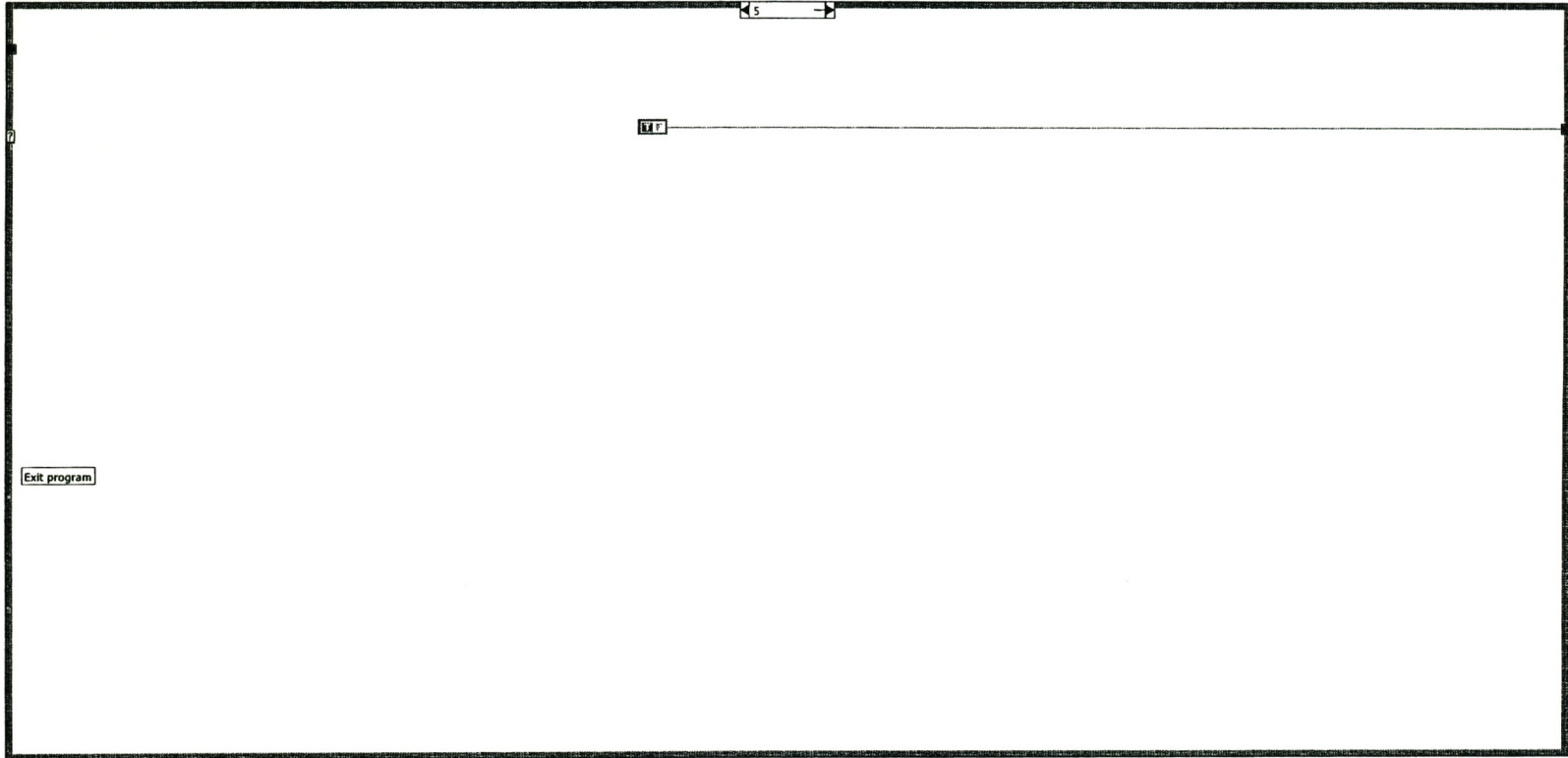


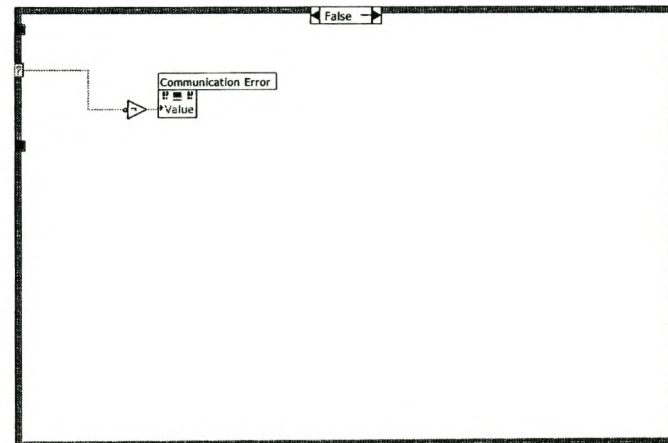
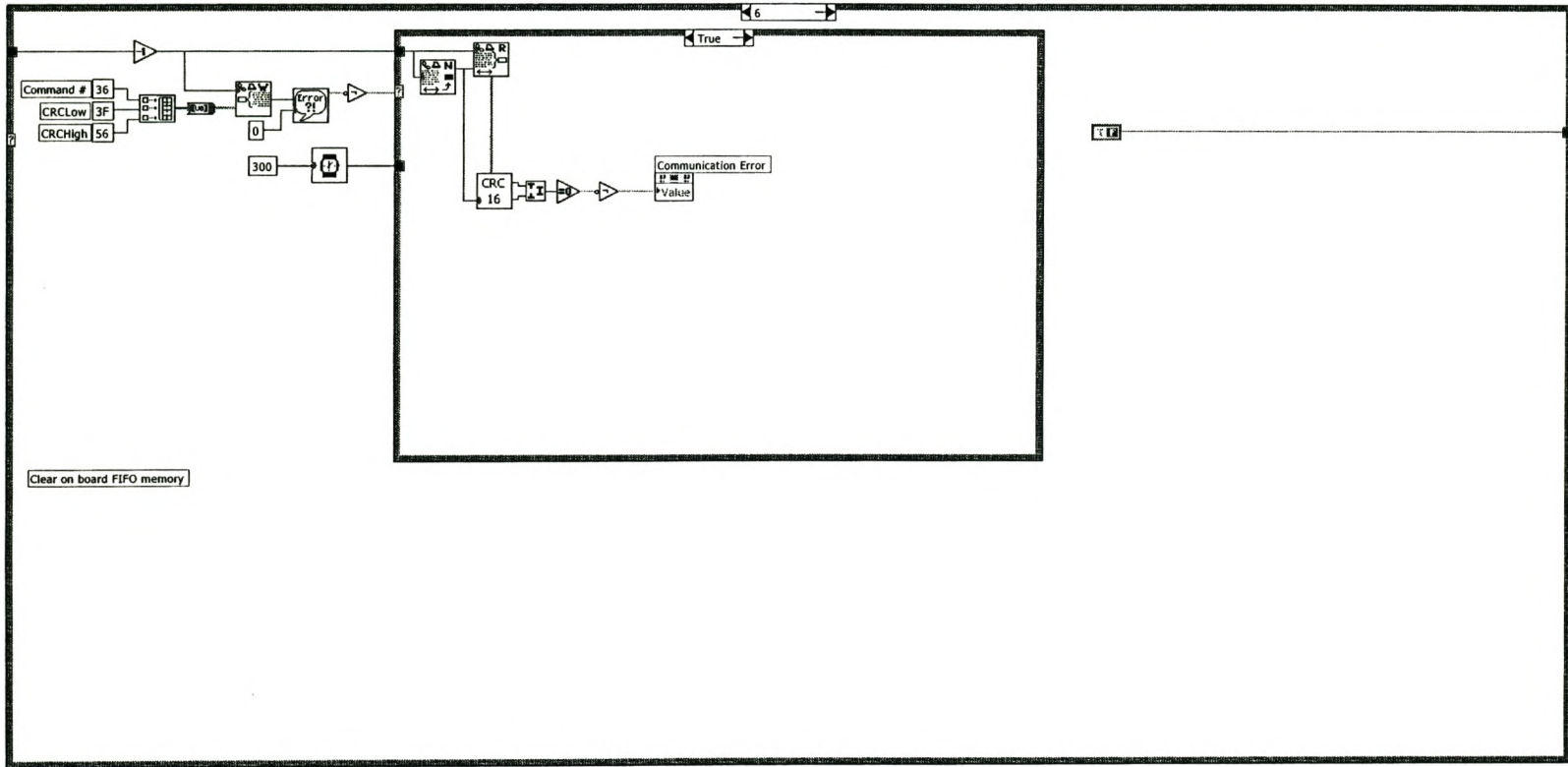


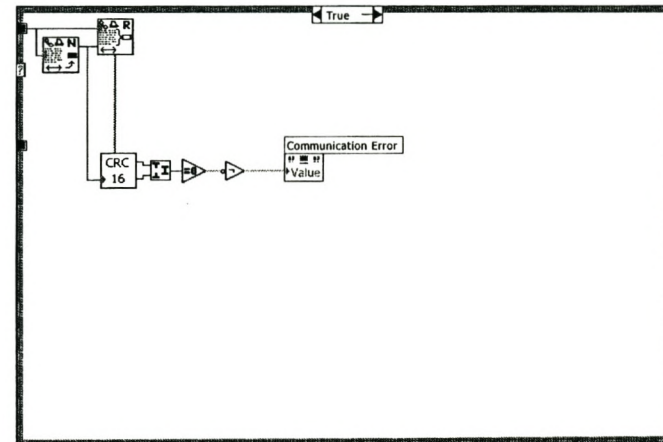
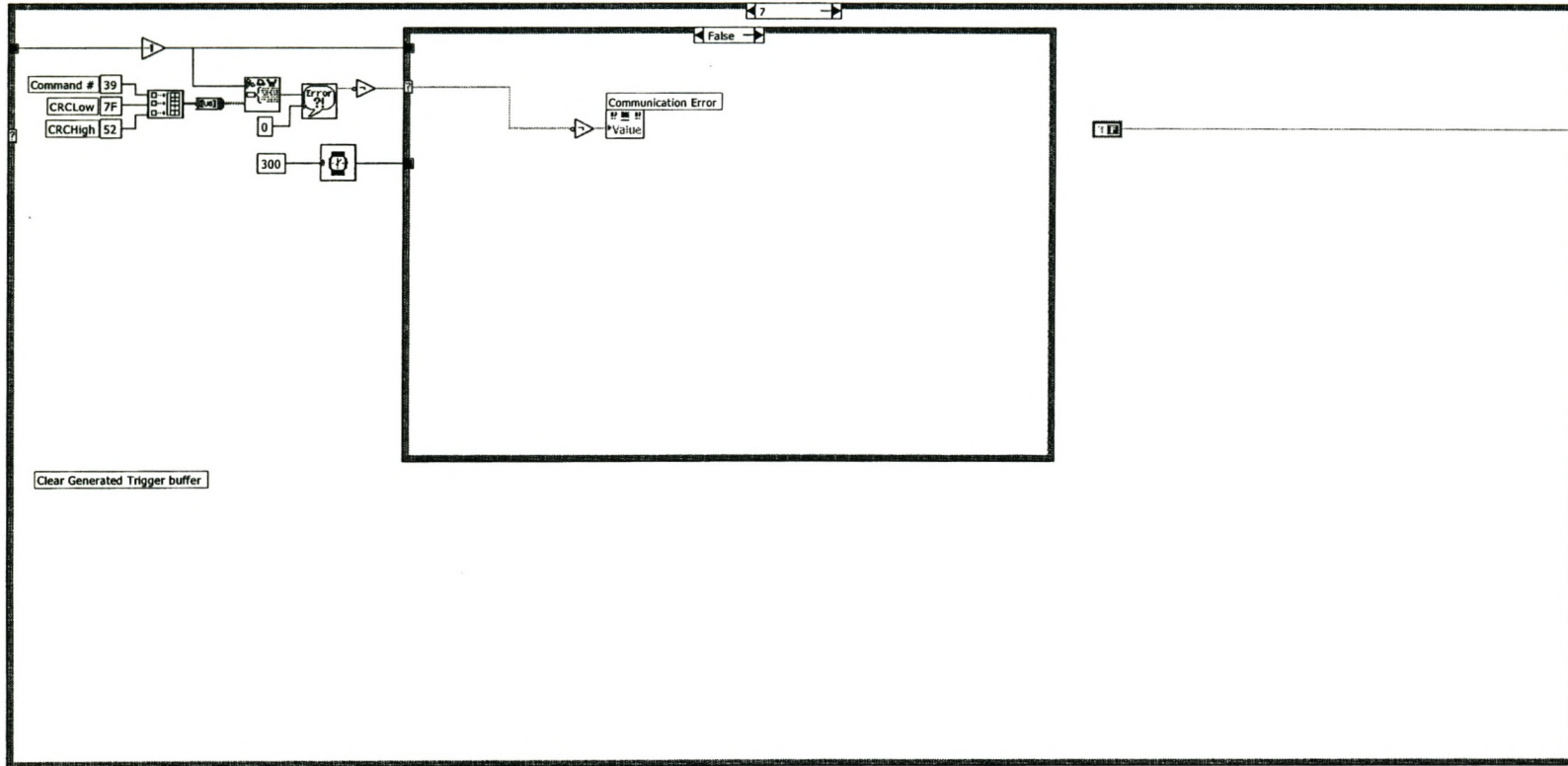


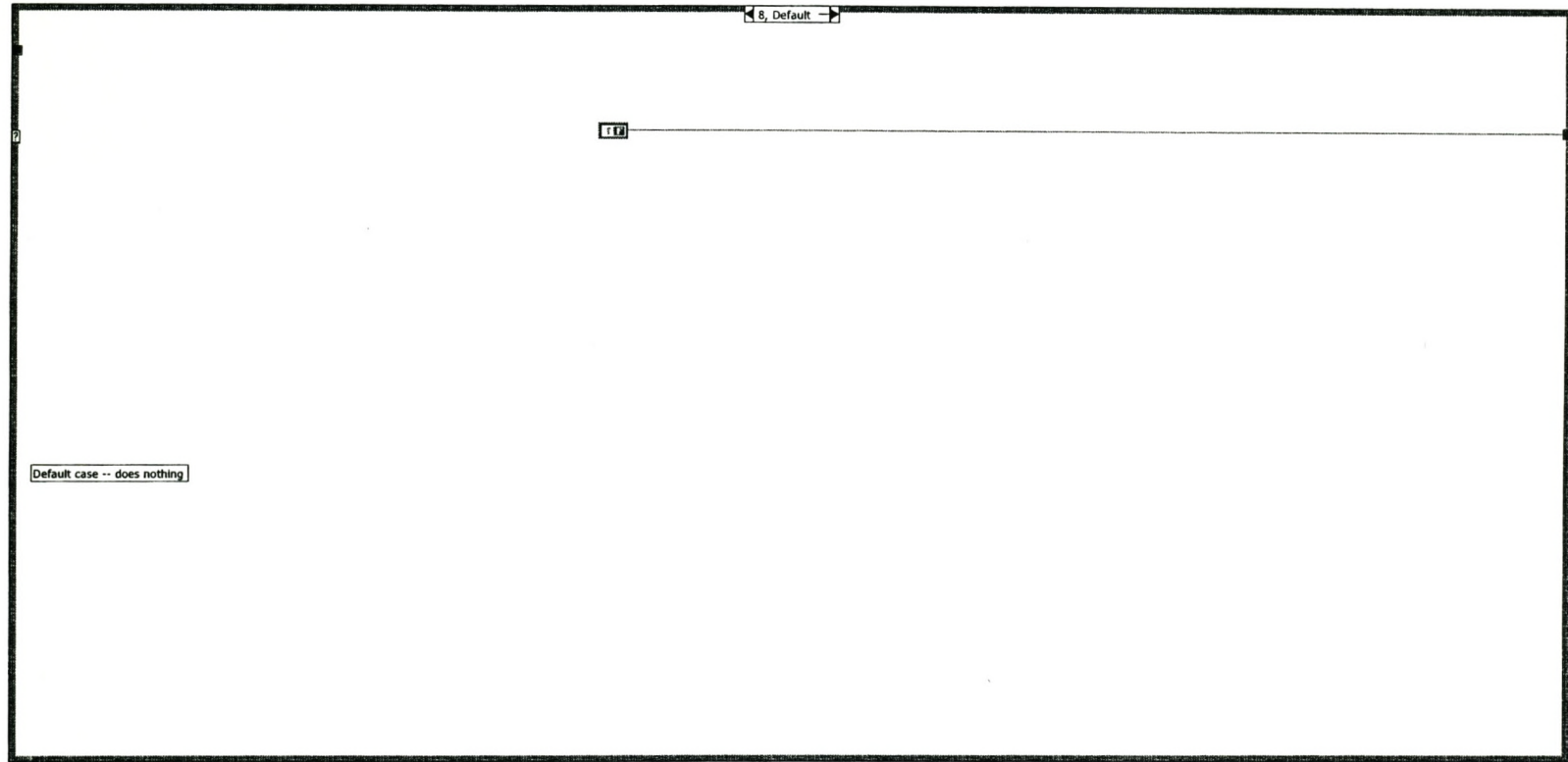




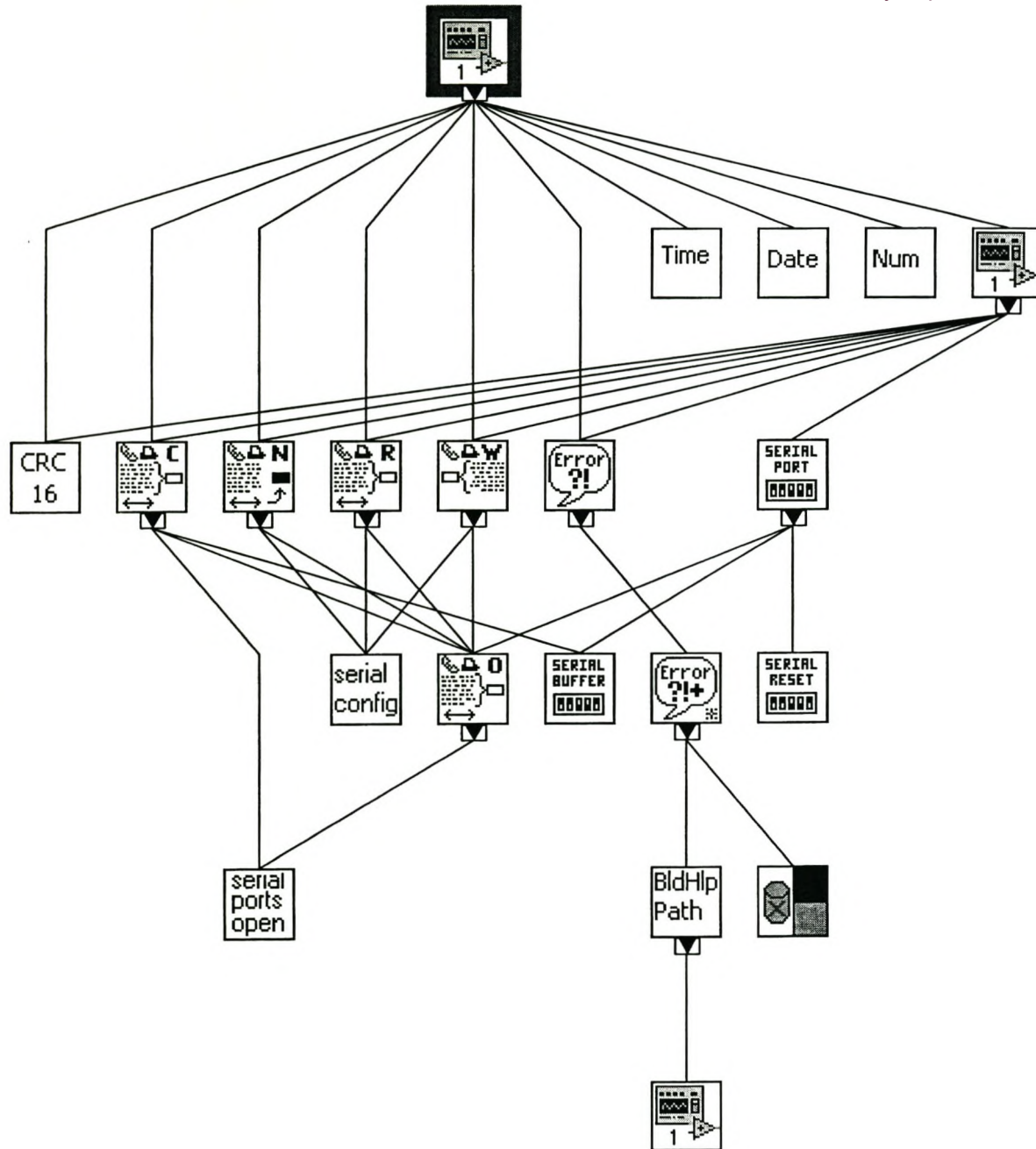








Position in Hierarchy



List of SubVIs

**Close Serial Driver.vi**

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Instr\Serial.llb\Close Serial Driver.vi

**Simple Error Handler.vi**

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Utility\error.llb\Simple Error Handler.vi

**ConfigureDB.vi**

C:\Documents and Settings\Thinus\My Documents\Work\Development\LabView\GPS Control\ConfigureDB.vi

**Serial Port Read.vi**

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Instr\Serial.llb\Serial Port Read.vi

**Bytes At Serial Port.vi**

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Instr\Serial.llb\Bytes At Serial Port.vi

**Serial Port Write.vi**

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Instr\Serial.llb\Serial Port Write.vi

**CRC16.vi**

C:\Documents and Settings\Thinus\My Documents\Work\Development\LabView\CRC VI\CRC16.vi

**Indexing.vi**

C:\Documents and Settings\Thinus\My Documents\Work\Development\LabView\GPS Control\Indexing.vi

**ConvertToTime.vi**

C:\Documents and Settings\Thinus\My Documents\Work\Development\LabView\GPS Control\ConvertToTime.vi

**Date.vi**

C:\Documents and Settings\Thinus\My Documents\Work\Development\LabView\GPS Control\Date.vi

History

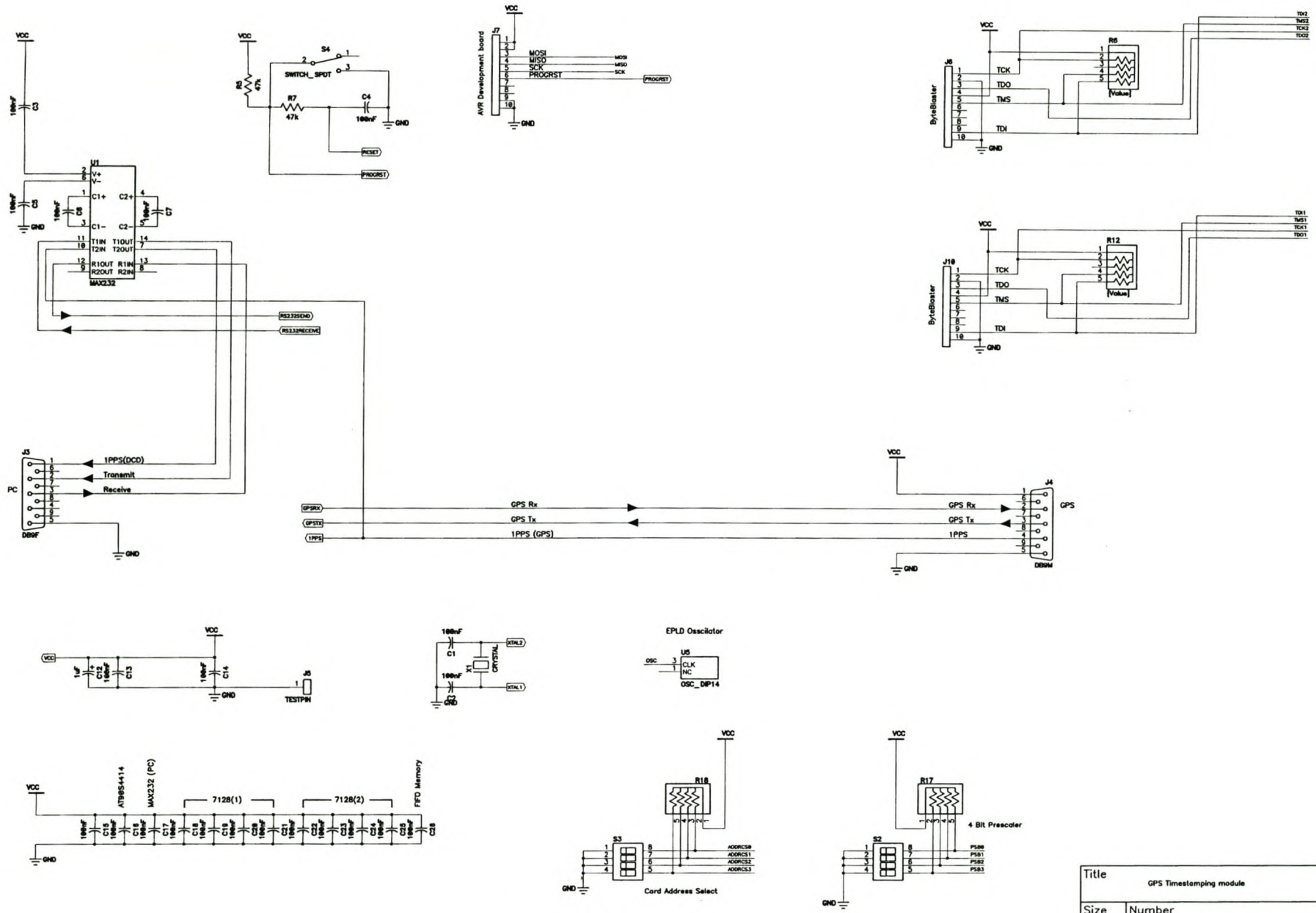
"GPSControl.vi History"

Current Revision: 355

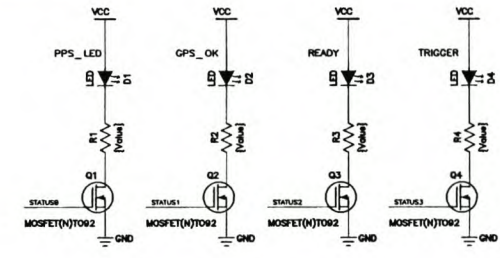
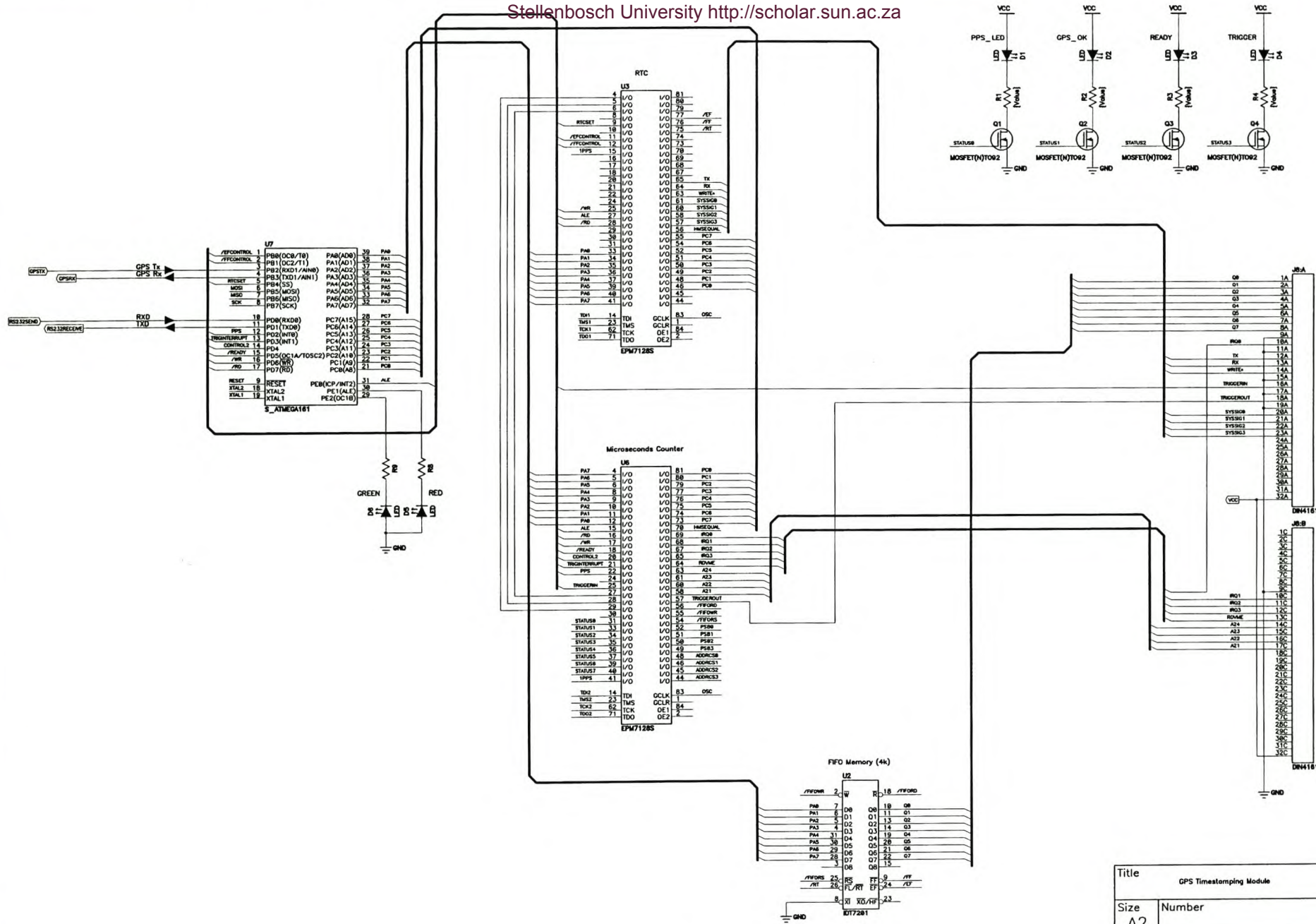
Appendix I

Schematic Diagrams

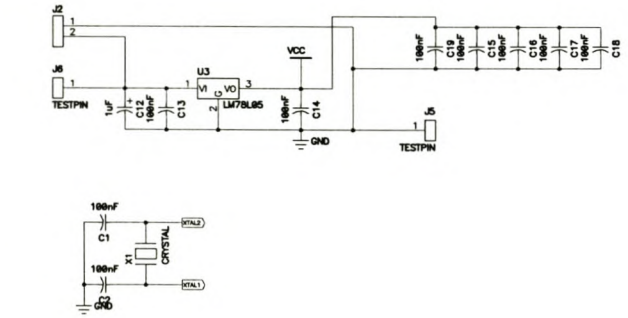
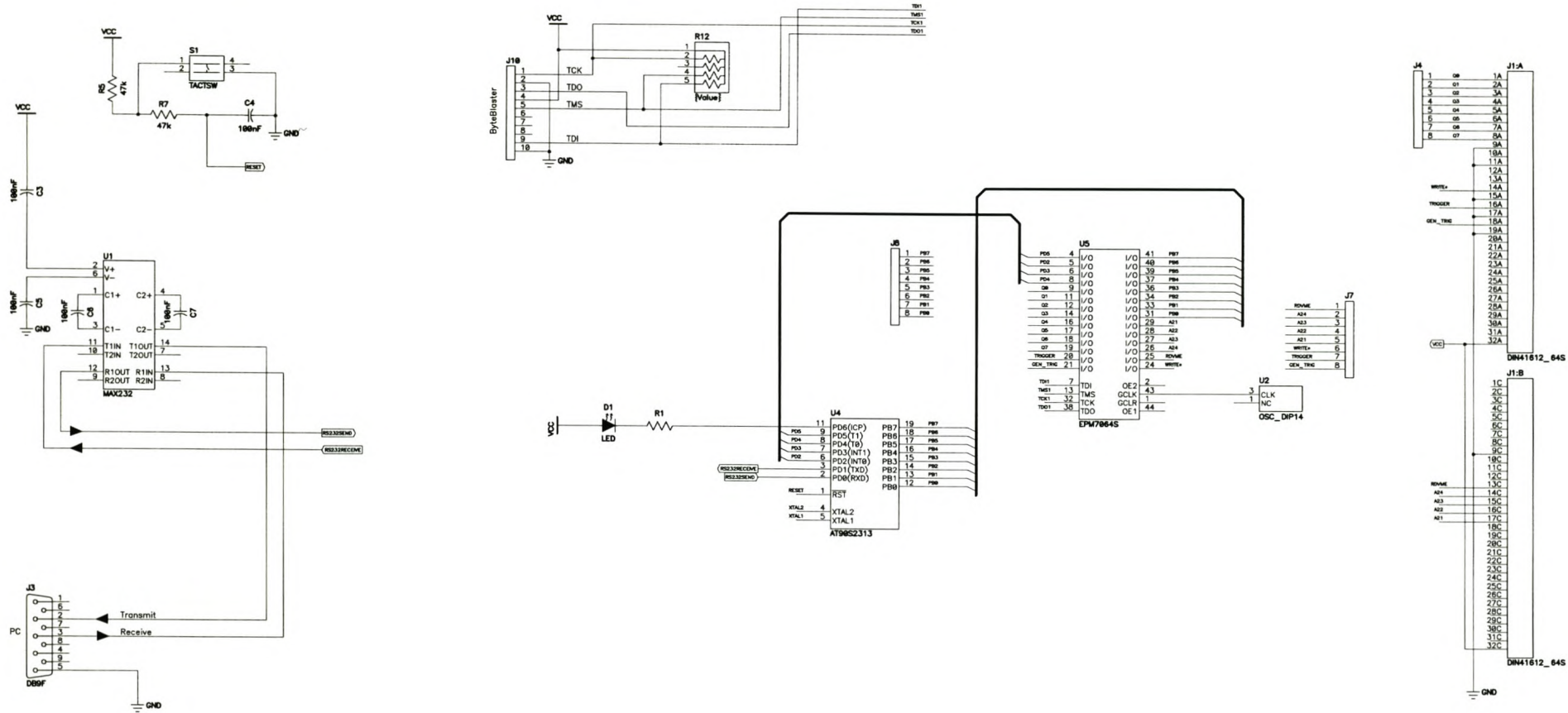
This Appendix presents complete circuit diagrams of the developed system and the system test backplane.



Title GPS Timestamping module		
Size A2	Number	Rev 2.0
Date 26/02/2002	Drawn by Thinus van As	
Filename GPS Board.sch	Sheet 1	of 2



Title GPS Timestamping Module		
Date 18/03/2003	Number Rev 2.0	Drawn by Thinus von As
Filename GPSTimeStampATmega.sch		Sheet 2 of 2



Title		
Size A2	Number	Rev
Date	Drawn by	
Filename	Sheet of	