UNIVERSITY OF CALIFORNIA

Los Angeles

Software for A Fault Tolerant Microcontroller Network

A thesis submitted in partial satisfaction

of the requirements for the degree Master of Science

in Computer Science

by

David Ju

1999

The thesis of David Ju is approved

_____

Algirdas Avizienis

_____

Milos D. Ercegovac

_____

David A. Rennels, Committee Chair

University of California, Los Angeles

1999

TABLE OF CONTENTS

LIST OF FIGURES

List of Acronyms

BIT	Built In Test
BPS	Bit Per Second
CCBs	Chip Configuration Bytes
CCRs	Chip Configuration Register
EEPROM	Electronically Erasable Program Only Memory
FIFO	First In First Out
IO	Input Output
LED	Light Emitting Diode
msec	Milli-second
PROM	Program Only Memory
RAM	Random Access Memory
RTI	Real Time Interrupt
Rx	Receive
SSIO	Synchronous Serial IO
Tx	Transmit
PAL	Programmable Array Logic
UART	Universal Asynchronous Receiver Transmitter
WR	Write

ABSTRACT OF THE THESIS

Software for A Fault Tolerant

Microcontroller Network

By

David Ju

Master of Science in Computer Science

University of California, Los Angeles, 1999

Professor David A Rennels, Chair

Low-cost commodity microcontrollers have very few fault-tolerant features in themselves. Thus it becomes an interesting challenge to implement fault-tolerance by using redundant configurations of these chips to enable their use in applications requiring high dependability. This thesis presents a software design approach used in implementing such a fault tolerant microcontroller network and an implementation of its operating system and support software. The basic processing nodes of this network consist of three or four microcontrollers that run redundant computations and use software comparison of output messages to detect and recover from faults and errors. Prototype boards have been built, and a breadboard of a node has been implemented to write, test, and verify the software written for the microcontroller network and to prove the concept of a voting and

recovery algorithm. The prototype uses the Intel 80196CA microcontroller, and we take advantage of many of its features such as serial IO, synchronous serial IO, digital IO ports, and external interrupts.

The design approach shares responsibility for error detection and recovery between a real-time synchronous executive and a set of comparison-voting procedures. The executive supports foreground and background application processes, and it also schedules recovery procedures needed to recover from error and fault conditions. Voting and recovery algorithms written for this test bed have been successfully run on the prototype boards, and their algorithms have also been partially verified. A support and test environment has also been implemented on an external PC that allows us to upload programs, generate inputs, capture outputs, and step through programs for debugging. This facility will be used for further development of the system.

1 Introduction and Fault-Tolerance Approach

1.1 The Node Architecture

A fault-tolerant node consists of two or more redundant microcontroller modules. Each module can be assigned one of three operating states: (1) Master, (2) Slave, and (3) Off-Line. The Master and Slave modules, designated “active” modules, execute identical application programs scheduled by a time-driven operating system on the basis of a common real-time interrupt (RTI). The applications programs execute at approximately the same time -- bounded by the frequency skew of their clocks within an RTI interval and slight differences in operating system functions in Master and Slaves. The Master is the primary processor that generates outputs, and the Slaves provide comparisons for error detection and participate in error and fault recovery operations. Off-line modules do not participate in the processing, but wait for commands from the Master module (or from an RS232 debug port in the testbed) to modify memory, perform tests, and reconfigure the module to go back on line.

Although applications programs are the same, the operating systems in Master or Slave modules carry out different functions for comparison, voting, and error recovery. The general strategy (abstracted from RENN-97) is:

Inputs -- All active modules independently sample all inputs. In order to achieve the same value for the input in all the modules the Master sends the value it received to the Slaves (via a special internal bus called the Master Channel). The Slaves determine if the value is within tolerance limits of what they sampled and respond with an *agree* or *disagree* symbol over a special Status Channel that all modules can see. If at least one Slave agrees, all the modules use the Master's value. Since IO events have bounded latencies, modules that do not respond can be detected by time-outs. Both disagreements and time-outs trigger recovery actions.

Outputs -- When an output is to be produced, the Master generates an output value and sends it to the Slaves. The Slaves do not output, but their operating systems do an exact comparison of this value with the outputs they would have generated, then responding with *agree* or *disagree* status symbols via the special status bus. At least one Slave must agree with the Master to enable an output. As in the case of inputs described above, time-outs and disagreements trigger recovery actions.

Figure 1 shows a block diagram of a fault-tolerant node including four microcontroller modules -- a Master, two Slaves and an Off-Line module. (Note that the operating states of the modules can be dynamically changed, so different modules may be Master, Slave, and Off-Line at different times due to fault-induced reconfigurations.)

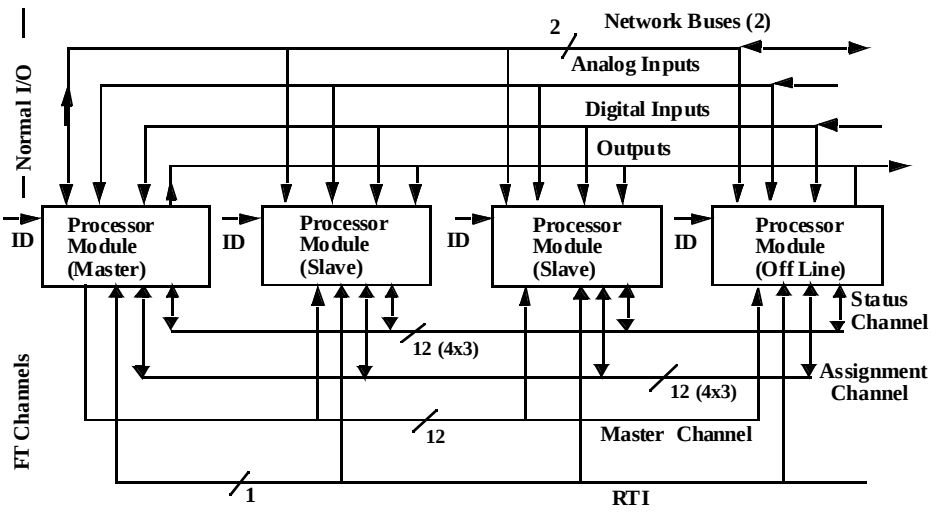

Figure 1 A Node Consisting of Four Microcontrollers and their Interconnects

The interconnects at the top of the diagram, designated normal IO, are the digital and analog inputs and digital outputs to external sensors and actuators. The inputs are bussed to all modules. The output connections are bi-directional (programmable from within the microcontroller modules). Only the Master module is configured to drive the output lines, and Slaves are configured to receive the output lines as inputs in order to do comparisons. If the Master module fails, another module can be assigned as Master and can be reconfigured to drive the outputs. A separate (redundant) hardware enable circuit is used in each module that independently prevents a faulty module from generating outputs. At least two modules must "vote" for a module to be Master (via the Assignment Channel described below) before this hardware allows it to generate outputs.

Fault-Tolerance Channels

The interconnects at the bottom of the diagram, labeled FT Channels are used for implementing fault-tolerance features. They consist of the Master Channel, Status Channel, and Assignment Channel.

The Master Channel -- The Master Channel is a bus used by the Master: i) to send data to all the Slaves for comparison and voting, ii) to command program rollbacks, and iii) to send commands to specific modules to command them on-line or off-line or to load and access data in their memories.

The Status Channel -- Since normal IO is constrained so that only the Master can generate outputs and messages to the other modules, it is necessary to provide a mechanism by which (1) the modules can synchronize their actions, and (2) the Slaves can communicate whether they agree or disagree with the actions of the Master. The Master, upon generating an output to the Master Channel, sends a status symbol requesting a comparison. The slaves respond with symbols indicating whether they agree or disagree. The collection of wires conveying the status messages from all the modules are designated the Status Channel as shown in Figure 2. Each module has three wires by which it sends a coded status symbol to the other modules (e.g., 000 null, 011 agree, 110 disagree, 101 compare).

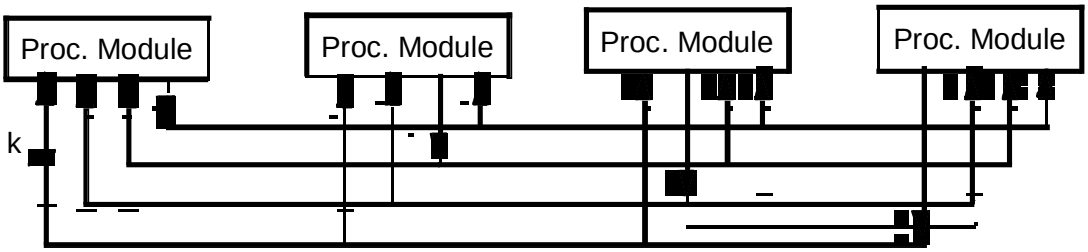

Figure 2 The Status Channel

The Assignment Channel. -- The Assignment Channel consists of four three-wire buses -- one from each module to the other three microcontroller modules (connected similarly to the Status Channel in Figure 2). It is used for each module to indicate its operating mode to the others and for slave modules to collectively select a Master by voting -- and replace one that is faulty. Each module always sends one of the following types of messages as 3-bit symbols indicating its status to the others over its dedicated lines: I) unpowered/testing, II) off-line, III) on-line requesting a restart, or IV) on-line voting for one of the four modules to be master.

Each module votes the messages on the Assignment Channel to determine whether it is the Master module (two or more votes - from itself and at least one Slave module.) This is done redundantly. The microcontroller software performs the vote, only enabling its output drivers and operating as Master if it “wins”. In addition each module has redundant hardware voting logic that must independently

agree on the vote before outputs are enabled. This is crucial to prevent a faulty master from taking over the system.

A time line of a voting process is shown below. The Master outputs a value to be compared, followed by raising a comparison request status ‘c’. The slaves, seeing the compare respond with an *agree* or *disagree* ‘a’. A null status ‘-‘ is used for handshaking to signal the start and end of the comparison process. (The x's represent the setting of an internal timeout counter and the symbol ^ indicates its being cleared upon completion). A more detailed description of this can be found in RENN 97.

```
Slave   ------------x--------aaaa^----
Slave   ------------------x---aaaa^----
Master  ----------------------xccc^-----
```

The general strategy for voting, assuming single faults, is as follows:

(1) If the Master and at least one slave agrees, the computation continues.

- If a master and one or more slaves agree, but a slave disagrees, then the disagreeing slave is commanded off-line by the master with concurrence from the agreeing spare. A recovery process to reinitialize and retry faulty modules (or to activate a new spare) is scheduled as a regular foreground process and carried out as soon as possible by the agreeing Master and Spare.

(2) A slave detects that it disagrees (when it can) by either a *timeout* when a message from the Master is expected or a *disagree* with the value generated by the Master. If a Slave detects that it disagrees, it signals *disagree* on the status channel and waits for subsequent events to be described below.

- If the slave is faulty, it is unclear what it will do, but it will be replaced by the Master and agreeing slave -- as in (1) above.
- If the Master is faulty, both slaves will signal *disagree*. When a slave disagrees, it waits for the other slave to signal disagreement also. If two slaves detect that they both disagree, they change their vote for Master to the lowest numbered Spare, thus replacing the Master with one of the Slaves and repeat the failed IO. The new Master commands the old Master off-line. It and the agreeing slave then schedule diagnostic checks, and if they pass, they reload/restart the discarded Master as a new Slave - as in (1) above.

(3) If the Master does not see at least one slave agree, it is probably at fault and, if it can, it commands itself off-line. (The slaves should do this also.)

The general strategy can also be described by saying that any module that sees that it is no longer a member of an agreeing set on outputs commands itself off-line, sets its assignment channel symbol to 000 (off), and runs a hardware diagnostic program. If that program passes, it looks to see if the system is still up -- i.e., there is still a Master and at least one agreeing Spare by examining the Assignment Channel. If the system is still up, it awaits commands from the Master to reinitialize it and bring it back on line. If it finds the system to be down (no Master

and Spare voting for it), it advertises by placing a Startup symbol on the Assignment Channel. If two modules reach the Startup State, they can then assign one to be Master, one to be a Slave, and then restart the system.

1.2 Software Structure for a Node

The software structure for a Node is shown in Figure 3.

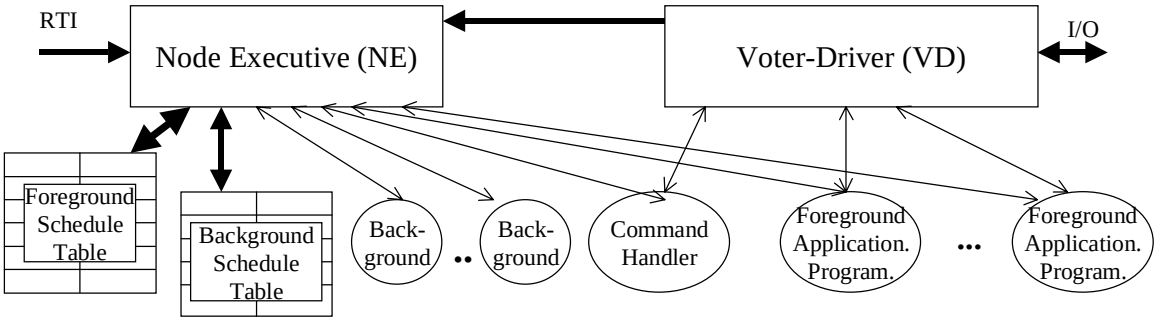

Figure 3 The Software Structure of a Node of Fault-Tolerant Microcontrollers

The Node Executive is driven by a real-time interrupt (RTI) at a rate of approximately 50 Hz. Foreground applications programs run in short segments within an RTI interval and are responsible for collecting inputs, generating outputs and starting other programs (or restarting itself) at precise points in time. A command handler foreground program is run at every RTI which checks incoming command buffers, and it schedules other foreground programs on the basis of commands received from a serial port or an external CAN bus. Then ongoing

foreground programs that are scheduled for the current interval are re-activated. Upon completing all foreground programs for a given time interval, background programs are restored. Background programs, started by foreground programs, are run across RTI intervals and do more complex processing functions.

Input-output is initiated by foreground programs that call special Voter-Driver programs to carry out IO and perform voting. If a module is a Master, outputs are generated, but if the module is a Slave, its outputs are simply compared with what the Master generated, and a voting status is generated. Several results are possible:

- If the Master and all slaves agree, the IO is successful, there are no errors, and control is returned to the foreground program.
- If a Master and a slave module agree, but another slave disagrees, the voter-driver programs in the Master and agreeing Slave command the miscreant off-line and schedule a recovery process as a normal foreground process.
- If the Voter-Driver of a Slave cannot find agreement (times out or does not agree with the Master), it waits a short time to see if the other Slave disagrees also. If that is the case, the slaves command the Master off-line and activate the lowest-numbered slave as a new Master, repeat the IO operation and then effect recovery as in the item above.
- If the Voter-Driver of a Slave cannot find agreement, but the other slave does not signal disagreement, or if the Voter-Driver of a Master cannot find agreement, the scheduler is cleared and the module declares itself off-line/testing and initiates a self-diagnosis.

Whenever a module is externally commanded off-line or voluntarily goes off-line, it sets its assignment channel state to unpowered/testing and initiates a diagnostic. If the diagnostic is unsuccessful, its state remains unpowered/testing. If successful, it returns its state to off-line. A foreground process is scheduled that periodically checks if the system is still operational as indicated by a Master and a Slave voting for it. If the system is operational, the module waits for commands from the Master to initialize and restart it. If it believes the system is not operational, it signals via the assignment channel (status = restart) that it wants to participate in a restart process. When two members send similar signals, they can establish a new master and restart the system.

1.3 The Testbed Configuration

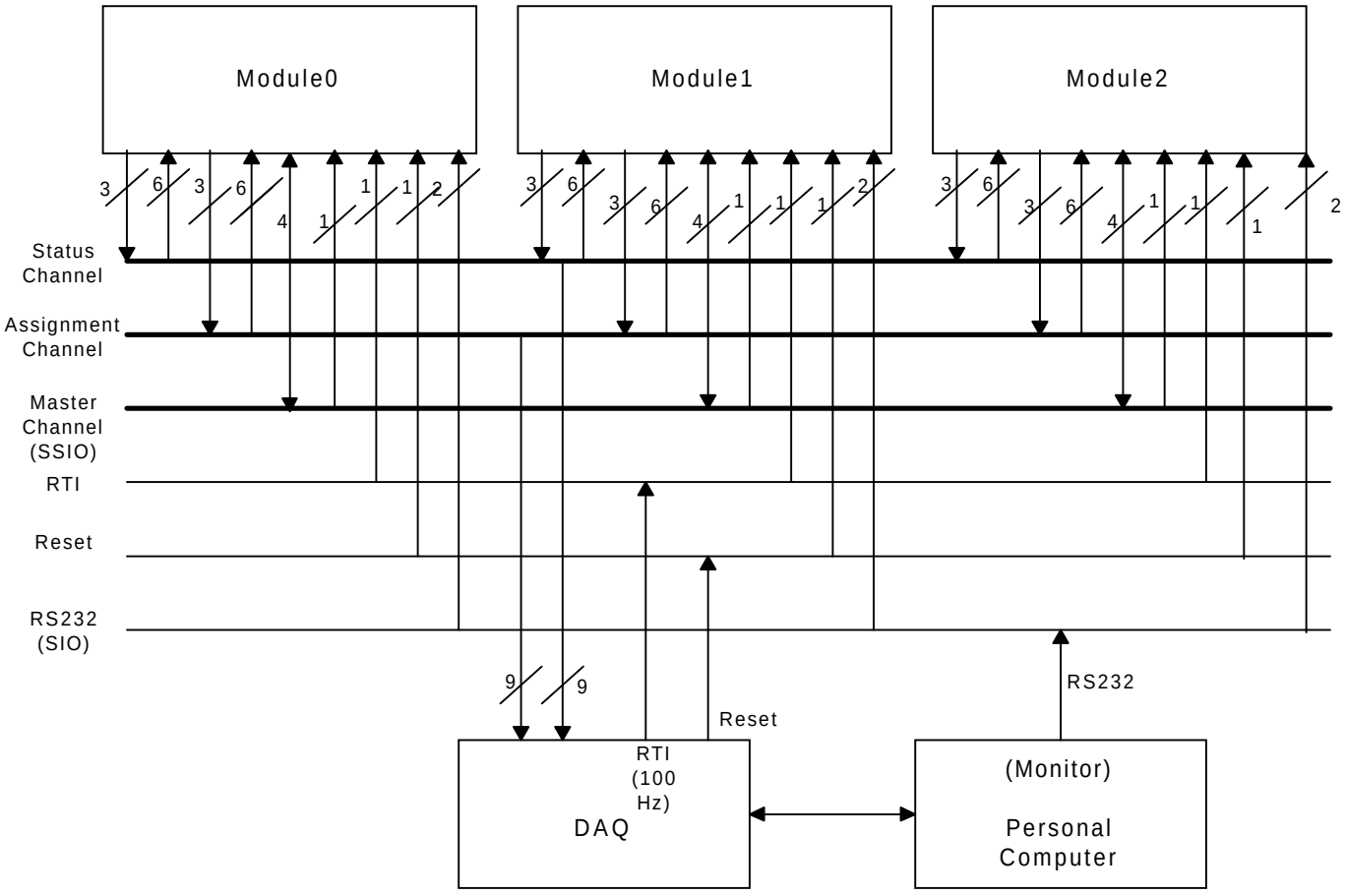

Figure 4 The Microcontroller Node Prototype Testbed

The following are the descriptions of the hardware configuration used in the preliminary testbed. The differences in the hardware configuration between the preliminary testbed and the final testbed shall be discussed in a later section.

The Testbed consists of three microcontroller modules that interface to a Monitor Computer using two Data Acquisition (DAQ) Cards (A 96-Bit Parallel Digital I/O Interface for PCI Bus Computers, PCI-DIO-96 and Multifunctional I/O Board for PCI Bus Computers, PCI-1200) and an RS232 port in the PC that can be connected to the microcontrollers via a switch box. The Monitor Computer can sample the value of the Status and Assignment channels via the DAQ.

The Monitor controls the microcontrollers by generating reset signals to each and by generating the common RTI. Since each processing step begins with the RTI in each microcontroller, it is possible to load and configure them individually from the Monitor using the RS232 serial ports and then single step them (by issuing single RTIs and observing their responses) for low-level debugging.

The interconnects between the microcontrollers and the Monitor Computer are summarized below:

Reset – Reset is a digital output that comes out of the DAQ board which is installed in a PC. This reset output is connected to all three modules. This signal serves as a POR to the microcontrollers. When a reset signal is asserted, all three modules restart.

RS232 – RS232 consists of Tx and Rx lines. Each module has Tx and Rx lines, and they are connected to the PC communication port. Since one port is used to communicate with modules, RS232 multiplexing box is connected between the modules and the PC communication port. RS232 multiplexing box has a switch, and the switch lets the user select a module to establish communication between the modules and PC.

Status Channel Monitor by DAQ – Three status channel output wires of each module are connected to the DAQ. A total of nine wires are connected to DAQ as digital inputs for monitoring purpose.

Assignment Channel Monitor by DAQ – Three assignment channel output wires of each module are connected to the DAQ. Total of nine wires are connected to DAQ as digital inputs for monitoring purpose.

RTI – RTI is a 100 Hz digital output from the DAQ. This signal has 50% duty-cycle, which means it is 5 volts for 5 msec and 0 volt for 5 msec. The RTI signal is connected to the external interrupt line of each module. It is used as a source of real-time clock.

Difference between preliminary testbed and final testbed configuration -

(1) **Number of nodes** – Four nodes shall be used in the final testbed configuration. In normal conditions, one will be the master, two will be slaves, and one will be in Off-line mode.

(2) **Total number of wires for assignment channel** – Since four nodes are available in the final testbed, the total number of wires used for the assignment channel in the final testbed configuration is twelve (4x3) instead of nine (3x3). Each module in the final testbed shall have three wires of output and nine wires of inputs.

(3) **Total number of wires for status channel** – Since four nodes are available in the final testbed, the total number of wires used for the status channel in the final testbed configuration is twelve (4x3) instead of nine (3x3). Each module in the final testbed shall have three wires of output and nine wires of inputs.

(4) **Master channel** – The Master Channel of the preliminary testbed was implemented with the SSIO provided by the microcontroller. This was because there was not enough space on the board to install an extra parallel IO device. This SSIO shall be replaced by the parallel IO device in the final testbed configuration.

(5) **Digital and analog IO** - The Monitor computer in the final testbed configuration shall generate xx digital outputs that are sampled as normal inputs by the microcontrollers, and read yy digital inputs that are normal

outputs of the microcontrollers. Also, it shall generate two analog signals via the DAQ that are read by the microcontroller.

(6) **CAN bus** (Provisional in final testbed configuration) – CAN bus shall be used to interconnect multiple microcontroller networks. Each microcontroller network shall consist of four microcontroller nodes. Networks of microcontroller network shall be achieved by interconnects with CAN bus among microcontroller networks.

2 Software Overview

The operating system provides a deterministic foreground scheduler, a background scheduler, and IO drivers. This operating system provides the same service for both master and slave application software. Therefore identical operating systems are used by both master and slave channels. A software flag indicates whether a particular module is a master channel or a slave channel, and its value is determined by the voter in real-time. The operating system runs after the basic environment is initialized properly by the bootstrap software. The bootstrap software is the code, which is in charge of setting up the interrupts, the default chip configuration, and the stack. When the system powers up, the bootstrap software is called first. The stack and the chip configuration registers are initialized as the very first thing by the bootstrap software. After the hardware is configured, interrupts are initialized with proper handlers. One of the interrupts initialized by the bootstrap software is the external interrupt. This external interrupt is initialized with a default handler, which is replaced later by the 100Hz RTI handler by the operating system software. When the operating system is called, the default handler for the external interrupt is replaced by the RTI handler. The RTI handler in the application software calls the foreground scheduler when the interrupt is triggered by the 100 Hz clock. When it is called, the foreground scheduler uses its task tables to dispatch the application software, which in turn calls the voter for

the comparison-voting that may activate (upon errors) the fault recovery software of the microcontroller network.

The following is the overview of the bootstrap software and the operating system software that are common for all master and slave modules.

Bootstrap software -- Bootstrap software is located at location 2080, to which the program counter (PC) is initialized by the CPU at startup. The bootstrap software fills the EEPROM block 2000 to 27FF with the addresses of the default interrupt handlers. The 80C196CA chip requires the interrupts to be initialized at a fixed location so that the interrupts can be handled as the chip starts up. Therefore, the bootstrap software provides a means to replace any existing interrupt handlers using the RAM at fixed locations, namely the *RAM interrupt vector table (at location 0x100)*. The default interrupt handlers of the bootstrap software calls the corresponding interrupt handlers in the RAM interrupt vector table. Interrupt handlers in the RAM interrupt vector table are the ones that can be replaced by applications at any time.

Two chip configuration registers (CCRs) have bits that set parameters for chip operation and external bus cycles. The CCRs are loaded from the chip configuration bytes (CCBs), which are located at 0x2018 and 0x 201A. When the processor is reset, the values at the CCBs are loaded into the CCRs. Therefore, the chip configuration bytes (CCBs) must be initialized properly at the fixed location in

the EERROM. Since the CCBs are located in the 2000 address block in the EEPROM and used to initialize the CCRs at power up, they should be initialized by the binary image of the bootstrap software.

The stack must be initialized to a fixed location that guarantees the maximum use of the RAM before any functions are called. Therefore, the stack pointer is initialized with the highest address available in the RAM by the bootstrap software.

Another important feature of the bootstrap software is the ability to debug the microcontrollers using a serial port. An asynchronous serial port that is provided by the 80C196CA chip is designated a debugging port. Specially written communications programs allow the PC to read or write to memory locations in any microcontroller. The serial port *receive* handler buffers an incoming character and calls the message decoder. When a legal sequence of read message characters is detected after a series of receive interrupts, the read message handler read a memory location and sends the read response message through the same port. Serial port transmit interrupts are generated by this internal event. The transmit handler simply sends a character at a time until the interrupt is disabled. The read and write feature played a major role in debugging and verifying our application software during the development process.

When the 80C196CA chip environment is properly set up, the application software (operating system) is called by the bootstrap software.

Operating system -- Major parts of the operating system are the foreground scheduler, the background scheduler, and IO drivers such as the LED I/O driver, parallel I/O driver, and synchronous I/O driver.

The foreground scheduler runs at the rate of the RTI. When an RTI is generated by the external clock source, an RTI interrupt is generated. In response to an RTI, the RTI interrupt handler checks if there is any foreground process running at the present time. When no foreground processing is found, the RTI interrupt handler calls the foreground scheduler. The foreground scheduler has a table called the *active task table*. This table contains a maximum of twenty tasks at one time. Tasks are allowed to be scheduled and unscheduled dynamically using the primitives provided by the foreground scheduler. The tasks in the active task table are processed at 100Hz and dispatched when schedule conditions are met (e.g., mode, schedule rate, values of source and destination used for comparison are equal, and value of a delay count).

The Background scheduler runs programs whenever foreground programs are completed. Whenever the foreground process returns control, the current background process resumes from where it was halted last time by the foreground

process. It yields the control of the program to the foreground process when the RTI interrupt is generated. A background active task table is used for background scheduling and is similar to the one in the foreground process. But there, no schedule conditions exist in the background scheduler. The background process simply dispatches any background tasks existing in the background active task table in a predetermined order and lets them run to completion.

Built In Test (BIT) - are tests run at startup to determine the health of the system. These tests can also be started by software requests anytime to check the health of the system. They include a RAM test, a RAM address test, and EERPOM checksum tests. Details of the tests are discussed in later sections.

Three IO drivers are provided for an application program. The Parallel IO driver is used by the voter and recovery programs to utilize the assignment and status channels. The LED IO driver is used by the BIT program to display the result of tests. Red and green LEDs are available, and they are used only for debugging purpose. A Synchronous IO driver is used by the voter and recovery programs to transmit and receive data across the microcontroller network. Details of the IO drivers are discussed in later sections.

The Following are the major blocks of the boot software and operating system software.

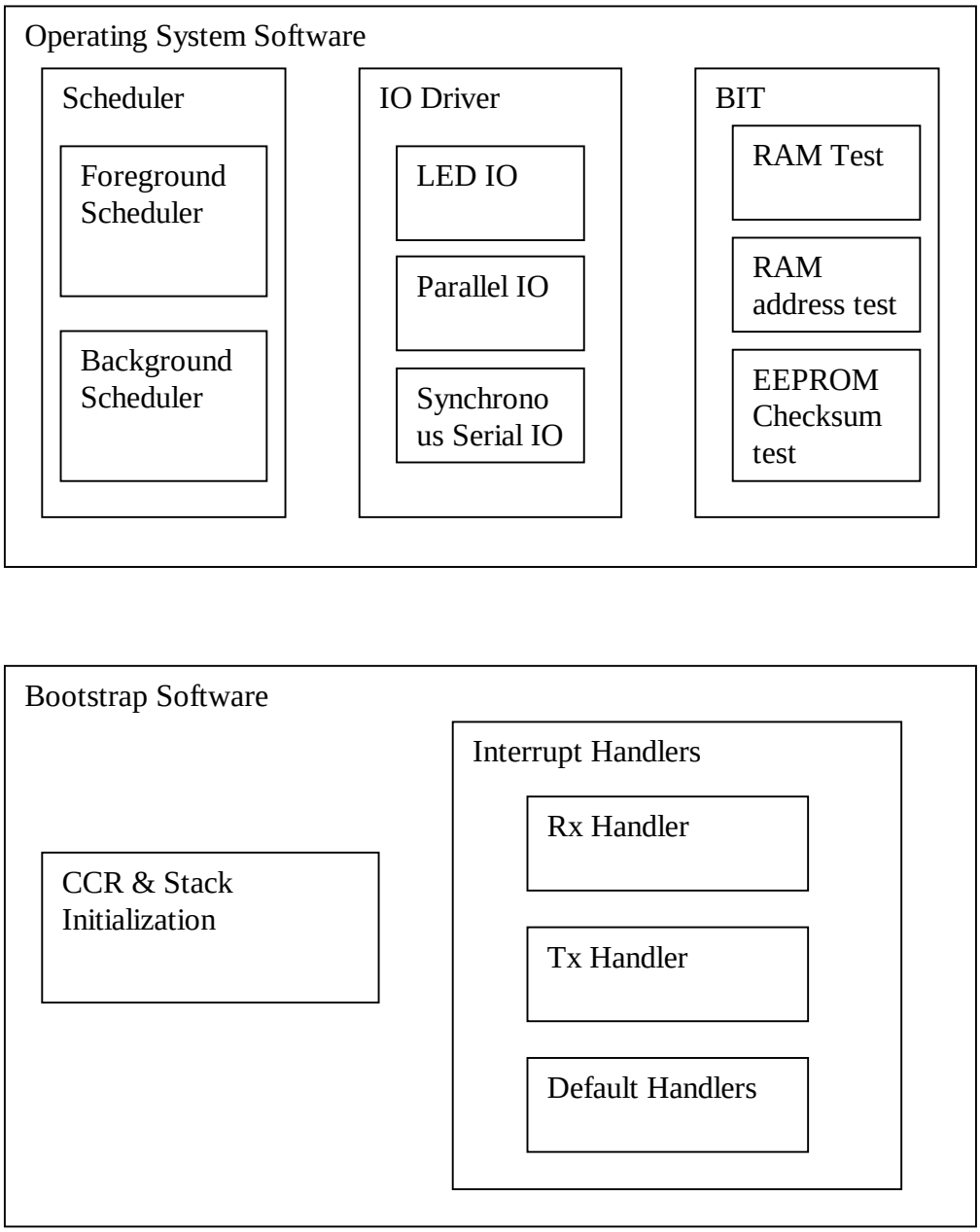

Figure 5 Bootstrap and Operating System Software Blocks

3 Software Design and Implementation

The following section provides details of the implementation of the various software modules.

3.1 Bootstrap Software

The bootstrap code is responsible for the initialization of the system before the application code is called. It initializes the stack pointer, clears both the register file and RAM. It also initializes ioport5.2 to drive the WR signal pin to interface external IO devices such as external RAM, LED and parallel IO. The Interrupt table is also initialized with default handlers.

3.1.1 Initialization of WR signal

Pin 5.2 of the 196 processor is used as the WR pin which is used to generate the WR signal to external devices such as RAM and parallel IO. The p5_dir, *p5_mode* and *p5_reg* registers are used to initialize the pin. The *p5_dir* register is used to set the direction of the pin, and the *p5_mode* register is used to set the pin either to a standard IO port pin or to a special function signal. In our application, this pin is initialized as a special function signal. The *p5_reg* register contains data to be driven out by the respective pins. When the port pin is configured as an input, the

corresponding bit of the *p5_reg* must be set. Bit 2 of *p5_reg* is initialized to 1 by default in this application. The following is the initialization algorithm.

-Initialize the p5_dir register

-Read value from p5_dir register.

-Turn on the bit 2 of the p5_dir register by OR-ing the value read with 0x04.

-Write the value back to the p5_dir register. By setting the pin 5.2 to high, this pin becomes input/open-drain output which can be used for input, output or bi-directional IO.

-Initialize the p5_mode register

-Read value from p5_mode register

-Turn on the bit 2 of the p5_mode register by OR-ing the value read with 0x04.

-Write the value back to the p5_mode register.

-Initialize the p5_reg register

-Read value from p5_reg register

-Turn on the bit 2 of the p5_reg register by OR-ing the value read with 0x04.

-Write the value back to the p5_reg register.

3.1.2 Stack Initialization

The stack grows downward when nested functions are called. To maximize the usage of the RAM, static memory is located at the lower half of the RAM, and the

stack is located at the upper half of the RAM. The stack pointer is initialized to the top of the memory 0xDFFE.

3.1.3 Register Memory and External RAM Initialization

To ensure a known and fixed state of the application at power up, external RAM and register memory are initialized to 0 by the bootstrap code at power up. This initialization code is executed very first at power-up to initialize the RAM before the application code is called. Assembly code is written to take care of the initialization, because the environment for high level C code is not ready at power-up. Memory being cleared ranges from 0x300 to 0x3FD and from 0xA000 to 0xDFFF.

3.1.4 Chip Configuration Register Initialization

The chip configuration 0 (CCR0) register controls powerdown mode, bus control signals, and internal memory protection. This register is located at 0x2018 and needs to be configured to a proper value in EEPROM to start up the 196 processor in a predefined state. The chip configuration 1 (CCR1) register enables the watchdog timer and selects the bus timing mode. This register is located at 0x201A and also it needs to be configured to a proper value in EERPOM to initialize the 196 chip to a predefined state.

CCR0 in this software is initialized to 0xDE.

Table 1 Chip Control Register 0 Initalization

Bit number	Value	Function
7:6	11	No read/write protection
5:4	10	Three wait states
3:2	11	Standard Bus Control Mode
1	0	8 bit only bus
0	0	Disable power down mode

CCR1 in this software is initialized to 0xEC.

Table 2 Chip Control Register 1 Initialization

Bit number	Value	Function
7:6	11	Standard mode for external access timing mode
5:4	01	Guarantee device operation
3	1	The first time it is cleared enable watch dog timer
2	1	8 bit only bus
1	1	Three wait states
0	0	Reserved to 0

CCR values are located in EEROM, therefore these two registers should be declared as constants in software so that they can appear in the hex file, which is used to burn the EEPROM. In this software the C196 compiler allows a fixed location CCR using the #pragma word.

3.1.5 Interrupt Table Initialization

The 196CA processor provides a flexible interrupt-handling system. One is the programmable interrupt controller and the other is the peripheral transaction server (PTS). In this application only standard interrupts are utilized. The Number of

interrupts available by the 196 processor is sixteen. Available interrupt sources and the corresponding interrupt handlers are the following.

Table 3 Interrupt Handlers

Interrupt Source	Name	Interrupt Handler
Nonmaskable interrupt	INT15	Default Handler
EXTINT Pin	INT14	Default Handler
CAN	INT13	Default Handler
SIO receive	INT12	RxHandler
SIO transmit	INT11	TxHandler
Slave port command buff full	INT10	Default Handler
Unimplemented opcode	INT09	Default Handler
Software Trap Instruction	INT08	Default Handler
Slave port input buff full	INT07	Default Handler
Slave port output buff full	INT06	Default Handler
A/D conversion complete	INT05	Default Handler
EPA Capture/Compare 0	INT04	Default Handler
EPA Capture/Compare 1	INT03	Default Handler
EPA Capture/Compare 2	INT02	Default Handler
EPA Capture/Compare 3	INT01	Default Handler
EPA Capture/Compare 4-9	INT00	Default Handler

Two types of interrupt handlers are available in the bootstrap software. One interrupt table is located in the EEPROM, which is the predefined location of the interrupt table by the microcontroller. The other is a RAM interrupt table which is located at 0x100 in the RAM, which is the predefined location of the interrupt table by the bootstrap software. When interrupts occur, default handlers in the EEPROM interrupt table are called by the microcontroller. Then, the interrupt handlers in the RAM interrupt table are called by the default interrupt handles in the EEPROM. Therefore, the interrupt handlers in the RAM are indirect interrupt

handlers and actual interrupt handlers. The RAM interrupt table is provided by the bootstrap code for the application code so that the default interrupt handlers can be replaced with ones written for application code. Replacement of the interrupt handlers in the RAM interrupt table can be achieved by the application code by simply overwriting the address of the old interrupt handler with a new one at a fixed location in the RAM interrupt table. Interrupt handlers, called by the microcontroller when interrupts occur, simply call the interrupt handlers installed in the RAM interrupt table. The RAM interrupt table is located at 0x100. The size of the table and the order of entries are the same as the ones of the EEPROM interrupt table. The External interrupt is initialized with a default handler by the bootstrap code. But this handler is replaced by the application interrupt handler to handle the 100 Hz external clock interrupt.

3.1.6 Serial IO (SIO) Initialization

The Built-in serial IO (SIO) port provides a means to communicate with external devices. As a debugging device the PC can be connected via a switch to x microcontroller's serial port. Each Serial port is initialized to 19200 BPS, no parity, 8 data bit, 1 start and 1 stop bit. Test software which will be discussed in detail is used for development and debugging.

3.1.7 Application Mode Selection

The Discrete input ioport0.4 is used for application mode selection. The Discrete input ioport0.4 is read after the series of initializations. When the ioport0.4 input is low at the time of the input, bootstrap mode is set to TRUE and enters into a infinite idle mode. But when the ioport0.4 input is high at the time of the input, the startup function of the application code is called.

3.1.8 Interrupt Handling

3.1.8.1 Default Handler

Most of the interrupts are handled by the default handler, since interrupts should not occur unless they are specifically enabled for certain purposes. The following is the list of interrupts that are not initialized and handled by the default interrupt handler.

EPA Capture4 Interrupt

EPA Capture3 Interrupt

EPA Capture2 Interrupt

EPA Capture1 Interrupt

EPA Capture0 Interrupt

AD Conversion Interrupt

Slave Port Output Buff Empty Interrupt

Slave Port Input Buff Full Interrupt

Unimplemented opcode Interrupt

Trap interrupt

Slave Port Command Buff Full Interrupt

SSIO Chan0 Transfer Interrupt

SSIO Chan1 Transfer Interrupt

CAN Bus Interrupt

External Interrupt

Nonmaskable Interrupt

3.1.8.2 SIO Rx Interrupt Handler

The Serial IO port is designated for the purpose of debugging. Communication through this serial port is triggered by a command sent by the PC. Two types of commands are available for the PC to communicate with the embedded software. One is a *read* command and the other is a *write* command. Both commands are recognized by the embedded software with a predefined message format. The message contains a header, a command ID, a data address and a checksum. When a *read* command is issued by the PC, the *read* message contains a header, a *read* command ID, and an absolute address of the *read* location along with one byte checksum. When a *write* command is issued by the PC, the *write* message contains a header, a *write* command ID, an absolute address of the *write* location, and a data to be written along with one byte checksum. When the *read* command is

decoded successfully, the SIO Rx handler reads the address requested by the *read* command. It forms a response message with the data read from the address requested by the *read* command. The response message is buffered and the *send request* flag is set to TRUE for the SIO Tx handler. When the *write* command is decoded successfully, the SIO Rx handler writes the data to the address location specified by the *write* command. Then it forms a response message that indicates the write command is successful. This response message is buffered and the *send request* flag is set TRUE for the SIO Tx handler. This *read* and *write* command reception and the handling of the commands happens while the application is running in the foreground or background. Since these handlers only get activated and go away through interrupts, the *read* and *write* commands are totally transparent to the application programs.

3.1.8.3 SIO Tx Interrupt Handler

The SIO Tx handler gets activated only by the SIO Rx handler. When a command by a PC is decoded, the response message is buffered in the transmit buffer and the SIO Rx handler activates the SIO Tx handler by enabling the Tx interrupt. When the SIO Tx handler is called, it checks if there are any characters in the transmit buffer. If the buffer is not empty, it sends a character to the SIO and returns the control. Since the Tx interrupt is not disabled, the Tx handler gets triggered again when the SIO Tx buffer becomes empty. The SIO Tx handler shall send a character at a time in the same fashion until the Tx buffer becomes empty. When

the last byte of the Tx buffer is transmitted by the transmit handler, it disables the Tx interrupt. Disabling of the transmit interrupt does not interfere with Tx operation of the last byte.

3.2 Startup Initialization

The initialization of the application is called after the bootstrap code has been executed and right before any application code is called. The first thing it initializes is the stack pointer. It initializes the stack to the top address available in the RAM, and it calls the 'C' function 'main'. When 'main' is executed, The Built In Test (BIT) is called to determine the health of the system before it runs any piece of an application code.

3.3 Built In Test

3.3.1 RAM Test

The purpose of the RAM test is to determine if the RAM is capable of reading and writing. While performing the RAM test, interrupts are disabled to prevent interrupt handlers from using the RAM while the RAM is being tested.

The following is the algorithm of the RAM test.

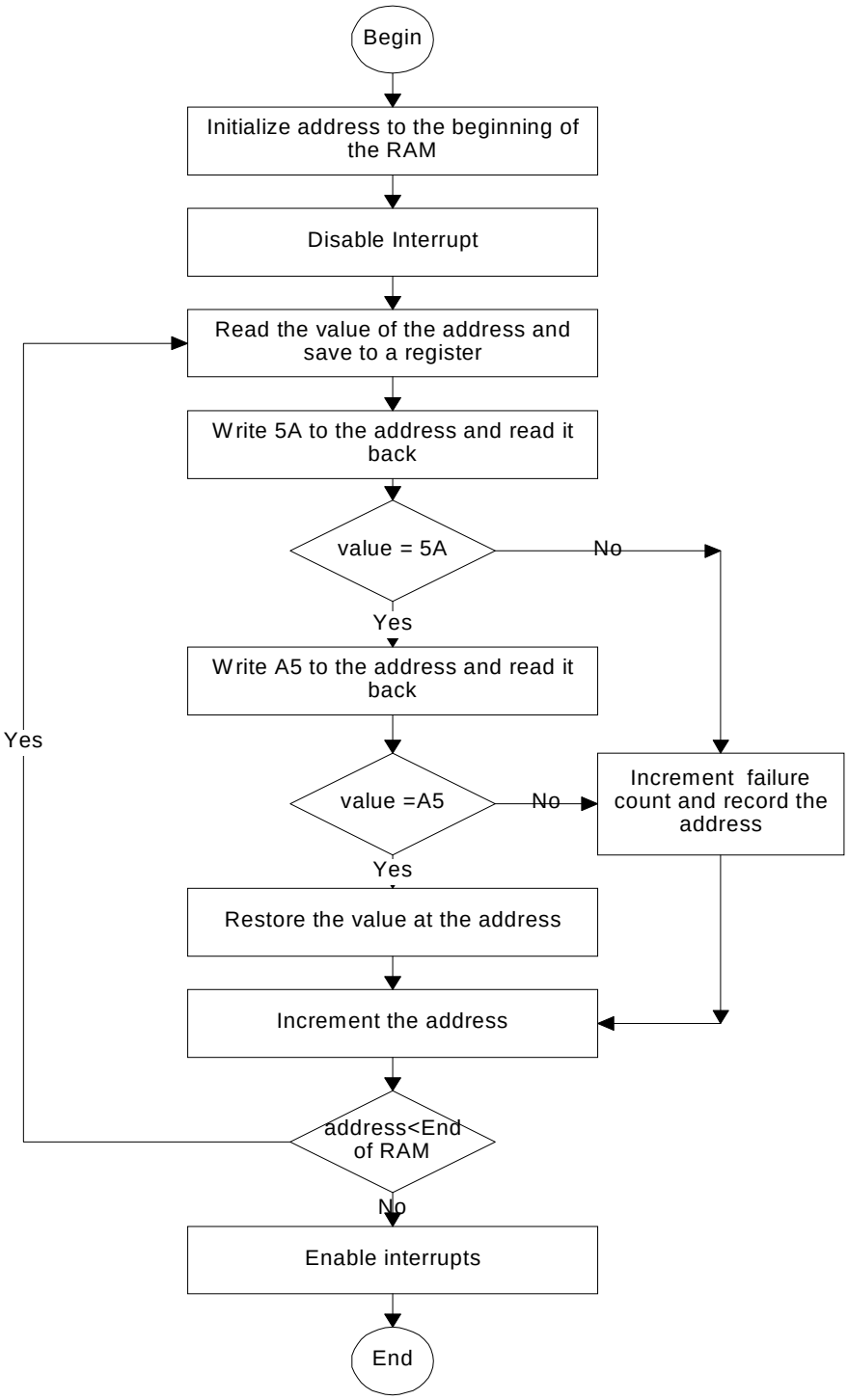

Figure 6 RAM Test Algorithm

3.3.2 RAM Address Test

The purpose of the RAM address test is to detect any problems in the address lines which control the RAM. The RAM address line can fail by being stuck at high or low. When the address line 0 is stuck at high, data written to an address 0xA000 shall be written to 0xA001 instead of the address 0xA000. On the contrary, when the address line 0 is stuck at low, data written to an address 0xA001 shall be written to 0xA000 instead of the address 0xA000. This situation can be easily tested and detected by the RAM address test. The RAM address test should be executed at startup, because there will be conflicts with codes that share the same address range being tested.

The following is the algorithm of the RAM address Test.

```
Initialize character pointer to RAM begin address
For bit location =0 to bit location 11
  Start
  Character pointer1 = character pointer + (1<<bit location)
  Save data pointed by character pointer
  Save data pointed by character pointer1
  Write 0 to *character pointer1
  Write 0x5a to *character pointer
  if *character pointer1 is 0x5a
    then declare ram address test failure
  endif
```

```
Write 0 to *character pointer
Write 0x5a to *character pointer1
if *character pointer is 0x5a
  then declare ram address test failure
endif
Restore data pointed by character pointer
Restore data pointed by character pointer1
End
```

3.3.3 EEPROM Checksum Test

The purpose of the EEPROM checksum test is to detect incorrect data in the EEPROM area. When the EERPOM is burned, the checksum of the entire code is pre-calculated and stored at a fixed location. This test calculates the checksum of the EEPROM and compares it with the predetermined checksum. A two-byte sum is used in this application.

3.4 Interrupt Handler

The interrupt table is initialized at power-up. Eighteen different interrupts are provided by the 196CA chip. In this application, three interrupts are handled with specific interrupt handlers. The rest of the interrupts are handled when they occur, but they are handled by the default handler.

3.4.1 External Interrupt Handler

The External interrupt is an interrupt generated by the external IO. The Data acquisition board is used to generate an 100 Hz interrupt which is connected to the IOPORT 2.2. When the external interrupt is called, it calls the foreground scheduler, which sets a flag *fg_entered* to TRUE. This flag *fg_entered* gets cleared to FALSE when the foreground scheduler returns the control at the end. If this flag is TRUE before the foreground scheduler is called, the call to the foreground scheduler is skipped and the foreground overrun count is incremented for monitoring purpose.

3.4.2 Serial IO Receive Interrupt

The serial IO receive interrupt is handled by the default interrupt handler which is installed by the bootstrap code. This interrupt is not reinitialized by the application code. When an interrupt occurs, it reads the character in the *sbuf_rx* and uses it for decoding. When the sequence of characters is detected as a read command, it generates a read command internally. When this command is generated, the read handler reads the memory location that is requested by the read message, forms a output message with the result, sends the first character of the message and enables the transmit interrupt.

3.4.3 Serial IO Transmit Interrupt

The serial IO transmit interrupt is handled by the default interrupt handler installed by the bootstrap code. This interrupt is not reinitialized by the application code. The transmit interrupts are generated by internal events. When the receive handler decides to send a data stream, it puts the stream of data in the buffer with the length to transmit and enables the transmit interrupt. So when the transmit interrupt occurs, the transmit interrupt is called by the processor, and this handler writes a character from the buffer set up by the receive handler. After the transmit of one character, the length of the buffer is decremented by one. When the length of the transmit buffer becomes zero, the transmit interrupt is disabled before the transmit interrupt returns the control.

3.5 Foreground Process

Two different priority groups run in the application. One is the foreground process and the other is the background process. The foreground process has higher priority than the background process. Whenever the foreground process is revoked, the background process loses the control to the foreground process and only regains the control when the foreground returns the control. Since the foreground is triggered by the 100 Hz clock, there is only 10 msec maximum time available to the foreground. When the foreground process exceeds 10 msec, we call this ‘overrun’ by the foreground process. Therefore the portion of the application programs that are dispatched by the foreground scheduler should

ensure that the sum of total foreground process time after any RTI should not exceed the 10 msec maximum time allocated. The foreground and background processes are implemented by the interrupt scheme. The foreground process is merely the 100 Hz interrupt handler that takes more time to finish its job than any other normal interrupts do. When the 100 Hz clock interrupt is generated via the external interrupt, the external interrupt handler invokes the foreground scheduler namely the foreground process. The design of the software should guarantee that the overrun conditions will not crash the software. If an overrun condition occurs, the external interrupt handler simply skips the execution of the foreground scheduler for the frame. It repeats skipping the frame until the foreground process that has caused the overrun condition releases control.

3.6 Background Process

The background has the lowest priority, so that it yields control to the foreground process and any other interrupts. The background process calls the background scheduler. The background scheduler runs the background tasks in the background active task table. Background tasks shall run until completion, and foreground tasks can schedule new ones and kill old ones. Details of the background scheduler shall be discussed in detail in the later sections.

3.7 Foreground Scheduler

The foreground scheduler is dispatched from the interrupt that is triggered by the 100 Hz external clock interrupt. Therefore the maximum rate group called by the scheduler is 100Hz. A function that is dispatched by the scheduler is called a task.

Each task scheduled by the foreground scheduler has a task descriptor. The task descriptor consists of a process ID (PID), a task function address, a schedule mask, a mode word, two address words and wait time.

The task function address is initialized at power up with the address of the function to be scheduled. The schedule mask word is a 2-byte unsigned word used by the scheduler to control the rate of the given task and the frame in which the task is scheduled. When all the bits in the 2-byte schedule mask are high, the scheduler dispatches the task at the maximum rate which is 100Hz. When the schedule mask word has only eight bits high, the scheduler shall run the task at the half of the maximum rate which is 50 Hz. The smallest frame that is used by the scheduler is called the minor frame. Since the scheduler uses a 16-bit schedule mask, the scheduler provides sixteen minor frames from zero to fifteen. The schedule mask is also used to control the frames in which the task is scheduled. For example, if the schedule mask of a task is 0x0101, the task shall be dispatched two times out of sixteen frames which makes 12.5 Hz and it shall be dispatched at the frame 0 and

8. A mode word is used to control scheduling of tasks based on the mode of the foreground process. When a task is initialized to run only in the normal mode, the scheduler ensures scheduling of the task happens only in the normal mode.

A Process ID, 2 address words and wait times are used for the primitives provided by the foreground scheduler. Two address words are used for the primitive *when*. Wait count is used by the primitive *wait*. Process ID is used by all four primitives which are *wait*, *when*, *start* and *end*. The usage of the primitives shall be discussed in detail in later sections.

Two types of task tables are used by the foreground scheduler. One is called an active task table and the other is called a task list table. An *active task table* is an array of task pointers which either points to one of the entries in the *task list table* or NULL. Any tasks in the *active task table* are processed by the foreground scheduler at 100 Hz and dispatched according to the schedule mask in the task. Tasks to be scheduled are called 'active' and they are pointed to by one of the pointers in the *active task table*. The fact that the task is pointed to by a pointer in the *active task table* tells that the task is active. When the task needs to be disabled, the pointer pointing to the task can be reinitialized to NULL. The maximum size of the *active task table* is twenty. When the scheduler is initiated, the scheduler starts dispatching the tasks from the first entry until the end of the *active task table* if the entries are not NULL pointers. If an entry of the *active task*

table is a null pointer, that entry is skipped. Thus the task table should be initialized to NULL pointers if the entries of the tables are not used. The following is the data structure of the *active task table*, the task descriptor and the elements of the task descriptor.

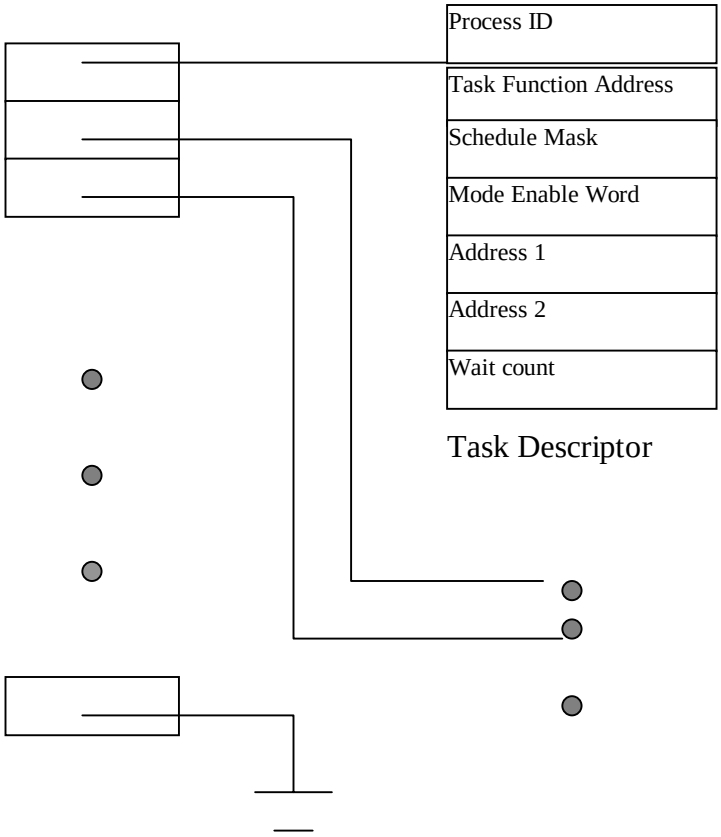

Figure 7 Data structure of Task Table and Task Descriptor for Scheduler

The task list table is an array of task descriptors. As discussed above, a task descriptor contains a process id, a function address, a schedule task, a mode word, an address 1, an address 2, and a wait count. An Array of forty tasks is predefined at start up. This task list table consists of tasks active from the startup and tasks that are not active at startup but possibly active throughout the span of the whole program life.

The Active task table is initialized at startup with tasks to be scheduled by default. A task entry from the task table can be added to the active task table using the primitive *start*, and any active task entry can be removed from the active task table using the primitive *end*. To identify which tasks to add to or remove from the active task table, the *start* and *end* primitives take the process id as a parameter.

The following are descriptions of each primitive provided by the foreground scheduler.

The primitive start takes a process ID as a parameter. When the primitive start is called, the entire task list table is searched with the process id provided as a parameter. When the task is found, the last available entry in the active task table is initialized with the address of the task found. A duplicate entry in the active task table is not allowed. Therefore the active task table is examined first to find a task with the same process ID.

The primitive end also takes a process id as a parameter. The active task table is searched from the first entry to the end of the table. When a task with the same process id is found, the pointer in the active task table is simply reinitialized to null.

The third primitive provided by the foreground scheduler is *when*. The primitive *when* takes a process ID, a pointer to a first word and a pointer to a second word. When this primitive is called, the task with the process ID is added to the active

task table if it does not exist in the active table. Also two addresses in the task descriptor are initialized to two pointers passed as parameters. The foreground scheduler simply checks to see if the contents pointed by these two pointers are the same. If they are the same, the task is processed. Otherwise it is skipped until the values pointed to by these two pointers match.

The last primitive provided by the foreground scheduler is *wait*. This wait primitive is merely a delay function. It takes a process ID and a *wait count*. This primitive requires a task to be active prior to the call. The foreground scheduler shall delay the execution of the task according to the wait count specified by the parameter.

3.8 Background Scheduler

The design approach and the implementation of the background scheduler are similar to the ones in the foreground scheduler. The background task descriptor consists of a process ID and a function address. Unlike the foreground task descriptor, the schedule rate mask is not a part of the background task descriptor because a typical background process runs longer than a foreground frame and is not restricted by any time frame.

The function address in the background task descriptor is used by the background scheduler to dispatch the process. A process ID is also provided to implement the

background scheduler primitives. There are two tables in the background. The background active task table and the background task list table are the two tables used by the background scheduler. The background active task table is an array of task pointers and has a maximum number of ten entries. The entries are either null pointers or point to a background task in the background task list table. A background task can be active when the task descriptor of the task is pointed to by one of the pointers in the background active task table. The background task list table is an array of background task descriptors. It contains twenty background tasks which are active from startup or are possibly active throughout the span of program life. So tasks which possibly can be active in the future are initialized in the background task list table at startup. Two background scheduler primitives are provided to activate or deactivate any background tasks. The following is the description of the background scheduler primitives.

The primitive *bkstart* takes a process ID as a parameter. When the primitive *bkstart* is called, the entire background task list table is searched with the process ID provided as a parameter. When the task is found, the last available entry in the background active task table is initialized with the address of the task found. Duplicate entries in the active task table are not allowed. Therefore the background active task table is examined first to find a task with the same process ID.

The primitive *bkend* also takes a process IDd as a parameter. The background active task table is searched from the first entry to the end of the table. When a task with the same process ID is found, the pointer in the background active task table is simply reinitialized to null.

3.9 IO

3.9.1 Synchronous Serial IO (SSIO)

The 196CA chip provides a synchronous serial IO port for simultaneous, bi-directional communications between this device and another asynchronous serial IO device. The SSIO is used for the communication between the network nodes.

(Note: The SSIO is used as a Master Channel for the preliminary testbed only. It will be replaced by a parallel IO device in the final testbed)

Each microcontroller node runs an application program which is a filter in this particular application. The output of the filter is shared for voting among the microcontroller network nodes. Output of the filter application is passed to a voting driver. The voting driver in the master channel outputs data across the channels using the SSIO device provided by the microcontroller. The voting driver in the slave channel uses its SSIO for input. When the voting driver in the slave channel is called by an application, it polls the Rx buffer of the SSIO with a timeout.

3.9.1.1 SSIO Device Initialization

Each SSIO device is initialized differently depending on the way in which it is used for IO process. The SSIO is a IO port that is used for synchronous communication. One needs to be a master device which drives the clock output

and the others need to be a slaves which use the clock output from the master SSIO to synchronize with the master SSIO.

3.9.2 Parallel IO

The 8255A from Intel is used as a parallel IO device. The Intel 8255A is a general purpose programmable IO device designed for use with Intel microprocessors. It has 24 IO pins which may be individually programmed in two groups of twelve and used in three major modes of operation. In the first mode (MODE 0), each group of twelve IO pins may be programmed in sets of four to be input or output. In MODE1, the second mode, each group may be programmed to have eight lines of input or output. The third mode of operation (MODE 2) is a bi-directional bus mode which uses eight lines for a bi-directional bus, and five lines for handshaking, borrowing one from the other group.

Parallel IO in our project is used for assignment channels or status channels which serve the essential part of microcontroller network's communication. In the prototype board, only three nodes were used to form a network compared to four nodes in the final board configuration. The assignment channel is used to vote for a master. Each channel needs three output wires to vote for a master channel and six input wires to read the votes from other channels. The status channel is used to share the voting information between channels. For the master channel, the status channel is used to send commands to slave channels such as 'check data', 'retry',

‘timeout’ and ‘internal error’. For the slave channels, the status channel is used to report the voting results to a master channel such as ‘agree’, ‘disagree’, ‘timeout’ and ‘internal error’. Therefore three groups of three wires are needed to form status channels. Each channel uses three output wires for its status channel and six input wires to read the status channel data from other channels.

3.9.3 LED IO

Two LEDs are used in the test bed for debugging purposes. One is red and the other is green. The red LED is used to indicate a failure in the system, and the green LED is used to indicate that the status of the system is healthy. When the system boots up, it flashes both LEDs 4 times at 2 Hz to indicate the system is powering up. When the bootstrap code finishes the initialization, the application is invoked by the bootstrap code. Built In tests, such as EEPROM checksum test, RAM test, and RAM address tests, are executed to check the health of the system. When all the tests pass, the green LED is lit. Otherwise the red LED is lit.

Addresses and operations of the LEDs are controlled by a PAL. When a write command is issued by a CPU, the command is decoded and recognized by the PAL. When the PAL recognizes the *write* command to LED addresses, the PAL turns the LEDs on and off, depending on the input from the CPU.

Following is the addresses of the LED IO.

LED	Address	On	Off
Red	F000	Write 1	Write 0
Green	F100	Write 1	Write 0

Figure 8 LED IO Address

3.10 Application

3.10.1 Filter Processing

A simple application software was introduced in the microcontroller network to test and verify the voting and failure recovery algorithms. The application used in the software is a low pass filter. A brief introduction to the low pass filter follows. How it was converted to the software which implements the low pass filter in a discrete time domain will be discussed in detail.

The transfer function of the low pass filter is:

$$H(s) = 1/(\tau s + 1) = \omega_p / \left(s + \omega_p\right) \quad \text{where} \quad \omega_p = \frac{1}{\tau}$$

The *s* domain can be translated into a discrete time domain using a substitution.

Using $s = \frac{2}{T}\left(\frac{1 - Z^{-1}}{1 + Z^{-1}}\right)$

Therefore,

$$\frac{y_n}{x_n} = \frac{\frac{T\omega_p}{T\omega_p + 2} + \left(\frac{T\omega_p}{T\omega_p + 2}\right) z^{-1}}{1 + \left(\frac{T\omega_p - 2}{T\omega_p + 2}\right) z^{-1}}$$

Solve for y,

$$y_n = \left(\frac{T\omega_p}{T\omega_p + 2}\right) x_n + \left(\frac{T\omega_p}{T\omega_p + 2}\right) x_{n-1} + \left(\frac{2 - T\omega_p}{T\omega_p + 2}\right) y_{n-1}$$

When we apply the formula above to programming,

y_n = Filter Output

x_n = Filter Input

y_{n-1} = Filter Output in the previous frame

x_{n-1} = Filter Input in the previous frame

T = Input Sampling Interval (e.g. 10msec if the filter function gets input at 100 Hz)

ω_P = cutoff frequency (e.g. 12.566 by the formula $\omega=2\pi f$ if 2Hz lowpass filter is used)

The code converted from the equation above is the following:

```
typedef struct {
   double   Xp;       // input in the previous frame
   double   Yp;       // output in the previous frame
```

```
   double   Kx;       // coefficient for X
   double   Kxp;      // coefficient for Xp
   double   Kyp;      // coefficient for Yp
} FilterDataType;

double LowPassFilter (double X, FilterDataType *FilterData)
{
  double Y;

  Y  = FilterData->Kx * X + FilterData->Kxp * FilterData->Xp
         + FilterData->Kyp * FilterData->Yp;
  FilterData->Xp = X;
  FilterData->Yp = Y;
  return Y;
}
```

4 Test Software and Test Environment

Two types of test tools were used to test the embedded software The first is the data acquisition board that had vendor-supplied software. The other is a communication window software written for this program. The software package that comes ~~out~~ with the data acquisition board allows users to program and control the data acquisition board in real time. The data acquisition board was used to provide digital inputs to the microcontroller network. One of the most important inputs from the data acquisition board is the real time clock input. The 100Hz pulse signal is provided to ioport2.2, which is an external interrupt source to the 196CA chip. This external input signal is connected to all 4 boards.

The other software is a window PC communication program. This program is written to provide the basic communication interface to the microcontroller network. This program uses the communication port of the PC to communicate with the microcontroller network. A special software handshaking mechanism was used in both the PC communication program and the embedded message handler in the 196 microprocessor. Details of the implementation of the test software are discussed in detail in the following sections.

4.1 Fault Tolerant Microcontroller Network Monitor

4.1.1 Basic Commands

The basic command set supported in the program is the followings.

Help : Displays command list.

Exit : Quits the program.

Port : Sets/Resets the communication port being used.

Read : Reads a word at a specific address.

Readb: Reads a byte at a specific address.

Write : Writes a word to a specific address.

Writeb : Writes a byte to a specific address.

R : Repeats previous command.

Help

Help is available for user to display available command sets. No arguments are needed.

Exit

This lets the user terminate the program. Although the communication, parsing and decoding process is terminated by the exit command, the actual window should be terminated by selecting the exit submenu under the file menu. Alternately, x on the

upper right corner of the window can be clicked on by the mouse to terminate the program.

Port

>*Port [communication port]*

e.g. >port 1 (Sets the default communication port to COM1)

The *port* command lets the user select the communication port of a PC. Available ports are 1 and 2. The command argument is the port number. Every time this command is issued, the communication port is not only selected for the following command but also is reinitialized with the default port setting (19200bps, 8 bit data, 1 start bit 1 stop bit). If a port command is not issued by a user at the startup of the program, COM1 is selected and initialized by default.

Read

>*read [address in hexadecimal number]*

e.g. >read 200C (Reads a word data at an address location 200C)

The *read* command takes one argument as the address to read. The address should be specified in hexadecimal without the symbol that indicates hexadecimal (eg leading 0x). When this command is issued by a user, a read command message is transmitted to a microcontroller. After the microcontroller responds, It displays the content of the address specified by the user command in hexadecimal format. If

there is no response for any reason, the program quits waiting after two seconds and displays a timeout error message.

Readb

>*readb [address in hexadecimal number]*

e.g. >readb 2001 (Reads a byte data at an address location 2001)

Readb command takes one argument as the address to read. It reads the value as a byte from the specified address by the user. The address should be specified in hexadecimal without the symbol that indicates hexadecimal (eg leading 0x). When this command is issued by a user, a read command message is transmitted, and when the microcontroller replies to the command, a byte is returned in hexadecimal format. If there is no response for any reason, the program quits waiting after two seconds and displays a timeout error message.

Write

>*Write [address in hexadecimal number] [data in hexadecimal number]*

e.g > write 1234 abcd (Writes a data abcd at the address 1234)

The *write* command takes two arguments. The first argument is an address to write in hexadecimal format. The second argument is the data to write in hexadecimal format. When a *write* command is issued by the user, a *write* command message is transmitted by the program through the communication port. It will wait for the response from the microcontroller. When the command is successfully executed,

the microcontroller sends back a message saying the command is successful. When this reply message is received by the program, it displays OK on the command prompt. If there is no response for any reason, the program quits waiting after two seconds and displays a timeout error message.

Writeb

>*Write [address in hexadecimal number] [data in hexadecimal number]*

e.g > write b000 ab (Writes a byte data *ab* at the address *b000*)

The *writeb* command takes two arguments. The first argument is an address to write in hexadecimal format. The second argument is a byte of data to write in hexadecimal format. When a *writeb* command is issued by a user, a *writeb* command message is transmitted by the program through the communication port. It will wait for the response from the microcontroller. When the command is successfully executed, the microcontroller sends back a message saying the command is successful. When this reply message is received by the program, it displays OK on the command prompt. If there is no response for any reason, the program quits waiting after two seconds and displays a timeout error message.

R

>R

This command does not require any argument. Any command used previously is buffered in the memory. When the R command is issued by a user, the command

used last is repeated. When a command has to be repeated for many times, this command is very useful.

4.1.2 Software Design and Implementation

4.1.2.1 Communication Port Initialization

A data sheet for the PC16550D Universal Asynchronous Receiver/Transmitter with FIFO was used to program the PC communication port. The 16550 is a UART generally used in PCs. The Default communication port selected at start up is COM1. The other default setup is 19200 BPS, no parity , 8 data bit, 1 stop bit and 1 start bit. The only changeable option provided to the user is the configuration of the communication port using the *port* command. Other setup conditions are assumed by default and fixed in this program. Unlike most of the communication program, this program did not use the interrupt feature of the UART. As soon as the user command is decoded, a command string is transmitted through the communication port, and the rx buffer of the communication port is polled for the response. When the communication port is not connected properly or the microcontroller is down for some reason, the polling routine can sit there and wait forever. To prevent this situation the time out mechanism is used in the polling algorithm. As soon as the command string is transmitted, the time is recorded, and the timer is constantly monitored while the rx buffer is being read. For both *read* and *write* commands, a two second time out is used for polling.

When there is no response for two seconds, the program indicates an error to the user by printing the error message on the command prompt.

4.1.2.2 Command Handler

A command parser and decoder runs right after the initialization of the communication port. It polls the user keyboard input and repeats parsing and decoding until the program gets terminated by the user with the *end* command. When a legal command is detected, it calls an appropriate handler for the command. Available command handlers and their descriptions are discussed in great detail in the following sections.

4.1.2.3 Read Command Handler

The first thing the *read command handler* does is to flush the Rx FIFO buffer of the UART. The *read command handler* waits for six characters before it starts decoding the reply message. Any characters available in the UART FIFO before the *read command handler* sends a read command message are not part of the reply message. These characters can cause an error when it decodes the reply message.

After the flushing of Rx buffer, the *read command handler* starts forming a message to transmit. The format of the read command message follows:

Byte 0 : Header 0x39

Byte 1: Message Length without checksum (=5)

Byte 2: Read message ID (=1)

Byte 3: Low byte of address to read

Byte 4: High byte of address to read

Byte 5: Checksum from byte 1 to byte 4

Once the message is formed it is transmitted through the port, and the *read command handler* starts a two-second timer. Decoding is started when the reply message is received within two seconds. Otherwise *the read command handler* quits polling of Rx buffers and displays the timeout error message on the command prompt. When the command is successfully executed, the data read from the read reply message is extracted and displayed on the command prompt.

4.1.2.4 ReadB Command Handler

The first thing the *readb command handler* does is flush the Rx buffer of the UART. The Algorithm is the same as the one in the *read command handler*.

The functional difference between *read and readb command handler* is the number of bytes to read. The r*ead command handler* issues a command that requests two bytes of data from the microcontroller. On the other *hand the readb command handler* requests one byte of data from the microcontroller. This command is

provided because some devices work in an 8 bit mode such as the LEDs and parallel IO.

After the flushing of the Rx buffer, the *readb command handler* starts forming a message to transmit a command message. The format of the readb command message is as follows.

Byte 0 : Header 0x39

Byte 1: Message Length without checksum (=5)

Byte 2: ReadB message ID (=2)

Byte 3: Low byte of address to read

Byte 4: High byte of address to read

Byte 5: Checksum from byte 1 to byte 4

Once the output message is formed the message is transmitted through the port. Timeout and decoding is handled the same as in the read command handler. When the command is successfully executed, one byte of data read in the *read* reply message is extracted and displayed on the command prompt.

4.1.2.5 Write Command Handler

The *write command handler* is called when a write command is issued. It flushes the Rx buffer of the UART like other handlers do. The following is the format of the write command message.

Byte 0 : Header 0x39

Byte 1: Message Length without checksum (=7)

Byte 2: Write message ID (=3)

Byte 3: Low byte of address to write

Byte 4: High byte of address to write

Byte 5: Low byte of data to write

Byte 6: High byte of data to write

Byte 7: Checksum from byte 1 to byte 4

As soon as the write command message is transmitted, it starts a two second timer for the reply message. When the write command is successfully executed from the microcontroller, the mocrocontroller sends an OK reply message and the program displays the OK message on the command prompt.

4.1.2.6 WriteB Command Handler

What *writeb command handler* does is similar to the write command handler. It flushes the Rx buffer of the UART, forms a writeb command message and sends it through the communication port. The following is the format of the *writeb* command message.

Byte 0 : Header 0x39

Byte 1: Message Length without checksum (=7)

Byte 2: Writeb message ID (=4)

Byte 3: Low byte of address to write

Byte 4: High byte of address to write

Byte 5: Low byte of data to write

Byte 6: Not used

Byte 7: Checksum from byte 1 to byte 4

As it appears above, byte 6 of the message does not contain any data because the command itself is for *byte write*. It displays an OK message when it receives the reply message. Otherwise it displays an error message on the command prompt.

5 Conclusion

The prototype boards have been built for this system. The system software was designed and partitioned into three major parts. Bootstrap software is built to initialize the stack, interrupts and other features provided by the 80196CA microprocessor. An operating system is built on top of the bootstrap software. The operating system includes a foreground process, a background process, a foreground scheduler, a background scheduler, IO drivers for serial IO, synchronous serial IO, LED IO, and parallel IO. An application was written and run on the top of the operating system. Core voting and recovery algorithms have been converted to software, and preliminary versions of the voting and recovery software were tested and verified using three prototype boards. Although additional work is needed to bring this system to its final form, the software developed in this project has been instrumental in the development of a new class of fault-tolerant systems based on the use of microcontrollers.

7 References

1. 8XC196Kx, 8XC196Jx, 8XC196CA Microcontroller Family User's Manual by Intel June 1995

2. PC1655D Universal Asynchronous Receiver/Transmitter with FIFOs by National Semiconductor, June 2, 1995

3. 8255A-5 Programmable Peripheral Interface by Intel, September 1993 (Order Number 231308-004)

4. RENN97 - David A. Rennels, UCLA "A Fault-Tolerant Embedded Microcontroller Testbed", Presented at the 1997 Pacific Rim International Symposium on Fault Tolerant system.