Abstract

Oil pipeline Supervisory Control and Data Acquisition (SCADA) systems monitor and control pipes transporting both crude and refined petroleum products. The SCADA system consists of a Master system that communicates with the Remote Terminal Units (RTU) to gather data and check the values for the optimum functioning of the pipeline. The SCADA system checks various parameters like the pressure, temperature, density of the oil inside the pipeline. If the pressure drops below a set threshold, then the pumps get activated to boost the pressure. Reliability of the SCADA system is extremely important for the proper transportation of the oil through the pipelines. In this project we developed some system architectures for improving the reliability of the SCADA system for three failure scenarios: the failure of the master SCADA system, failure in the communication link between the master and RTU’s, and the failure in the pump. The reliability improvements due to the modified system were validated by emulations on a specially developed multi-threaded Java-based emulator called Oil Pipeline SCADA Emulator (OPSE) deployed using Eclipse IDE.
Chapter 1

Introduction

During the last decade, great advances have been made in areas such as computer hardware, software and communications. These advances have facilitated the building of very sophisticated Supervisory Control and Data Acquisition (SCADA) systems for large engineering installations such as electrical networks, water and sewerage networks, oil and gas pipelines etc. Sophistication is not limited to system architecture and hardware capability but also extends to software functions that are available to SCADA operators.

The U.S. consumes about 19.5 million barrels per day (b/d) of petroleum products [6]. A complex and carefully choreographed network is in place to move the raw materials, which are mainly crude oils, from where they are produced to where they are processed, and the refined products from where they are processed to where they are consumed. Distances involved can be enormous. Crude and products arriving from the Middle East have already traveled more than 10,000 miles, and may be shipped thousands more across the U.S. This vast logistical ballet is the key to the oil market, and an outgrowth of the basic shape of the oil market. Areas that are relatively resource-rich supply areas those are relatively demand-hungry but resource-poor. This interplay of resource versus demand regions is a characteristic of world markets, where producing countries supply consuming countries, and of regional markets in the United States, where producing and refining areas supply consuming areas.
1.1 How does the Oil Pipeline System work?

Figure 1 is a brief high level architecture of a typical SCADA System [8]. Field instruments are placed at regular intervals along the oil pipelines and are used to read various parameters like the pressure, temperature etc. In Figure 1, P and T are the pressure and temperature sensors respectively. These sensors keep sensing values and send the values to the Remote Terminal Units. The RTU’s in turn send these values to the SCADA master system or the main control room. Different types of communication mechanism are used to communicate with the master system. LAN’s can be used for small systems and wireless communication mechanisms like the satellites can be used for huge systems like the oil pipelines. The master system checks if the values are in range or if they are going out of range. If the values are going out of range, the master sends a message or an alarm to the RTU’s asking them to perform the required action. The master system will also need to open/close valves at different places in the pipeline for the oil to reach different destinations. This action of opening or closing a valve is termed as “Making a Cut”.

1.2 Managing Oil Pipeline Flow

Oil is generally propelled through pipelines by centrifugal pumps. The pumps are sited at the originating station of the line and at 20 to 100 mile intervals along the length of the pipeline, depending on pipeline design, topography and capacity requirements [6]. Most pumps are driven by electric motors, although diesel engines or gas turbines may also be used.
Figure 1: SCADA System Architecture
Pipeline employees using computers remotely control the pumps and other aspects of pipeline operations. Pipeline control rooms utilize Supervisory Control and Data Acquisition (SCADA) systems that return real-time information about the rate of flow, the pressure, the speed and other characteristics. Both computers and trained operators evaluate the information continuously. Most pipelines are operated and monitored 365 days a year, 24 hours per day. In addition, instruments return real-time information about certain specifications of the product being shipped – the specific gravity, the flash point and the density, for example – information that are important to product quality maintenance [6].

Oil moves through pipelines at speeds of 3 to 8 miles per hour. Pipeline transport speed is dependent upon the diameter of the pipe, the pressure under which the oil is being transported, and other factors such as the topography of the terrain and the viscosity of the oil being transported. At 3-8 mph it takes 14 to 22 days to move oil from Houston, Texas to New York City [6].

Pipeline operators ship different petroleum products or grades of the same product in sequence through a pipeline, with each product or “batch” distinct from the preceding or following. Figure 2 shows how different petroleum products or different grades of the same product are being moved inside the pipeline [6]. One refined product or crude oil grade is injected and begins its journey, then another, and another. A batch is a quantity of one product or grade that will be transported before the injection of a second product or grade.

There is always a certain amount of intermixing between the first product and the second at the "interface," the point where they meet. If the products are similar, such as two grades of gasoline, the resulting mixture is added to the lower value product. If the products are dissimilar,
such as diesel and gasoline, the "transmix," the hybrid product created by intermixing at the interface, must be channeled to separate storage and reprocessed.

![Diagram of petroleum product flow through a pipeline]

**Figure 2: Typical sequence of Petroleum products flow through a pipeline**

Batching petroleum for pipeline transport has become more complex with the proliferation of product qualities. Colonial Pipeline, for instance, publishes specifications for over 100 different grades of gasoline. Crude oil pipelines, too, must meet market demands for delivering various crude types – such as high sulfur or low sulfur grades – to refineries to align with the refineries’ schedules for producing jet fuel, asphalt, diesel, and other products and to the refineries’ equipment. Lakehead pipeline's 1.3 million barrel per day system, for instance, can contain up to 50 batches of crude oil of distinct qualities [8].

Pipeline operators establish the batch schedules well in advance. A shipper desiring to move product from the Gulf Coast to New York Harbor knows months ahead the dates on which Colonial will be injecting heating oil, for instance, into the line from a given location [6].
Chapter 2
SCADA System

This chapter briefly describes the SCADA system and its application in oil pipelines.

2.1 How does the SCADA System work?

A SCADA system usually consists of the following subsystems.

- A Human Machine Interface
- A Supervisory Control System
- Remote Terminal Units
- Communication Infrastructure

- A Human-Machine Interface or HMI is the apparatus which presents process data to a human operator, and through this, the human operator, monitors and controls the process.

- A supervisory (computer) system collects data on the process and sends commands (control) to the process.

- Remote Terminal Units (RTUs) are connected to sensors in the processes, converting sensor signals to digital data and sends digital data to the supervisory system.

- Communication infrastructure connects the supervisory system to the Remote Terminal Units.

These functions are performed by four kinds of SCADA components: [7]

- Sensors (either digital or analog) and control relays that directly interface with the managed system.
Remote Terminal units (RTUs). These are small computerized units deployed in the field at specific sites and locations. RTUs serve as local collection points for gathering reports from sensors and delivering commands to control relays.

SCADA Master Units. These are larger computer consoles that serve as the central processor for the SCADA system. Master units provide a human interface to the system and automatically regulate the managed system in response to sensor inputs.

The Communications network that connects the SCADA master unit to the RTUs in the field.

2.2 SCADA system for Oil Pipelines

Oil is transported from the oil wells to the refineries and refineries to various places through the pipelines. Since the oil pipelines travel through thousands of miles, a single system cannot monitor the entire pipeline. Hence many devices like field instruments, programmable logic units and RTU’s are placed at different places throughout the pipeline. These field instruments and the programmable logic units are connected to RTU’s and the SCADA system is used to monitor all these RTU’s. Various parameters are being sensed by these devices namely pressure, temperature of the oil flowing inside the pipeline, density of the liquid flowing etc.

Since there are many RTU’s located at different places throughout the pipeline and since all these RTU’s communicate with the SCADA master system, rapid exchange of data takes place. The RTU’s sends the data to the SCADA master system and the SCADA master system checks if everything is working well or if there are any changes to be made. If there are any changes to be made, the SCADA master sends a message to the respective RTU and it performs the required action.
Any minor changes that occur in the pipeline can be detected by the SCADA master system and the required action can be taken. The application of SCADA has reduced huge amount of man power in the field which in turn has reduced the expenditure.
Chapter 3

Oil Pipeline SCADA Emulator

3.1 Description

Some of the emulators that are currently available in the market are the Field Interface Unit (FIU) etc. An FIU is one or more CPU’s running from a MOSCAD RTU [9]. We observed that for most of the existing emulators, the main emphasis is on the leak detection scenarios and how the emulators would handle those leaks. The other reliability issues like the failure of the Master system, the failure in the communication link between the RTU’s and the Master, the failure in the pump and many other security issues like securing the information of the Master system, backing up the data are ignored.

Therefore for analyzing reliability scenarios related to oil pipelines, we developed the Oil Pipeline SCADA Emulator (OPSE) to emulate an oil pipeline SCADA system. An emulator duplicates the functions of one system using a different system, so that the second system behaves like (and appears to be) the first system. OPSE is multithreaded and employs message passing between threads in Java. As everything runs in parallel in the real world, OPSE models the real world very well.

Here is a brief description of the system. Figure 3 shows the Class diagram for the OPSE. We have two threads namely PressureSensor and the TemperatureSensor. We assumed that the temperature inside the oil pipeline would be constant. Other classes that we used include RtuProcessor, RtuCommunicator, MasterCommunicator,
MasterProcessor, and Motor. For the scenarios, I added some other classes like the BackupMasterProcessor, BackupMotor etc.

The sequence diagram for SCADA system is given in Figure 4 and the details of the messages are given in Figure 5. The SensorSystem class initiates the thread PressureSensor. The PressureSensor thread senses the pressure values of the oil flowing inside the pipeline and sends them to the RtuProcessor class. The RtuProcessor class sends these values that it collected from the PressureSensor thread to the RtuCommunicator class. The RtuCommunicator class sends these values to the MasterCommunicator class. The MasterCommunicator class sends these values to the MasterProcessor class. The MasterProcessor class writes these values to a file and checks these values. If the values are in range, then it sends a message “Safe” to RtuProcessor class. If the pressure value is falling low, then it sends an “Increase” message to the RtuProcessor class asking it to increase pressure. When the RtuProcessor class receives an “Increase” message from the MasterProcessor class, it starts the Motor class. When the MasterProcessor class needs to send a message to the RtuProcessor class, it goes through the following path.

MasterProcessor to MasterCommunicator,
MasterCommunicator to RtuCommunicator,
RtuCommunicator to RtuProcessor.

Similarly when the RtuProcessor class needs to send a message to the MasterProcessor class, it goes through the following path.
RtuProcessor to RtoCommunicator,

RtuCommunicator to MasterCommunicator,

MasterCommunicator to MasterProcessor.

The complete code in Java for OPSE is given in the Appendix C.
Figure 3: Class Diagram for the OPSE
Figure 4: Sequence diagram for OPSE
1. PressureSensor thread senses the pressure inside the pipeline and sends the values to the RtuProcessor class.

2. RtuProcessor class checks for where the message came from. If the message is from the PressureSensor thread, it sends the pressure value to the RtuCommunicator class.

3. RtuCommunicator class checks for where the message came from. If the message is from the RtuProcessor class, it sends the pressure value to the MasterCommunicator class.

4. The MasterCommunicator class checks for where the message came from. If the message is from the RtuCommunicator class, it sends the pressure value to the MasterProcessor class.

5. The MasterProcessor class stores these values in a file and checks if the pressure values are in range. If they are in range it sends a “Safe” message to the MasterCommunicator class. If the pressure value is low, it sends an “Increase” message to the MasterCommunicator class.

6. The MasterCommunicator class checks for where the message came from. If the message came from the MasterProcessor class, it sends the same message to the RtuCommunicator class.

7. The RtuCommunicator class checks for where the message came from. If the message is from the MasterCommunicator class, it sends the same message to the RtuProcessor class.

8. The RtuProcessor class checks for where the message came from. If the message is from the RtuCommunicator class, it checks for if the message is “Safe” or “Increase”. If the message is “Increase” it sends a message to the Motor class asking it to activate.

**Figure 5: Legend for the messages in the Figure 4**
3.2 Emulation

The basic structure of the emulated system contains three units namely Sensor, Remote Terminal Unit and a Master Unit (SCADA system) as shown in Figure 6. Each Remote Terminal Unit consists of a RTU Processor and a RTU Communicator and each Master unit consists of a Master Communicator and a Master Processor. We considered the vMBusX-SP wireless pressure sensor (datasheet given in Appendix A) and the vMBusX-C wireless Remote Terminal Unit (datasheet given in Appendix B) for emulation [10, 11]. There are two sensors namely PressureSensor and TemperatureSensor for each RTU and one pump for a set of RTU’s.

![Diagram of Typical Oil Pipeline SCADA system architecture](image-url)

**Figure 6: Typical Oil Pipeline SCADA system architecture**
Figure 7 is a sample output from the OPSE. These are the pressure values at RTU’s 1, 2, 3, 4 etc. The average pressure inside the pipeline is between 50 and 60 psi. As the oil moves through the pipeline, its velocity gradually decreases due to the friction between the oil and the surface of the pipeline. We can see that there is gradual decrease in the readings. So as the pressure reading goes less than 50 psi, we need to start the pump (or motor) to boost the oil. So at 50 psi, when we start the motor the pressure increases to 60 psi and again gradually decreases as it moves.
The graph for the pressure values would look like Figure 7. We can see that the pressure decreases gradually as the oil flows through the pipeline. When it reaches 50 psi as the motor gets activated and boosts the oil, its pressure increases and then it again decreases gradually. The horizontal axis shows the RTU numbers and the vertical axis shows the pressure values in psi.

Figure 8: Graph that shows the changes in pressure values in psi with time at each RTU
Chapter 4

Reliability Analysis

4.1 Reliability of the SCADA System

Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time. Reliability is one of the important aspects to test in any application. Reliability scenario is nothing but a possible occurrence of an event in a system. This is to test if the system works under abnormal conditions. All the scenarios that are possible are emulated and tested before an application goes live in any company. The more the system is reliable, the more useful the system becomes. Some of the reliability scenarios for an oil pipeline SCADA system include what would happen if the Master station fails, what would happen if there is communication failure between the RTU’s and the Master station, what would happen if the pump fails to activate when required. How would the system react to these scenarios? The reason for choosing these scenarios is that these are considered the most critical components of the system. We emulated the system response to these events, augment the system to improve reliability, and then verify using OPSE that the augmented system is indeed reliable.
4.2 Master failure scenario analysis

In a real time SCADA system as all the actions are controlled by the master, any damage to the master would affect the total system badly. In order to make the system more reliable we used a backup master processor which can be used in the case of master system failure. The proposed architecture for this scenario is shown in Figure 9. The sequence diagram for this scenario is shown in Figure 10 and the messages are given in Figure 11.

Figure 9: Proposed Architecture for the Master station failure scenario
Figure 10: Sequence diagram for the Master station failure scenario
1. PressureSensor senses the pressure inside the pipeline and sends the values to the RtuProcessor.

2. RtuProcessor checks for where the message came from. If the message is from the PressureSensor, it sends the pressure value to the RtuCommunicator.

3. RtuCommunicator checks for where the message came from. If the message is from the RtuProcessor, it sends the pressure value to the MasterCommunicator.

4. The MasterCommunicator checks for where the message came from. If the message is from the RtuCommunicator, it sends the pressure values to the MasterProcessor.

5. The MasterCommunicator also sends the values to the BackupMasterProcessor.

6. The BackupMasterProcessor stores these values in a file and checks if the pressure values are in range. If they are in range, it sends a “Safe” message to the MasterCommunicator. If the pressure value is low, it sends an “Increase” message to the MasterCommunicator.

7. The MasterCommunicator checks for where the message came from. If the message came from the BackupMasterProcessor, it sends the same message to the RtuCommunicator.

8. The RtuCommunicator checks for where the message came from. If the message is from the MasterCommunicator, it sends the same message to the RtuProcessor.

9. The RtuProcessor checks for where the message came from. If the message is from the RtuCommunicator, it checks for if the message is “Safe” or “Increase”. If the message is “Increase” it sends a message to the Motor asking it to activate.

Figure 11: Legend for the messages in Figure 10
Figure 12 shows the pressure values recorded by OPSE for the Master failure scenario. We have a Backup Processor so that we can still make the system work using the Backup Processor. We can see that there are no values in the readings file as the master processor failed.

Figure 12: Pressure values recorded by OPSE for the Master station failure scenario
The graph of pressure values for the Master station failure scenario is shown in Figure 13.

Figure 13: Graph that shows the changes in pressure values in psi with time at each RTU for the master failure scenario
4.3 Communication link failure scenario analysis

Chances are very likely that the communication link between the RTU’s and the master processor fails. Since the entire system is dependent upon the master, any damage to the master would affect the entire pipeline system. We can overcome this situation by an extent by using the Intelligent RTU’s. Intelligent RTU’s are nothing but the RTU’s which are capable of managing the system when something happens to the master or if they do not receive the messages from the central SCADA. That is, instead of making the system going into a dead state, these RTU’s can perform some basic tasks like resetting the pressure etc.

The proposed architecture for the Communication link failure scenario is shown in Figure 14. Figure 15 shows the sequence diagram for the communication link failure scenario and Figure 16 shows the messages in Figure 15.

![Proposed Architecture for the communication link failure scenario](image)

Figure 14: Proposed Architecture for the communication link failure scenario
1. PressureSensor senses the pressure inside the pipeline sends the values to the RtuProcessor.

2. The RtuProcessor checks if the values are in range. If they are below the range it sends an “Increase” message to the Motor asking it to Increase Pressure.

**Figure 15: Sequence Diagram for the communication link failure scenario**

**Figure 16: Legend for the messages in Figure 15**
The output for the communication link failure scenario is shown in the Figure 17. There are no output files as the values did not reach the master.

**Figure 17: Output for the communication link failure scenario**
4.4 Pump failure scenario analysis

It is also possible for the pump to fail. When there is no pump, then the pressure readings would be drop continuously. Due to the friction between the oil and pipeline inside the pipeline its velocity decreases gradually and if there is no boost to the oil at right times, it takes forever for the oil to move from the source to the destination. One alternative for this scenario is using a backup pump. The proposed architecture for this scenario is to have a backup pump that can be used when there is failure in the main pump. The proposed architecture for the Pump failure scenario is shown in Figure 18.

![Proposed Architecture for the pump failure scenario](image-url)
Figure 19 shows the sequence diagram for the pump failure scenario and Figure 20 shows the messages for Figure 19.

Figure 19: Sequence diagram for the pump failure scenario
1. PressureSensor senses the pressure inside the pipeline sends the values to the RtuProcessor.

2. RtuProcessor checks for where the message came from. If the message is from the PressureSensor, it sends the pressure value to the RtuCommunicator.

3. RtuCommunicator checks for where the message came from. If the message is from the RtuProcessor, it sends the pressure value to the MasterCommunicator.

4. The MasterCommunicator checks for where the message came from. If the message is from the RtuCommunicator, it sends the pressure values to the MasterProcessor.

5. The MasterProcessor stores these values in a file and checks if the pressure values are in range. If they are in range it sends a “Safe” message to the MasterCommunicator. If the pressure value is low, it sends an “Increase” message to the MasterCommunicator.

6. The MasterCommunicator checks for where the message came from. If the message came from the MasterProcessor, it sends the same message to the RtuCommunicator.

7. The RtuCommunicator checks for where the message came from. If the message is from the MasterCommunicator, it sends the same message to the RtuProcessor.

8. The RtuProcessor checks for where the message came from. If the message is from the RtuCommunicator, it checks for if the message is “Safe” or “Increase”. If the message is “Increase” it sends a message to the BackupMotor asking it to activate.

Figure 20: Legend for the messages in Figure 19
The output for the pump failure scenario is shown in the Figure 21.
The graph for the pump failure scenario is shown in the Figure 22.

Figure 22: Graph that shows the changes in pressure values in psi with time at each RTU for the pump failure scenario
If we do not have a backup motor, then the pressure values drop continuously as shown in Figure 23.

**Figure 23:** Graph that shows the drop in the pressure if there is no backup pump
Table 1 gives a short description of the different scenarios and their effect on the performance, security and cost of the system. The reliability improvement architecture modification used for the failure of the master station is using a duplicate master. When the duplicate master is used, the overall speed of the system reduces as the data should be sent to two masters now and the system becomes a little vulnerable as we need to secure two masters. The cost incurred increases as we need to install a new master.

The reliability improvement architecture modification used for the failure of the communication link between the RTU’s and the master is using the intelligent RTU’s that can take decisions independent of the master. The cost incurred increases as we need intelligent RTU’s and they are a little expensive than the normal RTU’s. The performance of the system increases as the RTU’s can take decisions and act faster. The system becomes more vulnerable when using the intelligent RTU’s because they will targeted by the ones who want to destroy the system. However the communication link vulnerability is improved.

The reliability improvement architecture modification used for the failure in the pump is using a back up pump. The performance of the system decreases since we need to decide on which pump to use. The cost incurred increases as we need another pump and the system becomes a little more vulnerable since there are now two links to the pumps.
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Reliability Improvement Architectural Modifications</th>
<th>Effect on Performance</th>
<th>Effect on Security</th>
<th>Effect on Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure of Master station</td>
<td>Duplicate Master</td>
<td>Speed reduces as the data should be sent to two Masters now.</td>
<td>A little more vulnerable to the security attacks as we need to secure two Masters now.</td>
<td>Increases as we need to install a new Master.</td>
</tr>
<tr>
<td>Failure in the communication link between the RTU’s and master</td>
<td>Intelligent RTU’s that can take decisions independent of the master</td>
<td>Performance increases as the RTU’s can take the decisions and act faster.</td>
<td>A little more vulnerable as the RTU’s can be attacked; However communication link vulnerability is removed</td>
<td>Cost increases as we need intelligent RTU’s.</td>
</tr>
<tr>
<td>Pump failure</td>
<td>Backup pump</td>
<td>Performance decreases since we need to decide which pump to use.</td>
<td>A little more vulnerable since there are now two links to pumps.</td>
<td>Cost increases as we need another pump.</td>
</tr>
</tbody>
</table>

Table 1: Reliability Scenarios and Architectural Modifications
Chapter 5

Developing a Simulator for Oil Pipeline SCADA system

We have developed system architectures for some reliability scenarios like the failure of the master station, failure of the communication link between the RTU’s and the master, failure of the pump etc and verified the architectures using OPSE. A simulator can be used to test the reliability of SCADA systems under probabilistic variations of scenarios. For example, instead of linear decrease in pressure, we could have pressure variations following probability distributions. The results from simulation can be used to augment the system architectures developed using the emulator. In real situations, pressure values at different RTU’s may remain same, increase, decrease or vary randomly. The simulator can use the Poisson distribution formulae’s for pressure values, temperature values, pump settings, valve settings and failure estimation. The simulator can be further developed to download global information to the intelligent RTU’s. i.e., in case of a master failure, one RTU can take the responsibility of a master and capture global system information.

Such a simulator can be used for the following:

- Stress the system architecture to detect weak points.
- Analyze peak and average performance of different system components.
- Evaluate effectiveness of security measures.
- Estimate cost for improving reliability under different stress/performance/security scenarios.
Chapter 6

Conclusions and Future work

Before the development of the SCADA systems, we had to send people around the pipelines to close valves and turn on pumps at various times. This requires lot of effort and man power. The use of SCADA systems became popular in 1960’s as a need to arise to more efficiently monitor and control the state of remote equipment. A problem with early SCADA systems is that they required human oversight to make decisions as well as human support to maintain the information system. With the technology growing rapidly, the human intervention in the SCADA system is becoming very low. Human intervention is needed only for some very rare events. Reliability of the SCADA system is very important for the proper functioning of the system.

We have emulated some reliability scenarios like the master station failure, communication link failure and motor failure. We developed architectural modifications to improve reliability for these scenarios. This system can be further developed to work on some other issues like the security scenarios, a burst in the pipe, or doing a little more deep research into the system and improving the speed of the system etc.
Chapter 7

References


3. www.aopl.com

4. www.alyeska-pipe.com


8. www.wikipedia.com

9. http://findarticles.com/p/articles/mi_m3251/is_/ai_n25025872


vMBusX-SP - is a solution to meet the demands of the oil and gas industry for a reliable and cost-effective means to remotely monitor their applications. The vMBusX-SP incorporates our powerful vMAE micro RTU with our proven battery-operated wireless pressure sensor to produce a small, low-power, explosion-proof RTU-Sensor system that can be mounted directly to the process line (e.g., wellheads, pipelines, gas heads, etc.). Combined with a Gateway, the vMBusX-SP, installed at the remote site, can acquire real-time data for analysis and detect alarms. Easily customizable for many applications needing a wireless pressure sensor.

Some of the Benefits and Features
- Remote Configuration
- Selectable Wake-up interval from 1 second to 18 hours
- Report by exception
- Battery Level Indicator
- Built-in 128-bit encryption
- Selectable data reporting (Raw, Engineering units, SQRT, Linear and squint accumulation)
- Network Time Synchronization
- Data History
- Min/MAX and Percent change Alarms
- Diagnostic data (response time, message count and signal strength)

General Specifications

Inputs
- 2 A/D channels - 16 bit resolution (0-5 VDC or 0-50 mA)
- 1 Opto-Isolated Digital input channels (3-24 VAC or DC)
- 1 Digital channel monitors the status of a digital switch: used for tank monitoring & detecting alarms
- Independent A/D channel used for battery voltage level
- Independent A/D channel used for temperature
- Independent A/D channel used for radio diagnostics

Processor and Memory
- 8051 Micro-Controller running at 12.50MHz with 64Kb Flash Memory

Radio
- Plug-in radio modem operating at a frequency of 900 MHz - 2.4 GHz or Zigbee

Serial Port
- 1 Serial interface port-band rate up to 115,200 bps

Relays
- 1 Solid State Relays that can support up to 550 mA continuous current
- On-Off filtering

Power
- Input Range 9.3 to 12.2 VDC
- Low Power Operating Mode with <50 µA in sleep mode with wake up timer running
- Terminal blocks for providing regulated 6.77 V & 18 Vdc power sensors (providing up to 200 mA to power sensors or devices)
- Power consumption <10 mA without radio

Operation Modes
- Continuous
- Stand-alone
- Timer/Alarm
- On-board
- Wake up timer adjustable from 1 sec. to 255 hours

Operating Temperature
- -40°C to +85°C (-40°F to +185°F)

Humidity Range 5-90% non-condensing

Dimensions
- 6.8 cm x 6.0 cm

Support
- vMAE Communication Protocol
- M220BUS Protocol Support

Firmware
- Data Filter, Pulse Counter/Accumulation, Square Root Extraction, Engineering Units Conversion/Scaling
Radio Specifications for vMBusX-SP: (Range and Transmit Power may vary due to various factors, such as type and elevation of antenna, line of sight, the environment, and power output. A vMonitor Engineer can assist you with determining range, frequency, transmit power and other factors to develop a solution to meet your requirements.)

<table>
<thead>
<tr>
<th>Compatible Radios</th>
<th>Range</th>
<th>Transmit Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 GHz</td>
<td>Up to 16 km (10 miles)</td>
<td>50 mW-100 mW</td>
</tr>
<tr>
<td>2.4 GHz ZigBee</td>
<td>Up to 0.12 km (400 ft)</td>
<td>125 mW</td>
</tr>
<tr>
<td>900 MHz</td>
<td>Up to 32 km (20 miles)</td>
<td>5 mW-1000 mW variable</td>
</tr>
</tbody>
</table>

Other Wireless Products from the vMBus family:

- vMBusX-G
  (Gateway)

- vMBusX-C RTU Rev. 5.1
  (SAD, 2AO, 2DI, 2DO, RTD, BAT)

- vMBusX-NANO RTU Rev. 2.1
  (1AD, 1DI, 1DO, 1DO, 1DO, BAT, RSSI)

- vMBusX-MBO RTU Rev. 1.0
  (GAD, 8DL, 2FC/TC, 5RTU, 5AD, 5RTU, 5AD, RSSI, RSSI, RSSI)

- vMBusX-SP RTU Rev. 3.0
  (6AD, 2AO, 2DI, 2DO, RTD, BAT)

Keys:

- AD: Analog Input
- AO: Analog Output
- DI: Digital Input
- DO: Digital Output
- FC: Frequency Counter
- PC: Pulse Counter
- RLY: Solid State Relay
- RTD: Resistance Temp Device
- RTU: Remote Terminal Unit
- TM: Tank Monitoring

For more information on our products and services or to place an order, please check out our website at www.vMonitor.com or contact us at +1.886.514.4935 to speak to one of our engineers. Let us help you make the virtual oilfield a reality.
# vMBusX-SP

## Wireless Pressure Sensor

### Wireless Sensor Specifications

<table>
<thead>
<tr>
<th>Specifications without model designation apply for all models:</th>
<th>500kPa</th>
<th>5 PSI</th>
<th>10PSI</th>
<th>25PSI</th>
<th>50PSI</th>
<th>100PSI</th>
<th>150PSI</th>
<th>200PSI</th>
<th>250PSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure range: <strong>Ambient</strong></td>
<td>5kPa</td>
<td>0.05PSI</td>
<td>0.1PSI</td>
<td>0.25PSI</td>
<td>0.5PSI</td>
<td>1PSI</td>
<td>1.5PSI</td>
<td>2PSI</td>
<td>2.5PSI</td>
</tr>
<tr>
<td>Burst pressure: <strong>Ambient</strong></td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Burst pressure: <strong>Storage</strong></td>
<td>30PSI</td>
<td>2PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure -50°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure -10°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure -5°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure 10°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure 25°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
<tr>
<td>Pressure 40°C</td>
<td>50PSI</td>
<td>3PSI</td>
<td>15PSI</td>
<td>30PSI</td>
<td>60PSI</td>
<td>120PSI</td>
<td>180PSI</td>
<td>240PSI</td>
<td>300PSI</td>
</tr>
</tbody>
</table>

---

**Materials**

<table>
<thead>
<tr>
<th>Wall faced parts</th>
<th>Steel, stainless steel, brass, aluminum, copper, titanium, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaskets</td>
<td>Neoprene, Viton, EPDM</td>
</tr>
<tr>
<td>Internal transmission fluid</td>
<td>Synthetic (fluorocarbon oil for oxygen applications)</td>
</tr>
</tbody>
</table>

**Power supply: DC 5V, 10mA, 3-wire**

**Maximum load: Rs**

<table>
<thead>
<tr>
<th>Model</th>
<th>Rs (ohm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS-2-SP</td>
<td>5000</td>
</tr>
</tbody>
</table>

**Response time (10-90%):**

- 10ms for pressure changes above 100 kPa
- 50ms for pressure changes below 100 kPa

**Repeatability:**

- ±0.05% of span
- ±0.05% of full scale

**Permissible temperature:**

- **Ambient:** -50°C to 120°C
- **Storage:** -50°C to 120°C

## Remote Monitoring

- **Telemetry**
- **Oils, Gas Utility**
- **Waste Water & Water**

---

For more information on our products and services or to place an order, please check out our website at [www.vMonitor.com](http://www.vMonitor.com) or contact us at +1.866.514.4935 to speak to one of our engineers.

Let us help you make the virtual oilfield a reality.
vMBusX-C - is an intelligent remote terminal unit that combines the latest in wireless technology with low power consumption to provide a reliable and cost effective means to remotely monitor and automate your applications in the oil and gas fields. The vMBusX-C is housed in an explosion proof enclosure and is typically mounted on the process line (pipelines, gas lines etc.) coupled with a gateway, it can be a stand-alone RTU or part of an integrated network to acquire real-time data for analysis and detect alarms. Also it can be configured for remote automation of controls at remote sites. vMBusX-C has been successfully implemented for wellhead, ESP, downhole gauges, and gas lift valve monitoring and control. Easily customizable for a variety of other applications and third party equipment.

Some of the Benefits and Features:
- Remote configuration
- Selectable Wake-up Interval from 1 second to 18 hours
- Report by exception
- Battery Level Indicator
- Built-in 128 bit encryption
- Selectable data reporting (Flow, Engineering Units, SOFT, Linear and SOFT accumulation)
- Network Time Synchronization
- Data History
- Min/Max and Percent change Alarms
- Diagnostic data (response time, message count and signal strength)

General Specifications:
- Inputs
  - 1 A/D channels - 16 bit resolution (9.5 VDC or 4-20 mA)
  - 2 Opto-isolated Digital Input channels (8-24 VAC or DC)
  - Battery voltage level channel
  - Temperature channel
  - Humidity sensor (optional)
- 1-wire RTD Channel with 16 bit resolution

Outputs
- 2 RS485 channels - 12 bit resolution: current (4-20 mA) & voltage (0-5 V)

Processor and Memory:
- 6051 Micro-Controller running at 12.5 MHz with 64K Flash Memory
- Support for up to 512K-Byte flash data memory

Radio
- Plug in radio modem operating at a frequency of 500 MHz, 2.4 GHz or Zbge

Serial Ports
- 1 RS232/RS485 serial interface port with baud rate up to 115,200 bps
- 1 RS232 serial interface port that can be used for the radio

Relays
- 2 Solid State Relays that can support up to 350 mA continuous current

Power
- Input Range: 7.3 VDC
- Low Power-Operating Mode
- Terminal blocks for providing regulated 10 V & 19 V to power sensors

Temperature
- Operation: -40°C to +85°C (-40°F to +185°F)

Dimensions
- 12.5 cm x 11.5 cm x 5.5 cm

Support
- Modbus Communications Protocol
- MODBUS Protocol Support

Firmware
- Data Totallizer, Pulse Counter/Accumulator, Square Root Extration, Engineering units, Conversion/Scaling, PID, Alarms, Valve Control, Remote Configuration

For more information on our products and services or to place an order, please check out our website at www.Trimon.com or contact us at +1.866.514.4935 to speak to one of our engineers. Let us help you make the virtual oilfield a reality.
Radio Specifications for vMBusX-C: Range and Transmit Power may vary due to various factors, such as type and elevation of antenna, line of sight, the environment, and power output. A vMonitor Engineer can assist you with determining range, frequency, transmit power and other factors to develop a solution to meet your requirements.

<table>
<thead>
<tr>
<th>Compatible Radios</th>
<th>Range</th>
<th>Transmit Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 GHz</td>
<td>Up to 16 km (10 miles)</td>
<td>50 mW-100 mW</td>
</tr>
<tr>
<td>900 MHz</td>
<td>Up to 32 km (20 miles)</td>
<td>5 mW-1000 mW variable</td>
</tr>
</tbody>
</table>

Other Wireless Products from the vMBus Family:
- vMBusX-C (Gateway) (RadioBase)
- vN-Micro RTU Rev. 1.0 (2AD, 2DI, 1RLY, BAT, RSSI)
- vN-Nano RTU Rev. 2.1 (1AD, 1DI, 1TC, BAT, RSSI)
- vMBusX-MIC RTU Rev. 1.0 (848, 8DI, 1TC, 8RLY, 8AO, RS485, RS232, 12C)
- vMBusX-SS RTU Rev. 2.0 (64I, 1AO, 2DI, 2RLY, 2AO, 2DI, 2AO, 2DI, 2AO, 2DI, 2AO)

Keys:
- AD Analog Input
- AO Analog Output
- DI Digital Input
- FC Frequency Counter
- TC Pulse Counter
- RLY Solid State Relay
- RTU Remote Terminal Unit
- TM Tank Monitoring

For more information on our products and services or to place an order, please check out our website at www.vMonitor.com or contact us at 1.866.511.4911 to speak to one of our engineers. Let us help you make the virtual oilfield a reality.
Appendix C

OPSE Code

SensorSystem.java

public class SensorSystem {

    private static MasterProcessor masterProcessor = new MasterProcessor();
    private static BackupMasterProcessor bpMasterProcessor = new BackupMasterProcessor();
    private static MasterCommunicator masterCommunicator = new MasterCommunicator();
    private static RtuProcessor rtuProcessor = new RtuProcessor();
    private static RtuCommunicator rtuCommunicator = new RtuCommunicator();
    public static Object lock = new Object();
    public static Object pressureLock = new Object();
    public static Object temperatureLock = new Object();

    public static MasterProcessor getMasterProcessor() {
        return masterProcessor;
    }

    public static BackupMasterProcessor getBackupMasterProcessor() {
        return bpMasterProcessor;
    }

    public static MasterCommunicator getMasterCommunicator() {
        return masterCommunicator;
    }

    public static RtuProcessor getRtuProcessor() {
        return rtuProcessor;
    }

    synchronized public static RtuCommunicator getRtuCommunicator() {
        return rtuCommunicator;
    }

    public static int getI() {
        return i;
    }

    public static void main(String[] args) {
        TemperatureSensor ts = new TemperatureSensor("TempSensor1");
        PressureSensor ps = new PressureSensor("PressureSensor1");
    }
}
public class PressureSensor extends Observable implements Runnable, Observer {
    int pressure = 62;

    public PressureSensor(String name) {
        Thread prThread = new Thread(this);
        prThread.start();
    }

    public void run() {
        int i = 0;
        while (++i < 10) {
            pressure -= 2;
            deleteObservers();
            addObserver(SensorSystem.getRtuProcessor());
            synchronized (SensorSystem.lock) {
                System.out.println("Reading by Pressure Sensor = "+ pressure);
                setChanged();
                notifyObservers(new Integer(pressure));
                SensorSystem.lock.notify();
                System.out.println("Pressure Sensor Cycle completed");
            }
        }
    }

    synchronized public void update(Observable rtuProcessor, Object value) {
        if (pressure < 52) {
            System.out.println("RtuProcessor ------ > Pressure Sensor : "+ value);
            System.out.println("Resetting Pressure");
            pressure = 62;
        }
    }
}
public class RtuProcessor extends Observable implements Observer {

    private Motor motor = new Motor();
    private Object lock = new Object();
    private Observer sensor;
    //private BackupMotor backupmotor = new BackupMotor();

    synchronized public void update(Observable sender, Object value) {
        if (sender instanceof PressureSensor) {
            sensor = (Observer) sender;
            //backupmotor.setSensor(sensor);
            motor.setSensor(sensor);
            synchronized (lock) {
                deleteObservers();
                addObserver(SensorSystem.getRtuCommunicator());
                if (sender instanceof PressureSensor) {
                    System.out.println("PressureSensor -------- > RtuProcessor : "+ value);
                    setChanged();
                    notifyObservers(value);
                }
            }
        } else if (sender instanceof RtuCommunicator) {
            synchronized (lock) {
                deleteObservers();
                String strValue = (String) value;
                System.out.println("RtuCommunicator -------- > RtuProcessor : "+ value);
                if (strValue.equals("increase")) {
                    //addObserver(backupmotor);
                    addObserver(motor);
                    addObserver(sensor);
                    setChanged();
                    notifyObservers("increase");
                } else if (strValue.equals("safe")) {
                    //System.out.println("RtuCommunicator -------- > RtuProcessor: Pressure is Safe");
                }
            }
        }
    }
}
RtuCommunicator.java

import java.util.Observable;
import java.util.Observer;

public class RtuCommunicator extends Observable implements Observer{
    synchronized public void update(Observable sender, Object value) {
        deleteObservers();
        if (sender instanceof RtuProcessor) {
            addObserver(SensorSystem.getMasterCommunicator());
            System.out.println("RtuProcessor -------- > RtuCommunicator : " + value);
        }
        setChanged();
        notifyObservers(value);
        }
    
    else if (sender instanceof MasterCommunicator) {
        addObserver(SensorSystem.getRtuProcessor());
        System.out.println("MasterCommunicator -------- > RtuCommunicator : " + value);
        
        setChanged();
        notifyObservers(value);
    }
}
}
MasterCommunicator.java

import java.util.Observable;
import java.util.Observer;

public class MasterCommunicator extends Observable implements Observer{

    synchronized public void update(Observable sender, Object value) {
        deleteObservers();
        if (sender instanceof RtuCommunicator) {
            addObserver(SensorSystem.getMasterProcessor());
            //addObserver(SensorSystem.getBackupMasterProcessor());
            System.out.println("RtuCommunicator -------- > MasterCommunicator : " + value);
            setChanged();
            notifyObservers(value);
        } else if (sender instanceof MasterProcessor) {
            addObserver(SensorSystem.getRtuCommunicator());
            System.out.println("MasterProcessor -------- > MasterCommunicator : " + value);
            setChanged();
            notifyObservers(value);
        } else if (sender instanceof BackupMasterProcessor) {
            addObserver(SensorSystem.getRtuCommunicator());
            System.out.println("MasterProcessor -------- > MasterCommunicator : " + value);
            setChanged();
            notifyObservers(value);
        }
    }
}
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.util.Observable;
import java.util.Observer;
import java.util.Vector;

public class MasterProcessor extends Observable implements Observer{
    Vector values = new Vector();
    private int minPressure = 52;

    synchronized public void update(Observable reader, Object value) {
        deleteObservers();
        addObserver(SensorSystem.getMasterCommunicator());
        if (reader instanceof MasterCommunicator) {
            System.out.println("MasterCommunicator -------- > MasterProcessor : " + value);
            values.add(value);
            StringBuffer sb = new StringBuffer();
            for (int i = 0; i < values.size(); i++) {
                sb.append(values.get(i) + "\t");
            }
            try {
                FileWriter fstream = new FileWriter("C:/workspace/gen/Readings.txt");
                BufferedWriter out = new BufferedWriter(fstream);
                out.write(sb.toString());
                out.close();
            } catch (Exception e) {
                // TODO: handle exception
            }
            int reading = ((Integer) value).intValue();
            if (reading >= minPressure) {
                deleteObservers();
                addObserver(SensorSystem.getMasterCommunicator());
                setChanged();
                notifyObservers("safe");
                //System.out.println("Pressure is Safe ");
            } else if (reading < minPressure) {
                deleteObservers();
                addObserver(SensorSystem.getMasterCommunicator());
                setChanged();
                notifyObservers("increase");
            }
        }
    }
}
Motor.java

import java.util.Observable;
import java.util.Observer;

public class Motor extends Observable implements Observer {

    public Observer getSensor() {
        return sensor;
    }

    public void setSensor(Observer sensor) {
        this.sensor = sensor;
    }

    Observer sensor;

    public void update(Observable rtuProcessor, Object value) {
        System.out.println("RtuProcessor -------- > Motor : "+ value);
        String strValue = (String) value;
        if (strValue.equals("increase")) {
            addObserver(sensor);
            System.out.println("Activating motor");
            setChanged();
            notifyObservers("increase");
        }
    }
}
