DYNAMIC SUBSCRIPTION BASED MULTI TRANSPORT INTELLIGENT GATEWAY

MASTER OF ENGINEERING THESIS

SHILIN BAI

SUPERVISOR : DR. JOHN COLLINS

SCHOOL OF ENGINEERING

2011

Acknowledgements

In completing this research project I received tremendous help from a number of people.

First and foremost, I would like to thank my academic supervisor Dr. John Collins from AUT for granting me the opportunity to participate in this research. His invaluable continuous support, guidance and advice during the research will always be appreciated.

I wish to acknowledge all colleagues at iMonitor Research Ltd. (NZ) for their kindness and support in many aspects during the research. I would like to thank in particular my industrial supervisor Robin Alden from iMonitor who provided me with much of the programming help, encouragement, comments and feedback on all my progress.

Finally, thanks to all my friends and family for their support during the research studies.

Attestation of Authorship

' I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person, nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning. '

Shilin Bai

August 2011

Abstract

This research investigates a novel design for a software engine and associated data structures to implement a dynamically reconfigurable network gateway to integrate and control data flow among remote monitoring and control applications that use different communication protocols. The implementation of our software engine is divided into two main parsers. The "Protocol Parser" converts the protocol template written in XML to the internal C# objects. These objects are used as instructions to process the incoming data from various data links in the "Data Analyzer". The data content is examined and then transformed into packets appropriate for the user required data link. A testing protocol template file which contains all possible scenarios in the real protocol template has been successfully tested with the "Protocol Parser". Then data files, which simulate the real incoming data frames, were analyzed by the "Data Analyzer". Accurate values are read and delivered, based on the information provided by the parsed protocol template. The test results from the research show that our Linkable Fragment Format (LFF) can successfully implement a configurable system for managing the structure of a protocol template in XML.

Table of Contents

List of Figures

List of Abbreviations

ASCII	American Standard Code for Information Interchange
BOM	Behaviour-based Object Model
CBCB	Combined Broadcast and Content-Based
CBN	Content-Based Networking
CBR	Content Based Routing
CLR	Common Language Runtime
DNS	Domain Name Service
DOM	Document Object Model
DTD	Document Type Declaration
EOF	End of Frame
FCS	Frame Check Sequence
FFF	Flat Fields Format
GA	Generalized Automaton
GPIO	General Purpose Input / Output
GPRS	General Packet Radio Service
GPS	Global Positioning System
GROVE	Graph Representation of Property Value

HAL	Hardware Abstraction Layer
HAH	Hardware Access Header
HTML	HyperText Markup Language
IL	Intermediate Language
IP	Internet Protocol
KEPT	Key Element Parse Tracing
LFF	Linkable Fragments Format
MF	Micro Framework
NFF	Nested Fields Format
PAL	Platform Abstraction Layer
QoS	Quality of Service
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SOF	Start of Frame
WSN	Wireless Sensor Network
XESP	XML Element Sequence Pattern
XML	eXtensible Markup Language

Chapter 1 - Introduction

With today's technologies, when the capacities of different media allow storage of vast amounts of information and establishing communication between different systems to be much easier due to the development of standardized communication protocols and data formatting techniques, gathering large amounts of data from various systems is not the heroic venture it once was. The processing and redirecting of information of "interest" to the user within the amount of raw data collected has become of prime importance. We consider the term "Intelligent" to be fitting to describe the approach used in our gateway for gathering, transforming and delivering data from one source to another for the following reasons:

- The gateway knows *which* information it should collect
- It knows *when* and *how* to collect it
- It knows in *which way* the gathered information should be *transformed* so it will fit the data format required at the receiver
- It knows *where* it should send the information so it will be appropriately stored.

Logic that parses the received information uses XML templates that are stored in the gateway. Logic that transforms the data must obey the user's subscription rules. Logic that sends the data to the user is based on previously established standard communication protocols. The entire process is basically an intelligent decision making system that realizes the "interesting" data has appeared, collects it, analyses it, then transforms it according to the user's requirement and delivers it to the destination in correct format. This decision making process judges the direction of data flow not only by checking the header of the message, but also is able to look into the data content embedded within the message. Studies have been carried out in the area known as Content-Based Networking (CBN). A CBN is a communication network that features a new advanced communication model where messages are not given explicit destination addresses, and where the destinations of a message are determined by matching the content of the message against selection predicates declared by users.

Within this research, we focus on designing and implementing a flexible XML based protocol template and associated data structures to support the subscription regulated, content based data processing mechanism within an intelligent gateway facility. The XML protocol template will be converted into reconfigurable internal data structures by our software engine for data transformation and transportation purposes. In this research, we have reviewed the development history of the routing techniques and what other people have achieved for content-based networking. Then we give a brief introduction on the communication protocol used in our research followed by the associated work related to the creation of the XML based protocol template. XML based languages are the foundation of extensible frameworks, and their natural characteristics have led to their success. However, their complexity and self-descriptive nature, can incur significant performance overhead. This overhead can reduce the performance of XML processing in some cases. Although the rapid development of the performance of hardware will be helpful for speeding up the XML processing, we address the aspect of software improvements here.

We have implemented a "Registrar" database which acts like the Document Type Declaration (DTD) of an XML document. It defines the range of capability that the engine is able to handle. We have also implemented a number of C# activity structures which assist us with the template structure management work. Using the Linkable Fragments Format (LFF) structure introduced here, we are able to transform the protocol template structure into much more flexible and understandable shape. All our test results also show that the protocol template structure we have built no longer behaves as the traditional XML protocol document, but delivers the same outcome in a much simpler way.

The goal of this research was to develop an intelligent gateway that can be implemented in an embedded system. This research was carried out for a company that had additional requirements, including the use of the registrar and data tree structures described in this thesis.

Chapter 2 - Literature Review

Antonio Carzaniya [2] *et al.* proposed a routing scheme for content-based routing. They present a combined broadcast and content-based (CBCB) routing scheme for a content-based network. This scheme consists of a content-based layer superimposed over a traditional broadcast layer. The broadcast layer handles each message as a broadcast message, while the content-based layer prunes the broadcast distribution paths, limiting the propagation of each message to only the nodes that advertised predicates matching the message. To achieve this double layer routing scheme, the router has to run two different protocols: a broadcast routing protocol and a content-bused routing protocol. Then they focused the study on a “push-pull” mechanism that guarantees robust and timely propagation of a content-based routing protocol. The "push" mechanism is based on receiver advertisements. The "pull" mechanism is based on sender requests and update replies. Their research shows that content-based networking is a quite powerful scheme with network routing and traffic control applications. The simulation model created by Antonio Carzaniya *et al.* has been the subject of a large number of research projects and related to a number of advanced network services and distributed-system technologies, including IP multicast, other rendezvous based communication services such as the internet indirection infrastructure, intentional naming and distributed publish/subscribe systems.

Content-based routing differs significantly from traditional unicast and multicast communication, in that message is routed on the basis of their content rather than the IP address and port numbers of their destination. R. Chand and P.A. Felber [6] developed the XRoute content-based routing protocol for their XNet XML-based data dissemination system. Their protocol implements perfect routing, i.e. a message is routed only to the consumers that have registered a matching subscription. It takes advantage of subscription similarities to “aggregate” them in the routing tables, and hence minimize the space requirements and increase the filtering speed at the routers. Their evaluation result shows XRoute protocol: firstly, performs perfect routing of data in the network, i.e., when an event is published, all the consumers that are interested in that event, and only those, must receive it; secondly, achieves subscription aggregation which is able to minimize space and processing requirements at any nodes.

Informally, subscription aggregation is a mechanism that enables us to reduce the size of the routing tables by detecting and eliminating subscription redundancies. It is a key technique to scale to very large populations of consumers in a publish/subscribe system; thirdly, allows consumers to register and cancel subscriptions at any time. In particular, cancelling a subscription should leave the system in the same state as if the subscription were not registered in the first place

Silvia Bianchi [4] *et al.* created a DR-tree overlay, which uses R-tree based spatial filters to construct a peer-to-peer network optimized for selective dissemination of information. They have pointed out that an efficient publish/subscribe overlay should minimize the occurrence of false positives (a peer receiving an event that it is not interested in) and avoid false negatives (a peer failing to receive an event that it is interested in). False negatives are very critical to the consistency of the system. A straightforward approach for avoiding false positives and false negatives they have used is to organize the subscribers in a tree structure according to containment relationships. Another approach consists in building one containment tree per dimension and adding a subscription to each tree for which it specifies an attribute filter [7]. This solution tends to produce flat trees with high fan-out and generates a significant number of false positives. Silvia Bianchi *et al.* have improved both approaches by using bounded-degree height-balanced trees, while preserving the containment relationships that ensure accurate content dissemination. The evaluation result confirms the average false positives ratio is less than 5% and slightly decreases with the size of the subscription set with DR-tree overlay.

Content-based routing improves upon the existing Internet model by giving users the freedom to describe routing schemes in the application layer of the network packets. Content-based routers then inspect and interpret packet payloads and route packets according to the content of the packet. James Moscola [9] *et al.* implemented a reconfigurable architecture (Figure 1) for high-speed content-based routing which goes beyond simple pattern matching by implementing a parsing engine that defines the semantics of patterns that are parsed within the data stream.

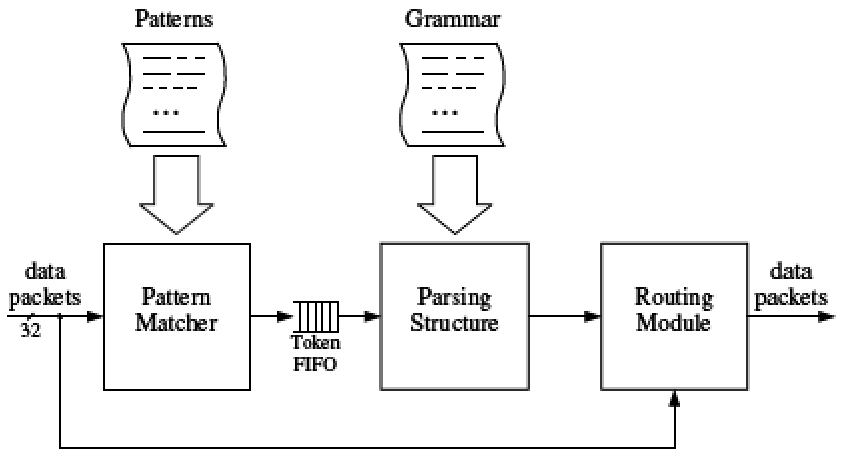


Figure 1 Reconfigurable Content-based Router Architecture

Yi-Min Wang [16] *et al.* developed a summary-based routing mechanism and introduced the notion of imprecise summaries to provide a trade-off between routing overhead and event traffic. They have also evaluated the effectiveness of the clustering techniques using realistic subscription and event distributions by comparing the result of Random Routing, R-tree based Routing and K-Mean Clustering [15].

Algorithm	Storage	Amortized Search Time	Total Traffic (Server Load)
Random	0	O(1)	High
Offline R-Tree	O(S)	O($log(S)$)	Lowest
Online R-Tree	O($N_s \cdot N_b$)	O($N_s \cdot N_b$)	Low
Offline K-Mean	O($S + N_s$)	O($I \cdot N_s$)	Medium
Online K-Mean	O(N_s)	O(N_s)	Medium

Few researches have focused on designing and reinforcing XML based protocols and templates, which could tremendously help to simplify the data extracting process, rather than the mechanism of extracting and routing events. There are many researches that have been carried out around how to parse and organize these XML based protocols and templates for different communication media. An efficient, well-structured XML based communication protocol will definitely help in understanding the incoming messages from a better perspective. Zhou, D. [17] presented the concept of Structure Encoding and the approaches to quickly identifying recurring structures, including one relying on a collision resistant hash function. He described in detail techniques to improve the performance of XML transmission, tokenization, parsing, and transformation by using Structure Encoding in a mobile environment. The structure of an XML document and the real content of the document are treated with different optimization approaches. His result implies us that a well-managed structural XML based template is definitely a key to improve the data parsing efficiency.

Takase, T. [11] *et al.* had a Document Object Model (DOM) implementation based on a hybrid data representation, literal XML and DOM object in order to overcome the large cost of parsing and serialization of XML messages in XML processing in web services. Their idea is to preserve the original literal XML representation of the XML data and reuse it when they serialize the data back to the literal XML representation. They have also introduced the idea of Partial XML parsing and partial XML tree construction (Figure 2) which leads to many further investigations around the creation of an XML fundamental module and is also being researched within our project.

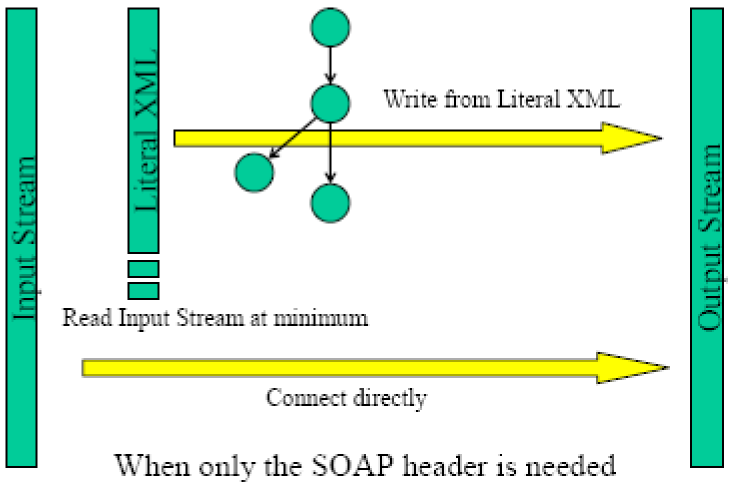


Figure 2 Hybrid between stream and object model

Lelli, F. [10] *et al.* investigated the limitations of XML for high performance and high-interactive distributed computing and introduced a new parser, the Cache Parser, which uses a cache to reduce the parsing time on the sender and receiver side, by reusing information related to previously parsed documents/messages. A common trade-off in computing is between the need of universality and performance, and this is particularly true when Web Services must be exploited to design a system in which both high performance and Quality of Service (QoS) requirements are mandatory. In particular, the features that make XML communication inefficient regard the primarily ASCII format of XML, and the verbosity of XML, due to the need of expressing tags and attributes besides the true information content. So speeding up the parsing algorithm should have a big impact on the total communication time, by largely reducing overheads. Lelli, F *et al.* are reducing the overheads on the receiver side, where the task of a parser is to deserialize the message by checking whether it conforms to the DTD/Xschema syntax, and extracting data from the textual XML representation. The research has shown that each subsystem routinely exchanges information by using very similar XML formatted messages. The exchanged XML information often has the same "structure", i.e. not only the same DTD/Xschema syntax, but also the same particular syntactic tree. When their system parses a new XML document, it first tries whether the structure matches an already know structure. This is quickly carried out by testing a document checksum. If the result is shown as positive, the document will be parsed with a fast algorithm that exploits the stored knowledge on the document syntactic tree. Otherwise, the document will be analyzed by using a standard parser, and a new entry will be created to store the syntactic tree of the new document. However, not all the structures of XML documents will be 100% the same. Their Cache Parser may speed up the parsing performance when it is dealing with an ideal XML document but will certainly encounter problems in the real world.

A similar idea was presented in Toshiro Takase [12] *et al.* as well. They proposed a novel mechanism for efficiently processing similar XML documents. Given a new XML document as a byte sequence, the XML parser normally avoids syntactic analysis but simply matches the document with previously processed ones, reusing those results. The parser is adaptive since it partially parses and then remembers XML document fragments that it has not met before.

A number of techniques have been developed to improve the performance of XML parsing, ranging from the schema-specific model to the streaming-based model to the hardware acceleration. Zhenghong Gao [16] *et al.* introduced a compiler-based approach for schema specific parsing that provides a flexible framework for implementing high-performance, schema specific parsers. Their approach is centred on an intermediate representation called the Generalized Automaton (GA). The GA serves a purpose similar to that of an intermediate language in a programming language compiler. The processing of XML in an application can be divided into three stages:

1) well-formedness to addresses whether or not the XML document is well-formed.
2) validity to addresses whether or not the structure is a valid instance of a given schema.
3) application to make sure the application actually uses the data in the XML.

We can implement the process in three ways listed in Figure 3.

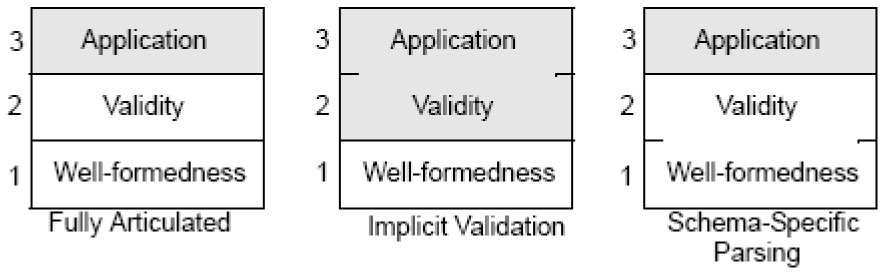


Figure 3 Ways of Processing an XML Application

If the stages are fully articulated, a general XML parser parses the XML into some kind of data structure representation of the XML. A validation pass is then made over the XML. The XML data structures are then presented to the application. The first two stages are usually packaged together into what is known as a validating parser. However, a common perception is that XML parsing is slow, and that XML validation is even slower. Thus, some applications do not implement the complete three-stage division. Instead, a general XML parser passes unvalidated XML to the application, which then implicitly validates it by error checking and possibly exception handling. Essentially, the second and third stages have been merged into one, as shown in Figure 3. Zhenghong Gao [16] *et al.* suggested the XML schemas may contain information can actually speed-up the lexical analysis and parsing of XML documents and Schema-specific parsing has been shown to be an effective technique for speeding the parsing of XML.

Gong Li [8] *et al.* had proposed a processing model for storing and building XML document in data transfer between XML and relational data source. In this model (Figure 4), a XML document is parsed and its elements are stored in a single table of database instead. It is not necessary to read the nodes according to their hierarchical structure, thus leveraging the workload of DOM building to memory by the algorithm called Tree-Branch inter growth. They introduced the idea of NodeTree, SortTree and LostTree and processing the model of the algorithm as shown in Figure 4 with Tree-Branch symbiosis algorithm described in Figure 5. However, the idea of the three tree structure and the algorithms of managing those three trees are over complicated. The work is done in memory and the cost is almost the same with the traditional methods in the precondition of high performance. Thus, higher cost would limit the popularity of their processing model.

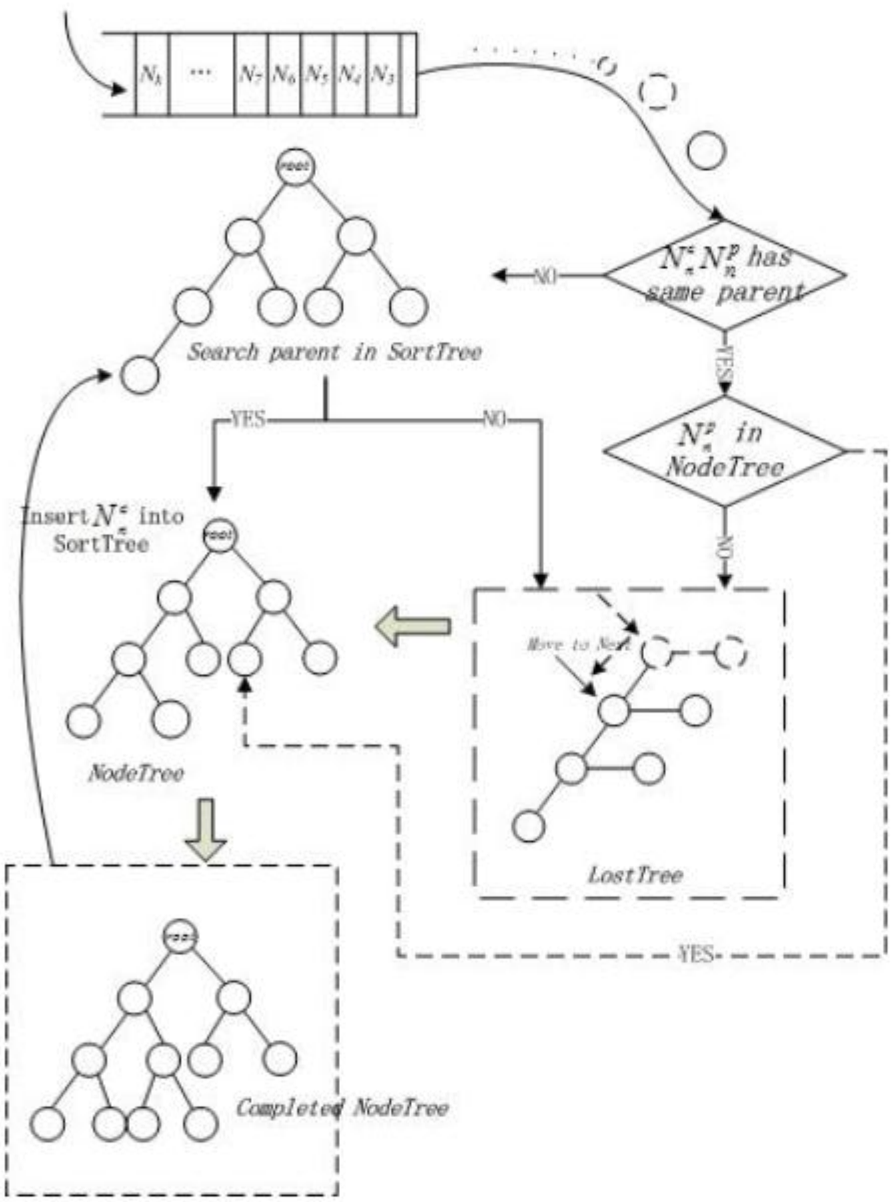

N_k ··· N_7 N_6 N_5 N_4 N_3
$N_n^c N_n^p$ has same parent
NO
YES
N_n^p in NodeTree
NO
YES
Search parent in SortTree
YES
NO
Insert N_n^c into SortTree
root
NodeTree
Move to Next
LostTree
Completed NodeTree

Figure 4 Processing model of the algorithm

```
Step1: Insert nodes into the NodeTree and LostTree
1.1 For each $N_n^c$, $N_n^p$, $N_l$ (H),$N_l^c$, $P_l^c$, $P_n^c$ set NULL
1.2 $N_n$ (H).nodeid=0;
1.3 $N_n^c$=RecordNode(recordset of data)
1.4 While $N_n^c$!=NULL
      1.4.1 if $N_n^c$.parentid == $N_n^p$.parentid
              {set $N_n^c$ and $N_n^p$ belong to a same parent
              Set $N_n^c$ as the brother of $N_n^p$ }
               else ($N_n^c$.parentid!=$N_n^p$.parentid)
               {search for parent of $N_n^c$ in SortTree}
      1.4.2 if find out the parent of $N_n^c$
               {set $N_n^c$ as the firstleftchild of its parent }
               else($N_n^c$'s parent is not NULL)
                   if $N_l$ (H)==NULL
                            {Initial $N_l$ (H)and$N_l^c$}
                  Set $N_n^c$ as the firstleftchild of $N_l^c$
               $P_n^c$ point to $N_l^c$
1.5 $N_n^p$ = $N_n^c$//set the current node as previous node
-----------------------------------------------
Step2: Insert nodes into the SortTree
2.1 {A typical recursive algorithm for building binary
search tree according to the "Nodeid" of a node.}
-----------------------------------------------
Step3:Add branch in LostTree to NodeTree
3.1 if ($N_l$ (H) is not NULL)
     { $P_l^c$ points to $N_l^c$
     set $N_l^c$ as the firstleftchild of $N_l$ (H)}
3.2 while ($N_l^c$!= NULL)
      If( find out the parent of $N_l^c$ in NodeTree)
       3.2.1addBranch to Tree:
             set $N_l^c$ as the firstleftchild of the parent
             while (the brother of $N_l^c$ is not NULL)
               {For each brother set the same parent}
            3.2.2addBranch to SortTree
      {The algorithm is same with step 2.}
            else
            {$P_l^c$ move to next node in LostTree}
```

Figure 5 The algorithm of Tree-Branch symbiosis

There has been a lot of discussion on the performance issue of a parallel XML parsing technique. Because of the semi-structured nature of XML, they were obliged to divide the data into well-formed XML chunks and then parse these chunks parallel. The division process is named as preparsing. As the preparsing is serial, it becomes the bottleneck of parallel XML parsing. Xiaosong Li [14] *et al.* addressed parallel XML parsing by Key Element Parse Tracing (KEPT) method (Figure 6) which parallelizes the preparsing and parsing process at element level. The method remolds the preparsing as a key element extracting process and schedules the processing of key elements in the framework of KEPT. Then the parsing process is parallelized as a whole.

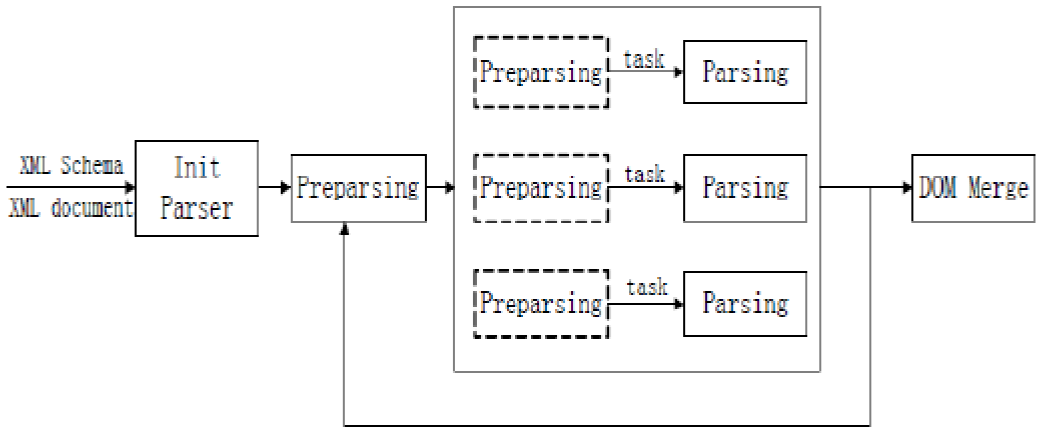


Figure 6 KEPT Architecture

Even though the scalability of XML parsing is enhanced by KEPT architecture, the preparsing stage as a whole is still serial. Possible techniques for quick positioning tags will obviously improve the performance of preparsing.

Boshi SUN [5] *et al.* presented a method for XML incremental validation based on simplified XML element sequence patterns, ensuring the XML document still conforms to the constraints established by the XML Schema. XML documents considered to be well-formed consist of declarations, elements, comments, processing instructions and other entities, marked as tags. The XML element tree is a conceptual model for XML documents and is used for abstracting XML documents as a tree model. XML element sequence pattern (XESP) is an XML data model composed of XML elements and the relationships among them. The evaluation result from their research also shows that a structural summary of the XML document based on XESP performs better than that of tree pattern extraction in the XML data model.

However, there has been little research on investigating the fundamental structure and characteristics of the underlying modules of the protocol template based on XML. How to internally combine these basic modules and make them to be able to talk with each other (linkable modules) obviously becomes important. DOM is a widely used data model for memory representation of the documents, but had shown some shortcomings in parsing and representing large or very large documents. Wasting resources and processing time overhead are results of applying large XML documents to the DOM parser. SGML is the XML parent. It was using Graph Representation of Property Value (GROVE) as its abstract data model to solve the problems that were encountered in the SGML family and as a standardized data model that represents the information contained within the SGML document. Yasmin Anwar [3] *et al.* had built the GROVE (Figure 7) by a grove builder. The research also delivered acceptable results of the memory overhead and the time required for processing an xml file.

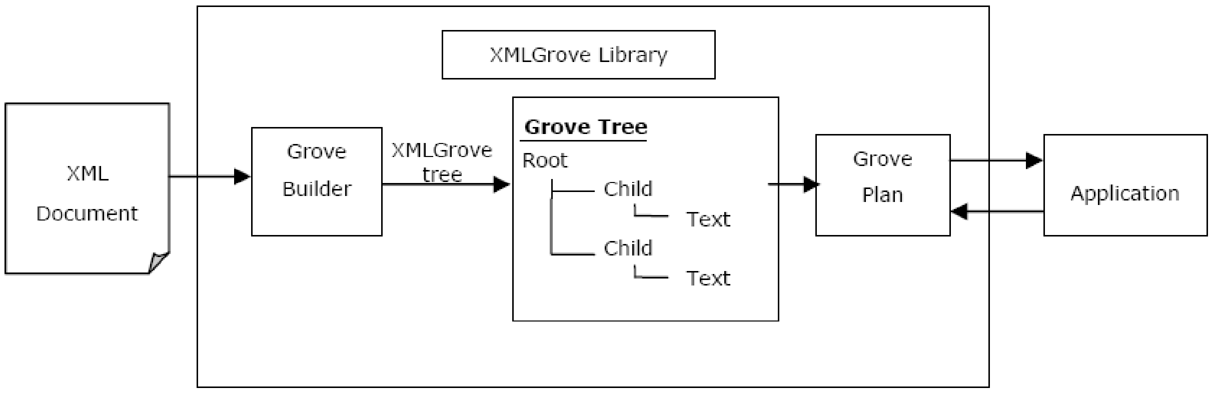


Figure 7 GROVE main components

XiaoLin Zhang [13] *et al.* introduced a new compression technology method to compress and decompress XML data stream. It first gets the structure data through analysing and parsing the

XML Schema, then encodes it with dynamic Huffman encoding, and finally completes the compression and decompression of XML data stream in real time. Using dynamic Huffman to encode the event sequence which is fit for the character of the data stream, resolves the question of transmission cost, such as XML data stream's bandwidth problem at the process of transmission.

Ajay Mane [1] *et al.* focused on parsing and creating the XML representation of spatial data. The spatial data that conforms to a DTD is "wrapped" in XML documents and represented graphically by parsing the XML document. The significant feature achieved in their work is that the graphical view of spatial data is generated dynamically. It is no longer a static image of a map that is loaded, but a graphical view is generated dynamically depending on the XML file that stores the spatial data information. This eliminates the necessity to store spatial data in the form of static images. But there are a lot of assumptions and constraints made in designing his DTD. The user has to load spatial data that conforms to the DTD and therefore the XML is very limited to use.

Chapter 3 - Research Background

3.1 Company Background

Global monitoring and control via the internet or cellular phone network is now being installed into food cool stores and into the transportation industry that supplies them, so that companies know exactly what is happening to their valuable product, where that product is, and most importantly, receive warnings of potential problems all along the delivery chain. And it doesn’t end with just food. "iMonitor Research Limited" [18] is a New Zealand company that has developed wireless technology that enables people to monitor and control situation at remote locations in real time. iMonitor develops customised solutions for virtually any product or equipment that needs monitoring and control. Their major product ranges from cell modem to locator are as follows.

The Cell Modem

This cell modem is one of the smartest and strongest available in the market. It contains a large number of features and transmits information via the standard GPRS cellular network and displays it on a cell phone or internet connection. The modem accepts a range of multiple input transmitters such as temperature, vibration, gas and models can also include onboard GPS. It operates on a range of supply voltages and battery back-up is standard on all models.

Product Monitor

The product monitor is a battery powered radio probe. The spike on the end of the probe is inserted into any product and measures its core temperature with great accuracy. In addition to this, it also measures surrounding air temperature and humidity. It will measure accurately temperatures from -40°C to 70°C. Two standard replaceable AA batteries will deliver up to 12 months of operation.

The Locator

Several locators are placed around the walls and ceiling of a storage area or cool store and measure temperature and humidity. Each locator collects information from up to twenty nearby product monitors. This information is then transmitted via radio to a data collection point for logging and display on a computer, or forwarded via an internet connection to your premises.

Our research investigation with iMonitor is carried out on the remote monitoring and control applications. The "Intelligent Gateway" facility and its associated software (Figure 8) to be implemented are responsible for collecting data, making suitable modification to the data according to user's subscription rules and publishing the filtered data to the interested end user. This entire process no longer uses the traditional routing mechanism such as IP routing, but complies with an advanced routing technique known as Content-based Routing.

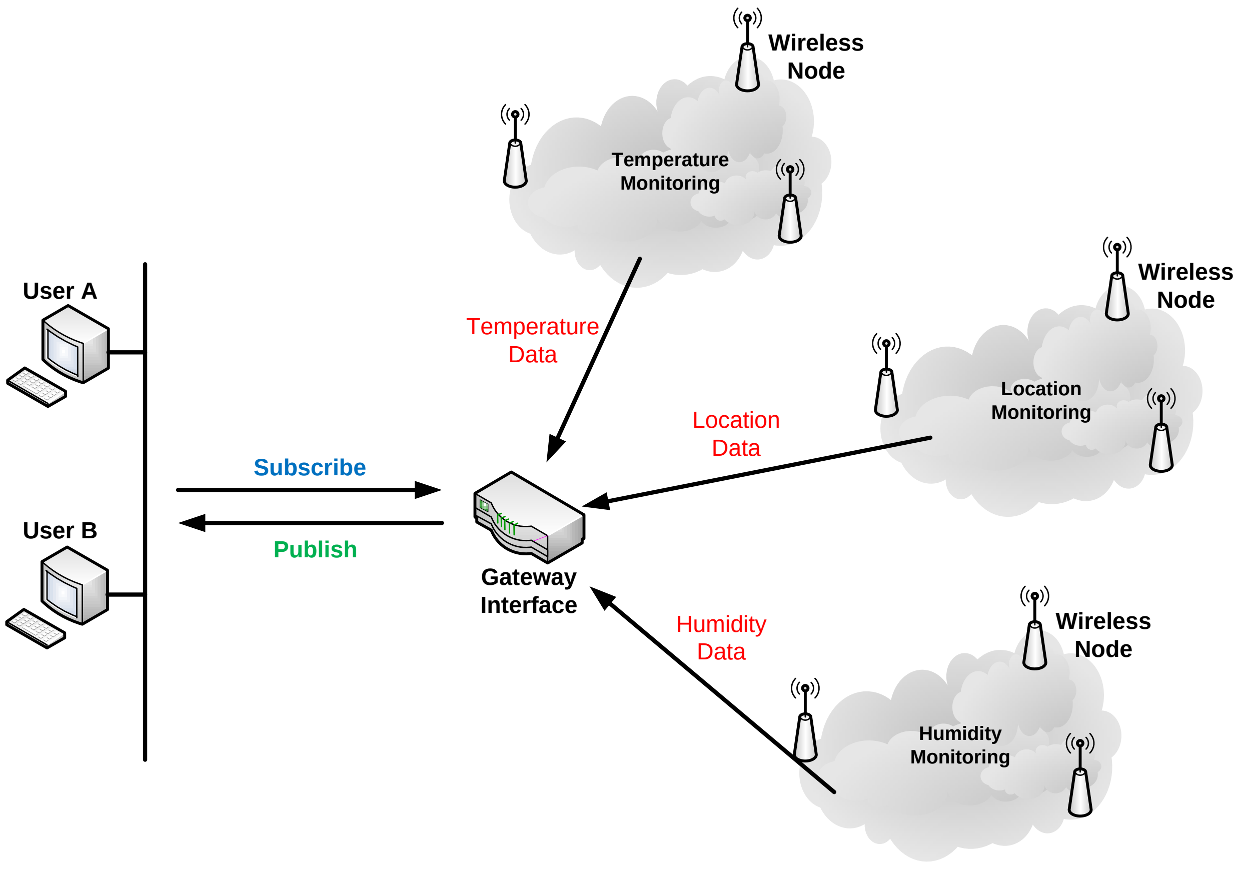


Figure 8 Remote Monitoring and Control Application

3.2 Software and Hardware Development Environment

3.2.1 .Net Micro Framework

The .NET Micro Framework (.NET MF) is a new execution model for extremely small, resource-constrained devices. It brings the advantages of .NET -- the security and reliability of managed code, the ease of development in Visual Studio, and the power of the C# language and the .NET libraries -- to a smaller class of device than Microsoft has ever targeted before. Another characteristic of .NET, often overlooked because most .NET applications are developed on and run on some flavour of Windows, is hardware-independence. Traditionally, embedded software has been tightly coupled to the hardware on which it runs. In .NET, however, applications are compiled to an intermediate language (IL) rather than to machine code, then are executed by a virtual machine called the Common Language Runtime (CLR). The .NET MF not only inherits the concept of the CLR, it incorporates two abstraction layers -- the hardware abstraction layer (HAL) and the platform abstraction layer (PAL) that help to further isolate applications from the hardware (Figure 9). Separating the hardware-dependent code from the application code allows the .NET MF to provide great flexibility in hardware selection and increases potential for code reuse from one device to the next.

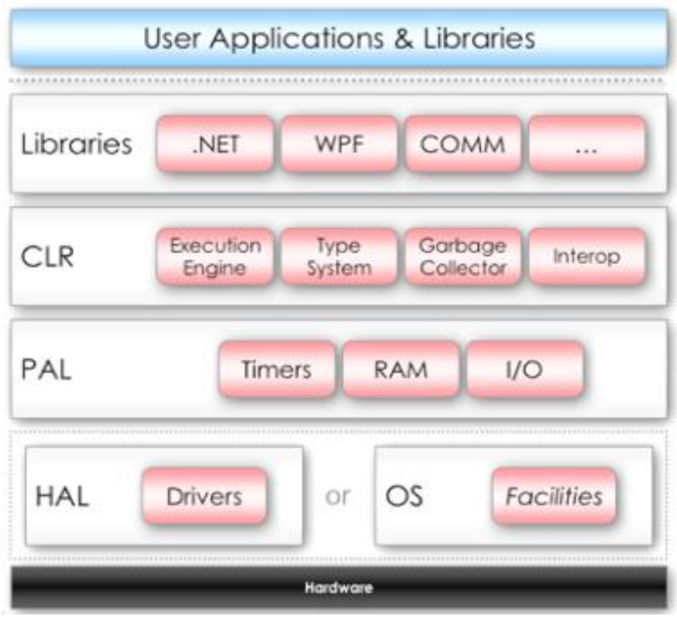


Figure 9 .NET Micro Framework Architecture

Because .NET MF allows us to develop C# application for small footprint devices and this research project is potentially going to be implemented on a Tahoe II board [19] which has already got .NET MF running, then all we have to do is simply focus on the application part of our system. Using the familiar Microsoft Visual Studio development suite, our C# applications can be developed to directly control hardware I/O. .NET MF also provides tight Visual Studio integration (including the ability to deploy managed code to a device and debug it while it's running there) and an extensible emulator that lets us run and test our embedded applications right on our PC.

3.2.2 Tahoe Development Kit

The Tahoe-II board (Figure 10) features a 3.5” touch-screen LCD, wired and wireless networking, USB function for interfacing to PCs, an accelerometer for innovative sensing and user interface applications. When you need to prototype a new device, the Tahoe-II has easy access to an array of expansion options; including serial ports, I2C, SPI and plenty of GPIO.

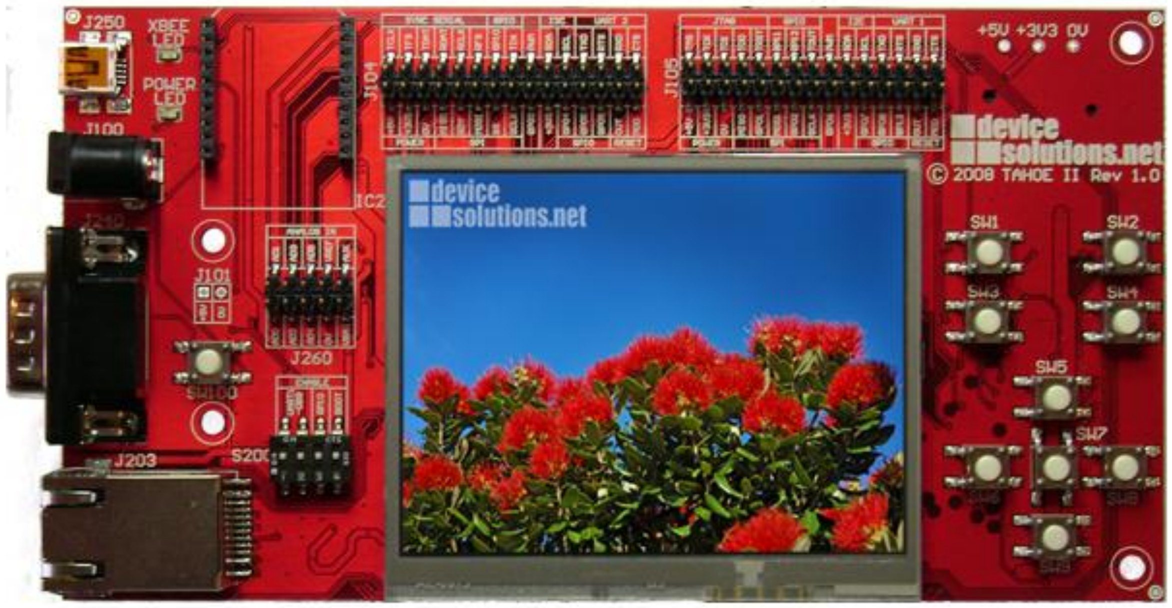


Figure 10 Tahoe-II Development Board

- Meridian CPU (ARM920 @ 100MHz, 4Mbytes Flash, 8Mbytes SDRAM)
- 3.5” Landscape TFT LCD with touch-screen
- 9 user input buttons
- RS232 serial (DB9)
- USB Function
- Ethernet
- Accelerometer, with support for event notification including free-fall detection
- SD Card interface
- Temperature sensor and 2x ADC channels
- Interface for XBee wireless module (and additional ADC channels if fitted)
- PWM output
- Expansion connectors that expose GPIO, I2C, SPI and UART signals

The Tahoe-II is built around the Meridian CPU (Figure 11), while features a Freescale i.MXS ARM9 processor, 4Mbytes of Flash, 8Mbytes of RAM. Moving from prototype to production is simple with the Meridian CPU.

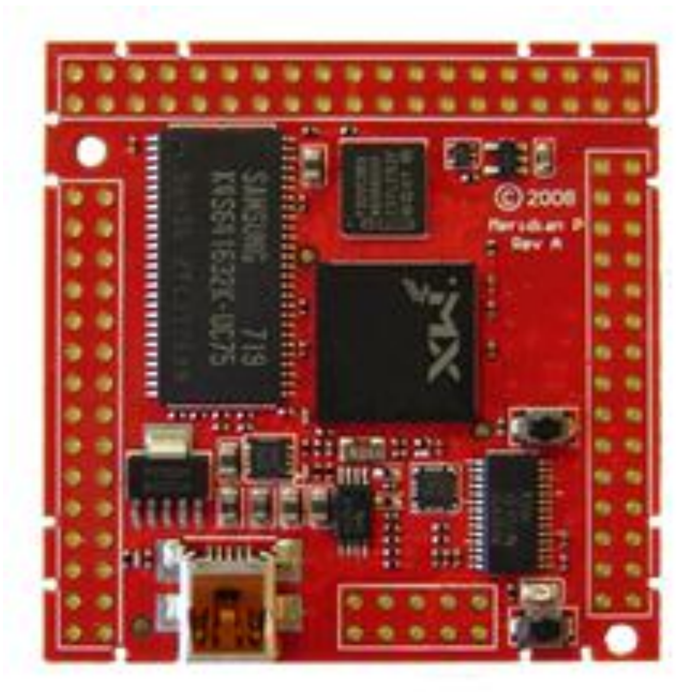


Figure 11 Meridian CPU

- Freescale i.MXS ARM920T at 100MHz
- 8MBytes SDRAM
- 4MBytes Flash
- USB Function for application download and debug
- 2 Serial Ports: 1 logic level + 1 RS232 level
- SPI
- I2C
- PWM
- Ethernet available with optional add-on board
- TFT LCD controller
- Button and LED accessible from application code
- 27 GPIO pins - all can be configured as interrupts
- Power via 5V or 3.3V
- All signal available via 0.1" headers
- Expansion connectors compatible with Tahoe-II expansion connectors

The Tahoe development Kit (Figure 12) incorporates the Meridian Processor Module onto an extensible I/O board. The Tahoe I/O board provides an ideal platform for developing applications that require a display and enhanced connectivity. Features include; RS232 serial, JTAG, switch inputs etc.

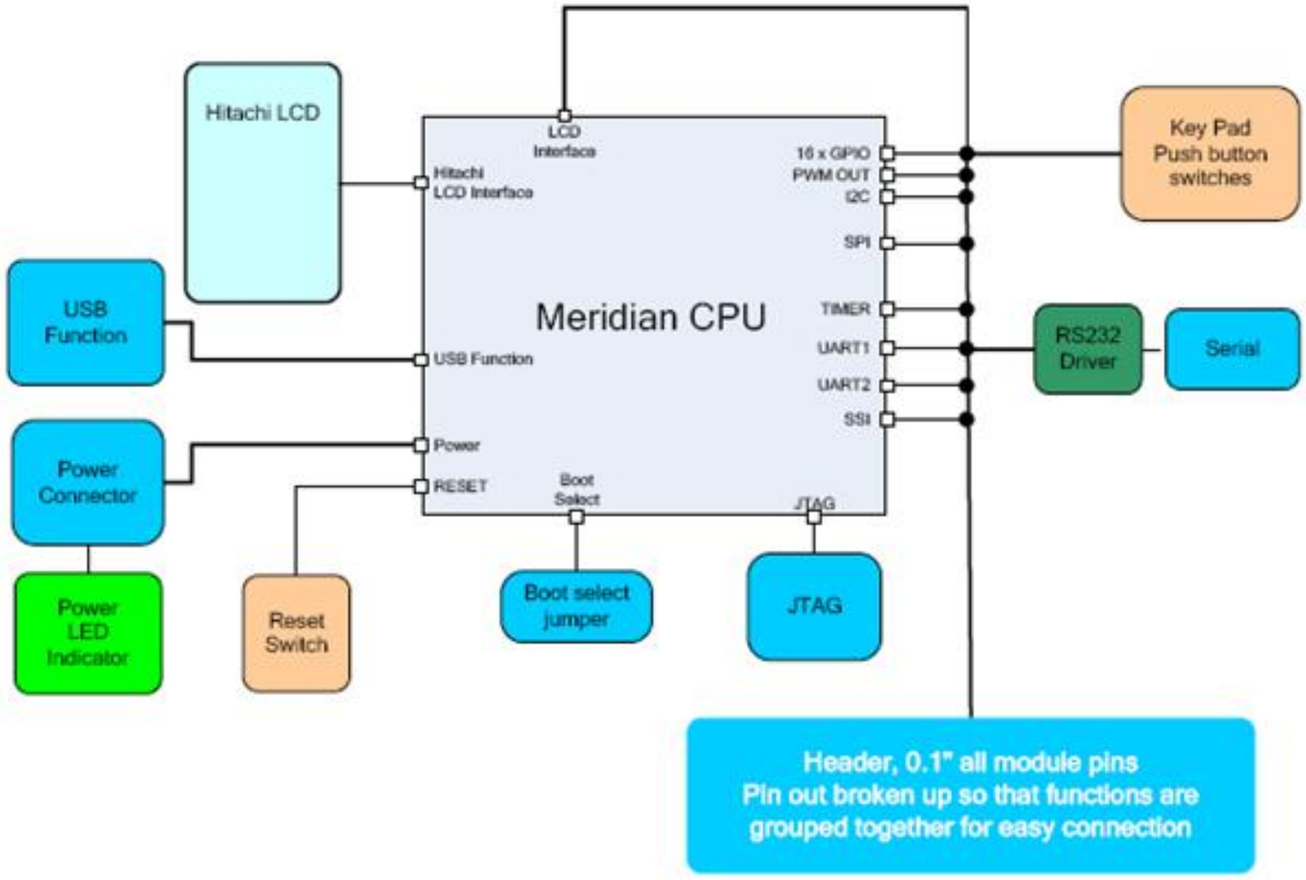


Figure 12 Tahoe Development Kit Schematic Drawing

3.3 Traditional Routing VS Content-based Routing

Routing of data from one place to another is one of the most basic and common problem facing any networked software application. Routing is defined as getting messages successfully delivered to the correct services. This routing can take many forms, such as email being sent to the correct recipient or network traffic being routed around the globe based on system names defined in Domain Name System (DNS).

3.3.1 Traditional Routing

In earlier days, there are many kinds of routing algorithms and techniques. The most widely used technique is called IP routing. Routing is accomplished by analyzing the message's header (i.e. IP address) and the message is simply forwarded to the specified destination by using the information retrieved from the header. IP Routing is a general term for the set of protocols that determine the path that data follows in order to travel across multiple network channels from the source to the destination. Figure 13 illustrates the IP routing mechanism. Data is routed from its source to its destination through a series of routers, and across multiple network channels. The IP Routing protocols enable routers to build up a forwarding table that correlates final destinations with next hop addresses. When an IP packet is to be forwarded, a router uses its forwarding table to determine the next hop for the packet's destination (based on the destination IP address in the IP packet header), and forwards the packet appropriately. The next router repeats this process using its own forwarding table, and so on until the packet reaches its destination. At each stage, the IP address in the packet header is sufficient information to determine the next hop.

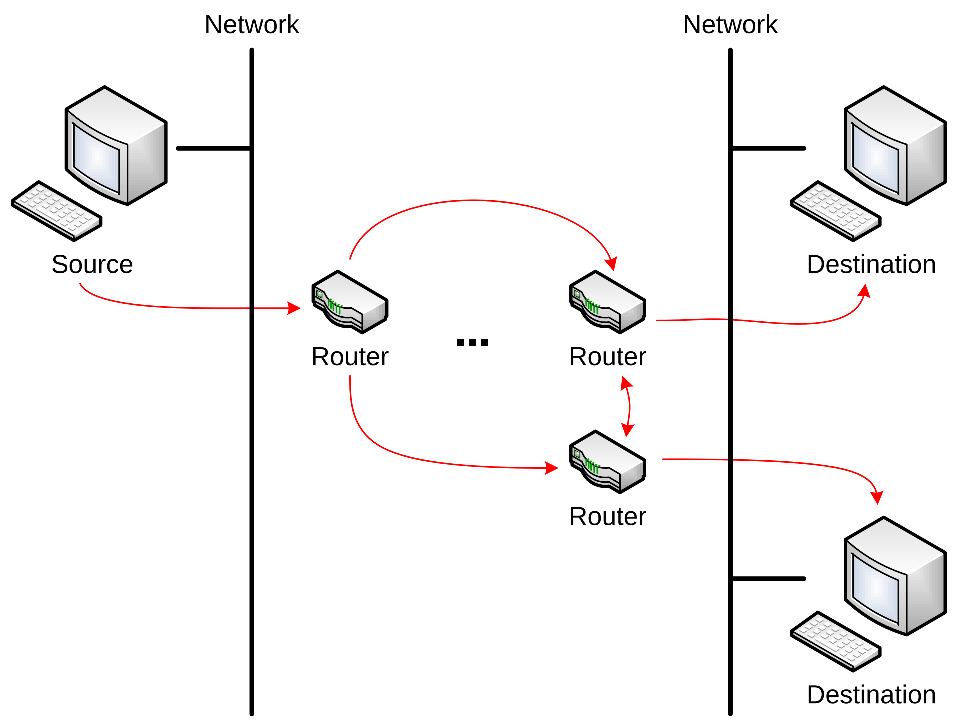


Figure 13 IP Routing

3.3.2 Content-based Routing

However, due to the tremendous growth on network technology, people started to be concerned more about the efficiency side of sending and receiving data. Many applications requires to minimize the network traffic control in order to increase the data flow efficiency. The old straightforward routing technique is no longer able to keep up with the high efficiency and flexibility that people demand from data transmission networks nowadays. A new concept of routing algorithm that routes data based on the data itself has been pointed out and discussed during recent years.

This newly emerged routing concept has drawn more and more attention to both developers and users. Especially in Wireless Sensor Network (WSN), routing messages based on content is a significant functionality that helps make WSN more flexible. Content-based Routing (CBR) routes messages based on the actual content of the data itself, rather than by a destination specified by the data. CBR works by looking into a data's content and applying a set of rules to the content to determine the data's destination.

By freeing the sender from needing to know everything about where a message is headed, content-based routing provides a high degree of flexibility and adaptability to change. The rules for decision making are commonly set by the user and they are able to be altered dynamically for different routing decision purposes. CBR has two main advantages over other routing algorithms:

- The sender no longer needs to maintain the extra information that is required to specify the destination for the data, sent with the data every time. Therefore, the size of the packet sent across the network is reduced.
- The data received by the destination user is more accurate and useful. The needless data has already been discarded in the transmission media.

A publish/subscribe system (Figure 14) consists of publishers (Wireless Node, Smart Phone & PC) that generate messages (data) and subscribers (User A & User B) that register interests (subscription rules) in all future messages. This system, implemented as a CBR broker, is responsible for routing published messages to interested subscribers. The gateway receives different formats of data from different media sources (publishers). The users (subscribers) subscribe their interests to the gateway. The gateway makes decision based on matching the subscription given. Whenever an "interesting" match is confirmed, the message from the publisher is delivered to the subscribed user.

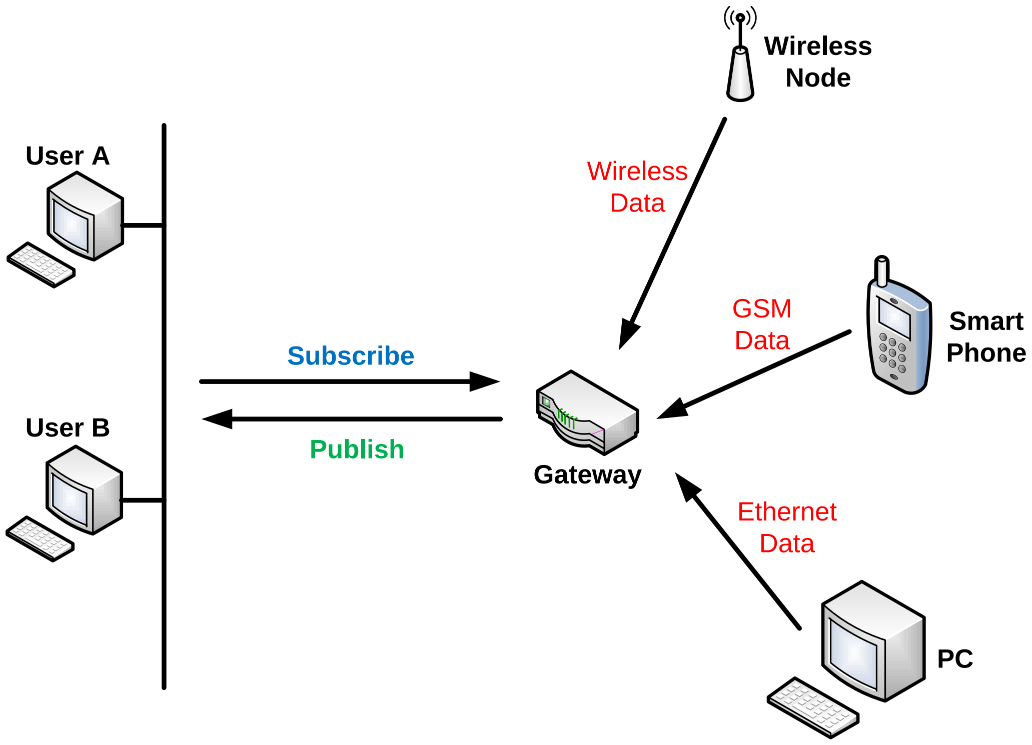


Figure 14 Publish/Subscribe System Example

3.4 eXtensible Markup Language (XML)

Before XML was developed, SGML and HTML were the most commonly used markup language worldwide. Standard Generalized Markup Language (SGML) provides an extremely powerful set of tools for the standardization of the markup syntax as well as its excellent language scalability. Therefore, it has been widely used for data indexing and data classification. However, the fatal flaw of SGML is its cost and language complexity. Several major browser vendors have clearly refused to support SGML. In contrast, HyperText Markup Language (HTML) is free and simple to use. It focuses on the description of the page forms which greatly enriched the home page visual ability and auditory effects. It has played an irreplaceable role to promote the rapid development of World Wide Web and network information exchange and knowledge dissemination. But HTML also has several fatal weaknesses. These weaknesses have seriously blocked the future application development to HTML. HTML is a specialized form of expression to describe the design of the home page, which neglects the internal information structure and its semantic description. It cannot meet the requirements of the growing number of information retrieval and archiving requirements nowadays. Increasingly bloated set of HTML tags and its loose structure of the syntax requirements make the document very hard to understand and organize. It leads to more and more complex design of the browser and reduces the efficiency of browsing time and space efficiency. XML is born as one such markup language which not only has the power of SGML and scalability, but also has the simplicity of HTML.

XML stands for eXtensible Markup Language. It is a self-descriptive markup language much like HTML. XML was created to structure, store, and transport information but not to display data. It has the following four advantages over SGML and HTML.

i) Good scalability. XML allows different users to develop their own unique set of tags according to their own needs. It does not require all browsers to handle thousands of markers and it also does not require a markup language to suit various fields of application.

ii) Separation of content and structure. As mentioned earlier, display of information in XML has been extracted from the information itself and placed in Document Type Declaration

(DTD). This expression change not only facilitates the alteration of document formation and easier data search, but also makes XML a good self-description language to describe the meaning of the information itself and even the relationship between them.

iii) Easy transmission of information between different systems. Many different systems often live among different enterprises and different departments. XML can be used for communication between different systems. It is an ideal internet media language.

Every simple XML document must contain four basic key components: Document Type Declaration, Encoding Declaration, XML Element/Tag, XML Attribute.

3.4.1 Document Type Declaration

XML provides a mechanism, the Document Type Declaration (DTD), to define constraints on the logical structure and to support the use of predefined storage units. A DTD can be declared either inline in the XML document or as an external reference. A well-formed XML document is valid if the document has an associated DTD declared inline in the XML document or as an external reference and that it complies with the constraints expressed in DTD. The DTD can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together. XML is always released as a version. Because there can be various versions, the first line that can be processed in an XML document must specify the version of XML you are using. At the time of this writing, the version of XML supported on the .NET Framework is 1.0. When creating an XML file, you should specify what version your file is conform with, especially if you are using a version higher than 1.0. For this reason, an XML file should start, in the top section, with a line known as an XML declaration. An example of such a line is:

```
<?xml version="1.0"?>
```

3.4.2 Encoding Declaration

Most of the characters used in the US English language are known as ASCII. These characters use a combination of 7 bits to create a symbol (because the computer can only recognize 8 bits, the last bit is left for other uses). Such an encoding is specified as UTF-8. There are other standards such as UTF-16 (for wide, 2-Byte, characters). To specify the encoding you are using, type encoding followed by the encoding scheme you are using, which must be assigned as a string. The encoding is specified in the first line.

```
<?xml version="1.0" encoding="utf-8" ?>
```

3.4.3 XML Element

An element in an XML document is an object that begins with a start-mark "<", contains a name in middle, and terminates with an end-mark ">". Based on this, the combination of a start-mark, the name, and the end-mark is called an XML element. A tag is just a generic name for a <element>. An opening tag looks like <element>, while a closing tag has a slash that is placed before the element's name: </element>. The real description/data lives in between. Every XML element declared must be properly closed with the exactly same name specified in the closing tag copied from the opening tag. The characters specified between "<" and ">" are case sensitive. <Element> and <element> are entirely two different tags. An Xml document is usually formed with many XML elements in orders as follows.

```
<University Id="AUT">
   <Engineering>
      <Mechanical></Mechanical>
      <Electrical></Electrical>
   </Engineering>
   <Science>
      <Chemistry></Chemistry>
   </Science>
   <Business>
      <Marketing></Marketing>
      <Accouting></Accouting>
   </Business>
</University>
```

3.4.4 XML Attributes

Attributes often provide extra information that is not a part of the data. You might think of an attribute as an adjective describing the element it is within. Attributes are formed in name=value pairs. Attribute values must always be " " quoted. Either single or double quotes can be used. Thus, in XML, you would never write <dog white /> - that would be incorrect. One way to think about it is to think of the most generic instance of the adjective you are using. If you're describing your "Dog" element as "big", "white", and "smart", then you should probably have three attributes: size, colour, and intelligence. Then you could have one <Dog colour="white" size="big" intelligence="smart"> and another element <Dog colour="calico" size="medium" intelligence="stupid">

```
<Dogs>
  <Dog colour="White", size="Big", intelligence="Smart">Max</Dog>
  <Dog colour="Calico", size="Medium", intelligence="Stupid">Jazz</Dog>
</Dogs>
```

3.4.5 XML Document as a Tree Structure

An example of a simple XML document is written below. This example shows us how to keep track of books in XML document format in a bookstore. The XML document is also illustrated as a tree diagram in Figure 15.

```
<!DOCTYPE bookstore [
  <!ELEMENT book (title,author,year,price)>
  <!ATTLIST book catagory CDATA #IMPLIED>
  <!ATTLIST title lang CDATA #IMPLIED>
]>

<bookstore>
  <book category="COOKING">
    <title lang="EN">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="SPORTS">...</book>
</bookstore>
```

<bookstore> is the highest level element which is known as the root element in this XML document. All <book> elements are one level lower than <bookstore> element and each of them

has four sub-elements: <title>, < author>, <year>, <price>. Each sub-element may or may not contain one or more associated attributes in order to characterize itself. In this example, <bookstore> root element contains two <book> elements as its children. Each <book> element differs from the other by the "category" attribute that describes which category the particular book falls into, i.e Cooking book or Sports book. Every <book> element contains four individual elements as its children: <title>,< author>,<year>,<price>. In between each of those element tags, there is text element which stores the real information embedded in this XML document. Again, <title> element has its own "lang" attribute which is for characterization purposes.

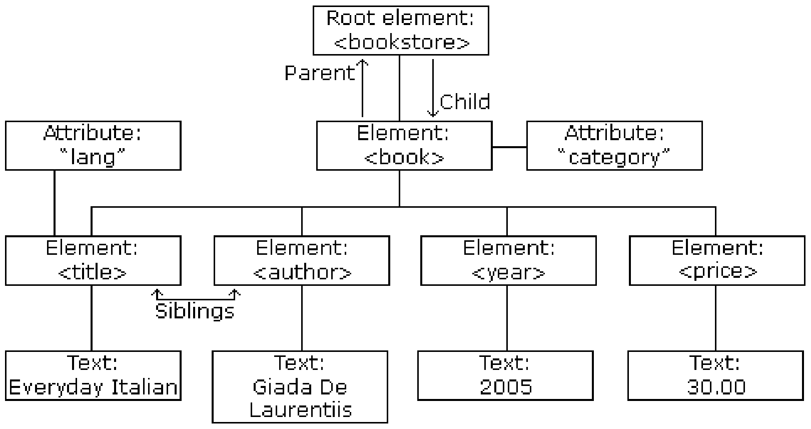


Figure 15 Tree Model for a Book Store

As can be seen from Figure 15, the example XML document can be entirely represented as an upside down tree structure. The structure should always only have one top root as the root element of the XML document, then it can be expanded as low as the XML document goes. The lower level we go, the more specific information we observe. An interesting phenomenon we immediately discover from this structure is the Parent & Children relationship as shown Figure 15. <bookstore> element acts as the parent of <book> element. <book> element obviously plays the role of child of <bookstore> element. All <book> elements treats each other as siblings. This structural relationship exists through the entire tree branches. We are able to use this relationship to perform an easy navigation between one branch and another anywhere on the tree.

3.5 "iMonitor" Protocol Structure Layout

iMonitor Research Limited has defined its own communication protocol for various remote monitoring and control applications. This protocol is mainly used for data collection from temperature, humidity and location sensors via Zigbee [20]. The structure of the protocol has been carefully studied and illustrated within this section.

3.5.1 Frame Payload

The Zigbee data receiver is responsible for collecting Zigbee data sent from different sensor devices and converts the received data to standard RS232 data format. i.e. Zigbee data in and RS232 data out. Then it feeds the RS232 data via its serial port to the software engine inside the intelligent gateway. One RS232 frame payload (Figure 16) represents one isolated data frame. The structure of this data frame is formed by five smaller sections which are named Start of Frame (SOF), Length, Payload, Frame Check Sequence (FCS) and End of Frame (EOF).

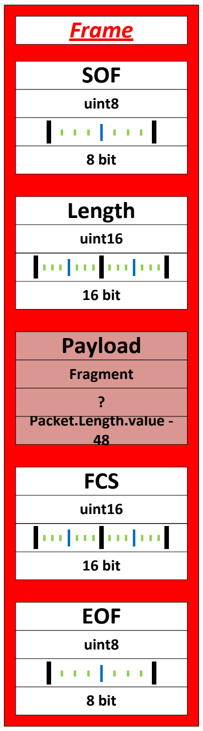


Figure 16 Standard RS232 Frame Structure

SOF - This is the Start of Frame flag. It stores an 8-bit value. The value of this section should always be 0x7E. This is used to mark the beginning of a data frame.

Length - The length section indicates the length of the frame between the SOF flag and EOF flag bytes there including itself. It stores an 16-bit value. The value from this section is used to read the entire frame.

Payload - This payload section usually contains the data which will be passed on to the iMonitor gateway hardware for analyzing purpose.

FCS - This is the Frame Check Sequence section. The FCS enables a high level of physical error control by allowing the integrity of the transmitted frame data to be verified. The sequence is first calculated by the transmitter using an algorithm based on the values of all the bits in the frame. The receiver then performs the same calculation on the received frame and compares its value to the value calculated by the transmitter.

EOF - This is the End of Frame flag. It stores an 8-bit value. The value of the EOF Flag is always 0xAA. This is used to mark the end of a frame.

3.5.2 Hardware Access Header

iMonitor gateway hardware receives the standard TCP/IP format data from the multiplexer. The gateway interface analyzes the data by looking into its hardware access header in the Payload of figure 16. The Hardware Access Header (HAH) is shown in Figure 17.

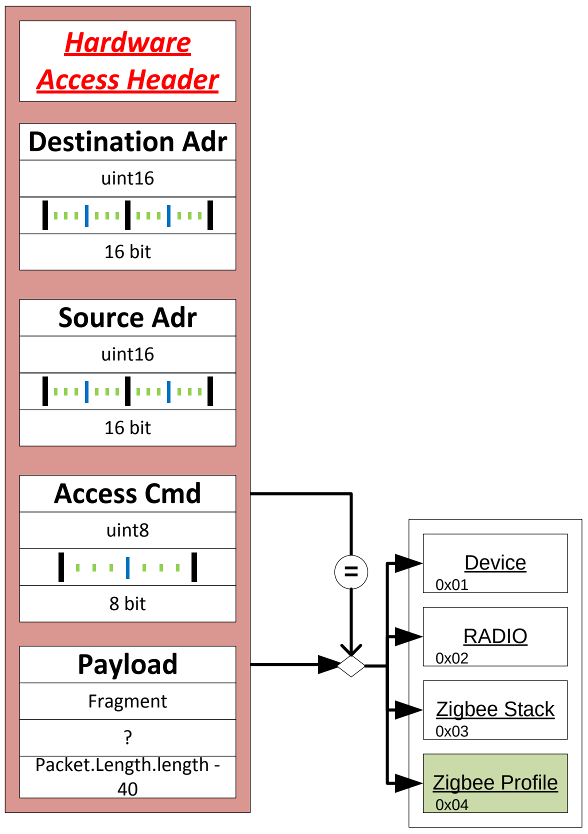


Figure 17 Hardware Access Header

Destination Address - This field specifies which Network the packet is being sent to. It stores a 16-bit value.

Source Address - The field specifies which Network the packet is originating from. It stores a 16-bit value.

Access Command - This field specifies the software destination or originator of the payload. It stores an 8-bit value.

Payload - The information contained in this Payload section differs according to the access command defined in Hardware Access Command. There are four standard types defined as shown in Figure 17, only Zigbee Profile (0x04) is associated within this research project.

3.5.3 Zigbee Profile

Zigbee Profile (Figure 18) may be a WSN Profile (0x0108) or a Coolstore Profile (0x0308) based on the value provided in Profile Id section. Under each profile, the value stored in the Frame Type section indicates the different types of information contained in the Payload section. Zigbee Cluster Library (ZCL) payload is common to both profiles, but may carry specific commands and cluster ID that belong to different profiles. Visualization payload only exists in the Coolstore profile and it contains 3D coordinate and sensor measurement values. The detailed description on ZCL payload and Visualization payload is given in section 3.6.4 and 3.6.5. Sensor Group and Alarms payload types are not supported yet and are not used in this research.

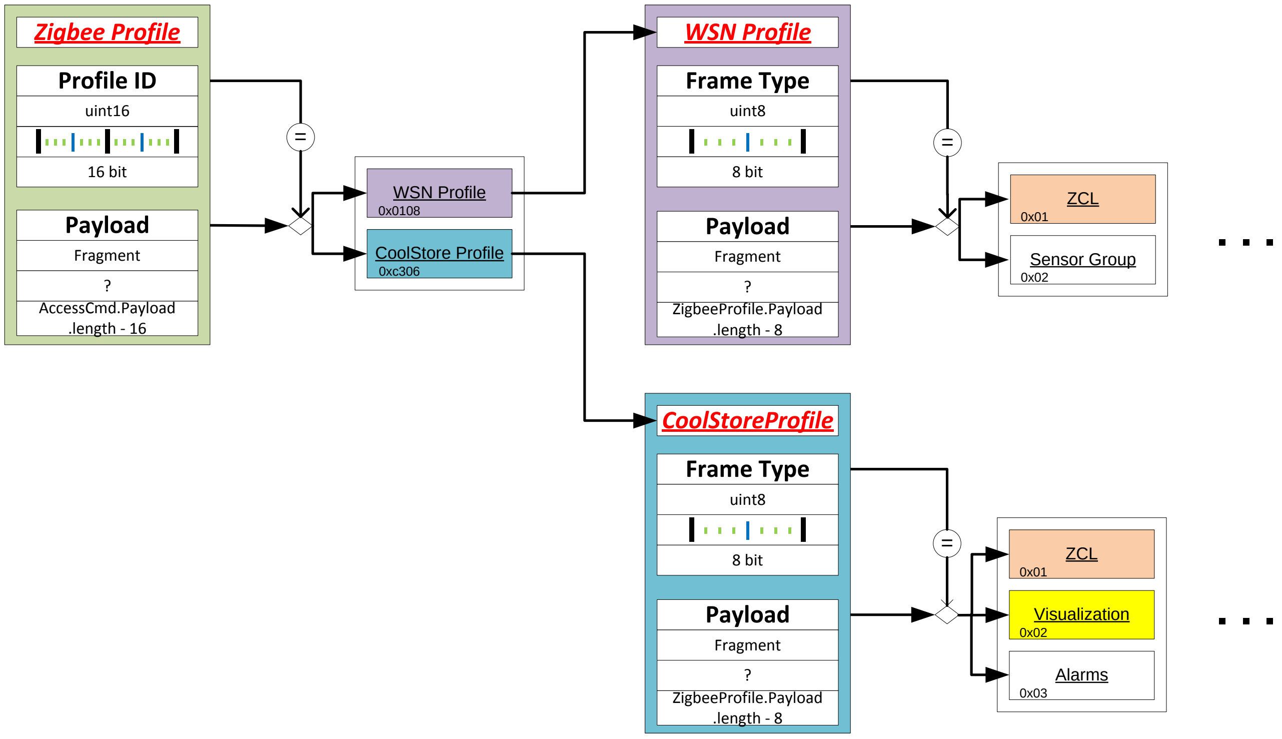


Figure 18 Zigbee Profile

3.5.4 ZCL Payload

Device Address - This section specifies the short address of the device within the network where the packet is being sent or originated from.

Endpoint - The end device that the packet is being sent to.

Cluster ID - This section defines what the payload is referring to, e.g. temperature, humidity or 3D coordinate.

Command - The command section represents the type of command being sent, e.g. write or read

Specific - The specific section represents whether the command is cluster specific.

Direction - The direction section specifies the client/server direction for this command. If the value is set to 1, the command is being sent from the server side of a cluster to the client side of a cluster. If the value is set to 0, the command is being sent from the client side of a cluster to the server side of a cluster.

DisableDefaultRsp - The disable default response section is 1-bit in length. If it is set to 0, the default response command will be returned as a response. If it is set to 1, the default response command will not be returned.

Manufacture Code - The manufacturer code section is 16-bits in length and specifies the Zigbee assigned manufacturer code for proprietary extensions to a profile. This field shall only be included in the ZCL frame if the manufacturer specific field of the frame control field is set to 1.

SeqNum - The transaction sequence number section is 8-bits in length and specifies an identification number for the transaction so that a response style command frame can be related to a request style command frame. The application object itself shall maintain an 8-bit counter that is copied into this field and incremented by one for each command sent. When a value of

0xff is reached, the next command shall restart the counter with a value of 0x00. The transaction sequence number section can be used by a controlling device, which may have issued multiple commands, so that it can match the incoming responses to the relevant command.

CmdLen - The command length section is 8-bits in length and specifies the cluster command payload Length.

CmdPayload - The command payload section has a variable length and contains information specific to individual command types. The maximum payload length for a given command is limited by the stack profile in use, in conjunction with the applicable cluster specification and application profile. Fragmentation will be used where available.

3.5.5 Visualization Payload

Product ID - The Product ID section specifies the type of device that sends the packet and therefore what sensors data it contains.

Network Address - The Network Address section specifies which Zigbee network the packet is being sent to or originated from.

IEEE Address - The IEEE Address section contains the devices fixed MAC address. This can be used as a serial number for the device as it will not change over time.

Device Address - The Device Address section specifies the short address of the device within the network the packet is being sent or originated from. The devices short network address may change over time.

Endpoint - The end device that the packet is being sent to or originated from.

Cluster ID - The cluster ID section specifies what packet is referring to, e.g. temperature, humidity or 3D coordinate.

Sensor Payload - The actual sensor data depends on what ID is specified in the Cluster ID section.

CHAPTER 4 - Implementation

XML based languages are the foundation of many extensible frameworks. Its natural characteristics have led to its success, however its complexity and self-descriptive nature can incur significant performance overhead. This overhead can reduce the performance of XML processing in some cases. Although the rapid improvement of the performance of hardware will be helpful for speeding up the XML processing, we address the aspect of software improvements here.

In this chapter, firstly we will give a brief description of the architecture of our "Construct" software engine. Then we will introduce the underlying class model which we have investigated during this research. After that, we will present the detailed implementation process of the key components investigated in our research. Finally, we will describe step by step how the XML Parser, Data Analyzer and their associated objects operate in our software engine.

4.1 Software Engine Architecture

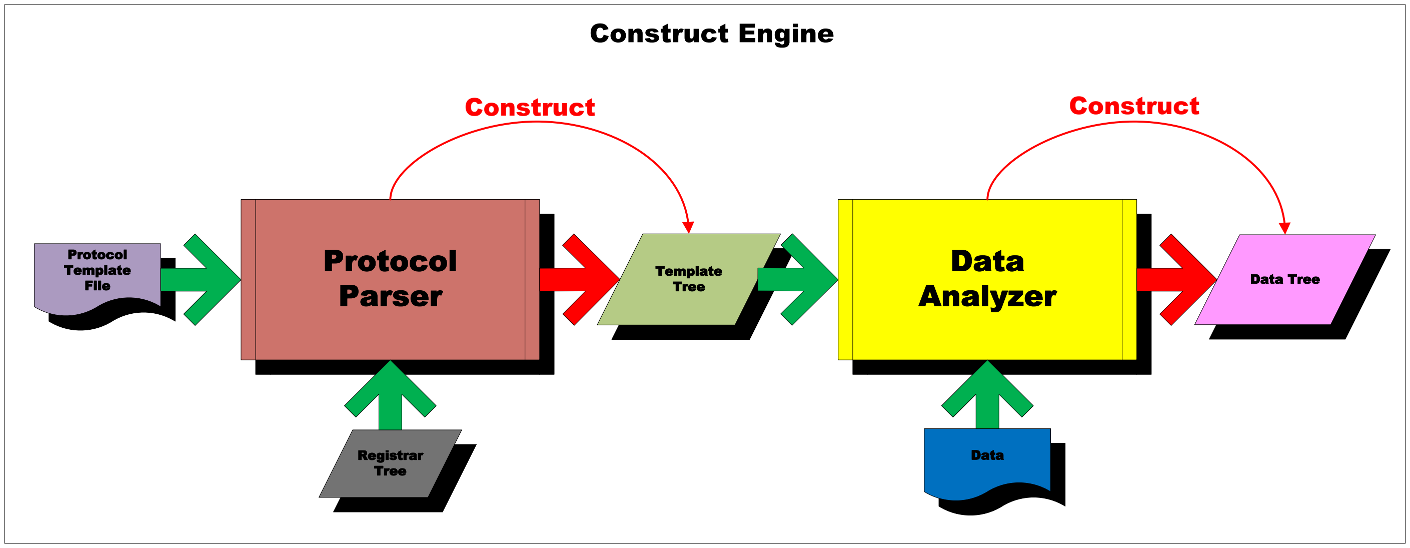


Figure 19 Software Engine Architecture

The software engine we have developed is called "Construct" (Figure 19). The core of this engine is formed from two main pieces. They are the "Protocol Parser" and "Data Analyzer". There are also a few other key components which play an important part in the XML parsing and data analyzing processes in the software engine. "Protocol Template File" is an XML-based text file which specifies the communication protocol structure used for data transfer. "Registrar Tree" acts as a database/reference which supports the protocol parsing mechanism. "Template Tree" is a parsed internal C# object model/tree which represents the structure of the protocol template file. The template tree guides the data analyzer engine as it processes the incoming data frames. "Data" refers to the incoming data frames. It can be in many forms such as Zigbee, Ethernet, GSM and others. The way the data analyzer handles the data is totally dependent on the given protocol template file. "Data Tree" is a tree constructed from the incoming data frames. The data tree has a similar structure to the template tree but also includes the data values stored in each individual field. Because this tree contains the actual data values, it will be used for the future decision making process within the intelligent routing facility. The detailed implementation of the protocol parser and data analyzer will be addressed in later sections.

4.2 Underlying Class Infrastructure

4.2.1 Interface Inheritance

Interfaces describe a group of related functionalities that can belong to any class. We can define an interface by using the "interface" keyword in red, as shown in the following example.

```
/// <summary>
/// Represents our lowest level of capability
/// </summary>
public interface IBehaviour
    : IHaveOwner
{
    // The Behaviour’s Actual Behaviour
    IBehaviour Actual { get; }

    // The Behaviour's Definition
    IBehaviourDefinition Def { get; }
}
```

An interface is a reference type object with no implementation. We can think of it as an abstract class with all the implementation stripped out and everything that is not public removed. Abstract classes are classes that cannot be instantiated. No properties or methods are actually coded in an interface, they are only declared. So the interface does not actually do anything, but only has a signature for interaction with other classes or interfaces. When a class implements an interface, the class provides an implementation for all of the members declared by the interface. The interface itself provides no functionality that a class can inherit in the way that base class functionality can be inherited. However, if a base class implements an interface, then a derived class inherits that implementation. The derived class is said to implement the interface implicitly. Classes implement interfaces in a manner similar to how classes inherit a base class, with two exceptions:

- A class can implement more than one interface.
- When a class implements an interface, it receives only the method names and signatures, because the interface itself contains no implementations.

Interfaces make for a much easier "code world" in which to navigate. Imagine if instead of learning how to drive a car and then being able to drive any car, we have to learn how to drive each instance of every car we get into. It would be really inefficient if after learning how to drive the Toyota we had to start all over again in order to figure out the BMW. A much more efficient way is to deal with the car's interface: the steering wheel, indicator signals, clutch, accelerator and brake. This way, no matter what is implemented on the backend of the interface, we don't really care because in the end it subscribes to the basic car contract and that is how we will deal with it through the interface. Following the car analogy, if we produce a component of a car, such as a car tyre, it would be good if the tyre could be used in multiple types of cars. This is crucial to the success of the business if we need to sell the tyres to multiple vendors and provide them with a basic manual on how to put it on and make it work. The customer does not need to know anything about how we have implemented the interfaces. If we make a change in the product, we still keep the contract the same (the interface) and our customers are still able to put on any new upgraded tyres we develop.

To make our code more reusable, we have provided interfaces in our software engine implementation. We can always add more signatures to an interface without breaking things, but if we alter the existing "hooks" into our implementation we lose one of the primary benefits of using the interface. Because this research project is developed using C# language, we can only inherit from one class but can implement any number of interfaces. Because of this inheritance limitation, we are very careful which class we choose to inherit from because we can only inherit from one class. This is a place where scalability is negatively impacted. Once we have an inheritance hierarchy set up, it is difficult to change. We can achieve the same results as inheritance through class composition and exposing the functionality of a wrapped object through an interface and our classes will not use up our "one shot" inheritance. Figure 20 shows the interface structure we have designed for our software engine.

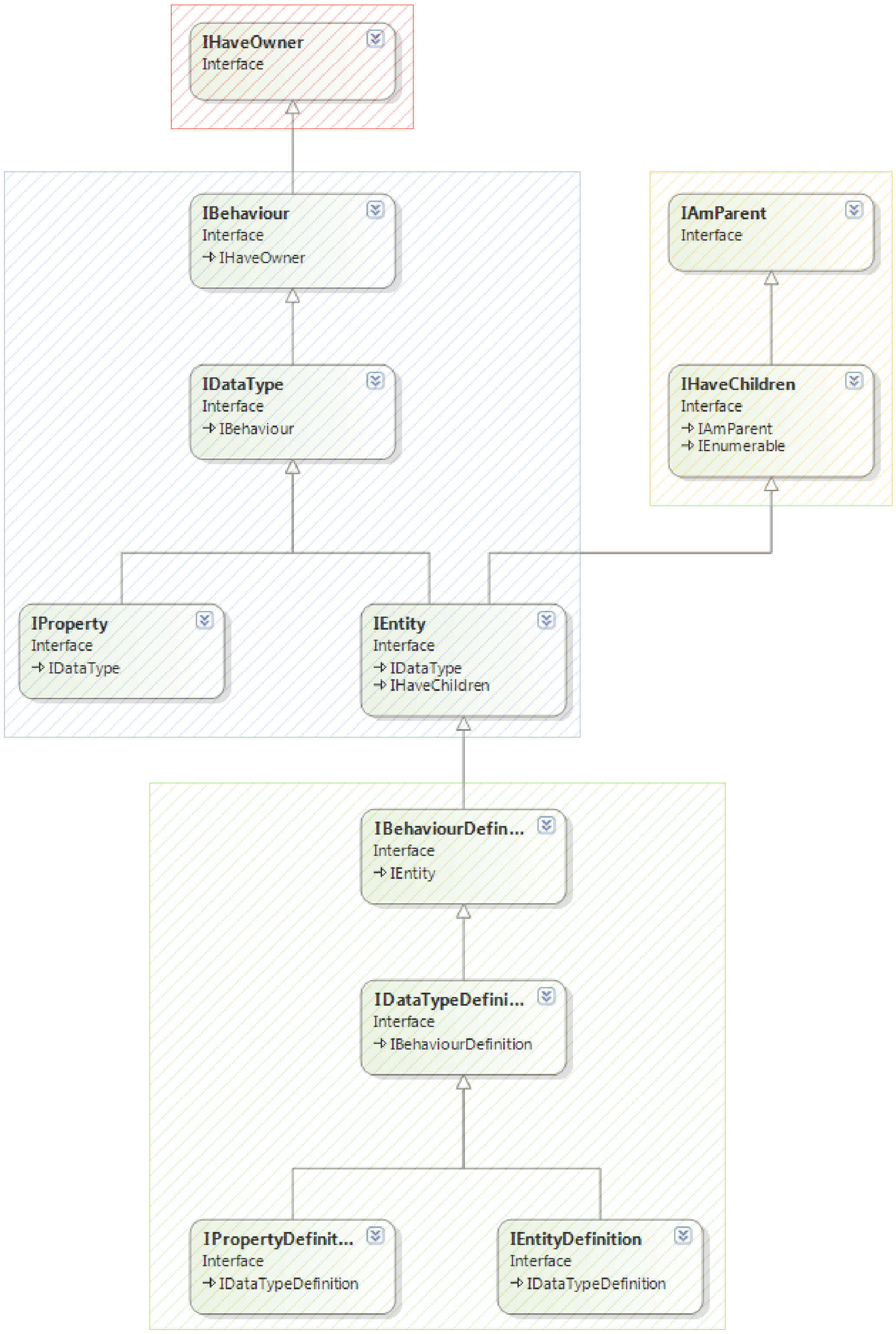
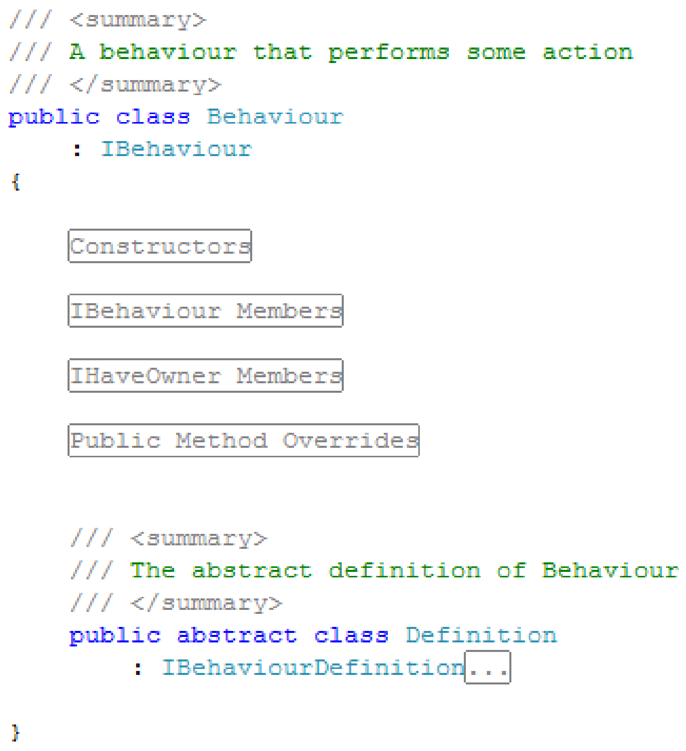


Figure 20 Software Engine Interface Inheritance

In Figure 20, each grey block represents a separate interface and the arrows tell us how the interfaces inherit from other interfaces. The "IHaveOwner" interface in the red grid is the base interface of our interface structure. All the other main interfaces in the blue and green grids inherit from it. There are also two branch interfaces in the yellow grid which are specially designed for the "IEntity" interface. Therefore, only "IEntity" and the interfaces in green grid inherit the functionalities defined in the "IHaveChildren" and "IAmParent" interfaces. The interfaces in the blue and green grids are our main interfaces. "IBehaviour", "IDataType", "IProperty" and "IEntity" are implemented by four key C# classes "Behaviour", "DataType", "Property" and "Entity" in our software engine. Each of these classes also contains a sub-class called "Definition" which implements "IBehaviourDefinition", "IDataTypeDefinition", "IPropertyDefinition" and "IEntityDefinition" as shown in the example below.



```
/// <summary>
/// A behaviour that performs some action
/// </summary>
public class Behaviour
    : IBehaviour
{

    Constructors

    IBehaviour Members

    IHaveOwner Members

    Public Method Overrides


    /// <summary>
    /// The abstract definition of Behaviour
    /// </summary>
    public abstract class Definition
        : IBehaviourDefinition...

}
```

Using interfaces has greatly increased the scalability and flexibility of our program structure. We are able to implement the same method differently in different classes even though they have got the same signature from the interfaces. Because the interfaces also have an inheritance structure like C# classes do, we can easily cast C# objects to an interface type and pass different object types as parameters that still retain their correct data type. As a result of the above two advantages of using interfaces, we are able to create a generic program structure with a recursive pattern. This advanced generic pattern will be addressed later in the parser implementation section.

4.2.2 Class Inheritance

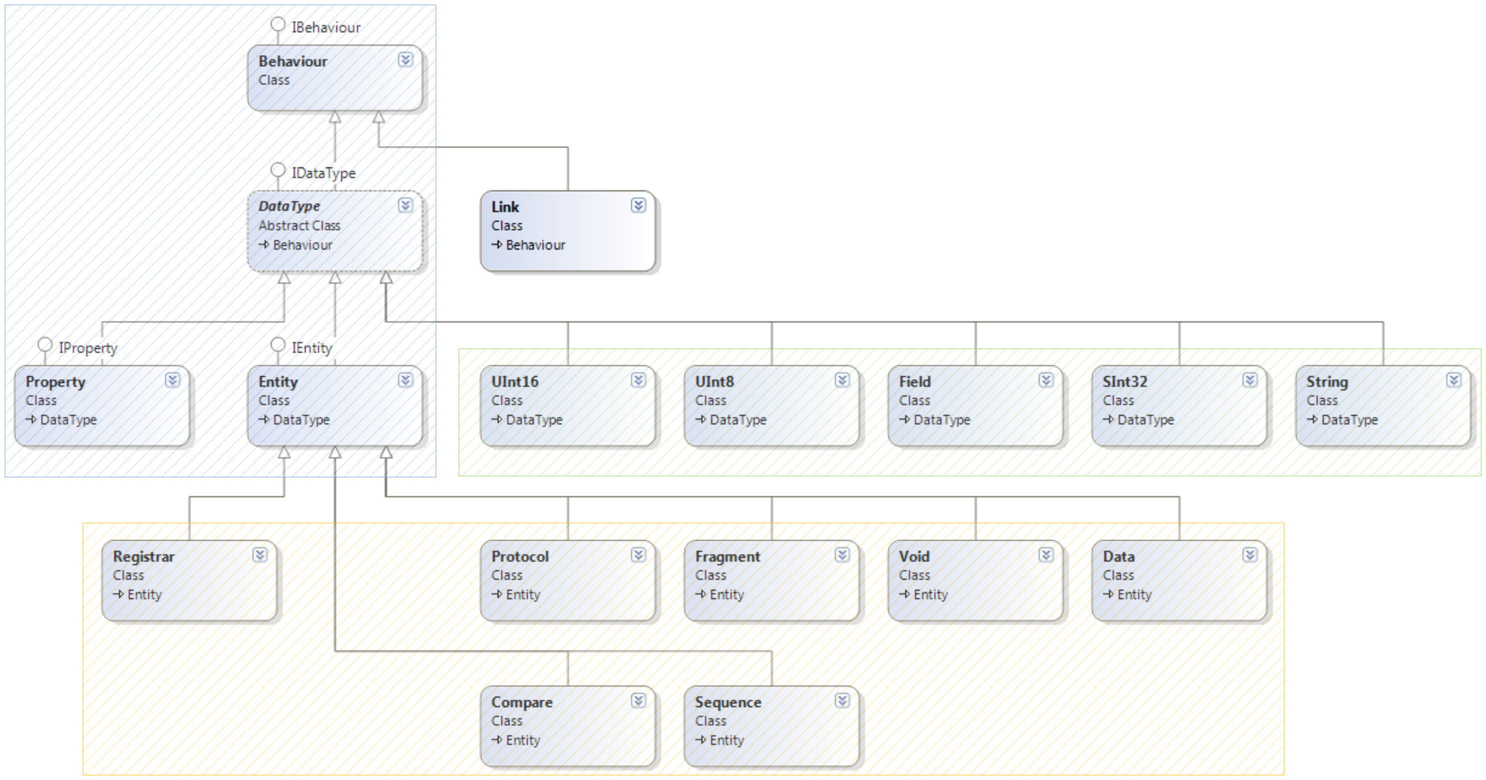


Figure 21 Software Engine Class Inheritance

As we have mentioned before, we have four key classes "Behaviour", "DataType", "Property" and "Entity" which implement their corresponding interfaces shown in the blue grid in Figure 21.

The word "Behaviour" has its English meaning. It refers to the actions of a system, usually in relation to its environment, which includes the other systems around as well as the physical environment. It is the response of the system to various stimuli or inputs, whether internal or external, conscious or subconscious, overt or covert, and voluntary or involuntary.

In our program, a "Behaviour" represents a C# object which is able to perform a certain designated activity or action in our software engine. Because the "Behaviour" class sits on top of the inheritance tree, the instances from any classes derived from the "Behaviour" class can be seen as Behaviours. Any instances of type "DataType", "Property" or "Entity" are able to be cast to "Behaviour" type. Therefore, we have called this inheritance structure a "Behaviour-based Object Model" (BOM). In our BOM, Behaviours are executable. "Executable" means that the Behaviour object will contain a method which can be used to evaluate itself. The result of executing a Behaviour can be either a value or another Behaviour. If the result is a value, then this is the end of the execution and the value gets stored. If the result is another Behaviour, the program will keep evaluating the Behaviour until it finds a value. This kind of evaluation mechanism is called "Recursion". More details will be given in a later section regarding the use of recursion in our program structure.

The "DataType" class is an abstract class. It is derived from the "Behaviour" class and does not perform any functionality at all. However, we have also created several classes which implement the "DataType" abstract class. These classes are shown in the green grid in Figure 21. The instances of these classes are able to store the results from the evaluation process we have described in the previous paragraph, i.e. the data value. These "DataType" classes are mostly the same as Visual Studio system data types, but having our own "DataType" enables us to do something like the following:

```
        <_DataTypes.UInt16>32</_DataTypes.UInt16>
```

We can create our own customized behaviour for any particular "DataType". In this case, we have a UInt16 subtract method which can perform a subtract operation on two UInt16 type numbers.

Most programming environments with recursive subroutines use a stack for control flow. This structure typically also stores local variables, including subroutine parameters. The Forth language often does not have local variables, however, nor is it call-by-value. Instead, intermediate values are stored on a second stack known as the Forth stack. Instructions operate directly on the topmost values on the Forth stack. As usual, values are stored on the Forth stack on a last in first out basis.

We have implemented a "Forth" class to perform Forth stack operations which include add, subtract, multiply, divide and equal. The instance of the "Forth" class is able to operate the basic push and pop operations on the Forth stack, as well as a few data manipulation methods such as clear, count and peek. Forth operations pop the top two values off the stack, perform the specified operation, and then push the result value back onto the stack. The forth engine also can perform operations on strings as well as numbers.

The "Property" class is derived from the "DataType" class. A Property instance is the only object which is capable of having an identifier, in our software engine. By having an "Id" attribute, properties are able to be located anywhere inside our software engine, and can be found by searching for the Id value. We have implemented a method called "FindExactly(String theId)" for finding specific property objects. A property has another attribute called "Value", which is used to store class instances as its content. In general, a property is a special case of a "DataType" instance, which is traceable by its Id anywhere inside our software engine. Once objects are named, we are able to start assigning internal relationship between these objects. This job will be carried out by our "Entity" instance next.

The meaning of the English word "Entity" is something that has a distinct, separate existence, although it need not be a material existence. An entity could be viewed as a set containing subsets. In philosophy, such sets are said to be abstract objects.

In BOM, we have defined "Entity" a bit different from its accepted meaning in computing. An "Entity" is a kind of object which is allowed to have sub-objects, i.e. children. An "Entity" instance is an object which manages the relationships between objects in our software engine. Such classes are shown in the yellow grid in Figure 21. "Registrar" is an entity type object which registers all the object structural information used in our software engine. Whenever we are create a new object from an XML document (create a C# object according to an XML tag), the software engine will read the "Registrar" for information about how to create that particular type of object. More information will be given on "Registrar" in the following sections. The "Protocol", "Fragment", "Void" and "Data" classes are used as containers which hold certain types of objects in our program. The "Link", "Compare" and "Sequence" classes are created for special purpose use. Instances of these classes perform the core operations of our software engine using recursion.

"Link" is a class which is responsible for establishing and maintaining the inter-relationship between objects in our software engine. "Link" has a "Path" attribute which stores the text based navigation information specified in the XML template. The path contains a combination of characters and symbols which are used to represent the relationship between objects in our program. An example of a link instance is shown below:

```
<_Behaviours.Link Path="$iMonitorZigbee.AccessCommand" />
```

This link instance points to the data tree (the symbol "$" represents the data tree constructed from the incoming data frames), then iMonitorZigbee data, AccessCommand field. When this link instance is executed, the engine will recursively search for the expected object following the path information. The engine will firstly jump onto "Data Tree" and search for an object named "iMonitorZigbee". If the expected object is found, the engine will initiate another search for an object named "AccessCommand" under the found object. If the expected object is found, it will be returned as the outcome of the link. Otherwise, an error message will be sent to the engine which reports that an error occurred during the link operation.

"Compare" is a class designed to perform a comparison activity using the Forth stack. It is derived from the "Entity" class. An instance of this class compares two values on the Forth stack. If the result returns true, then the link stored under the instance will be run. Otherwise, it will exit the current compare process and carry on with the next compare activity. A "Compare" instance will be used when the software engine needs to make a decision on the type of data it is going to receive next, based on the values received and stored in other places previously. When there is more than one type of information that can be received, the software engine needs to make a decision between these different choices. We will demonstrate how a compare instance works with an example below.

```
<_Behaviours.Compare>
   <_Behaviours.Sequence>
      <_Behaviours.Link Path="!iMonitorZigbee.Packet.Payload.AccessCommand.Command" />
      <_DataTypes.UInt8>1</_DataTypes.UInt8>
      <_Behaviours.Link Path="_DataTypes.UInt8.Equal" />
   </_Behaviours.Sequence>
   <_Behaviours.Link Path="$iMonitorZigbee.Device" />
</_Behaviours.Compare>
```

This compare instance example compares the value stored in the "AccessCommand.Command" field with a UInt8 type value 1. First, the engine will run the first link behaviour and locate the value stored in the Command field. Then the found result gets pushed on the Forth stack. After that, the UInt8 type value 1 gets pushed on the Forth stack as well. Finally, the engine will perform an "Equal" operation on the two values on the Forth stack. If the result is true, the link behaviour stored as the second child of compare will be run and it will lead to other structures or values in our program. If the result is false, the engine quits the current compare process and starts the next compare instance until it finds a match. The compare mechanism is designed to find one positive match. However, if there no match is found, the compare evaluation process will pop up an error message to our engine to report an abnormal state.

"Sequence" is a class which represents a series of activities to be performed on the Forth stack. When an instance of the "Sequence" class is run, the engine pushes the first two children of the sequence onto the Forth stack. Then the engine evaluates the third child with the previous two children on the Forth stack and retains the final result on the Forth stack. The result then can be used in other parts of the program by popping it off the Forth stack. An example of such operation is given below:

```
<_Behaviours.Sequence>
   <_Behaviours.Link Path="!iMonitorZigbee.Packet.Length" />
   <_DataTypes.UInt16>6</_DataTypes.UInt16>
   <_Behaviours.Link Path="_DataTypes.UInt16.Subtract" />
</_Behaviours.Sequence>
```

This example performs a subtract Forth stack operation which calculates the payload length. The engine firstly gets the value stored in the "Packet.Length" field (the length of the packet) and pushes it onto the Forth stack. Then the engine pushes a UInt16 type value 6 onto the Forth stack. This is the total length of the other fields in the packet, excluding the payload field. Finally, the engine will perform a subtract operation between the two values on the Forth stack. The result is kept on the stack and it is available for other objects to use in the program.

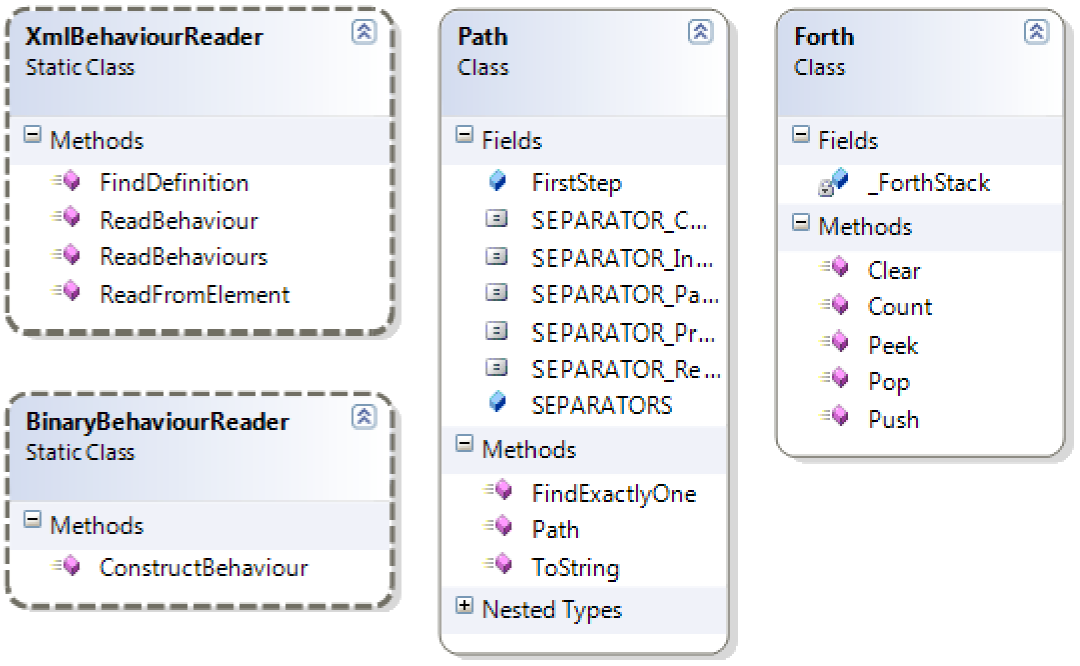


Figure 22 Software Engine stand alone classes

We have also implemented a few classes (Figure 22) which do not implement or inherit from other classes or interfaces. The "XmlBehaviourReader" and "BinaryBehaviourReader" classes are implemented as static classes. Therefore, we are able to use their methods without creating any instances. They are used to parse the XML template and incoming binary data packets.

The "Path" class is used to retrieve navigation information from the text-based values specified in the path attribute in the XML template. A path object takes a path written in text, breaks it up into meaningful fragments and uses the fragments as steps for navigation purposes.

4.3 Registrar

The "Registrar" database provides the structural information needed to create objects from XML tags in the protocol template. The "Registrar" registers all necessary "Definition" singletons for our engine in a systematic manner. Each entry definition is stored uniquely and attached to the designated category as shown in Figure 23. The "Registrar" acts as a lookup dictionary for our software engine. It limits the types of object our software engine is able to create from the XML-based protocol template. Currently, "Registrar" contains three main categories: Behaviours, DataTypes and Protocols.

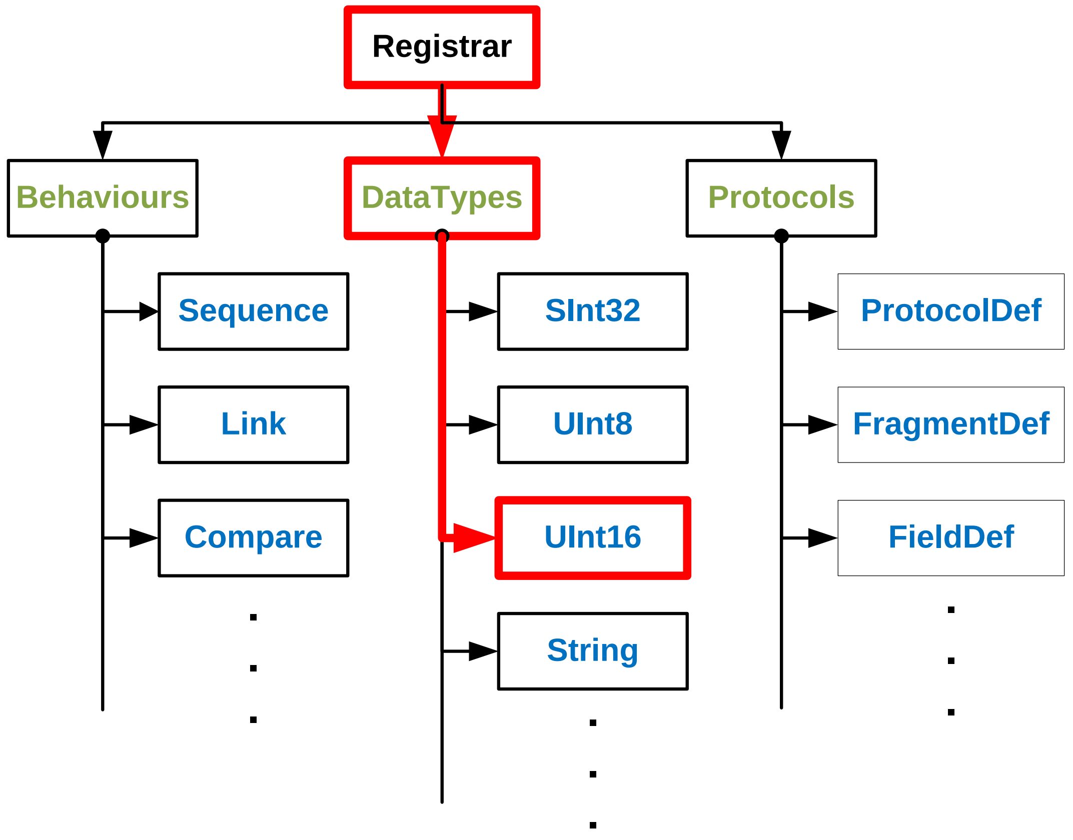


Figure 23 Registrar Tree Representation

Each XML tag in a protocol template is specified as a path to a specific location in the "Registrar". The software engine evaluates the path, navigates to the specified location according to the information drawn from the path, and obtains the definition for that particular XML tag in the "Registrar". The definition lays out the characteristics of the object and grants the functionalities to the object to be created. There is a built-in function called "Construct" in every "Definition" class in our program. This method ensures all our definition instances acquired from "Registrar" are able to construct C# objects from the XML tags in the protocol templates.

For finding a particular entry definition stored in the "Registrar", we have defined and integrated a few special symbols and a basic rule. The rule applied to our "Registrar" is the "Parent and Child Inter-Relationship". Every entry definition in "Registrar" maintains a record of its parent and its children at the moment they get registered. We use the "**_**" (underscore) symbol to indicate a registrar instance, the "**.**" (dot) symbol to indicate a "Child" and the "^" (up arrow) symbol to indicate a "Parent" in our program. In order to locate an entry definition in "Registrar", we need to specify a path to the location of the expected definition. The correct way of creating a path must always have the "_" symbol in front of any other characters or symbols, because every search begins with "Registrar". Then, one of the three default definition category names must be specified after "_" and followed by a "**.**" symbol, because the three definition category objects are registered as the children of "Registrar". The dot tells the engine to search for a particular category specified in the path. Finally, the phrase which represents the definition we are searching for should be provided after the last "**.**" symbol. The phrase is treated as the definition name which we are searching for under the definition category defined before. For example, if we want to create a "Sequence" object, the path in the XML tag should be written as follow:

```
< _ Behaviours . Sequence >
```

The meaning of this path is: Search within "Registrar" under "Behaviours" category for a definition instance named "Sequence". The path is broken up into three steps in the "Path" class. The first step is the "**_**" (Registrar) step which takes us to the registrar instance. The second step is a child step because of the "**.**" (dot) symbol. The text between the dot symbol and the registrar symbol is processed as the name of the object which we are searching for within "Registrar". In this case the word is "Behaviours". The third step is also treated as a child step because there are no other special symbols placed after the last phrase. So the last step will search for the phrase "Sequence" under the "Behaviours" definition category. Once we have reached the end of the last step, the object located in "Registrar" will be the expected "Sequence" definition instance.

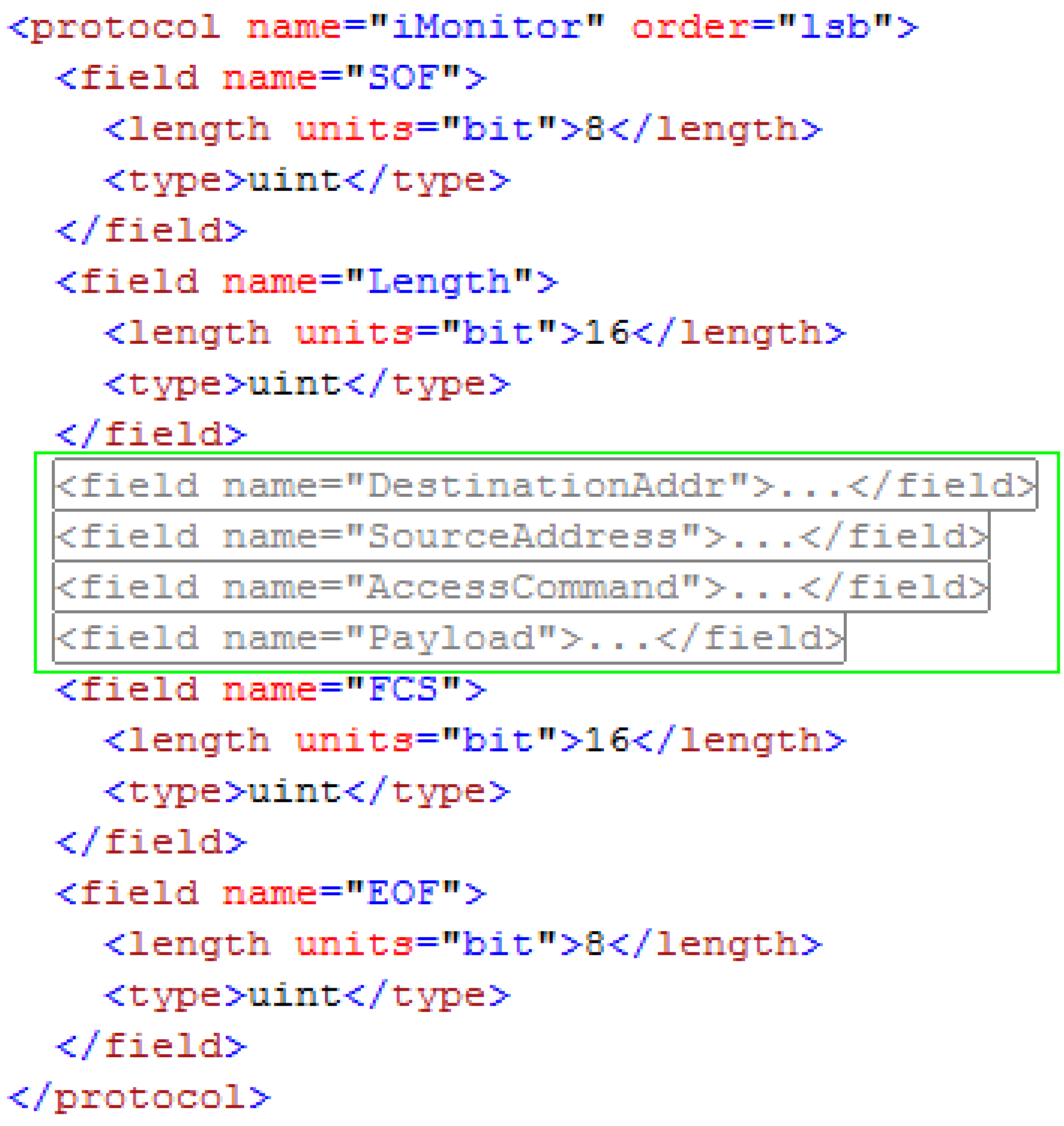
Having "Registrar" effectively reduces the chance of a software error in our engine due to any unknown objects we may come across during the XML parsing stage. The "Registrar" gives the user a brief idea of what types of object the engine is capable of constructing, like the DTD of an XML document. We can also keep "Registrar" up to date by simply adding a new piece of definition information at any time.

4.4 Protocol Template

An XML protocol template is an XML based document which specifies the structure of a particular communication protocol. An XML document essentially represents a tree-structured data model. The XML document thus can be regarded as the serialization of this tree model in a depth-first, left-to-right traversing order. XML has advantages in both structuring and representing information. That is the reason we have chosen it as our tool to implement the protocol template. In this section, we will describe a few ways of implementing the protocol template, which we have discovered during this research. The figures below just show the concept of how to build a protocol template from different approaches investigated within this research. The full image of the entire XML template is not listed because the size of the entire protocol is too big to draw.

4.4.1 Flat Fields Format

A protocol usually describes the format of data in blocks of information such as header, body/payload, checksum, etc. The easiest way to create a template is to flatten the entire protocol document in a depth-first, left-to-right traversing order in XML format. We call this the Flat Fields Format (FFF) approach (Figure 24). It simply lists out each individual block of information from the protocol document on a flat-field basis. Each field tag has its own name drawn from the protocol document as well as its length and type information. However, we can immediately notice the negative impact of this approach. It does not show us any structural information about the protocol document at all. It is a pure text based protocol template without having any interrelationship between any of the field tags.

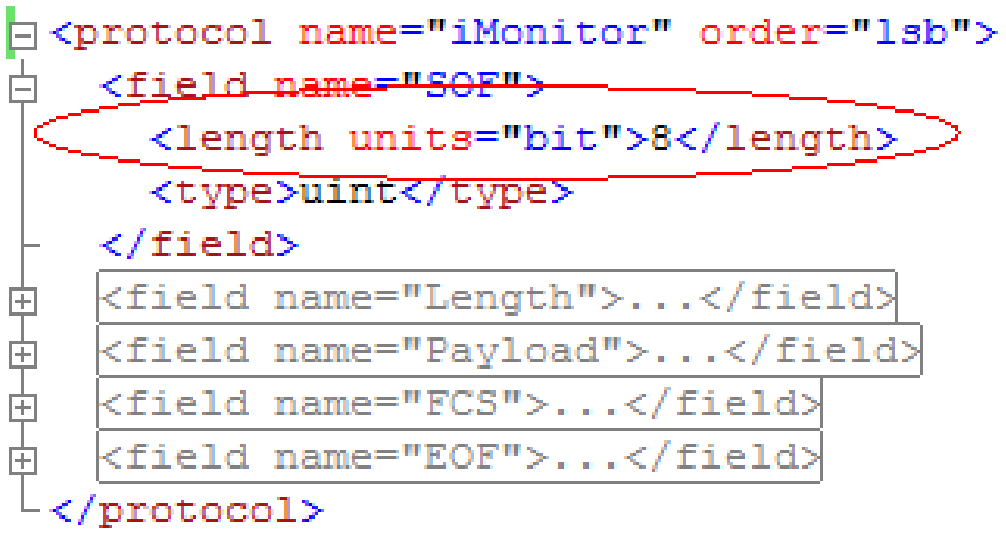


```
<protocol name="iMonitor" order="lsb">
  <field name="SOF">
    <length units="bit">8</length>
    <type>uint</type>
  </field>
  <field name="Length">
    <length units="bit">16</length>
    <type>uint</type>
  </field>
  <field name="DestinationAddr">...</field>
  <field name="SourceAddress">...</field>
  <field name="AccessCommand">...</field>
  <field name="Payload">...</field>
  <field name="FCS">
    <length units="bit">16</length>
    <type>uint</type>
  </field>
  <field name="EOF">
    <length units="bit">8</length>
    <type>uint</type>
  </field>
</protocol>
```

Figure 24 FFF Structure

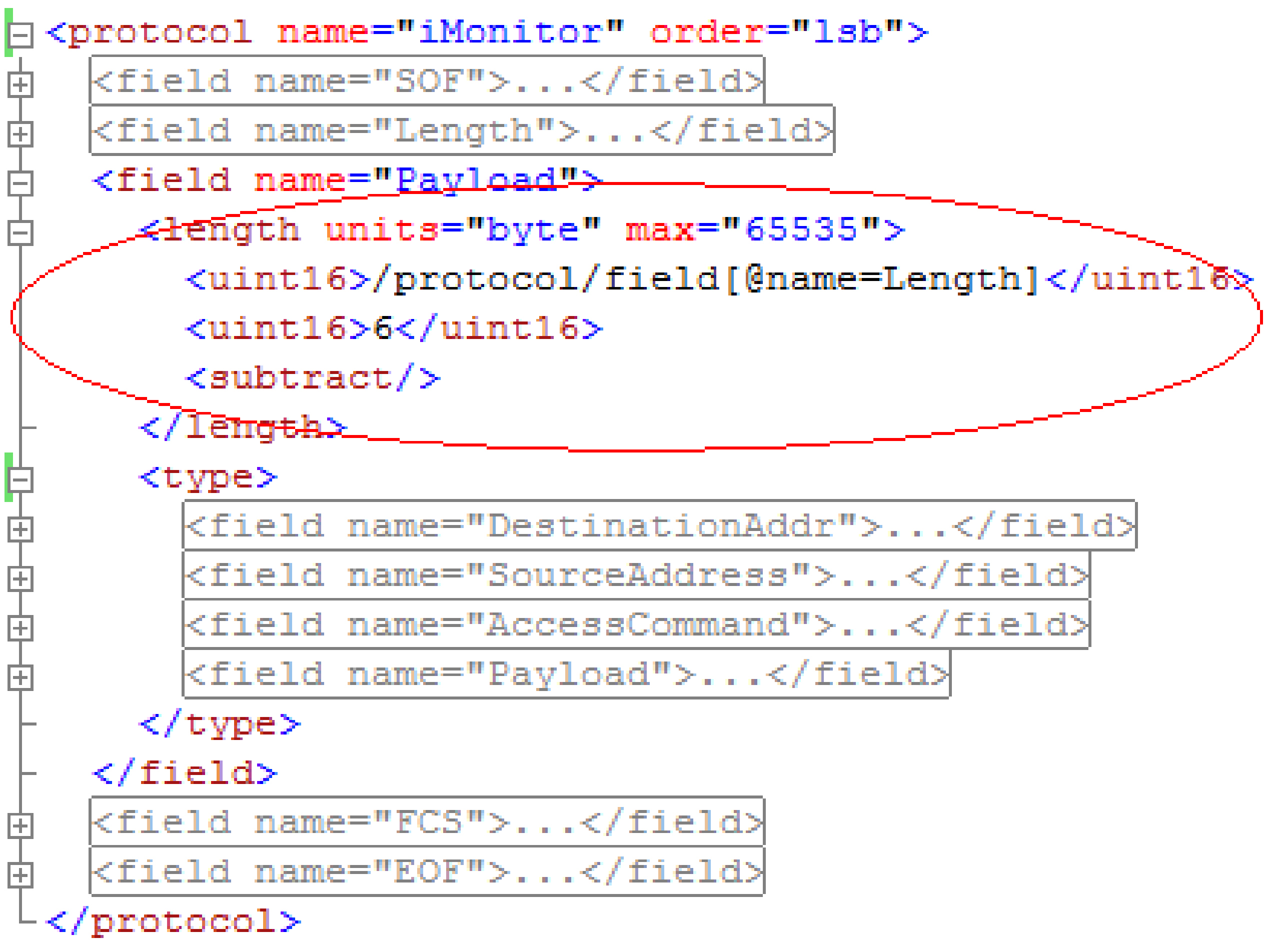
4.4.2 Nested Fields Format

In the Nested Fields Format (NFF) approach, the structure of the protocol template follows the order of the protocol document in the same way as FFF does, but it forms information into different groups to indicate the structure characteristics of a protocol document. The root element is called "Protocol". XML rules state the root element is the top level tag of an XML document. "Field" tags are embedded under the root element with unique "name" attributes. Each field tag represents a certain block of information listed in the protocol document. There are another two special tags created under every field tag. One is named "length" which tells us the size of the field (e.g. 8-bit). The other is called "type" which indicates the data type of the field (e.g. integer).



```
<protocol name="iMonitor" order="lsb">
  <field name="SOF">
    <length units="bit">8</length>
    <type>uint</type>
  </field>
  <field name="Length">...</field>
  <field name="Payload">...</field>
  <field name="FCS">...</field>
  <field name="EOF">...</field>
</protocol>
```

Figure 25 Fixed Length Field with NFF

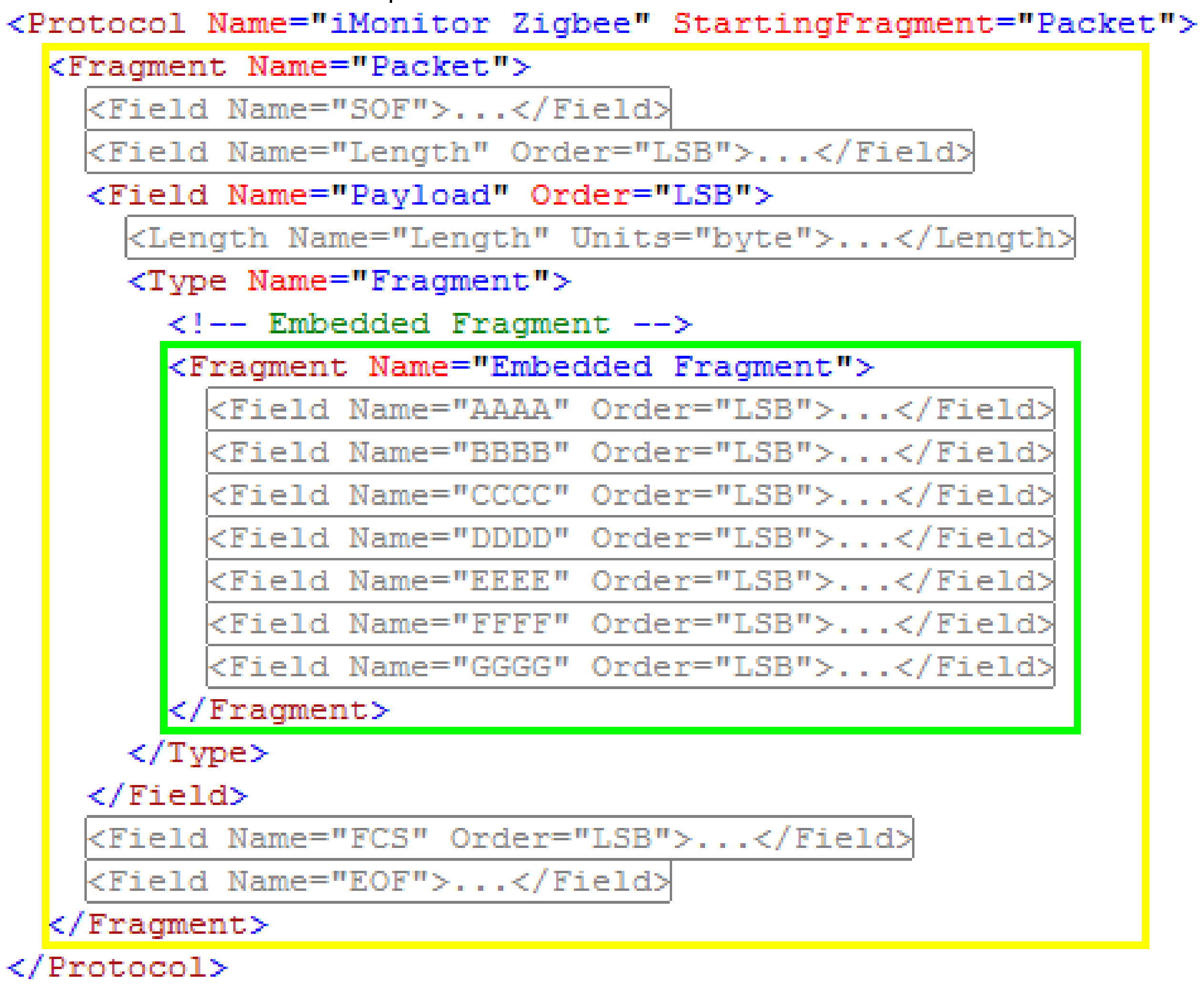


```
<protocol name="iMonitor" order="lsb">
  <field name="SOF">...</field>
  <field name="Length">...</field>
  <field name="Payload">
    <length units="byte" max="65535">
      <uint16>/protocol/field[@name=Length]</uint16>
      <uint16>6</uint16>
      <subtract/>
    </length>
    <type>
      <field name="DestinationAddr">...</field>
      <field name="SourceAddress">...</field>
      <field name="AccessCommand">...</field>
      <field name="Payload">...</field>
    </type>
  </field>
  <field name="FCS">...</field>
  <field name="EOF">...</field>
</protocol>
```

Figure 26 Variable Length Field with NFF

A field like "SOF" is designed to hold a constant value in the protocol document. So we have specified a fixed number 8 to the "length" tag text-element in the protocol template (Figure 25) which indicates the size of the field is 8-bit, whereas a field like "Payload" may contain a group of values or series of characters. The size of this kind of field may vary and has to be evaluated dynamically according to the values stored in other fields at runtime (Figure 26).

If a protocol template is written in NFF, the structure layout of the template follows the exact structure of the protocol document. The user will immediately have a good understanding of what the software engine is dealing with at each level of the protocol template by following the original protocol document. However, this structure does not immediately give us any information hidden in any lower levels. In fact, we must go through each level to find out what is contained in the next level. This really is a time consuming process. The structure of a protocol template in NFF is also hard to manage when a future upgrade occurs. Any changes made in the middle of the template may easily result a faulty logic in the entire body because the entire protocol template is written as a whole. In order to overcome this side effect of the NFF, we have a new idea of using a "Fragment" which groups certain fields as an integrated body. By adding the "Fragment" tag, the protocol template has been transformed to a brand new shape shown in Figure 27. This new concept leads us to our latest approach – a Linkable Fragment Format (LFF).

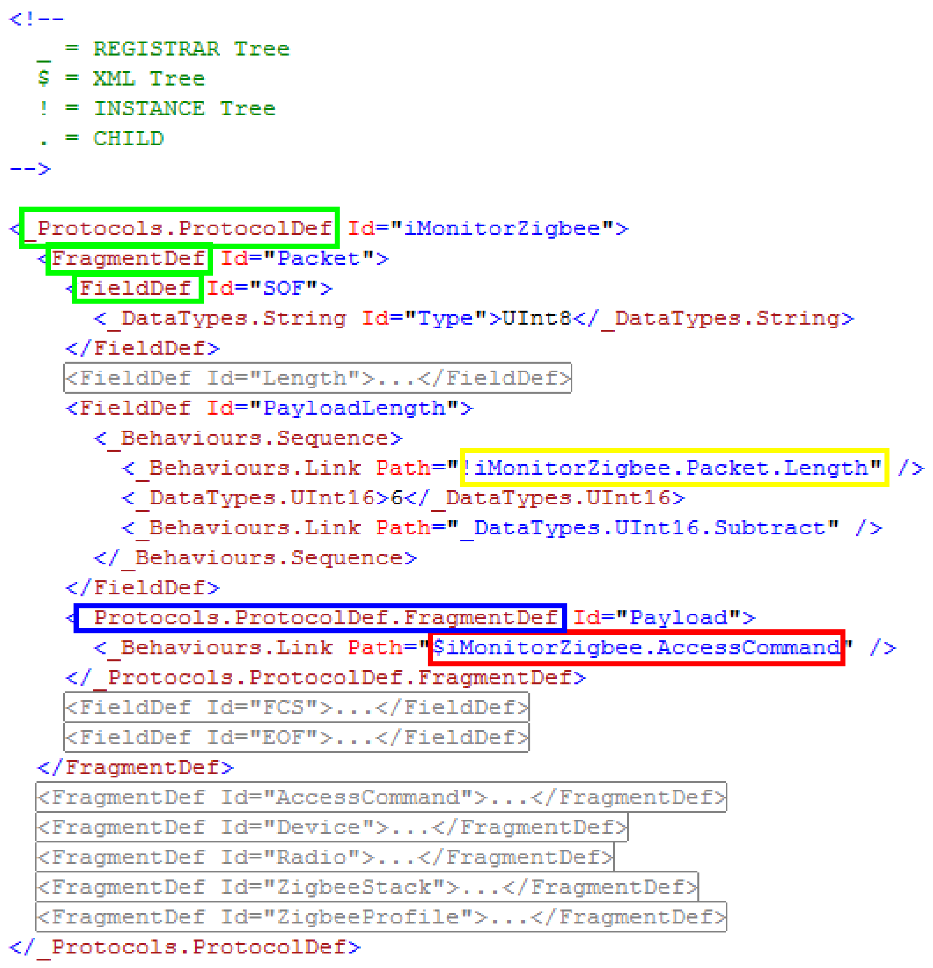


```
<Protocol Name="iMonitor Zigbee" StartingFragment="Packet">
  <Fragment Name="Packet">
    <Field Name="SOF">...</Field>
    <Field Name="Length" Order="LSB">...</Field>
    <Field Name="Payload" Order="LSB">
      <Length Name="Length" Units="byte">...</Length>
      <Type Name="Fragment">
        <!-- Embedded Fragment -->
        <Fragment Name="Embedded Fragment">
          <Field Name="AAAA" Order="LSB">...</Field>
          <Field Name="BBBB" Order="LSB">...</Field>
          <Field Name="CCCC" Order="LSB">...</Field>
          <Field Name="DDDD" Order="LSB">...</Field>
          <Field Name="EEEE" Order="LSB">...</Field>
          <Field Name="FFFF" Order="LSB">...</Field>
          <Field Name="GGGG" Order="LSB">...</Field>
        </Fragment>
      </Type>
    </Field>
    <Field Name="FCS" Order="LSB">...</Field>
    <Field Name="EOF">...</Field>
  </Fragment>
</Protocol>
```

Figure 27 XML Template with "Fragment"

4.4.3 Linkable Fragment Format

The latest protocol template structure (LFF) combines both the advantages of easy protocol information accessibility and excellent protocol structure management from the previous two approaches. In the LFF approach, the XML tags are implemented differently from previously. A LFF XML tag uses a series of characters and symbols instead of using a specific name. These characters and symbols are written in a particular order to form a path that points to a location in the "Registrar". The information stored at this location is what we need to use to construct the C# object according to the XML tag. We call this information the "Definition" of the object to be constructed. We have introduced three different types of path in the LFF approach. Figure 28 gives a brief view on the formation of these paths.



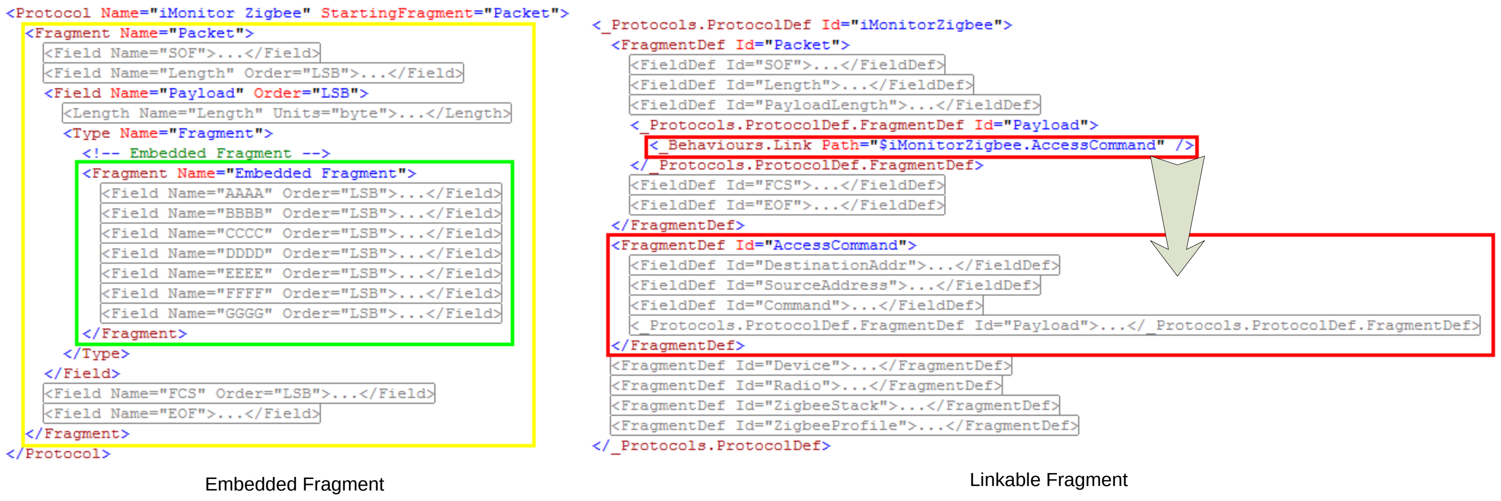
```
<!--
  _ = REGISTRAR Tree
  $ = XML Tree
  ! = INSTANCE Tree
  . = CHILD
-->

<_Protocols.ProtocolDef Id="iMonitorZigbee">
  <FragmentDef Id="Packet">
    <FieldDef Id="SOF">
      <_DataTypes.String Id="Type">UInt8</_DataTypes.String>
    </FieldDef>
    <FieldDef Id="Length">...</FieldDef>
    <FieldDef Id="PayloadLength">
      <_Behaviours.Sequence>
        <_Behaviours.Link Path="!iMonitorZigbee.Packet.Length" />
        <_DataTypes.UInt16>6</_DataTypes.UInt16>
        <_Behaviours.Link Path="_DataTypes.UInt16.Subtract" />
      </_Behaviours.Sequence>
    </FieldDef>
    <_Protocols.ProtocolDef.FragmentDef Id="Payload">
      <_Behaviours.Link Path="$iMonitorZigbee.AccessCommand" />
    </_Protocols.ProtocolDef.FragmentDef>
    <FieldDef Id="FCS">...</FieldDef>
    <FieldDef Id="EOF">...</FieldDef>
  </FragmentDef>
  <FragmentDef Id="AccessCommand">...</FragmentDef>
  <FragmentDef Id="Device">...</FragmentDef>
  <FragmentDef Id="Radio">...</FragmentDef>
  <FragmentDef Id="ZigbeeStack">...</FragmentDef>
  <FragmentDef Id="ZigbeeProfile">...</FragmentDef>
</_Protocols.ProtocolDef>
```

Figure 28 Different Links in LFF Approach

The first type is called the "Registrar Path". In the LFF approach, every XML tag is a path. It tells the "Protocol Parser" where to find the structural information about the object to be constructed in the "Registrar". For example, the green box in Figure 28 is a registrar path. The `<_Protocols.ProtocolDef>` tag means the "ProtocolDef" definition in "Registrar" (we have chosen "_" symbol to represent the registrar instance) in the "Protocols" category. The software engine will search the provided location for the required definition. The found definition will be used to construct a "ProtocolDef" C# object which represents the `<_Protocols.ProtocolDef>` tag from the protocol template. An immediate concern may arise for the "FragmentDef" tag and "FieldDef" tags. These two paths have no "_" symbol in front but they are still in a registrar path. These are reference paths. These paths are written in short form. We have a section of code in our program which handles these scenarios. The engine uses the path information from its parent tag `<_Protocols.ProtocolDef>` as the reference. The reference gets passed on to the XML tag construction with the `<FragmentDef>` and `<FieldDef>` tags, i.e. "FragmentDef" will treat "`_Protocols.ProtocolDef`" path as its construct reference, and add "`FragmentDef`" after it. The new path for "FragmentDef" will be "`_Protocols.ProtocolDef.FragmentDef`". The same applies for the "FieldDef" tag. There are a few rules on the use of construct references. In this case, the XML tag in the blue box will not work because this "FragmentDef" tag sits on the same level as the "FieldDef" tags, so it cannot be referenced again from the previous `<FragmentDef>` tag. Therefore, we have to use the XML tag in long form as `<_Protocols.ProtocolDef.FragmentDef>`. Apart from the registrar path example we have demonstrated, there are also a few other registrar path examples such as "`_Behaviours.Link`" and "`_DataTypes.UInt16`". These registrar paths work the same way as the example we have shown above.

The second type is called the "XML Path". The red box in Figure 28 indicates a path of this type. The "XML Path" provides internal links between all the XML tags within the entire protocol template. With this type of path introduced in LFF, the layout of the protocol template has been transformed to a new dimension. The structure of the protocol template normally should follow the structure of the protocol document. This really limits the scalability of our protocol template because the structure of the protocol template has to be constructed in NFF as shown in Figure

29 (left side). But with LFF, we are able to move some of the tags (FragmentDef and FieldDef) buried deep inside the protocol template and place them on the front line. Then we need to insert the XML paths at their original location where we have taken the tags away. The software engine will work out where the tags have moved to in the protocol template according to the XML path provided. An example of such a path is shown in the red box in Figure 28. "`$iMonitorZigbee.AccessCommand`" is an XML path. The "`$`" sign at the very beginning tells the engine this is an XML path and we need to search on the "Template Tree". "Template Tree" is the tree which has been constructed from the protocol template. "`iMonitorZigbee`" is the name of the protocol we are looking for. This protocol information is stored on "Template Tree" where the search will be carried out. "`AccessCommand`" is the name of the fragment we require. The engine will execute the path and link the fragment to its original location where it should be. With the XML path, we are able to extract blocks of information (fragments) from any level inside the protocol template and place them on the same front line. The XML template management work has become easier than before. We are able to add a new fragment to describe any new data structure separately from the entire structure of the whole protocol template. Once we have written the fragment description, we simply need to insert an XML path where the new data structure occurs in the protocol template, rather than rearranging the entire protocol template structure to cope with the insertion of the new data structure.



```
<Protocol Name="iMonitor Zigbee" StartingFragment="Packet">
  <Fragment Name="Packet">
    <Field Name="SOF">...</Field>
    <Field Name="Length" Order="LSB">...</Field>
    <Field Name="Payload" Order="LSB">
      <Length Name="Length" Units="byte">...</Length>
      <Type Name="Fragment">
        <!-- Embedded Fragment -->
        <Fragment Name="Embedded Fragment">
          <Field Name="AAAA" Order="LSB">...</Field>
          <Field Name="BBBB" Order="LSB">...</Field>
          <Field Name="CCCC" Order="LSB">...</Field>
          <Field Name="DDDD" Order="LSB">...</Field>
          <Field Name="EEEE" Order="LSB">...</Field>
          <Field Name="FFFF" Order="LSB">...</Field>
          <Field Name="GGGG" Order="LSB">...</Field>
        </Fragment>
      </Type>
    </Field>
    <Field Name="FCS" Order="LSB">...</Field>
    <Field Name="EOF">...</Field>
  </Fragment>
</Protocol>
```

```
<_Protocols.ProtocolDef Id="iMonitorZigbee">
  <FragmentDef Id="Packet">
    <FieldDef Id="SOF">...</FieldDef>
    <FieldDef Id="Length">...</FieldDef>
    <FieldDef Id="PayloadLength">...</FieldDef>
    <_Protocols.ProtocolDef.FragmentDef Id="Payload">
      <_Behaviours.Link Path="$iMonitorZigbee.AccessCommand" />
    </_Protocols.ProtocolDef.FragmentDef>
    <FieldDef Id="FCS">...</FieldDef>
    <FieldDef Id="EOF">...</FieldDef>
  </FragmentDef>
  <FragmentDef Id="AccessCommand">
    <FieldDef Id="DestinationAddr">...</FieldDef>
    <FieldDef Id="SourceAddress">...</FieldDef>
    <FieldDef Id="Command">...</FieldDef>
    <_Protocols.ProtocolDef.FragmentDef Id="Payload">...</_Protocols.ProtocolDef.FragmentDef>
  </FragmentDef>
  <FragmentDef Id="Device">...</FragmentDef>
  <FragmentDef Id="Radio">...</FragmentDef>
  <FragmentDef Id="ZigbeeStack">...</FragmentDef>
  <FragmentDef Id="ZigbeeProfile">...</FragmentDef>
</_Protocols.ProtocolDef>
```

Figure 29 Embedded Format VS Linkable Format

The third type is called a "Data Path". The yellow box in Figure 28 indicates a data path. A "Data Path" provides links to the "Data Tree", which is the tree of the real data values constructed from the data frames being received. This kind of path is written in the protocol template, but it actually accesses the information at runtime and dynamically links to the data that has been analyzed by the "Data Analyzer" in our software engine. Like the "Protocol Parser" we have mentioned before, the "Data Analyzer" is the other core parser in our software engine. It reads in the data based on the "Template Tree" constructed by the "Protocol Parser" and builds a new tree called the "Data Tree". An example "Data Path" is shown below.

```
<FieldDef Id="PayloadLength">
  <_Behaviours.Sequence>
    <_Behaviours.Link Path="!iMonitorZigbee.Packet.Length" />
    <_DataTypes.UInt16>6</_DataTypes.UInt16>
    <_Behaviours.Link Path="_DataTypes.UInt16.Subtract" />
  </_Behaviours.Sequence>
</FieldDef>
```

This section of code calculates the size of the payload at runtime. Because the payload's length may vary, we can only calculate it when we have received a particular data frame. "`!iMonitorZigbee.Packet.Length`" is a data path. The "`!`" sign tells the engine this is a data path. Therefore it will be executed in the "Protocol Parser" for the "Template Tree" construction process. It only runs when we are actually reading the data frame. "`iMonitorZigbee`" is the name of the object we are searching for. "`Packet`" is the name of the fragment we require. "`Length`" is the field on the "Data Tree" where the length value gets stored at runtime.

With these three types of path, creating a protocol template in LFF becomes much simpler than before. We are able to construct the protocol template in many different forms. The scalability of our protocol template has been greatly improved.

4.5 Parser Implementation

The core of our software engine is formed with two main parsers: the "Protocol Parser" and the "Data Analyzer". Each individual parser functions differently. In this section, we will describe the frame of each parser and how the key function blocks operate inside the engine.

4.5.1 Protocol Parser

The "Protocol Parser" (Figure 30) is designed for parsing the protocol template only. At the moment, it only supports XML parsing. The reason is because our current version of the protocol template is structured in XML format. "Protocol Parser" can be divided into four main function blocks. They are "XML Reader", "Path Creation", "Navigation" and "C# Object Creation".

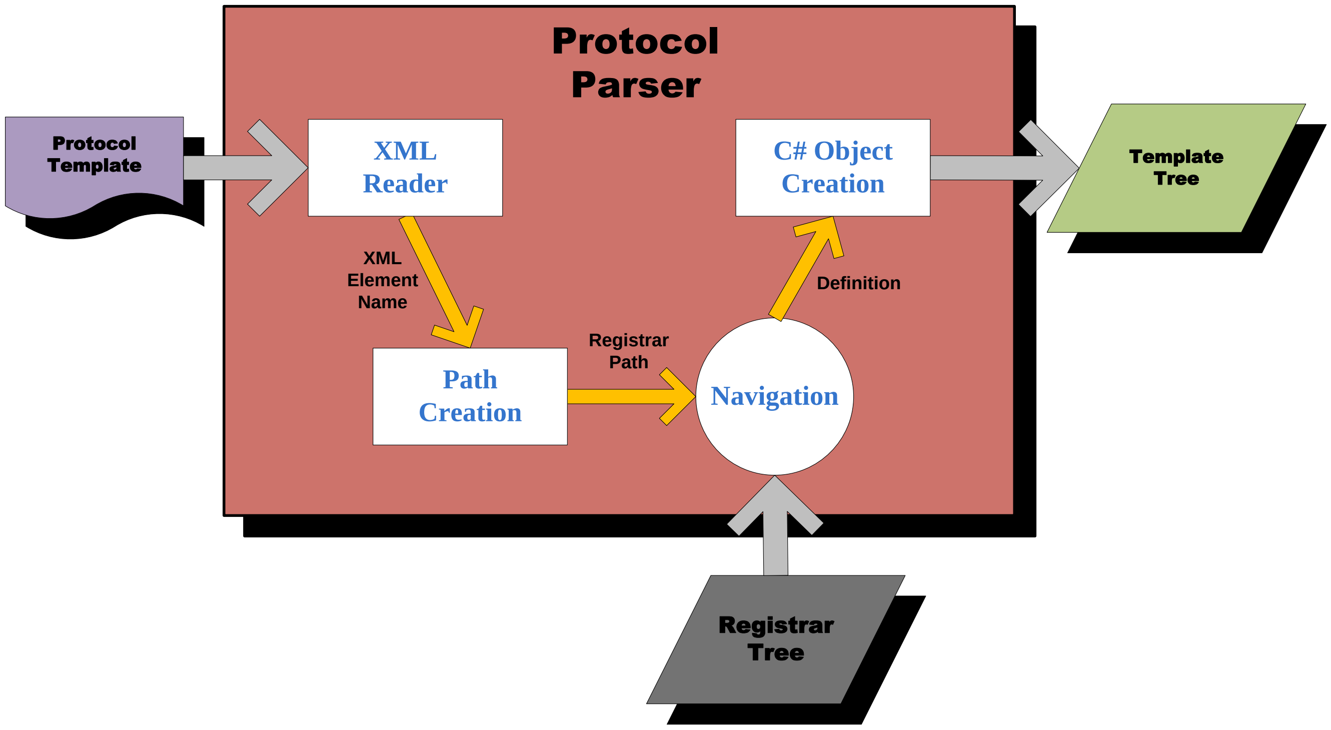


Figure 30 Protocol Parser Architecture

1) XML Reader

The protocol template written in XML is read by the "XML Reader". The file is read to a .NET stream and the stream is passed to the .NET XmlTextReader as follows:

```
    Stream aStream = new FileStream(OpenXmlFileDialog.FileName,FileMode.Open))
    XmlTextReader aXmlTextReader = new XmlTextReader(aStream);
```

The .NET XmlTextReader reads all XML elements and attributes one after another through the entire XML file. Our parser recursively checks each XML tag with its element name, attribute and text content. Then it creates the corresponding C# object which holds all the information gathered from the protocol template. All the created objects are maintained with relationships according to their position in the protocol template. The relationships between the objects will be used in the "Navigation" block.

2) Path Creation

Because each XML tag in our protocol template provides a registrar path, the "Path Creation" block is responsible for gathering this information from the protocol template and translating it into a meaningful expression for the "Path Evaluation" block. The XML tag has been processed in "XML Reader" and stored in a string variable. The "Path Creation" block translates this string into separate steps which will be used to find the location of the definition in the "Registrar". We have created an abstract "Step" class with two variables which will maintain the sequence of the steps.

```
    /// <summary>
    /// The previous step in the path
    /// The value will be null if the step is the first step in a path
    /// </summary>
    public readonly Step PreviousStep;

    /// <summary>
    /// The next step in the path
    /// </summary>
    public Step NextStep;
```

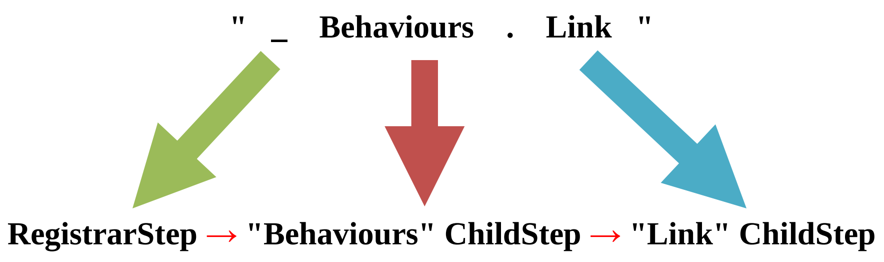
There are six different basic step classes which implement the abstract "Step" class. They are "GeneralStep", "RegistrarStep", "ParentStep", "ChildStep", "Instance Step" and "ProtocolStep". These "Step" class instances will take us to different internal objects inside our software engine. This mechanism is known as "Navigation". "Navigation" helps us to find the object we require by evaluating the step instances in our engine. It will use the objects' relationships which were stored by the "XML Reader" block. Then it selects the routine to find the required object.

```
/// <summary>
/// A Step that takes us to any object
/// </summary>
public class GeneralStep

/// <summary>
/// A Step that takes us to the Registrar. Always the first and only step
/// </summary>
public class RegistrarStep

/// <summary>
/// A Step that takes us to the parent of the current object
/// </summary>
public class ParentStep

/// <summary>
/// A Step that takes us to the child of the current object
/// </summary>
public class ChildStep

/// <summary>
/// A Step that takes us to the Instance on the Data Tree
/// </summary>
public class InstanceStep

/// <summary>
/// A Step that takes us to the Protocol
/// </summary>
public class ProtocolStep
```

The path given with each XML tag is a combination of characters and symbols. We have a "Path" class which handles the task of breaking down the path. It takes in the path stored by "XML Reader", then recognizes the symbols and breaks the path into different segments. Each segment is then converted into one of the six types of steps above. The following symbols are defined in our software engine for identifying the different types of steps:

```
public const char SEPARATOR_Registrar = '_';
public const char SEPARATOR_Parent = '^';
public const char SEPARATOR_Child = '.';
public const char SEPARATOR_Instance = '!';
public const char SEPARATOR_Protocol = '$';
```

For instance, we have a registrar path like "`_Behaviours.Link`". The engine will firstly detect "`_`" symbol which means "Registrar". So a RegistrarStep instance is created and its NextStep variable gets updated with the registrar step. The PreviousStep stores a null value because the registrar step is the first step. Then the engine will recognize the "`.`" symbol which means "Child". A ChildStep instance is created and its NextStep variable is updated with the child step. The registrar step created previously is stored in the PreviousStep variable. The name of the child step is the characters between the two symbols. i.e. `Behaviours`. If no symbol is found at the end of the path, it will also be treated as "Child". Therefore another ChildStep instance is created. The PreviousStep and NextStep variables are updated automatically as before. The name of the new child step is the characters between the last found symbol and the end of the path. i.e. `Link`.



3) Navigation

Once the path has been decomposed to steps in the "Path Creation" block, the steps are passed on to the "Navigation" block. This block evaluates the steps according to the values stored in the PreviousStep and NextStep variables as follows.

```
// Evaluate the path starting with the first step
Step aNextStep = this.FirstStep;
IBehaviour aInstance = null;
// Looping through all steps until no step is found
while (aNextStep != null)
{
   // Resolve the next step using the current instance
   aInstance = aNextStep.FindExactlyOne(aInstance);
   // Move to the next step
   aNextStep = aNextStep.NextStep;
}
return aInstance;
```

The engine always starts from the first step and evaluates each step recursively. We get the first step, and use it as a reference to get the next step. The "aNextStep" variable gets updated in every evaluation cycle with the value stored in the "NextStep" variable. The recursion stops when the value of the "NextStep" variable equals null. This means there are no more steps and the found definition instance will be returned. The result returned will be the definition instance stored in "Registrar" that the path points to. It is the required definition which will be used to construct the corresponding C# object.

4) Object Creation

The "Object Creation" block constructs C# objects and places them in the "Template Tree". Every definition stored in the "Registrar" features a built-in Construct() function. This function analyzes the definition instance found from the "Navigation" block and creates C# objects according to the information stored in the definition instance. We can think of this activity as an example of creating images from a stamp. The stamp itself is the "Definition" and the images stamped out are the objects we have created. Different stamps will produce different images. Then the constructed objects are placed on the "Template Tree" in the same way as they are listed in the protocol template. The "Template Tree" is the internal C# object structure which will guide our "Data Analyzer" when processing the incoming data frames.

4.5.2 Data Analyzer

The "Data Analyzer" (Figure 31) is designed for parsing incoming data frames only. At the moment, it only supports binary data parsing. Parsing of data that consists of delimited strings is to be investigated in the future. The "Data Analyzer" is divided into three main function blocks. They are "Evaluation", "Verification" and "Binary Reader".

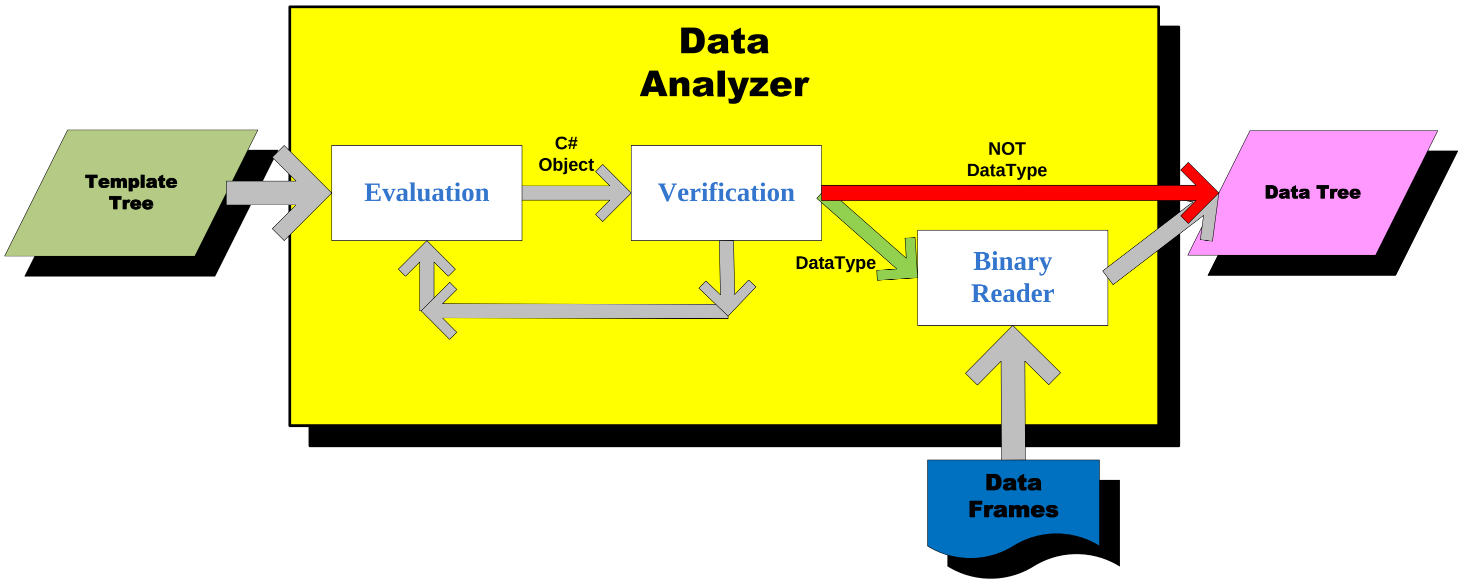


Figure 31 Data Analyzer Architecture

1) Evaluation

The "Template Tree" is a representation of the communication protocol document which is used for our data communication. The "Evaluation" block evaluates the objects in "Template Tree". The objects in the tree can be in many forms, such as "ProtocolDef", "FragmentDef", "FieldDef" or "Sequence". All of these types of objects are passed to our evaluation engine. The engine will use the definition objects to construct appropriate types of objects which will be used to hold the incoming data. A "Protocol" type of object will be constructed according to "ProtocolDef"; a "Fragment" type of object will be constructed according to "FragmentDef"; a "Field" type of object will be constructed according to "FieldDef"; "Sequence", "Link" and "Compare" will be evaluated using the "Linkable Fragment Format" approach described in the "Protocol Template" section. The entire "Template Tree" is consumed by the evaluation process and a "Data Tree" will be produced as a result. The evaluation code is listed below for reference.

```
/// <summary>
/// Returns the evaluated value from the value's actual
/// </summary>
/// <returns>The datatype evaluated</returns>
public static IDataType Evaluate(Forth theForth, IBehaviour theInstance)
{

    IBehaviour aValue = theInstance;
    //Evaluate the value recursively until we get a datatype or null
    while (true)
    {
        //Evaluate the value's actual to find the value
        aValue = aValue.Def.Execute(theForth, theInstance);
        // Value is null, return
        if (aValue == null)
            return null;
        // We get a datatype, return
        IDataType aDataType = aValue as IDataType;
        if (aDataType != null)
            return aDataType;
    }
}
```

2) Verification

The results from the "Evaluation" block are used to store incoming data frames. As we have defined in our program, only a "DataType" object can store a data value. Therefore the "Verification" block is responsible for verifying the output objects from the "Evaluation" block. If the object is not a "DataType" object, we place it in the "Data Tree" straight away because it is treated as the structural object. For instance, a "Fragment" type object cannot be used to store data, its purpose is to provide structure for the "Data Tree" so we can see the structure of the incoming data. We recursively evaluate the objects in the "Template Tree" until a "DataType" object is found. The found object is passed to the "Binary Reader" block to read its actual data value from the incoming data frame.

3) Binary Reader

Once an object has passed the "Verification" block, the parser will treat it as a "DataType" object. This object will be passed to the "Binary Reader" block which reads the incoming data in binary format as follows. The object value is then placed in the "Data Tree" for future use in our software engine.

```
    /// <summary>
    /// Construct a Behaviour using the specified Construct
    /// </summary>
    public static IBehaviour ConstructBehaviour(IBehaviourDefinition
theConstruct, Stream theStream, IEntity theParent, IProperty theProperty)
    {
        //Construct a Behaviour from the Construct
        IBehaviour aBehaviour = theConstruct.Construct();
        //Create a property to hold our behaviour
        IProperty aChildProperty = new Property(theProperty.Id, aBehaviour);
        //Add the property to its parent
        theParent.Add(aChildProperty);

        //Check the type of the Behaviour
        IDataType aDataType = aBehaviour as IDataType;
        if (aDataType != null)
        {
            //Read the Child from the Stream.
            aDataType.Def.ReadFromStream(aDataType, theStream);
        }

        return aBehaviour;
    }
```

The structure of the "Data Tree" is similar to the structure of the "Template Tree". The "Template Tree" is constructed based on the protocol template. We have also inserted behavioral and descriptive tags such as "Link", "Sequence" or "Compare" in the protocol template. The protocol template includes all the possible scenarios which may occur in the incoming data packets, when there are choices to be made. However, only one case will occur in a specific data frame. The "Data Tree" is constructed based on the data we have received. It follows the structure of the "Template Tree" but without the extra descriptive features such as "Link", because these have now been evaluated.

Chapter 5 - Testing

5.1 User Interface

We have built a customized .NET user interface (Figure 32) for testing the performance of our software engine.

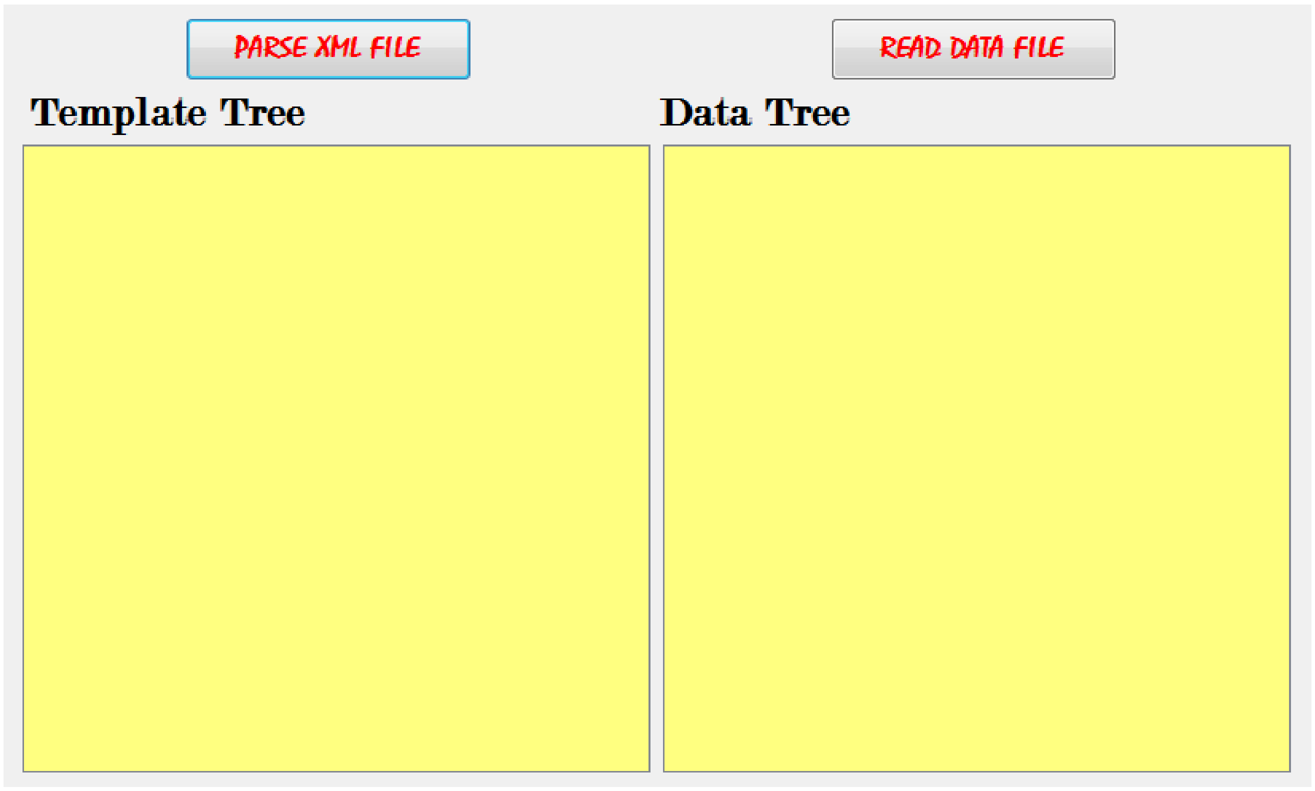
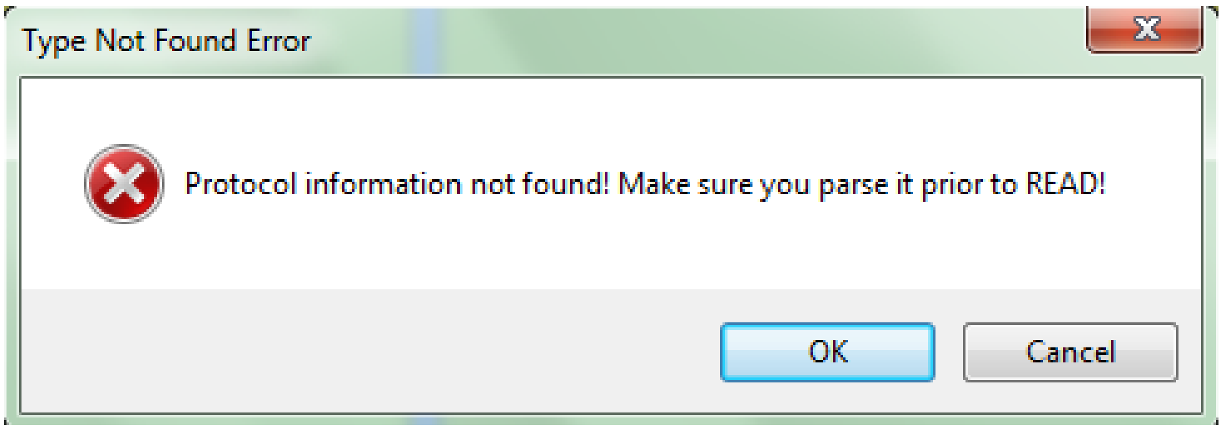


Figure 32 Testing User Interface

There are two buttons on the main form for parsing both protocol template and incoming data frame. Another two user customized control forms are placed below the corresponding buttons for displaying the structure information in .NET tree view format from the parsed files. On the left side, the template tree constructed from the protocol template file will be displayed. On the right side, the data tree constructed from the testing data file will be displayed. We must have the protocol template parsed prior to parsing any data files. Otherwise the warning window (Figure

33) will be pop up and force us to parse a protocol template file first. This protects our engine from attempting to parse any data before it has the structure of the data.


Type Not Found Error

Protocol information not found! Make sure you parse it prior to READ!

Figure 33 User Interface Warning Window

We have also used four different icons which can help us distinguishing the difference between objects in the tree view diagram.

- □ "Behaviour" type object.
- ○ "DataType" type object.
- ↓ "Entity" type object.
- ⊘ "Property" type object.

5.2 Testing Procedures

5.2.1 Parse a protocol template

We need to select a protocol template file from our computer. To open a file, we have to click on the "Parse XML File" button on the UI. In our case (Figure 34), we have chosen the protocol template file created for testing purposes as shown in the dialog box. This file does not fully implement the entire "iMonitor" communication protocol, but it includes all the possible scenarios which may occur in the real protocol template file. Once we click on the "Open" button, the XML file is sent to our "Protocol Parser" and the "Template Tree" will be created. Then the "Template Tree" will be displayed in the template tree user control form in tree view format. "Registrar" holds all the definitions for constructing C# objects from the protocol template. "Protocols" holds the parsed protocol template information.

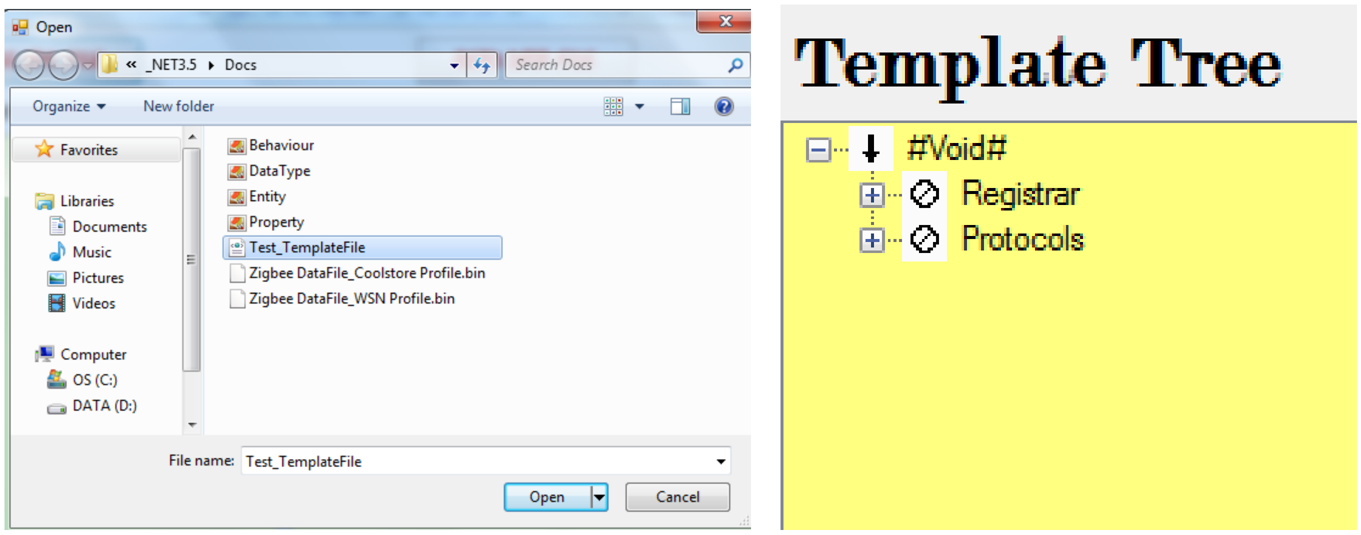


Figure 34 XML file selection and Parsing

The shape of the tree view in Figure 35 indicates how the definitions are registered in "Registrar". Figure 36 displays the structure of our testing protocol template. All the objects in the tree are constructed from their corresponding registrar path information stored in the protocol template file. This demonstrates that our "Registrar" concept works correctly for parsing our protocol template.

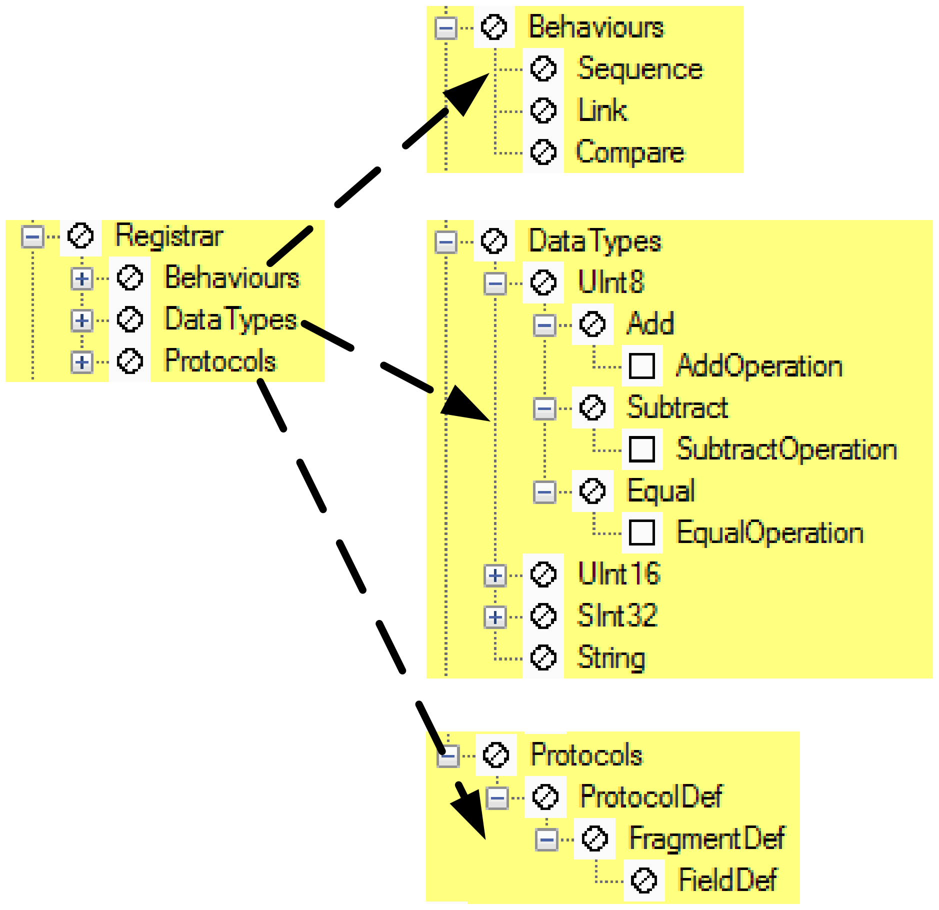
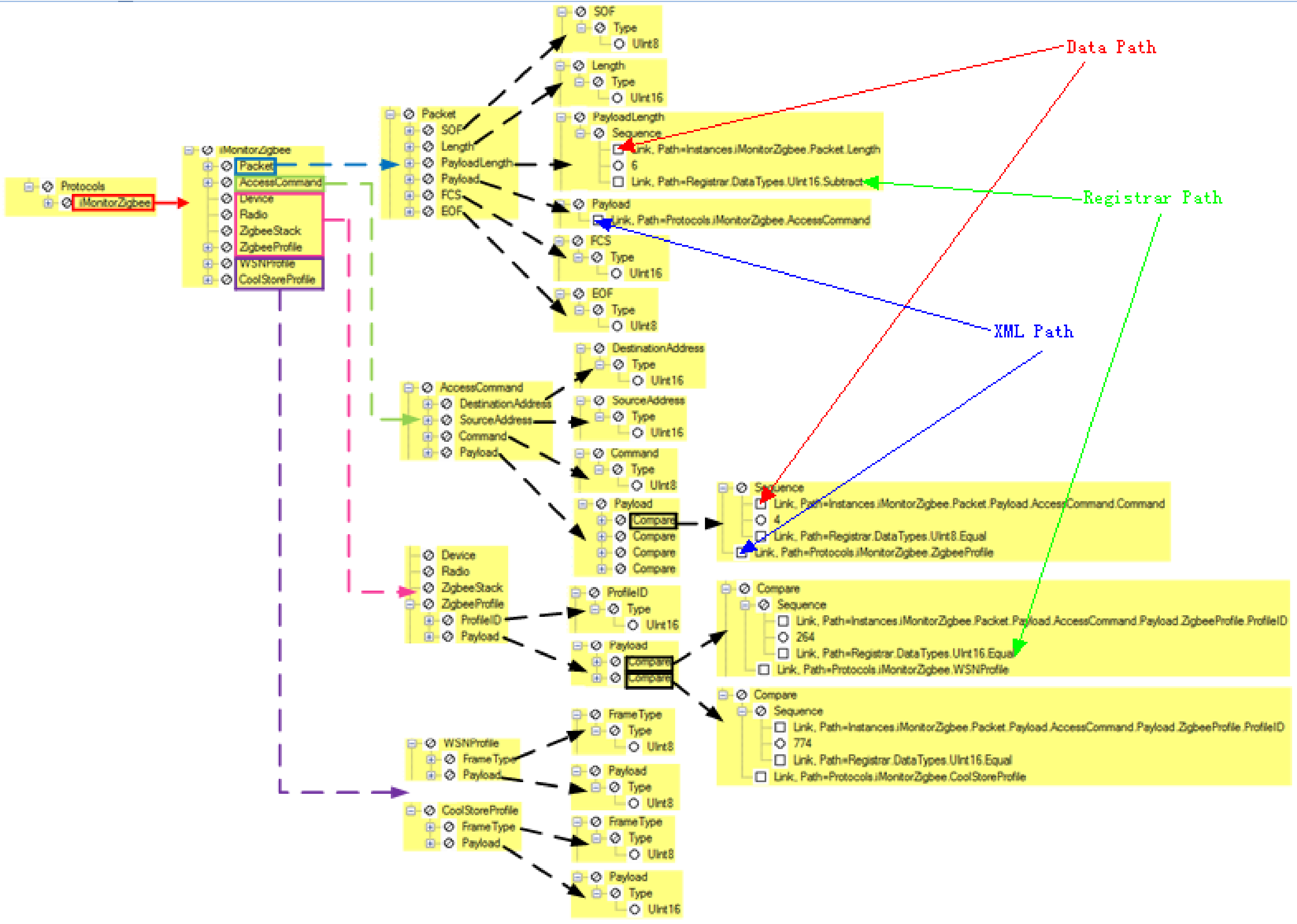


Figure 35 "Registrar" Tree View

There are many activity structures defined in the protocol template such as links, sequences and compares. These structures do not exist in the original protocol document. They are used to assist in maintaining our protocol template in our Linkable Fragment Format. These activity structures will be turned into activity objects in "Protocol Parser". These objects will be exercised and consumed in "Data Analyzer" and eventually they will disappear after "Data Tree" is constructed. We have XML path and data path implemented as shown in Figure 36. XML path is used to manage the structure of our protocol template. In Figure 36, we have inserted an XML path in the packet's payload field. Then we have moved the structure contained in the packet's payload from its original position and placed it at the same level as the packet fragment. A new fragment named "Access Command" is created for holding the payload structure we have taken out. The link, which points to the "Access Command" fragment, will be executed and whatever information we have moved will be placed back at its original location when we start reading the data frame. There is also a data path set in the "PayloadLength" field's sequence operation. This data path is used to access a data value at program runtime. The link, which points to the real data value stored in the packet's length field, will be executed in "Data Analyzer". The value returned will be used in the sequence operation to calculate the payload size. The details on how "Registrar Path", "XML Path" and "Data Path" operate can be referred to in sections 4.4 and 4.5.



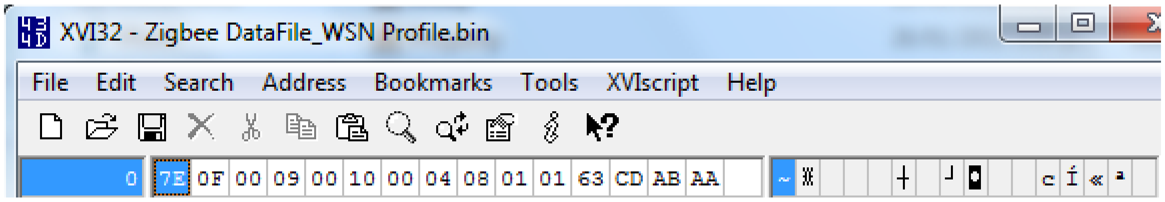
Protocols
iMonitorZigbee
Packet
AccessCommand
Device
Radio
ZigbeeStack
ZigbeeProfile
WSNProfile
CoolStoreProfile
SOF
Length
PayloadLength
Payload
FCS
EOF
Type
UInt8
UInt16
Sequence
Link, Path=Instances.iMonitorZigbee.Packet.Length
6
Link, Path=Registrar.DataTypes.UInt16.Subtract
Link, Path=Protocols.iMonitorZigbee.AccessCommand
DestinationAddress
SourceAddress
Command
Compare
Link, Path=Instances.iMonitorZigbee.Packet.Payload.AccessCommand.Command
4
Link, Path=Registrar.DataTypes.UInt8.Equal
Link, Path=Protocols.iMonitorZigbee.ZigbeeProfile
ProfileID
Link, Path=Instances.iMonitorZigbee.Packet.Payload.AccessCommand.Payload.ZigbeeProfile.ProfileID
264
Link, Path=Registrar.DataTypes.UInt16.Equal
Link, Path=Protocols.iMonitorZigbee.WSNProfile
774
Link, Path=Protocols.iMonitorZigbee.CoolStoreProfile
FrameType
Data Path
Registrar Path
XML Path

Figure 36 "Protocol Template" Tree View

5.2.2 Parse a data file

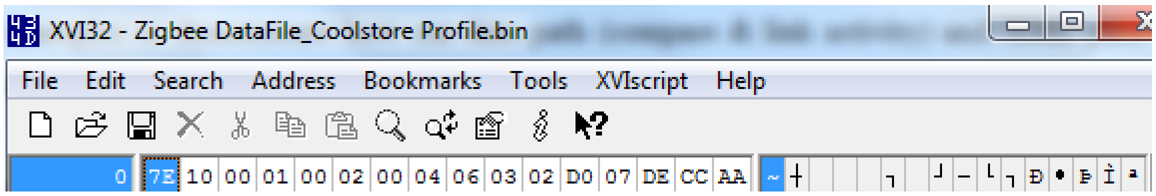
After we have the protocol template parsed and stored in our software engine, we are now ready to analyze the data frames. We need to click on the "Read Data File" button to choose a binary file that simulates a data frame. We have two binary files defined (Figure 37), to simulate two different possible data frames that can be received. The binary files were created using a binary file editor. The contents of the two data files are shown in hex below. We will use "WSN Profile" and "Coolstore Profile" payload examples to demonstrate how our data path (compare & link activity) and XML path (link activity) parse the binary data files in "Data Analyzer". Note: For 16-bit numbers, the least significant byte is stored before the most significant byte in the binary files, i.e. 0F 00 represents hex value 0x000F.

WSN Profile Payload binary data


XVI32 - Zigbee DataFile_WSN Profile.bin
File Edit Search Address Bookmarks Tools XVIscript Help

```
0  7E 0F 00 09 00 10 00 04 08 01 01 63 CD AB AA
```

Coolstore Profile Payload binary data


XVI32 - Zigbee DataFile_Coolstore Profile.bin
File Edit Search Address Bookmarks Tools XVIscript Help

```
0  7E 10 00 01 00 02 00 04 06 03 02 D0 07 DE CC AA
```

Figure 37 Actual Binary Data Payload

According to the "iMonitor" protocol template, the data frame structure is as shown in Figure 38. The main frame contains five fields: SOF, Length, Payload, FCS and EOF. We have added an extra field called "PayloadLength" in the "Protocol Template" to record the size of the "Payload" which can only be calculated at runtime. This field does not exist in the original protocol document. We use it for our own reference and to demonstrate in the test how a data value can be accessed and linked with a "Data Path" at runtime. The "SOF" and "EOF" fields mark the start and the end of the entire data frame. They will always have the fixed 8-bit hex value 0x7E (decimal value 126) and 0xAA (decimal value 170) respectively. The "Length" field indicates the size of the entire data frame. This 16-bit value can vary and depends on what type of payload data we may receive in the data frame. The "FCS" field performs error checking and stores a 16-bit result value. It will ensure the data frame received is not corrupted. The "Payload" field contains the first level payload content, named "Access Command" in the data frame. We will look through each level in detail in this demonstration.

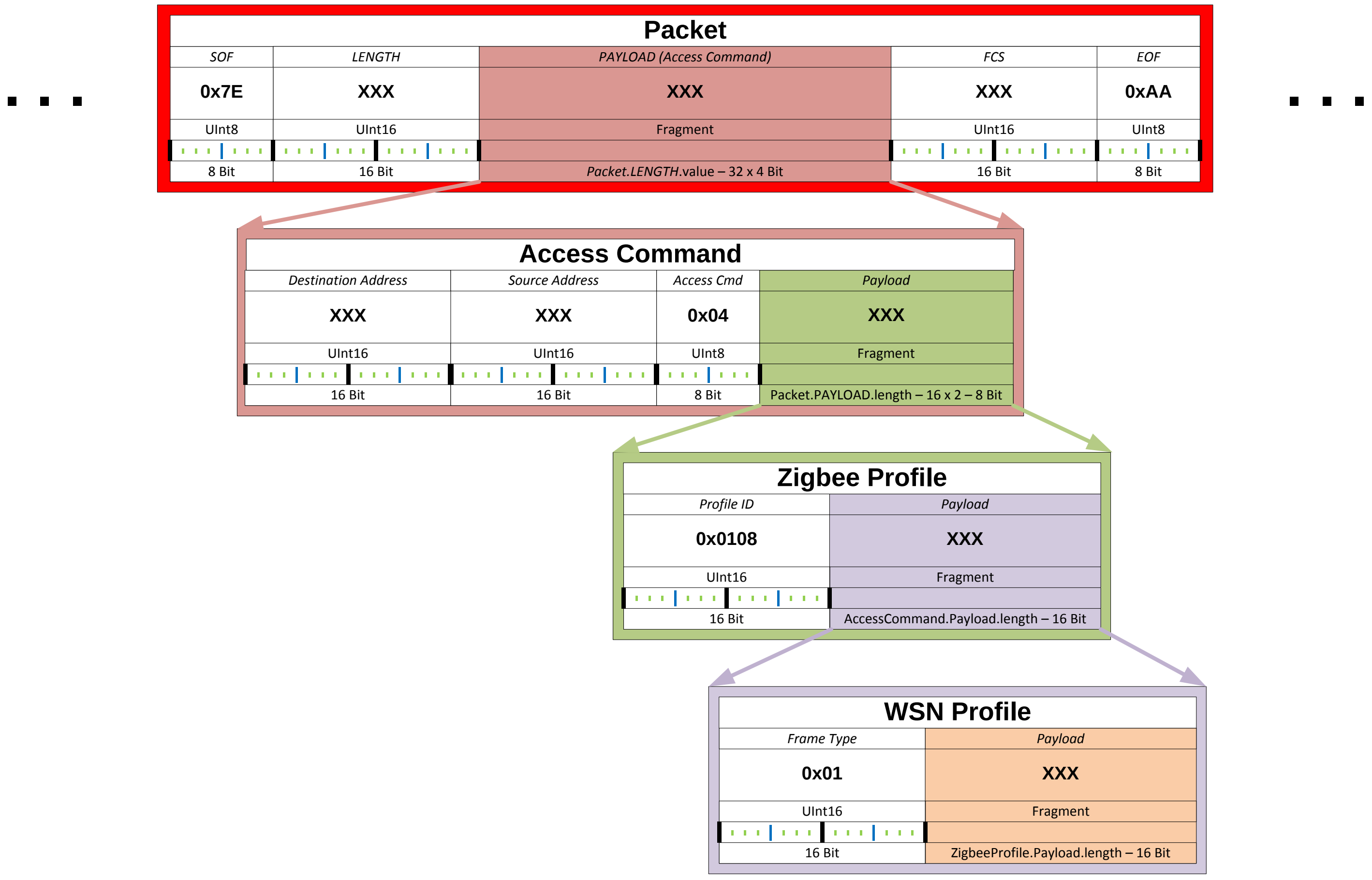


Figure 38 "WSN Profile" Data Frame

"Access Command" is formed from four fields: Destination Address, Source Address, Access Cmd and Payload. The "Destination Address" and "Source Address" fields contain 16-bit address values for the data source and data destination. The 8-bit value stored in the "Access Cmd" field will decide the type of payload at the second level payload content. In this case, a hex value 0x0004 (decimal value 4) indicates that the second level payload will be "Zigbee Profile". There are also other fixed values available for "Device", "Radio" and "ZigbeeStack" payload content. "Zigbee Profile" contains "Profile Id" and "Payload" fields. The "Profile Id" field indicates the type of profile payload. A hex value 0x0108 (decimal value 264) indicates the third level payload content is "WSN Profile". "WSN Profile" contains "Frame Type" and "Payload" fields. For demonstration purpose, we have given the "Frame Type" field a fixed 8-bit hex value 0x01 (decimal value 1) and the "Payload" field a fixed 8-bit value 0x63 (decimal value 99) rather than another level of payload content. Figure 39 shows the "WSN Profile" data file in blocks according to the protocol template layout. The red rectangular box is the data packet received with "SOF" value 0x7E (decimal value 126) and "EOF" value 0xAA (decimal value 170) marked in blue background. The "Length" field value 0x000F (decimal value 15) means the size of the data frame will be 15 bytes long. The light red box is "Access Command" payload with two addresses of 0x0009 (decimal value 9) and 0x0010 (decimal value 10) and "Access Cmd" value 0x04 (decimal value 4) which leads us to the "Zigbee Profile" payload. The green box is the "Zigbee Profile" payload with "Profile ID" value 0x0108 (decimal value 264) which indicates the payload is "WSN Profile". The purple box is the "WSN Profile" payload with "Frame Type" value 0x01 (decimal value 1) and final "Payload" value 0x63 (decimal value 99).

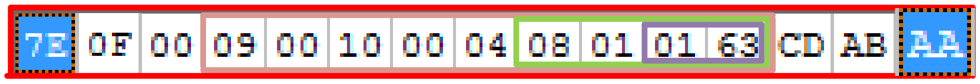
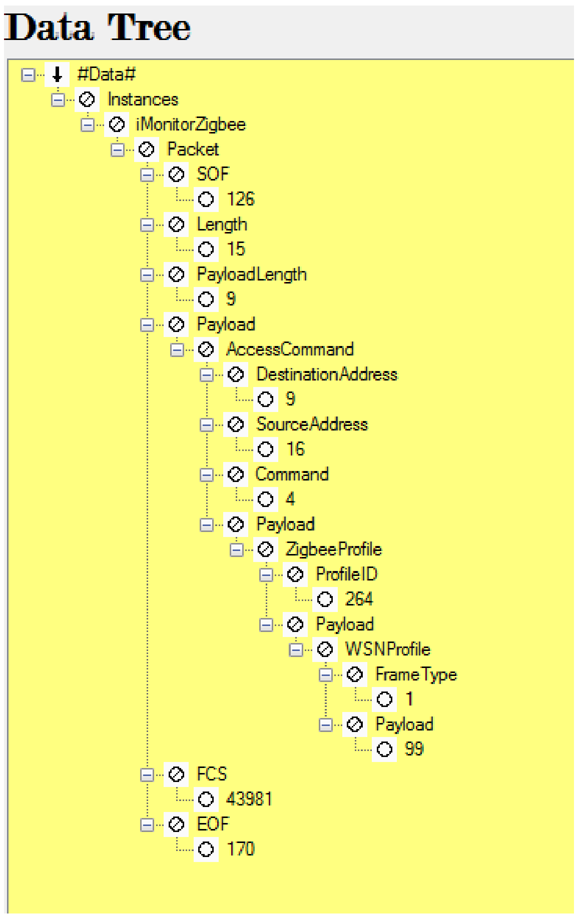


Figure 39 "WSN Profile" Data in Binary Format

Once this data has gone through the "Data Analyzer", the "Data Tree" will be populated in the tree view diagram (Figure 40) according to the values we have listed above.

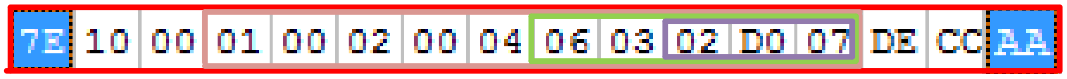


```
Data Tree
#Data#
  Instances
    iMonitorZigbee
      Packet
        SOF
          126
        Length
          15
        PayloadLength
          9
        Payload
          AccessCommand
            DestinationAddress
              9
            SourceAddress
              16
            Command
              4
            Payload
              ZigbeeProfile
                ProfileID
                  264
                Payload
                  WSNProfile
                    FrameType
                      1
                    Payload
                      99
        FCS
          43981
        EOF
          170
```

Figure 40 "WSN Profile" Data Tree View

The value displayed on each branch of the tree accurately matches each individual field value defined in the "WSN Profile" data file. This demonstrates all our paths and activity objects created in the protocol template work correctly. As can be seen, the structure of "Data Tree" no longer keeps any of the path nor activity objects. The structure of "Data Tree" exactly follows the structure of the original protocol document. All the fragments we have managed and moved around are now back to where they were in the actual data frame. For example, the "Access Command" payload is now displayed under the packet's payload field. It is no longer a standalone fragment structure but a data branch of the payload. The same applies to both the "Zigbee Profile" and the "WSN Profile" fragments.

Now consider the input data file to the "Coolstore Profile" as shown below.



We will have the "Coolstore Profile" data branch displayed off the "Zigbee Profile" rather than the "WSN Profile" (Figure 41). This occurs because the value 0x0306 (decimal value 774) defined in the "Profile ID" field from the "Zigbee Profile" segment is linked into the "Compare" activity in the protocol template at runtime. After the comparison process, the payload decision made by the engine is based on the real time value which indicates the type of payload is "Coolstore Profile". The "Coolstore Profile" payload contains "Frame Type" value 0x02 (decimal value 2) and final "Payload" value 0x07D0 (decimal value 2000).

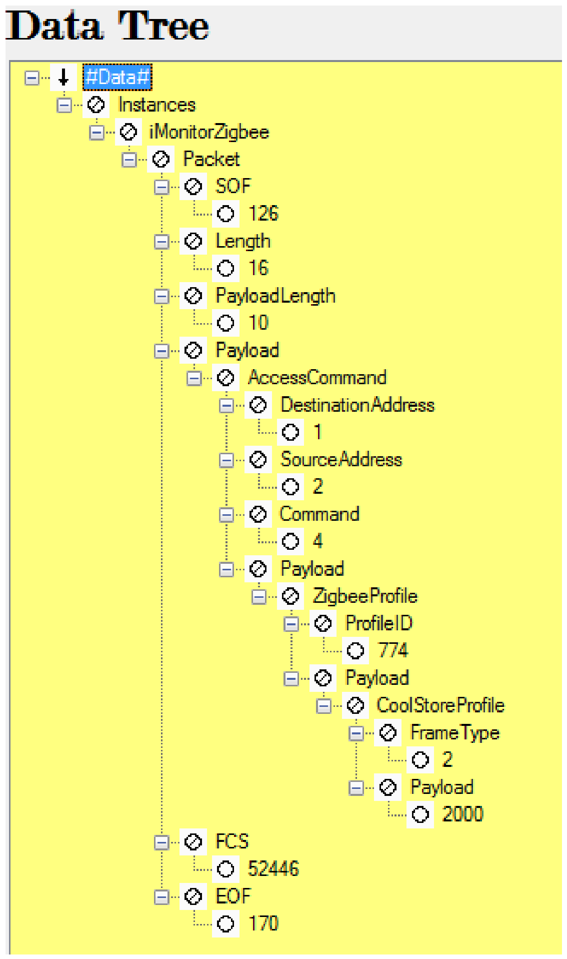

```
Data Tree
#Data#
  Instances
    iMonitorZigbee
      Packet
        SOF
          126
        Length
          16
        PayloadLength
          10
        Payload
          AccessCommand
            DestinationAddress
              1
            SourceAddress
              2
            Command
              4
            Payload
              ZigbeeProfile
                ProfileID
                  774
                Payload
                  CoolStoreProfile
                    FrameType
                      2
                    Payload
                      2000
        FCS
          52446
        EOF
          170
```

Figure 41 "Coolstore Profile" Data Tree View

Chapter 6 - Conclusion and Future Work

6.1 Conclusion

This research introduced a novel software engine implementation for configuring hardware without reprogramming. The hardware platform we have chosen is the Tahoe-II development board which runs .NET Micro Framework. The benefit of using .NET MF is that we are able to focus on the application part of the system. Using the familiar Microsoft Visual Studio development suite, our C# applications can be developed independent of the hardware platform. .NET MF also provides tight Visual Studio integration and an extensible emulator that lets us run and test our embedded applications right on our PC. The software parser we have implemented is able to convert any particular protocol written in XML format into the more generic form of C# objects. These objects themselves are interrelated and they also can be reconstructed back into their original protocol format or some other protocol format, depending on the final end point requirements. The idea is to have this intelligent software engine running on the Tahoe-II board (gateway) which is able to control data flow between the independent links across the network. The gateway is to provide data flow management regardless of the data protocol or the link where the information is originated from or sent to. The software engine core consists of two main parsers - the protocol parser and the data analyzer. "Protocol Parser" converts the protocol template which is produced in XML to internal reconfigurable C# objects. These objects are used as instructions to help in processing the incoming data from the various data links in "Data Analyzer". The data content will be examined and then transformed into packets appropriate for the user required data link.

A testing protocol template file which implements all the possible scenarios in any real protocol template has been successfully parsed by the "Protocol Parser". Then a data file which simulates real data frames off the network has been analyzed by the "Data Analyzer". The data is accurately parsed based on the information provided by the XML protocol template. All the test results from the research have clearly shown that the Linkable Fragment Format (LFF) derived in this research is an accurate and flexible approach for managing and manipulating the structure of an XML protocol template.

The XML protocol template developed in this research is different from other regular XML protocol documents. The important new part we have developed is that we have embedded the "Interconnection" concept into the XML elements. The traditional XML element only contains a word or a phrase to describe the particular XML element or its content within the XML < > tag. We have attached a series of characters and symbols which represents a path in the XML element name instead of a simple word or phrase. The path is the key to finding the interrelationship between the current XML element and the other related XML elements. When the XML file is getting processed by the "Protocol Parser", the path will be executed and it acts as a pointer which points to the object at the specific location.

By having the path, we can easily customize the formation of the XML protocol template. In the traditional XML protocol document, the structure of the document must strictly follow the order of the protocol. But now we can place a block of information which describes a specific protocol mechanism anywhere we like within the entire protocol document, as long as we have a path pointing to its location. We can then access that particular piece of information from anywhere in the XML template by executing its path.

With the concept of interconnection embedded in the XML element, we are able to assign more relationships between the XML elements rather than just having a parent and child relationship as usual. The traditional XML document usually just has a simple relationship which provides a way of recognizing the sub-elements of the current element. We have successfully defined and tested more complex relationships such as "Link", "Sequence" and "Compare" within our XML protocol template. We have called these relationships "Behaviours" because each of them performs a special behaviour in the XML protocol template. By having these behaviours, we are able to integrate the Forth stack operation and branch selection mechanism into the XML protocol template. The test results show the behaviours in our protocol template provide a flexible mathematical calculation process and improve the decision making process.

The XML language has a highly descriptive characteristic advantage over other languages. The path mechanism introduced and integrated with the XML element in this research actually gives the user a systematic way of storing and acquiring the information on any element in the XML document. We have implemented a "Registrar" entity which keeps a record of definitions of all available objects within the scope of the current document. It defines the range of objects that the software engine is capable of creating. When the "Protocol Parser" processes the XML protocol template, it always refers to the "Registrar" entity and looks there for the information on how to create the corresponding C# object. This way of registering and searching structure information really makes the entire parsing process more flexible compared to the traditional XML processing mechanism.

Our test results have shown the XML protocol template structure built in this research is a significant enhancement the traditional XML protocol document that increases the flexibility of the protocol document.

6.2 Future Work

While this research has been successful in creating a flexible new XML template structure and associated data processing mechanisms, there are a number of developments that could build on this research.

More generic data type classes can be created and investigated to handle different types of more complicated incoming data frames such as byte array, etc.

Further research is required to define the user subscription rules and regulations which allow users to subscribe to their required data content.

A more intelligent routing scheme could be developed based on the subscription rules applied to the incoming data frames. This mechanism is known as content-based routing. The user subscribes to particular data content embedded within the incoming data frames. These data get parsed and analyzed by the "Protocol Parser" and the "Data Analyzer", then the subscription rule is applied to the processed data and the data is transformed into the protocol format required by the user. Finally, the converted data will be routed to the required user end point.

The methods for converting the objects in the data tree into the appropriate end point data packets needs to be undertaken, by reversing the logic structure of the "Data Analyzer".

Bibliography

[1] Ajay Mane, D Ramesh Babu, M Chetan Anand. (2004). XML Representation of Spatial data. *IEEE INDIA ANNUALCONFERENCE 2004*, (pp. 67-69). INDICON.

[2] Antonio Carzaniya. Matthew J. Rutherford. and Alexander L. Wolf. (2004). A Routing Scheme for Content-Based Networking. (pp. 918-928). Boulder. Colorado: IEEE.

[3] Anwar, Y. ; Kamel, A. ; Ahmed, A.S. (28-30 April 2009). Grove data model for efficient representation of XML documents. *WOCN '09. IFIP International Conference*, (pp. 1 - 6). Cairo.

[4] Bianchi, S; Felber, P; Gradinariu Potop-Butucaru, M. (2009-08-18). Stabilizing Distributed R-trees for Peer-to-Peer Content Routing. *Parallel and Distributed Systems, IEEE Transactions on : Accepted for future publication* .

[5] Boshi SUN, Xiaojie YUAN, Hong KANG, Xiaocheng HUANG and Ying GUAN. (20-22 Aug. 2010). Incremental Validation of XML Document Based on Simplified XML Element Sequence Pattern. *Web Information Systems and Applications Conference (WISA)*, (p. 110). Hohhot.

[6] Chand, R., & Felber, P. (16-18 April 2003). A Scalable Protocol for Content-Based Routing in Overlay Networks. *Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium*, (pp. 123 - 130).

[7] E. Anceaume, A.K. Datta, M. Gradinariu, G. Simon, and A. Virgillito. (2006). A semantic overlay for self- peer-to-peer publish subscribe. *the 26th International Conference on Distributed Computing Systems (ICDCS 2006).*

[8] Gong Li ; Liu Gao-Feng ; Liu Zhong ; An Ru-Kui . (21-24 May 2010). XML processing by Tree-Branch symbiosis algorithm. *Future Computer and Communication (ICFCC), 2010 2nd International Conference,* (pp. V1-669). Wuhan.

[9] James Moscola, Young H. Cho, John W. Lockwood. (23-25 Aug. 2006). A Reconfigurable Architecture for Multi-Gigabit Speed Content-Based Routing. *High-Performance Interconnects, 14th IEEE Symposium,* (p. 61). Stanford, CA.

[10] Lelli, F. ; Maron, G. ; Orlando, S. (18-22 Sept. 2006). Improving the performance of XML based technologies by caching and reusing information. *Web Services, 2006. ICWS '06. International Conference,* (p. 689). Chicago, IL.

[11] Takase, T. ; Tajima, K. (23-26 Sept. 2008). Lazy XML Parsing/Serialization Based on Literal and DOM Hybrid Representation. *Web Services, 2008. ICWS '08. IEEE International Conference*, (p. 295). Beijing.

[12] Toshiro Takase , Hisashi MIYASHITA , Toyotaro Suzumura , Michiaki Tatsubori. (2005). An adaptive, fast, and safe XML parser based on byte sequences memorization. *14th international conference on World Wide Web* (pp. Pages: 692 - 701). Chiba, Japan: ACM.

[13] Xiaolin Zhang ; Guofeng Zhai ; Rong Tian. (18-20 Sept. 2009). XML Schema Based Compression Technology over XML Data Stream. *Web Information Systems and Applications Conference, 2009. WISA 2009. Sixth* (pp. 27 - 31). Xuzhou, Jiangsu: IEEE.

[14] Xiaosong Li ; Hao Wang ; Taoying Liu ; Wei Li ; . (8-11 Dec. 2009). Key Elements Tracing Method for Parallel XML Parsing in Multi-Core System. *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference,* (p. 439). Higashi Hiroshima .

[15] Yi-Min Wang, Lili Qiu, Chad Verbowski, Dimitris Achlioptas, Gautam Das, and Paul Larson. (October 2004). Summary-based routing for content-based event distribution networks. *SIGCOMM Computer Communication Review* , Pages: 59 - 74 , Volume 34 , Issue 5.

[16] Zhenghong Gao ; Yinfei Pan ; Ying Zhang ; Chili, K. (10-13 Dec. 2007). A High Performance Schema-Specific XML Parser. *e-Science and Grid Computing, IEEE International Conference*, (p. 245). Bangalore.

[17] Zhou, D. (14-18 July 2008). Exploiting Structure Recurrence in XML Processing. *Web Engineering, 2008. ICWE '08. Eighth International Conference* (p. 311). San Jose, CA: IEEE.

[18] iMonitor Research Limited. http://www.imonitor.co.nz/

[19] Tahoe II Development Board. http://devicesolutions.net/Products/Tahoe.aspx

[20] Zigbee Protocol. http://www.zigbee.org/