DEVELOPING A SENSOR NETWORK FOR DATA ACQUISITION IN THE OILFIELD ENVIRONMENT

By

RICHARD L. CHRISTIE

Bachelor of Science

DeVry Institute of Technology

Columbus, Ohio

1986

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1996

DEVELOPING A SENSOR NETWORK FOR DATA

ACQUISITION IN THE OILFIELD

ENVIRONMENT

Thesis Approved:

[illegible]

Thesis Advisor

[illegible]

Jacques E. LaFrance

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my thesis advisor, Dr. Mitchell Neilsen for his intelligent supervision, constructive guidance, and friendship. I would also like to thank my thesis committee members, Dr. Marvin Stone and Dr. Jacques LaFrance, for their guidance in the fulfillment of this project.

Finally, I would like to express my gratitude to my friends at Dowell Schlumberger. Without their valuable insight and tireless guidance, I would have been unable to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

API	Application Program Interface
CTSI	Coiled Tubing Sensor Interface
SIM	Sensor Interface Module
mA	Milliamperes
VDC	Volts Direct Current
A/D	Analog-to-Digital
msec	Milliseconds
PSI	Pounds Per Square Inch

CHAPTER I

INTRODUCTION

The increasing demand on service quality and customer satisfaction at Schlumberger Dowell, Inc. has created the need for a new data acquisition system for the coiled tubing business line. This is a proposal of the new data acquisition system currently being developed at the Coiled Tubing Engineering facility in Rosharon, Texas.

Schlumberger Dowell, Inc. (Dowell) is an oilfield service company that specializes in the placement and pumping of wellbore fluids for well construction and well completion. The readers of this proposal may be familiar with Haliburton Well Services; Dowell is a direct competitor of Haliburton. The coiled tubing business line is a unique service that specializes in the placement of fluids in the wellbore by placing sections of small (0.75-3.5 inch) flexible metal tubing inside the larger diameter wellbore. The operational concept of a coiled tubing system involves feeding a continuous string of small diameter tubing into a well to perform specific well servicing operations without disturbing existing wellbore casings and equipment. A Coiled Tubing Unit (Figure 1) is a portable, hydraulically-powered service system designed to insert and retrieve a continuos string of tubing concentric to larger inside diameter of wellbore casings. At present, coiled tubing is available in sizes from 0.75 inch to 3.5 inch outside diameters and can have lengths up to 20,000 feet.

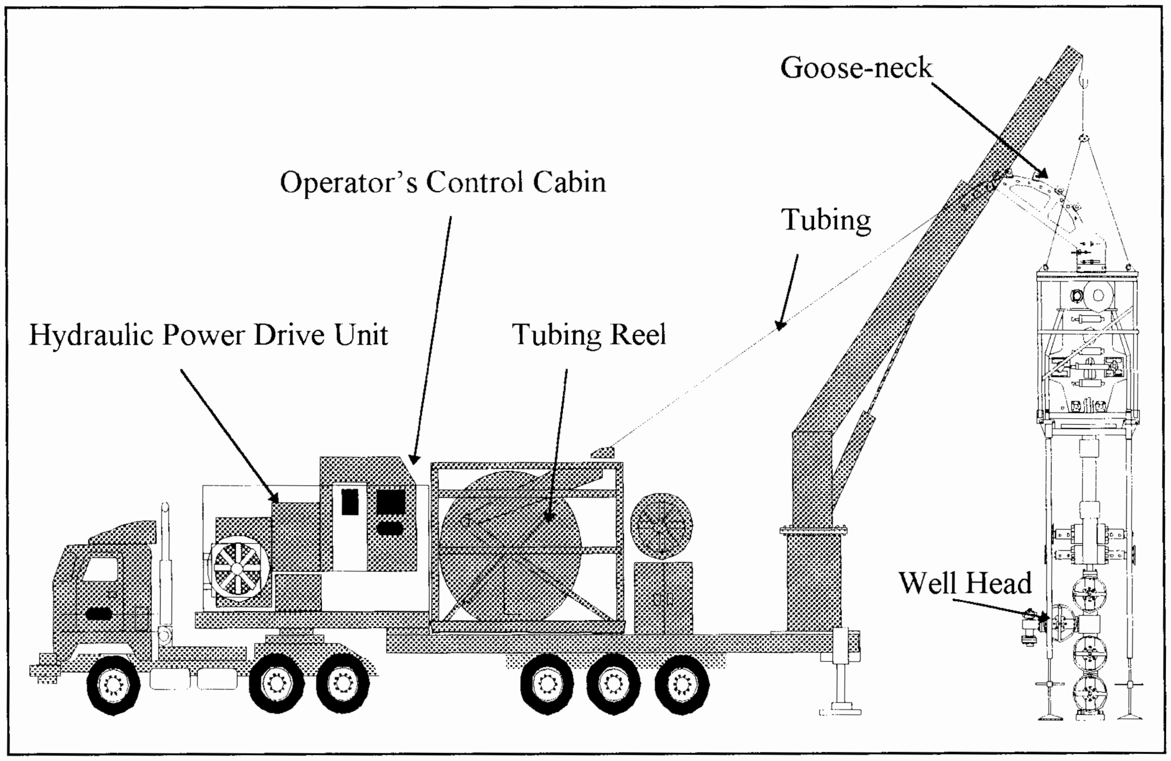

Figure 1: Typical Coiled Tubing Unit

Dowell has been in the fluid pumping business for over 40 years, but only aggressively entered the coiled tubing market in 1988. This relatively new business line has lead to problems in the area of data acquisition. Dowell has many different business lines that deal with the pumping of wellbore fluids (Cementing, Fracturing, Acidizing, etc.). All of these services monitor and record the same type of standard analog and frequency signals. Thus, all existing data acquisition systems can only acquire these standard signals (4-20 milliamp, 0-5 volt, and 12 volt frequency signals). The coiled tubing service has many specialized sensors that do not produce these standard outputs. The most vital sensor used in coiled tubing applications is the depth encoder. The depth encoder is used to measure the length of tubing in the wellbore. The depth encoder outputs a quadrature signal. A quadrature signal is a series of frequency signals that can be decoded to determine the speed and direction that the tubing is moving. Direction indicates that the tubing is being inserted in the wellbore or being extracted from the wellbore.

The second group of sensors that makes coiled tubing data acquisition differ from other standard pumping services is the requirement to monitor downhole sensors. Downhole sensors can be connected to the end of the coiled tubing and are used to transmit real-time data back to the surface monitoring equipment through a cable (wire-line) inside the tubing itself. These signals must be conditioned once they are received at the surface. Dowell's existing data acquisition systems can only directly acquire standard analog (4-20 milliamp, 0-5 volt), frequency, and serial type signals. Most downhole sensors currently require some type of signal conditioning device to condition the signals before being transmitted to the main acquisition system. These stand-alone conditioning devices usually require an additional human interface to calibrate the downhole sensors (pressure, temperature). These devices, along with their associated human interfaces, add unnecessary complexity to the daily system setup. Also, the number of electronic devices inside the Coiled Tubing Unit control cabin become impossible to manage.

There are many additional reasons to justify the need for a new data acquisition system, but the need to monitor quadrature sensors and the ability to monitor downhole sensors directly sets the coiled tubing data acquisition requirements apart from the existing Dowell data acquisition systems.

CHAPTER 2

MARKETING SURVEY

At the outset of this project, the Marketing group conducted a survey of the field operations, and the development of a new data acquisition system was initiated in response to the following market conditions [1]:

- 80% of coiled tubing jobs are not fully recorded.
- 65% of coiled tubing jobs are performed by service supervisors.
- CoilLIFE and Reel Databases are updated manually.
- Coiled Tubing Unit operator's cabins are 'cluttered' with excessive electronic devices.

Eighty percent of coiled tubing jobs are not fully recorded

It is a company standard that all well treatment service 'jobs' are to be fully recorded. The term 'job' is the generic name given to the process of completing the particular service from start to finish. There are many factors that contribute to the fact that not all coiled tubing jobs are being recorded. 'Fully recorded' indicates that all pertinent parameters are monitored and recorded, so that the data may be reproduced at a later time. Engineering has tried to make modifications to existing data acquisition systems to meet the new requirements of the coiled tubing business line. The required signal conditioning devices and new software releases have made the previous systems theoretically operational.

However, the system is too complex for the average coiled tubing end user and the data recording is often ignored. With the need for the data acquisition device itself and all of the signal conditioning/display devices, most Dowell field locations find the systems far too expensive and refuse to purchase the required equipment to record jobs properly.

Another significant reason that 80% of the jobs are not recorded is that most coiled tubing jobs are very 'low-tech'. 'Low-tech' refers to the fact that from the Dowell operator's and the customer's (Shell, Chevron, etc.) point of view, the job is very simple and the self contained mechanical and hydraulic indicators are sufficient to monitor the progress of the coiled tubing service. To a certain extent they are correct, but when something out of the ordinary or something catastrophic occurs, there is no record of the job. Catastrophic failures include breaking (parting) the tubing into two sections or losing control of the tubing reel so that tubing unspools out of control into the wellbore. When either of these two phenomenon occur, an investigating team of experts is usually contacted to perform a complete analysis of the failure. Without the pertinent job data recorded, the team is limited in the types of failure analysis that can be performed. Properly recorded data can significantly aid in the failure analysis.

Finally, the 'low-tech' job must be recorded so that the CoilLIFE reel database can be updated. The CoilLIFE process is explained in detail in the CoilLIFE section, but it will be shown that this data must be recorded, even for the simplest of coiled tubing operations.

Sixty-five percent of coiled tubing jobs are performed by service supervisors

What makes this statement relevant to the implementation of a new data acquisition is that the marketing personnel are trying to make a point about the computer literacy of the people performing our coiled tubing services. A service supervisor is the individual who is responsible for the day to day activities at the wellsite, and who has little or no computer training. The implementation of any new data acquisition system must consider the computer skills of its end users (65% of the end users have little or no computer training).

CoilLIFE and Reel databases are updated manually

CoilLIFE is a Dowell proprietary software program that calculates the fatigue and 'life' remaining in the particular tubing. 'Life' refers to the amount of usage remaining in a reel of tubing before it weakens and can part or separate. Referring to Figure 1, it can be seen that each time the tubing is run in the well, it is bent over the top of the goose-neck. Each time the tubing is bent over the goose-neck it fatigues. The amount of fatigue is based on the amount of tubing in the wellbore, the weight or tension pulling on the tubing, and the amount of fluid pressure inside the tubing. After each job, the operator enters certain job parameters into the CoilLIFE software package and the amount of 'life' is recalculated. This information is used to determine if the tubing reel can be used on another job(s). After a tubing reel's life has reached a specified minimum life, the reel is retired. Today during coiled tubing jobs, the operator manually records the pertinent data each time the tubing passes over the goose-neck. Upon returning to the office, the data is entered into the CoilLIFE program and the new 'life' is calculated. The process, as just explained does work in certain situations where the job had a fairly short record time and the number of passes over the goose-neck were minimal. The problem arises when the number of manual entries becomes too large to maintain easily. There are two main reasons why this might occur: 1) numerous trips over the goose-neck or 2) the unit could be installed offshore where the operator may perform 6-10 jobs before returning to the office to perform the CoilLIFE calculations.

Each location may have anywhere from 2-10 reels of tubing and associating a reel's 'life' remaining with the wrong physical reel could be disastrous. The field users feel that with today's technology this process should be automated and incorporated into the new data acquisition system.

Coiled tubing operator cabins are 'cluttered' with excessive electronic devices

In order to keep up with changing demands of the coiled tubing business line, new sensors/devices have been designed to meet these demands. Unfortunately, these new devices do not easily interface with existing data acquisition systems without the need for some type of signal conditioning device or individual human interface. For a complex coiled tubing job (maximum number of sensors/devices), as many as twelve individual electronic components can be present inside the operator's control cabin. With the current system, the operator has to program four individual keyboards/displays properly to configure the system, not to mention the individual cables required to connect all of the sensors/devices. A new data acquisition system should be able to reduce drastically the number of devices/human interfaces (keyboards/displays) required to monitor and record a coiled tubing job.

CHAPTER III

PREVIOUS WORK

Portable Automatic Cement Recorder

Dowell introduced it's first electronic data acquisition device the Portable Automatic Cement Recorder (PACR), in 1984. The PACR is a simple portable data acquisition system that monitors and records standard analog and frequency signals at the wellsite. The PACR was based on the eight-bit microprocessor and was equipped with a self contained printer and magnetic tape recording device. The PACR worked quite well for the standard pumping services, but there was no mechanism to store sensor configuration information therefore, the system had to be configured on a daily basis. This constant reprogramming (normally with end users with little computer skills) of the system lead to frequent setup errors and data recording problems. The system was originally released to record simple fluid pumping services, but over the years the system was upgraded in an attempt to handle other business lines. The upgraded system worked reasonably well for other pumping services except coiled tubing. The additional cost of the signal conditioning devices, the clutter in the operator's control cabin and the other reasons explained in the marketing section explain why the PACR was deemed not acceptable to record coiled tubing services.

Portable Acquisition Computer

The Portable Acquisition Computer (PAC) was introduced in 1993. The PAC is a ruggidized, portable 'front-end' data acquisition unit designed to perform data acquisition (similar to the PACR) for all business lines. Once again the coiled tubing business line needs were not directly addressed and the same issues were still present that made the original PACR system unacceptable.

The PAC has one major area of advancement over the PACR, serial communications. The PAC has several serial communication protocols; standard RS-232 communications and a RS-485 protocol (BITBUS). A significant feature that the PAC offers is the ability to network several pieces of equipment. This BITBUS protocol was designed so that several pieces of equipment could be connected to the PAC and upon scanning the network, the PAC would recognize and communicate with all devices on the network automatically. The communication scheme was implemented on the Siemens SAB525 under a Dowell proprietary protocol. An internal staff of engineers from all the various engineering groups defined the protocol. All future equipment was to be designed to comply with this protocol, thus any future piece of equipment that would be developed would be able to communicate directly with the PAC without any software modifications required to the PAC. To date, there are a total of four different devices that communicate over this protocol. None of the four devices are actually used in the coiled tubing business line, but they have shown the effectiveness and robustness of device networking.

CHAPTER IV

PROPOSED SOLUTION

The Coiled Tubing Sensor Interface (CTSI) is the proposed integration platform for data monitoring and recording for the coiled tubing business line. The CTSI provides the data acquisition capabilities and serves as the front-end system to transmit data to a back-end data analysis system. The CTSI (Figure 2) architecture is based on a network of smart sensors and the CTSI main enclosure. The sensor network provides the actual data acquisition, while the CTSI main enclosure provides system power, network power, job recording, and data storage capabilities of the system.

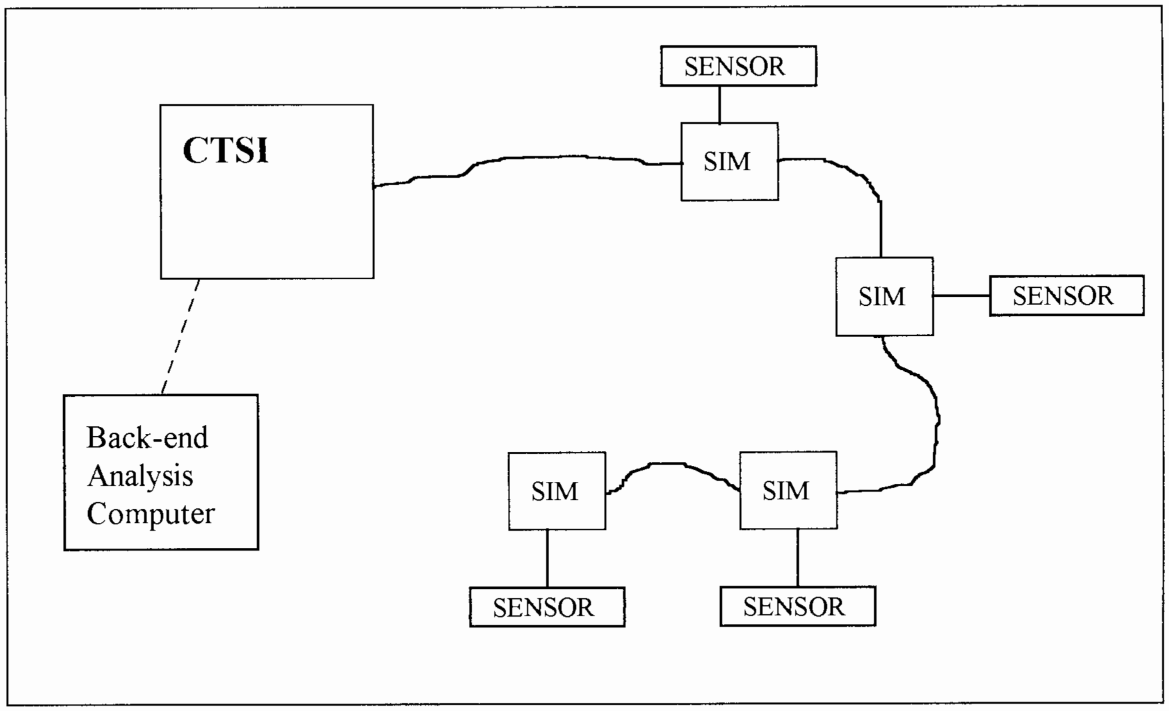

Figure 2: CTSI Block Diagram

Sensor Network

The Coiled Tubing Sensor Interface (CTSI) is based on a network of 'smart' sensors connected together to form a sensor bus. Standard analog and frequency sensors become 'smart' sensors when connected to a Sensor Interface Module (SIM). Each sensor is directly hardwired to a SIM by means of a short cable (1-6 feet). During installation qualified field personnel, normally an electronics technician, would configure the SIM for the particular sensor. Configuration information stored in the SIM would include: sensor name, sensor type (analog, frequency), minimum/maximum values, calibration information, sensor serial number, etc. This information is stored in the SIM between jobs and updated as needed (after periodic calibration). Each time the CTSI is powered on, the sensor bus is scanned and the connected sensors are automatically recognized by the system and no daily sensor configuration is required by the end user.

Sensors are connected together on location by a single cable originating at the CTSI, looping around and between equipment. This loop of four shielded wires distributes data to and from each sensor node on the first pair of 18 gauge wires, while powering (24 volts) the nodes on the second pair of wires.

Various sensor buses were formally investigated for suitability for coiled tubing operations [8]. It was determined from a cost, hardware availability, robustness, and simplicity point of view that the CAN (Controller Area Network) bus was the appropriate choice. Long term component availability, longevity, ruggedness (-40 to 125 degree C) are also features that CAN has by virtue of the fact that it is used in automotive applications.

CAN is a sensor network standard that was originally designed for use in the automotive industry to implement engine control and braking (ABS) systems. CAN has since found its way into factories and manufacturing plants as a method to reduce wiring complexity for process control. CAN is typically implemented as a communication port on a single microcontroller chip. These microcontrollers come in many flavors and are available from several suppliers such as Intel, Phillips, Siemens and Motorola. The microcontrollers can be easily interfaced to analog to digital converters and EEPROM memories.

Sensor Interface Module Hardware

The Sensor Interface Module (SIM) (Figure 3) design is based on the Motorola 68HC05X32 microcontroller. The microcontroller is a single chip implementation of a 68HC05 CPU, CAN controller, required ROM, RAM, EEPROM, and UART support. A 16-bit, four channel A/D converter is included to convert sensor analog values to digital values suitable to be transmitted over the network. Power for the sensors is provided by the sensor bus. The twenty four volt sensor bus power, supplied by the CTSI, is converted to the individual sensor's power requirements by voltage regulators located on the SIM circuit board.

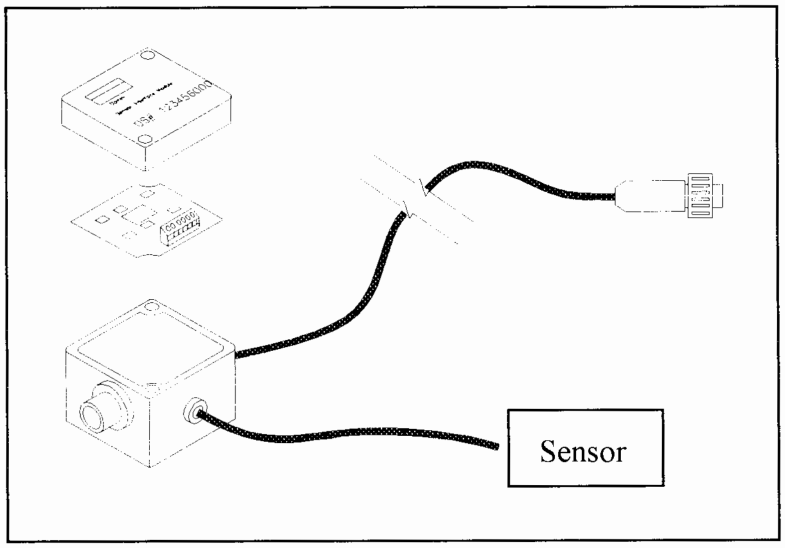

Figure 3: Sensor Interface Module (SIM)

The SIM module circuit board is totally encapsulated in epoxy with the exception of a corrosion resistant screw terminal strip. The circuit board is packaged in a small weatherproof box. The SIM module has three electrical connections. The first connection is a hardwired (cable gland) style connection to connect directly to sensors. The remaining electrical connections on the SIM module are used to connect all SIMs in a back-to-back configuration (daisy-chain) on the sensor bus.

There are three different implementations of the SIM module: the Standard SIM, the Injector SIM, and the Reel Database SIM. All of these SIMs are equipped with the Motorola 68HC05X32 microcontroller, but a few minor changes have been incorporated to address a few distinct features. The Standard SIM (Figure 4) is the most basic implementation of the SIM. The Standard SIM is designed to be interfaced with a single sensor. The SIM is equipped with the microcontroller, a four channel 16-bit analog-to-digital converter, a 24 to 12 volt DC/DC converter, and a 24 to 5 volt DC/DC converter. The SIM also contains the necessary terminal strip connections to make the required connections to the sensor and the network wires. Each Standard SIM is designed to acquire one of three signal types: current, voltage, or frequency signals. The current (0-20 or 4-20 mA) and voltage (0-5 VDC) signals are acquired through the analog-to-digital converter. The frequency signals are obtained by frequency counters provided by the microcontroller. The SIM is directly hardwired to one of the above mentioned sensor types through the cable gland connection supplied on the SIM. After the physical connections are completed, the SIM is programmed to indicate to which physical sensor it is connected (i.e., actual variable name that is written to SIM memory).

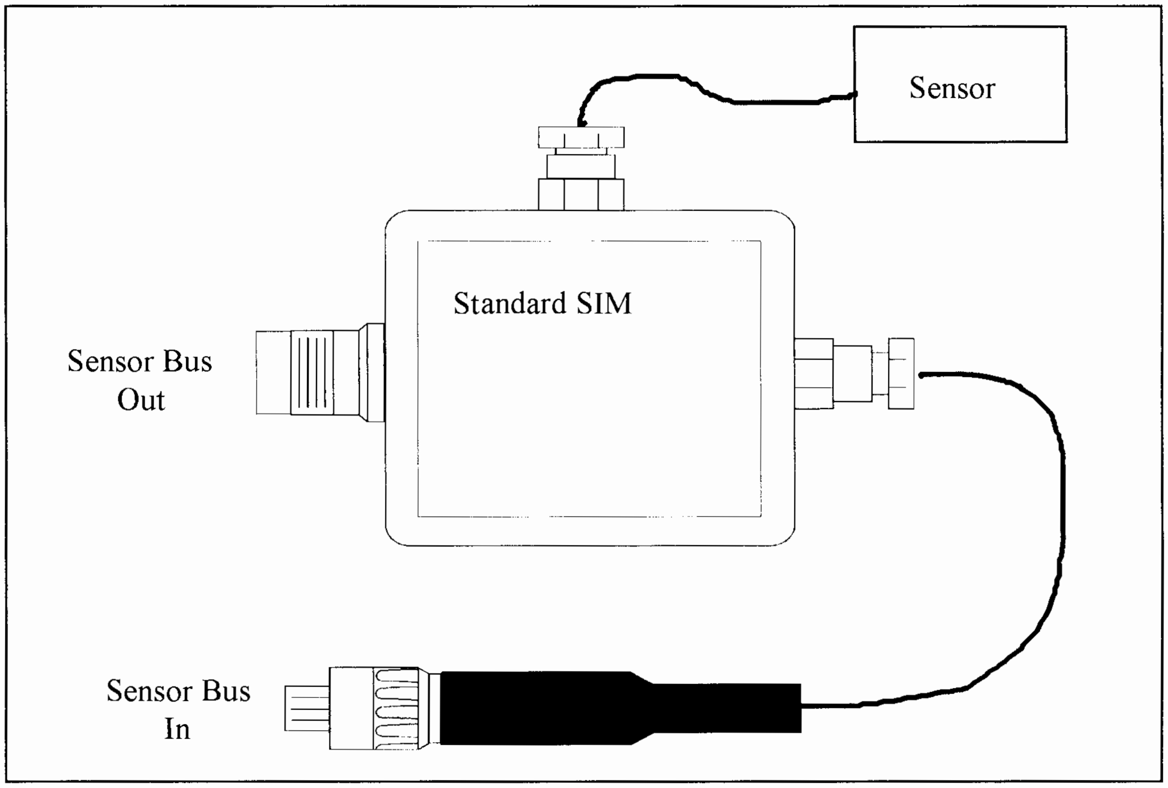

Figure 4. Standard SIM

The Injector SIM (Figure 5) is basically the same circuit board design as the Standard SIM, with two minor changes incorporated. The first modification is that a quadrature integrated circuit is added to allow the Injector SIM to acquire the depth encoder signal. The second modification is that the SIM is packaged in a larger enclosure so that additional sensors can be connected to the Injector SIM. An additional terminal strip was also added to handle the connections for the additional inputs. A total of four cable gland connections have been installed in the SIM. The SIM can acquire four separate signals. The end user now has the option to use four of seven possible signals: three current inputs, one voltage input, two frequency inputs, and the quadrature input. An input identifier has been placed on these inputs to allow the software applications and the end users to distinguish between the possible inputs:

Input 1	Current Channel 1
Input 2	Current Channel 2
Input 3	Current Channel 3
Input 4	Voltage Channel 1
Input 5	Frequency Channel 1
Input 6	Frequency Channel 2
Input 7	Quadrature

The third implementation of the SIM module is the Reel Database SIM (Figure 6). The Reel Database SIM is not designed to acquire sensor data. This SIM is designed to store the Reel Database, which is the information required to calculate the tubing's remaining 'life'. The SIM is equipped with the same Motorola CAN bus interface, but no circuitry for the sensor acquisition is installed. The SIM is equipped with a two mega byte FLASH EEPROM. This additional memory device will allow the CTSI to update the Reel

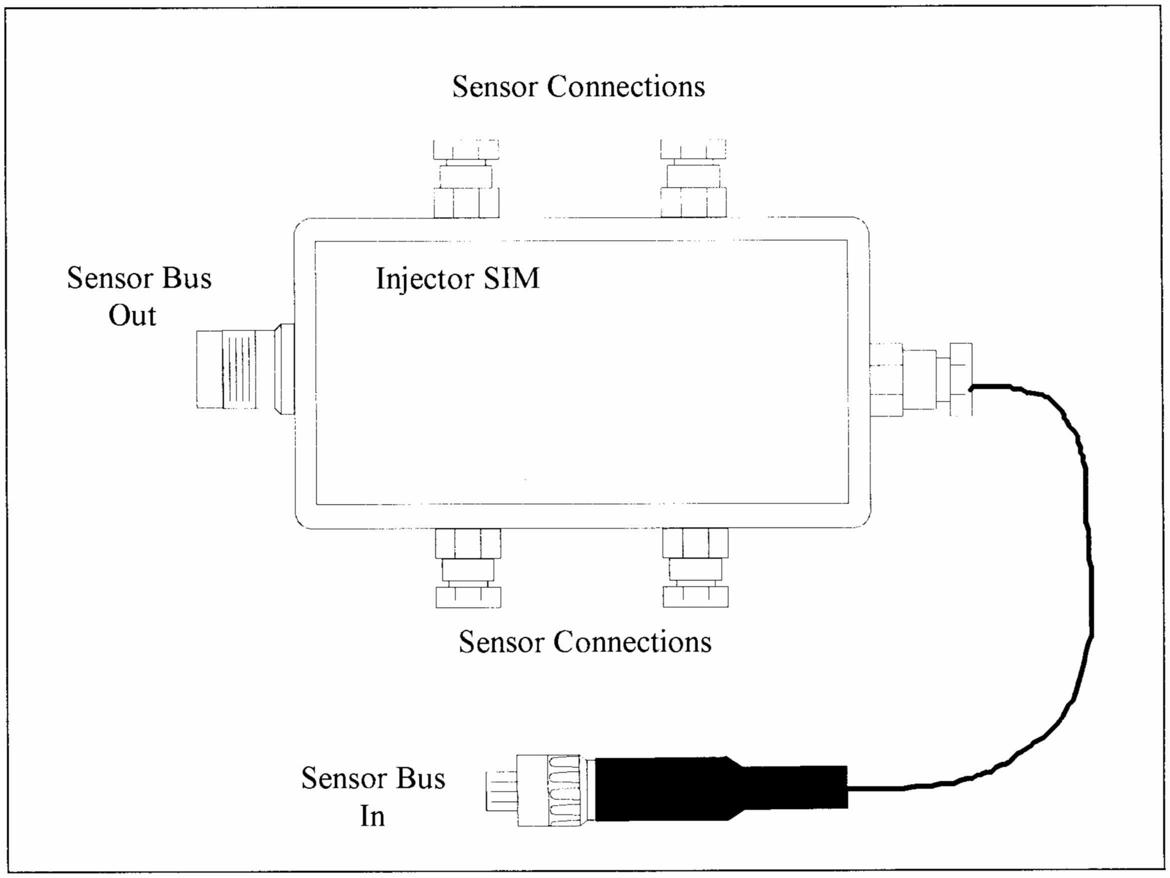

Figure 5. Injector SIM

Database automatically. At power on, the CTSI automatically retrieves the database from the Reel Database SIM. At shutdown, the database is automatically returned to the SIM. This process will eliminate the need for the end user to record the pertinent data manually at each pass over the goose-neck, as required with the existing acquisition systems.

The Reel Database SIM also provides support for a serial input. One of the required sensors on a coiled tubing unit supplies a serial output. This sensor is located in close proximity to the Reel Database SIM, so the serial support was added to the SIM.

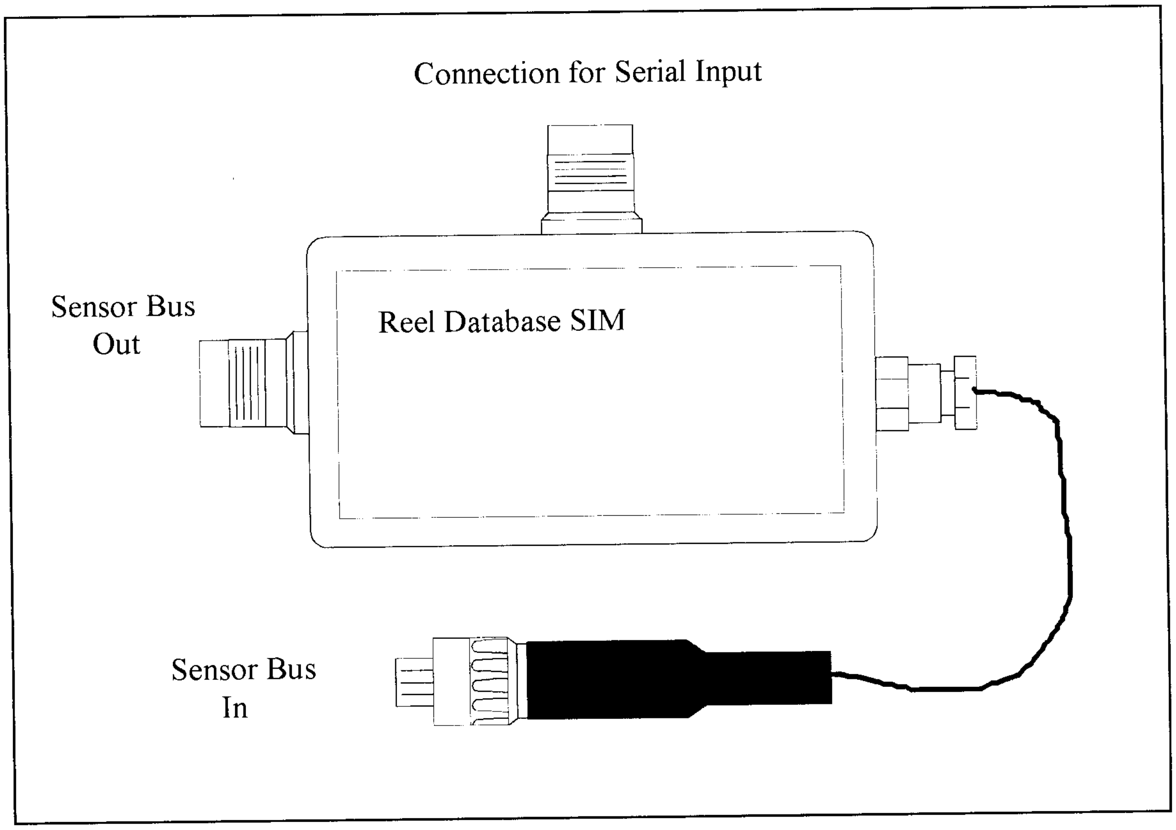

Figure 6. Reel Database SIM

Coiled Tubing Sensor Interface (CTSI) Main enclosure

The CTSI main enclosure (Figure 7) provides the 24 volt sensor bus power, job recording, data storage, and diagnostics capabilities in the system. The CTSI will have no keyboard or human interface. For day-to-day operations, the operator will not have to enter sensor configuration or job setup information. A standard VGA port will be available to the operator to connect an optional VGA display to view default sensor values during operation. The human interface will be provided via a back-end computer system that will be developed as a separate engineering effort. The back-end system will provide the human interface to program the SIMs, perform system diagnostics, file transfer, and other related functions. The CTSI communicates with the back-end system via TCP/IP over Ethernet.

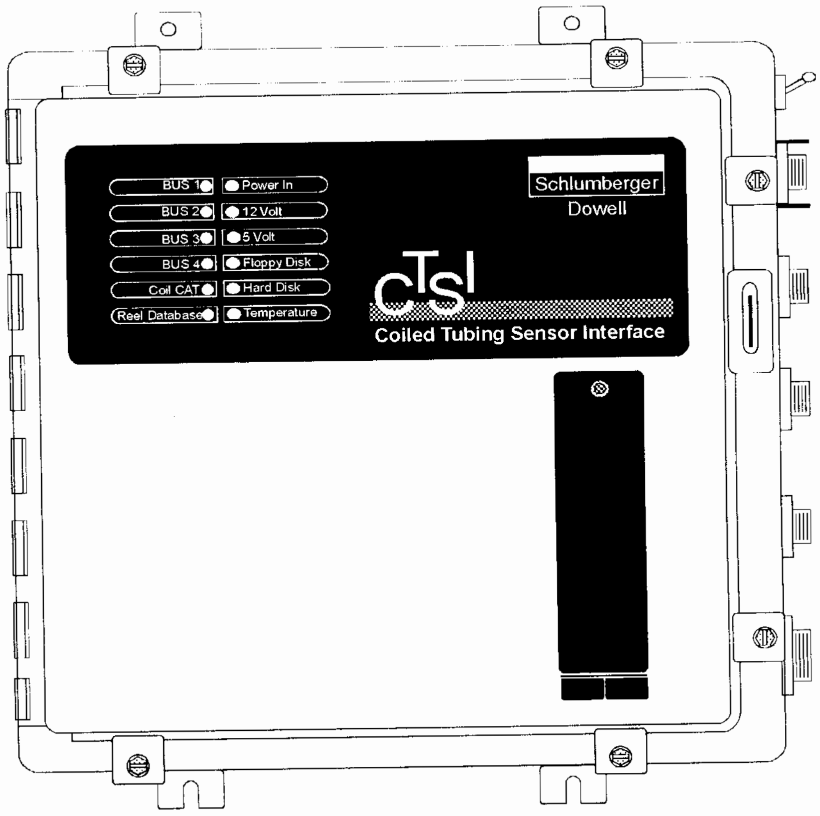

Figure 7: CTSI Main Enclosure (Front View)

The CTSI is a permanently installed, rugged, splash proof enclosure with a single board CPU, sensor power supplies, and the sensor bus interface. The CPU is based on PC compatible architecture with a passive backplane (ISA bus). Aside from the CPU card, the system contains a standard 3Com Ethernet card and the CAN bus driver card. Six slots in the backplane are available for future expansion. These additional slots could be used to install the data acquisition boards needed to acquire the downhole sensors directly, without the need for additional keyboard/display interfaces. The required interfaces for calibration and configuration will be implemented in the back-end analysis computer.

Program and data storage is provided by a 1.8 inch form factor 720 mega byte IBM harddisk. The system also has a standard 1.44 mega byte 3.5 inch floppy drive for extracting data files.

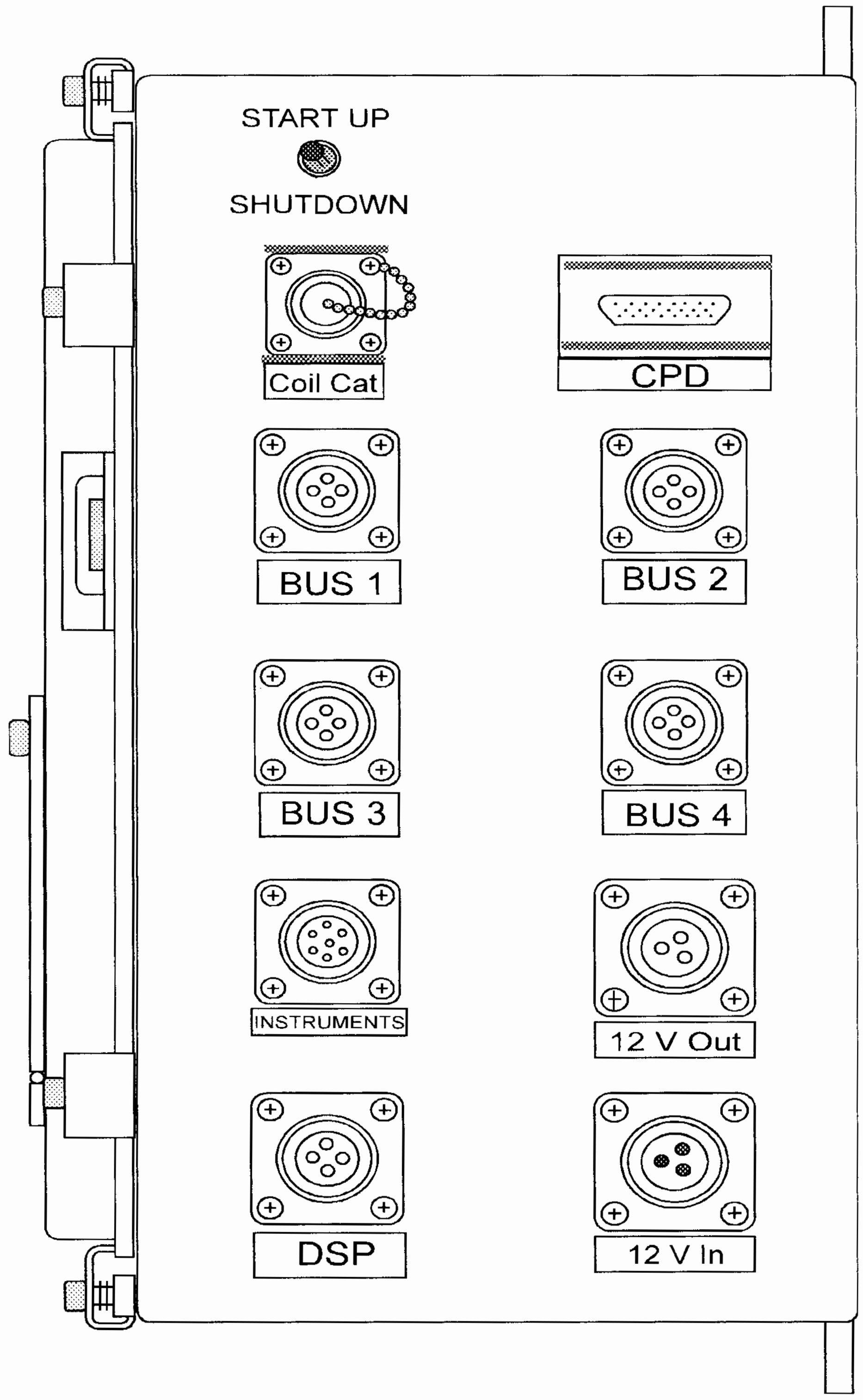

Figure 8. CTSI Main Enclosure (Side View)

CTSI Electrical Connections:

START UP - SHUTDOWN Switch: This switch activates and deactivates the system.

CoilCAT - This in the coax connector for the Ethernet connection to the back-end analysis computer (CoilCAT is the name used for the analysis computer).

BUS 1-BUS 4 - These are the connections to the sensor network. Currently the BUS 1 connection is the only bus that is activated. The next release of the CTSI will have all four busses active.

Instruments: This is the connection that connects the depth display panel and the weight dial gauge to the system.

12 V Out: This 12 volt DC output can be used to supply power to an external device. This output is intended to power an inverter, which can be used to supply power to the analysis laptop.

DSP: Currently not used. In a future release, this connector may be used to connect the DSP (Downhole Sensor Package) sub-surface instruments.

12 V In: Connection where the CTSI receives the required 12 volts from the coiled tubing unit.

CHAPTER V

SYSTEM SOFTWARE

The CTSI system software is a multi-tasking multi-threaded system implemented on the OS/2 Version 2.1 operating system. OS/2 was the operating system selected because of its ability to be implemented easily into the PC world as an 'embedded' mode of operation. 'Embedded' mode refers to the fact that no display or keyboard is required to activate the system (i.e., no system logon is required). Figure 9 is a block diagram of the system software architecture. At power on, the operating system automatically initiates four processes: 'CTSI Main', 'DataLibrarian', 'CPD' (Critical Parameter Display), and the 'Reel Database' process. After the system has completed the booting process, the processes are initiated by a simple command file (Startup.cmd).

CTSI Main

The CTSIMain process is the process that is responsible for all the data acquisition and data storage in the CTSI. This process accounts for approximately 90 percent of the total application software in the CTSI. This chapter will first discuss some of the software features that are common to all threads (semaphores, global variables), then each individual thread will de discussed. The more important threads will be covered in detail and the other general support threads will be discussed in broad terms.

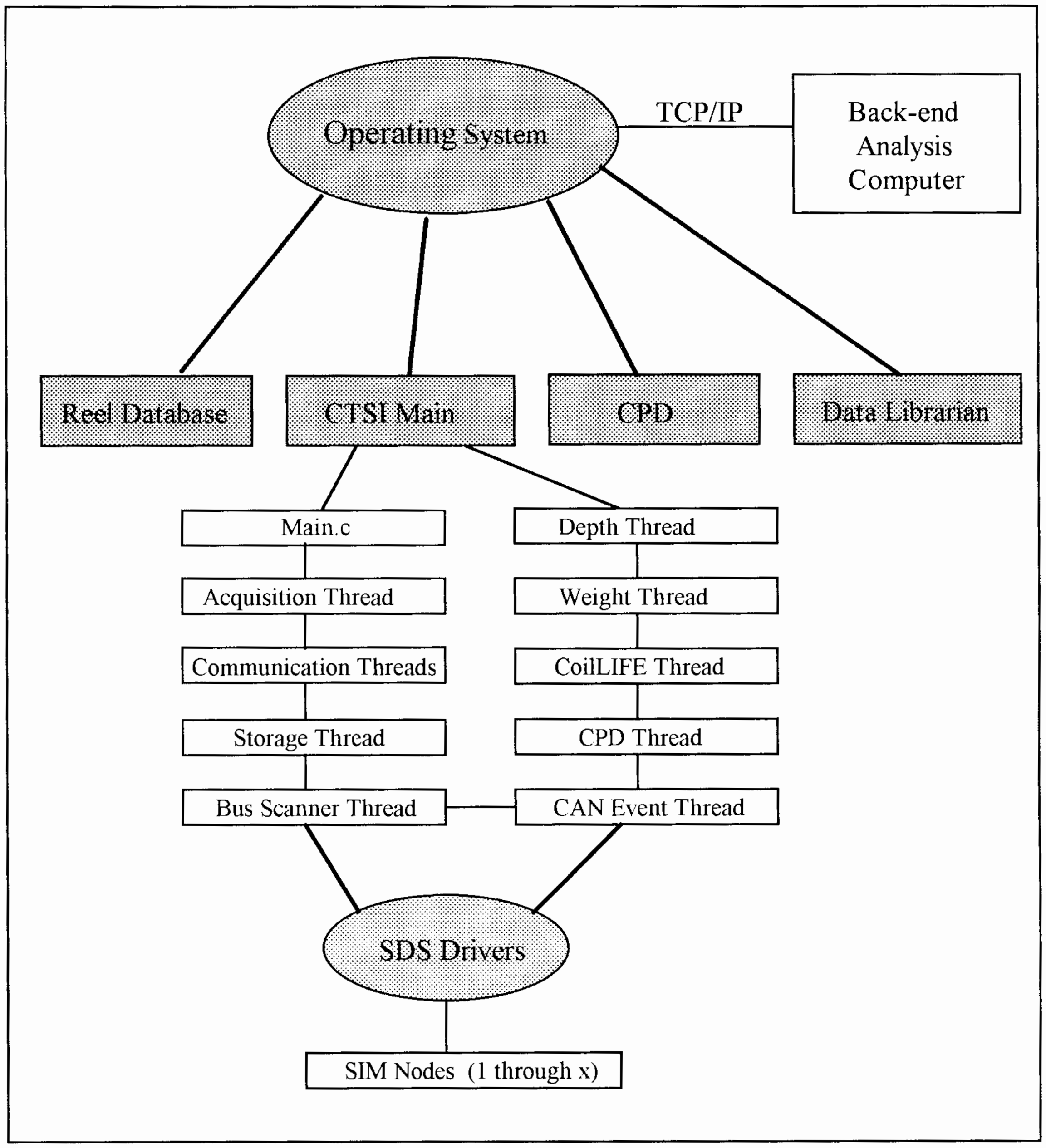

Figure 9. CTSI Software Architecture

<u>Global Variables</u>

All engineering data recorded in the CTSI is derived from the actual physical parameters (sensors values) being acquired. Each parameter being measured has a unique pre-defined name that is well known by the end users. These parameter names are defined in a global header file (#define) which includes an indexed list that can be used to refer to the parameters by their actual name or by their index value. A few examples of these defined variable names are:

```
#define Weight 5
#define Depth 14
#define Speed 15
#define WellHeadPressure 16
#define PumpRate 27
```

The application software can refer to these variables by name or by index. From a software point of view, the name Weight is equal to the number 5. For the first release of the CTSI there are a total of 70 pre-defined variable names. This variable list is shared between CTSI and the back-end analysis computer. This common list allows the two systems to coordinate the required data exchange. Each variable name has a global structure associated with it to allow the required operations to be performed on the variables.

```
struct GlobalVariable
{
UCHAR        caSIMName[],                  /* custom description    */
             cSensorName,                  /* sensor name Weight-5 */
             cSIMParameters,               /* bit mask for # inputs*/
             cSensorIndex,                 /* which 1 of 7 is it    */
             cElecType,                    /* 4-20mA or 0-20mA      */
             caCalibrationDate[],          /* last calibration      */
             caCalibrator[],               /* E.T. responsible      */
             caSerialNumber[],             /* MFG default value     */
             caUnitsLabel[],               /* PSI, SCFM, LBS, etc. */
             cAddress;                     /* SIM address < 125     */

LONG         lRawCounts;     /* raw counts from frequency a/d etc */

double       dUserValue;   /* float pt value in engineering units */

float        fOffsetUserValue,             /* offset value 4-20 mA */
             fZeroOffsetValue,             /* zero offset value    */
             fRawToUserUnits;              /* convert SI to User   */

HMTX         hmtxLock;                     /* semaphore data type  */

} sCTSIGlobalVariable;
```

Another key to the operation of the CTSIMain process is the notion of 'alive' variables. 'Alive' variables refers to the number of variables (sensors) actually being acquired by the CTSI. These 'alive' variables are a subset of the current list of 70 possible variables. There are normally between 10 and 15 'alive' variables being acquired on a standard coiled tubing job.

Semaphores

Since the CTSI is a multi-threaded design, there needs to be some mechanism to serialize access to the global variables. The CTSI design has implemented standard mutual exclusive semaphores to serialize this access to the global variables. Each time a particular process needs to manipulate a global variable, a semaphore for that variable must be locked to prevent the remaining threads from accessing the variable. Once the process has completed its manipulation, the semaphore must be released. This process was implemented using the standard OS/2 semaphore Application Program Interfaces (APIs): DosCreateMutexSem, DosRequestMutexSem, and DosReleaseMutexSem. One of the first functions the Main.c module performs is to create and initialize the mutual exclusive semaphores for the global variables.

```
/*
**  Create/initialize all mutual exclusive Semaphores
*/
for(iCount=0; iCount <= NumberOfCTSIVariables; iCount++)
{
    DosCreateMutexSem(NULL,              /* Semaphore name not needed */
                      &sCTSIVariables[iCount].hmtxLock,
                                         /* Place to return handle    */
                      0,                 /* No options                */
                      FALSE);            /* Initially unowned         */
}
```

The global variable structure has a HMTX data type. This data type is used to hold the handle for each semaphore. This 'for-loop' creates NumberOfCTSIVariable (100) semaphores, even though the first release of CTSI only has 70 of the 100 global variables defined.

Details of requesting and releasing the mutual exclusive semaphores are shown below. All threads that manipulate a global variable must follow this procedure or invalid data could be processed.

```
/*
**  Lock the Weight Semaphore, if not available wait indefinitely
*/
DosRequestMutexSem(sCTSIVariables[Weight].hmtxLock,SEM_INDEFINITE_WAIT);

      sCTSIVariables[Weight] = sCTSIVariables[RawTension].dUserValue -
            sCTSIVariables[Tare].dUserValue);
/*
**  Release the Weight Semaphore
*/
DosReleaseMutexSem(sCTSIVariables[Weight].hmtxLock);
```

Event semaphores have also been implemented to synchronize each thread's execution. The event semaphore is used to ensure that the thread executes at precise intervals. The event semaphores are also created in the CTSIMain.c module.

```
/*
**  Create the timer event semaphore, name given in the #define,
**  shared or globally visible and initially cleared or false
*/
apiReturnCode = DosCreateEventSem(AcquisitionThreadTimerSemaphore,
                                  &hteAcquisitionThread,
                                  DC_SEM_SHARED,
                                  FALSE);

apiReturnCode = DosStartTimer((ULONG)AcquisitionThreadPeriod,
                              (HSEM)hteAcquisitionThread,
                               &htimerAcquisitionThread);
```

The first time the particular thread is executed, the event semaphore must be opened. This is normally performed immediately after the variable declarations, before any 'while' loops are entered.

```
/*
** Open Storage Event semaphore
*/
apiReturnCode = DosOpenEventSem(AcquisitionThreadTimerSemaphore,
                                &hteAcquisitionThread);
```

The actual thread synchronization normally takes place at the end of the particular threads' 'while' loop. The event semaphore API is to delay a specific amount of time defined in the DosStartTimer API.

```
/*
**  Delay until Event semaphore cleared
*/
apiRetCode = DosWaitEventSem(hteAcquisitionThread, SEM_INDEFINITE_WAIT);
   if(apiRetCode == NO_ERROR)
   {
       DosResetEventSem (hteAcquisitionThread, &ulJunk);
   }
   else
   {
       fprintf(stderr,"AcquisitionThread.c: DosWaitEventSem() %ld\n",
                                                    apiReturnCode);
       DosExit(EXIT_THREAD, AcquisitionExit+1);
   }
```

This API will cause the function to delay (block) until AcquisitionThreadPeriod seconds (defined as one half second) has expired. The timer is reset by executing the DosResetEventSem API.

Main:

The Main.c module is responsible for initializing all of the global variables, creating all of the required semaphores, and creating the sockets required to communicate with the back-end analysis computer. This module also creates and initiates the remaining threads. After all of the required initialization is completed, the module enters a continuos loop. Each execution of the continuous loop checks to see if each individual thread is running. If a thread is found not to be running, the module restarts the thread.

```
Main()
{
        declarations: thread ID'd, timer handles for event semaphores

        create all mutual exclusive semaphores
        create all event semaphores

        initialize all sockets (create, bind, and listen)

        create all threads

        while(TRUE)
        {
                check each thread to see if it is running
                        if thread not running restart
                delay two seconds
        }
}
```

The OS/2 APIs _beginthread and DosSetPriority are used to create each thread. The following code sample from Main.c shows the AcquisitionThread being created. This procedure is repeated for each individual thread.

```
tidAcquisitionThread = _beginthread(AcquisitionThread,
                                    NULL,
                                    ThreadStackSize,
                                    (PVOID) 0);
    if (tidAcquisitionThread)
    {
      DosSetPriority(PRTYS_THREAD,
                     PRTYC_REGULAR,
                     AcquisitionThreadPriority,
                     tidAcquisitionThread);
    }
```

The _beginthread API is passed the actual thread source file name and the requested thread stack size. The ThreadStackSize for all threads in the CTSIMain process are set to 62K.

The PRTYS_THREAD argument in the DosSetPriority API indicates that only the priority of this thread is being set.

The second argument, PRTYC_REGULAR, sets the class of the priority. There are four possible choices for the priority class: *time critical*, *server*, *regular*, and *idle*. The *time critical* class has the highest priority. This class is used for threads or processes that have to be processed at precise intervals. A few of the CAN acquisition drivers are configured with this priority. The *regular* class is the most common class. Most user programs will run in the regular class. Most of the threads in CTSIMain are configured to run in the regular class. The *server* class has higher priority than the regular class. This class is used for applications that may need to run one level higher than the user applications (regular class). None of the threads or processes in the CTSI are configured with the server class. The *idle* class has the lowest priority. This class is used for non-critical processes that typically sleep most of the time. Some electronic mail programs use a daemon process that wakes during low CPU usage to check for incoming mail messages. Neither the server class nor the idle class have been implemented in the CTSI.

The AcquisitionThreadPriority argument sets the actual priority level of this thread. The priority level can be set form -31 to 31. The majority of the threads in the CTSIMain process are set to zero.

The final argument, tidAcquisitionThread, is simply the thread ID number. The thread ID is returned by the _beginthread call.

AcquisitionThread:

The AcquisitionThread is the module that is responsible for converting the raw data acquired by the low-level SDS drivers to engineering data that is meaningful to the end user. Aside from the data conversion, this thread also builds the initial global variable list.

The creating of the initial global variable list is the first function executed by the AcquisitionThread. This global variable list is what is refereed to as the 'alive' variables. At the time of the thread's first execution, the BusScannerThread has already built an array with the SIM addresses, of the SIMs that were responding during the bus scanning procedure. After determining the addresses of the SIMs that are responding, the AcquistionThread can access the SIMs memory contents to determine how many and which variables (sensors) are currently connected to the sensor bus.

SIMs can be configured with a combination of the seven possible inputs, as described in Chapter IV. A Standard SIM has one input and an Injector SIM has a maximum of four inputs. A Standard SIM with only one input would only have one global variable associated with it. Whereas the Injector SIM could have as many as four global variables associated with it. This indicates that the SIM would have to store enough sensor information to handle four variables. A second global variable was created to handle this multiple input SIM. This variable, sSIMStructure, is basically the same data type as the standard sCTSIVariables global, but some of the sensor input specific elements have been expanded to arrays to handle all the possible combination of inputs.

```
struct SIMStructure
{
UCHAR        caSIMName[],                /* custom description    */
             cSensorName[7],             /* sensor name           */
             cSIMParameters,             /* bit mask of inputs    */
             cSensorIndex[7],            /* which 1 of 7 is it    */
             cElecType,                  /* 4-20mA or 0-20mA      */
             caCalibrationDate[],        /* last calibration      */
             caCalibrator[],             /* E.T. responsible      */
             cSerialNumber[],            /* MFG default value     */
             caUnitsLabel[7][],          /* PSI, SCFM, LBS...     */
             cAddress;                   /* SIM address < 125     */

float        fOffsetUserValue[7],        /* offset value          */
             fZeroOffsetValue[7],        /* zero offset value     */
             fRawToUserUnits[7];         /* convert SI to User    */

HMTX         hmtxLock;                   /* semaphore data type   */
} sSIMStructure;
```

The key to manipulating the SIM memory contents is the cSensorIndex field. This index indicates which of the possible seven inputs the variable information relates to:

```
cSensorIndex[0]     Milliamp Input 1
            [1]     Milliamp Input 2
            [2]     Milliamp Input 3
            [3]     Voltage Input 1
            [4]     Frequency Input 1
            [5]     Frequency Input 2
            [6]     Quadrature Input
```

The sensor/variable information is read from the SIM memory and placed in the sSIMStructure variables according to the sensor input (sensor index). This information from the global structure is processed to determine the actual 'alive' variables found during the bus scanning process. For example, if sSIMStructure[2].cSensorName = 14 (or WellHeadPressure as known to the end user), all of the remaining information in the structure at index [2], would be associated with the WellHeadPressure variable.

The CTSI's implementation of the SDS model leaves 166 bytes of memory available to the application programmer. This remaining memory, located in the EEPROM area of the microcontroller, is used to store the sensor information specific to each individual SIM. This sensor information allows the CTSI to recognize automatically what sensors are connected to the sensor bus.

SDS Attribute	Bytes	SIM Structure value	
13 Catalog Listing	24	cSensorName	(7 bytes)
		caCalibrator	(16 bytes)
14 Vendor Name	12	caCalibrationDate	(9 bytes)
		cElecType	(1 bytes)
		cSIMParameters	(1 bytes)
15 Device Name	24	cUnitsLabel[0]	(4 bytes)
56 Tag Name	24	caSIMName	(24 bytes)
71 EEPROM R/W Area	82	fOffsetUserValue	(28 bytes)
		fRawToUserUnits	(28 bytes)
		caUnitsLabel	(24 bytes)

Table 1. Model of SIM Memory Contents

SDS attribute 13 Catalog Listing: contains the calibrators name, along with seven bytes that hold the possible sensor names. The sensor names are the indices that correspond to the global list of 70 variable names. The end user selects the actual strings when configuring a SIM, but the internal handling of the variable names is performed using indexes to the variable list. There is not enough memory available in the SIM to store all four variable name strings. (i.e., the number 16 is stored in the SIM memory instead of the string “WellHeadPressure”)

SDS attribute 14 Vendor Name: contains the calibration date information, electrical type and the SIM parameters bit masks. These two bit masks can be decoded to indicate information about all seven possible inputs. The cElecType bit mask is used to indicate if the first three analog inputs (milliamp) are 4-20 mA or 0-20 mA. An one indicates the input is 4-20 and a zero indicates 0-20. For example, if the cElecType value is five, 0101 in binary, this would indicate that input 1 and input 3 are 4-20 mA and input 2 is 0-20 mA or unused. The bit pattern is read from right to left (LSB is input 1 and MSB is input 7).

The cSIMParameter bit mask is used to determine which inputs are currently configured in the SIM. As with the cElecType bit mask, the binary values are read from right to left. For the cSIMParameter bit mask, an one indicates the input is being used and a zero means the inputs is not used. For example, a standard SIM would only have one bit set; an 1 would indicate input 1 is configured or a 16

(10000 in binary) would indicate that the frequency 1 input (input 5) is being used. An Injector SIM can have multiple inputs, so its bit mask could be 67 (1000011 in binary) indicating that the quadrature (input 7), milliamp 2 (input 2), and milliamp 1 (input 1) are configured in the SIM.

SDS Attribute 15 Device Name: only uses four of the 24 bytes available. The unit label for the first sensor index is stored at this location. The unit label describes the unit of measure (PSI, DEG F, etc.) for the sensor that is connected to input 1.

SDS Attribute 56 Tag Name: contains the 24 byte identifier of the SIM. This value is a custom description given to the SIM by the end user.

SDS Attribute 71: is a defined area of the SIM memory that has been set up as a free area available to the application program. This area contains the floating point numbers required to convert the raw data to engineering data. This area also contains the unit labels for the remaining six inputs. The fOffsetUserValue is used to indicate the offset required for a 4-20 mA input. The fRawToUserUnits is the value that the raw data is multiplied by to convert to engineering units.

After all the SIM memory information is processed and the 'alive' variables list has been created, the header buffer is generated. The header buffer is written to the harddrive and forwarded to the back-end analysis computer to inform both systems of what sensors/variables are being acquired. Aside from the actual variables being acquired, there are also other calculated variables that are recorded. These variables are also added to the 'alive' list. Every coiled tubing job has the depth, weight, and speed calculations added to the list of 'alive' variables. These values are derived from the actual measured raw values. These calculations can involve simple subtraction of initial offset values or more complex derivatives of another parameters.

After the SIM variables and the calculated variables have been added to the 'alive' list, the SIM data acquisition is activated and the continuous looping section of the AcquisitionThread in entered. Once in the 'while' loop section, the AcquisitionThread updates the global variables at a two hertz rate (twice a second).

```
AcquisitionThread()
{
      Open event semaphore(set timer to 1/2 second)

      /*
      **  Load initial SIM memory contents into global variables
      */
      for( Number of `alive' SIMs )
            Load SIM memory into sSIMStruct variables

      Update global variables

      Create global variables for calculated data
      (variables not acquired by SIMs, Calculations and totalizers)

      Determine SIM inputs being used and start SIM acquisition

      /*
      ** Begin `looping' part of acquisition
      */
      while(TRUE)
      {
            /*
            **  Obtain standard data acquired by SIM (sensor data)
            */
            for( Number of alive variables)
            {
              lock global variable semaphore
                  copy low-level raw data value to global.lRawValue
                  convert raw value to engineer value
              release semaphore
            }

            /*
            **  Perform totalizer calculations
            */
            lock global variable semaphore
                 calculate total flow rate
            release global variable semaphore


            /*
            **  Get raw total pulses for pump rate SIMs
            **   (no semaphores needed, this is the only thread
            **    that manipulates the fluid volume)
            */
            for all four possible pumps
                 copy low-level raw data value to global.lRawValue

            temp total = total of all four pumps
            lock fluid volume semaphore
                 calculate fluid volume
            release fluid volume semaphore

            execute delay to ensure thread runs at 2 hertz

      } end of while
}
```

CommunicationThreads:

The CommunicationThreads are the modules that are responsible for the communication between CTSI and the back-end analysis computer. The communications are performed via Berkeley (4.3BSD) sockets [11] over the Ethernet hardware layer. The following section describes the first of several system calls used to process the client-server communication scheme between CTSI and the analysis computer. The network communication is implemented as a connection-oriented protocol. A connection-oriented service requires that the CTSI and the analysis computer establish a logical connection with each other before the communication can take place. The logical connection is completed during the accept/connect system calls. Figure 10 shows a time line of a typical scenario that takes place to complete a connection-oriented transfer of data.

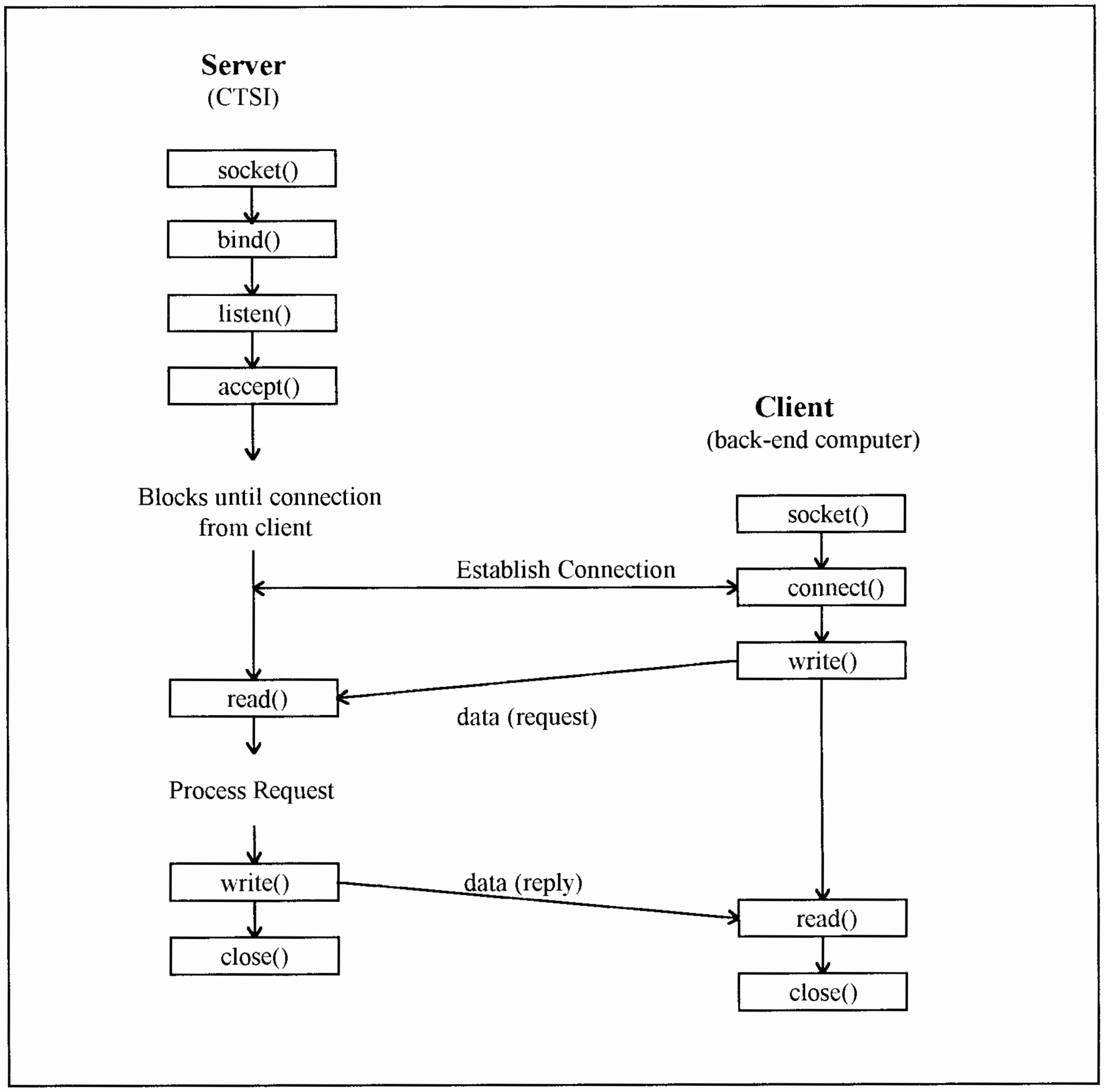

Figure 10. Socket System Calls

Before discussing the details of the system calls, the terms *connection* and *association* must be defined. A *connection* is used to define the communication link between two processes. The term *association* is used to define the 5-tuple that completely specifies the two processes that make up a connection:

{protocol, local-address, local-process, foreign-address, foreign-process} host (CTSI) and the foreign host (analysis computer). The *local-process* and *foreign-process* identify the specific processes on each system that are involved in the connection.

socket: The socket system call is the first call required to create the socket connection. The socket call specifies the *family* and *type* of socket.

```
int socket( int family, int type, int protocol)
```

The *family* can be one of four protocols: UNIX internal protocols, Internet protocols, Xerox NS protocols, or IMP link layer. All of the sockets implemented in the CTSI use the Internet protocols family. The socket *type* for all sockets in the CTSI are implemented as stream sockets. The *protocol* argument is usually set to zero for most user applications

The socket system call returns a integer value referred to as a socket descriptor, similar to a file descriptor. The socket system call specifies the first element in the 5-tuple, the protocol.

bind: The bind system call assigns a name to the socket created in the socket system call.

```
int bind( int socket-desc, struct sockaddr *serv-addr, int addr-len)
```

The socketaddr structure is defined in the `<sys/socket.h>` include file:

```
struct sockaddr {
      u_short     sa_family;  /* address family, AF_INET  */
      char        sa_data[14];      /* up to 14 bytes of the
                                         protocol-specific address*/
};
```

The bind system call enters the *local-addr* and the *local-process* elements of the association 5-tuple.

connect: To establish connection to the server connection, a client process connects to the socket descriptor returned by the socket system call.

```
    int connect(int socket-desc, struct sockaddr *serv-addr, int addr-len)
```

The *connect* is used by the client process to assign the local address. The client process does not perform the *bind* system call. The connect system call assigns four of the 5-tuple elements: *local-addr, local-process, foreign-addr, and foreign-process.*

listen: The listen system call is used by the server to indicate that it is willing to receive connections.

```
        int(int socket_desc, int backlog)
```

The backlog argument indicates how many connection requests can be queued by the system, while it waits for the server to execute the accept system call. For the CTSI server implementation, the backlog limit has been sent to two.

accept: The accept system call is executed directly after the listen system call. The accept system call is waiting for a client process to connect to the server process. The accept takes the first connection request on the queue and creates a new socket descriptor. If there are no connection requests pending, this call blocks the caller until one arrives.

```
        int accept(int socket_desc, struct sockaddr *peer, int *addr-len)
```

The *peer* and *addr-len* arguments are used to return the address of the client process.

send/recv: These system calls are similar to standard read and write operations. The arguments are simply the socket descriptor (descriptor created during the accept), a pointer to the character buffer, and the size of the character buffer.

```
        int send( int socket_desc, char *buffer, int num_bytes, 0)
```

close: This system call is used to close the socket when the processes have completed the required data transfer. The socket descriptor argument is the descriptor created during the accept system call.

```
int close( int socket_desc );
```

The following code segments contain actual code from the Main.c and the CommunicationThread.c modules. Main.c is where the socket is initiated with the socket, bind, and listen system calls. CommunicationThread.c performs the accept and the associated send system call that is used to pass the engineering data from the CTSI to the back-end analysis computer.

Code segments from Main.c

```
/*
**  Create and Bind Communication Socket
*/
if((iCommTcpSocket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    psock_errno("CTSIMAIN.c: Socket() call failed");
    DosExit (EXIT_PROCESS, MainExit+1);
}
```

The socket *family* is set to Internet and the *type* is set to streams sockets. The value iCommTcpSocket is the socket descriptor returned by the socket system call. If an error code is returned, an error message is logged and the process is terminated.

```
sCommTcpServer.sin_family = AF_INET;
sCommTcpServer.sin_port = htons(CommSocketPort);

if(bind(iCommTcpSocket, (struct sockaddr *)&sCommTcpServer,
sizeof(sCommTcpServer)) < 0)
{
    psock_errno("CTSIMAIN.c: Bind() call failed on CommPort");
    soclose(iCommTcpSocket);
    DosExit (EXIT_PROCESS, MainExit+2);
}
```

The bind system call takes the socket descriptor returned from the socket system call and assigns it to the socket port address specified by CommSocketPort, which is defined in a header file as port address 6000. The *hton*() function converts the pre-defined socket address to network byte ordering.

```
if(listen(iCommTcpSocket, SocketListenLimit) != 0)
{
    psock_errno("CTSIMAIN.c: Listen() call failed on
      CommSocketPort");
    soclose(iCommTcpSocket);
    DosExit (EXIT_PROCESS, MainExit+3);
}
```

The listen system call ensures that no more than two accepts are queued.

Code segments from CommunicationThread.c

```
if((iConnected = accept(iCommTcpSocket, NULL, NULL)) == -1)
{
      SocketErrorBeep(50);
      psock_errno("CommunicationThread.c: accept() call failed");
      DosExit (EXIT_THREAD, CommunicationExit+5);
}
```

```
if(send(iConnected, cCTSIDataBuf, strlen(cCTSIDatatBuf), 0) < 0)
      psock_errno("CommunicationThread.c: Send() call failed!");
```

The accept system call creates a new local socket descriptor. This descriptor is used to execute the send system call.

```
soclose(iConnected);
```

The soclose system call closes the local socket descriptor.

There are four individual modules that communicate with the back-end analysis computer: CommunicationThread.c, HeaderThread.c PutSIMStructureThread.c, and SendRawDataToCoilCATThread.c. Each module uses a unique socket port address of 6000, 7000, 8000, and 9000 respectfully.

CommunicationThread.c: This module transmits the engineering sensor data to the analysis computer at a two hertz rate. Once the socket is opened, the function enters a ‘do-while’ loop, transmitting the global data buffer containing the latest sensor data. Using the standard mutual exclusive semaphores, the global data is copied into a local buffer before being transmitted to the analysis computer.

```
/*
** Lock Sensor data buffer while updating
*/
DosRequestMutexSem(hmtxBufferLock, SEM_INDEFINITE_WAIT);

        strncpy(cBufferOut, cCoilCATBuffer,
                strlen(cCoilCATBuffer));
        memset(cCoilCATBuffer, '\0', CommBufferSize);

/* Release Sensor data buffer semaphore */
DosReleaseMutexSem(hmtxBufferLock);
```

The socket remains opened and transmitting data at a two hertz rate until the communication link is discontinued. The loss of connection is detected by the send system call return codes. Once communication is lost, the ‘do-while’ loop is terminated, the socket is closed, and the CommunicationThread thread is terminated. Main.c will restart the CommunicationThread during the next acquisition cycle. The CommunicationThread will then pend on the accept command, waiting for communication from the analysis to be continued.

```
CommunicationThread()
{
        accept()
        send( initial global data information )
        if( send error )
                close(socket)
                exit

        do
        {
                lock global data buffer
                copy global data to local buffer
                release global data buffer

                send( local data buffer to analysis computer )
                if( send error )
                break out of `do-while'
            delay( half a second )
        } while(TRUE)
        close(socket)
}
```

HeaderThread.c: The HeaderThread is the module that informs the back-end analysis computer of which parameters are being acquired by CTSI. This information, referred to as the header buffer, is a collection of ASCII strings that can be easily decoded to determine what variables are being acquired by the CTSI. The header buffer also contains important sensor configuration data such as, offset values, the conversion factor to convert from raw data to engineering data, calibration date, serial number of the SIM, etc.

Example of a typical header:

```
%26:02:96:12:37:44 Ver1.0B,ENP0 17 3
SIM_Name: Injector_SIM Injector_SIM Injector_SIM WHP_SIM Cir_SIM
NoSIM NoSIM NoSIM NoSIM NoSIM
BIT_mask: 67 67 67 1 1 0 0 0 0
SensorIndex: 0 1 6 0 1 1 0 255 255 255 255
ElecType: 3 3 3 1 1  0 0 0 0
SensorName: 2 16 6 18 17 14 8 15 5
CalDate: JAN-11-96 JAN-11-96 JAN-11-96 FEB-23-96 MAR-01-96 NoSIM
NoSIM NoSIM NoSIM
ET: Bill Eddie Bill Walter Mitch NoSIM NoSIM NoSIM NoSIM
SerialNumber: 17664 17664 17664 17633 17470 0 0 0 0
Address: 99 99 99 119 120 255 255 255 255
Units: LBF PSI FT PSI PSI FT FT F/MN LBF
Offset: -120000.00 -3750.00 0 0 0 0 0 0 0
Conversionfactor: 3.0518 0.286107 0.00208333 0.02796 0.01944 0 0 0 0
```

The module initially blocks on the accept system call. When the connection is made from the analysis computer, the global header buffer semaphore is locked and copied into the local header buffer. This local header buffer is then sent to the analysis computer via the send system call. After the send command is executed the socket is closed and the thread is terminated. The Main.c module will then restart the thread and the thread will again pend on the accept system call until the analysis computer requests another header.

```
HeaderThread()
{
      accept()

      lock semaphore for global header buffer
           copy global buffer to local header buffer
      release semaphore

      send( header buffer )
            if( send error )
                  log error

      close socket
}
```

PutSIMStructureThread.c:

This module receives the updated SIM configuration information from the back-end analysis computer and writes the updated information to the particular SIM's memory. The updated SIM information is received as a string. The string is parsed to determine which parameters have been modified. Only modified parameters are written to the SIM to reduce unnecessary bus traffic. The modified parameters are then written to the SIM memory and a the new header buffer is generated and written to the harddisk. The new header buffer is also forwarded to the back-end analysis computer. The analysis computer verifies that the SIM information received from the CTSI corresponds to the SIM changes that were just submitted. This process ensures that all processes involved in updating a SIM executed as expected: CTSI received correct information, CTSI wrote/read the correct information to/from the SIM, and the Header buffer was correctly configured and forwarded to the analysis computer.

Example of updated SIM information received from the analysis computer:

118 6 Injector_SIM 67 3 6 MAR-11-96 JohnDoe FT 0.0 0.002833

118 - SIM Address
6 - Sensor Index (Quadrature input)
Injector_SIM - User defined SIM Name
67 - Bit mask for inputs configured (1000011)
3 - Electrical type bit mask (011 in binary, inputs 1 & 2 are 4-20mA)
6 - Variable name (from global list RawDepth = 6)
MAR-11-96 - calibration date
JohnDoe - Person responsible for calibration
FT - Variable's unit label
0.0 - Offset value
0.002833 - Conversion factor for raw to engineering data

The PutSIMStructureThread socket is created and closed after each exchange of data from the analysis computer. The thread initially blocks on the accept system call until a connection is received from the analysis computer. The updated SIM information is retrieved via the recv system call. Once the information is forwarded, the socket is closed and the thread is terminated. Within two seconds, Main.c will re-create the thread and the thread will return to the blocking state until a request is received from the analysis computer.

```
PutSIMStructureThread()
{
      accept()
      recv()

      if( number of characters received > 0 )
            parse input string
            if ( 11 parameters received )
                  read current SIM parameters
                  determine which parameters need updated
                  write updated information to SIM

      close socket
}
```

<u>SendRawDataToCoilCAT.c</u>:

The SendRawDataToCoilCAT module is multi-purpose function that is used to handle various data requests from the back-end analysis computer. The first and most obvious function of this module is to send raw sensor data to the analysis computer. Raw data is the data representation of the measured parameter just after it is obtained from the physical sensor (i.e., the output of the analog-to-digital converter is termed raw). The analysis computer only monitors and records engineering data, but in certain situations, such as calibration or troubleshooting, the end user may need to examine the raw data before it is converted to engineering data.

This module is also used to handle other information requests from the analysis computer: zeroing sensors, entering weight parameters, or entering depth parameters. Once the accept system call is acknowledged, the module enters a 'while' loop and performs a recv every two seconds. The module continues to process the requests from the analysis computer until no request is detected during the recv operation (i.e., no bytes received). Once the recv system call determines that the connection is lost, the socket is closed and the thread is terminated. The only request that is normally repeated every two seconds is the raw data request. The remaining requests are only used to request single operations from the CTSI (i.e., zero a parameter or enter offset value).

The operation requested by the analysis computer is determined by the first character received (request key). The remaining parameters received are particular to the requested operation.

```
SendRawDataToCoilCAT
{
      accept
      while(TRUE)
      {
            bytes received = recv()

            if( bytes received <= 0 )
                  break;
            else
                  switch( first character recv )
                  {
                    case 0:                /* raw data request */
                        Load buffer with raw data for parameter
                              requested by second character in
                              the recv buffer
                        send buffer to analysis computer

                    case 1:            /* Zero standard Sensor */
                        CTSI to perform zero offset for the
                              parameter requested by second char
                              is recv buffer

                    case 2:            /* Zero Weight request */
                        CTSI to perform zero offset of weight
                        value

                    case 3:            /* Enter Depth Offsets */
                        CTSI to update global depth parameters
                        with values received to recv  operation
```

```
                case 4:           /* Receive Depth Tag:*/
                    CTSI to update global depth information
                    with values received from analysis
                    computer

                default:
              }

          Delay 2 seconds
      }
      close socket
      exit thread
  }
```

BusScannerThread:

At power-on, the BusScannerThread determines which SIMs are connected to the network. The thread issues a SDS NOP command to each possible SIM address (addresses 1-125). This command causes all 'alive' SIMs to respond. The responses are collected and the addresses on these SIMs are placed in the `iAliveSIMs` global array. This array is used by the AcquisitionThread to scan the 'alive' SIMs for the 'alive' variables.

The BusScannerThread is also responsible for controlling the front panel lights.

StorageThread:

The StorageThread.c module is one of the most vital modules in the CTSI system software. StorageThread is responsible for creating the data file, writing sensor data to the harddrive, and building the global header buffer to be placed in the data file. The header buffer is a small paragraph of information that defines the tabular data that is to follow in the data file. Any time there is a change the sensors being acquired, a new header is placed in the data file to re-define the tabular data to follow. The header buffer is also sent to the analysis computer upon request to the inform the analysis computer of which parameters are currently being acquired by the CTSI.

Example of data written to harddrive:

```
%26:02:96:12:37:44 Ver1.0B 9 3
SIM_Name: Injector_SIM Injector_SIM Injector_SIM WHP_SIM Cir_SIM NoSIM
NoSIM NoSIM NoSIM
BIT_mask: 67 67 67 1 1 0 0 0 0
SensorIndex: 0 1 6  1 1  255 255 255 255
ElecType: 3 3 3 1 1  0 0 0 0
SensorName: 2 16 6 18 17 14 8 15 5
CalDate: JAN-11-96 JAN-11-96 JAN-11-96 FEB-23-96 MAR-01-96 NoSIM NoSIM
NoSIM NoSIM
ET: Bill Eddie Bill Walter Mitch NoSIM NoSIM NoSIM NoSIM
SerialNumber: 17664 17664 17664 17633 17472  0 0 0 0
Address: 99 99 99 119 120 255 255 255 255
Units: LBF PSI FT PSI PSI FT FT F/MN LBF
Offset: -120000.00 -3750.00 0 0 0 0 0 0 0
Conversionfactor: 3.0518 0.286107 0.00208333 0.02796 0.01944 0 0 0 0
ChangeOfValue: 65432 65535 65432 65432 65432 0 0 0 0
1 5670 -3749 0.0 4.48 0.00 -90 0.0 5670 0.00
2 5685 -3748 0.0 4.48 0.00 -90 0.0 5670 0.00
3 5652 -3749 0.0 4.48 0.00 -90 0.0 5670 0.00
4 5661 -3749 0.0 4.44 0.00 -90 0.0 5670 0.00
5 5652 -3749 0.0 4.44 0.00 -90 0.0 5670 0.00
```

The data file is created upon initial execution of the StorageThread. The file name for the data file is derived by executing the OS/2 DosGetDateTime API. This command returns a structure with all of the pertinent date information. The file name is then derived by concatenating this date information into a string. (i.e., MAR2296.1). The initial file extension is set to one. The OS/2 DosOpen API is then executed to see if the file can be opened. If the file cannot be opened because it already exists, the file extension is incremented and the DosOpen API is retried. This procedure is repeated until the DosOpen API executes successfully. Each time the StorageThread is started, at power up or when the thread being restarted, a new file with an incremented file extension is created. The file extension can increase up to 999. The only instance that would cause the thread to be activated that many times would be some type of hardware failure (repeated power cycling) and the file extension would not be incremented after 999 is reached. In normal operation, there would not be more than two or three files created on any particular day. During installation, troubleshooting, or training, the number of files created could be as many as 20 to 30.

The header buffer is generated by looping through all of the 'alive' variables and concatenating all of the common elements together to build a paragraph of ASCII strings. The first line of the header buffer contains the date and time, followed by the current CTSI software version. The software version is followed by the number of 'alive' variables and the number of SIMs present. The remaining strings are generated by looping through all elements of the 'alive' variables as shown below:

```
strcat(cBuffer, "SIM_Name: ");
for(iCnt=0; iCnt<iNumberOfAliveCTSIVariables; iCnt++)
{
  strncat(cBuf,sCTSIVariables[iAliveCTSIVariables[iCnt]].cSIMName, 24);
  strcat(cBuf, " ");
}
  strcat(cBuf, "\r\n");


strcat(cBuffer, "BIT_mask ");
for(iCnt=0; iCnt<iNumberOfAliveCTSIVariables; iCnt++)
```

```
{
   sprintf(cTemp, " %d",
            sCTSIVariables[iAliveCTSIVariables[iCnt]].cSIMParameters);
   strcat(cBuffer, cTemp);
 }
   strcat(cBuffer, "\r\n");
```

Each string starts with a seconds counter and ends with a line feed and carriage return. The seconds counter is used to keep track of time between headers. The seconds counter is used on each data entry instead of the complete date and time string to reduce wasted disk space. This procedure of cycling through the structure elements is continued until all elements of the global variable structure (sCTSIVariables) have been processed.

```
StorageThread.c
{
      open event semaphore ( 1 second )

      create data file()
      write initial Header to harddrive()

      while(TRUE)
      {
            if number of `alive' parameters has changed
                write new header to file

            for all `alive' variables
            {
                  lock semaphore
                      load engineering values into local array
                  release semaphore

            }

            convert local array of floating point numbers into
              string of ASCII characters

            place date & time at start of ASCII string
            if( message request from Depth panel)
                  concatenate to ASCII string

            /*
            **  This global buffer to read by the analysis
            **  computer at a 2 hertz rate
            */
            lock global analysis data buffer semaphore
                  copy ASCII data string to global data buffer
            release semaphore

            concatenate current data sting to local data buffer
            increment loop counter
            /* only write to harddisk every 30 seconds */
            if( loop counter = 30 )
```

```
                    write data buffer to disk
                    clear data buffer

        }  /* end of while(TRUE) */
}
```

DepthThread:

The DepthThread contains two source files: DepthThread.c and DepthPanelThread.c. The DepthThread.c is the module responsible for calculating the actual depth and speed engineering values. The depth value refers to the amount of tubing (in feet) in the wellbore. The speed value indicates how fast the tubing is being place in or being removed from the wellbore. The final depth engineering value is simply the raw depth value obtained by the encoder sensor added to the various offset values.

The speed value is calculated from consecutive depth readings. The speed (in feet per minute) is calculated by taking the difference of the last two depth values and multiplying by 120 to obtain feet per minute. The difference in the depth readings (feet per 0.5 second) is multiplied by 120 because the DepthThread executes at a two hertz rate (feet per 0.5 second * 120 = feet per minute).

DepthPanelThread.c is the module the operates the depth display panel. The depth panel is a small serial display device for displaying the depth and speed values. Aside from displaying the depth and speed parameters, the depth panel provides a simple human interface for performing simple operations. These operations include: zeroing certain analog sensors, entering depth and weight offsets. The depth panel can also display all of the ‘alive’ variables and their values.

WeigthThread:

The main functionality of the WeightThread is to calculate the final weight value. The weight value refers to the amount of tension that is being applied at the injector head. The weight sensor (load cell) is located at the bottom of the injector head and measures the forces being applied to the injector head. The tubing weight is a positive value when the tubing is being pulled out of the wellbore and can be negative as the tubing is being pushed into the wellbore. Excessive tension being pulled on the tubing can cause the tubing to part or break. Forcing the tubing into the wellbore at excessive tension can cause the tubing to buckle or become stuck in the wellbore, where it can not be easily removed.

The weight value is calculated by subtracting the injector head weight and initial weight offset (Tare) from the actual raw weight reading measured by the physical load cell sensor.

CPDThread:

The CPDThread.c module is the function that sends the updated engineering data to the CPDProcess. The CPDProcess is a separate process that displays nine default sensor values on an optional VGA display. Since the CPDProcess is actually a separate process, a mechanism must be implemented to allow the CPDThread (located in the CTSIMain Process) to forward the updated engineering data to the CPDProcess. A pipe is the mechanism that has been implemented to allow the CPDThread to pass data to the CPDProcess. A pipe is simply a system-controlled buffer used to relay information between processes. A pipe can be described a serial line between two pieces of equipment. The one device transmits the data over the serial line, and the second device receives the data. For a pipe, one process transmits the data through the pipe, and the second process receives the data from the pipe. The pipe used in the CPDThread is a one-way pipe. Data is simply forwarded to the other process and no return data is required.

The actual pipe is created in the CPDProcess using the OS/2 DosCreateNPipe API. The local CPDThread must connect to the pipe before sending data to the pipe. The pipe is opened using the OS/2 DosOpen API. This is the same API that is used to open a data file.

```
/*
** Connect to the Named Pipe created in the CPDProcess
*/
      DosOpen("\\PIPE\\CPD",                      /* pipe name      */
              &hPipe,                             /* pipe handle    */
              &ulAction,                          /* return code    */
              0L,                                 /* File size = 0  */
              0L,                                 /* File attribute */
              FILE_OPEN,                          /* Action taken   */
              OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE |
                                      OPEN_FLAGS_FAIL_ON_ERROR,
              NULL);                /* action pipe can perform */
```

The data is forwarded through the pipe using the standard DosWrite API. Any time an error is detected in the DosOpen or DosWrite APIs, the pipe is closed and the thread is terminated. CTSIMain will then re-start the thread and the pipe will be re-connected.

```
CPDThread()
{
      declarations
      open event semaphore
      open pipe

      while
      {
            for all variables on CPD
              lock semaphore
                  get latest engineering value
              release semaphore

            send data to CDPProcess
            delay with event semaphore
      }
}
```

<u>CoilLIFEThread:</u>

The CoilLIFEThread is the module that builds the update file used by the Reel Database process. This thread writes the depth and pressure values to a file each time there is a large enough change in two consecutive readings of the depth or the pressure. Pressure refers to the pressure of the fluid inside the coiled tubing (CirculatingPressure). This file is written to the Reel Database SIM when the system shutdown has been detected. The update file is only written to the SIM after the system determines that the analysis computer has not updated the database. The updating process is explained in detail in the Reel Database Process chapter.

CanEventThread:

The CanEventThread.c is the thread that receives all of the SDS event messages generated by the SIMs. The three possible events that could be generated by a SIM are a Diagnostics Error event (event 0), End of Timer event (event 3), or a COV event (event 6). At the present time, all Diagnostics events are being logged in a data file, but no real-time actions are being taken. The COV events are currently implemented in the SIM software, but due to a few earlier software defects, these values have been set to maximum values to reduce unnecessary bus traffic. Some of these earlier defects included incorrect values being placed at these locations. These incorrect values caused the SIM to try an respond more than 100 times a second which had negative affects on the available network bandwidth. The SIM software has since been updated not to respond more than 10 times a second. The COV functionality will be available in the next release of the SIM software.

The majority of the event messages generated by the SIMs are due to the End of Timer interrupts being processed. Each SIM automatically responds with a data packet each time the Cyclic Timer (attribute 10) expires. If a SIM has more than one input selected in the End of Timer list attribute, the SIM will respond with as many data packets as inputs selected. For example, an Injector SIM with four inputs configured in the End of Timer list attribute would respond with four separate data packets.

The CanEventThread looping interval varies from the majority of the threads in the CTSIMain process. At the end of each 'while' loop iteration, the thread checks to see if there are more event packets pending in the buffer. If there are more event packets pending in the input buffer, the thread continues to loop until all packets are received. If there are no packets pending, the thread delays 10 msec before continuing to check for new event packets.

SDSDriver:

The SDSDriver.c module is not an actual thread. The SDSDriver module provides the low-level APIs to read and write information to the SDS component model.

The low-level APIs provided in this module are:

ReadSDSAttribute
WriteSDSAttribute
WriteSDSAction
AnnounceSDSAttribute
AnnounceSDSAction
ReadSDSEvent

All of these functions are implemented in a similar manner. All of these functions follow the same process of events: lock the CAN bus semaphore, set the thread priority to time critical, write/read the requested value, reset the priority to regular class, release the semaphore, and return. The following section describes the WriteSDSAttribute function in detail.

```
WriteSDSAttribute(USHORT usCANBus, USHORT usAddress, USHORT usAttribute,
                                   USHORT usAttributeType, void *vData)
{
  /*
  ** Reserve the CAN Bus # in question
  */
   DosRequestMutexSem(hmtxCANBusLock[usCANBus],SEM_INDEFINITE_WAIT);

   DosSetPriority(PRTYS_THREAD, PRTYC_TIMECRITICAL, 0L, 0L);

       /*
       **  Call the actual SDS attribute read routine
       */
       err = WrSdsAtt(usCANBus, usAddress, usAttribute,
                                          usAttributeType, vData);

   DosSetPriority(PRTYS_THREAD, PRTYC_REGULAR, 0L, 0L);

  /*
  ** Release the CAN Bus # in question
  */
   DosReleaseMutexSem(hmtxCANBusLock[usCANBus]);

return(err);
}
```

The first function argument is the CAN bus number (usCANBus), which is always bus zero for the current release. Future releases of the CTSI will have up to four possible CAN buses. The second argument is the address of the device/SIM (1-125). The third attribute is the particular attribute being processed (i.e., Attribute 10, Cyclic Timer). The attribute type argument is passed the number of bytes in the attribute data element. The last argument is a pointer to the actual data element that is to be written to the attribute.

The CAN bus semaphore is locked to ensure that no other thread tries to access the bus while writing to the SIM. The priority is set to the time critical class to ensure that this action is completed in the shortest amount of time possible. Once the attribute has been written, the priority is returned to regular class and the semaphore is released.

Data Librarian

The Data Librarian process is a simple command file implemented using the script command language REXX. The REXX script language allows standard DOS calls to be easily implemented. The Data Librarian process has two main functions: 1) writing the job file to the floppy drive at shutdown 2) ensuring that the harddrive has ample storage space to record the next job. The following is an outline of the Data Librarian process:

```
/*
**  Copy job file to floppy drive
*/
Check status of floppy drive (DosOpen API)
      if( DosOpen successful )
            Delete all files on floppy
            Set floppy status OK
      if( Format not acceptable )
            Format floppy
            Set floppy status OK
      if( No floppy detected )
            Set floppy status NOT OK

      if( Set floppy status OK )
            Redirect DIR/OD command output to file
            Parse file to determine latest job file
            Determine number of files with same date
            Totalize size of files to copy
            if( File size to copy greater than 1.44 Meg )
                  Zip files
            Copy files to Floppy drive

/*
**  Clean up harddisk
*/
Output UNIX command DF to file
Parse file to determine disk usage
While( Disk usage greater than 75 percent )
      Output DIR/OD command output to file
      Parse file to determine oldest job file
      Output UNIX command DF to file
      Parse file to determine disk usage
```

At shutdown, if a valid floppy is detected, the Data Librarian process copies the current data file to the floppy drive. The status of the floppy drive is detected by executing the

OS/2 DosOpen API. Three possible return codes from the DosOpen API are processed: file opened successfully, unrecognized format, or no floppy present. If the file is opened successfully, all files on the floppy are deleted to ensure the job file will fit on the 1.44 mega byte floppy. If an unrecognized format is detected, the floppy is formatted. If no floppy is detected, the command file ignores the floppy update process.

Once all files on the floppy drive have been deleted, the Data Librarian determines which files to copy by executing the standard DOS command DIR/OD. This DOS command returns a directory of the job files directory in chronological order. The directory output is redirected to a file. The file is then parsed to determine the file name of the latest file created. The process must also check to see if multiple files were created during the particular job. These common files would have the same file name, but have different file extensions (i.e., FEB1596.1, FEB1596.2, etc.). After the number of files to copy has been determined, the system calculates the total number of bytes to copy by totaling the sizes of all of the files to copy. If the total bytes to copy is greater than 1.44 megabytes, the files are compressed. The files are compressed using the standard PKZIP executable. The files are then copied to the floppy drive.

The second function that the Data Librarian process performs is monitoring the harddisk. The Data Librarian process executes the UNIX DF (describe file system) command. This executable returns, along with other harddisk specific information, the amount of disk space currently being allocated. The output from the DF command is redirected to a file. This file is then parsed to determine the disk space remaining. If the disk space remaining is less than 75 percent, the oldest job files are deleted until 25 percent of free disk space is available. The CTSI is equipped with a 720 mega byte disk. It has been estimated that 100 jobs can be recorded before any jobs files are deleted. After the disk space issues are resolved, all temporary files that were created are deleted.

Reel Database

The Reel Database process is the process that is responsible for automatically retrieving and replacing the Reel Database from the Reel SIM. The Reel database contains the data that is used by the CoilLIFE program to calculate the 'life' remaining in a particular reel of coiled tubing. At power on, the Reel Database process retrieves the current database located in the Reel SIM. After retrieving the database, the database is placed in the specified directory and renamed DATABASE.PRE. The process also checks to determine if any update files exist on the Reel SIM. An update file is a file that is created during the job that records the depth and pressure readings each time there is a large enough change in consecutive readings of the depth or pressure values. These update files, along with the Reel database, are used by the back-end analysis computer to compute the remaining 'life' in the tubing. The update files are generating during each job, but are only written to the Reel SIM if the analysis computer is not being used during the job. If the analysis computer is being used, the Reel's 'life' can be calculated at the end of the job.

After CTSI has retrieved the DATABASE.PRE file, along with the present update files, the analysis computer can copy the files from the CTSI harddrive. The analysis computer can now re-compute the 'life' remaining from the original database and the update files, if any exist. After calculating the 'life', the analysis computer places the updated 'life' information back on the CTSI harddrive. At this time, the file is named DATABASE.PST, for post job database. This is an indication to CTSI that the database has been updated and should be placed in the Reel SIM at shutdown. The update file created during the job is deleted and not forwarded to the Reel SIM. The update file is only forwarded to the Reel SIM, if the database is found still to have the .PRE extension.

```
ReelDataBase()
{
      /*
      ** at CTSI Power-on:
      */
      retrieve Reel database:
      rename database DATABASE.PRE
      retrieve any Update files if present

      /* CTSI is creating the update file during operation */

      /*
      ** At Shutdown:
      */
      check status of DATABASE file
      if DATABASE.PST exists
            write DATABASE.PST to Reel SIM
      else if DATABASE.PRE still present
            write Update file to Reel SIM and increment extension

}
```

Critical Parameter Display ('CPD') Process

The CPD process is used to display a default set of parameters on the optional VGA display. The term *critical parameters* is generic name given to the parameters that the end users considers vital to a operation of a coiled tubing job. The nine parameters that are displayed on the VGA display are: Depth, Weight, Speed, Wellhead Pressure, Pump1Rate, Pump2Rate, Total Fluid Volume, Circulating Pressure, and Tubing average diameter.

CHAPTER VI

SENSOR INTERFACE MODULE SOFTWARE

The acquisition software embedded in the SIM is based on the standard Smart Distributed Systems (SDS) protocol as defined by Honeywell [9]. This thesis is not intended to provide a complete understanding of the SDS protocol, but a few key items that are vital to the SIM's data acquisition will discussed. The SDS protocol is a application layer service that is designed to be used with the CAN data link layer. The key to implementing the SDS protocol is defining the component model. The component model represents the networks visible structure and behavior. The component model refers to the specific SDS implementation for each individual design. Each SDS implementation may have a different data structure (attributes), but the component model as a whole must comply with the SDS standards and guidelines.

The SDS component model is made up of attributes, actions, and events. An attribute is the location of the data elements in the logical device (SIM). The collection of attributes can be seen as the data structure of the SIM's data elements. Actions and events comprise the SDS behavior of the SIM. Actions are referenced to direct the SDS model to initiate actions (i.e., change address). A event is used to report the occurrence of events within the SIM (i.e., interrupt). The following tables list the attributes, actions, and events incorporated in the SIM's implementation of the SDS protocol [2].

Attribute ID	Item	Variable
0	I/O Type	
1	Baud Rate	Unsigned 8
2	Device Type	String 3
3	Vendor Identification	
4	Device Address	
6	Un/Solicited Mode	
7	Software Version	
8	Diagnostic Error Counter	
9	Bus Diagnostic Flags	
10	Cyclic Timer	
11	Serial Number	Unsigned 32
12	Date Code	String 24
13	Catalog Listing	String 24
14	Vendor Name	String 12
15	Device Name	String 24
18	Input (bits 1..2)	
19	Quadrature Count	Unsigned 32
20	Analog 0 Input	Unsigned 16
21	Analog 1 Input	Unsigned 16
22	Analog 2 Input	Unsigned 16
23	Analog 3 Input	Unsigned 16
24	Analog 4 Input	Unsigned 8
34	Output (bits 1..16)	
55	Manufacturing Codes	Bits 8
56	Tag Name	String 24
60	NO/NC	Unsigned 8
61	COV Mask, Digital	Unsigned 8
62	COV Mask, Quadrature	Unsigned 16
63	COV Mask, Analog 0	Unsigned 16
64	COV Mask, Analog 1	Unsigned 16
65	COV Mask, Analog 2	Unsigned 16
66	COV Mask, Analog 3	Unsigned 16
67	COV Mask, Analog 4	Unsigned 16
68	End of Timer List	Unsigned 8
69	Acknowledge Preset Timer	Unsigned 8
70	EEPROM Address Pointer	Unsigned 8
71	EPROM Data	Unsigned 8
72	Digital Input 1 Rate	Unsigned 16
73	Digital Input 2 Rate	Unsigned 16
74	Digital Input 1 Total	Unsigned 16
75	Digital Input 2 Total	Unsigned 16

Table 2. SIM Component Model Attributes

Action ID	Action Name
0	NO-OP
1	Change of Address
6	Enroll

Table 3. SIM Component Model Actions

Event ID	Event Name
0	Diagnostics Error
3	End of Timer
6	COV Event

Table 4. SIM Component Model Events

This section discusses some of the attributes that are vital to the SIMs operations. Attributes 0-15 are very standard attributes that appear in most SDS component models. The remaining attributes are customized for each component model implementation. The attributes that contain the user (sensor) specific configuration information are discussed in the AcquisitionThread section. The attributes that are essential to the SIM's data acquisition are attributes 6, 10, 18-24, 72, 73, and attribute 68. Attributes 19-24, 72, and 73 are the locations that contain the actual raw data values acquired by the corresponding acquisition component. The quadrature raw data value is located at attribute 19. The outputs from the A/D converter are located in attributes 20-23. The two frequency values are located in attributes 72 and 73.

Attribute 68 (End of Timer List) is a bit mask that indicates to the system which inputs are to respond when the Cyclic Timer (attribute 10) expires. In other words, Attribute 68 indicates which inputs are currently configured in the SIM. The End of Timer List bit mask is defined as follows:

Bit:	7	6	5	4	3	2	1	0

Bit 0:	Attribute 18 (Digital Input)
Bit 1:	Attribute 19 (Quadrature Decoder)
Bit 2:	Attribute 20 (Analog Input 1)
Bit 3:	Attribute 21 (Analog Input 2)
Bit 4:	Attribute 22 (Analog Input 3)
Bit 5:	Attribute 23 (Analog Input 4)
Bit 6:	Attribute 72 (Frequency Input 1)
Bit 7:	Attribute 73 (Frequency Input 2)

After the required inputs have been selected (Attribute 68), the Cyclic Timer (Attribute 10) is set to respond automatically with the latest raw sensor data. The Cyclic Timer is the timer, specified in increments of ten milliseconds, that activates an interrupt for the SIM to respond with the latest raw data values for the inputs selected in the End of Timer List attribute. Most of the SIMs in the CTSI are set to respond every 500 msec.

After the Cyclic Timer is configured, the Un/Solicited Mode (attribute 6) is set to actually start the SIM's acquisition. The SIM's acquisition is initiated by writing a one to attribute 6. The acquisition can be halted by writing a zero to attribute 6.

The SIM's acquisition can also be generated by a Change Of Value (COV) event (interrupt). An COV interrupt is generated when a change in consecutive raw data values is larger than a defined value selected by the user. The COV value is entered by the user from the back-end analysis computer. This value is stored in attributes 60-66, according to the input selected. Any time the change in consecutive raw data values is larger than this user defined value, the SIM will respond with the latest acquired value. This COV feature can be used to save network bandwidth for sensors that rarely change values. The COV feature also aids in capturing data spikes that may fall between the normal 500 msec timer acquisition.

The actions that are currently being used in the SDS model are the NOP and Change of Address actions. The NOP command is issued by the BusScannerThread to determine the addresses of the SIMs that are present on the sensor bus. The Change of Address action is used to modify a SIMs address. This action is used when a new SIM is placed on the sensor bus. All new SIMs are received from the manufacturer at address one. After recognizing a SIM at address one, the CTSI re-assigns the SIM's address to a new address that is not currently being used.

The Enroll action is not currently implemented in the first release of the CTSI, but the Enroll functionality is one of the highest priorities for next software release. The Enroll action allows the CTSI to detect the existence of duplicate devices at any device address (i.e., SIMs that have the same address). This address conflict can be resolved by executing the Enroll action and a special option of the Change of Address action. The Enroll action requests each SIM (node) to respond with its address and serial number. If two nodes respond with the same address, the nodes serial numbers are used to distinguish between them. Then a special option of the Change Address action is used to modify one of the nodes address. The Change of Address action can use the four byte serial number as part of the data packet. If this argument is present in the data packet, then only the node having the same address and serial number will respond and change its address. All other nodes will ignore the change of address request.

CHAPTER VII

CTSI OPERATION

Now that all of the vital software and hardware designs have been discussed, this chapter will provide a simplified overview on how to operate the CTSI. The following operational procedures explain the steps necessary to monitor/record a standard coiled tubing job using the CTSI. The following operational procedure is what is often referred to as the "Switch ON-Switch OFF" mode of operation.

Steps to set up and record a standard coiled tubing job:

1. Connect all SIMs in a back-to-back (daisy-chain) configuration to the BUS 1 connection on the CTSI.
2. Place the Start Up/Shutdown switch in the Start Up position.
3. Wait for the system to complete the booting process.
4. Zero out the required weight and depth readings using the depth display panel.
5. Perform the coiled tubing job.
6. When the job is completed, place a floppy in the floppy drive.
7. Move the Start Up/Shutdown switch to the Shutdown position.
8. When the CTSI shuts off, remove the floppy.

1. Figure 11 shows a simplified diagram of a typical wiring configuration to connect all of the SIMs that are required to monitor/record a standard coiled tubing job. For most coiled tubing units, all of the SIMs, except the Pump Rate SIM are permanently mounted on the coiled tubing unit. The only sensor bus cable that would normally need to be connected, is the cable that connects the Treating Pressure SIM to the Pump Rate SIM (proximity

switch). The Pump Rate SIM is permanently connected to the pump truck and the proximity switch is hard wired to the SIM.

2. Placing the Start Up/Shutdown switch in the Start Up position performs the following operations: 1) applies power to the system 2) boots the computer 3) initiates the four software processes 4) scans the bus for SIMs 5) starts acquisition/recording 6) retrieves the Reel Database.

3. It takes approximately three minutes for the system to boot and scan the sensor network.

4. The operator can enter a zero offset for the weight and depth readings if necessary. These values are entered using the depth display panel numerical keypad.

5. The coiled tubing service is performed. The job could take anywhere from two hours to three days to complete.

6. Any time before the system is shutdown, a floppy can be placed in the floppy drive.

7. When the system is shut down, the CTSI first checks to see if a floppy is in the floppy drive. If a valid floppy is found, the system copies the ASCII file to the floppy drive. The

CTSI then places the necessary database or update file in the Reel Database SIM. After the Reel Database SIM is updated, the system power is turned off.

From the given operational procedure, it has been shown that any end user with little or no computer skills would be able to monitor and record any standard coiled tubing job.

Figure 12 shows a more detailed drawing of the CTSI/SIM wiring configuration. The Reel Database SIM, Injector SIM, Wellhead Pressure SIM, CTSI Main enclosure, Instruments Junction Box, and the Circulating Pressure SIM are normally permanently mounted on the coiled tubing unit. The only cable that the user has to configure on a daily basis is the cable that connects the Injector SIM to the Pump Rate SIM.

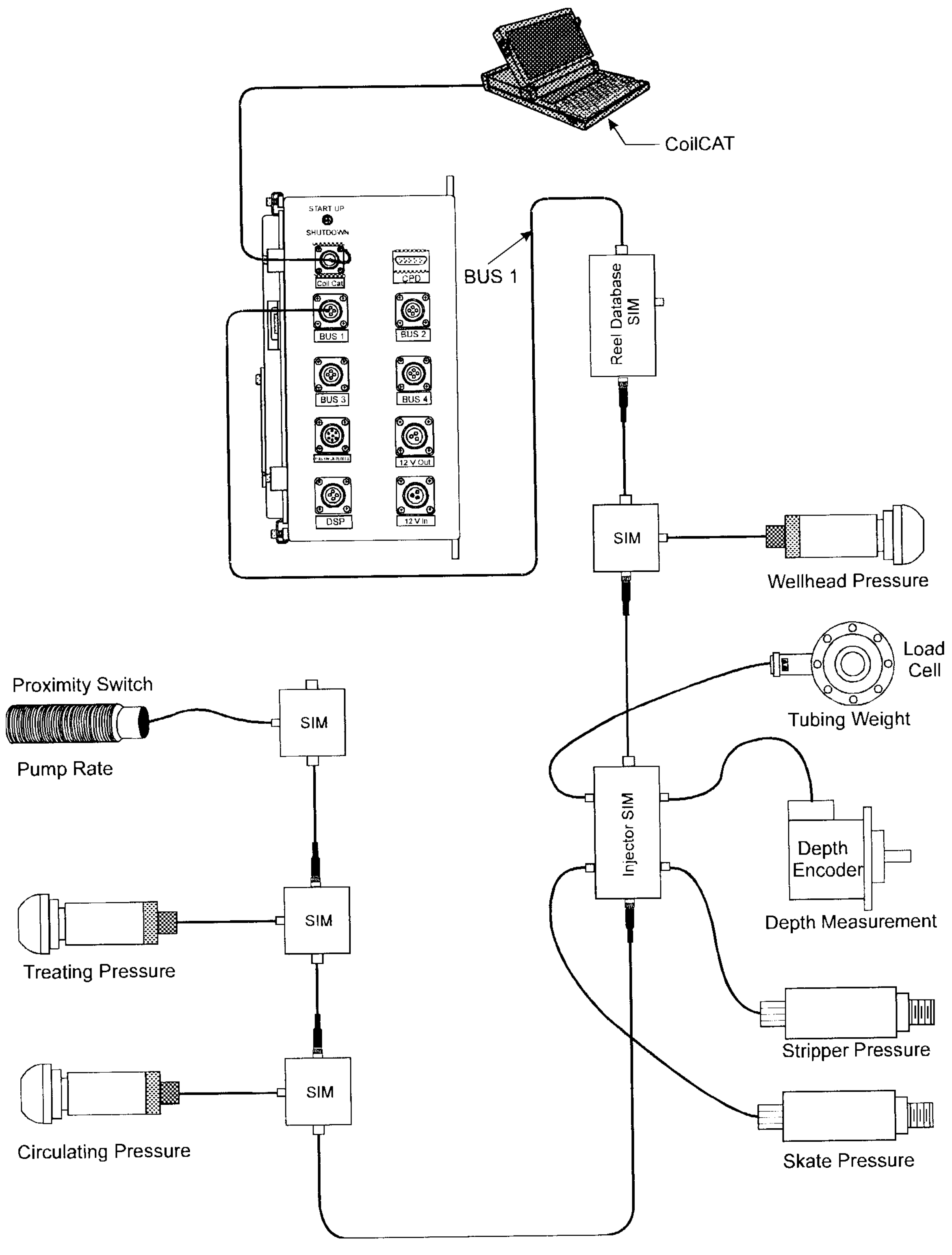

Figure 11. Simplified CTSI Wiring Configuration

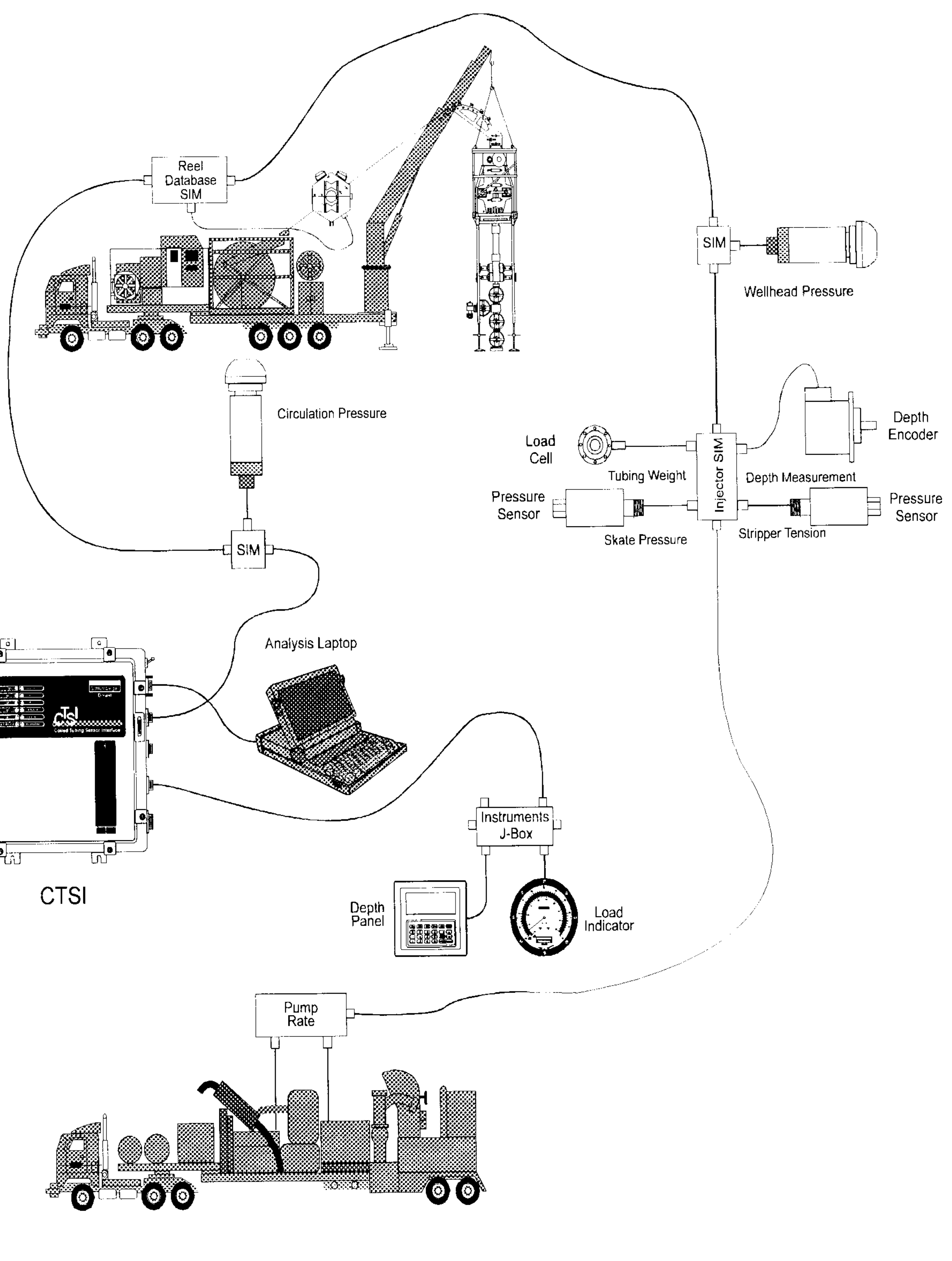

Figure 12. Typical CTSI Configuration

CHAPTER VIII

CONCLUSION

The CTSI has been designed to address all of the data recording problems currently being encountered in the coiled tubing business line. Aside from being able to acquire the depth (quadrature) and downhole sensors directly, the CTSI also addresses all of the concerns expressed in the marketing section. By design, the 'switch ON- switch OFF' scheme itself addresses most of the marketing concerns.

This single switch operation addresses the marketing concerns that 80% of the jobs are not fully recorded and 65% of the coiled tubing job are performed by service supervisors. There should not be any situation where the job is not recorded if the operator is capable of operating an on/off switch. The concern that CoilLIFE and Reel Databases are updated manually is addressed through the Reel Database SIM and ReelDatabase Process. The CoilLIFE Databases are now updated automatically after each job by simply shutting down the system. The concern that coiled tubing operator cabins are 'cluttered' with excessive electronic devices is also addressed with the CTSI implementation. The number of electronic devices in the operator's cabin should be reduced to a maximum of three devices: the CTSI, optional VGA monitor, and an optional back-end analysis laptop for experienced users.

The CTSI is currently in the beta testing phase. As of the May 10, 1996, there have been eight CTSI systems installed across Texas, California, and Colorado. As of the above date, there have been 127 successful jobs recorded with the CTSI. There have been a few hardware related failures, but there have been zero failures due to software related problems (system crashes or system lock-ups). A total of 150 jobs are to be recorded before the system is deemed commercial. At that time, which will be within the next year, there will be over 125 systems deployed through out the world.

REFERENCES

1. David Shand, *Coiled Tubing Sensor Interface Marketing Plan*, Houston, Texas, April 1995.

2. D.I.P. Inc, *SDS Sensor Input Module (SIM) Multiple Input/Multiple Output Interface Specification,* Revision 1.0, Moreno Valley, CA, 1995.

3. German Hallet, *OS/2 2.1 REXX Handbook: Basics, Applications and Tips*, Van Nostrand Reinhold, New York, NY, 1994

4. H.M. Deitel and M.S. Kogan, *The Design of OS/2,* Addison-Wesley, New York, NY, 1992

5. Herbert Schildt, *C: The Complete Reference*, Second Edition, Osbore McGraw-Hill, Berkeley, California, 1987

6. Jody Kelly, Craig Swearingen, Dawn Bezviner, Theodore Shrader, *OS/2 2.1 Application Programmers Guide*, Van Nostrand Reinhold, New York, NY, 1994.

7. Naba Barkakati, *The Waite Group's Microsoft C Bible*, Second Edition, SAMS, Carmel, Indiana, 1990.

8. S. Adnan, R. McBride, R. Christie, *Coiled Tubing Sensor Interface Feasibility Report*, Houston, Texas, 1995

9. *SDS Smart Distributed System Application Layer Protocol Specification*, Second Addition, Honeywell, Inc. a MICRO SWITCH Division, 1994 and 1995.

10. V. Mitra Gopual, *Developing C/C++ Software in the OS/2 Environment*, Van Nostrand Reinhold, New York, NY,1994

11. W. Richard Stevens, *UNIX Network Programming*, Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1990.

12. *World Oil's Coiled Tubing Handbook*, Gulf Publishing Company, Houston, Texas, 1993

APPENDIX A

GLOSSARY

ACIDIZING:	The pumping service that places acid in the wellbore to assist the formation in producing oil or gas.
BACK-END:	A complete data analysis computer system that receives sensor data digitally from a front-end device that performs the actual acquisition from the physical sensors.
CEMENTING:	The process of securing the casings in the wellbore by placing cement between the casing and earth.
ETHERNET:	The physical hardware layer which allows computers to be networked together.
FRACTURING:	The process of pumping fluids/materials in the wellbore to make actual fractures in the earth to create an avenue for the oil or gas to enter the wellbore.
FRONT-END:	A data acquisition device that acquires and digitizes actual sensor data for transmission to a more complex analysis computer system.
HUMAN INTERFACE:	A generic term given to the interactive components of a computer system, normally refers to keyboards and displays.
INTERRUPT:	A technique whereby hardware can automatically initiate a software process.
NETWORK:	A cluster of devices linked to together through a common communication scheme.
OPERATOR'S CONTROL CABIN:	A small (4' x 6') enclosed compartment to house the control console and data acquisition equipment. Also used to protect the operator and equipment from the environment.
PROTOCOL:	A formal set of rules and requirements governing the format and relative timing of message exchange between two communication systems.

RAM:	Random Access Memory, the area of memory that is used during operation and is lost after the system is turned off.
REEL:	A fabricated steel spool used to hold tubing.
REEL DATABASE:	The database that is feed into the CoilLIFE program to calculate the 'life' remaining for the particular tubing.
ROM:	Read Only Memory is memory that is programmed by the manufacturer and cannot be changed by the end-user.
NODE:	A particular device located on a network.
SEMAPHORE:	A key tool used by multiple threaded applications to protect resources properly and signal other threads about events occurring.
SOCKET:	A bi-directional pipe for incoming and outgoing data that allows an application program to access the TCP/IP protocols.
TCP/IP:	Transmission Control Protocol/ Internet Protocol is a family of software protocols used for computer communications.
UART:	Universal Asynchronous Receiver Transmitters are the electronic components that enable devices to communicate serially.
WELLBORE:	A generic term being used to refer to the hole that is drilled in the earth that is used to extract oil from the earth.

VITA

Richard L. Christie

Candidate for the Degree of

Masters of Science

Thesis: DEVELOPING A SENSOR NETWORK FOR DATA ACQUISITION IN THE OILFIELD ENVIRONMENT

Major Field: Computer Science

Biographical:

Personal Data: Born in Butler, Pennsylvania, on September 28, 1965, the son of Richard and Carol Christie.

Education: Graduated from Moniteau High School, West Sunbury, Pennsylvania, in May 1983; received a Bachelor of Science in Electronics Engineering Technology from DeVry Institute of Technology, Columbus, Ohio, in October, 1986. Completed requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July 1996.

Experience: Employed by Dowell Schlumberger, Inc. since February, 1987. Duties have varied from field electronic technician to software developer. Have been involved in projects in the areas of portable data acquisition devices, process control equipment, and fluid quality monitoring devices.