AN ARCHITECTURE FOR INTEGRATING MULTIPLE REAL TIME DATA FEEDS

Neil Roodyn

A thesis submitted for the degree of

Doctor of Philosophy

University of London

Department of Computer Science
University College London

1998

Abstract

This thesis investigates 'industry strength' solutions for integrating multiple real time data feeds. It addresses one of the major problems challenging computing namely 'information overload'. The thesis focuses on those problems faced by a user of multiple data sources, either real time data feeds or databases. The thesis' goal is a system architecture providing a generic interface which allows the connection of one or more data sources to a unified collection point. On entering the system the data is screened and filtered for information which is pertinent to the end user. This data is then available for other programs that use the data, through a standard interface.

The thesis comprises four parts: a critique of existing commercial real time products, and then the invention of three experimental real time systems, constituting the core of this research. These systems have been created in collaboration with industry, providing unique end user feed back, and also ensuring a degree of compliance with industry standards. The three experimental systems comprise: (i) a 16 bit real time system for providing private investors with financial information; (ii) a 32-bit system for tracking buses in real time; (iii) a 32 bit 'generic' real time system which caters for multiple feeds. The three systems have all been designed and implemented by the author, and the systems are now being tested in a commercial environment.

The critique examines several commercial real time systems and analyses their key features relating to information overload. It studies the fundamental aspects of real time data processing with specific reference to the industry standard Microsoft Windows in the PC environment.

The second part describes the design and implementation of a 16-bit real time system for providing end users with stock exchange data from one or more data sources. To provide experimental data on the use of the system it was built in collaboration with Updata Software. It utilises 16 bit DLLs for multiple process information sharing and providing data in a 'soft' real time manner.

The third part describes the design and implementation of a 32-bit real time system. This system provides feedback on vehicle positions and with the aid of Hampshire County Council is being tested for passenger information systems on buses. It utilises shared global memory blocks to share information between processes.

The final part presents the new 'generic' system architecture, whose design draws on experience gained from the previous two systems and the feedback, provided from users. Lessons learnt from previous systems include the need for integration of filtering into the system and providing an open interface to ease data input from new data feeds. This system also addresses the need for compliance with industry standards. It makes strong use of object architectures and, being WOSA compliant, provides COM interfaces to satisfy the need to share information and provide an open interface. The two chapters covering this work contain the design of the system and the strategy employed to implement the design, and then test the system.

The thesis and the software implemented make three contributions to science: i) it reduces information overload by integrating the data from multiple sources and providing a single interface for accessing that data, ii) it provides techniques for bringing real time data to the personal computer desktop, and iii) it helps to personalise the data by providing a simple filtering mechanism and an interface for more complex filtering systems. An innovation of this thesis is that the implementations of the designs have been taken through to industry products, in order to provide feedback.

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

The author wishes to thank Professor Philip Treleaven for his advise and guidance on this work, and Professor Anthony Finkelstien for his comments on the first draft of this thesis.

I also would like to thank the people from industry who aided in this work. Amongst them are; Graeme Ellisson for his help at Updata Software, Joe Field and Jeff Hayes for their input to the Bus Tracking system, and the clients of Black Ace Software Engineering.

Chapter 1

INTRODUCTION

This chapter presents an introduction to real time systems, the issues surrounding information overload and how real world applications currently cope. It also contains the aims, contributions and organisation of this thesis. The major contribution of this thesis is to present a software architecture that integrates multiple real time data feeds.

1.1. An Architecture for Multiple Real Time Data Feeds

With the growing amount of information available to end users on personal computers there is an increasing need to integrate this data. Much of this data is available in the form of a stream of data, known as a data feed. This thesis presents three experimental architectures for integrating multiple real time data feeds. Each architecture builds upon its predecessor in order to obtain a more generic architecture. These architectures address the issue of 'information overload', which is one of the major problems challenging computing today. The thesis focuses on solving the problems faced by a user of multiple data sources, either real time data feeds or databases. The thesis' goal is to create an architecture that provides a generic interface allowing the connection of one or more data sources to a single collection point. This architecture will allow the data to be filtered for information that is relevant to the end user. This data is then made available for other programs to use through a standard interface. It is this architecture that integrates multiple real time data feeds that is the main contribution of this thesis to solving the problem of information overload.

1.2. Information Overload

Jan Wyllie, an industry pundit, recently made the following comment in a paper entitled 'Turning information overload into useful knowledge resources' [5]:

'Information is a resource which - unlike physical resources - is not scarce. People are overwhelmed by the sheer volume – huge and ever growing numbers of newspapers, television programs, learned journals and conversations. The World Wide Web and email simply make a desperate situation worse.'

It highlights the ever growing problem of information overload; there is too much information for the user to cope with. Nowhere is this more apparent than in the domain of financial information systems, where along with thousands of stock prices updated every second, a vast quantity of news and financial reports are available, an example of this can be seen in figure 1-1.

G. Miller states that the human mind can only simultaneously cope with seven (plus or minus two) chunks of information [49]. It seems unlikely that humans will be able to absorb more information so the problem may be alleviated by simply reducing the amount of data with which users have to handle [16].

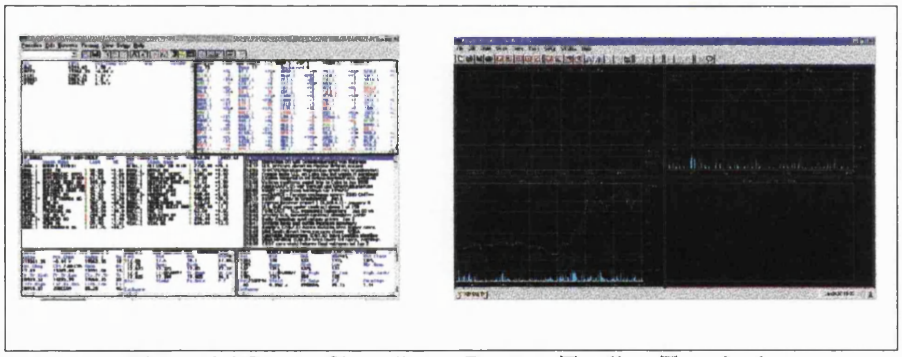


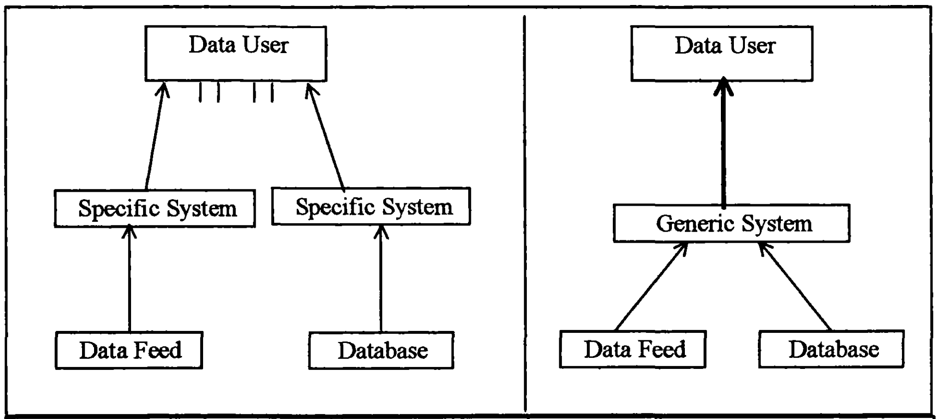
Figure 1-1 Screen Shots From Reuters Trading Terminal

This approach is ideally suited to be carried out by a computer system, rather than the end user. The computer can sift through vast quantities of data in order to retrieve, or filter out, that which is useful information for the end user. This solution also fits in with current industry thinking on providing a single interface for multiple tasks; one of the reasons Microsoft's Windows has been so successful.

There are some systems that attempt to address this problem within specific information domains. An example is email systems that provide filters based on a set of rules entered by the users, such as Microsoft Exchange. There is not a generic system available that allows new filters to be simply added to an existing system without being concerned about the information domain.

One area, in which much work has been carried out to lessen the impact of information overload, is that of stock market surveillance systems. The quantity of information, provided by a stock market, can be too excessive to monitor without the aid of filtering tools. Many of the tools available concentrate on filtering the data from a single proprietary feed.

It was this problem of information overload which inspired the ideas for this thesis, to provide a single collection point for all data on the desktop, so that the data may then be filtered for information which the user actually wanted to see. Figure 1-2 illustrates the concept of providing a unified collection point as shown in B, where the generic system software provides the task of collating the data, rather than the data user.



A. Multiple Systems, one for each data source.	B. Single Generic System, collects and filters from multiple data sources.

Figure 1-2 Comparison of multiple systems to single sytem

1.3. Real Time Systems

This thesis addresses the issue of information overload with real time systems. In general 'systems' can be broken down into three main groups:[1]

1. Batch: The response time is not important to the user and often could be hours or even days.

2. On-line: Results are expected within a short time scale of seconds or minutes, but the time is not critical.

3. Real time: The results must be delivered within a definite short time, milliseconds to seconds, or the system will not work.

Batch systems usually require lengthy processing using resources that are scarce or expensive. Quite often the user will pre process information on a local computer before sending a job to a remote computer. These jobs will usually be sent in groups or batches, hence the name. The results will then be sent back to the local computer when the job has completed. Due to the non-urgent nature of this type of processing, transmissions may be queued until communication rates are cheaper, or more processor time is available.

On-line systems often don't use local processing, but instead rely on terminals communicating with a central computer, which handles transactions using some from of time slicing. These are sometimes known as 'soft' real time systems, since they are not always time critical, and the response time is often dependent upon the amount of activity.

Real time systems on the other hand are always time critical, and are sometimes called 'hard' real time systems. With these systems a delayed response in considered unacceptable.

Although there are a number of architectures for real time systems there is not one that exists to solve the problem of collecting data from multiple sources on a widely used operating system (such as Microsoft Windows). The major reason for this is that such operating systems are not designed to support such systems. The architectures provided in this thesis attempt to overcome the shortcomings of the Microsoft Windows operating system, with respect to handling real time data.

1.4. Real World Applications

The information overload problem can currently be seen in many areas. One of which is the financial markets where an enormous amount of data is already available for PCs, in the form of stock prices, news and financial reports. This data is generally incomprehensible as simply prices with times and dates, and is best viewed using charts or some other analysis tools. It is also useful to be able to view the fluctuations in the market with respect to other events, which take place in the World such as the News. Certain applications, like Reuters 2000 and Bloomberg, are starting to provide some functionality in this area, although generally they are targeted at one proprietary data feed. They work by linking into a proprietary API, which is controlled by the feed provider. This then forces the user to use that one data source, and does not permit comparisons (except manually) with other information providers.

Another area in which information overload is already becoming a problem is the Internet, with email, the World Wide Web and news groups providing a seemingly endless stream of information to sort through[2]. This is possibly the area in which most work has been done in order to help alleviate the current situation of information volume by providing various means of sorting and filtering. But mostly this processing is done off-line after the data has been received and not in real time as it is collected.

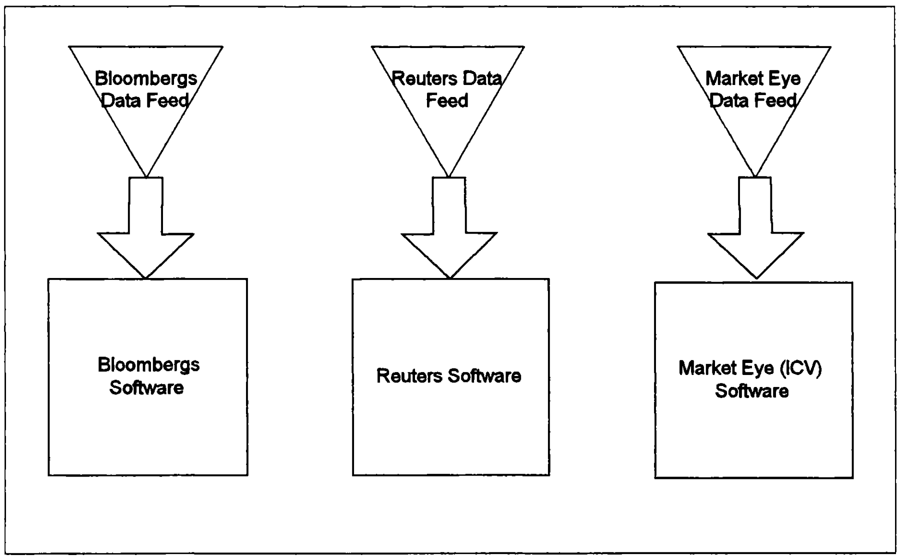


Figure 1-3 Each data supplier provides a proprietary system to view the data

M.A.I.D. plc, which is currently one of the leading suppliers of on-line business information, has realised there is a problem and has created a business out of working towards a solution. Dan Wagner, Chief Executive of M.A.I.D. plc stated in a press release[14]:

> *"Two of the key problems in the online industry have been information overload and unpredictable, variable costings. M.A.I.D have met these problems head on and now have in place a series of products that will give companies more control over their information budget without any of the problems of information overload."*

But once again this solution is only for data provided by M.A.I.D. over the Internet, and although this is currently quite extensive, it doesn't cover everything which is available. It isn't clear how to plug a third party data source into the system, or even if it's possible.

What is required is a more open and generic system which isn't specifically targeted at a proprietary data feed, but which provides an interface to allow any third party feed provider enter data into the system.

1.5. Research Motivations

This thesis was initially conceived, to create an architecture to receive data from various information sources (some live and real time), store the data, allow for the retrieval of that data and display it. Initially the data was to be financially based only with a view to expanding the system to cope with other types of data in the future. It quickly became apparent that it would be a far more powerful tool if the data could be of any type. So a generic system to handle all real time data types was written.

It was also realised that in order to be of any use the system being designed would have to be usable by the majority of the people suffering from the information overload problem. To this end it would be pointless designing a system that provided a specific programming language style interface, it would have to be configurable by a non technical person.

Microsoft has already provided guidelines on designing simple real time systems under their Windows systems using COM (Component Object

Model), represented in the Windows Open System Architecture (WOSA) extensions for real time (XRT). These would have to be followed in order to be compatible with any other systems that follow these guidelines.

This thesis provides an architecture that extends the WOSA guidelines by adding a more detailed storage hierarchy, allowing for simple data input and catering for historical data. This architecture has the following features:

[1] Multiple sources of data can be input simultaneously. The data can be of multiple types and can be linked together to provide a simple way of traversing through it.

[2] The data can be filtered for information that is pertinent to the end user. A standard interface is provided to easily allow new filtering tools to be added. The system starts by selecting only the information that is actually chosen by the user to be collected.

[3] An interface is provided in order to retrieve the data which has been filtered. Multiple data clients can obtain the either all or a subset of the filtered data.

[4] Data clients can request to be notified of new filtered data. In this way clients can easily be written which take an action upon a change in the data being filtered.

1.6. Thesis Organisation

This thesis is organised into 8 chapters.

Chapter Two - Real Time Processing of Data Feeds, surveys existing systems, and their usage. These include Reuters, Bloombergs and Fairshares. Also covered are issues involving real time data processing on personal computers running Microsoft Windows, this is important as Windows now presents the majority of installed users in the World and it is deemed commercially important to provide a system that most users can adopt.

Chapter Three - Real Time System For Private Investment, presents a new architecture for integrating multiple real time data feeds and contains a case study of the design and implementation of a 16 bit real time system written for the private investment market, and currently marketed by Updata software Ltd. This system is based on the first experimental architecture for integrating multiple data feeds.

Chapter Four - Real Time System For Bus Tracking, studies the design and implementation of a 32 bit real time system, written in order to track buses and provide passenger information. This system was designed and written in conjunction with Hampshire County Council, and is currently in use and providing feedback. The architecture for this system builds upon the architecture presented in the previous chapter.

Chapter Five - Generic RTD System For Multiple Data Feeds, describes the final architecture presented in this thesis and examines the methodologies used for engineering the generic software system, along with an in-depth view of the design of the system. Including ways it could be made more future proof. The architecture for this system is the third presented in this thesis and builds upon those presented in the previous two chapters.

Chapter Six - Generic RTD System Implementation, describes in detail the ultimate target of this thesis. Covering the implementation phase of the project, examining the language choice and style as well as specific details of implementation. Also considered in this chapter are testing strategies.

Chapter Seven - Assessment, assesses the three different architectures presented in previous chapters. Pointing out what can be learned from each one and how these lessons have changed the way in which the next system was created.

Chapter Eight - Conclusions And Future Work, rounds up the thesis by making notes to the success of the various systems in the real world, and suggesting what work could be carried out in the future.

1.7. Research Contributions

This thesis provides an architecture that makes four contributions to science.

Uniform Data Collection Point. This provides an interface for filtering and sorting data, by providing an 'engine' to which any feed can be added using only a few simple lines of Visual Basic.

Standard Real Time Engine. The data that is collected can be real time. By providing a system for immediately notifying data users of changes to the data, the system can cope with data that is constantly changing. This real time engine is fully compliant with the Microsoft WOSA XRT specifications. The system also provides a mechanism for handling historical data.

Resolving Information Overload. To resolve information overload, the system provides a somewhat more complex interface so that programs can be written to examine the data and filter out the required information. Selecting only the information that is of interest to the end user can do this.

The 'intelligence' of the sorting and filtering software will dictate the level to which the information overload problem can be resolved. The simplest would be to allow the user to choose categories of information from which to collect data, if they choose an area then all the data would be collected. A more intelligent approach would be to provide an agent which actually searched the incoming data for topics of interest, or keywords [17]. Either way it is expected that the information overload problem will be lessened to some extent.

The Industry Aspect. A unique point of this thesis is that the experimental software, in order to be realistic, has addressed two points:

[1] Industry compliance; the architectures presented have all attempted to overcome the shortcomings of the Microsoft Windows operating system in order to provide systems which would gain commercial acceptance, and thus be able to prove the worth of the architecture in the real world.

[2] End user feedback; by providing architectures which could be deployed in the real world it was possible to obtain a level of end user feedback that is not usually available to research projects.

Chapter 2

REAL TIME PROCESSING OF DATA FEEDS

This chapter examines several commercial real time systems and identifies their key features relating to information overload. It studies the fundamental aspects of real time data processing with specific reference to the industry standard Microsoft Windows in the PC environment.

2.1. Introduction

To provide an understanding of the complex nature of real time data delivery, along with mechanisms for achieving the goals of this thesis, this chapter provides an insight into the methods employed by existing systems to provide large amounts of data to the user in a timely manner. The financial sector is the largest market for real time systems on PC's. These systems aim to provide end users with the price of shares as soon as they change in the markets, along with news and financial reports.

The second section of this chapter studies some of the various techniques which can be employed to create these types of systems, under what is essentially not a real time operating system, namely Microsoft Windows.

2.2. Existing systems

The financial markets provide large quantities of data that the end user requires in the shortest possible time frame. The data consists of anything that is quantifiable that occurs within the markets, such as prices of shares, exchange rates, times of trades and volumes of trades. Each item for which data is provided is called an instrument. This data is usually provided in the form of a 'feed', which is simply a stream of the data. Three systems are examined.

Reuters Traders Workstation

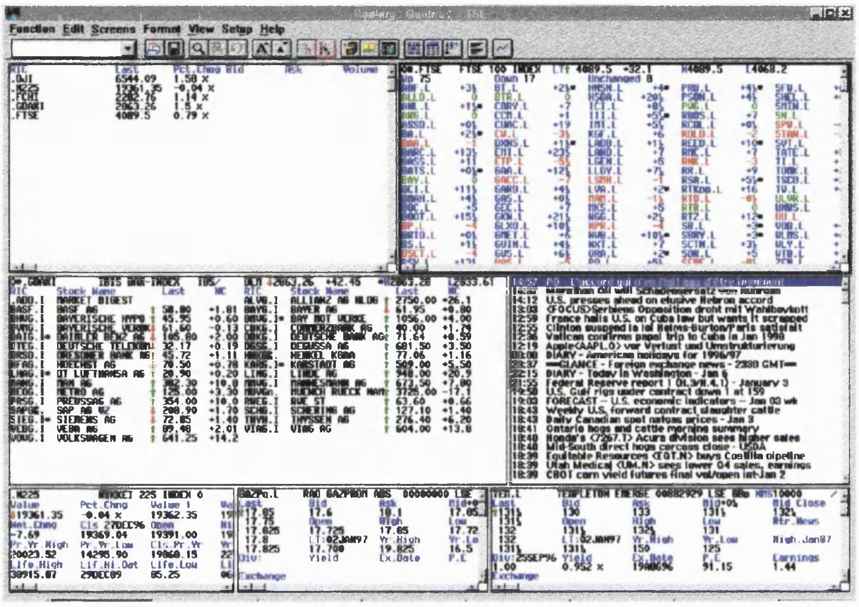
Reuters provide a range of real time financial systems for the PC market place, the one which is examined here is their Trader's Workstation. This links directly to a Reuters feed, through a card that plugs into the PC. Reuters aim to provide the information to the desktop within one second of a price change occurring on the trading room floor.

The software they provide has a Windows style interface, with some standard menus, right click popup menus, and child windows.

It has some useful features, such as the ability to build up a set of child windows, as shown in the screen shot in figure 2.1, and then save that layout as a workspace for future use. This helps solve the information overload problem by allowing the users to define what they view. By double clicking on an instrument a more detailed child window opens, with information pertaining specifically to that instrument. The user can also perform searches, change the formatting of the output along with standard Windows features, such as changing fonts. It is possible to draw charts on an instrument, by selecting to draw a chart, a separate application is launched, Reuters Graphics. Charts are a common method that it is used to lessen the information overload problem, by providing a view of the data that is easier for humans to understand.

This system also provides a means of getting data out of this application into other applications, it does this by means of a DDE (Dynamic Data Exchange) link. This provides some clues as the underlying technologies used to create this system.

The first thing to note is the fact that the system runs under Windows For Workgroups as well as Windows 95. This would indicate that it is a 16 bit system. While running the Reuters software there are always a couple of background tasks that are running. It would appear that these provide the link from the feed to the front end software. Through further examination, using software spies, it would seem likely that they use DDE as a communication protocol. The fact that it uses DDE (Dynamic Data Exchange) linking rather than COM (Component Object Model) also suggests use of old 16 bit technologies. While there are commercial considerations for Reuters to provide a 16 bit system, such as a large existing 16 bit user base, it does mean that they are not utilising the full potential power available to them. By using older technologies the Reuters system has the benefit of backward compatibility, but it does incur the penalty of not utilising the more advanced up to date technologies which would provide faster data handling and more open interfaces.


Function Edit Screens Format View Setup Help

```
RIC      Last      Pct.Chng Bid     Ask     Volume
.DJI     6544.09   1.58 x
.N225    19361.35  -0.04 x
.FCHI    2202.76   1.14 x
.GDAXI   2063.26   1.5 x
.FTSE    4089.5    0.79 x
```

```
0#.FTSE   FTSE 100 INDEX   LT↑ 4089.5  +32.1      H4089.5    L4068.2
Up 75             Down 17           Unchanged 8
ABF.L   +3½    BT.L    +2½    HMSN.L  +4     PRU.L   +4½    SFW.L
ALLD.L  0      BTR.L   0      HSBA.L  +20½   PSON.L  +4½    SHEL.L
ANL.L   +1½    CBRY.L  +7     ICI.L   +8½    PWG.L   0      SMIN.L
ARG.L   0      CCM.L   +1     III.L   +5½    RBOS.L  +7     SN.L
ASSD.L  +0½    CUAC.L  +19    IMT.L   +5½    RCOL.L  +0½    SPW.L
BA.L    +2½    CW.L    -3½    KGF.L   +6     ROLD.L  -2     STAN.L
BAA.L   -1     DXNS.L  +1½    LADB.L  +1½    REED.L  +10    SVT.L
BARC.L  +13½   EMI.L   +23½   LAND.L  +7     RMC.L   +7     TATE.L
BASS.L  +11    ETP.L   -5½    LGEN.L  +5     RNK.L   -3     TI.L
BATS.L  +0½    GAA.L   +12½   LLOY.L  +7½    RR.L    +9     TOMK.L
BAY.L   0      GACC.L  -7     LSMR.L  -1     RSA.L   +5½    TSCO.L
BCI.L   +11½   GARD.L  +4½    LVA.L   +2     RTKpp.L +16    TV.L
BMAH.L  +4½    GAS.L   +0½    MAM.L   -1½    RTO.L   -0½    ULVR.L
BOC.L   +5     GEC.L   +7     MKS.L   +5     RTR.L   0      UNWS.L
BOOT.L  +15½   GKN.L   +21½   NGG.L   +2½    RTZ.L   +12    UU.L
BP.L    -4     GLXO.L  +10½   NPR.L   -4     SB.L    +3     VOD.L
BRTO.L  +0½    GMET.L  +6     NWB.L   +10½   SBRY.L  +3     WLMS.L
BS.L    +1½    GUIN.L  +4½    NXT.L   +7     SCTN.L  +3½    WLY.L
BSCT.L  -4     GUS.L   +6½    ORA.L   +2     SDR.L   +5     WTB.L
```

```
0#.GDAXI      IBIS DAX-INDEX  IBS/     DEM ↓2063.26  +42.45   H2063.26   L2033.61
RIC      Stock Name            Last      NC      RIC      Stock Name            Last      NC
.ADD.I   MARKET DIGEST                           ALVG.I   ALLIANZ AG HLDG     ↑ 2750.00  +26.1
BASF.I   BASF AG             ↑ 58.80    +1.81    BAYG.I   BAYER AG            ↓ 61.95    +0.80
BHWG.I   BAYERISCHE HYPO     ↑ 45.95    +0.60    BMWG.I   BAY MOT WERKE       ↑ 1056.00  +4.00
BVMG.I   BAYERISCHE VEREINK  ↓ 61.60    -0.13    CBKG.I   COMMERZBANK AG      ↑ 40.00    +1.74
DAIG.I   DAIMLER BENZ AG     ↓ 105.80   +2.00    DBKG.I   DEUTSCHE BANK AG    ↑ 71.64    +0.59
DTEG.I   DEUTSCHE TELEKOM    ↓ 32.17    +0.19    DGSG.I   DEGUSSA AG          ↑ 681.50   +3.50
DRSD.I   DRESDNER BANK AG    ↑ 45.72    +1.11    HNKG.    HENKEL KGAA         ↑ 77.06    +1.16
HFAG.I   HOECHST AG            70.50    +0.78    KARG.I   KARSTADT AG         ↑ 509.00   +5.50
LHAG.I   DT LUFTHANSA AG     ↑ 20.90    +0.20    LING.I   LINDE AG            ↑ 948.00   +20.9
MANG.I   MAN AG              ↑ 382.30   +10.8    MMWG.I   MANNESMANN AG       ↑ 673.50   +7.80
MEOG.I   METRO AG            ↑ 125.00   +3.30    MUVGn.   MUENCH RUECK NAM    ↑ 3725.00  -17.1
PRSG.I   PREUSSAG AG         ↑ 354.00   +10.0    RWEG.I   RWE ST              ↑ 63.60    +0.66
SAPGE.   SAP AG VZ           ↓ 208.90   +1.70    SCHG.I   SCHERING AG         ↑ 127.10   +1.40
SIEG.I   SIEMENS AG          ↓ 72.05    +1.40    THYH.I   THYSSEN AG          ↑ 276.40   +6.20
VEBG.I   VEBA AG             ↑ 89.48    +2.01    VIAG.I   VIAG AG             ↑ 604.00   +13.8
VOWG.I   VOLKSWAGEN AG       ↑ 641.25   +14.2
```

```
      Marathon Oil will Schadensersatz von Rohrprom
14:12 U.S. presses ahead on elusive Hebron accord
13:03 <FOCUS>Serbiens Opposition droht mit Wahlboykott
12:59 France hails U.S. on Cuba law but wants it scrapped
12:55 Clinton suspend la loi Helms-Burton/Paris satisfait
12:36 Vatican confirms papal trip to Cuba in Jan 1998
12:19 Apple<AAPL.O> vor Verlust und Umstrukturierung
00:00 DIARY - American holidays for 1996/97
23:37 ==GLANCE - Foreign exchange news - 2330 GMT==
22:15 DIARY - Today in Washington - Jan 6
21:55 Federal Reserve report 1 (H.3/H.4.1) - January 3
19:50 U.S. Gulf rigs under contract down 1 at 159
19:03 FORECAST -- U.S. economic indicators -- Jan 03 wk
18:43 Weekly U.S. forward contract slaughter cattle
18:43 Daily Canadian spot natgas prices - Jan 3
18:41 Ontario hogs and cattle morning summary
18:40 Honda's <7267.T> Acura division sees higher sales
18:40 Mid-South direct hogs carcass close - USDA
18:39 Equitable Resources <EQT.N> buys Costilla pipeline
18:39 Utah Medical <UM.N> sees lower Q4 sales, earnings
18:39 CBOT corn yield futures final vol/open int-Jan 2
```

```
.N225          NIKKEI 225 INDEX 0
Value          Pct.Chng       Value 1
↓19361.35      -0.04 x        19362.35
Net.Chng       Cls 27DEC96    Open
-7.69          19369.04       19391.00
Pr.Yr.High     Pr.Yr.Low      Cls.Pr.Yr
20023.52       14295.90       19868.15
Life.High      Lif.Hi.Dat     Life.Low
38915.87       29DEC89        85.25
```

```
GAZPq.L     RAO GAZPROM ADS  00000000 LSE
Last        Bid          Ask          Mid+0
17.85       17.6         18.1         17.85
17.75       Open         High         Low
17.825      17.725       17.85        17.72
17.8        LT:02JAN97   Yr.High      Yr.Lo
17.825      17.700       19.825       16.5
Div:        Yield        Ex.Date      P.E
Exchange
```

```
TEM.L       TEMPLETON EMERGE 00882929 LSE GBp NMS10000
Last        Bid          Ask          Mid+0%       Mid Close
↑131½       130          133          131½         132½
131½        Open         High         Low          Rtr.News
132         131½         132½         131
132         LT:02JAN97   Yr.High      Yr.Low       High.Jan87
131½        131½         150          125
Div:25SEP96 Yield        Ex.Date      P.E          Earnings
1.00        0.952 x      19AUG96      91.15        1.44
Exchange
```

Figure 2-1 Screen shot of Reuters Trader's Workstation

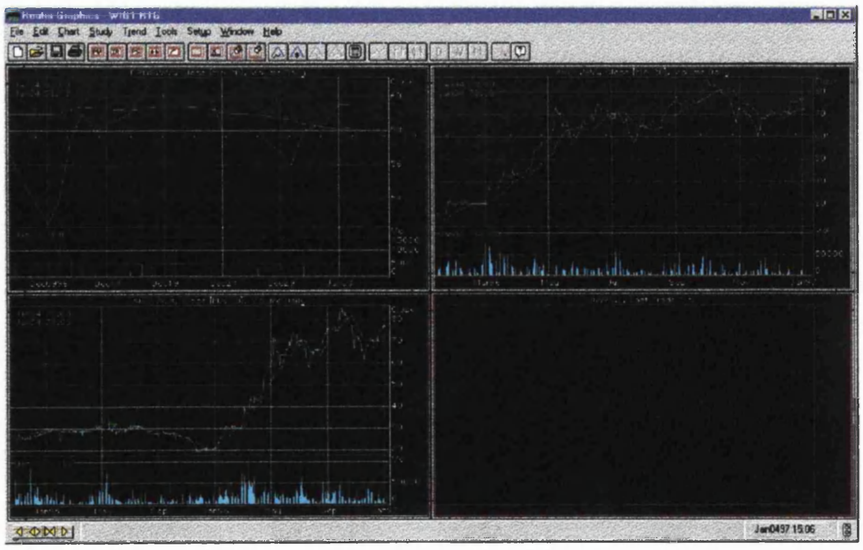
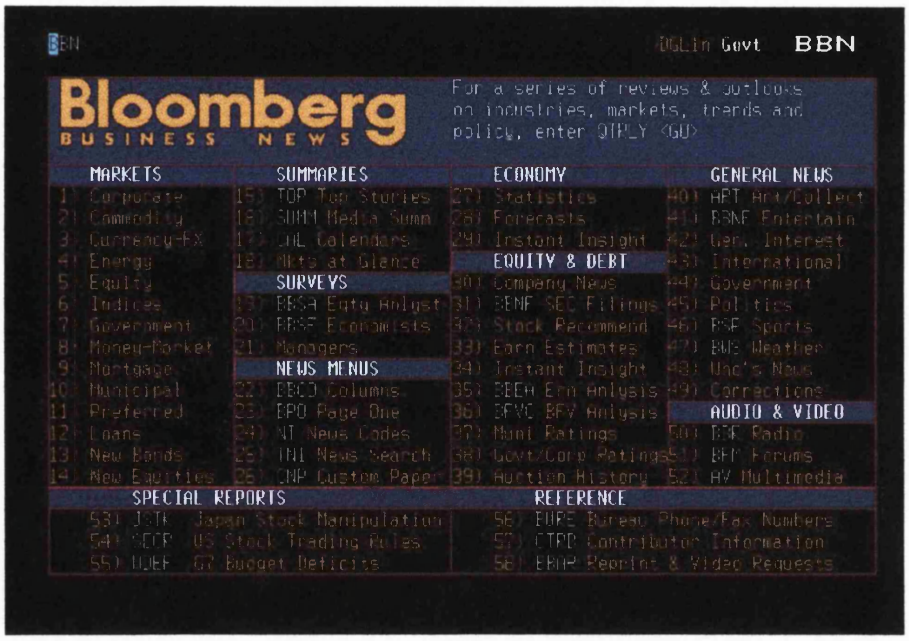


Figure 2-2 Screen shot of Reuters Graphics

Bloomberg

Bloomberg provide an integrated package of software along with their feed, which requires a card inside the PC. This provides on-line data along with radio and TV, so could be considered to provide multiple feeds. It provides two main screens for viewing the information, and for this reason two actual VDU's are usually used, splitting the desktop between them. This helps to reduce the amount of information displayed on each screen and therefore reduce the information overload.

This system does not have a Windows 'look and feel', it has no pull down menus or icons, in fact it looks very much like a text based terminal.



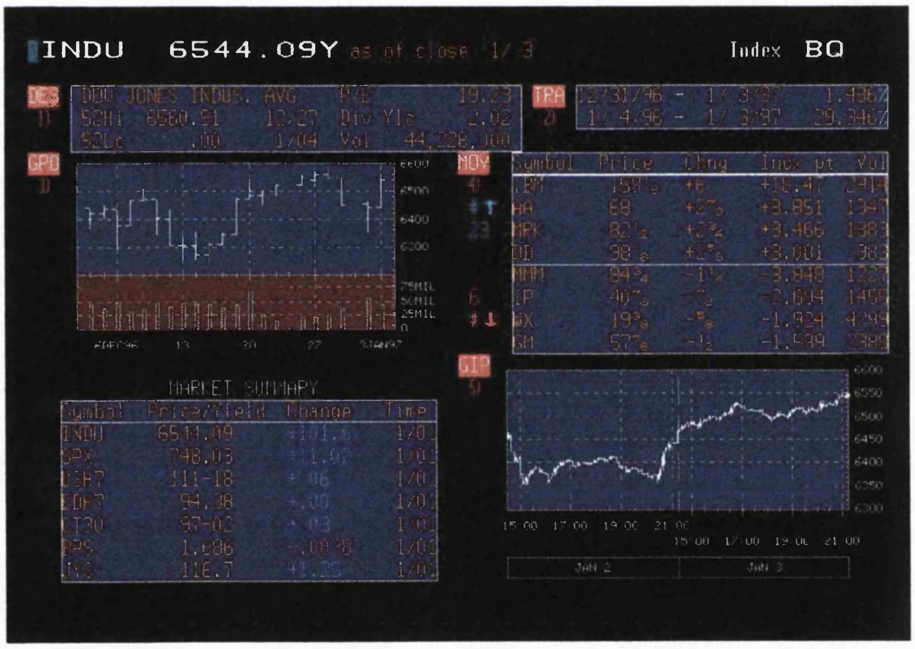
```
BBN                                                    DGLin Govt   BBN

Bloomberg                        For a series of reviews & outlooks
BUSINESS NEWS                    on industries, markets, trends and
                                 policy, enter QTRLY <GO>

   MARKETS         SUMMARIES              ECONOMY               GENERAL NEWS
 1) Corporate      15) TOP Top Stories    27) Statistics        40) ART Art/Collect
 2) Commodity      16) SUMM Media Summ    28) Forecasts         41) BBNF Entertain
 3) Currency-FX    17) CAL Calendars      29) Instant Insight   42) Gen. Interest
 4) Energy         18) Mkts at Glance         EQUITY & DEBT     43) International
 5) Equity             SURVEYS            30) Company News      44) Government
 6) Indices        19) BBSA Eqty Anlyst   31) BBNF SEC Filings  45) Politics
 7) Government     20) BBSF Economists    32) Stock Recommend   46) BSF Sports
 8) Money-Market   21) Managers           33) Earn Estimates    47) BWE Weather
 9) Mortgage           NEWS MENUS         34) Instant Insight   48) Who's News
10) Municipal      22) BBCO Columns       35) BBEA Ern Anlysis  49) Corrections
11) Preferred      23) BPO Page One       36) BFVC BFV Anlysis      AUDIO & VIDEO
12) Loans          24) NT News Codes      37) Muni Ratings      50) BBR Radio
13) New Bonds      25) TNI News Search    38) Govt/Corp Ratings 51) BFM Forums
14) New Equities   26) CNP Custom Paper   39) Auction History   52) AV Multimedia
         SPECIAL REPORTS                            REFERENCE
   53) JSTK  Japan Stock Manipulation     56) BURE Bureau Phone/Fax Numbers
   54) SECP  US Stock Trading Rules       57) CTRB Contributor Information
   55) UDEF  G7 Budget Deficits           58) ERRP Reprint & Video Requests
```

Figure 2-3 Screen shot of Bloomberg system

In order to navigate around the software it is necessary to learn a proprietary navigation system. It has a system of menus that are activated by certain keystrokes. Bloomberg do provide a proprietary keyboard which has labels on the keys for the various menus. In order to aid the user the system provides an integrated tutorial session which literally talks the user through the entire system. By following these tutorials it quickly becomes apparent that this system is huge and not only provides the user with information but also connects all the Bloomberg users together. So it is possible to send messages to other users of the same system. It contains areas where ideas can be tested and portfolios can be created.

The graphing it provides is integrated into the same system but is rather primitive. The user can draw charts on almost any instrument but the charts cannot be manipulated. There are sets of predetermined types of chart that can be viewed.

These charts can be customised to the extent that the user can choose to view more than one instrument on a chart but the kind of zooming features provided by the Reuters package are not available, although it is possible to view the chart full screen.



INDU 6544.09Y as of close 1/ 3 Index BQ

DOW JONES INDUS. AVG			P/E		19.23
52Hi	6560.91	12/27	Div Yld		2.02
52Lo	.00	1/04	Vol		44,228,000

12/31/96	-	1/ 3/97	1.486%
1/ 4/96	-	1/ 3/97	29.346%

Symbol	Price	Chng	Indx pt	Vol
IBM	158⅞	+6	+10.47	2914
AA	68	+2⅞	+3.851	1347
MRK	82½	+2¾	+3.466	1883
DD	98⅛	+2⅝	+3.081	983
MMM	84¾	-1¼	-3.848	1227
IP	40⅞	-⅞	-2.604	1455
WX	19⅜	-⅝	-1.924	4299
GM	57⅞	-½	-1.539	2388

MARKET SUMMARY

Symbol	Price/Yield	Change	Time
INDU	6544.09	+101.6	1/01
SPX	748.03	+11.07	1/01
USH7	111-18	+.06	1/01
EDH7	94.38	+.00	1/01
CT30	97-02	+.03	1/01
BPS	1.686	-.0038	1/01
JYS	116.7	+1.35	1/01

Figure 2-4 Screen shot of Bloomberg charts screen

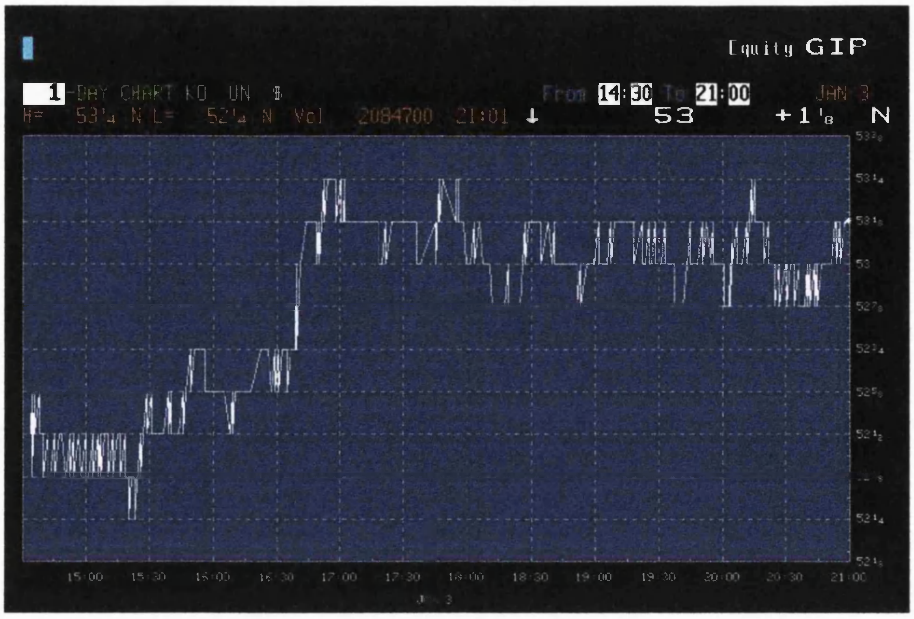


Figure 2-5 Screen shot of Bloomberg chart full screen

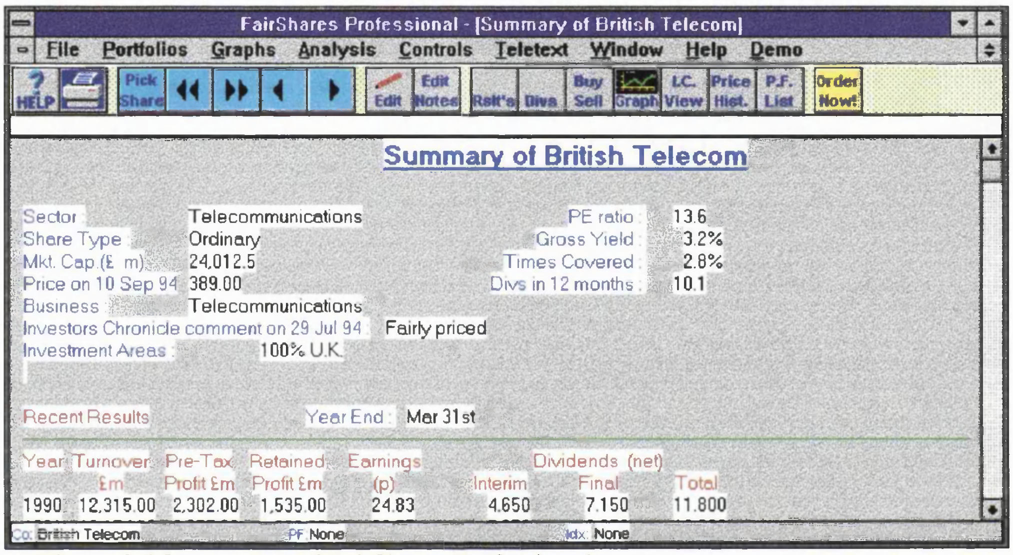
So what is the underlying technology behind this system? Well it would appear that it is simply a terminal based front end with a fast connection to a powerful real time system managed by Bloomberg. So really the system on the desktop isn't real time at all, it is on-line to a real time system which resides elsewhere.

FairShares

It worth taking a look at a far simpler package here, FairShares is certainly aimed at the lower end of the market than either the Reuters or Bloomberg systems. FairShares main strength is portfolio management. The professional product is a Windows rewrite of the original DOS product. It offers general coverage of technical analysis and a facility to record fundamental information on a company. On the portfolio management side the software covers just about everything a private investor could need including: CGT, Dividends, Scripts and rights. Reports can be produced for tax purposes and portfolio valuation and performance as well as compound growth rate.

While Fairshares is not a dedicated technical analysis package it does provide a small number of tools. Some are quite complex, but there is a lack of simple tools such as trend lines. The package has facilities to enter data on PE, Yields and Company Results. It is possible to plug various thrid party feeds into this system, e.g. Teletext or Market Eye, none of these provide the accuracy of either Reuters or Bloomberg but they are less than half the price.

The fact that it is a port of a DOS based system and that it supports DDE indicates that it is 16 bit, it does provide a much more Windows type look and feel than either of the two more expensive packages, unfortunately it also feels cheaper as can be seen from the screen shot shown in figure 2-6.


FairShares Professional - [Summary of British Telecom]
File Portfolios Graphs Analysis Controls Teletext Window Help Demo
HELP
Pick Share
Edit
Edit Notes
Rslt's
Divs
Buy Sell
Graph
I.C. View
Price Hist.
P.F. List
Order Now!

Summary of British Telecom

Sector	Telecommunications	PE ratio	13.6
Share Type	Ordinary	Gross Yield	3.2%
Mkt. Cap.(£ m)	24,012.5	Times Covered	2.8%
Price on 10 Sep 94	389.00	Divs in 12 months	10.1
Business	Telecommunications		

Investors Chronicle comment on 29 Jul 94 Fairly priced
Investment Areas : 100% U.K.

Recent Results Year End : Mar 31st

Year	Turnover £m	Pre-Tax Profit £m	Retained Profit £m	Earnings (p)	Dividends (net) Interim	Final	Total
1990	12,315.00	2,302.00	1,535.00	24.83	4.650	7.150	11.800

Co: British Telecom PF None Idx: None

Figure 2-6 Screen shot of FairShares Professional

2.3. Real Time Data Processing under Windows

This section examines the mechanisms that can be employed to achieve the goal of creating software that deals with data in 'real time'. This section clarifies what is meant by real time. It is important at this time to explain some of the fundamental aspects of why Windows is not a real time system, and how commercial programmers can get around this limitation. It is assumed that the reader has a working knowledge of abstract data types, and the Windows environment.

There are Operating Systems designed especially for real time applications (RTOS's) and Microsoft Windows is certainly not one of them [11]. The reason for studying ways of creating a system for Microsoft Windows is commercial acceptance. More and more companies are trying to use Windows as a standard environment at all levels of the industrial hierarchy. So it is becoming a requirement that real time data be handled within this environment. Another good reason for selecting Windows as an environment is the broad and powerful API (Application Programming Interface) which it provides. There are many good development platforms and compilers which support this API, and also many skilled programmers who know the API well. This leads to the fact there are many other applications already available for Windows, some of which may complement a real time system being written.

Definition of a Real Time System

Real time, as stated in the previous chapter, is any data that is only of use during a certain time frame. Data that can still be of use some time after this time frame has elapsed (e.g. for statistical analysis) is considered to be historical data.

One of the most comprehensive definitions was found in the real time FAQ on the Internet's comp.realtime news group: "A real time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation, but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."

To fulfil this definition, a couple of basic requirements must be met:

1. Meet deadlines. After an event has occurred an action must be taken within a predetermined time limit. Missing a deadline is considered a software fault.

2. Simultaneity or simultaneous processing. Even if more than one event happens simultaneously, all deadlines for all these events should be met.

The first requirement does not mean that it is a software fault if a text editor or spreadsheet application reacts slowly and thus slows down the user's progress. This would be considered as a performance problem that could be solved by using a faster processor. Using a faster processor may not necessarily relieve the problem of missing deadlines.

Simultaneity suggests that a real time system requires inherent parallelism in the form of either multiple processors and/or multi-tasking.

The data involved will generally come from an external source, the user will wish to view the changes in that data within a specified time frame. Each application will tend to be different, but generally real time data becomes historical data once that specified time frame has elapsed. The aim of the software is therefore to get the data to the user within that time frame.

An example of this is software which handles data from the financial markets; users of this data need to know the values of their shares before that value changes. Reuters claim to deliver the price to a user's screen within 1 second of it changing in the marketplace. So, for this type of data, the time frame is measured in milliseconds.

Hard and Soft Real Time Systems

Dr. M. Timmerman, in his paper 'Windows NT as Real Time OS?' [9], makes a distinction between hard and soft real time systems based on their properties.

The properties of a hard real-time system are:

1. No lateness is accepted under any circumstances

2. Useless results if late

3. Catastrophic failure if deadline missed

4. Cost of missing deadline is infinitely high

Whereas a soft real-time system is characterised by:

1. Rising cost for lateness of results

2. Acceptance of lower performance for lateness

A hard real time system must, without fail, respond to some kind of event within a specified time window. This response must be predictable and independent of other activities undertaken by the operating system. This implies that all system calls will have a specific measured latency period.

Latency refers to the time taken for the CPU to acknowledge and handle an interrupt. This generally involves three steps.

1. The CPU must finish processing the current instruction, flush the instruction pipeline, read the interrupt vector, locate the address of the handler, and jump to that address.

2. The handler records the current state of the computer, and creates a frame that records the state of the thread which was interrupted (this includes program counters and registers). The handler then starts an interrupt dispatcher which determines the source of the interrupt. This dispatcher then transfers control to either an interrupt service routine (ISR) or an internal kernel routine. The ISR would have been provided by a device driver for a particular device which caused the interrupt.

3. Finally the ISR starts an I/O transfer from the device, and can execute other threads when the transfer completes. When this is done the CPU is again interrupted for service.

Windows 3.x

Within the following section the word 'Windows' will apply to Windows 3.x and not Windows NT.

Microsoft produced a short white paper on real time systems under its Windows environment, a section of it is included below.

The information in this article applies to - Microsoft Windows Software Development Kit (SDK) for Windows versions 3.0 and 3.1
In no sense can Microsoft Windows be considered a "real-time" system. It is a message-driven, event-polling system, with nonpreemptive scheduling. The following is additional information:

1. It is possible to write a Windows-based application that sits on some interrupt in order to watch board-level activity. In general, this facility is used by manufacturers of boards to write drivers that watch and respond to interrupts used by the board. It is extremely dangerous to allocate/sit on any interrupt used by Windows itself (keyboard, mouse, etc.).
2. Dedicated systems (those that will not run general-purpose Windows applications) may sit on the timer interrupt, as long as Windows is eventually notified of the timer ticks. Because of the nonpreemptive aspect of the Windows system, it is insensitive to delays in the arrival of timer ticks. Trying to run more than one time-critical application that sits on the timer interrupt is likely to fail.
3. You must arrange for a Windows "library" of routines to provide access to information available from a board. The issue is one of how to divide the work between the following portions:
   a. The driver portion, which watches the interrupt, logs data, and notifies clients [through PostMessage()] of the availability of data
   b. The client portion, which obtains data from the driver portion, and processes, displays, and stores data

   Drivers can be implemented as Windows libraries, and clients of the device can be implemented as Windows-based applications. The driver portion can be made NEAR real time; the application portion is going to be message driven.

Copyright Microsoft Corporation 1995.

It is important at this stage to understand some fundamental aspects of how Windows works. Windows presents the illusion of multi-tasking, in fact it simply switches between applications while they are idle. So it would seem that the obvious solution to writing a system which involves constant processing would be to never allow the system to become idle. This is suggested in point 2 above, and this would work but no other applications would be able to run. It would only be usable in a dedicated system, this defeats the whole purpose of using Windows.

In the article produced by Microsoft point 1 states how a driver for a board can sit on an hardware interrupt, it then suggests that these interrupts be used to post messages to an application. This assumes that the software for the driver knows about the applications which are going to use the data it provides. This is not always the case, and some driver software gets around this by allowing applications to register themselves as users of the data. The driver software can then post the data to the registered applications. This solution is fine if the amount of data is limited and the application using the data wishes to know about all the data. It is very inefficient if the application using the data only wishes to process a very small amount of the data being collected by the driver.

For example a card which collects data from the stock exchange may be pulling in information about thousands of share prices. If the application being written is for drawing a graph for one or two share prices, it certainly doesn't want to be called every time any share price changes. One way in which this application could receive its data would be to ask the driver every so often for the latest change in the two prices in which it is interested; this is polling.

Polling Vs. Idle Loop Processing

Polling

Within the Windows environment polling is very easily achieved by setting up a timer and then requesting the data during the timer event. There are several ways of using timers:

1. A window handle is given when the timer is set up, this will place WM_TIMER messages into the given window's message queue. This is a fairly safe method.

2. A pointer to a call-back function is given when the timer is set up, the call-back function will then be called every time the timer goes off. This can be dangerous.

3. No handle or function pointer is given when the timer is set up, this will cause the WM_TIMER messages to be placed on the applications message queue. This is by far the safest way of using a timer.

The second method can lead to problems if the processing being done within the call-back function takes longer than the interval between timer calls. This will cause the function to be called again while the first instance is still working, this could lead to unpredictable behaviour.

The first and third methods are safer as the messages will be queued if the application is still busy processing the last timer message or dealing with user interaction. Specifying a specific window for a timer message could be useful if the application has multiple windows each of which are interested in different data, although it is important to remember that there is a limit to the number of timers available, within Windows. For this reason the safest method would be to create a single timer for the application. For more information see the Microsoft documentation on the SetTimer API [50].

Idle Loop Processing

When an application is not doing anything it enters a loop, the default behavior for which is to simply monitor for incoming messages then yield some time for other applications, this is known as the idle loop. It is possible to perform some other functionality within this loop, but it is important not to spend too long in here as it will cause other applications to slow down. This technique is useful if the application wishes to know about changes in a large amount of data, each time it enters the idle loop it could pop one change from a queue of the latest changes.

Summary

The section above considers Windows 3.x, it is possible that this environment could be used for a soft real time system, and the user should be aware that data is likely to be missed if it starts to come in at a rate faster than it can be handled.

Windows 3.x, even on the fastest available machines is useless for a hard real time system, as one application can keep the control forever and block the rest of the system, Windows 3.x is co-operative.

This hasn't stopped people from trying to create real time systems under Windows, and there are currently several packages available on the market which claim to be real time, and although they are obviously soft real time, they do still deliver data within some form of constrained time limits.

Windows NT

Being a full operating system rather than just a GUI environment Windows NT shows rather more promise as a platform for real time systems. Once again it is worth looking at what Microsoft have to say about using NT for real time systems. This time they provide much more information, hinting that they envisage NT as a more suitable platform. Although the point that NT is not a real time OS (RTOS) is made very clear [33]:

> *'..Microsoft® Windows NT™ Workstation is not a hard real-time operating system. Rather, it is a general-purpose operating system that has the capability to provide very fast response times, but is not as deterministic as a hard real-time system.'*

In order to judge just how good Windows NT is at providing a platform for real time systems we need to take a look at some of the facilities it provides. As some of the developers of the VMS operating system helped to create Windows NT, some real time characteristics have been introduced into the system. An example is the real time class processes that are scheduled in the same way as they would be in a RTOS.

Windows NT is a multi-threaded and pre-emptive OS, unlike Windows 3.x which requires co-operation from the applications, NT has a scheduler. This scheduler can pre-empt any thread in the system, in order to give resource to a thread which needs it more. The OS also enables pre-emption at the interrupt level by allowing multiple levels of interrupts.

In order to find which thread needs a resource the most, the OS needs to know when a thread has to finish its job and how much time it needs in order to do so. At the moment no OS can do this at it is simply far too complex to implement. The solution is to give each thread a priority, this is the job of the system designer, who has to convert deadline requirements into thread priorities. Windows NT provides 32 priority levels, of which 16 are reserved for the operating system and real time processes. Each process has a base priority class, the priority spectrum diagram in figure 2-7 shows which priorities are used for different types of process.

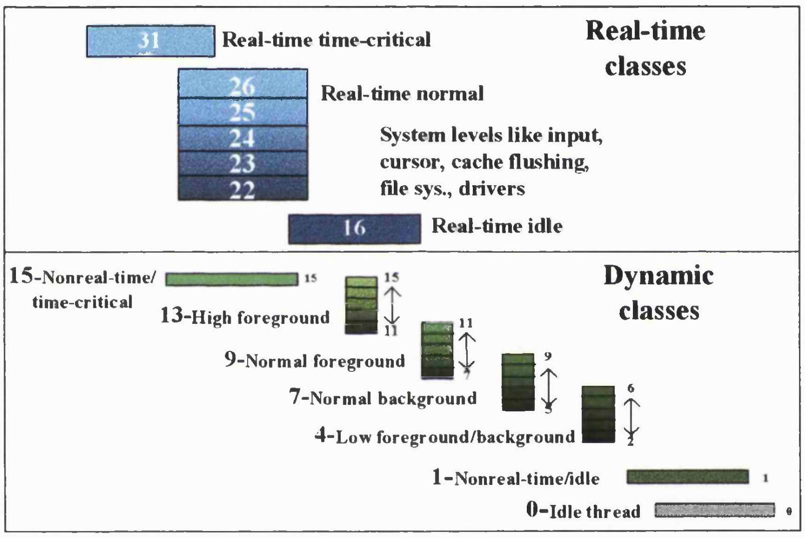


Figure 2-7 Priority Spectrum For Windows NT[11]

Each process will have one or more threads, and each thread will inherit its priority from the associated process. Using an API each thread can have its priority varied by ±2, so for a given process there can only be five different thread priorities. For example the threads of a process with base priority 24 can only have priorities in the range of 22 - 26. Many RTOS's have 256 levels of priority, this allows the system to be designed for more predictable outcomes, and the best way to design a system is to give each thread a different priority.

One way around this is to use different processes and pass information between them, this increases the number of priorities to 16, but does bring up another problem which is that of the time taken to switch between the processes. This context switch time is higher than the time to switch between threads in the same process. This is because threads in the same process share their memory address space, unlike separate processes, which each need an individual address space in order to avoid any interference with other processes.

In fact it is this which distinguishes a thread from a process, the synchronisation and inter thread communication mechanism which exist within a processes address space but not across process boundaries. Older versions of UNIX are multi-tasking but not multi-threaded, each task is a process that can only communicate via pipes and shared memory, both of which use the file system, which cannot provide predictable behaviour.

One problem, which often gets uncovered in real time systems is priority inversion. For this condition to occur at least three threads of different priority are required to be involved. When the lowest priority thread has locked a resource which is shared with the highest priority thread, and the middle priority thread is running. This leads to a situation where the highest priority thread is suspended until the resource is unlocked, and the resource will only be unlocked by the lowest priority thread once the middle priority thread has finished running. Hence the highest priority thread is waiting on the lowest priority thread - priority inversion. This can be avoided by the OS allowing priority inheritance to boost the lowest priority thread above the middle one, a blocking thread inherits the priority of the thread it has blocked if it is higher than its own priority. Windows NT does not do this and so extra care needs to be taken in order to avoid such a situation.

Another problem related to priority inversion, occurs due to the way in which some API calls are implemented in a synchronous manner. They block the calling thread until the API call has completed. This implies that a lower level real time class thread could prevent the upper ones from running.

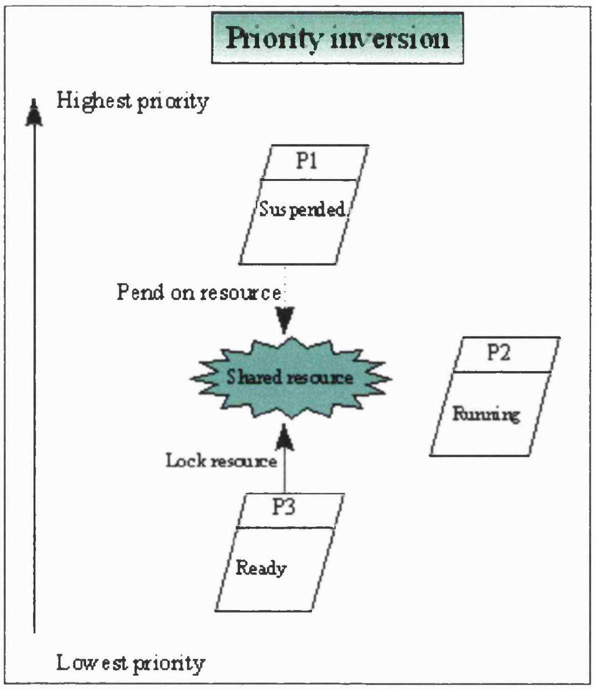


Figure 2-8 Priority Inversion

Memory Management

One of the features of NT is its virtual memory system, this is not necessarily such a problem for real time systems as it might first appear. The paging I/O (disk swapping) occurs at a lower priority than real time processes, paging can still occur within a real time process, but this is meant to ensure that background memory management doesn't interfere with any processing at real time priorities. Another point is that Windows NT allows an application to lock itself into physical memory (as long the machine actually has enough). This will ensure that the application is not affected by paging within its own memory address space. The last thing that Windows NT provides is memory mapping, which allows multiple processes to share the same physical memory. This permits very fast data transfers between co-operating processes.

One of the problems with using a general purpose OS such as Windows NT can be seen in the memory caching technique which is used to increase the overall system performance. This method uses a small amount of high speed physical memory to hold the most recently used code or data, if the next piece of data required is not in this cache memory then it must be retrieved from the slower main memory. This can lead to unpredictable behaviour as far as accurate timing of latency periods.

Conclusion

Windows NT is not written specifically for use as a real time operating system, but it does provide some features that allow for fast response times. It is therefore conceivable that a reasonable performance could be expected from a not too complex real time system written to run on Windows NT.

2.4. Industrial Strength Software Engineering

The software created in this thesis is far larger and more robust than the demonstration software produced by most research projects. This is due to the fact that it needed to be tested in a commercial environment. In order to create software that will stand up to use within this environment it needs to be of industrial strength.

In order to create industrial strength software a strict approach to the design and implementation needs to be taken along with a broader view of the requirements for the system being created. This section outlines the steps that were taken to create each of the three systems in this thesis.

All of the systems created for this thesis utilised a loose form of the waterfall model. A strict adherence to the waterfall model would ensure that any previous steps could not be addressed after the next step had been taken. This is not usually a realistic constraint to put on a commercial system, where the market place is constantly changing [40].

The Waterfall model

This model views the system being designed as a whole from the highest level and breaking the project down a stage at a time, as shown in Figure 2-9. Each stage is explained in turn.

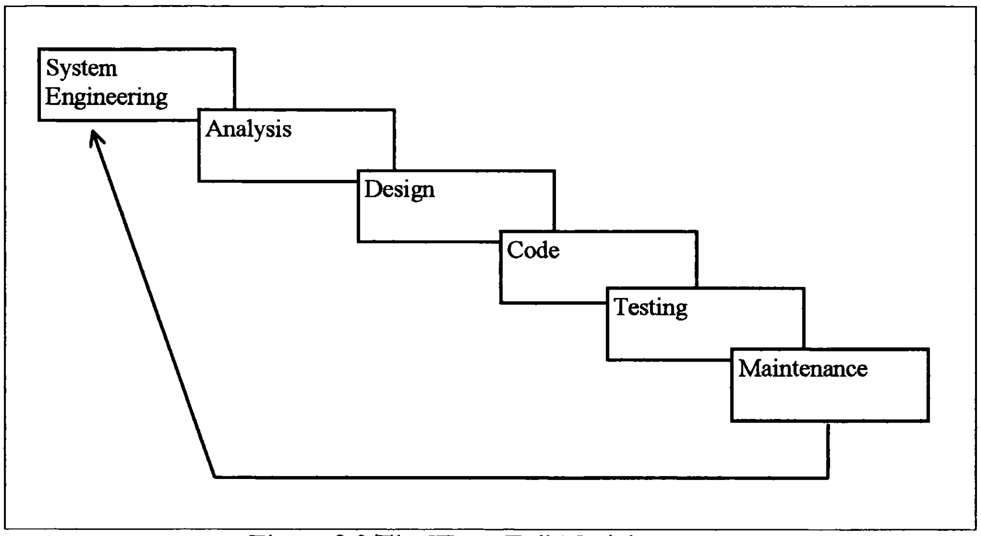


Figure 2-9 The Water Fall Model

System Engineering. This stage involves establishing the requirements for all system elements including those that are not software. Then the relationship of the software to the rest of the system should be considered along with a small amount of top level design and analysis.

Analysis. During this stage the information domain of the system being created should be understood along with the required functions, performance and interfacing. These should all be documented and reviewed with the management or client.

Design. This stage can be broken down into a multi-step process, focusing on the different attributes of the system. These are the data structures, the software architecture, procedural detail and interfaces. Again this stage should be fully documented, as it will become part of the software configuration.

Coding. If the design has been done properly the coding should become as simple mechanistic task of translating the design into a machine-readable form.

Testing. Once the code has started being generated testing can begin. This testing should initially focus on the logical internals of the software, making sure that all the statements have been tested. Then testing should move onto the functional externals, ensuring that defined inputs will produced the required results.

Maintenance. All systems require modifications after they have been finished, either because bugs need to be fixed or extra functionality is required. Maintenance applies each of the preceding steps to an existing system rather than a new one.

These steps are what Booch[48] terms the Macro processes. He also specifies a set of Micro processes, which are also used within the software developed for this thesis, they are:

1. Identify the classes and objects

2. Identify the semantics of these classes and objects.

3. Identify the relationships between these classes and objects.

4. Specify the interfaces and implementation of these classes and objects.

Evolutionary Approach

While each system was developed using a loose form of the waterfall model an evolutionary approach has been used throughout this thesis. Each system designed has built upon the previous system in order to obtain the final architecture.

Summary

By adhering to a known commercial design methodology it should be possible to create software that was not merely a proof of concept but could be readily tested within a commercial environment.

Chapter 3

REAL TIME SYSTEM FOR PRIVATE INVESTMENT

This chapter examines the first experimental architecture created for this thesis. The architecture was used for the design and implementation of a 16-bit real time system for monitoring share prices for private investors. This work was done in collaboration with Updata Software Ltd. who markets the system.

3.1. Introduction

The aim of this experiment was to create an architecture where a single collection point could be used for multiple sources of data. This would help ease the information overload problem that exists for users of data from multiple sources. The system built around this architecture was designed to deal with large volumes of stock market data in real time. Various vendors supply data and so the system had to be flexible enough to cope with as yet unknown data feeds. This real time data then had to be made accessible to a variable number of applications, which would want to perform different tasks on either all the data or a subset of the data.

3.2. System Requirements

Initial Requirements

The first stage in any design process is to gather all the known tasks required of the system, and then clarify any unclear issues. The initial system requirements were identified are as follows:

1. Input stock market data from one or more sources.
2. Process that data within a specified short time frame.
    - 2.1. Store every item of data input.
    - 2.2. Sort data into groups.
    - 2.3. Archive the data into history files.
3. Provide access to the data from external applications.
4. Run within the Microsoft Windows 3.x environment.

The first requirement indicated that it would be necessary to create an interface for applications to pass data into the system. In order to create this interface it would be necessary to identify exactly what data would be provided, 'stock market data' does not provide enough information. Actual fields and value ranges needed to be specified.

The second requirement had been broken down into the identified tasks of the data processing. The fact that this stage must occur within a specified time frame was a clear indication that this is some form of real time system. In order to decide whether it was a 'hard' or 'soft' real time system, further questions needed to be asked. The groups mentioned above in requirement 2.2 also needed more explanation.

The third requirement would mean that an interface would have to be created to allow other applications access to the data. The data required by other applications needed to be fully specified in order for this interface to be designed properly.

The final requirement was purely commercial and meant that any issues surrounding development within the Windows 3.x platform would be addressed during the design and implementation stages of this system.

Refined Requirements

The first stage of identifying the requirements threw up the following questions:

1. Exactly what data would be passed into the system?
2. What should happen if the data is not processed within the specified time frame?
3. What is meant by 'groups', when sorting the data?
4. Exactly what data is required by the external applications?

The answers to these questions are given below and they did much to help understand what was required of the system.

1. Exactly what data would be passed into the system?

As already stated the data passed into the system would be stock market data. What was needed was an exact break down of the fields that went up to make an item of stock market data. The following table shows the fields that would be input into this system.

Field	Description	Source
Name	The name of the item	Teletext, ICV, Reuters, User
Last	Last trading day's price	Teletext, ICV, Reuters
Low	Low for the day	Teletext, ICV, Reuters
High	High for the day	Teletext, ICV, Reuters
Current	Latest price	ICV, Reuters
Open	Open price for the day	ICV, Reuters
Bid	Best bid price	ICV, Reuters
Ask	Best ask price	ICV, Reuters
Alpha	Alpha volatility	Teletext, ICV, Reuters
Beta	Beta volatility	Teletext, ICV, Reuters
Stop Loss	Stop loss value	User
Volume	The volume traded	Teletext, ICV, Reuters
Date	Date of current price	Teletext, ICV, Reuters
Last Date	Date of last price	Teletext, ICV, Reuters
Time	Time of current price	ICV, Reuters

Table 3-1 Fields collected by the system

2. What should happen if the data is not processed within the specified time frame?

Asking this question would identify whether this was to be a 'hard' or 'soft' real time system. A 'hard' real time system would have been considered to fail if the processing did not occur within the given time frame. Whereas a 'soft' real time system would mean that some form of penalty would be incurred if the data processing failed to be performed within the time constraints.

A requirement was to produce a system which would deliver the data to the end user as fast as possible. If the data could not be processed within the time frame, it would mean that end users would be seeing data later than required. This was not considered a software failure, and as long as no data was lost, and would be acceptable during extremely busy times within the financial markets.

3. What do 'groups' mean, when sorting the data?

Previous software produced by Updata Software had grouped the stock market data items into 'folders'. These 'folders' were merely a logical way of grouping together related items. For example an Equities 'folder' would group all the data items from the Equities market. It was required that the new system behaved in a similar manner, by grouping related items.

4. Exactly what data is required by the external applications?

The answer to this question would help to identify the requirements for the interface to provide data to external applications. The external applications would need to have access to all the data in table 3.2.

Data required by external application	Description
The 'folders' by which stock market items are grouped.	The groups of data which the system provides. Identified by a unique name
Each stock market item in a folder	The instruments that are stored in each group. Each item can be identified by more than one name.
The value in each field from a stock market item, including which 'folder' it belongs to.	The most recent for all fields except the current price, where every change for the day may be required.
A list of the historical values of fields in a stock market item	The values for previous days trading at the end of the day.
The source of the data	The vendor from where the data was acquired.

Table 3-2 Data required by external applications

System Specification

From the initial requirements and the answers to the questions given in the previous section, a system specification could be drawn up.

This system will perform the following tasks:

1. Provide an interface to allow the following data to be fed into the system: A Name and one or more of the following fields: Last, Low, High,

Current, Open, Bid, Ask, Alpha, Beta, Stop Loss, Volume, Date, Last Date, Time.

2. Attempt to process any data within a 1 second of it being input into the system. If the processing has not finished within this time frame it will not cause any other data which is input into the system to be lost.

3. Each item will be sorted into a 'folder' or group of related items.

4. The system will be possible to archive each item into a historical data file.

5. The system will provide an interface so that other applications can access the data. The interface will allow the applications access to a list of 'folders' each item within a folder and the value of each field within an item. There will be functions for:
   a. searching for a specific folder by name,
   b. searching for a specific item by one of its six names,
   c. reading the values from any specified field of an item
   d. archiving an item to its historical file,
   e. reading the values of specified fields in a named historical file.

3.3. Overview Of The System Design

Traditionally companies that had delivered products targeted at the users of financial information produced 'all in one executables'. They did not break the processes down into a modules which could be plugged together to produce a final product.

An idea for a modular based product range was summarised with the diagram shown in figure 3-1.

The flow of data begins from the bottom of the diagram at the data feed, then into the data repository and finally into the end-user applications.

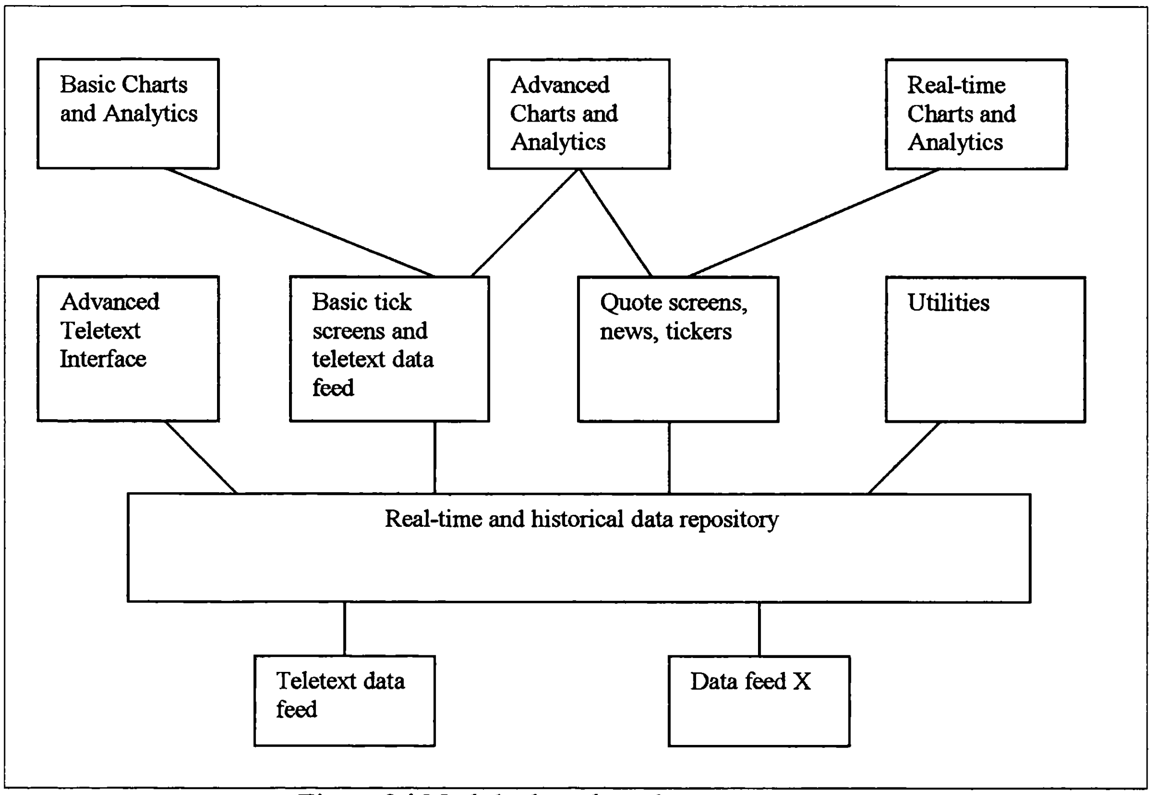

Basic Charts and Analytics
Advanced Charts and Analytics
Real-time Charts and Analytics
Advanced Teletext Interface
Basic tick screens and teletext data feed
Quote screens, news, tickers
Utilities
Real-time and historical data repository
Teletext data feed
Data feed X

Figure 3-1 Modular based product range

A first step towards modularity was designed. The general data flow model for this first version is shown in figure 3-2.

In both figures 3-1 and 3-2 the key component is the real time and historical data repository. By utilising the fact that under 16 bit Windows, DLL's (Dynamic Link Libraries) only have one instance of their data which can be accessed by more than one application a more detailed design was drawn up, as shown in figure 3-3. In this diagram the real time and historical data repository has been broken down into the two DLL's on the left.

The two key components in the diagram are the Live Data DLL and the Historical Data DLL. These DLL modules would provide an API that would be available to any programmers working on other parts of the system

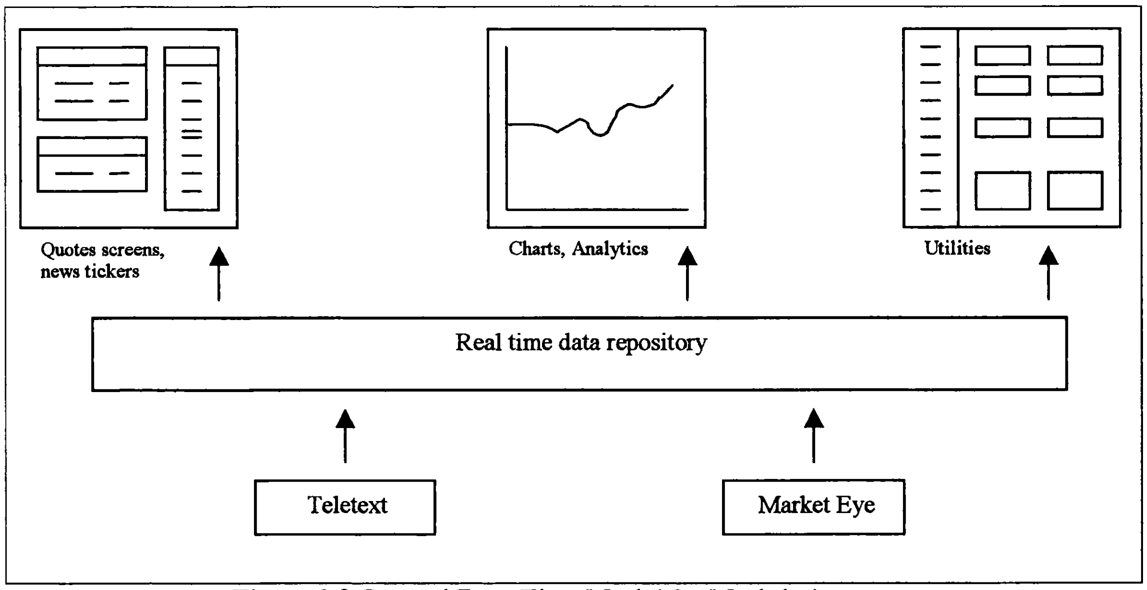


Figure 3-2 General Data Flow Model for Modularity

There were four good reasons for breaking the system down into the modules shown in the figure 3-3.

1. It created a more generic system, which allows for different viewer to be 'plugged' into the system at a later date.
2. New feeds when they became available could be easily added into the system.
3. Development could be carried out on the modules individually without having to worry about the rest of the system.
4. If at a later stage the functionality of a key component needed to be changed, only one module would need to be changed, thus minimising any side effects.

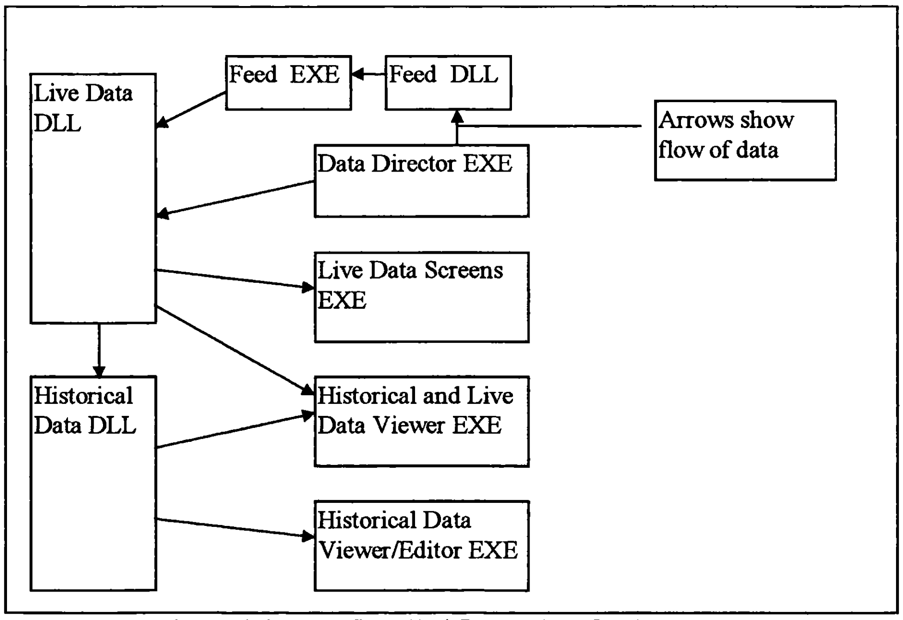


Figure 3-3 More Detailed Data Flow Design

From this stage an object-oriented approach was taken to the design of each of the modules, starting with the two key components, the Live Data and Historical Data DLL's.

3.4. Detailed Design

The software engine to cope with the data is in two 16 bit DLLs which export 16 bit functions for handling live data (tempdata) and historical data (Pricefiles). They both call functions exported from each other.

Design of the Live Data DLL

The Live Data DLL deals with the storage and retrieval of data that is relevant within one day of trading. It would therefore have to deal with not just the current price but also previous prices for that day. A basic design of the objects was drawn up as shown in figure 3-4.

As the data is collected during the day it will be stored in memory in order to enable fast access. Each item of data is categorised and stored in a data object of that type. The objects are logically grouped into folders and these folders are managed by the DLL.

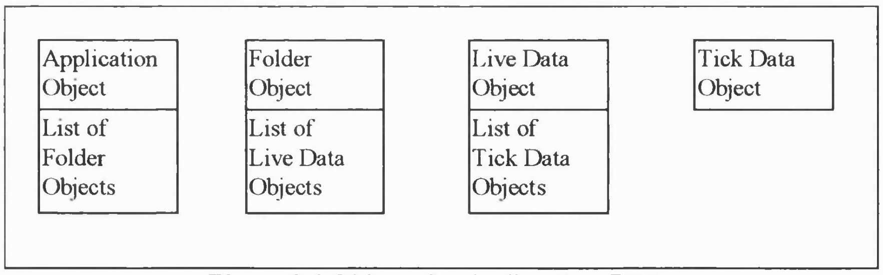


Figure 3-4 Objects for the live data DLL

The Application Object This deals with the functionality that is required for the program to run as a DLL. It also provides the API for the external applications. It contains a list of folder objects.

The Folder Object This provides a logical way of grouping together items of data that are of the same type. For example a folder could be used to represent all the Equities while another could be used to group together Gilts. It contains a list of Live Data Objects. It provides a means by which all the items in the list of Data Items can be iterated through.

The Live Data Object This represents one day of values for a single instrument or item. It contains a set of fields for that day of trading, and a list of tick data objects. The tick data object list is used to keep track of all the current prices for that item for that day. The fields required are shown in table 3.3.

Field	**Description**
a tick list	A list of the days ticks
Name	The name of the item
Synonyms 1-6	Other names for the item
Folder Name	The name of the folder in which the item is stored
Source	The source of the data i.e. what's the feed
Last	Last trading day's price
Low	Low for the day
High	High for the day
Current	latest price
Open	open price for the day
Bid	best bid price
Ask	best ask price
Alpha	alpha volatility
Beta	beta volatility
Stop Loss	stop loss value

Volume	the volume traded
Date	date of current price
Last Date	date of last price
Time	time of current price

Table 3-3 Fields Required for Live Data Object

The Tick Data Object This represents a current price at a particular time. It contains a price and a time

Field	Description
Price	a price value
Time	the time that the item became the price value

Table 3-4 Fields Required for Tick data Object

The Interface

The Live Data Objects needed to be accessible for integration by existing 16 bit applications, these were written in C and use a linked list of structures. So it was necessary to export the data into such structures, it was also a requirement to be able to import one of these structures and then place the data into a Live Data Object. The list of the interfaces designed can be found in Appendix 1.

Design Of The Historical Data DLL

This DLL handles the storage and retrieval of data from previous days of trading, this data is stored in a 'Pricefile'. A Pricefile is the historical data for one instrument on the market. It contains a collection of Live Data Objects, going back through a period of time, with one Live Data Object for each day. The Pricefile also stores information about that specific file, for example its name, folder name, the date of the first record in the file, and the date of the last record in the file.

The objects required were decided as shown in figure 3-5.

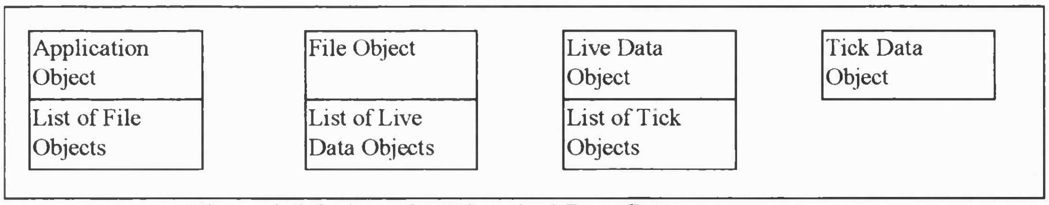


Figure 3-5 Objects for Historical Data DLL

The Application Object deals with the functionality that is required for the program to run as a DLL. It also provides the API for the external applications. It contains a list of file objects.

As can be seen from this description it is very similar to the Application Object in the Live Data DLL, and therefore should share some code.

The File Object provides a logical way of grouping together all the previous days Live Data for a particular instrument. It represents a 'Pricefile'. It contains a list of Live Data Objects.

It provides a means by which all the items in the list of Data Items can be iterated through.

Again this is similar to a Folder Object in the live data DLL and some base code functionality should be shared.

The each Pricefile needs to be able to store values for the following entries:

Entry	Description
a Live Data list	a list of Live Data Objects
Name	the filename of the item
Synonyms 1-6	other names for the item
Folder Name	the name of the folder in which the Live Data Object is stored
Start Date	the date of the first Live Data Object in the list
End Date	the date of the last Live Data Object in the list
Display Flags	a set of flags indicating the format to display the data

Table 3-5 Price file entries

The Live Data Object represents all the data for one day of trading for a single item or instrument.

This contains a list of tick data objects and also a set of fields for that day of trading. The tick data object list is used to keep a track of all the current prices for that item for that day.

This is identical to the Live Data Object in the Live Data DLL and so the entire object can be reused

The Tick Data Object Represents the current price at a particular time. It contains a price and a time

This is identical to the Tick Data Object in the Live Data DLL and so the entire object can be reused

The Interface

The 'Pricefiles' need to be accessible for integration by existing 16 bit applications, these are written in C and currently use a header structure and a linked list of structures. So it was necessary to export the 'Pricefile' into such structures, it was also a requirement to be able to import one of these structures and then place the data into a 'Pricefile' object.

A set of utilities for manipulating the 'Pricefiles' also needs to be provided, 'Pricefiles' need to be renamed, have items added and removed from them, and the data needs to be validated.

The full interface as it was defined can be found in Appendix 2.

3.5. Overview Of The System Implementation

Choice of Implementation Language

It was decided at an early stage that one of the main considerations in this system would be the speed of the processes. This meant that we would have to choose a compiled language rather than one that is interpreted at runtime.

Much of the existing code base from previous projects undertaken by the same company was in straight C, and so they had an existing core of programmers who knew C.

The system would have to run within the Windows 3.x environment, and so a compiler which supported all the Windows 3.x API's would be needed.

The design had been drawn up using some object-oriented (OO) methodologies. So a language which supported OO would make the translation of design to implementation far simpler.

All these points led to the decision to implement this system using C++. Using C++ would also allow a pre written class library to be used. This would increase productivity as much of the underlying Windows code was already written. The chosen class library, was the Microsoft Foundation Classes, as this wrapped the Windows SDK more fully than any other library.

It was also decided to implement the interfaces for external applications in straight C. This would mean that it would be possible for other developers to integrate existing applications into the system, using C, C++, Visual Basic or any other language which could import C function (this covers most MS Windows programming languages).

3.6. Detailed Implementation

The Objects

The main objects being used are listed below:

- Tick Data - represents a single price for a particular time
- Live Data - represents one days data for a particular subject, e.g. FTSE
- Pricefile - represents an historical list of days data

Live Data Objects

The Live Data objects contain the fields needed as mentioned in the design above, along with full access member functions. The class description for these objects is given in Appendix 3.

The Price File objects

The price file objects contain the fields needed as mentioned in the design above, along with full access member functions. The class description for these objects is given in Appendix 4.

The Relationships

Relationships in the Live Data DLL

The data in this library can be seen to be stored in a tree structure, or directory type structure, which is one deep. All the data items (Live Data Objects) are stored in folders, there are no folders in folders, and there are no data items in the root. The diagram in figure 3-6 shows the cardinality of the data that is stored.

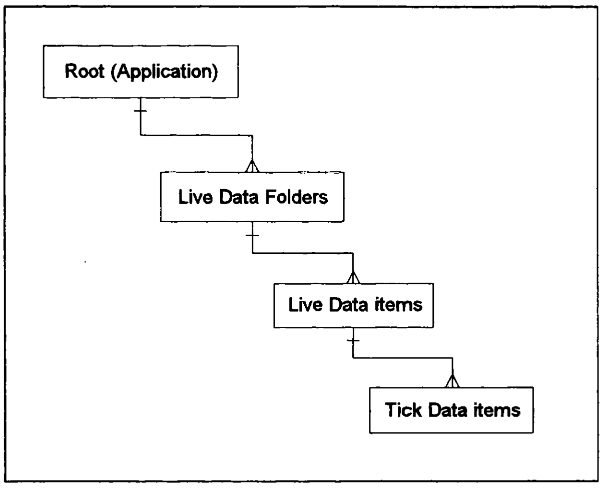


Figure 3-6 Cardinality of Data Relationships in Live Data DLL

In order to access a data item its folder must be opened, and then the item for that folder can be retrieved. The class hierarchy for the items in the Live Data DLL can be seen in figure 3-7. The objects prefixed with a C are from the Microsoft Foundation Classes and the objects prefixed UD are the specific objects created for the Live Data library.

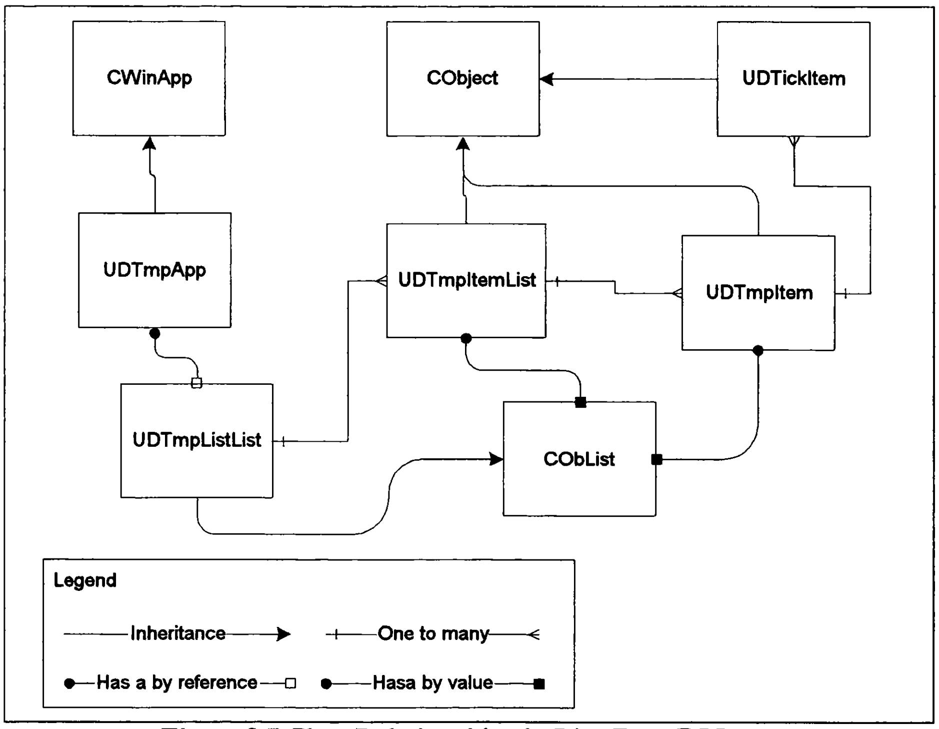


Figure 3-7 Class Relationships in Live Data DLL

Relationships in the Pricefile DLL

Figure 3-8 shows the relationships between the objects in the price file. It shows that each price file will contain one or more items of Live Data and that each item of Live Data will contain one or more tick item.

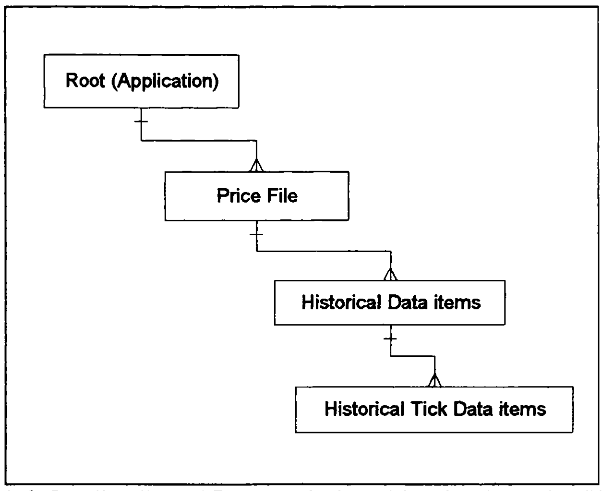


Figure 3-8 Cardinality of Data Relationships in the Pricefile DLL

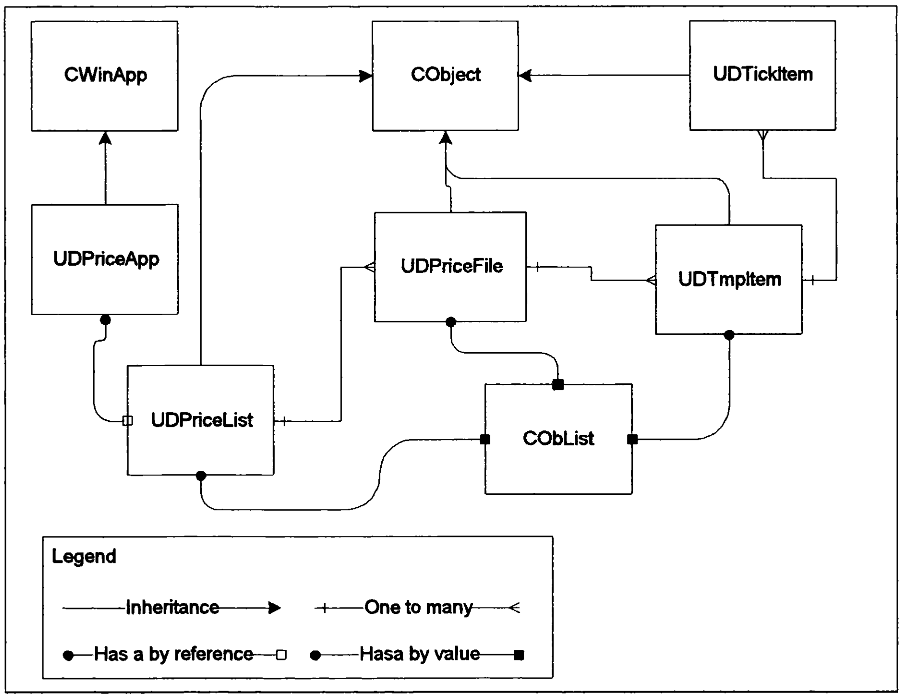


Figure 3-9 Class Relationships in the Pricefile DLL

The Pricefile DLL contains a list of the Pricefiles that it has open. The class hierarchy for the items in the Pricefile DLL can be seen in figure 3-9. The objects prefixed with a C are from the Microsoft Foundation Classes and the objects prefixed UD are the specific objects created for the pricefile library.

3.7. The Events

At run-time the data will be changing, the applications that use the data will need to be notified of these changes as and when they occur. This can be resolved by several means all of the following have been tried and evaluated:

- Callback functions: The applications using the DLL register a callback function with the DLL. When a new item of data comes in the DLL makes a call to that callback function to notify the application that there is a new data item available.
- Idle loop notification retrieval: during the idle loop of the application using the DLL, the application calls the DLL and asks if there has been an update since it last asked. If there has then the application queries the DLL for the latest changes.
- DDE: Dynamic Data Exchange, the DLL registers itself as a DDE server and the Applications register themselves as DDE

clients. The clients then request that the server notifies them of any changes, The server then notifies the client when it gains any new data.

The problem with all the above methods is the shear volume of traffic causes an enormous amount of messaging to occur between the applications and the DLL. This is especially true for DDE where for each notification several messages must be sent. The server tells the clients a new item is available, the client then asks the server for the new item, the server gives the new item to the client, the client then notifies the server that it has received the new item. What is needed is some form of refinement so as not to waste time sending messages, especially when the Application may not even want that specific item of data.

The final solution was a combination of the first two methods. The application registers a callback function with the DLL. When the DLL gets new data, it adds the data to a list for each client application then notifies the client of the data through the callback. The application then builds up a list of notifications, and in the idle loop starts removing items from the head of the list. If the application requires data for an item it requests it from the DLL.

All the classes require external access to some of their member's variables. Internally within each DLL this is easily solved by providing public get and set methods in classes that are required to provide external access to their members. As far as external applications are concerned all member variables will have to be accessed through the API of external calls which are presented by the DLL.

The date may change while the program is running. This will cause a problem to the DLL handling Live Data, as each item of Live Data represents one day of data. When a date change is detected the Live Data Objects will need to be archived to the relevant price file and then the contents of the Live Data Objects needs to be reset to start afresh for the new day.

3.8. The Problem Domain Component

Each of the three objects, Pricefile, Live Data and Tick Data, will be inherited from the generic MFC class CObject. This provides some basic functionality such as serialization, and collections (in both lists and arrays).

The Pricefile class will be implemented in a DLL with other interface classes to support the DLL. The Tick Data and Live Data classes will be implemented in another DLL.

3.9. The Task Management Component

Each time a new piece of data is added to a Live Data Object, the value in current field in the Live Data will become a Tick Data item and the new item will become the current item. This will be wrapped up in the interface to the Live Data Object. External access will be initially through 'C' style exported functions from the DLLs these will make up the API for the data management.

If the date of incoming data changes the current Live Data Object needs to be archived to historical Pricefiles, the Live Data Object needs to be cleared out and the new days data needs to begin being stored in the Live Data Objects. This can be triggered externally through the API, and will be monitored for internally in the Live Data Object class each time new data comes in.

3.10. Data Management Component

All the classes contain a serialisation method, which serialises to disk the members that are required to persist. This is generally all the members of the objects mentioned, Tick Data, Live Data and Pricefiles.

The API presents an external method of forcing the DLL's to save their data. These are:

- GLTMP_SaveFile and GLTMP_LoadFile for loading and saving lists of Live Data Objects
- GLTMP_ArchiveDataToFiles to place Live Data Objects into historical price files
- GLPRC_SaveFile, GLPRC_SaveFileAs and GLPRC_LoadFile to load and save price files within the Pricefile DLL.

3.11. End User Feedback

There was a consensus among the end users that being able to view information from multiple sources was a big step forwards. The problem with the system implemented here was that it was not simple to integrate a new data feed into the system, as a working knowledge of the proprietary API was required. Another drawback was the fact that each feed had to provide its own filtering mechanism. This meant that it was fairly difficult to write a feed to place data into the system. The other proprietary component, the storage mechanism also provided some complaints, as users wished to be able to view the data in other applications. The ability to store the data in a standard SQL database would solve this.

3.12. Conclusion

The architecture and system presented in this chapter provided a way of bringing multiple data feeds together to one location. This had not been done before on the Microsoft Windows operating system. A drawback was that each feed had to provide its own filtering mechanism.

For commercial reasons this system was 16 bit and therefore did not take advantage of threading or multiple processor technologies. This also meant that it had a proprietary interface and was not open like COM.

This product is now in production and being marketed by Updata Software. It is currently one of the leading applications in the private investment software market within the United Kingdom.

The architecture presented in this chapter needed to be tested using a different information domain to prove its worth as a generic architecture. It would also be important to test this architecture on a 32 bit Windows platform such as Windows NT, so as to gain a wider commercial acceptance. As the design is based around features of 16 bit Windows it is envisioned that some changes will have to be made. The following chapter modifies this architecture for use on Windows NT with a different data domain.

Chapter 4

REAL TIME SYSTEM FOR TRACKING VEHICLES

This chapter explores the second architecture created for this thesis and used for the design and implementation of a 32-bit real time system, developed in conjunction with Hampshire county council in order to monitor the positions of buses. It provides information for passengers. This system is currently operating in Winchester.

4.1. Introduction

The aim of this system was to reduce information overload for passengers on a public bus service. This system tracks in real time the location of vehicles and provides feedback to the user of approximately how long it will be before each vehicle reaches its following destinations. The initial system has been designed to inform passengers how long they will have to wait at a bus stop for the next bus.

This is a study not only of the engineering of a soft real time system, but also the design and implementation of a solution for a real world problem. Covering the stages and issues that typically occur within the software engineering industry today.

4.2. System Requirements

This project had to be tendered for from Hampshire County Council, the company that won the tender had already submitted a full functional requirement specification as part of this process. The author was then employed to design and implement a system that would meet the specification. From the original specification document the following main requirements were identified:

1. Track the positions of multiple vehicles in real time.
2. Store the positions of vehicles for historical analysis
3. Provide the positions of vehicles to client machines
4. Predict when a vehicle is likely to arrive at a set of given destinations

5. Server .to run on Windows NT, while clients run Windows for Workgroups.

4.3. Overview of the System Design

The system runs on NT Advanced Server, and provides feedback to various other locations via client PC Workstations running Windows for Workgroups, as shown in figure 4-1.

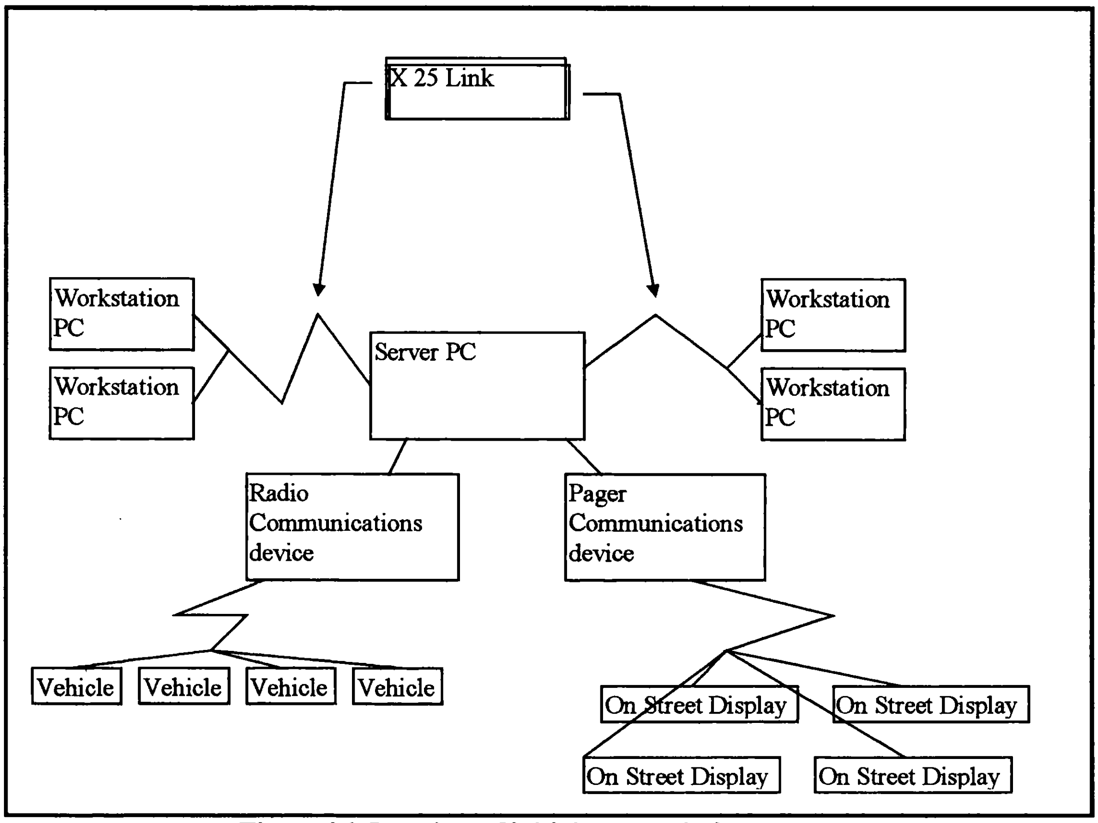


Figure 4-1 Overview of initial system design

The part of this system that is of most interest to us, as it deals with the real time aspect of the data, is in the software residing on the server PC as shown in Figure 4-2.

Each of the DLL's (Dynamic Link Libraries) exports an API (application programming interface) which provides the other modules with access to their functionality. All the DLL's sit in the process space of the Main EXE (Executable). In order to communicate with the communication EXE's (Pager and Radio) a DLL was created which contains shared memory buffers, allowing inter process conversations. This shared memory DLL uses the principles of mapped memory as discussed in chapter 2.

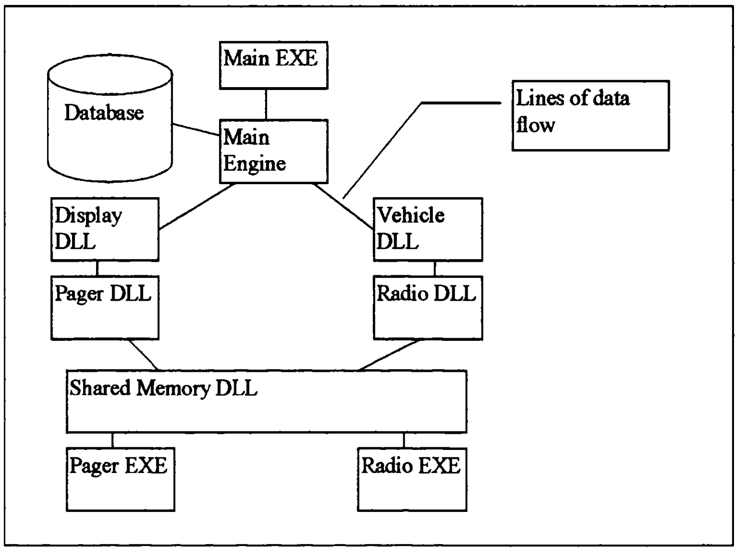


Figure 4-2 Overview of Data Flow in Server Software System.

This inter process communication could have been achieved using a predefined protocol such as DDE or COM. The reason for not using these is twofold.

1. These predefined protocols are defined so as to create an open interface for all programs to communicate with each other. It was not a requirement of this product to be able to share data with any outside applications.

2. Speed of operation is a factor when dealing with a system of this nature. COM and DDE are notoriously slow, requiring a number of handshaking procedures to take place. Also the fact that the data has to pass through a whole separate set of DLL's (controlling COM or DDE) means that they will take longer to process the data.

4.4. Detailed System Design

Each module is now examined in turn.

Display DLL This handles messages going to the passenger information displays on the street. It is passed messages from the Main Engine DLL, this module then decides whether these messages should be sent down to the actual displays. There is a set of rules governing when different types of message should be sent. These rules are encapsulated within this DLL. The main engine also queries this DLL for the information that is actually on the displays.

Pager DLL The main function of this DLL is to encode the messages to send to the displays into the format required by the Pager communications device. It therefore contains an encoding routine along with a queue for buffering messages to be sent.

Pager EXE The actual handling of the protocols required to send messages to the hardware is performed within this executable. It receives messages already encoded from the Pager DLL and sends them to the Pager device for broadcast. As it is a separate process it will run in its own address space and the OS will provide it with time to perform its own processing. This design would allow this process to run on a separate processor or even a different machine in order to increase performance.

Vehicle DLL This DLL stores a list of all the active vehicles that are out on trips. It acts as a buffer for messages going to the vehicles and messages coming back from the vehicles.

Radio DLL As with the Pager DLL the Radio DLL deals with encoding messages, which are sent out by Radio. The Radio DLL also has to deal with decoding messages which are received, from the radios on board the vehicles, and then pass the message back up to the correct vehicle in the Vehicle DLL.

Radio EXE Again similar to the Pager EXE, the Radio EXE handles the actual protocols required by the Radio hardware to send and receive encoded messages which are received from the Radio DLL. Also as with the Pager EXE as this is a separate process it could be run on a different processor or machine in order to increase performance.

Main Engine DLL This is where most of the work for the system is carried out. Including predicting when vehicles are going to arrive at their destinations, handling requests from the Workstation via the Database, and keeping the Database up to date with vehicle information and display sign information. Due to its complexity the main engine is broken up into the following modules, each of which is looked at in turn:

- Application
- Vehicle module
- Custom Message module
- Trip module
- Schedule module
- Display module

Application The functionality of this module is to act as the main application that loads and unloads executables (Pager and Radio). It then waits for events. When these events occur they are dispatched to the relevant module. It also handles start of day initialisations and any other global events, which occur at specified periods.

Vehicle Module On notification from trip handler module this module passes messages on to the vehicle DLL: On a user request this module makes calls to the vehicle DLL. On receipt of a vehicle's location from the vehicle DLL this module calls the trip handler module, passing the latest location. It processes received ticket machine data, as got form the vehicle DLL. It processes received vehicle errors and write the relevant data to vehicle and system error database tables, via the data module.

Custom Message Module This module gets custom messages from the database, then sends custom messages to the display handler module. It maintains a buffer of messages for each display (a list of lists), and decides when each message is activated and subsequently deactivated. This module also responds to a user clearing messages, by removing the message from its internal list and making sure it is removed from the display.

Trip Module This is probably one of the most complex parts of the entire system. The duties of this module include getting assignment data from an allocations table. It receives vehicle locations from vehicle handler module, and then determines a vehicle's location on a trip, it then writes vehicle locations (plus other vehicle data) to the database for workstations to query.

Trips are monitored and diagnostic functions on vehicles then perfomed. Vehicle transit times are sent to the link data module. Forecasts are made within this module and then sent to the display handler module. When a trip is finished this module will archive the trip to the database along with performance details. Finally it enables and disables communications for each vehicle by calling the vehicle handler module.

Schedule Module The schedule module receives vehicle transit times from the trip handler module and maintains a list of all the links with associated transit times. It then gives transit times to the trip module on request. This means that future forecasts can be based on previous transit times. It is in this module that any clever forecasting algorithms could be added at a later date, for example looking at not just average times to travel along a certain link but the time of day that the link is being traversed. It also performs any end of day archiving necessary.

Display Module This initialises the display handler DLL by adding displays from the database. It receives display message requests from the trip handler and the custom message handler, and then passes requests to display handler DLL. It also responds to messages from the display handler DLL that a display is empty or that an error has occurred. The display contents are written to the database when the display handler DLL returns an appropriate status. It also downloads string tables to the displays at end of the day or when a request is made.

4.5. Overview Of The System Implementation

This project could obviously not be completed on time by the author alone. The company already had two good full time programmers and one contractor. The contractor was a C programmer, one of the full time staff had strong Visual Basic skills and the other had strong database and C++ skills.

It was decided that the front ends on the client machines, running Windows for Workgroups, would be written in Visual Basic. The staff member with the Visual Basic skills took on this job.

Meanwhile the other staff could concentrate on the server side which was to be written in C++ with C interfaces on the DLLs. The member of staff with the strong database skills worked on the database, while the contractor started on some of the simpler DLLs. The author concentrated on overseeing the entire operation as well as coding much of the server code.

4.6. Detailed Implementation

The Objects

Each module has a main application object which deals with the details of how that module should behave, as either an in-process DLL or a stand alone application with it's own process space.

The objects of each module are covered in turn.

Display DLL The application object contains a collection of Display objects, and a timer which when triggered notifies each Display object in the list of the timer event.

Each Display object represents an actual display on the street, and contains a list of test strings representing what is currently being displayed. This object applies a set of rules to decide whether a new message should be displayed.

The Display object updates itself on receipt of a timer event notification from the application class. The Display object then sends a message to the Pager DLL to send the current display text to the actual displays.

Pager DLL The application contains a list of Pager objects. Each Pager object represents a pager EXE which can send messages to the on street displays. This object contains a method for encoding the messages and passing them on to the Pager EXE. The messages are passed used the shared memory DLL.

Pager EXE The application contains a method for reading messages from the shared memory DLL and using function calls to the underlying hardware, sends messages via a pager to the actual on street display.

Vehicle DLL The application contains a list of Vehicle objects. Each Vehicle object represents an actual bus. It contains information about its current position, which route it's currently on and what its next route is to be. The vehicle object receives positional data from the Radio DLL and on request returns this data to the main engine DLL.

Radio DLL The application object contains a list of Radio objects. Each Radio object represents a Radio EXE. This object contains methods for both encoding and decoding radio messages. Messages received from the Vehicle DLL are encoded and passed on to the Radio EXE. Message received from the Radio EXE are decoded and passed on to the Vehicle DLL.

Radio EXE The application object contains a method for reading messages from the shared memory DLL and then broadcasts them on the Radio device. This object also receives message from the radio device, which it passes on to the Radio DLL, through the shared memory DLL.

Main Engine DLL The application object contains an object to represent each of the six components of the Main Engine DLL. It also contains the time for the start of day and a list of registered executables. It contains access methods for each of the component objects, in this way each component can communicate to any other by accessing it through the application class. Each module or component is examined in turn by looking at the classes present in that module.

Vehicle Module

Module class - this represents the activities of the module, which are:

- Handle vehicle error
- Handle incoming vehicle locations
- Handle incoming ticket machine data
- Add vehicle
- Remove vehicle
- Remove all vehicles

Custom Message Module

Module Class - this represents the activities of the module, it contains a list of custom message displays and has a method to initialise the displays.

Custom Message Display Class -a class which represents a display contains a stack of custom messages, along with methods to perform the following tasks:

- Add a message
- Remove a message
- Send the current message
- Clear the current message

Custom Message Class - contains the following members:

- List of custom message lines
- Whether or not message is persistent (flag)
- Whether or not message is part of cluster (flag)
- Start date and time
- Finish date and time

Custom Message Line Class - contains the following members:

- Line number
- Type of message
- Message table number

Trip Module

Module Class - This polls the allocations table for new allocations and scans active vehicles. It can then perform location updates. It contains the following members:

- List of active vehicles
- Pointer to the database
- Today's full schedule
- Tomorrow's limited schedule

<u>Active Vehicle Class</u> - this represents an active vehicle, which is either on a trip or about to start a trip. It writes the vehicle location to the database, creates active trips when required using allocation from allocations list and static data, handles trip cancellations, writes errors to database, and updates link data. It contains the following members:

- List of day's allocations
- List of next trips
- Position on current trip
- Trip status

This module also provides the following public methods:

- Add allocation
- Cancel allocation
- Location update

<u>Active Trip Class</u> - used to represent a trip which is about to start or has started. It contains a method for archiving the trip data when it is finished along with the following members:

- Ordered list of trip displays
- Ordered list of links for trip
- Ordered list of timing points for trip
- Start offset
- Finish offset

<u>Trip Display Class</u> -this represents a single display on a trip and contains a method for updating the forecast message on the display. It also contains the following members:

- Location
- Message number
- Forecast arrival time
- Displayed minutes
- Time of display

Trip Searcher Class - This class polls the database for vehicle allocation changes, and then passes changes to the module class.

Display Module

Module Class - handles errors from display handler DLL, writes display contents to the database, adds and removes displays, downloads the message table, retrieves scheduled times from schedule module when a display is empty, receives forecasts and clears from the trip handler, and receives custom messages from the custom message handler. This class provides public methods to:

- Handle forecasts
- Handle custom messages
- Handle display errors
- Handle empty displays
- Download the message table

Schedule Module

Module Class - contains a list of tables, and methods to:

- Create tables starting at a given date
- Get next vehicle
- Set vehicle arrival

Table Class - contains a list of cell lists, one for each display, and has the following methods:

- Get next vehicle
- Set vehicle arrival

Cell List Class - contains a list of cells. It also has methods to:

- Get next vehicle
- Set vehicle arrival

Cell Class - contains the following members:

- Board number
- Trip number
- Due time
- Actual arrival time
- (Vehicle number)

4.7. The Events

The key event that occurs during the running of this system is that of a receipt of a new position for a vehicle. This leads to a new prediction being made that then gets sent to the signs on the street. With a small number of buses and signs to cope with this is a relatively trivial task. As the number of buses increases the software performance needs to be kept high as the signs on the streets need to be refreshed at least every twenty seconds.

By using separate processes to deal with the actual hardware communications performance benefits can be leveraged by scaling up the number of processors running the server software. The performance bottleneck then becomes the communication bottleneck between the main system and the communication software. This system uses shared memory to pass data across the process boundary, as this is the fastest method.

4.8. The Problem Domain Component

The main problem that this system solves is information dissemination. The information informs passengers when the next bus will arrive at a bus stop, and the system ensures the passenger gets this information updated regularly.

4.9. Data Management Component

All the data for this system is stored in an industry standard SQL database on the server. This can then be used to perform an historical analysis of the performance of various factors, such as the routes or the vehicles. It also allows for other software to be written to perform analysis on the data without knowledge of the internal structures of this system.

4.10. End User feedback

Feedback for this system has come from three sources Hampshire County Council, the Bus Company and bus passengers.

Both the council and the Bus Company complained about the fixed nature of the system, that it was not easy to change the route a bus was on, or details of the timetable. In terms of the real time data delivery there was no complaint.

Most of the feedback from the passengers was regarding the lack of accuracy of predictions. This is due the unpredictable nature of transport systems, and has little relevance to this thesis.

4.11. Conclusion

This system was written with a view to making it usable in other scenarios than tracking buses. It could be used to track any vehicle that is travelling on a predefined route, and make predictions of arrival times. It could also be used without the prediction component to simply track the location of any mobile object.

The bus tracking and passenger information system derived from the core technology is currently running in Winchester, and is easing the information overload problem for bus passengers wishing to know the arrival time of their next bus.

The architecture presented in this chapter has expanded the ideas initially provided in the previous chapter to create a system that accepts multiple data feeds and runs on the Windows NT operating system. The idea of memory sharing has been transferred, although implemented differently. Also hierarchical storage of the data has been utilised as in the first system.

In order to take this architecture further it was now necessary to generalise the data types being accepted and stored and also provide a generic interface for filtering. The architecture and system presented in the following two chapters attempts to do this.

Chapter 5

GENERIC RTD SYSTEM FOR MULTIPLE DATA FEEDS

This chapter analyses how the generic RTD system was designed to take any form of real time data from multiple sources and provide the data to a number of client programs. This system was developed in collaboration with Black Ace Software Engineering a company specialising in real time systems on PC's.

5.1. Introduction

It was realised from the previous projects that it would be possible to create a system that could handle data in multiple formats from multiple sources. The architectures studied in the previous chapters have the ability to take data from more than one source, the problem was that the data has to be in a fixed format. Another drawback of the previous architectures was the lack of a general model for filtering the data.

This was going to have to change in order to create a more generic system, where the format of the data cannot be predetermined. In order to create such a generic system the architecture was designed based on previous architectures but addressing the issues of generic data types and an open model for filtering. This chapter looks at the software engineering techniques employed to create such a system along with the actual design. Also considered is the issue of how the system was designed in order to plan for future changes. This system is called RTD, which stands for Real Time Data

The rest of this chapter looks at the first three stages of the waterfall model as applied to the engineering of the RTD system.

5.2. System Engineering

The reasons for developing RTD were to create a system that could:

1. Take any type of data as its input; whether the data was from a financial institution or the position of a vehicle would not make any difference as

to how it is input to the system. There would need to be an input interface that could cope with multiple data types.

2. Take data from more than one feed; this input interface would also need to cope with input from more than one source.
3. Filter the data for information that was of interest to the user; some mechanism for deciding what information the user wants would be required.
4. Provide the data to one or more client programs; an interface for exporting the data in a timely manner to client programs would be required.
5. Store the data for later analysis; the system would require some form of database system to archive the data to a storage medium.

From this list the diagram in figure 5-1 was drawn up.

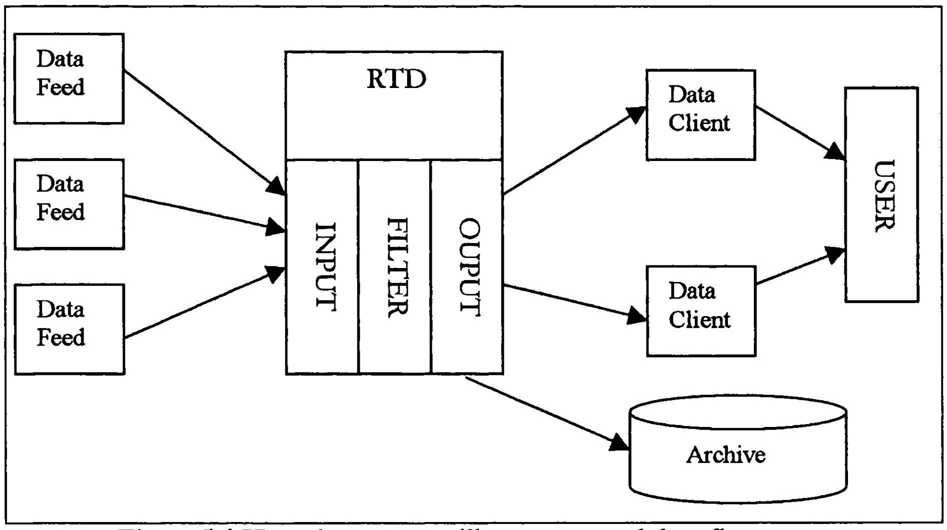


Figure 5-1 How the system will act to control data flow.

5.3. Analysis

The information domain of the system was not easy to define, as there was not a specific goal for this system but rather to provide a generic solution to a problem that had been encountered previously. All that could be said was that any data entering the system could be placed in this system and then filtered and provided to client programs.

This system would be required to filter the data and only keep that data which the user would require. It would also have to archive the data, it would be prudent to allow the user to choose what should be archived and how often. Depending on the inputs and filtering huge amounts of data could amass very quickly.

At this stage is would not be possible to gauge performance levels as the input volume is not a known factor. For this reason the system could not be defined as a hard-real time system.

The interfaces required are both for the input of data as well as the extraction of data. There would also need to be some way of extracting the archived historical data. In order to provide a generic interface that was expandable an open and uniform data transfer protocol would need to be chosen.

5.4. Top Level Design

The diagram in figure 5-2 was drawn up as an initial idea for the lines of communication between the code modules and the storage system. This was based on the architectures presented in the two previous chapters.

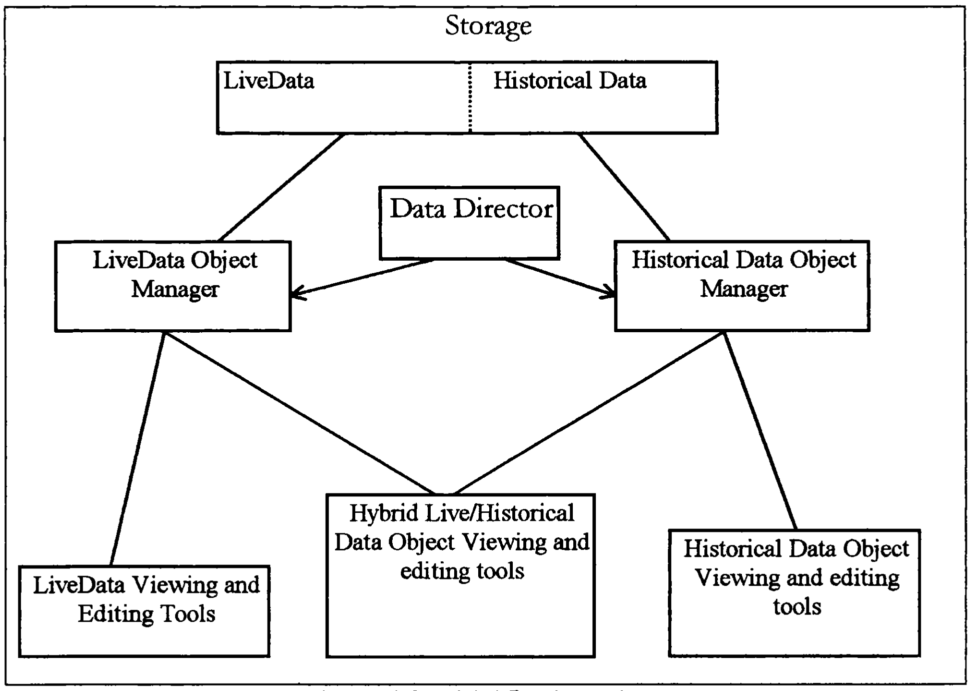

Storage
LiveData
Historical Data
Data Director
LiveData Object Manager
Historical Data Object Manager
LiveData Viewing and Editing Tools
Hybrid Live/Historical Data Object Viewing and editing tools
Historical Data Object Viewing and editing tools

Figure 5-2 Initial Design Ideas

Storage

The storage engine would need to be user definable as either an ODBC compliant relational database, or a proprietary storage mechanism which would be based on the Compound Storage Model using the COM Structured Storage implementation. It would be necessary to provide COM storage methods for the data objects in order to link and embed them in other container applications and so it would be fairly trivial to take this a stage further and provide the full storage mechanism if no database is required. An ODBC relational database would contain the tables shown in figure 5-3.

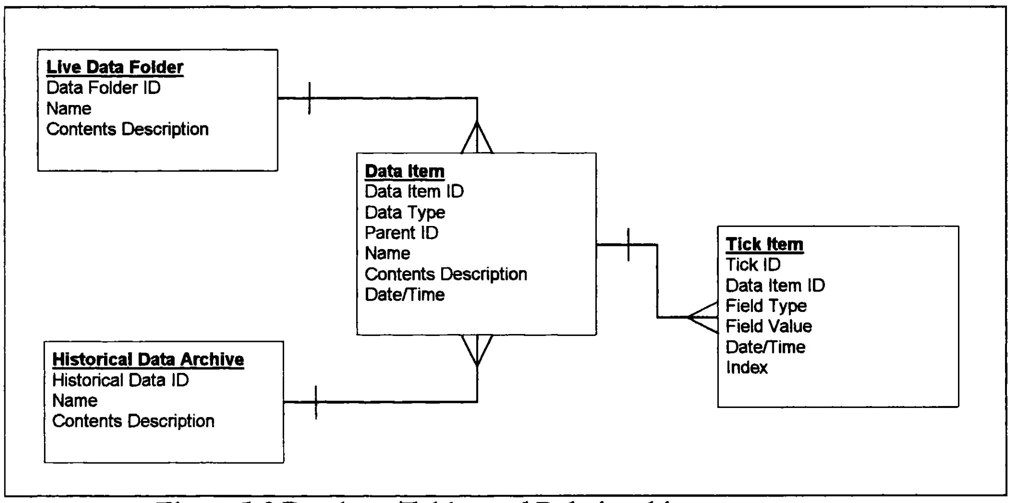


Figure 5-3 Database Tables and Relationships

System Folder Table - used as a means of grouping the Data Items that are live.

Historical Data Archive Table - for grouping together data items relating to the same thing over a period of time.

Data Item - can represent either a live piece of data or an archived piece of data, the Data Type field will indicate which. The parent ID will then reference either a System Folder or Historical Data Archive.

Tick Item - represents a field and its value at a given time for a Data Item. The Data Item ID is a foreign key pointing to the Data Item to which this field belongs. The Field Type field indicates which field this tick represents E.g. Trade price. The Field Value gives the value of that field. The index can be used to indicate the number of ticks for the field given.

LiveData Object Manager

The LiveData Object Manager would be a process in the system which accesses the data base and converts the live data into exportable COM objects conforming to the WOSA standards plus proprietary extensions.

Historical Data Object Manager

The Historical Data Object Manager is another process in the system that accesses the database and exports entire histories of a certain Data Item wrapped as an COM object.

Data Director

This would provide the user with the functionality to decide where the data is stored and in what format. It will also be the method by which feeds register themselves for automatic execution and termination. It would therefore be a container application for the feeds property sheets. This could be integrated into the LiveData Object Manager.

Applications And Tools

COM aware applications and tools could then be written and used to manipulate this data. It was foreseen that the Object Managers would have a permission system, so that only those tools with the correct permissions could write data back into the database. This would allow for differing levels of developers. Ones that know the API's to write data into the Objects, and ones that only know how to read the data.

Feeds would simply become a subset of the tools that talk to the Live Data Object Manager, and have permissions to write data back to the database. They would also need the extra functionality of being able to create new Data Item Objects.

No programs should ever read or write directly from/to the database. This would provide a layer between the applications and the Database so if the method of storage changes, say to an OO database system, only the Managers will need to be changed.

Networking

In order to ensure that the product will work over a network it was considered important to look into designing this capability from the beginning. By using a Database to store and retrieve the data the client-server model could be used as a layer between the database and the object handler modules.

Using full RPC (rather than LRPC) COM is now providing a distributed mechanism for conversation with its interfaces across machine boundaries known as DCOM. This would mean that if the interfaces are provided as COM interfaces they could be used over a distributed system.

WOSA Compliant Real Time Engine

Microsoft have already produced some guidelines for creating real time systems for financial market data, in a paper called WOSA Extensions for Real Time Market Data [4], otherwise known as WOSA/XRT. WOSA stands for Windows Open Systems Architecture.

The design specification they provide basically sets some rules for how to use COM in order to follow a standard, which would mean that other applications which also follow the standard would be able to share the data. The extra features that were addressed within this paper have now been integrated into COM.

Storage of Real Time Data

The aim of this section is to set out the structures and objects in which live data (real time) is stored in this project. This section is split into two parts, the first part covers the storage of real time data at run time, the second part examines offline storage of real time data.

The application, which is derived from this design, will be a 32 bit Executable, it must support an COM interface, for data retrieval and display purposes. It would also provide an COM interface for data submission.

It was important to follow the guidelines given in the WOSA XRT design specification and so where possible comparisons have been made with the XRT specification.

Run-Time Storage of Real Time Data

The Objects / Structures

Table 5-1 lists the objects being used in the structure of the Live Data application. Each object (barring the collections) will be able to draw itself. In this way an object linked or embedded into a document will update its contents and display the update without the containing document having to worry about it.

Black Ace Software Engineering Name	**WOSA/XRT Name**	**Class Name**	**Standard COM Interfaces supported**
LiveData	Application	BACApplication	
System Folder List	DataObjects Collection		
System Folder	Data Object	BACRTDDoc	IDispatch IDataObject
Live Data Item List	DataItems Collection	BACDataItems	
Live Data Item (Tempdata)	Data Item	BACDataItem	IDataObject
Field List	Properties Collection	BACFields	
Field	Property	BACField	IDataObject
Value List	Properties	BACValues	
Value	Property	BACValue	

Table 5-1 Objects in the Live Data Application

BACValue - This object contains a value and a time. The value is stored as a VARIANT type and the time as the time and date.

BACValues -A collection of Value objects

BACField -This object contains a name and a value list. The name is a CString and the value list a COblist of values. It also contains a pointer to the COM server object for this item, which is really just a helper class, and another pointer to the data object class object which owns it.

BACFields -A collection of Field objects.

BACDataItem - An object which contains a BACProperties object. Generally known as a Live Data item.

BACDataItems -A collection of data items, or a List of Live Data Items.

BACRequest -An object which contains a BACPropeties object. This is part of a new concept, where a list of requested data is stored in the Data Object.

BACRequests -A collection of request objects.

BACDataObject - An object which contains a BACDataItems object, and a BACRequests object. Generally known as a system folder.

The Relationships

The relationships as suggested by the WOSA/XRT specifications can be seen in figure 5-4. Alongside the WOSA/XRT names I have placed the Black Ace Software Engineering interpretation in a text box.

The WOSA compliant system in figure 5-4 caters only for data now (live or real time data) and doesn't cope with ticks or lists of other attached information such as news files or movies. Each Property contains one value.

A tick is in fact an object with name, value and time properties, so a list of ticks would be a list of these objects. The WOSA/XRT specification doesn't have this concept.

In order to cope with additional lists of Values within each Live Data Item there will have to be a new interface to reach previous ticks. In this way we remain WOSA/XRT compliant and reuse the objects we already have in order to attach further lists of data to the Live Data Items.

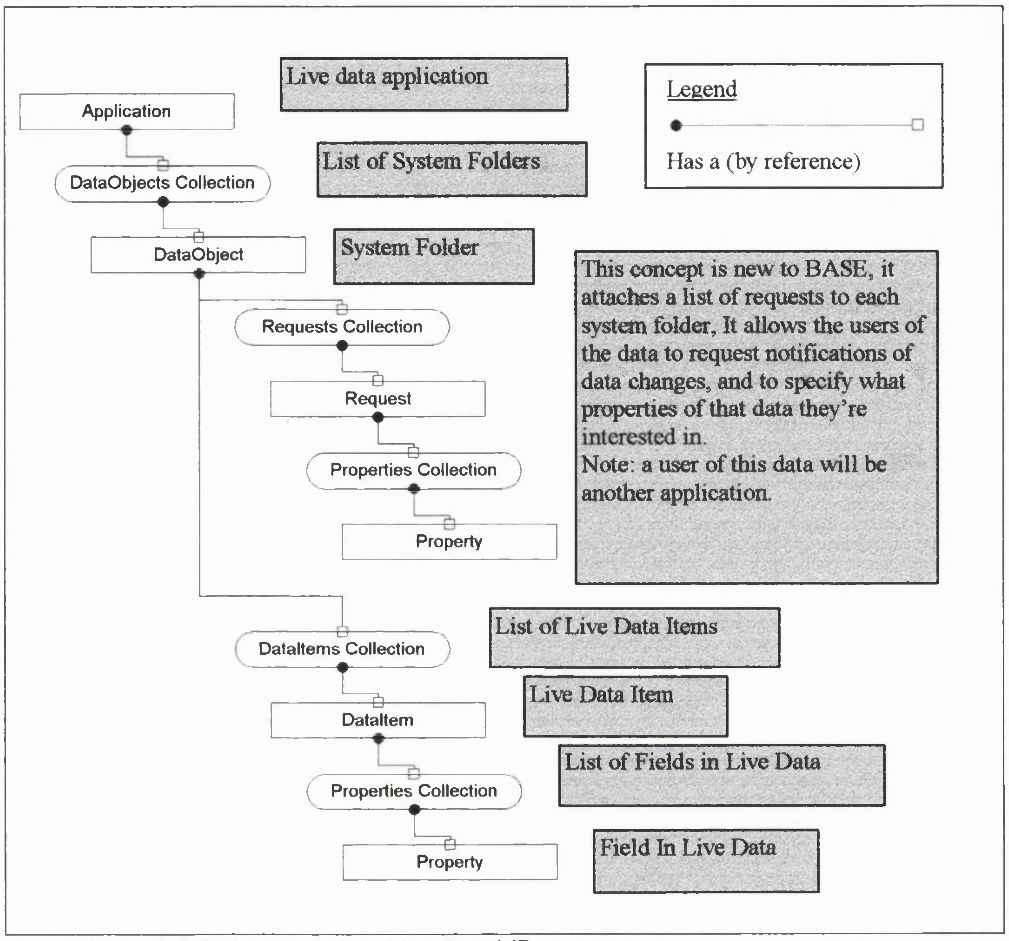

Application
DataObjects Collection
DataObject
Requests Collection
Request
Properties Collection
Property
DataItems Collection
DataItem
Properties Collection
Property
Live data application
List of System Folders
System Folder
List of Live Data Items
Live Data Item
List of Fields in Live Data
Field In Live Data
Legend
Has a (by reference)
This concept is new to BASE, it attaches a list of requests to each system folder, It allows the users of the data to request notifications of data changes, and to specify what properties of that data they're interested in.
Note: a user of this data will be another application.

Figure 5-4 Object Relationships for RTD

The document-centric view of the world, as perceived by Microsoft, is still maintained in that the application has the Data Objects as documents. The Data Objects in Black Ace Software Engineering terms are System Folders and so the Live Data application must support MDI.

Disk Storage of Real Time Data

Using the Structured Storage model of file storage this project needed to implement its own storage based on the Compound File storage implementation provided by Microsoft.

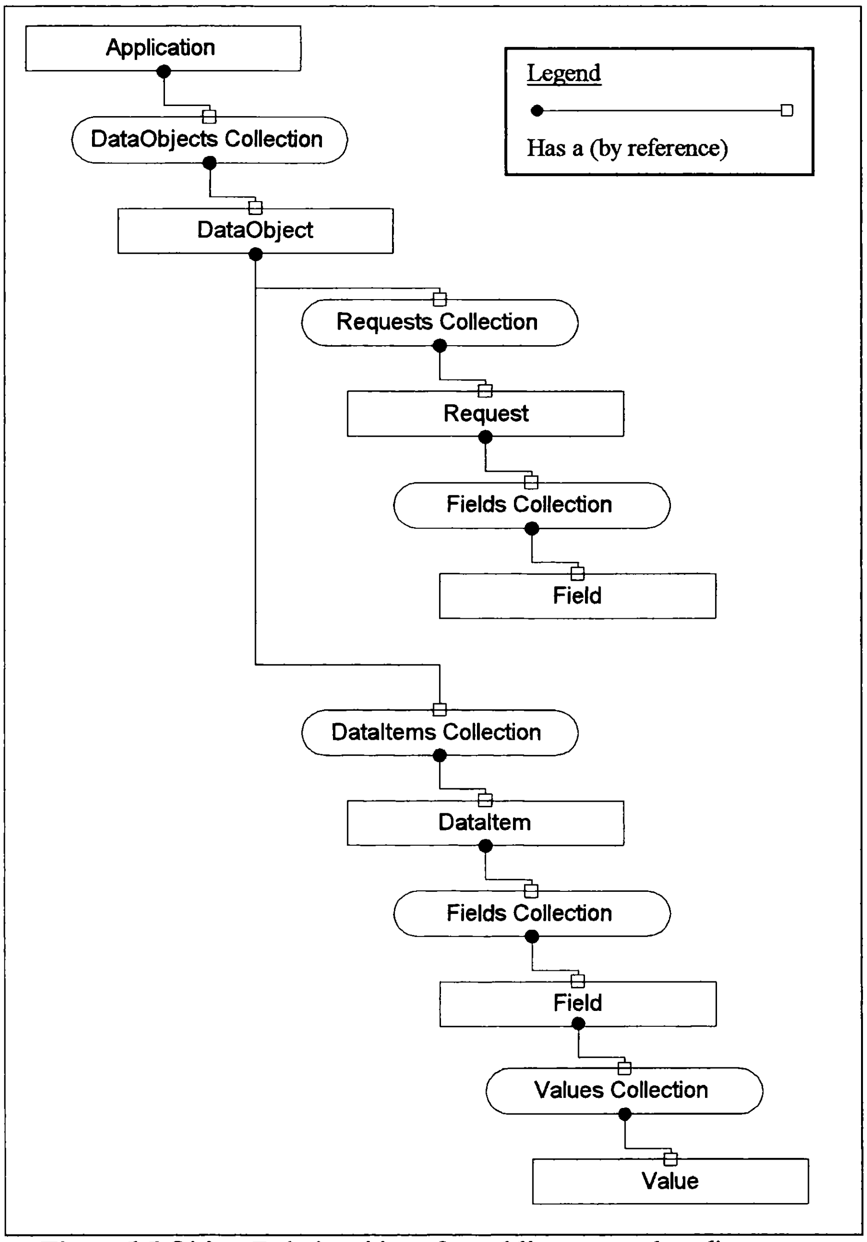


Figure 5-5 Object Relationships after adding new values list

The main reason for doing this is initially to present Windows 95 & NT users with long filenames and property sheets. In the long term the Cairo NT OS will be enforcing use of Compound Files as it runs on a new filing system called OFS (Object Filing System).

The Objects / Structures

The structured storage model provides two basic objects, a storage and a stream. A storage can be seen as a directory and a stream as a file, the difference being that these all live in one file. In essence a file system within a file.

For implementation purposes COM provides IStorage and IStream objects.

The Relationships

Initially lets take a look at the choices of storage for Live data.

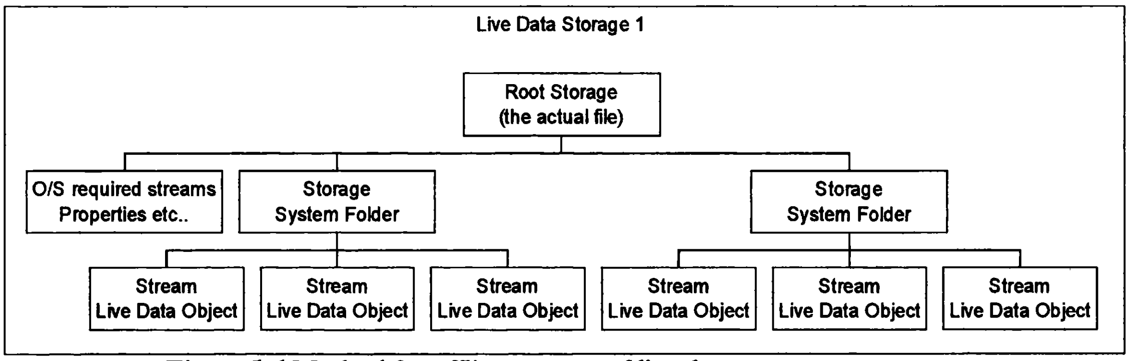


Figure 5-6 Method for offline storage of live data

This first approach limits us to only having one level of folders within live data. This is similar to how it is done within Updata Software's 4th generation of software, as described in chapter 3.

Another approach would be as shown in figure 5-7. This approach looks a lot more complicated, but implementation of this would be very little above that of the first method. The problem would be in how to deal with data like this in applications.

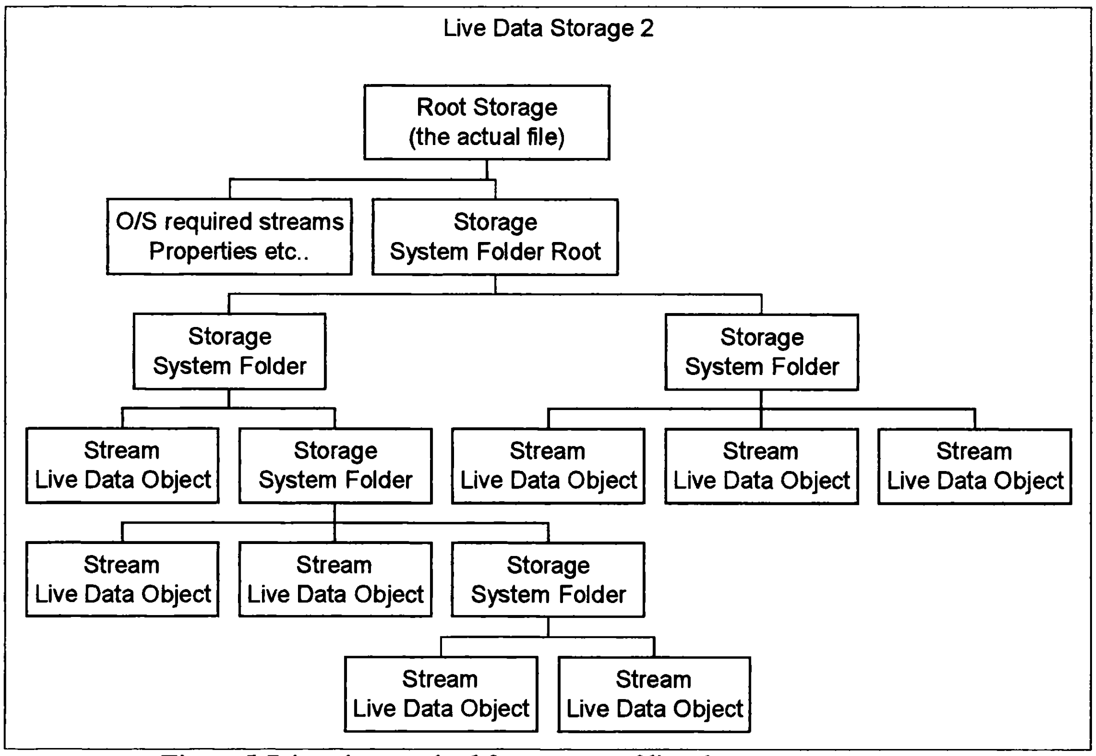


Figure 5-7 Another method for storage of live data

It would not be impossible to move from the method presented in the first place to the second suggestion. For this reason the first method could be used initially for storage of live data, and then the second method moved to if needed.

A completely new approach would be to look at live data with a document-centric view where each folder could be seen as a document containing lists of data items which are related in some manner. This would mean that there could be several files to represent the live data at any one time.

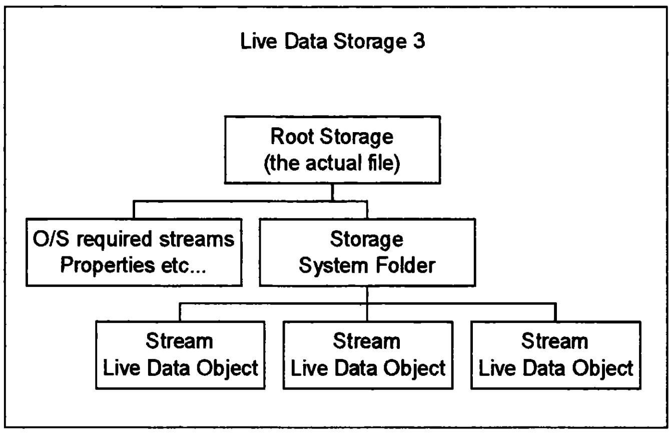


Figure 5-8 Third method for live data storage

In order to store these objects we need to work from the bottom up, as by defining a mechanism to store the smallest components first, the larger components will find much of the work has been done for them because they are essentially collections of the smaller components.

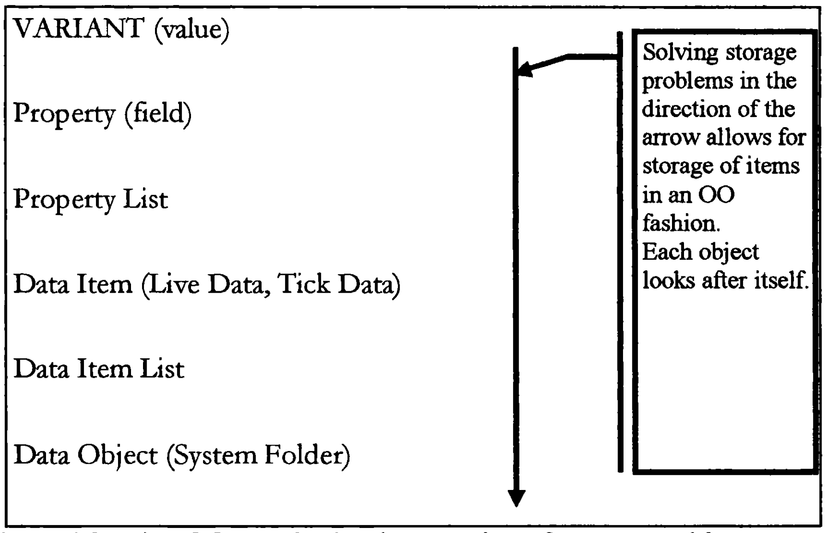

VARIANT (value)
Property (field)
Property List
Data Item (Live Data, Tick Data)
Data Item List
Data Object (System Folder)
Solving storage problems in the direction of the arrow allows for storage of items in an OO fashion.
Each object looks after itself.

Figure 5-9 Using OO to solve implementation of storage problem

Storage of Historical Data

System Folders

The next problem is how to deal with historical data, the data derived from live data. The simplest way would be to extend the concept of 'Price Files' (as used in the Private Investment System in chapter 3). into compound files as shown in figure 5-7.

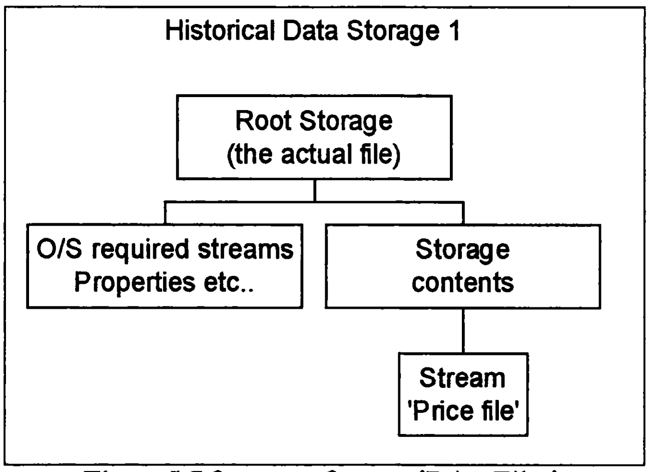

Historical Data Storage 1
Root Storage (the actual file)
O/S required streams Properties etc..
Storage contents
Stream 'Price file'

Figure 5-7 Structure Storage 'Price Files'

Another method would be that each file represents a system folder. As shown in figure 5-8.

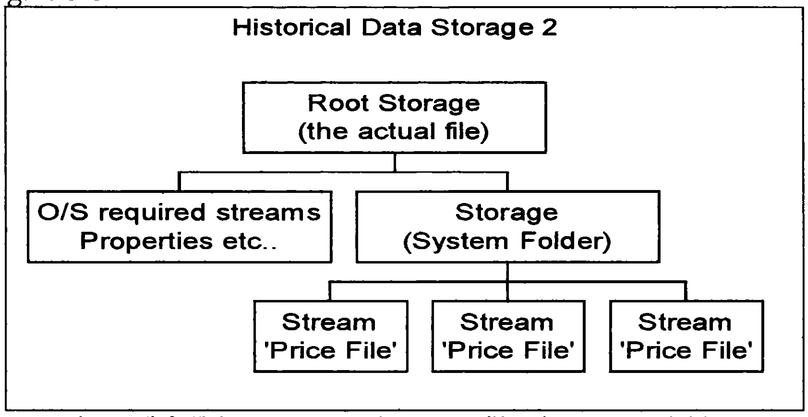

Historical Data Storage 2
Root Storage (the actual file)
O/S required streams Properties etc..
Storage (System Folder)
Stream 'Price File'
Stream 'Price File'
Stream 'Price File'

Figure 5-8 Using structured storage files for system folders

This would solve the problem that exists with system folders being directories, there is no way to attach information to a whole system folder. It would also create other problems that we would have to be solved, such a picking a price file to draw a chart on.

Taking it to its extreme, the whole world of information could be stored in one file.

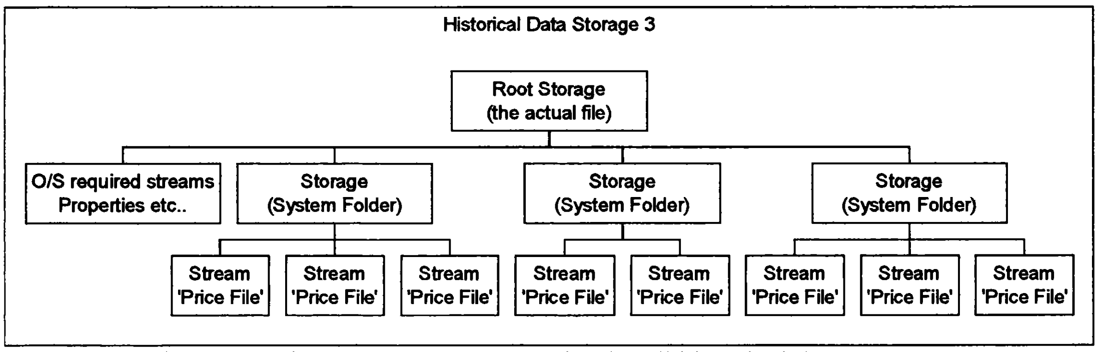


Figure 5-9 Using a single structure storage file for all historical data

This would make for an enormous cumbersome file, but may have advantages in that we could set properties for all the data in one go. Also it would be easy to keep track of links between folders. This approach would be most suitable if some form of database management system was being used.

Method 2 looks like the most feasible, where each actual disk file will represent a folder, a container of price files. The reason for not choosing the 3rd method is that the files would become huge as large quantities of data gets collected. It would be quite feasible to have files greater than several hundred megabytes in size, this is not currently regarded as sensible, although as the cost of storage falls it may be worth returning to this concept in several years.

Custom Folders

The difference between Custom Folders and System Folders is that Custom folders only store a link to the data files whereas System Folders actually store the data files. The Custom Folders will store their links in streams which each represent a System Folder.

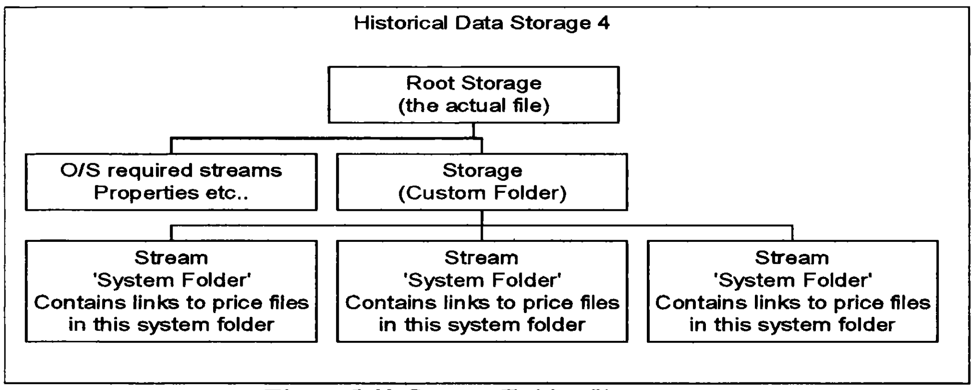

Historical Data Storage 4
Root Storage (the actual file)
O/S required streams Properties etc..
Storage (Custom Folder)
Stream 'System Folder' Contains links to price files in this system folder
Stream 'System Folder' Contains links to price files in this system folder
Stream 'System Folder' Contains links to price files in this system folder

Figure 5-10 Custom Folder files

The Problems

One of the main problems in the past has been that in order to store a field it needed to be present in the records being stored. These records were determined at design time and therefore any expansion of the system that was required meant that the record size would change and therefore a conversion routine would have to be written. The WOSA/XRT specification along with COM compound files solves this by allowing us to specify what is being stored in a stream.

Data Access and Inter Module Communication

This section looks at how the data in the system will be accessed externally by other applications and also how messages will be passed between various modules involved in the system, both internally and externally.

As specified in the WOSA/XRT design paper the system will provide the COM automation interfaces shown in Appendix 3. It will also provide additional interfaces in order to increase the available functionality of the system.

Downlists

In order to accept data from more than one feed it is considered necessary to have some kind of generic data stripping system. The private investment system described in chapter 3 used a system called Downlists (short for download lists). This is a list of what data to strip off the feed and where to place this data in the system. Up until now these downlists have been feed specific, each feed has had its own downlist structure and system. What was needed was a more general-purpose downlist system that is built into the Live Data engine. The requests system as specified in the WOSA/XRT specification is exactly this.

In order to make this more generic an interface would need to be provided to allow third applications to make adjustments to the requested data. This is the system that will be used for filtering the data. The first version will merely allow the user to select that data that they wish to collect. Future versions could use some intelligent systems to adjust the requests at run-time and provide a powerful filtering system.

Interface for Additional Components

One of the key components which was required for this system was a data feed program. An aim of this project was to make it very simple to plug data into the system. To this end an OLE automation interface has been created to allow data to be pumped into the system using a very few lines of Visual Basic code. Using OLE Automation this would be possible and the extract of code in figure 5-11 gives an example of how data could be input into the system.

```
Private Sub Form_Load()
   Set rtd = CreateObject("BASE.RTD.APPLICATION")
   If rtd Is Nothing Then
      MsgBox ("Error Loading Real time engine")
   Else
      Set dataFeed = rtd.Feed
      If dataFeed Is Nothing Then
         MsgBox ("error getting data feed pointer")
    End If
End Sub

Private Sub SendData_Click()
   If dataFeed Is Nothing Then
      MsgBox ("error getting data feed pointer")
    On Error Resume Next
  Else
    lval = dataFeed.FeedData("News", "Finance", "Headline", Text1.Text)
    lval = dataFeed.FeedData("News", "Finance", "Story", Text2.Text)
  End If
End Sub
```

Figure 5-11 Extract of Visual Basic code to input data

5.5. Summary

This chapter has started to present the architecture designed for the RTD system. It builds upon the lessons learnt from the previously examined architectures.

This architecture allows data to be input from multiple sources, filtered and provided to data clients. The data type is not constrained and the architecture has been designed to ensure a timely provision of the data.

The following chapter takes the design into the implementation stage and then looks at how the system can be tested while being implemented and after implementation has finished.

Chapter 6

GENERIC RTD SYSTEM IMPLEMENTATION

This chapter describes how the RTD generic system was implemented, and tested. It examines the reasons for choosing certain programming languages, the objects that were created and relationships between them. It also examines the methodologies of testing systems, and how they were applied to the RTD system.

6.1. Introduction

Before the implementation stage could begin a programming language had to be chosen. The first section of this chapter looks at how a suitable language was chosen. The next section in this chapter shows how the design was taken to the stage of coding, and how the implementation solves the specific problems of event handling and data management. The final part of this chapter shows the testing methods used to ensure that the implementation produced a robust and solid product.

6.2. Language Choice And Style

It is important when embarking on the implementation stage of a piece of software to pick not only the correct language but also a coding style, which should be adhered to strictly throughout the life cycle of the software. In this way all the code within the project will have the same look and feel, making it far easier to maintain. Setting a coding style is also done for commercial reasons, the code from RTD is likely to be maintained or enhanced by other programmers in the future. By picking a set of rules in the beginning it is more likely that these other programmers will be able to understand the code and also follow the rules.

The language chosen depends on the environment being programmed for as well as the design structure. The system has been designed using an object oriented (OO) methodology, and so it would make sense to use an OO language for implementation. The problem with many OO languages such as Smalltalk or Java is that they are interpreted and not compiled. This would lead to unacceptable performance for a system that has to deal with data as quickly as possible. For this reason C++ was chosen. Being more of a hybrid language it allows an OO structure to the program while permitting some lower level coding to perform time critical tasks. Another good reason for choosing this language is the vast support that is available for it in terms of compilers produced and knowledge bases of information.

The paper shown in Appendix 6 was drawn up in order to lay down the rules for programmers on the project.

6.3. The Objects

In order to simplify the implementation process an existing class library was chosen to provide a foundation of functionality, from which the classes for this project could be derived. The library chosen was the Microsoft Foundation Classes. The main reason for choosing this library is that it wraps the functionality provided by the OS API more fully than any other library available.

Appendix 7 provides a complete listing of all the classes created for this project. As can be seen these follow very closely to the objects drawn up at the design stage. This is a good thing as it shows that the design was correct. If at this stage it was discovered that the implementation of the classes didn't closely follow the objects designed, it would be worth going back and reviewing the design stage. The following section describes the classes implemented. Appendix 8 provides diagrams to show the classes and their members and methods.

The Application Class Provides all the OLE functionality required for the OLE automation Application object.

The Data Objects Class This class provides the interface for collection of data objects in the application. It provides a hidden method for retrieving the enumerator.

The Data Object Enumerator Class This class provides the functionality for traversing the data object collection. It basically wraps the IEnumVariant COM interface, which is represented here by the nested XEnumDataObjects class.

The Data Object Class This is the most complex class in the whole system, it represents both the document as seen in Microsoft's document centric view of the World, and also the COM data object providing an IDataObject interface and an IDispatch interface.

The Data Items Class This class provides the interface for a collection of data items in a data object. It provides a hidden method for retrieving the enumerator, the same as the data objects class.

The Data Item Enumerator Class This class provides the functionality for traversing a data item collection. It basically wraps the IEnumVariant COM interface, which is represented here by the nested XEnumDataObjects class.

The Data Item Class This class represents a set of data for one particular item, such as a share price or a particular vehicle.

The Fields Class This class represents a collection of field classes. As with the previously examined collections it contains an enumerator for iterating through the fields.

The Field Enumerator Class Similar to the previous enumerator classes.

The Field Class Contains a list of values for the field.

The Values Class This class represents a collection of value classes. As with the previously examined collections it contains an enumerator for iterating through the values.

The Value Enumerator Class Similar to the previous enumerator classes.

The Value Class This contains a time and a value for that time. It also contains a pointer back to the field to which it belongs.

The Requests Class This class represents a collection of request classes. As with the previously examined collections it contains an enumerator for iterating through the requests.

The Request Enumerator Class Similar to the previous enumerator classes.

The Request Class This contains the name of the request, a data object that it refers to and a list of requested fields.

6.4. The Relationships

The class relationships are shown in figure 6-3. All the classes beginning with the BAC prefix are Black Ace Software Engineering classes created for this project. The classes beginning with just a C are part of the Microsoft Foundation Classes library. The interfaces exposed by the classes are shown in Appendix 5.

6.5. The Events

This section examines the externally triggered events that can occur while the system is running and how the implementation of RTD deals with these events.

Data Input

The first event examined is the input of data from a feed. This occurs when an external feed program calls the feed data automation function. This function takes 4 parameters, the name of the data object (folder), the name of the data item, the name of the field for which this data belongs and the value. The first three must be of type string (BSTR), and the last is a variant, it one can be of multiple types. This allows the system to store both numerical values as well as text for different fields. One problem that this does lead to is that there is no way to type check individual field values at run time.

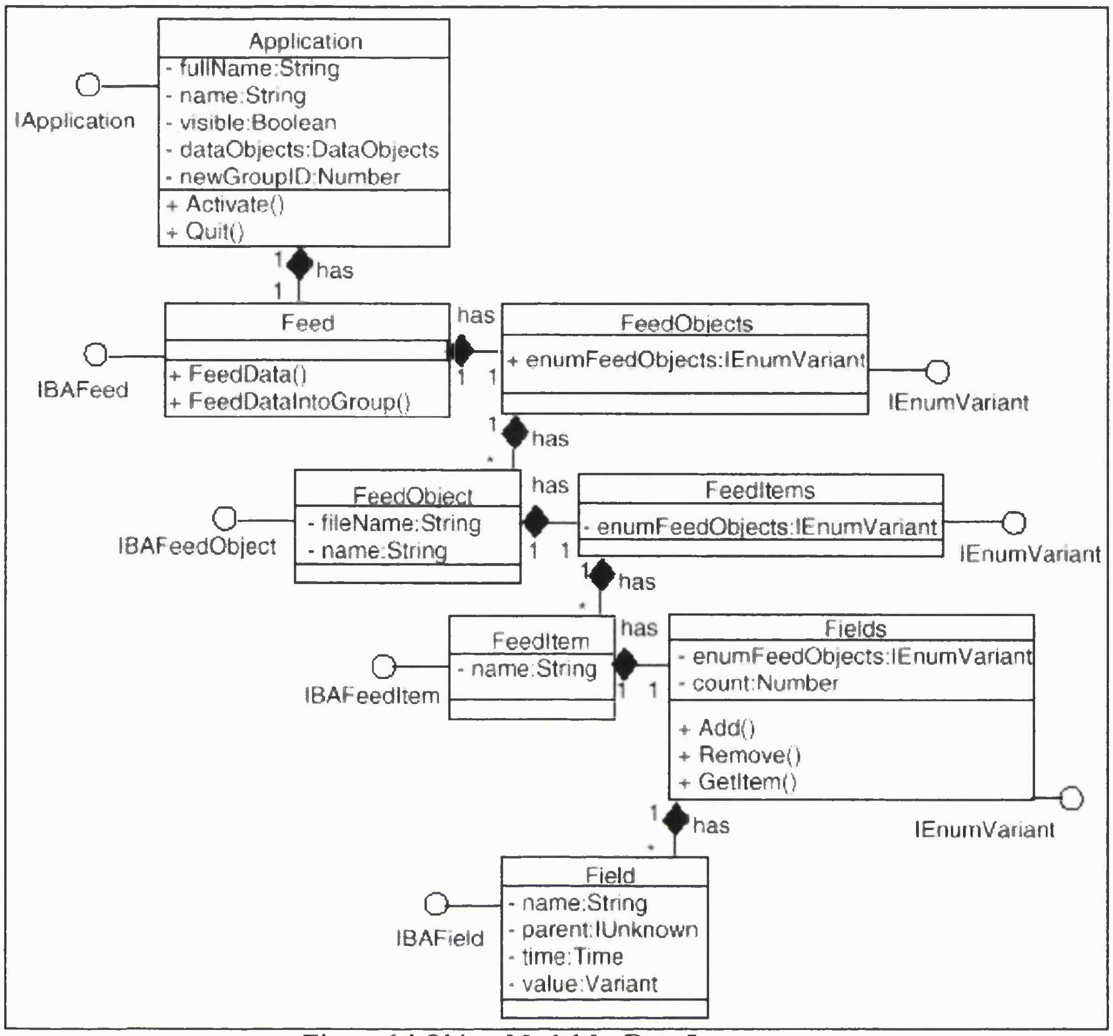


Figure 6-1 Object Model for Data Input

Figure 6-1 shows the object model for the input component. The `Application` class is the only class of the `Input` component that can be directly accessed from other, possibly distributed COM component. These are likely to be implementations of data feed components that act as adapters and convert proprietary data formats into a form supported by RTD. The `Application` class within RTD is a singleton, which means that there will only one instance of that class. In order to feed data into the RTD system, the Data Feed components will need to gain access to a `Feed` object, from the `Application` object. Data can be entered into the system by invoking `FeedData()` and `FeedDataIntoGroups()` from a `Feed` object. For example, the `FeedObject` could represent a type of feed, such as ReutersSSL4, the `FeedItem` could represent a financial instrument, such as British Telecom shares and a `Field` could represent the bid price. The concept of a data group was created in order to allow the feed component to create links between certain types of data. A `Feed` object can get a group identifier from the Application using `NewGroupId()`.A group identifier is a unique number for the RTD system.

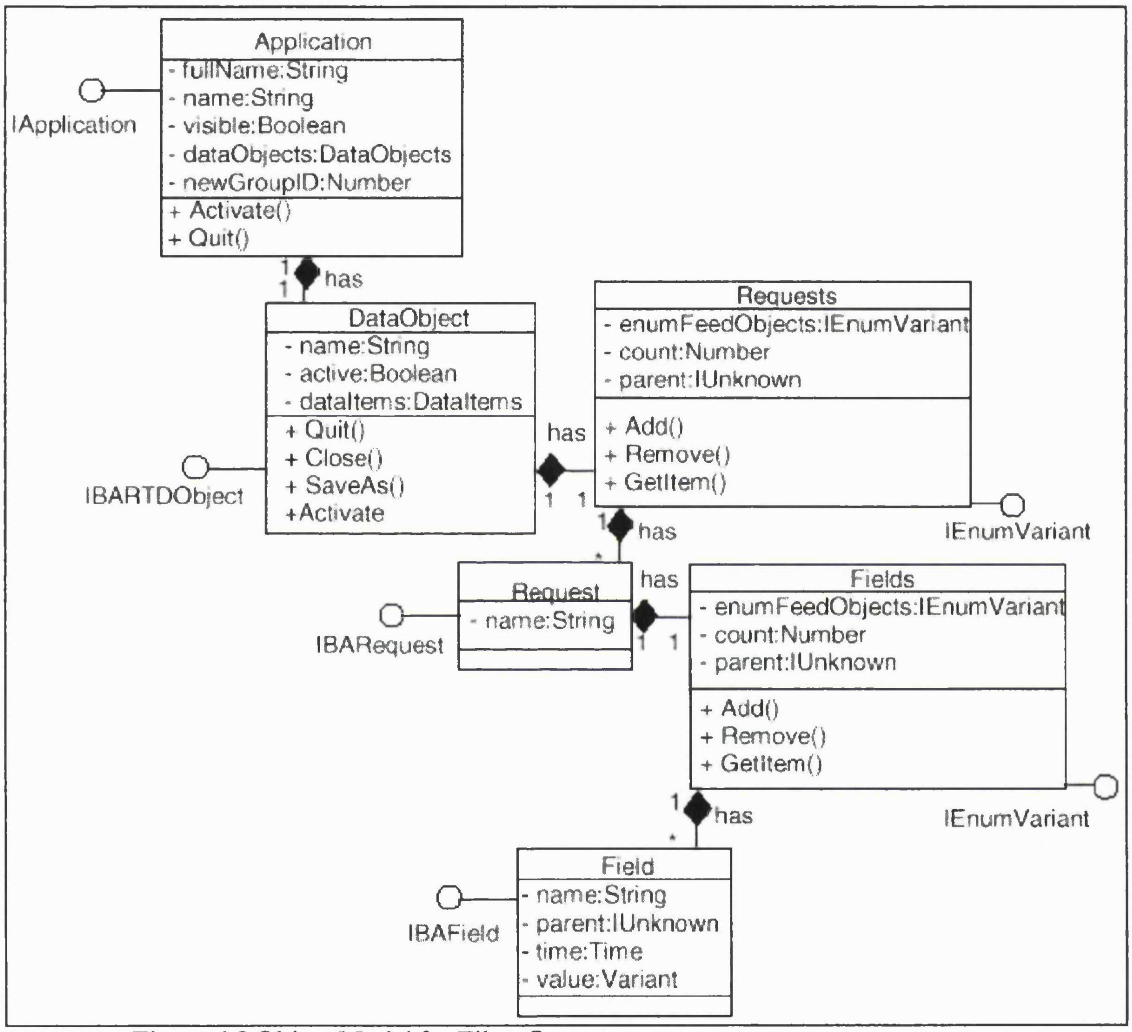


Figure 6-2 Object Model for Filter Component

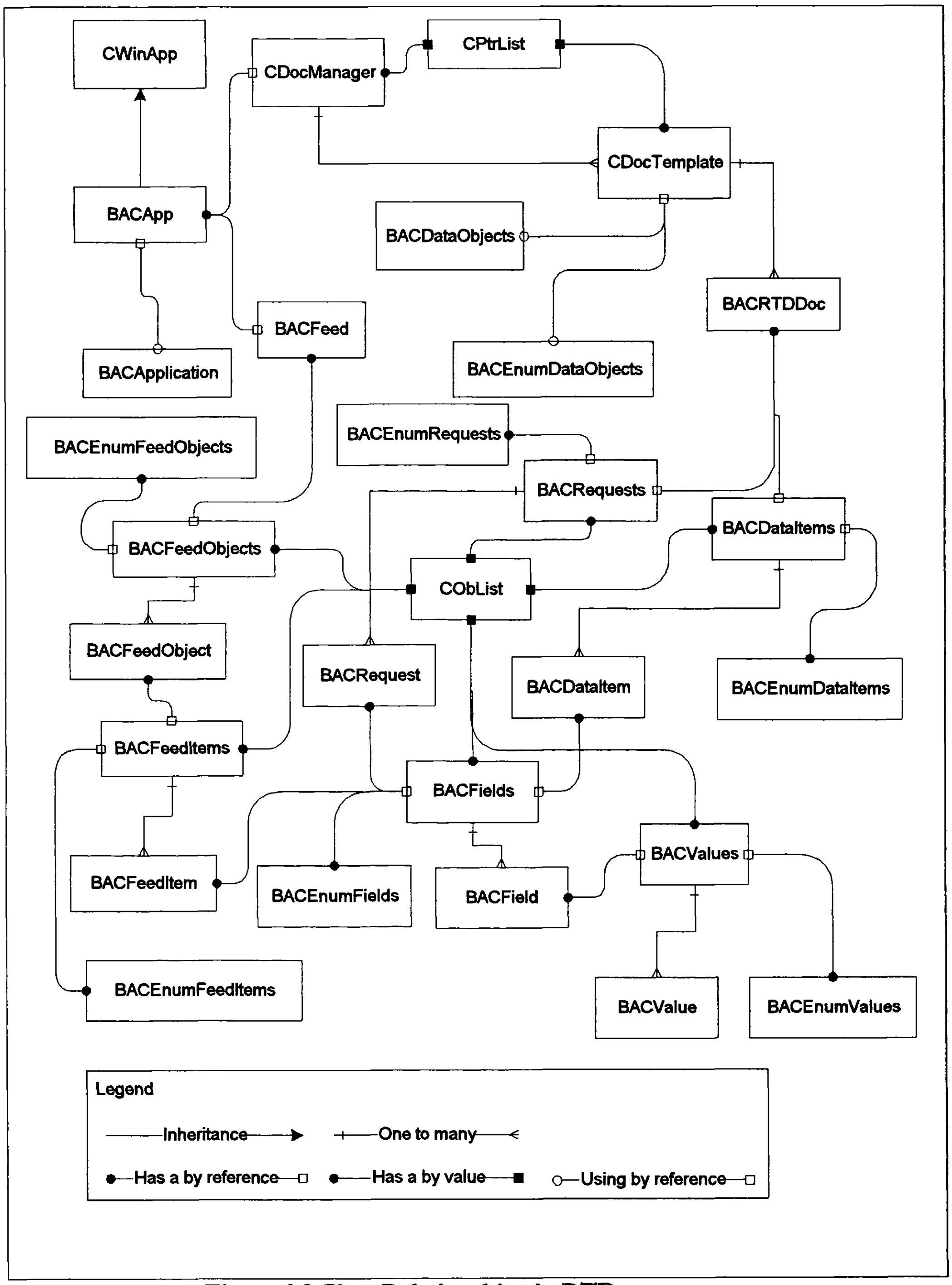


Figure 6-3 Class Relationships in RTD

Figure 6-4 Object Model for Output Component

Data Output

The second event examined is the output of data to a client application. This can occur in one of two ways:

1. The client application is polling for the data and therefore making a request

2. The client application has registered to be notified of any changes to the data.

In the first scenario, the client application will simply make an automation call to the RTD system requesting the value for a field of a data item in a data object.

The second case provides a more interesting problem that requires a hot link to be set up for the data item. COM provides a technology for creating and using hot links, known as Connection Points. The RTD system acts as a COM server presenting each data item as a COM server item, to which clients can connect.

The diagram in Figure 6-4 shows the object model used by the Output component. The structure again reflects the aggregation hierarchy that we exploited also in the data input and filtering components. The Application object acts as the single root from which the hierarchy of available objects can be accessed. The DataObject class will be the same as the one used for filtering. If we request for data in the London Stock Exchange DataObject, it will be the same instance of DataObject as the one used to access the data collected for the London Stock Exchange. The DataItem object represents the actual item that RTD is collecting the data for, so an example again would be British Telecom. This DataItem then contains a collection of Fields, these Field objects are from the same class as the Fields presented in the Input and Filter components. The difference is that now the field can contain a list of values to represent data items at specific times from the feed.

6.6. Data Management

There are two issues with managing the data within the RTD system:

1. Filtering the data, deciding which data inputs to keep and which to discard.

2. Storing the data, the mechanisms for storing the data to keep.

Each of these is examined.

Filtering

The system of requests as suggested in the WOSA/XRT design specification works well in an environment which is predetermined, the user can be provided with a list of what available and make requests from that list. In the situation which is created with a system like this, a feed could plug it's data into the system at run time. Without knowledge of the data provided by this feed, the end user would not be able to request that the data be collected. For this reason a feed list is created by the system each time any feed places any new form of data into the system. This list consists of the data object, which contains data items, which contain fields. From this list a user can request any field for which they wish to collect data.

The purpose of the Filter component is to filter the data that is passed via any of the Input components and to select that subset that users are interested in. The object model for the Filter is shown in Figure 6-2. Again, Application is a singleton that acts as a root from where the hierarchy of filtering related objects can be retrieved. To request that particular types of data be collected by an RTD implementation, a Request object needs to be created within the DataObject.The DataObject represents a collection of Request objects.As an example, a DataObject could represent the London Stock Exchange and a Request could represent British Telecom shares, a particular financial instrument. Each Request object then contains a list of the data fields to be collected. So an example would be to collect the Bid and Offer fields only. There is a similarity that can be noted between the architecture for the filtering and the input. In particular, the Application, Fields and Field class and their respective interfaces are reused.

As this functionality is all provided through an OLE automation interface, there is no reason that a program could not be written to select the fields from the feed list. In this way some AI techniques could be applied to select the data be collected.

Data Storage

It was decided that although the job of data storage could be easily performed by a third party database it would not necessarily be the quickest mechanism, it would also mean that the end user would have to already have access to the database engine provided on their machine. For this reason a proprietary storage mechanism has been provided, along with the option of storing to a third party ODBC compliant database if the user so wishes.

The proprietary storage for the RTD system uses a file to store each data object. This file, or document, stores all of its data items, each data item stores all of its fields and each field stores all of its values. This is accomplished by using the compound file technology, which is Microsoft's implementation of the Structured Storage model examined in the Storage section in the previous chapter.

6.7. Testing Methodologies

One of the big problems with software today is that of quality, obviously one of the aims of this project was to produce high quality software. This begged the question - What is software quality? In 'Quality is Free'[13], a book by Philip Crosby, this is discussed:

> *The problem of quality management is not what people don't know about it. The problem is what they think they do know....*

In this regard, quality has much in common with sex. Everybody is for it. (Under certain circumstances of course.) Everyone feels they understand it. (Even though they wouldn't want to explain it.) Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.) And, of course, most people feel that problems in these areas are caused by other people. (If only they *would take the time to do things right.)*

Although this is quite amusing there is also a lot of truth, in what he says. Too many people seem to think that writing software is like doing simple maths, simply work out the correct answer to a formula and it will all work. Unfortunately it's not that simple, there are many external forces at work with software, there are large number of variables to cope with.

In order to deal with this a strategy for testing the system needed to be developed. The first stage involved testing sections of code as they were written by tracing through the functions, attempting to follow each possible path. Once the entire system had been put together it would be possible to start doing black box testing. This involves creating inputs and testing the output.

White Box Testing

In order to perform a set of white box tests each function needed to be broken down into the possible paths which could be taken through it. It would then be possible using the debugging tools to trace each path, along with the variables contained within the function. For example the following function builds up a list of all the field names in a data object.

```
CStringList* BACRTDDoc::BuildFieldList()
{
        m_lstFields.RemoveAll();
        // go through all the data items
        POSITION pos = NULL;
        for (pos = m_pDataItems->GetHeadPosition();pos!=NULL;)
        {
                BACDataItem* pDataItem = m_pDataItems->GetNext(pos);
                // if we have a valid data item
                if (pDataItem)
                {
                        //get the fields
                        BACFields* pFields = pDataItem->InternalGetFields();

                        if (pFields)
                        {
                                POSITION fieldPos = NULL;
                                // iterate through all the fields
                                for (fieldPos = pFields->GetHeadPosition(); fieldPos!=NULL;)
                                {
                                        BACField* pField = pFields->GetNext(fieldP os);
                                        if (pField)
                                        {
                                                //get the name for each field
                                                CString strFieldName = pField->GetStrName();
                                                // if the name is not already in our list of fields
                                                if (!m_lstFields.Find(strFieldName))
                                                {
                                                        // add the name to the end of our list of fields

                                                        m_lstFields.AddTail(strFieldName);
                                                }//endif (!m_lstFields.Find(strFieldName))
                                        }//endif (pField)
                                }//end for (fieldPos = pFields->GetHeadPosition(); fieldPos!=NULL;)
                        }//end if (pFields)
                }//endif (pDataItem)
        }// end for (pos = m_pDataItems->GetHeadPosition();pos!=NULL;)

        return &m_lstFields;
}
```

This code can be viewed as the flow chart in figure 6-5, this can then be converted into the flow graph shown in figure 6-6.

From the flow graph the cyclomatic complexity can be calculated in one of three ways:

1. The complexity equals the number of regions in the flow graph.
2. Complexity, V(G), for the flow graph G is: V(G) = E – N +2. Where E is the number of edges (lines) in the flow graph, and N is the number of nodes.
3. Complexity, V(G), for the graph G is: V(G) = P +1. Where P is the number of predicate nodes in the graph. A predicate node can be identified as it has more than one line emanating from it.

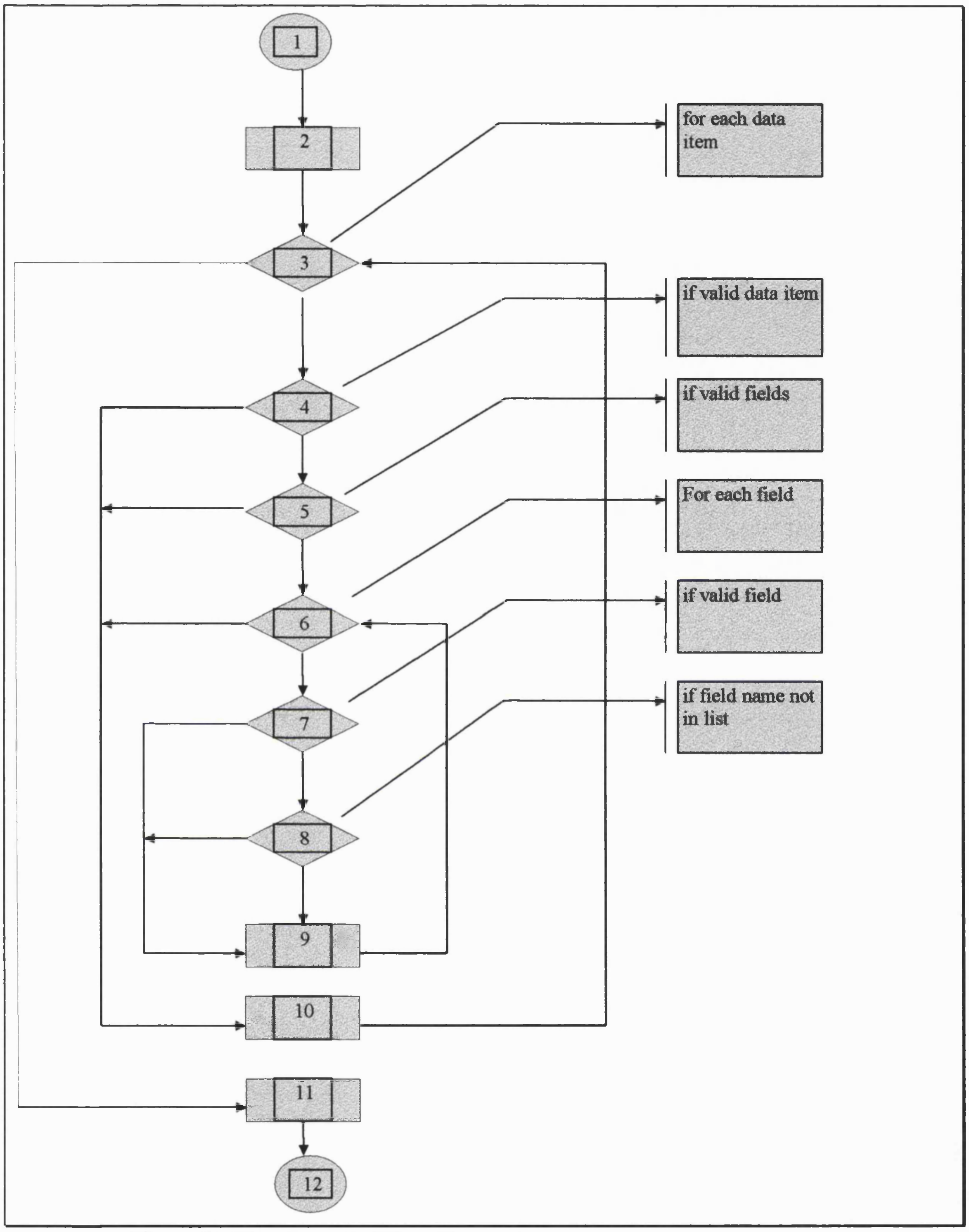


Figure 6-5 Flow Chart for White Box testing

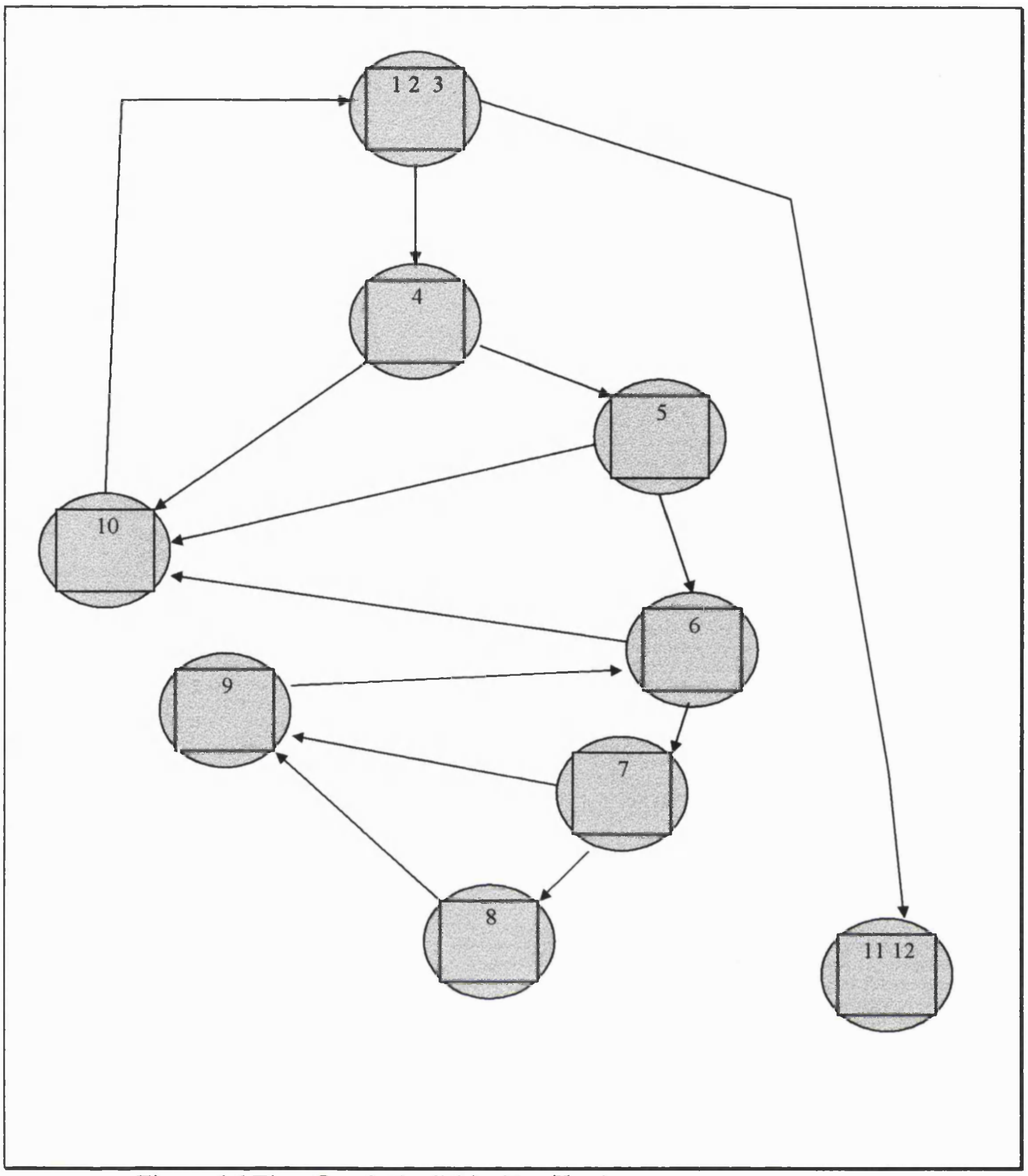


Figure 6-6 Flow Graph For White Box Testing

The simplest is to count the number of regions in the graph. The graph in figure 6-6 has six regions and therefore the complexity is six.

Using the second method there are 13 edges and 9 nodes so the complexity V(G) = 13 – 9 +2 = 6.

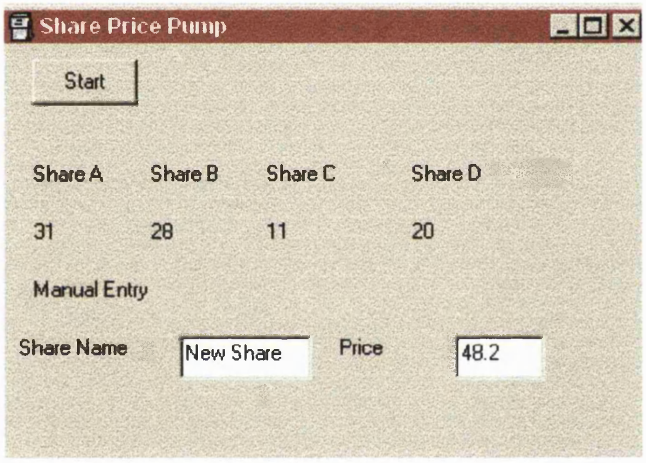
Using the third method V(G) = 5 predicate nodes +1 = 6;

This means that there are six paths that need to be tested.

Black Box Testing

Once the system started to reach completion, it was possible to write test harnesses in order to perform black box testing. The test harnesses had to be simple programs, so as to minimise the likelihood of bugs existing within them. They were written in Visual Basic, this allowed them to be put together quickly and for changes to be made easily.

The first test program to be written was one to pump data into the system. It pumped four random numbers into a set field of four different data items in the same data object. It also provided an interface to place a user-defined number into a user defined data item.

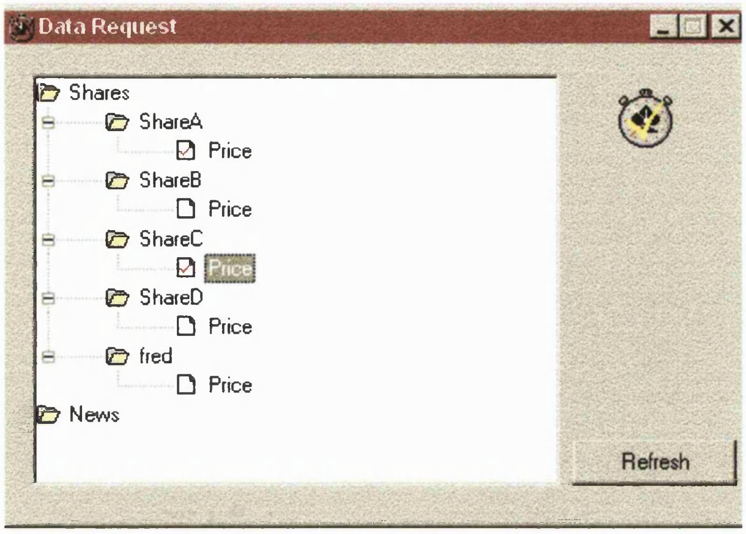


Share A	Share B	Share C	Share D
31	28	11	20

Manual Entry

Share Name: New Share
Price: 48.2

The next program to be written was one that could read the feed list and allow the user to choose which fields from which data items to collect data from.



Once this was achieved it would be possible to create a program which viewed the values being pumped into the system.

Using Real Data

Once the system was capable of performing the basic required functionality it was possible to start using real data. Writing a simple feed handler to take the data from a financial feed provider provided this.

The RTD system has been tested using data from Datastream's Market Eye, Reuters SSL 4 and ISMA's TRAX feeds. Although these are all providers of financial information each provides their data in a different format.

6.8. Conclusion

This concludes the examination of the implementation and testing of the RTD system, a real world solution to handling multiple data feeds in one system. The architecture of the system has built upon the architectures presented in chapters 3 and 4 by providing a single interface to allow filtering engines to attach to the system. It also allows for a range of basic data types to be collected, filtered and output by the system in a timely manner.

The system created from the architecture is capable of taking data from any number of feeds, through an open COM interface. It provides an interface for filtering the data, which allows a third party program to define the rules for filtering. It stores recent data in memory for timely data retrieval and archives historical data to either a database or proprietary file system. The RTD architecture has been licensed to several companies to use as a basis for their own real time systems.

Chapter 7

ASSESSMENT

This chapter provides an assessment of each of the three systems based on the experimental architectures, the Private Investment System, the Vehicle Tracking System and the Generic RTD System. It examines the lessons that were learnt from each of these systems and how they were applied in the next system.

7.1. Introduction

The goal of this thesis has been to create an industrial strength architecture to help solve the problems of information overload by integrating multiple real time data feeds. Throughout this thesis the issues associated with designing architectures to create these solutions have been addressed by exploring 3 architectures and the systems that have been created from each architecture. These systems presented in this thesis are examined in this chapter highlighting the features that were introduced and the drawbacks discovered.

The lessons learnt, both from creation of each system, and provided from end user feedback are examined. An explanation of how these lessons have influenced the next system, along with how these were applied, is covered in this chapter.

7.2. Private Investment System

The system examined in chapter 3 for private investors to monitor the information provided by stock markets. It was designed for 16 bit Windows and can cope with multiple data feeds. The following section describes the strengths and weakness of the architecture for this system.

A uniform mechanism for data entry and data retrieval

The main strength of the architecture for Private Investment System is that it provides a new way of bringing multiple data feeds together to one location by using a 16 bit DLL, which exports an API. This takes advantage of the fact that under 16 bit Windows a DLL is loaded only once and shared between applications. Therefore any data placed into the DLL can be accessed by any application which calls an API the DLL provides. As an experiment it was successful in solving the problem of integrating data from multiple sources into a single collection point.

However a weakness of the architecture is that there still needs to be multiple applications to place the data into the DLLs, any new feed which needs to be integrated has to have some non trivial code written to talk to the API.

A Single Place for Filtering

The Data Director application provides the user with one place to control and filter the feeds. But each feed needs an application to actually perform the filtering and this 'feed application' needs to comply with the proprietary API of the Data Director. This is one of the drawbacks and means that it is fairly difficult to write a feed to place data into the system. Third party data providers have found difficulty in integrating their specific feeds, and generally require input from an experienced developer.

This is overcome in the RTD system by providing a filtering system in the main engine, and so enabling a feed to place it's data in the system with only a few lines of code.

Data Storage for Timely Retrieval

The Private Investment System provides its own hierarchical mechanism for storing 'live' data in memory. This means that it is very fast to retrieve the data, which is required, to deliver to the end user within fixed time frames. It also provides a mechanism for archiving historical data to files on disk. These files have a proprietary format and are not compatible with other systems.

Drawbacks

For commercial reasons the Private Investment System is 16 bit and therefore does not take advantage of threading or multiple processor technologies. This also means that it has a proprietary interface that was not open like COM.

This interface has been designed for a specific task and was therefore created with fixed fields for data entry, these fields had a strong financial bias, meaning it is not possible to use the system for tracking other data. It also means that in order to add a new field to the system the code has to be changed and recompiled.

7.3. Vehicle Tracking System

This section assesses the bus tracking system described in chapter four, pointing out the features it provides along with the benefits and drawbacks.

Multi-Processing

Being a 32 bit system means that it can take advantage of multi-processing. This is done by splitting the task up into components that can run as individual processes. The strength of this architecture, over threading, is if a particular component fails the rest of the system continues to operate. The disadvantage of this is the time the processor takes to context switch between each process. This time will be minimal if there is enough physical memory to hold all the processes code and data.

Global Memory for Data Sharing

This assesses the solution to the problem of transferring data between the processes. The system provided by the Private Investment System could not be used, as it relies on the fact that 16 bit DLLs are loaded once for the whole system. 32 bit DLLs are loaded per process and so another method had to be employed to transfer data.

The fastest mechanism of data transfer between 32 bit processes is through the use of shared memory blocks. This takes advantage of Windows NT's memory mapping as described in chapter 2. Specific libraries were written to manage these memory blocks and lock them into physical memory to prevent them being swapped into a secondary storage device. This system for transferring data proved to be very successful with good feedback times reported by the system users.

Information Overload

This system helps to solve the information overload problem for passengers of buses by providing them with a simple list of buses expected at each stop along with estimated times of arrival. This means the passenger does not have to look at the timetable.

Drawbacks

Like the private investment system the Bus Tracking system was designed to solve a specific problem (tracking vehicles) and can not handle generic data. It has fixed data entry types, and specific interfaces, which are not as open as an OLE interface. There was a good commercial reason for this, providing an open interface carries the overhead of longer development times, and possibly lengthier response times.

7.4. RTD System

The architecture for the RTD system was designed from the beginning to be a general solution to handle multiple real time data inputs. For this reason it has many features that were not required by the previous two systems.

Memory Storage for Timely Data Retrieval

Using memory storage for the real time data provides a fast retrieval time for data, this system that was first applied in the Private Investment System using a DLL to control the storage. This was refined in the Vehicle Tracking System where a global memory block was used. The RTD process controls the real time data, which is locked into its own memory address space and COM interfaces provide access to the data.

Hierarchical Data Storage

The hierarchical data storage system initially designed in the Private Investment System was refined into a much deeper hierarchy for the RTD system. This was also carried through to the archiving provided by the system. A hierarchical folder and file like system helps to alleviate the information overload problem by grouping like data together.

Single Filtering System

The RTD system provides a mechanism for filtering. This filtering system was based around the request system outlined in the WOSA specification [4]. This helps to overcome the information overload problem by providing a single place from which all data can be filtered.

Open Interface

Being based strongly on COM, RTD provided an open interface for both data entry and retrieval. This meant it is very simple to write an application to place data into the system or to read data that the system is holding, as shown in figure 7-1.

```
LgroupID = xrt.NewGroupID

lval = dataFeed.FeedDataIntoGroup("Shares", "ShareA", "Price", CInt(sharePriceA), lGroupID)
```

Figure 7-1 Visual Basic Code to insert price for ShareA

Also the fact that COM was used allowed for a generic data entry type in the form of Variants. By doing this, the system is non-specific, and can be used for multiple purposes, such as share prices, news stories or map co-ordinates.

One of the problems with the Private Investment System was the fact that the data is stored in proprietary files. This meant that third party products cannot easily manipulate the data. The Vehicle Tracking System used an industry standard SQL compliant database which allowed the data to be queried and analysed by other systems. The RTD system provides multiple storage options, including a proprietary mechanism or use of any ODBC compliant Database.

7.5. Lessons Learnt

The storage of data in fixed memory provided fast response times and so this system has been employed and refined in each of the other systems. This was initially used in the Private Investment System, with a single DLL controlling the memory in which the data was stored. The Vehicle Tracking System uses shared global memory blocks, which can be accessed through a DLL. The RTD System uses a single instance COM server application to control the memory within its own address space.

The non standard format for data archiving in the Private Investment System provided some problems for some end users who wished to perform their own analysis on the data. The Vehicle Tracking system uses a standard SQL database for data storage and the RTD system provided the choice between a database or a proprietary file format.

The fixed fields in the Private Investment System provides problems when integrating new feeds. Some feeds provided data in different formats to that expected. This was not a problem for the Vehicle Tracking System, which was a closed system, and therefore any data that a vehicle could provide was known at the requirement stage of the development. The RTD System overcomes this by allowing the feed to provide data in multiple formats and building up the data hierarchy of fields as the data is input.

The Private Investment System requires each feed to provide its own filtering mechanism. This has proved to be a real problem with each feed provider having to rewrite a filtering system to integrate their feed into the system. This is solved in the RTD system by providing a single filtering system within the RTD COM Server. In order to allow for future enhancements the RTD System also provides COM interfaces for the filtering system.

7.6. Summary

Each of these systems has built on the lessons learnt from the previous system. It has been an evolving process to reach the final goal of achieving completion of the RTD system.

This RTD system helps to overcome the issues being addressed in this thesis. It provides a uniform collection point for multiple data feeds and multiple data types. It has a built in filtering system to enable the user to selectively remove data they do not wish to view. It contains a mechanism for data archival, in order to allow for analysis to be performed on historical data. Finally each system has been deployed in the real world to provide feedback from end users and test the architectures applicability to solving real world problems.

The following chapter draws some conclusions from this process and looks at how the system can be taken further.

Chapter

8

CONCLUSIONS AND FUTURE WORK

This chapter draws conclusions from all the work carried out in this thesis. Recommendations are made for areas for future work, based on what has been learnt from this project.

8.1. Introduction

The major contribution of this thesis is the new architectural style presented to address the issue of information overload on personal computers by integrating data from multiple sources. In order to achieve this a series of three architectures have been created and tested, culminating in a final architecture which collects generic data types from multiple feeds, provides a component for filtering the data, and provides the data to many data clients. This chapter looks at the conclusions that have been drawn from the architectures presented as well as looking at further work in this area.

8.2. Conclusions

This chapter concludes this thesis by reviewing the research carried out and identifies the contributions made by each new architecture towards addressing the issues of information overload by handling multiple real time data feeds. As each architecture has been used to produce a system that has been deployed in the real world, the feedback provided from these systems will be utilised to understand the success of each architecture.

Real Time Processing of Data Feeds and Information Overload

Chapter 2 provided an understanding of real time data delivery, along with discussing the mechanisms used for achieving the goals of this thesis. This chapter also provided an insight into the methods employed by existing systems to provide large amounts of data to the user in a timely manner.

Many systems in the financial industry use data feeds for the exchange of information. A data feed can be considered as a continuous stream of data. Data feeds are also used in the transport sector, for example in air traffic control and vehicle tracking systems. Data feeds are produced by one system and processed, filtered, viewed and archived by other systems. An example is the integration of different trading systems for financial products, which feed data to back-office systems where these trade data are processed and subsequent financial transactions are started. There are commercial providers of data feeds, such as Reuters and Bloomberg, which provide subscribers with up-to-date price information about trades that have recently been completed at the stock exchanges.

Following multiple data feeds is too laborious for humans and they are often overwhelmed by the sheer amount of information that is presented to them. The data feeds of Reuters and Bloomberg are good examples. They provide new data items every few seconds, whenever a trade has been completed at the stock exchange. It is impossible for humans to follow several of these feeds over prolonged periods of time. This situation is referred to as information overload.

Many data feeds have to be processed as quickly as possible. Traders at the stock exchange, for example, might miss opportunities if they are not informed about changes in the market as they happen. Hence, many of the information systems used in this setting have real-time response-time requirements. In safety-critical systems, such as cruise control in aircraft or reactor controls in nuclear power plants, real-time constraints are hard and could lead to disasters if the system does not respond in time. The response-time requirements in financial systems are soft in that slow responses do not render the system incorrect, but they would lead to a low acceptance of the system.

All the systems examined in the second chapter cater for a specific proprietary feed and do not permit the addition of a new feed. Neither do they provide an interface to allow the addition of new filtering systems. No existing systems were discovered to provide an open interface, which would allow them to become reusable tools for the integration and filtering of data.

One of the goals of this thesis was to test the architectures invented in a real world environment, in order to provide feedback as to the robustness and performance of the systems created from each architecture. For this reason the second half of chapter 2 examines the issues of processing real time data under the Microsoft Windows environment. Microsoft Windows was chosen because its wide acceptance within industry meant that it would be possible to deploy the systems created in real world applications.

The difference between the 16 bit and 32 bit architectures provided by different versions of Windows were examined. 32 bit Windows was shown to have many more features that would enhance the development of a real time system, such as threading, multiprocessor support, a scheduler and memory management functions. A limited number of available thread priorities and the inability to guarantee a latency period means that it would never be possible to create a hard real time system under the 32 bit Windows architecture. Techniques to enhance the timely delivery of data within 32bit Windows are examined, including using memory mapped files, multiple processes to increase the number of thread priorities, and issues with priority inversion.

The Private Investment System

A new architecture to integrate multiple data feeds was introduced in chapter 3. The development lifecycle of the system based on this new architecture was described. It is a 16 bit Windows application to provide financial stock market information to private investors. The architecture around which it is designed collates data from multiple sources and stores the latest values in memory in order to provide timely retrieval of the data by data clients.

The problem of Information Overload is addressed by providing a single point from which all the feeds input into the system could be controlled and filtered. Multiple Real Time Data Feeds can be used to place data into the system. Each feed requires its own software to place the data into the system by calling the APIs provided. This is achieved by having a set of 16 bit DLLs that export the APIs. In 16 bit Windows, DLLs are shared between applications and so there is only ever one instance of a DLL running, this enables more than one application to place data into the same DLL and allows for multiple applications to read the data.

This applicability of this architecture to solve a real world problem is proved by the fact that this system is currently one of the best selling private investment software packages in the United Kingdom.

The Vehicle Tracking System

The forth chapter introduces a new architecture based on the one presented in chapter three. A system to provide bus passengers with information about the arrival time of their next bus was built around this architecture Information Overload problems are addressed by providing the end uses (bus passengers) with the information which they most readily need, when the next buses are going to arrive. This reduces the need to look up the information from a timetable. This information is based on the current location of the busses approaching the bus stop.

Multiple Real Time Data Feeds provide the locations for each bus on the system. Each bus uses a radio transmitter to send information about its location to a central station, where the information about each bus is collated. In order to provide a single interface to place the data into a single location, a 32 bit DLL was written which uses a shared memory block. Each application can then load its own copy of the DLL and use the interface it provided to access the data in the shared memory block.

One of the issues also addressed by this system was the fact that it provides information to multiple 'clients' (the signs at the bus stops). This system therefore not only acts as a collection point for multiple information sources but also acts as a source.

The effectiveness of this architecture to solve real world problems was demonstrated by collaborating with Hampshire Country Council, and using Winchester as a test site.

The Generic RTD System

The final architecture presented in this thesis builds upon the two architectures presented in chapters 3 and 4. This final architectural style attempts to overcome the drawbacks discovered in the previous two. The software created based upon this architecture is a generic system to collect different formats of data from multiple sources and provide an open interface for filtering, data collection and data retrieval. The development of this system was examined in chapters 5 and 6.

Integrating the filtering into the system reduces information overload and thus, unlike the Private Investment System, provides a single point from which all the data that is entered into the system can be filtered. This single filtering system also provides several COM interfaces to enable other applications to customise the filtering.

Multiple real time data feeds can easily input their data into the system using a simple COM interface, which is provided by the COM server RTD executable. Making this executable a single instance application ensures that the data can all be input to the same collection point. RTD then keeps all the most recent data in memory and archives the historical data.

Creating the RTD system for Black Ace Software Engineering has showed the industrial relevance of this architecture, as this RTD system has since been used as the core technology for more than three financial real time systems.

Summary

This thesis has provided an architecture that has made four contributions to science.

A Uniform Data Collection Point has been created by integrating multiple data feeds through a standard interface and providing a single place from which the data can filtered and sorted.

A Standard Real Time Engine has been proposed. By using techniques to overcome the limitations of the Microsoft Windows operating system, such as memory mapped files and locking data into physical memory, it has been possible to create a system in which the data that is collected can be real time. By providing an architecture system for notifying data clients of changes to the data, the systems created can cope with data that is constantly changing.

Resolution of Information Overload has been achieved to a certain extent by providing an architecture that can collate information from multiple sources and exposes an interface to allow the filtering of all the data that passes into the system.

The Industry Aspect of creating an architecture that would solve real world needs has been addressed by building systems based on the proposed architectures that have been tested in a commercial environment. This has ensured that the architectures complied with industry standards and would gain commercial acceptance, along with providing end user feedback on each system deployed in order to more fully understand the requirements of the architecture provided.

8.3. Further Work

The architecture presented in this thesis and the work carried out has lead to the formation of a company to specifically provide real time data solutions to primarily the financial markets. This section describes some of the further work which has already been undertaken, along with exploring the possible directions of work that may be taken in order to further enhance this architecture.

RTD has been further instantiated in the Black Ace Software Engineering product named BASE Market Monitor. This product tracks share prices and news from financial markets. Feeds have been written to accept data from Datastreams Market Eye, Teletext and World Wide Web pages. Features of RTD have also been used within Cognitech's Market Surveillance System and also more recently in the Visual Global Markets product. These products use more 'heavy-weight' feeds from ISMA and Reuters respectively. The BASE Market Monitor product is aimed at smaller investors and home users. The feeds that it connects to provides either soft real time or time delayed data. This product runs on a single machine and the RTD engine is a single instance application. Many clients packages then talk to this engine in order to provide different views of the data to the user, such as tickers, charts and price screens. The architectural style holds up well to this model. If a single client fails, it does not bring down any other clients or the RTD engine, as they each sit in their own protected process spaces. The response times and therefore performance of the system depends heavily on the activity of the feeds. When the market is busy, the load increases and the response time drops. Actual timing measurements have not been taken but existing users of the system seem content with the performance, and the author is the first to know when there is a problem!

The Market Surveillance System required a 'harder' real time system. The surveillance team in a stock exchange needs to know within seconds if market makers or traders are not adhering to the rules of the market place. Initially this system was written with the European Exchange EASDAQ where ISMA provide the TRAX data feed. By collecting the data on a server machine the RTD system was extended to provide many client machines with the real time data. The classical client-server model has been used for this system and it is currently being used by the surveillance team within EASDAQ. Cognitech then decided to take the architecture further in the Visual Global Markets product. This product takes full advantage of the DCOM model and is based on a three tier architecture. The data provided by Reuters enters a server machine, where it is also archived. This database server advises a second server of data changes. This second server has a set of complex filtering and calculating components. These middle layer components apply the business rules to the Reuters data. Several client machines then can be advised of information that is of use to the end user. To date two different versions of the client have been written. One which is a fully functional program for filtering, sorting and editing the data, and the other which is an add-in to Microsoft's Excel application, where the data can be further manipulated. It is this ability to link to such an industry standard as Excel that makes the RTD architecture so powerful.

Currently there are three key areas in which more work is being carried out on enhancing the RTD system. They are:

[1] Integrating RTD into more real world systems;

[2] Enhancing the filtering mechanisms;

[3] Providing more front-end tools for handling the data.

Each is examined in turn.

Until RTD gets integrated into more real world solutions it will not be possible to evaluate its success as a generic architectural style. For this reason it is now being licensed to various companies, as the back-end engine for their real time data needs. It will be essential to integrate this system into more real world solutions in order to prove its worth. One of the important features of RTD is the fact that it has a built in filtering mechanism, with an open interface. By enhancing this system to provide more intelligent filtering, the RTD system will become a more attractive tool to use. One of the areas in which the filtering could be improved would be to have an intelligent system of searching for specific key words in the incoming data. Thirdly, it will become necessary as more systems use the RTD system, to provide some more front-end tools for manipulating the data. This would enable both developers and system administrators to have a better view of how the system works, and therefore how to best use it.

8.4. Summary

The aim of this thesis was to create and present an architecture to solve the problem of information overload on personal computers by integrating multiple real time data feeds. This has been done through creating a series of three experimental architectures each of which built upon the previous one. In order to gain a greater insight into the worth of each architecture a system has been written and deployed into the real world based upon the architecture The final architecture presented (RTD) is an architectural style that can be used to solve multiple needs. RTD has been deployed three times in financial systems. RTD is now being evaluated to be used to solve a diverse range of problems, from geographical tracking, through financial data filtering, to handling real time audio inputs. One of the largest barriers to overcome was the fact that Windows is not a real time operating system and so does not provide the support required of a 'hard' real time system. But, as this thesis has shown, it is possible to write a 'soft' real time system, which has good performance, to run on the Windows platform.

Glossary

ActiveX	COM based technology
Agent	An object that can operate upon other objects. Usually in some autonomous manner.
API	Application Programming Interface
Class	A set of objects that share a common structure and functionality.
COM	Component Object Model – Microsoft's object technology
COM Interface	A collection of methods and properties exposed by a COM object.
DCOM	Distributed Component Object Model
DDE	Dynamic Data Exchange
DLL	Dynamic Link Library
EXE	Executable
Interface	The outside view of an object or class, emphasising its abstraction.
MFC	Microsoft Foundation Classes – a Windows class library
OLE	Object Linking and Embedding - COM based technology for sharing data.
Real time system	A system whose essential processes must meet certain time critical deadlines.
RTD	'Real Time Data' – the generic system created in chapters 5 and 6 for Black Ace Software Engineering Ltd.
RTOS	Real time operating system
WOSA	Windows Open Services Architecture
XRT	Extensions for Real Time

Appendix 1

THE LIVE DATA DLL API

Below each exported function is listed along with a brief description of its functionality.

CreateNewList

This creates a new root list of folders and disposes of any existing folders and there contents which may be currently stored in the DLL.

SaveFile

The SaveFile function saves the current root and any folders and files within the folders to a file with the name given in the szFilename parameter.

LoadFile

The LoadFile function loads the folders and datafiles from a file, given by the szFilename parameter.

GetNumberOfFolders

The GetNumberOffFolders gets the number of folders in the root.

GetNameOfNthFolder

The GetNameOfNthFolder retrieves the name of the nth folder in the roots list of folders.

GetOpenFolder

The GetOpenFolder function gets the number and name of the folder which is currently opened in the library.

GetFolderByName

The GetFolderByName function gets the a folder number from the name of the folder. The number is placed into the address pointed to by pnFolderNumber. This function can be used to check if a folder of that name exists by passing NULL in as the foldernumber pointer.

OpenNthFolder

The OpenNthFolder function opens the nth folder in the roots folder list.

CreateNewFolder

The CreateNewFolder function creates a new folder in the root folder list, of the name pszFolderName. The number of the new folder is placed in pnFolderNumber.

DeleteNthFolder

The DeleteNthFolder function deletes the nth folder from the root folder list.

GetNumberOfItems

The GetNumberOfItems function will get the number of items in the folder which is currently open.

GetNameOfNthItem

The GetNameOfNthItem function will get the name of the nth item in the folder which is currently open.

DeleteNthItem

The DeleteNthItem function will delete the nth item in the folder which is currently open.

CreateNewItem

The CreateNewItem function will create a new item in the folder which is currently open

GetItemByName

The GetItemByName function will get an item number in the folder which is currently open from a name given.

OpenItemByNames

The OpenItemByNames function will find an item from folder name and item name and fill a structure with the item data.

CloseItemByNames

The CloseItemByNames function will find an item from the names given and copy the data in the structure into the items data.

GetItemByNames

The GetItemByNames function will get a pointer to a Live Data object from the item and folder names given.

GetTickCountForItem

The GetTickCountForItem function will get the number of ticks for the item given by its name and folder name.

AddTickToItem

The AddTickToItem function will add a new tick to the item given by name and folder name.

GetNthTickInItem

The GetNthTickInItem get the data in the nth tick for the item given by its name and folder name.

ArchiveDataToFiles

The ArchiveDataToFiles function will store the Live Data Objects into their respective history files. If bClearData set to TRUE then the tempdata items will also be cleared of all current data.

Appendix 2

PRICE FILE DLL API

Below each exported function is listed along with a brief description of its functionality.

GetFolderFromFileName

The GetFolderFromFileName function gets the name of the system folder from the filename of the price file, this only works if the filename contains the path of the file.

CreateNewList

The CreateNewList function creates a new empty list of loaded price files.

SaveFile

The SaveFile function saves the file called szFilename. The file still remains loaded in memory.

SaveFileAs

The SaveFileAs function saves the file called szFilename as szNewName. The file still remains loaded in memory as szNewName.

LoadFile

The LoadFile function loads a file called szFilename into the file list

CloseFile

The CloseFile function closes the file called szFilename freeing the memory.

AddTempdataToFile

The AddTempdataToFile function adds a Live Data Object from the Live Data DLL (if there is any) to the end of the file of the given name.

NewFile

The NewFile function creates a file called szFilename and places it in the file list.

GetNumberOfFiles

The GetNumberOfFiles function gets the number of files currently loaded in the root list.

GetNameOfNthFile

The GetNameOfNthFile function gets the name of the nth file in the list of currently loaded files.

GetFileByName

The GetFileByName function gets a file number from a name. The number represents its position in the list of loaded files.

AddNewTmpItemToNthFile

The AddNewTmplItemToNthFile function adds a new Live Data Object to the nth loaded file in the list using data from a dialog box which is displayed from this DLL.

AddTmpItemToNthFile

The AddTmplItemToNthFile function adds a Live Data Object to the nth loaded file in the list.

GetNthTmpItemFromNthFile

The GetNthTmplItemFromNthFile function gets the nth Live Data Object from the nth file.

GetNumberOfItemsInNthFile

The GetNumberOfItemsInNthFile function gets the number of Live data objects in the nth file in the list of loaded files.

RemoveNthTmpItemFromNthFile

The RemoveNthTmplItemFromNthFile function removes the nth Live Data Object from the nth file.
NOTE: this does delete the item.

UnlinkNthTmpItemFromNthFile

The UnlinkNthTmplItemFromNthFile function removes the nth Live Data Objectfrom the nth file.
NOTE: this doesn't delete the item.

RemoveTicksFromNthFile

The RemoveTicksFromNthFile function removes all ticks from the nth file except for those in the the last nLeaveLast Live Data Objects.
NOTE: this does delete the ticks for good

MergeFiles

The MergeFiles function merges the contents of file2 into file1.

ValidateNthFile

The ValidateNthFile function validates the data in the nth file.

SplitPremiumNthFile

The SplitPremiumNthFile function adds fPremium to all prices before wBeforeDate in the file given by nFileNumber.

SplitBonusNthFile

The SplitBonusNthFile function multiplies fRatio to all prices before wBeforeDate in the file given by nFileNumber.

SplitAutoNthFile

The SplitAutoNthFile function multiplies (price after wBeforeDate / price on wBeforeDate) to all prices before wBeforeDate in the file given by nFileNumber.

GetNthTmpStructFromNthFile

The GetNthTmpStructFromNthFile function get a tempdata structure for the nItemNumber'th item in the nFileNumber'th file.

.

SetNthTmpItemInNthFileFromStruct

The SetNthTmpItemInNthFileFromStruct function sets the values in the nth tempdata item in the nth file to the same as those passed in the structure.

.

GetFileStructFromNthFile

The GetFileStructFromNthFile function gets a File structure from the nth file.

SetFileStructInNthFile

The GetFileStructFromNthFile function copies the data in a File structure into the nth file.

GetNthPriceLRecFromNthFile

The GetNthPriceLRecFromNthFile function fills a PriceLRec structure from the nItemNumber'th item in the nFileNumber'th file.

.

GetFirstNPriceLRecsFromNthFile

The GetFirstNPriceLRecsFromNthFile function gets the first nItems of Live Data and copies data from them into PriceLRec structure for the nFileNumber'th file.

Appendix 3

THE TEMPDATA OBJECTS

The tempdata items are stored in objects of the class UD_TMP_Item as shown below.

```
class UD_TMP_Item : public CObject
{
CTime   m_currentTime;
CTime   m_lastTime;

CObList m_tickList;

CString szName1;                    // the priceline file name
CString szName2;                    // 2nd name
CString szName3;                    // 3rd name
CString szName4;                    // 4th name
CString szName5;                    // 5th name
CString szName6;                    // 6th name

CString szFolder;               // Folder to store tempdata item in
CString szSource;                   // source of data

CStringList m_movieList;  // a list of conected movies
CStringList m_newsList;         // a list of conected news items

float fLast; // last trading day's price
float fLow; // Low for the day
float fHigh; // High for the day
float fCurrent; // Latest price
float fOpen; // open price for the day
float fBid; // Best bid price
float fAsk; // best ask price
float fOpenInterest; // Open Interest
float fAlpha; // Alpha volatility
float fBeta; // Beta volatility
float fStopLoss; // StopLoss value
WORD wVolume; // not used
WORD wDate; // date of current price
WORD wLastDate; // date of last price
WORD wFlags; // display format bit flags
WORD wDataSource; // identifies the data source type (eg. teletext, mkteye)

DWORD dwTime; // time field
DWORD dwSysTime; // system time field

BYTE bLinked; // is it dde linked

UD_TMP_Item* udTmpNext; // pointer to the next tempdata item
UD_TMP_Item* udTmpPrev; // pointer to the previous tempdata item

protected:
```

```
DECLARE_SERIAL(UD_TMP_Item)
public:

// constructors and destructor
UD_TMP_Item();
UD_TMP_Item(const char* szName,  const char* szFold);

~UD_TMP_Item();
// set and get item values
BOOL SetName1(const char *szName);
const char* GetName1();

BOOL SetName2(const char *szName);
const char* GetName2();

BOOLSetName3(const char *szName);
const char* GetName3();

BOOL SetName4(const char *szName);
const char* GetName4();

BOOL SetName5(const char *szName);
const char* GetName5();

BOOL SetName6(const char *szName);
const char* GetName6();

BOOL SetFolder(const char *szName);
const char* GetFolder();

CStringList* GetMovieList();
CStringList* GetNewsList();

BOOL SetSource(const char *szName);
const char* GetSource();

BOOL SetLast(float fVal);
float GetLast();

BOOL SetLow(float fVal);
float GetLow();

BOOL SetHigh(float fVal);
float GetHigh();

BOOL SetCurrent(float fVal);
float GetCurrent();

BOOL SetOpen(float fVal);
float GetOpen();

BOOL SetBid(float fVal);
float GetBid();

BOOL SetAsk(float fVal);
float GetAsk();

BOOL SetOpenInterest(float fVal);
float GetOpenInterest();
```

```
BOOL SetAlpha(float fVal);
float GetAlpha();

BOOL SetBeta(float fVal);
float GetBeta();

BOOL SetStopLoss(float fVal);
float GetStopLoss();

BOOL SetVolume(WORD wVal);
WORD GetVolume();
BOOL SetDate(WORD wVal);
WORD GetDate();

BOOL SetLastDate(WORD wVal);
WORD GetLastDate();

BOOL SetFlags(WORD wVal);
WORD GetFlags();

BOOL SetDataSource(WORD wVal);
WORD GetDataSource();

BOOL SetTime(DWORD wVal);
DWORD GetTime();

BOOL SetSysTime(DWORD wVal);
DWORD GetSysTime();

BOOL SetLinked(BYTE bVal);
BYTE GetLinked();

BOOL SetNext(UD_TMP_Item* udTmpNext);
UD_TMP_Item* GetNext();
BOOL SetPrev(UD_TMP_Item* udTmpPrev);
UD_TMP_Item* GetPrev();

// validate the data
BOOL Validate();
// clear out all the data
BOOL ClearData();

// save and load the data in the item
virtual void Serialize(CArchive& ar);

// tick data member functions
BOOL AddTick(float fPrice, DWORD dwTime);
BOOL DeleteNthTick(int nTickNumber);
int  GetTickCount();
UD_TICK_Item* GetNthTick(int nTickNumber);

// overload the assignment operator
virtual void operator=(UD_TMP_Item& src);
};
```

Appendix 4

THE PRICEFILE OBJECTS

The pricefiles are stored in objects of the class UD_PRICE_File as shown below.

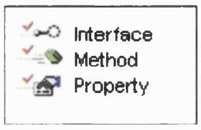
```
class UD_PRICE_File:public CObject
{
CString szFolderName;
CString szFileName;

CString szName1;
CString szName2;
CString szName3;
CString szName4;
CString szName5;
CString szName6;

WORD wStartDate; // date of 1st record
WORD wEndDate; // date of 1st record
WORD wFormat; //no of dec. points on in the prices mode

CObList m_contents; // list of tempdata items
CObList m_charts; // chart objects for this price file

protected:
DECLARE_SERIAL(UD_PRICE_File)

public:
// contructors
UD_PRICE_File(const char* szFolder, const char* szName);
UD_PRICE_File();
//destructor
~UD_PRICE_File();

// retrieve items from list
UD_TMP_Item* GetFirstItem();
```

```
UD_TMP_Item* GetNthItem(int n);
UD_TMP_Item* NewItem(const char* sz_name);
UD_TMP_Item* GetItemByDate(WORD wDate);
// add an item
BOOL AddItem(UD_TMP_Item* pTmpItem);
// remove item, this will delete it
BOOL RemoveNthItem(int n);
                                   // unlink item, this just takes out of the list but doesn't delete it
BOOL UnlinkNthItem(int n);
// remove ticks from the temp items in the file
// leaving the last n temp items alone
BOOL RemoveTicks(int nLeaveLast);
// get the number of items in the list
int GetItemCount();
// get the start and end dates
WORD GetStartDate();
WORD GetEndDate();
// get the names
const char* GetName1();
const char* GetName2();
const char* GetName3();
const char* GetName4();
const char* GetName5();
const char* GetName6();

// get the folder name
const char* GetFolderName();
// get the file name
const char* GetFileName();
// rename the file
BOOL RenameFile(const char* pszNewName);


// set the names
BOOL SetName1(const char* szName);
BOOL SetName2(const char* szName);
BOOL SetName3(const char* szName);
BOOL SetName4(const char* szName);
BOOL SetName5(const char* szName);
BOOL SetName6(const char* szName);
// set the folder name
BOOL SetFolderName(const char* szName);
// Set the file name
BOOL SetFileName(const char* szName);
// set the start and end dates
BOOL SetStartDate(WORD wDate);
BOOL SetEndDate(WORD wDate);

// validate all the data in the file
BOOL Validate();
// save and load the data in the item
virtual void Serialize(CArchive& ar);

// file utilities
// merge another file into this file
BOOL MergeIn(UD_PRICE_File* pSourceFile);
// add a premium to all prices before a certain date
BOOL SplitPremium(float fPremium, WORD wBeforeDate);
// multiply a ratio by all prices before a certain date
BOOL SplitBonus(float fRatio, WORD wBeforeDate);
                                  // multiply to all prices before wBeforeDate price after date / price on date
BOOL SplitAuto( WORD wBeforeDate);
// sort the file into date order
BOOL Sort();

// chart object functions
CObList* GetChartList();
int GetChartCount();
BOOL RemoveNthChart(int n);
UD_ChartItem* GetFirstChart();
UD_ChartItem* GetNthChart(int n);
UD_ChartItem* NewChart();

};
```

Appendix 5

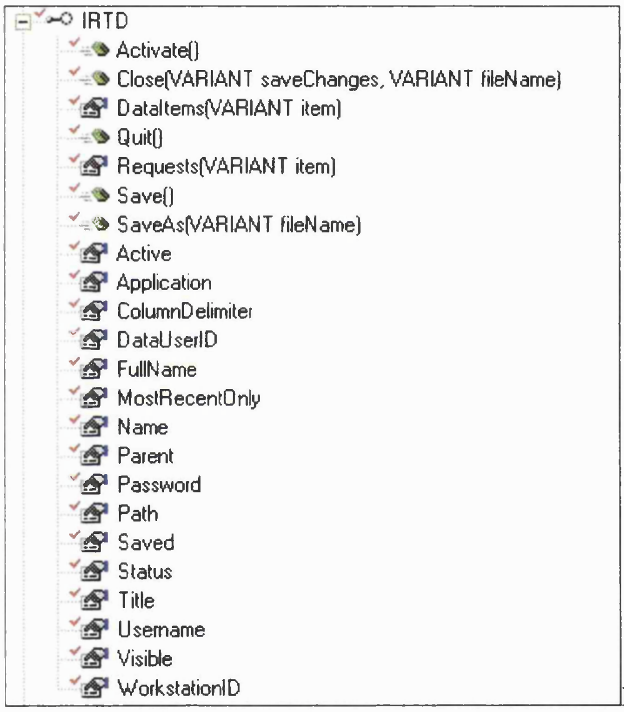
GENERIC SYSTEM INTERFACES

The interfaces provided by the RTD system described in Chapters 5 and 6 are shown in the diagrams below. Using the following key:



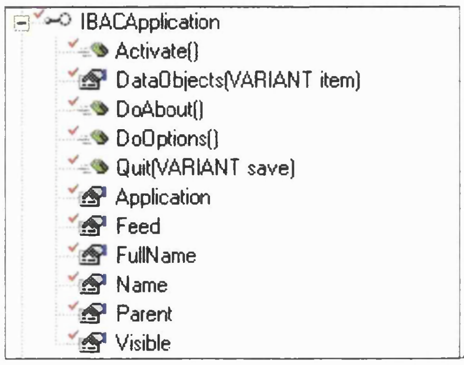
Interface
Method
Property

Key for Interfaces Diagrams



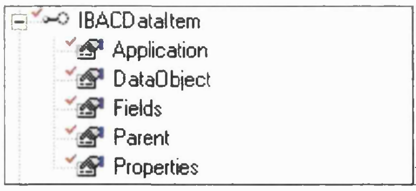
```
IRTD
    Activate()
    Close(VARIANT saveChanges, VARIANT fileName)
    DataItems(VARIANT item)
    Quit()
    Requests(VARIANT item)
    Save()
    SaveAs(VARIANT fileName)
    Active
    Application
    ColumnDelimiter
    DataUserID
    FullName
    MostRecentOnly
    Name
    Parent
    Password
    Path
    Saved
    Status
    Title
    Username
    Visible
    WorkstationID
```

RTD Interface



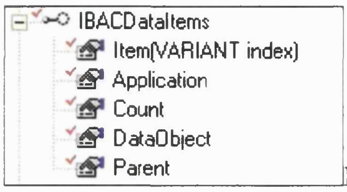
```
IBACApplication
    Activate()
    DataObjects(VARIANT item)
    DoAbout()
    DoOptions()
    Quit(VARIANT save)
    Application
    Feed
    FullName
    Name
    Parent
    Visible
```

Application Interface



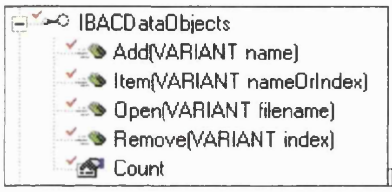
```
IBACDataItem
    Application
    DataObject
    Fields
    Parent
    Properties
```

Data Item Interface



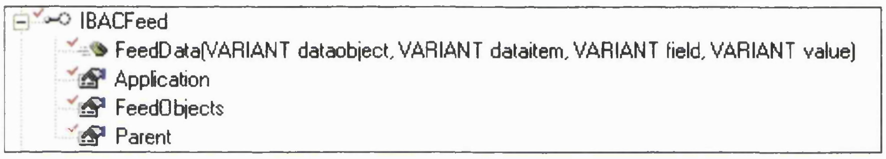
```
IBACDataItems
    Item(VARIANT index)
    Application
    Count
    DataObject
    Parent
```

Data Items Interface



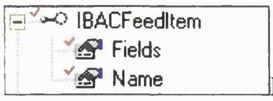
```
IBACDataObjects
    Add(VARIANT name)
    Item(VARIANT nameOrIndex)
    Open(VARIANT filename)
    Remove(VARIANT index)
    Count
```

Data Objects Interface



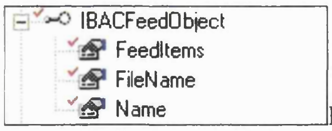
```
IBACFeed
    FeedData(VARIANT dataobject, VARIANT dataitem, VARIANT field, VARIANT value)
    Application
    FeedObjects
    Parent
```

Feed Interface



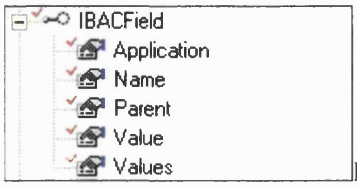
```
IBACFeedItem
    Fields
    Name
```

Feed Item Interface



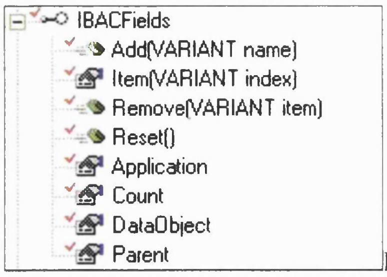
```
IBACFeedObject
    FeedItems
    FileName
    Name
```

Feed Object Interface



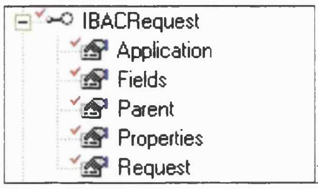
```
IBACField
    Application
    Name
    Parent
    Value
    Values
```

Field Interface



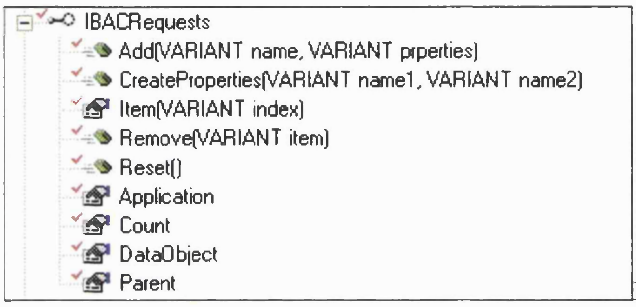
```
IBACFields
    Add(VARIANT name)
    Item(VARIANT index)
    Remove(VARIANT item)
    Reset()
    Application
    Count
    DataObject
    Parent
```

Fields Interface



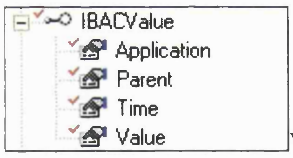
```
IBACRequest
    Application
    Fields
    Parent
    Properties
    Request
```

Request Interface



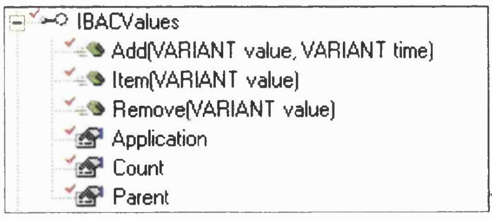
```
IBACRequests
    Add(VARIANT name, VARIANT prperties)
    CreateProperties(VARIANT name1, VARIANT name2)
    Item(VARIANT index)
    Remove(VARIANT item)
    Reset()
    Application
    Count
    DataObject
    Parent
```

Requests Interface



```
IBACValue
    Application
    Parent
    Time
    Value
```

Value Interface



```
IBACValues
    Add(VARIANT value, VARIANT time)
    Item(VARIANT value)
    Remove(VARIANT value)
    Application
    Count
    Parent
```

Values Interface

Appendix 6

DEVELOPERS RULES

This document describes how every developers machine should be set up for working with the BASE RTD project. It also provides coding conventions that should be adhered to. This is NOT a guideline it is the LAW.

Connecting to the Server

The server being used for this project is the BASE Server administered by Neil. You can use the Dial Up Networking provided by Windows 95 in order to connect to it.

The server is called BASE SERVER and provides each user with 3 drives which they can connect to:

Vss	Visual Source Safe
BASE	BASE project documents and source
<username>	a home directory, scratch pad area

This server will be available on 0181 343 4089.

The server is a Windows NT machine with Remote Access Server running on it.

Do not use software compression, Microsoft says that there might be problems with it!

The protocol you should use to connect is NetBEUI.

Drives

Each machine must have the following drive mappings. Where the actual files are stored doesn't matter and is up to each individual developer.

Description	Drive Letter
documents	U:

Local source	P:
MSDEV	M:

Network drives

Each machine should have the following network drives for connecting to the server.

UNC	**Drive Letter**
\\BASE SERVER\Vss	V:
\\BASE SERVER\BASE	T:
\\BASE SERVER\<Username>	R:

The best way that I have found to do this is to have a couple of batch files which will set up this environment for you and then one which removes the environment. Below are some examples of how you could do this in batch files.

InitAoP.bat

```
subst Z: /D
subst Z: d:\projects\AoP\Docs

subst y: /D
subst y: d:\projects\AoP\Database

subst x: /D
subst x: d:\DXSDK

subst w: /D
subst w: d:\projects\AoP\Source

subst v: /D
subst v: d:\projects\AoP\Media

subst U: /D
subst u: d:\msdev
```

NetInitAoP.bat

```
net use s: /DELETE /YES
net use s: "\\BASE SERVER\Vss"

net use t: /DELETE /YES
net use t: "\\BASE SERVER\AoP"
```

AoPClean.bat

```
subst Z: /D
subst y: /D
subst x: /D
subst w: /D
subst v: /D
subst U: /D

net use s: /DELETE /YES
net use t: /DELETE /YES
```

Version Control

In order to keep track of the source files being created and edited, we are going to use Visual Source Safe.

Getting Visual Source Safe Running

This you should find in your V: drive. The source safe data will be found on your T: drive in the source safe directory. When you run the Visual Source safe Explorer, you will need to point it to this directory. You can do this by running it with the /s command line switch followed by the T:\ drive:

```
V:\win32\SSEXP.EXE /ST:\
```

If you have a client license you can install the client version of Visual Source Safe from the V: drive by running Netsetup. You can then use the Exe on your local machine, but you will still have to point it to the data on the network using the /S command switch:

```
D:\Tools\vss\WIN32\SSEXP.EXE /ST:\
```

Using Visual Source Safe

When you run Visual Source Safe for the first time you'll need to set up a working directory for each of the projects. The source code working directories will all be on your P: drive and should have the same name as the projects.

In order to have a local copy of files on your hard disk you should 'get' all the projects from Visual Source Safe. Do not use 'check out' unless you want to make changes to a file.

I have looked briefly at how Visual Source Safe is integrated into the MSDev enironment and it doesn't seem to provide a way to specify where the Visual Source Safe data files are located. Until this is solved don't try to check file's in or out using the MSDev environment.

Code Conventions

As much as possible stick to the Hungarian notation as used by most Windows programs nowadays, also try to use the following guidelines :

1. Class names should begin with BAC
2. Structure names should begin BAS
3. Exported 'C' style functions should begin BAX
4. COM Interfaces should begin IBA
5. Member variables of classes and structures should start with m_
6. All variables should have meaningful names - NOT myVar
7. Try to make comments meaningful explaining why things are being done

Resource ID's

Try to stick to the following prefixes:

Prefix	**Type of symbol**	**Example**
IDR_	Identification shared by multiple resources of different types	IDR_MAINFRAME
IDD_	Dialog resource	IDD_SPELL_CHECK
HIDD_	Dialog resource Help context	HIDD_SPELL_CHECK
IDB_	Bitmap Resource	IDB_LOGO
IDC_	Cursor Resource	IDC_PENCIL
IDI_	Icon Resource	IDI_NOTEPAD
ID_	Command from menu item or toolbar button	ID_TOOLS_SPELLING
HID_	Command Help context	HID_TOOLS_SPELLING
IDP_	Message Box prompt	IDP_INVALID_PARTNO
HIDP_	Message box Help context	HIDP_INVALID_PARTNO
IDS_	String Resource	IDS_COPYRIGHT
IDC_	Control within Dialog box	IDC_RECALC

Variable Prefix Naming Conventions

Prefix	**Type**
ch	char
b	BOOL
n	int
n	UINT
w	WORD
l	LONG
dw	DWORD
p	*
lp	FAR*
Lpsz	LPSTR
Lpsz	LPCSTR
H	handle
Lpfn	callback

OLE Naming

OLE provides guidelines for naming it's objects, collections and enumerators. These can be found in the OLE2 Programmers Reference and in the OLE Automation Programmer's Reference both by Microsoft Press.

Choose names for exposed objects, properties, and methods that can be easily understood by the users of your application. The guidelines in this section apply to all the items you expose:

- Objects (implemented as classes in your application)
- Properties and methods (implemented as members of a class)
- Named arguments (implemented as named parameters in a member function)
- Constants and enumeration's (implemented as settings for properties and methods)

Use entire words or syllables

It is easier for users to remember complete words than to remember whether you abbreviated Window as Wind, Wn, or Wnd.

When you need to abbreviate because an identifier would be too long, try to use complete initial syllables. For example, use AltExpEval instead of AlternateExpressionEvaluation.

Use	**Don't use**
Application	App
Window	Wnd

Use mixed case

All identifiers should use mixed case, rather than underscores, to separate words.

Use	**Don't use**
ShortcutMenus	Shortcut_Menus, Shortcutmenus, SHORTCUTMENUS, SHORTCUT_MENUS
BasedOn	basedOn

Use the same word you use in the interface

Use consistant terminology; don't use names like HWND that are based on Hungarian notation. Try to use the same word your users would use to describe a concept.

Use	**Don't use**
Name	Lbl

Use the correct plural for the class name

Collection classes should use the correct plural for the class name. For example, if you have a class named Axis, you should store the collection of Axis objects in an Axes class. Similarly, a collection of Vertex objects is stored in a Vertices class. In cases where English uses the same word for the plural, append the word "Collection."

Use	**Don't use**
Axes	Axiss
SeriesCollection	CollectionSeries
Windows	ColWindow

Using plurals rather than inventing new names for collections reduces the number of items a user must remember. It also simplifies the selection of names for collections.

Source Files

All source files should start with a commented section as follows

```
//-------------------------------------------------------------------------
----------
// Source file name
// description of the contents of this file
//
// Created by: <Name of author>
//        On: <date of creation>
//
// Last modified by: <Name of programmer>
//        On: <date of last modification>
//
//-------------------------------------------------------------------------
----------
```

Code Style

Use the automated indenting provided by MSVC4. Tabs should be 4 spaces long.

Brackets

Always use brackets to wrap conditional parts of the code, use the following template:

```
if (xxxxxxx)
{
   function_one
}
```

and NOT:

```
if (xxxxxxx){
   function_one
}
```

OR:

```
if (xxxxxxx)
   function_one
```

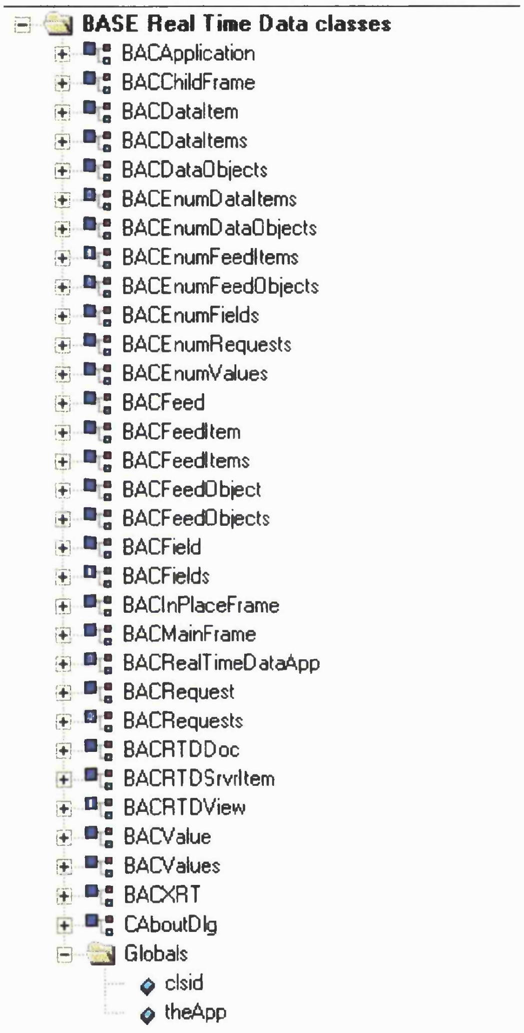
In Line functions

In line code should not exist. All declarations should be in header files. All implementation code should be in CPP files.

Appendix 7

CLASS LISTING FOR GENERIC SYSTEM

This diagram shows all the classes created for the generic system.

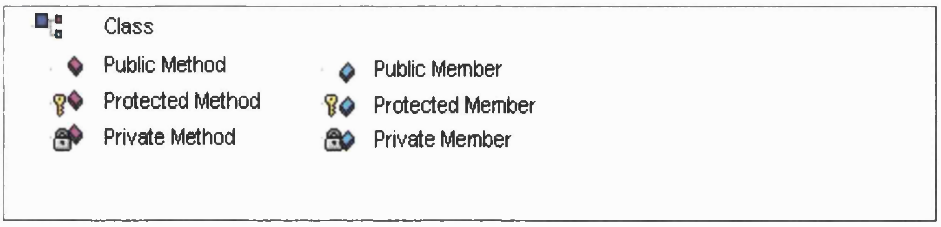


```
BASE Real Time Data classes
    BACApplication
    BACChildFrame
    BACDataItem
    BACDataItems
    BACDataObjects
    BACEnumDataItems
    BACEnumDataObjects
    BACEnumFeedItems
    BACEnumFeedObjects
    BACEnumFields
    BACEnumRequests
    BACEnumValues
    BACFeed
    BACFeedItem
    BACFeedItems
    BACFeedObject
    BACFeedObjects
    BACField
    BACFields
    BACInPlaceFrame
    BACMainFrame
    BACRealTimeDataApp
    BACRequest
    BACRequests
    BACRTDDoc
    BACRTDSrvrItem
    BACRTDView
    BACValue
    BACValues
    BACXRT
    CAboutDlg
    Globals
        clsid
        theApp
```

Appendix 8

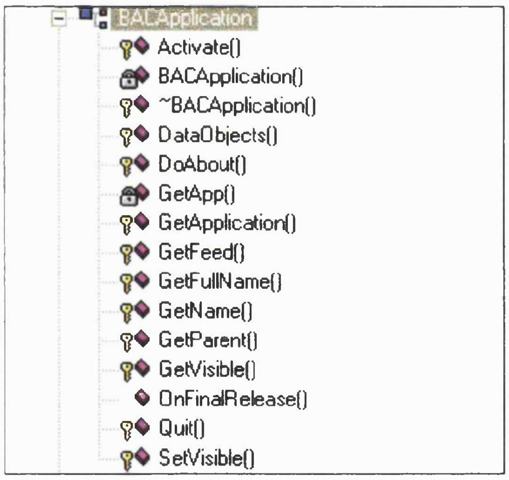
CLASSES FOR GENERIC SYSTEM

This appendix contains a class diagram for each of the main classes in the generic RTD system. The diagrams use the key shown below.



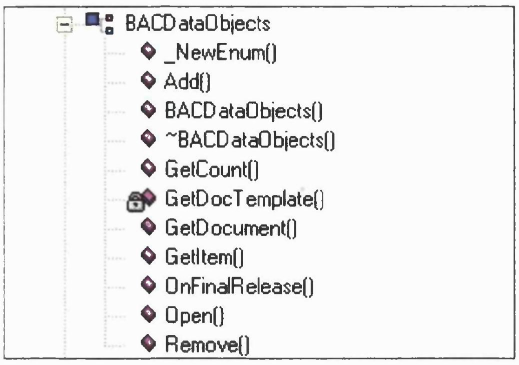
Class	
Public Method	Public Member
Protected Method	Protected Member
Private Method	Private Member

The Application Class



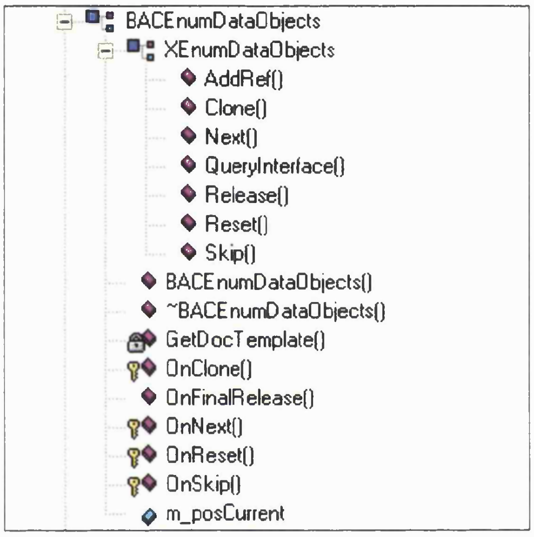
BACApplication
- Activate()
- BACApplication()
- ~BACApplication()
- DataObjects()
- DoAbout()
- GetApp()
- GetApplication()
- GetFeed()
- GetFullName()
- GetName()
- GetParent()
- GetVisible()
- OnFinalRelease()
- Quit()
- SetVisible()

The Data Objects Class



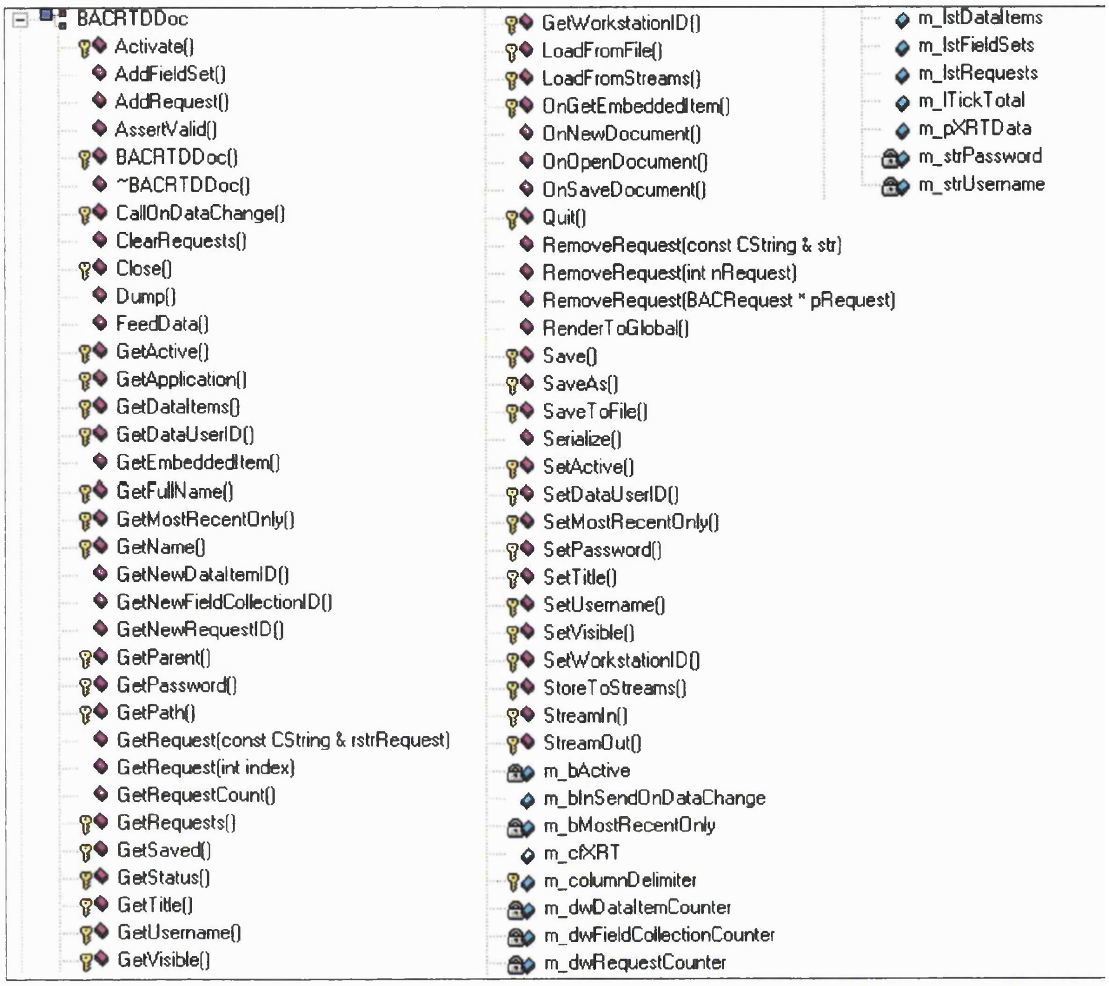
```
BACDataObjects
    _NewEnum()
    Add()
    BACDataObjects()
    ~BACDataObjects()
    GetCount()
    GetDocTemplate()
    GetDocument()
    GetItem()
    OnFinalRelease()
    Open()
    Remove()
```

The Data Object Enumerator Class



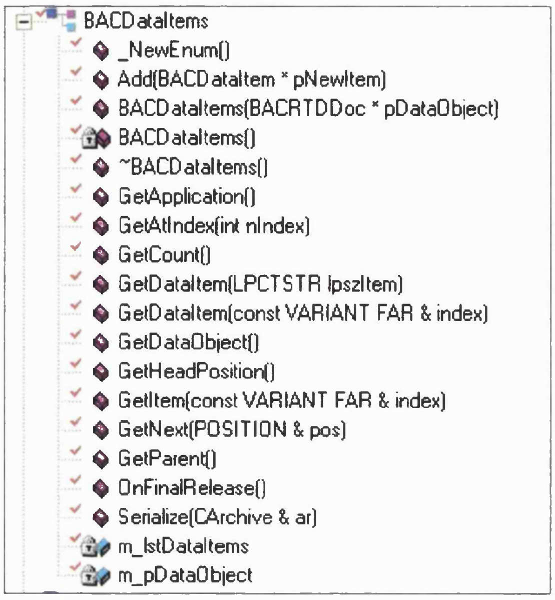
```
BACEnumDataObjects
    XEnumDataObjects
        AddRef()
        Clone()
        Next()
        QueryInterface()
        Release()
        Reset()
        Skip()
    BACEnumDataObjects()
    ~BACEnumDataObjects()
    GetDocTemplate()
    OnClone()
    OnFinalRelease()
    OnNext()
    OnReset()
    OnSkip()
    m_posCurrent
```

The Data Object Class



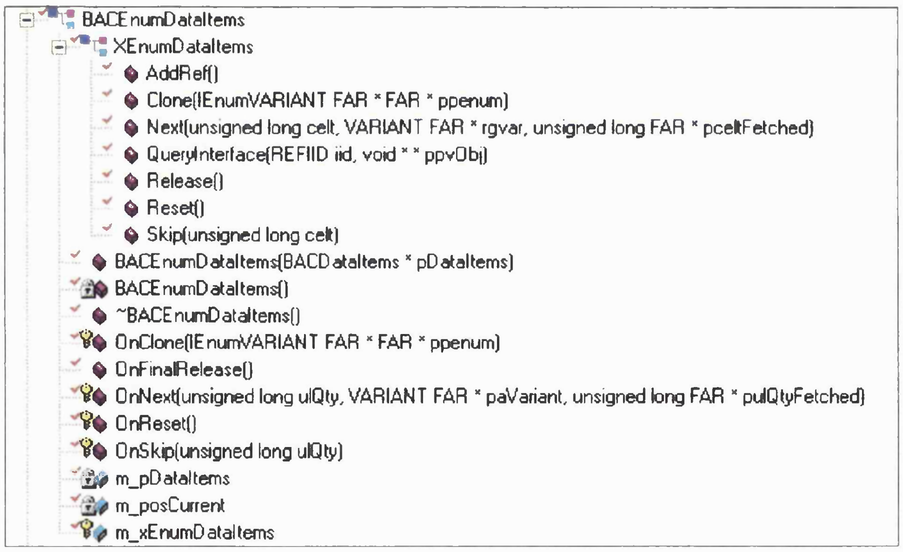
```
BACRTDDoc
    Activate()
    AddFieldSet()
    AddRequest()
    AssertValid()
    BACRTDDoc()
    ~BACRTDDoc()
    CallOnDataChange()
    ClearRequests()
    Close()
    Dump()
    FeedData()
    GetActive()
    GetApplication()
    GetDataItems()
    GetDataUserID()
    GetEmbeddedItem()
    GetFullName()
    GetMostRecentOnly()
    GetName()
    GetNewDataItemID()
    GetNewFieldCollectionID()
    GetNewRequestID()
    GetParent()
    GetPassword()
    GetPath()
    GetRequest(const CString & rstrRequest)
    GetRequest(int index)
    GetRequestCount()
    GetRequests()
    GetSaved()
    GetStatus()
    GetTitle()
    GetUsername()
    GetVisible()
    GetWorkstationID()
    LoadFromFile()
    LoadFromStreams()
    OnGetEmbeddedItem()
    OnNewDocument()
    OnOpenDocument()
    OnSaveDocument()
    Quit()
    RemoveRequest(const CString & str)
    RemoveRequest(int nRequest)
    RemoveRequest(BACRequest * pRequest)
    RenderToGlobal()
    Save()
    SaveAs()
    SaveToFile()
    Serialize()
    SetActive()
    SetDataUserID()
    SetMostRecentOnly()
    SetPassword()
    SetTitle()
    SetUsername()
    SetVisible()
    SetWorkstationID()
    StoreToStreams()
    StreamIn()
    StreamOut()
    m_bActive
    m_bInSendOnDataChange
    m_bMostRecentOnly
    m_cfXRT
    m_columnDelimiter
    m_dwDataItemCounter
    m_dwFieldCollectionCounter
    m_dwRequestCounter
    m_lstDataItems
    m_lstFieldSets
    m_lstRequests
    m_lTickTotal
    m_pXRTData
    m_strPassword
    m_strUsername
```

The Data Items Class



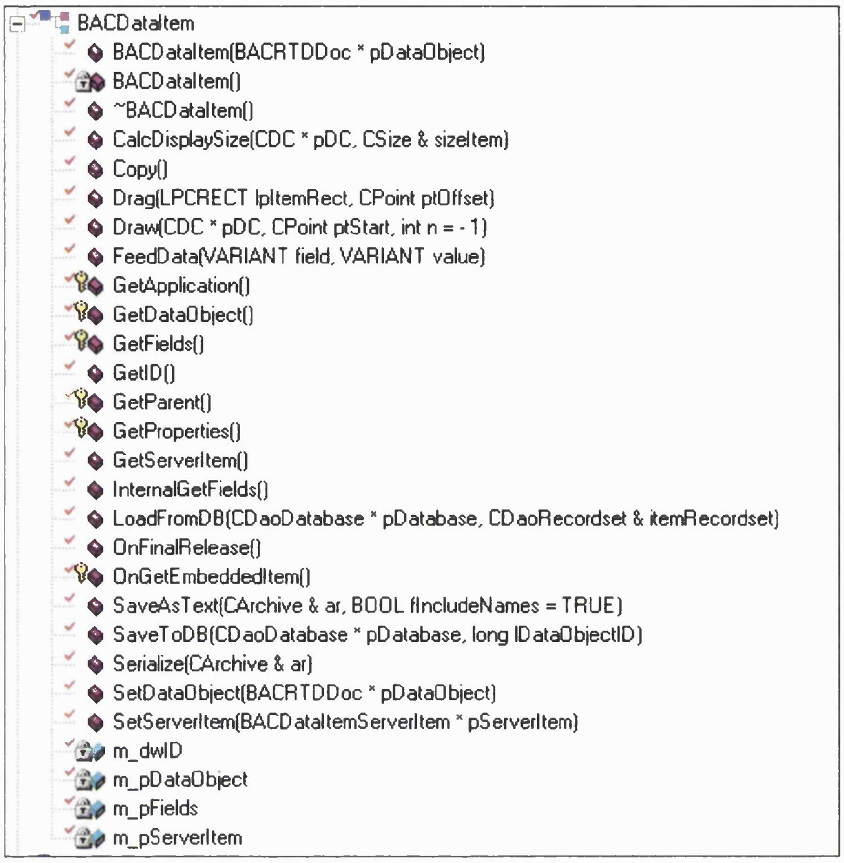
```
BACDataItems
    _NewEnum()
    Add(BACDataItem * pNewItem)
    BACDataItems(BACRTDDoc * pDataObject)
    BACDataItems()
    ~BACDataItems()
    GetApplication()
    GetAtIndex(int nIndex)
    GetCount()
    GetDataItem(LPCTSTR lpszItem)
    GetDataItem(const VARIANT FAR & index)
    GetDataObject()
    GetHeadPosition()
    GetItem(const VARIANT FAR & index)
    GetNext(POSITION & pos)
    GetParent()
    OnFinalRelease()
    Serialize(CArchive & ar)
    m_lstDataItems
    m_pDataObject
```

The Data Item Enumerator Class



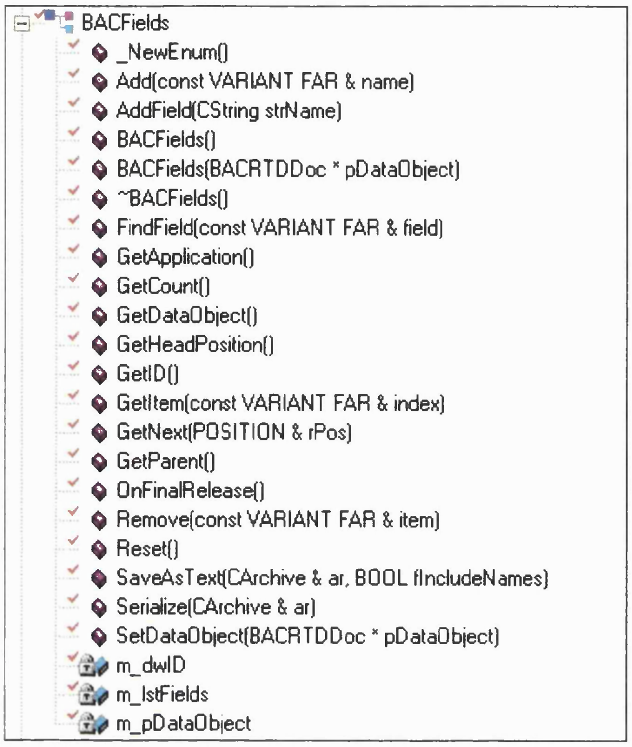
```
BACEnumDataItems
    XEnumDataItems
        AddRef()
        Clone(IEnumVARIANT FAR * FAR * ppenum)
        Next(unsigned long celt, VARIANT FAR * rgvar, unsigned long FAR * pceltFetched)
        QueryInterface(REFIID iid, void * * ppvObj)
        Release()
        Reset()
        Skip(unsigned long celt)
    BACEnumDataItems(BACDataItems * pDataItems)
    BACEnumDataItems()
    ~BACEnumDataItems()
    OnClone(IEnumVARIANT FAR * FAR * ppenum)
    OnFinalRelease()
    OnNext(unsigned long ulQty, VARIANT FAR * paVariant, unsigned long FAR * pulQtyFetched)
    OnReset()
    OnSkip(unsigned long ulQty)
    m_pDataItems
    m_posCurrent
    m_xEnumDataItems
```

The Data Item Class



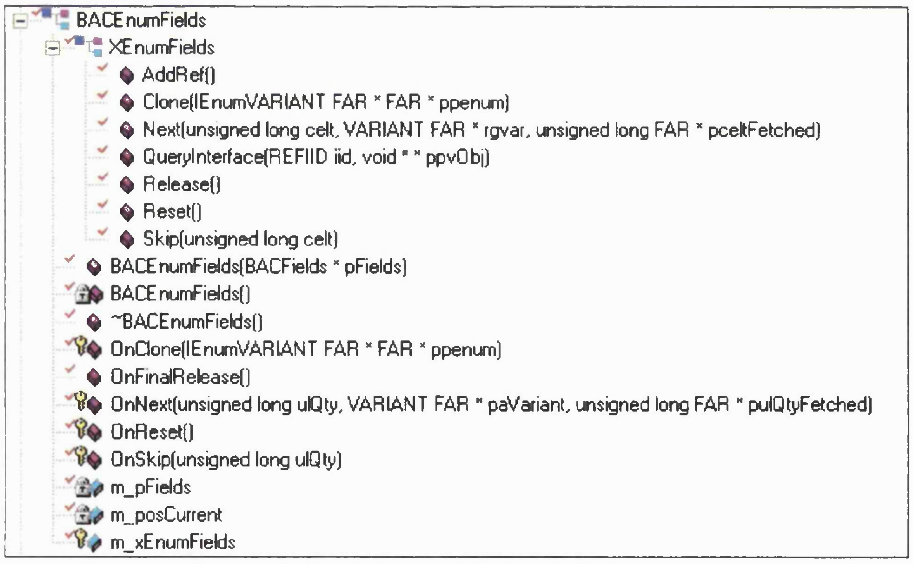
```
BACDataItem
    BACDataItem(BACRTDDoc * pDataObject)
    BACDataItem()
    ~BACDataItem()
    CalcDisplaySize(CDC * pDC, CSize & sizeItem)
    Copy()
    Drag(LPCRECT lpItemRect, CPoint ptOffset)
    Draw(CDC * pDC, CPoint ptStart, int n = - 1)
    FeedData(VARIANT field, VARIANT value)
    GetApplication()
    GetDataObject()
    GetFields()
    GetID()
    GetParent()
    GetProperties()
    GetServerItem()
    InternalGetFields()
    LoadFromDB(CDaoDatabase * pDatabase, CDaoRecordset & itemRecordset)
    OnFinalRelease()
    OnGetEmbeddedItem()
    SaveAsText(CArchive & ar, BOOL fIncludeNames = TRUE)
    SaveToDB(CDaoDatabase * pDatabase, long lDataObjectID)
    Serialize(CArchive & ar)
    SetDataObject(BACRTDDoc * pDataObject)
    SetServerItem(BACDataItemServerItem * pServerItem)
    m_dwID
    m_pDataObject
    m_pFields
    m_pServerItem
```

The Fields Class



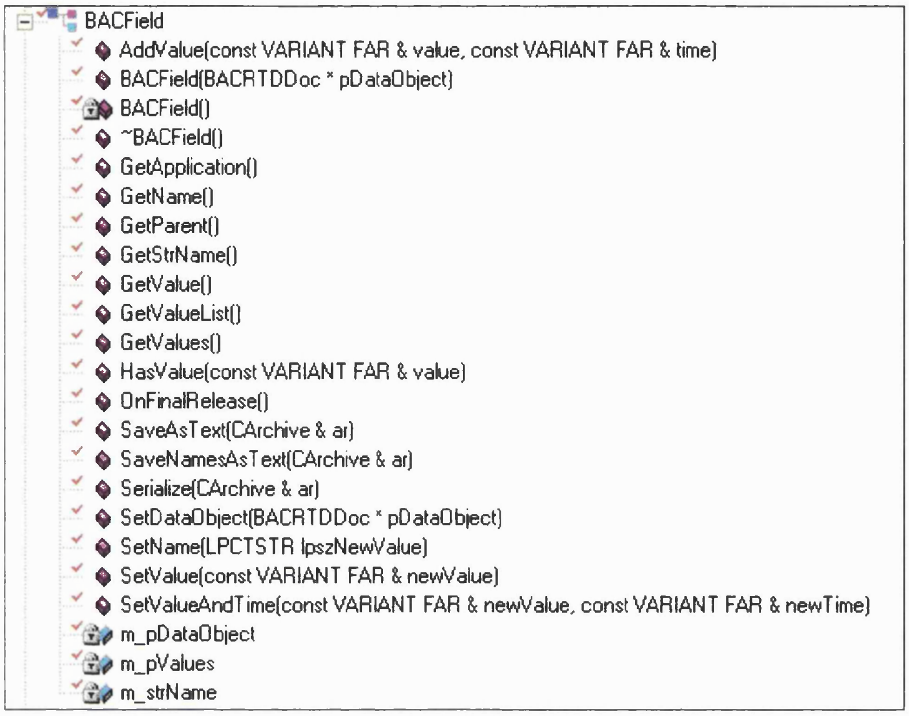
```
BACFields
    _NewEnum()
    Add(const VARIANT FAR & name)
    AddField(CString strName)
    BACFields()
    BACFields(BACRTDDoc * pDataObject)
    ~BACFields()
    FindField(const VARIANT FAR & field)
    GetApplication()
    GetCount()
    GetDataObject()
    GetHeadPosition()
    GetID()
    GetItem(const VARIANT FAR & index)
    GetNext(POSITION & rPos)
    GetParent()
    OnFinalRelease()
    Remove(const VARIANT FAR & item)
    Reset()
    SaveAsText(CArchive & ar, BOOL fIncludeNames)
    Serialize(CArchive & ar)
    SetDataObject(BACRTDDoc * pDataObject)
    m_dwID
    m_lstFields
    m_pDataObject
```

The Field Enumerator Class



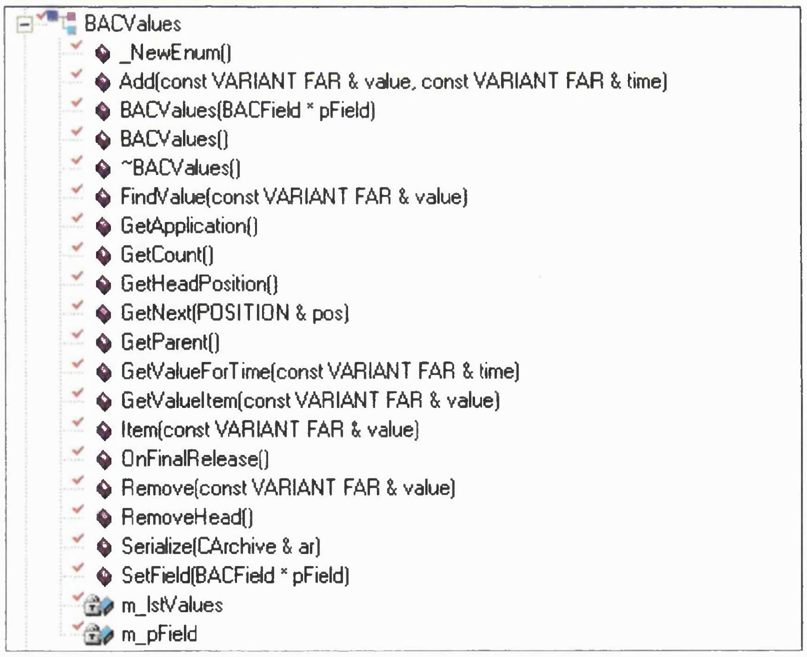
```
BACEnumFields
    XEnumFields
        AddRef()
        Clone(IEnumVARIANT FAR * FAR * ppenum)
        Next(unsigned long celt, VARIANT FAR * rgvar, unsigned long FAR * pceltFetched)
        QueryInterface(REFIID iid, void * * ppvObj)
        Release()
        Reset()
        Skip(unsigned long celt)
    BACEnumFields(BACFields * pFields)
    BACEnumFields()
    ~BACEnumFields()
    OnClone(IEnumVARIANT FAR * FAR * ppenum)
    OnFinalRelease()
    OnNext(unsigned long ulQty, VARIANT FAR * paVariant, unsigned long FAR * pulQtyFetched)
    OnReset()
    OnSkip(unsigned long ulQty)
    m_pFields
    m_posCurrent
    m_xEnumFields
```

The Field Class



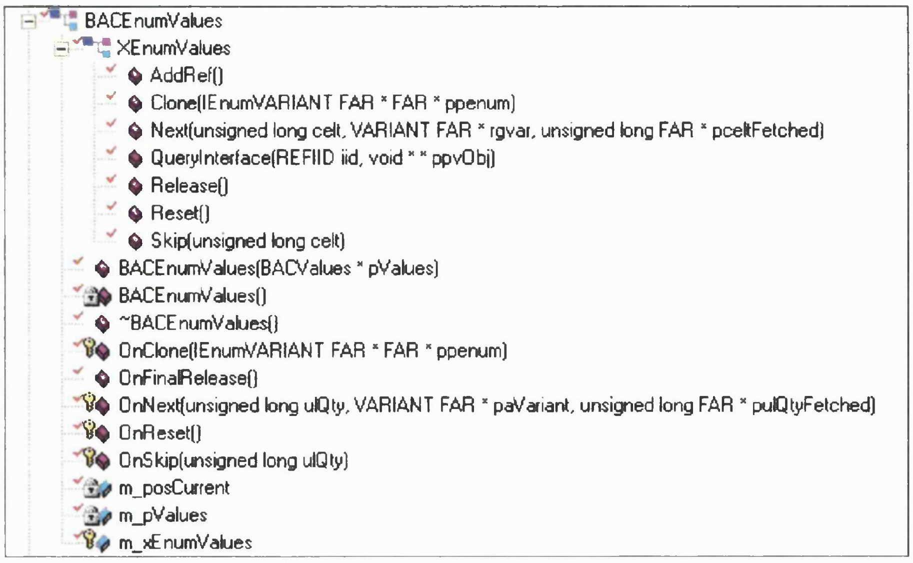
```
BACField
    AddValue(const VARIANT FAR & value, const VARIANT FAR & time)
    BACField(BACRTDDoc * pDataObject)
    BACField()
    ~BACField()
    GetApplication()
    GetName()
    GetParent()
    GetStrName()
    GetValue()
    GetValueList()
    GetValues()
    HasValue(const VARIANT FAR & value)
    OnFinalRelease()
    SaveAsText(CArchive & ar)
    SaveNamesAsText(CArchive & ar)
    Serialize(CArchive & ar)
    SetDataObject(BACRTDDoc * pDataObject)
    SetName(LPCTSTR lpszNewValue)
    SetValue(const VARIANT FAR & newValue)
    SetValueAndTime(const VARIANT FAR & newValue, const VARIANT FAR & newTime)
    m_pDataObject
    m_pValues
    m_strName
```

The Values Class



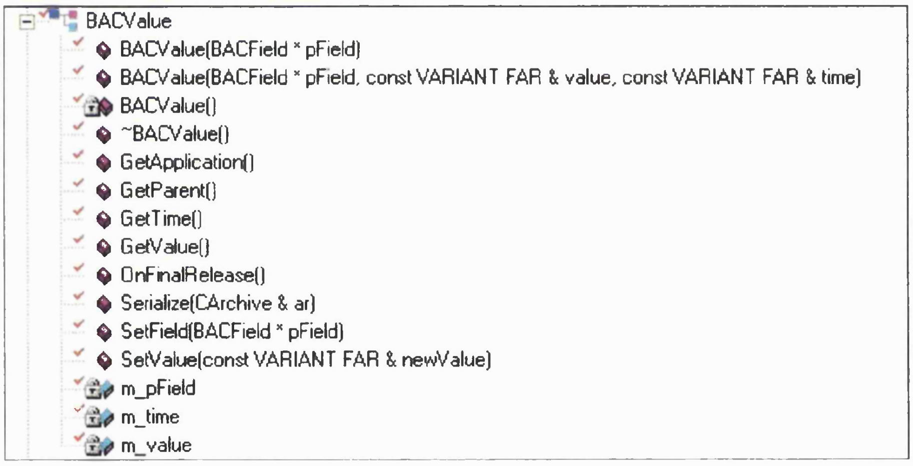
```
BACValues
    _NewEnum()
    Add(const VARIANT FAR & value, const VARIANT FAR & time)
    BACValues(BACField * pField)
    BACValues()
    ~BACValues()
    FindValue(const VARIANT FAR & value)
    GetApplication()
    GetCount()
    GetHeadPosition()
    GetNext(POSITION & pos)
    GetParent()
    GetValueForTime(const VARIANT FAR & time)
    GetValueItem(const VARIANT FAR & value)
    Item(const VARIANT FAR & value)
    OnFinalRelease()
    Remove(const VARIANT FAR & value)
    RemoveHead()
    Serialize(CArchive & ar)
    SetField(BACField * pField)
    m_lstValues
    m_pField
```

The Value Enumerator Class



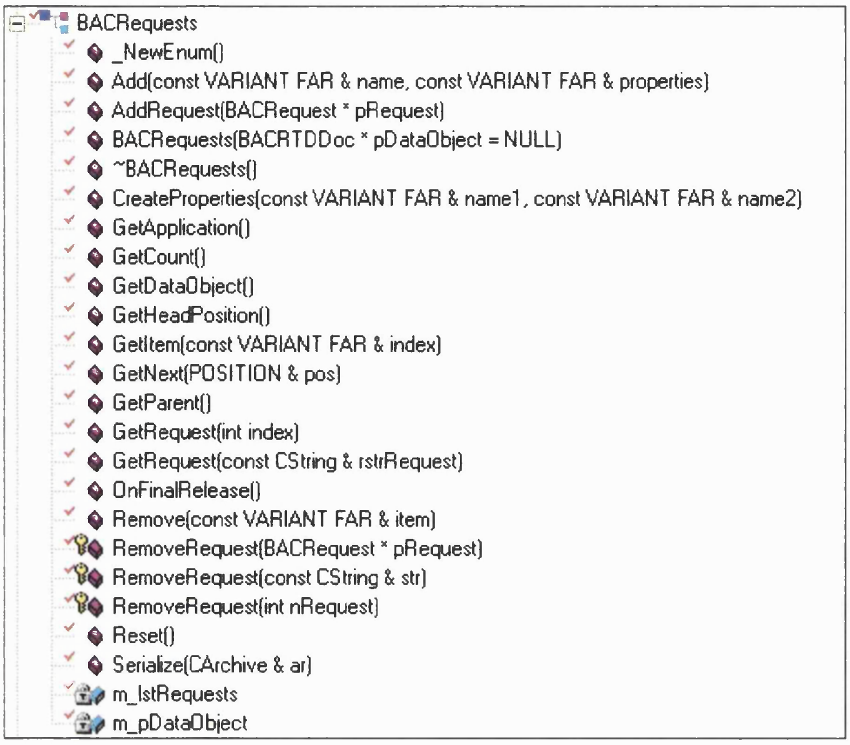
```
BACEnumValues
  XEnumValues
    AddRef()
    Clone(IEnumVARIANT FAR * FAR * ppenum)
    Next(unsigned long celt, VARIANT FAR * rgvar, unsigned long FAR * pceltFetched)
    QueryInterface(REFIID iid, void * * ppvObj)
    Release()
    Reset()
    Skip(unsigned long celt)
  BACEnumValues(BACValues * pValues)
  BACEnumValues()
  ~BACEnumValues()
  OnClone(IEnumVARIANT FAR * FAR * ppenum)
  OnFinalRelease()
  OnNext(unsigned long ulQty, VARIANT FAR * paVariant, unsigned long FAR * pulQtyFetched)
  OnReset()
  OnSkip(unsigned long ulQty)
  m_posCurrent
  m_pValues
  m_xEnumValues
```

The Value Class

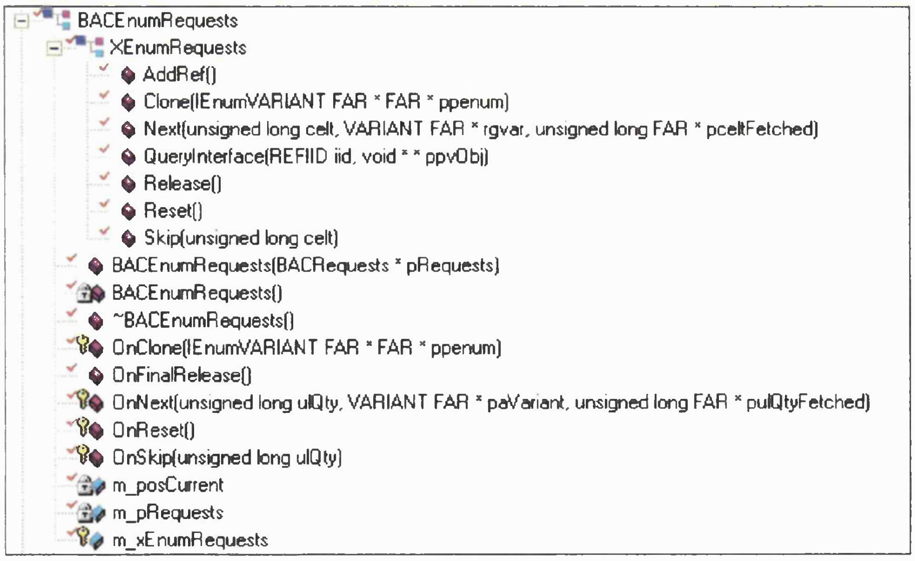


```
BACValue
  BACValue(BACField * pField)
  BACValue(BACField * pField, const VARIANT FAR & value, const VARIANT FAR & time)
  BACValue()
  ~BACValue()
  GetApplication()
  GetParent()
  GetTime()
  GetValue()
  OnFinalRelease()
  Serialize(CArchive & ar)
  SetField(BACField * pField)
  SetValue(const VARIANT FAR & newValue)
  m_pField
  m_time
  m_value
```

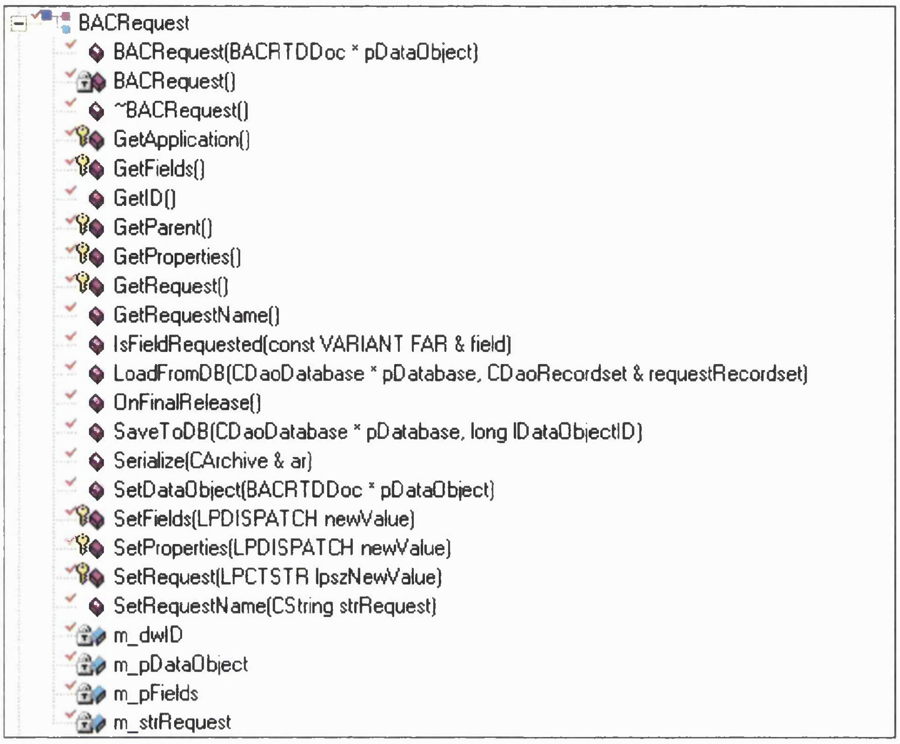
The Requests Class


```
BACRequests
    _NewEnum()
    Add(const VARIANT FAR & name, const VARIANT FAR & properties)
    AddRequest(BACRequest * pRequest)
    BACRequests(BACRTDDoc * pDataObject = NULL)
    ~BACRequests()
    CreateProperties(const VARIANT FAR & name1, const VARIANT FAR & name2)
    GetApplication()
    GetCount()
    GetDataObject()
    GetHeadPosition()
    GetItem(const VARIANT FAR & index)
    GetNext(POSITION & pos)
    GetParent()
    GetRequest(int index)
    GetRequest(const CString & rstrRequest)
    OnFinalRelease()
    Remove(const VARIANT FAR & item)
    RemoveRequest(BACRequest * pRequest)
    RemoveRequest(const CString & str)
    RemoveRequest(int nRequest)
    Reset()
    Serialize(CArchive & ar)
    m_lstRequests
    m_pDataObject
```

The Request Enumerator Class


```
BACEnumRequests
    XEnumRequests
        AddRef()
        Clone(IEnumVARIANT FAR * FAR * ppenum)
        Next(unsigned long celt, VARIANT FAR * rgvar, unsigned long FAR * pceltFetched)
        QueryInterface(REFIID iid, void * * ppvObj)
        Release()
        Reset()
        Skip(unsigned long celt)
    BACEnumRequests(BACRequests * pRequests)
    BACEnumRequests()
    ~BACEnumRequests()
    OnClone(IEnumVARIANT FAR * FAR * ppenum)
    OnFinalRelease()
    OnNext(unsigned long ulQty, VARIANT FAR * paVariant, unsigned long FAR * pulQtyFetched)
    OnReset()
    OnSkip(unsigned long ulQty)
    m_posCurrent
    m_pRequests
    m_xEnumRequests
```

The Request Class



```
BACRequest
    BACRequest(BACRTDDoc * pDataObject)
    BACRequest()
    ~BACRequest()
    GetApplication()
    GetFields()
    GetID()
    GetParent()
    GetProperties()
    GetRequest()
    GetRequestName()
    IsFieldRequested(const VARIANT FAR & field)
    LoadFromDB(CDaoDatabase * pDatabase, CDaoRecordset & requestRecordset)
    OnFinalRelease()
    SaveToDB(CDaoDatabase * pDatabase, long lDataObjectID)
    Serialize(CArchive & ar)
    SetDataObject(BACRTDDoc * pDataObject)
    SetFields(LPDISPATCH newValue)
    SetProperties(LPDISPATCH newValue)
    SetRequest(LPCTSTR lpszNewValue)
    SetRequestName(CString strRequest)
    m_dwID
    m_pDataObject
    m_pFields
    m_strRequest
```

BIBLIOGRAPHY

[1] Cooling, J E. *Software Design for Real-time Systems.* International Thomson Computer Press, 1995.

[2] Yin Leng Theng, Harold Thimbleby, Matthew Jones, *Reducing information overload: A comparative study of hypertext systems*, School of Computing Science, Middlesex University

[3] Dennis Eskow, *Beat Information Overload*, February 1997, PC World

[4] Brockschmidt, Kraig. *Inside OLE* Microsoft Press, 1995.

[5] Wyllie, Jan. *Turning information overload into useful knowledge resources.* Internet 1996.

[6] WOSA Extensions for Real Time Market Data (WOSA/XRT) Design Specification. Open Market Data Council For Windows

[7] Pressman, *Software Engineering, A Practitioners Approach*, McGraw Hill

[8] *OLE2 Programmer's Reference Vols One & Two.* Microsoft Press, 1993.

[9] Crittenden, John. *Information Overload* Feature Articles PacificByte 1996

[10] Trufitt, Ken. *Information Overload* Internet 1996.

[11] Dr. M. Timmerman Monfret, *Windows NT as Real-Time OS?*, Real Time Magazine

[12] Brookes *'The Mythical Man Month'*

[13] Crosby, P. 'Quality Is Free', McGraw-Hill.

[14] M.A.I.D plc Targets Information Overload in 1997, Internet November 1996

[15] Robert M. Losee, Parameter Estimation for Probabilistic Document-Retrieval Models, Journal of the American Society for Information Science, 39(1), 1988, p. 8-16.

[16] Baclace, Paul E (1991): Personal Information Intake Filtering. In: Proceedings of Bellcore Workshop on High-Performance Information Filtering (Morriston, N.J.).

[17] Baclace, Paul E (1992): Competitive Agents for Information Filtering. Commun. ACM 35(12, December), 50.

[18] Belkin, Nicholas J; Croft, W Bruce (1992): Information Filtering and Information Retrieval: Two Sides of the Same Coin?. Commun. ACM 35(12, December), 29-38.

[19] Chimera, Richard; Shneiderman, Ben (1994): An Exploratory Evaluation of Three Interfaces for Browsing Large Hierarchical Tables of Contents. ACM Transactions on Information Systems 12(4, October), 383-406.

[20] Fisher, G; Stevens, C (1991): Information Access in Complex, Poorly Structured Information Spaces. In: Human Factors in Computing Systems CHI'91 Conference Proceedings (New Orleans, La. Apr. 1991). (Eds: Robertson, Scott P; Olson, Gary M; Olson, Judith S) ACM, New York, 63-70.

[21] Gant, Stephen P (1995): A Portarit of Potential Adopters of Information Filters. In: ASIS'95. Vol. 32. (Ed: Kinney, Tom) Information Today, Medford, New Jersey, 167-171.

[22] Losee, Robert M Jr (1989): Minimizing Information Overload: the Ranking of Electronic Messages. Journal of Information 15, 179-89.

[49] Miller G. 'The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information'. The Psychological Review vol.63(2)
[50] Microsoft Windows 3.1 Programmer's Reference, Microsoft Press, 1992
[51] Roodyn, Neil and Emmerich, Wolfgang. 'An Architectural Style for Multiple Real-Time Data Feeds'. ICSE 1999