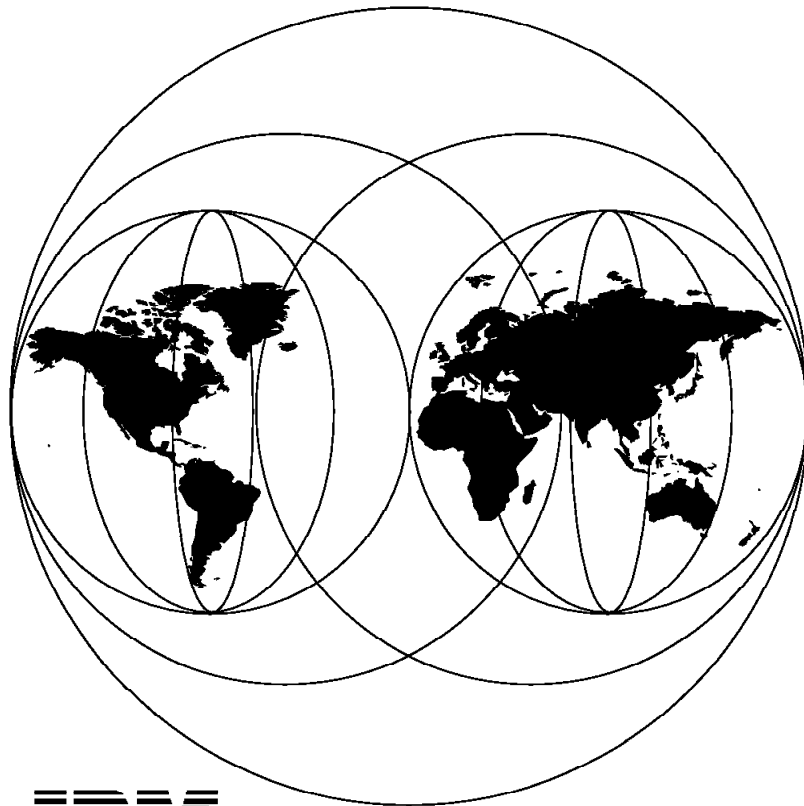


International Technical Support Organization

SG24-4664-00

**MQSeries Three Tier
Examples for Windows Clients and AIX Servers**

March 1996



IBM

**International Technical Support Organization
Raleigh Center**



International Technical Support Organization

SG24-4664-00

**MQSeries Three Tier
Examples for Windows Clients and AIX Servers**

March 1996

Take Note!

Before using this information and the product it supports, be sure to read the general information under "Special Notices" on page xv.

First Edition (March 1996)

This edition applies to Version 1.0 of IBM MQSeries Three Tier for AIX, part numbers 33H2163 and 33H2168, program number 5765-321.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader's feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. HZ8 Building 678
P.O. Box 12195
Research Triangle Park, NC 27709-2195

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1996. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Abstract

This redbook describes how to integrate Windows client workstations into an MQSeries Three Tier (3T) client/server environment. It contains an introduction into the 3T product, explains its object-oriented three-tier logic, and provides step-by-step instructions on how to design, build, and test applications.

3T clients (the first tier) run on Windows or OS/2 systems. 3T servers (the second tier) run on AIX or OS/2 machines. The third tier can be existing MQSeries programs running in the host or on any platform supported by MQSeries.

This document describes how to set up a Windows client and an AIX machine for the development of a 3T application. The client part of the application is a graphical user interface written in Visual Basic and developed on a Windows workstation. The applications in the server are either existing MQSeries programs or MQSeries Three Tier objects written in C.

This redbook was written for those who quickly want to learn what IBM MQSeries Three Tier is about and how it works. It will be of help for programmers who have to develop message queuing applications for Windows workstations and AIX servers. Data conversion between the two platforms is discussed, too. Some knowledge of DOS, Windows, and AIX is assumed.

Programmers with no knowledge of MQSeries and Visual Basic may wish to use this publication as a textbook.

(268 pages)

Contents

Abstract	iii
Special Notices	xv
Preface	xvii
How This Document is Organized	xviii
Related Publications	xviii
International Technical Support Organization Publications	xix
How Customers Can Get Redbooks and Other ITSO Deliverables	xix
How IBM Employees Can Get Redbooks and ITSO Deliverables	xx
Acknowledgments	xxiii
Chapter 1. IBM MQSeries Three Tier Overview	1
1.1 The Three Tiers	2
1.1.1 Run-Time Components	3
1.1.2 Messages	4
1.1.3 Message Flow	5
1.2 3T Facilities	6
1.2.1 Class Definition Compiler	6
1.2.2 Application Program Interface	7
1.2.3 Application Simulator	8
1.2.4 Test Harness	8
1.2.5 Job Viewer	8
1.2.6 Self-Defining Data Manager	8
1.2.7 Visual Basic Support	9
1.3 Application Design	9
1.3.1 The 3T Application Model	11
1.3.2 The 3T Application Development Process	12
1.3.3 Structured Application Design	14
Chapter 2. Installation	15
2.1 AIX Server	16
2.1.1 MQSeries Base Product	16
2.1.2 CSD 14 for MQSeries	21
2.1.3 Creating MQM Objects	22
2.1.4 MQSeries Three Tier for AIX	25
2.2 Windows Client for Development	28
2.2.1 IBM DOS 7.0	28
2.2.2 MS Windows Version 3.1	29
2.2.3 TCP/IP	30
2.2.4 MQSeries Windows Client	34
2.2.5 3T Windows Client	39
2.2.6 Visual Basic	40
2.2.7 Visual Basic Support for Windows Clients	41
2.3 Windows Client for Production	42
Chapter 3. Using Visual Basic	43
3.1 Introduction	43
3.2 The SupportPac	43
3.3 The Visual Basic / MQ3T Interface	44

3.4	Parameter Passing	47
3.5	SupportPac Content	50
3.5.1	Base Functions	50
3.5.2	Sample Programs	51
3.5.3	Sample Read Only Code Fragments	61
3.5.4	A Template for Your Own Program	63
3.5.5	Run-Time Utility	64
3.6	Application Programming Interface Calls	64
3.6.1	Types of API Calls	65
3.6.2	Notes to API Calls	66
3.7	Using the Visual Basic 3T Sample Programs	67
3.7.1	Preparations on the AIX Server	67
3.7.2	Preparations on the Windows Client	71
3.7.3	Running the HELLO1 Sample	73
Chapter 4. File Transfer Example		77
4.1	Application Description	78
4.2	Set Up and Run the MQI Application	80
4.2.1	Set Up of the Sender Workstation	80
4.2.2	Set Up of the Receiver Workstation	83
4.2.3	Running the File Transfer Example	84
4.3	Set Up and Run the MQ3T Application	86
4.3.1	Set Up the Sender Workstation	86
4.3.2	Set Up the Receiver Workstation	89
4.3.3	Set Up the Windows Workstation	89
4.3.4	Running the MQ3T File Transfer Example	90
4.4	Developing the MQ3T Application	94
4.4.1	Defining Class Source Files	94
4.4.2	Compiling Class Source Files	97
4.4.3	Routing Messages	99
4.4.4	Writing the Business Logic	100
4.4.5	Writing the Presentation Logic	104
4.4.6	A Software Distribution Application	107
Chapter 5. The Bacon Lettuce and Tomato Sandwich		111
5.1	Requirements	112
5.2	Business Analysis	113
5.2.1	Objects and Their Functions	114
5.2.2	Message Flow between Objects	116
5.2.3	GUI Prototypes	118
5.3	3T Design	121
5.3.1	3T Classes	121
5.3.2	Messages	122
5.3.3	Class Descriptions	127
5.3.4	Rules and Methods	128
5.4	Design Crosscheck	139
5.5	Building the GUIs	141
5.5.1	Project Konrad	144
5.5.2	Project Luigi	151
5.5.3	Project Gremlin	155
5.5.4	Project Repair List	159
5.5.5	Project Shopping List	166
5.6	Building the Business Logic	173
5.6.1	Creating Skeleton Files	173
5.6.2	Creating The Business Logic	179

5.7 System Test	198
Chapter 6. Data Conversion	203
6.1 Creating a Conversion DLL for AIX	205
6.2 Creating a Conversion DLL for OS/2	210
Appendix A. Class Source Files for BLT Example	213
A.1 Messages for The BLT Example	213
A.2 Class Descriptions for The BLT Example	218
A.3 Class Source File for BASKET	220
A.4 Class Source File for BREADBOX	223
A.5 Class Source File for FRIDGE	226
A.6 Class Source File for GREMLIN	229
A.7 Class Source File for GROCER	230
A.8 Class Source File for KAREN	232
A.9 Class Source File for KONRAD	236
A.10 Class Source File for LUIGI	238
A.11 Class Source File for MICRO	240
A.12 Class Source File for REPAIR	242
A.13 Class Source File for SHOPPING	243
A.14 Class Source File for TOASTER	245
A.15 Definitions for Class Source Files	247
A.16 Definitions for Visual Basic	249
Appendix B. Summary of MQ3T APIs	251
Appendix C. Diskette Contents	257
List of Abbreviations	259
Index	261

Figures

1.	The Three Tier Run-Time Components	3
2.	The Three Tiers of 3T	5
3.	Programmer's OS/2 Workstation	9
4.	Programmer's Windows Workstation	10
5.	Programmer's AIX Workstation	10
6.	Project's Configuration	15
7.	MQSeries for AIX Installation, Device Selection	17
8.	MQSeries for AIX Installation	18
9.	Select Root Volume Group (VG)	22
10.	Add a Journaled File System	23
11.	QM.INI File	24
12.	MQSeries 3T for AIX Installation	25
13.	TCP/IP Configure Menu	31
14.	Ping a Host	34
15.	MQSeries for Windows Clients: Files	35
16.	Verify the Host Connection	38
17.	Issue Single MQI APIs	38
18.	MQ3T Icons in Windows Program Manager	40
19.	Visual Basic Icons	41
20.	Register a Window with 3T	44
21.	Unregister a Window with 3T	45
22.	Parameter Passing between PL and PLM	47
23.	PL (GUI) Receives MQ3T Events	48
24.	PL (GUI) Sends a Message	49
25.	BMQVBX.BAS File	51
26.	Sample HELLO1: HELOGU1W.MAK	52
27.	Sample HELLO1: Visual Basic's Dialog Box Description	53
28.	Sample HELLO1: Display the Window	53
29.	Sample HELLO1: Declarations	54
30.	Sample HELLO1: Display Completion Codes	54
31.	Sample HELLO1: Receive Messages from BL	54
32.	Sample HELLO1: Process Fixed-length Messages from BL	55
33.	Sample HELLO1: Menu Item Close	55
34.	Sample HELLO1: Send a Fixed-length Message to the BL	56
35.	Sample HELLO1: Close the Window	56
36.	Sample HELLO1: Exit the Program	56
37.	HELLO1H.BAS and HELLO2H.BAS Files	58
38.	Sample HELLO2: HELOGU2W.MAK	59
39.	Sample HELLO2: Send a Variable-length Message to the BL	59
40.	Sample HELLO2: Process Variable-length Messages from BL	60
41.	Sample PFCUST: Find Customer GUI	61
42.	Sample PCUST: Customer Details GUI	62
43.	Template GUI Provided with SupportPac	63
44.	SPEEDUP Program: GUI	64
45.	Messages for MAKE of HELLO1 Sample	70
46.	Options for Modification of the File Transfer Program	77
47.	Shell Script File "foo1.cmd"	80
48.	Utility "killmqm.cmd"	82
49.	Shell Script File "first.cmd"	85
50.	MQSeries Three Tier Window	91
51.	STRPLM Window	92

52.	Pop-up: PLM Started	92
53.	STARTJOB Window	93
54.	GUI for File Transfer Program	93
55.	PL Class File "hellogu1.cs"	96
56.	Class Header File "helloms1.ch"	96
57.	BL Class File "helob1cx.cs"	96
58.	Header File "hello1x.h"	96
59.	Compiling a Class Source File	97
60.	Command File "redo"	98
61.	Routing Messages to "hellodl1"	99
62.	Profile with Server and Class Sections	99
63.	Skeleton File "hellobl1.c"	100
64.	MQ3T File Transfer: Sender Program (BL) "hello1x.c" (Part 1)	101
65.	MQ3T File Transfer: Sender Program (BL) "hello1x.c" (Part 2)	102
66.	MQ3T File Transfer: Declarations	104
67.	MQ3T File Transfer: Exit	104
68.	MQ3T File Transfer: Display the Window	105
69.	MQ3T File Transfer: Process an Event Message	105
70.	MQ3T File Transfer: User Input	106
71.	MQ3T File Transfer: Send Input Parameters to BL	106
72.	Software Distribution Application	107
73.	Profile for Server Supporting Multiple Classes	109
74.	Server Profile with Class and Queue Definition	109
75.	BLT: Message Flow between You and Your Wife	111
76.	BLT: Objects and Message Flow, Production	116
77.	BLT: Objects and Message Flow, Maintenance	117
78.	BLT: Circular Message Flow	117
79.	GUI Prototype for Konrad	118
80.	GUI Prototype for Gremlin	119
81.	GUI Prototype for Shopping List	119
82.	GUI Prototype for Repair List	120
83.	GUI Prototype for Luigi	120
84.	BLT: Messages in Production Process	122
85.	BLT: Messages in Inventory Control Process	123
86.	BLT: Messages in Food Order Process	124
87.	BLT: Messages in Exception/Maintenance Process	124
88.	A Message Description	126
89.	Message Structures	127
90.	Class Descriptions	127
91.	Input File for Design Crosscheck: "classes.lst"	139
92.	Output File from Design Crosscheck: "classes.xck"	140
93.	Profile to Start Five PLs	141
94.	Microsoft Visual Basic (design) Window	142
95.	Add BMQNTFY.VBX to a Project	143
96.	Add MQ3T Files to a Project	143
97.	Generic Frame and Project Window for BLT	144
98.	Konrad's Frame at Design and Run Time	145
99.	Form_Load Procedure	146
100.	Form_Unload Procedure	146
101.	Quit Procedure	146
102.	Visual Basic: Create a New Procedure	146
103.	BLT: Declarations	147
104.	Konrad: BLT Push Button Procedure	148
105.	Konrad: Pizza Push Button Procedure	148
106.	BLT: Display Messages in Text Box	149

107. BLT: Events from MQ3T	149
108. Konrad: Process Messages	150
109. Luigi's Project File	151
110. Luigi's Frame at Design Time	152
111. Luigi: Deliver Procedure	153
112. Luigi: Process PL Events	154
113. The Gremlin's Project File	155
114. The Gremlin's Frame at Design Time	156
115. The Gremlin's Radio Button Procedure	157
116. The Gremlin's Push Button Procedure	158
117. The Gremlin's Event Procedure	159
118. The Repair List's Project File	159
119. The Repair List's Frame at Design Time	160
120. The Repair List's Declarations	161
121. BLT: Display Messages in a List Box	162
122. Repair List: Radio Button Procedure	162
123. Repair List: Send Inquiry Request	163
124. Repair List: Send Repair Message	163
125. Repair List: Process PL Events (Part 1)	164
126. Repair List: Process PL Events (Part 2)	165
127. Shopping List's Frame at Design Time	166
128. Shopping List: Type a Quantity	168
129. Shopping List: Radio Button Procedure	168
130. Shopping List: Send Inquiry Message	169
131. Shopping List: Send an Order to the Grocer	170
132. Shopping List: Process PL Events (Part 1)	171
133. Shopping List: Process PL Events (Part 2)	172
134. Karen's Export File	173
135. Karen's C Skeleton File (Part 1)	175
136. Karen's C Skeleton File (Part 2)	176
137. Karen's Make File	178
138. Karen's Method "bltorder"	180
139. Karen's Method "bltmake" (Part 1 of 2)	181
140. Karen's Method "bltmake" (Part 2 of 2)	182
141. Karen's Method "bltserve"	183
142. Karen's Method "bltnone"	183
143. Refrigerator's Method "fridge1" (Part 1)	184
144. Refrigerator's Method "fridge1" (Part 2)	185
145. Refrigerator's Method "fridge1" (Part 3)	186
146. Food Inquiry Method "foodinq"	188
147. Food Delivery Method "delivery"	189
148. Food Preparation Method "cook"	190
149. Grocer's Method "grocer1.c" (Part 1)	191
150. Grocer's Method "grocer1.c" (Part 2)	192
151. Grocer's Method "grocer1.c" (Part 3)	193
152. Common Method "xclear.c"	194
153. Common Method "xrepair.c"	194
154. Common Method "xignore.c"	195
155. Common Method "xinqury.c"	196
156. Common Method "xgremlin.c"	197
157. BLT: Redo all BLs	198
158. BLT: Queue Definitions "bltcoma.tst"	199
159. BLT: Profile for Presentation Logic	200
160. BLT: Profile for Business Logic	200
161. Message Description with Conversion DLL	203

162. Message Description without Conversion DLL	204
163. Message Structure that Needs Data Conversion	205
164. Data Exit Source File	205
165. C Source Program for Conversion Conversion Exit (Part 1)	207
166. C Source Program for Conversion Conversion Exit (Part 2)	208
167. Make File MSG100.mak for AIX	209
168. Make File MSG100.mak for OS/2	211
169. MSG100.def File for OS/2	211

Tables

1.	Addresses for TCP/IP Customization	31
2.	MQREG Parameters	45
3.	MQSETS Parameters	46
4.	MQUREG Parameters	46
5.	MQSEND Parameters	49
6.	Icons Used for Objects in Visual Basic Support	50
7.	MQQRYE Parameters	57
8.	MQQRYM Parameters	57
9.	MQENDE Parameters	57
10.	Files for HELLO1 Sample	69
11.	Files for MQI File Transfer Example	81
12.	Files for MQ3T File Transfer Example	88
13.	Client's Files for MQ3T File Transfer Example	90
14.	Class Descriptions - hellopr1.ch	95
15.	Objects and Their Functions	114
16.	BLT: 3T Classes	121
17.	BLT: Message Summary	125
18.	BLT: Methods for Presentation Logics	129
19.	BLT: Rules for Presentation Logics	131
20.	BLT: Methods for Business Logics	133
21.	BLT: Rules for Business Logics	134
22.	MQTIME Parameters	136
23.	Gremlin: Actions and Destinations for Radio Buttons	158
24.	Repair List: Actions and Destinations for Radio Buttons	162
25.	Shopping List: Actions and Destinations for Radio Buttons	168
26.	MQSeries 3T APIs for Visual Basic	251

Special Notices

This publication is intended to help application and system programmers with additional guidance in using the MQSeries Three Tier for AIX product and the Visual Basic support for Windows clients. The information in this publication is not intended as the specification of any programming interfaces that are provided by:

- IBM MQSeries Three Tier for AIX
- IBM MQSeries Three Tier for OS/2
- IBM MQSeries for AIX
- Microsoft Visual Basic Version 3.0
- Microsoft Windows Version 3.1

See the PUBLICATIONS section of the IBM Programming Announcement for the MQSeries product family for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594 USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following document contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples contain the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

AIX	AIX/6000
AIXwindows	IBM
MQSeries	MQSeries Three Tier
Operating System/2	OS/2
RISC System/6000	RS/6000
SUPPORTPAC	

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and the Windows 95 logo are trademarks of Microsoft Corporation.

Other trademarks are trademarks of their respective companies.

Preface

This document is intended to help programmers in setting up Windows clients and AIX servers for both MQSeries Three Tier application development and production environments. It explains to application developers and designers how MQSeries Three Tier can be used to quickly design and code fairly complex applications.

3T clients (the first tier) run on Windows or OS/2 systems. 3T servers (the second tier) run on OS/2 or AIX machines. The third tier can be existing MQSeries programs running in the host or on any platform supported by MQSeries.

MQSeries Three Tier consists of application development facilities and run-time programs. The development facilities help designing application models, consisting of classes, the message flow between them, and methods to process the messages. It simplifies the development of client/server applications. The run-time programs interface between application and MQSeries, providing functions such as message routing and automatic triggering.

Through structured programming 3T requires the definition of objects or classes, each performing a specific task in either the client or server workstation. 3T classes are not the same as classes in object-oriented programming. However, they are considered a step in that direction.

This document explains:

- How to set up a Windows workstation for the development of client programs with a graphical user interface using Visual Basic.
- How to set up a Windows workstation in a production environment.
- How to set up an AIX machine with additional software for the development of 3T server applications.
- How to integrate existing MQSeries programs into a Three Tier application.
- How to design, write, and test a complex client/server application.

During the course of this book two example applications are developed:

The first example is an existing file transfer application between two RS/6000 machines. It consists of two programs: sender and receiver. These programs are integrated into an MQSeries Three Tier application, together with a graphical user interface (GUI) written in Visual Basic. The GUI, running in the client under Windows Version 3.1, invokes the file transfer between the two RS/6000. This example explains the options for invoking existing programs from a GUI.

The second example is more complex and demonstrates the 3T development facilities. Step by step, an application with several classes is designed, written and tested. This example demonstrates how to make a solid application design and what the 3T infrastructure saves you on writing code.

With this publication a diskette is provided that contains the code for the various development steps for the two examples. The readers can execute each step in their own environment.

How This Document is Organized

The document is organized as follows:

- Chapter 1, “IBM MQSeries Three Tier Overview”

This chapter provides a brief overview of MQSeries Three Tier for AIX and MQSeries Three Tier for OS/2. Both are part of the MQSeries set of products. This chapter covers the concepts and facilities of 3T and introduces you to the application model that 3T supports.

- Chapter 2, “Installation”

This chapter describes what software is required for 3T clients and servers, how to install it, and how to verify the connection between clients and servers. It discusses how to add the 3T software to an AIX machine and how to use it for the development of 3T applications. It describes how to install all software required for a client development system and client production workstation.

- Chapter 3, “Using Visual Basic”

This chapter describes the 3T Visual Basic support. It also outlines how to run the sample programs supplied with the Visual Basic support for Windows clients.

- Chapter 4, “File Transfer Example”

This chapter describes how an existing AIX file transfer application, consisting of a sender and a receiver program, is integrated into a Three Tier application. First, a GUI is added that allows a Windows client to start the file transfer between two RS/6000 systems. Then the AIX sender program is modified to use 3T APIs instead of MQI APIs. This allows the reader to compare a 3T application with a non-3T MQSeries application.

- Chapter 5, “The Bacon Lettuce and Tomato Sandwich”

In this chapter we design an application that consists of several classes that run in three tiers. The clients run under Windows Version 3.1. The application requires several client windows that can run on one or more client workstations. As a server we use an AIX machine. Several business logic programs are developed. They may run on one or more AIX systems. The third tier is a program that uses MQI APIs and runs on an AIX machine as well. This application takes into account messages that are late or never arrive.

- Chapter 6, “Data Conversion”

This chapter explains how data is converted when messages are exchanged between PCs and AIX machines. A data conversion program is provided.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document.

- *MQSeries Clients*, SC33-1632-01
- *MQSeries Command Reference*, SC33-1369
- *MQSeries Application Programming Reference*, SC33-1370
- *MQSeries Installation and System Management Guide*, SC33-1371

- *MQSeries Three Tier Administration Guide*, SC33-1451
- *MQSeries Three Tier Application Design*, SC33-1636
- *MQSeries Three Tier Application Programming*, SC33-1452
- *MQSeries Three Tier Reference Summary*, SX33-6098
- *Messaging & Queuing Using the MQI, McGraw-Hill Series on Computer Communications*, SR28-5857

International Technical Support Organization Publications

- *TCP/IP Tutorial and Technical Overview*, GG24-3376
- *Examples of Using MQSeries on S/390, RISC System/6000, AS/400 and PS/2*, GG24-4326
- *IBM MQSeries Three Tier for OS/2, Experiments and Experiences for Beginners*, SG24-4509

A complete list of International Technical Support Organization publications, known as redbooks, with a brief description of each, may be found in:

International Technical Support Organization Bibliography of Redbooks, GG24-3070.

How Customers Can Get Redbooks and Other ITSO Deliverables

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **IBMLINK**

Registered customers have access to PUBORDER to order hardcopy, to REDPRINT to obtain BookManager BOOKs

- **IBM Bookshop** — send orders to:

usib6fpl@ibmmail.com (USA)
bookshop@dk.ibm.com (Outside USA)

- **Telephone orders**

1-800-879-2755	Toll free, United States only
(45) 4810-1500	Long-distance charge to Denmark, answered in English
(45) 4810-1200	long-distance charge to Denmark, answered in French
(45) 4810-1000	long-distance charge to Denmark, answered in German
(45) 4810-1600	long-distance charge to Denmark, answered in Italian
(45) 4810-1100	long-distance charge to Denmark, answered in Spanish

- **Mail Orders** — send orders to:

IBM Publications
P.O. Box 9046
Boulder, CO 80301-9191
USA

IBM Direct Services
Sortemosevej 21,
3450 Allerod
Denmark

- **Fax** — send orders to:

1-800-445-9269
45-4814-2207

toll-free, United States only
long distance to Denmark

- **1-800-IBM-4FAX (USA only)** — ask for:
 - Index # 4421 Abstracts of new redbooks
 - Index # 4422 IBM redbooks
 - Index # 4420 Redbooks for last six months
- **Direct Services**
 - Send note to softwareshop@vnet.ibm.com
- **Redbooks Home Page on the World Wide Web**
 - <http://www.redbooks.ibm.com/redbooks>
- **E-mail (Internet)**
 - Send note to redbook@vnet.ibm.com
- **Internet Listserver**
 - With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibm.com with the keyword `subscribe` in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How IBM Employees Can Get Redbooks and ITSO Deliverables

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** — to order hardcopies in USA
- **GOPHER link to the Internet**
 - Type GOPHER
 - Select IBM GOPHER SERVERS
 - Select ITSO GOPHER SERVER for Redbooks
- **Tools disks**
 - To get LIST3820s of redbooks, type one of the following commands:
 - TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET GG24xxxx PACKAGE
 - TOOLS SENDTO CANVM2 TOOLS REDPRINT GET GG24xxxx PACKAGE (Canadian users only)
 - To get lists of redbooks:
 - TOOLS SENDTO WTSCPOK TOOLS REDBOOKS GET REDBOOKS CATALOG
 - TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
 - TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET LISTSERV PACKAGE
 - To register for information on workshops, residencies, and redbooks:
 - TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
 - For a list of product area specialists in the ITSO:
 - TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
- **Redbooks Home Page on the World Wide Web**
 - <http://w3.itso.ibm.com/redbooks/redbooks.html>
- **ITSO4USA category on INEWS**
- **IBM Bookshop** — send orders to:
 - USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL

- **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibm.link.ibm.com with the keyword `subscribe` in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

Acknowledgments

This project was designed and managed by:

Dieter Wackerow
International Technical Support Organization, Raleigh Center

The authors of this document are:

George Carey
IBM USA

Claudia Degli Esposti
IBM Italy

Simon Miller
IBM Hursley, England

Dieter Wackerow
International Technical Support Organization, Raleigh Center

This publication is the result of a residency conducted at the International Technical Support Organization, Raleigh Center.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Paul Beaven
John Kelly
Ian McCallion
IBM Hursley

Chapter 1. IBM MQSeries Three Tier Overview

IBM MQSeries Three Tier for OS/2 and IBM MQSeries Three Tier for AIX are software products, designed by IBM for the message queuing environment. They are called 3T for short. The products use MQSeries for OS/2 and MQSeries for AIX, respectively, to send and receive messages.

IBM MQSeries Three Tier lets you build applications which access departmental data, enterprise data, and inter-enterprise data, residing on many different systems using the MQSeries Interface. The product helps the user to write applications that give useful and timely responses even when some systems are temporarily not available.

IBM MQSeries Three Tier will reduce cost and risk developing MQSeries distributed applications. It provides a designed way to write client/server applications on an MQSeries backbone. It aids in creating scalable and manageable applications that can easily be deployed across the enterprise.

Programmers can concentrate on business applications instead of spending time on writing "system code" that retrieves messages from queues and decides which piece of "application code" to schedule. In MQSeries, a program sends a message to a queue and other programs must provide the code that retrieves it. 3T, however, provides true application to application connectivity.

IBM MQSeries Three Tier for OS/2 requires a graphical user interface (GUI) to interact with the user. IBM MQSeries Three Tier for AIX runs in servers only. There is no front end for AIX machines. However, 3T's development facilities apply to both OS/2 and AIX.

Note: For the project covered in this publication we use Windows clients. The GUIs are developed using Visual Basic.

IBM MQSeries Three Tier enhances the functions of MQSeries. It provides:

- Advanced program triggering and message management, together with an enhanced application model
- Enhanced GUI integration; links to GUI programming tools, such as IBM VisualAge(TM) and Microsoft Visual Basic
- Application development tools

The 3T product consists of:

- Development components, such as:
 - Class Compiler
 - Application Simulator
 - Test Harness
- Run-time components for:
 - OS/2 clients (called Presentation Logic Manager)
 - Windows clients (called Presentation Logic Manager)
 - OS/2 servers (called Business Logic Manager)
 - AIX servers (called Business Logic Manager)

- Service tools
 - Service Level Utility
 - Error Log Browser
 - Trace utilities
- Sample programs written in C, COBOL, PL/I, VisualAge

Note: Visual Basic support and sample programs are available as a SupportPac.

1.1 The Three Tiers

The IBM MQSeries Three Tier for OS/2 and IBM MQSeries Three Tier for AIX products were developed to make it easier for the application developer to write client/server applications. As the names indicate, the products allow you to divide the application into three tiers. They are called:

Presentation Logic (PL)

Business Logic (BL)

Data Logic (DL)

Each logic has a distinct purpose. A tier is further subdivided into classes or objects, each representing a specific piece of work.

The following explains the purpose of the three tiers:

- Presentation Logic (PL)

This is the front end of the application, a graphical user interface that collects data from the user and displays information for the user. Its purpose is to interact with the end user and to request services from one or more servers and/or host systems. However, a PL is not restricted to those tasks. It can perform any kind of processing, more than the validation of input data. A PL can simultaneously send several requests to different servers.

Note: Presentation Logics do not run on AIX machines.

- Business Logic (BL)

The second tier is usually running on a server. It processes data on behalf of the PL client and may require the services of other BLs, residing in the same or other server machines. A BL may request additional services from a host. You can have as many BLs as you want in this second tier, each processing a certain request, such as calculating interest, obtaining customer data, or updating an account.

- Data Logic (DL)

Usually, the third tier is a host program that obtains data requested by either a BL or PL from a database. However, a DL does not have to be a host program; it may run in any server machine and even in the end user's workstation. DLs do not use 3T but do use MQSeries. Therefore, they can run on systems that have MQSeries but not 3T installed. This feature allows you to include existing MQSeries programs in your application, such as CICS or IMS applications.

The difference between BL and DL is that a DL processes messages without a 3T header. This enables 3T classes to talk to MQSeries-based programs that have been written without 3T, but use MQSeries APIs.

1.1.1 Run-Time Components

Presentation Logics and Business Logics are managed by the run-time programs supplied with 3T:

- PLM: Presentation Logic Manager (for OS/2 only)
- BLM: Business Logic Manager (for OS/2 and AIX)

These programs act as an interface between PLs and BLs, and MQSeries. Applications put messages on one or more queues. PLMs and BLMs ensure that those messages are routed to the correct destination, client or server, and that the appropriate application program is scheduled.

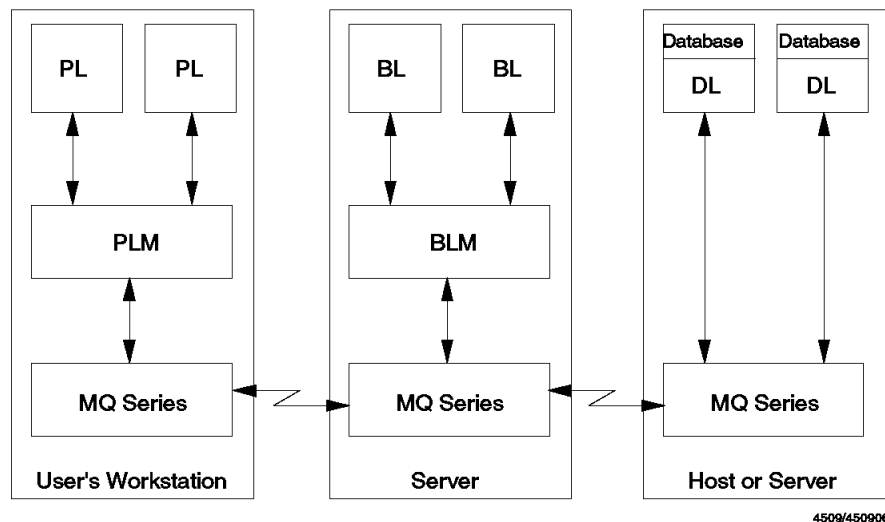


Figure 1. The Three Tier Run-Time Components

- Presentation Logic Manager

The 3T PLM provides the client functions of MQSeries 3T:

- Sending and receiving MQSeries messages
- Starting up programs containing a PL

If a message arrives at a workstation that should cause a new window to be displayed, then the PLM responds by starting the program that implements that window if it is not already running.

- Pre-allocates MQSeries resources and shares them between PL programs
- Provides 3T APIs for PLMs
- Working storage management
- Error handling and cleanup

For practical reasons the PLM normally uses the MQSeries Client to access an MQSeries MQM running on a server machine.

- Business Logic Manager

The 3T BLM provides the server functions of MQSeries 3T:

- Sending and receiving MQSeries messages

- Scheduling of the appropriate methods (programs) to handle received messages
- Passing of incoming messages, instance data and other information to the methods (programs) it schedules
- Provides 3T APIs for BLMs
- Working storage management
- Error handling and cleanup

1.1.2 Messages

Messages sent between PLs and BLs contain 3T headers. Messages sent to or received from a DL do not have that header. They are automatically added by the Presentation Logic Manager (PLM) and Business Logic Manager (BLM). 3T headers are removed when a message is sent to a DL.

There are three types of messages:

- *Inform messages* are sent when the sender does not expect a response back, such as broadcast messages.
- *Request messages* are sent when the receiving class has to perform some work on behalf of the sender and expects a response back. Usually, this job is executed in a server or host machine. Each request must have a response associated with it.
- *Response messages* carry the result of the requested work back to the requestor. The receiver of a request message must respond with a reply message. There is a one-to-one relationship between requests and replies; 3T does not allow multiple reply messages to respond to one request message.

Note: You can send several requests to different objects. 3T takes care of receiving the multiple replies and triggering the appropriate program to process them. You could schedule one program in the case all replies arrive within a specified time. Another program could be invoked when only some of the replies arrived in time. You can also trigger programs for each late reply, if you wish.

Each 3T message has a name. 3T requires that you create for each PL, BL, and DL a special file, a *class source file*, that describes which messages can be sent and received by each class. Class files are written using 3T's *Class Definition Language* (CDL). The product provides a function that reads all class files and crosschecks the message flow between classes. This ensures, for example, that each request has a response.

3T messages can be of fixed length or variable length. You define the structures of a fixed length message in a special *structure file*. The class compiler of 3T checks whether or not each fixed length message is defined.

Variable length messages are assembled in *sets*. The fields in a variable length message (set) are called elements. 3T provides API calls to one of its features, the *Self-defining Data Manager* (SDDM), to work with elements in a set.

3T provides the option to save (known as "harden") messages on a hard disk to ensure that data is delivered in case of a failure.

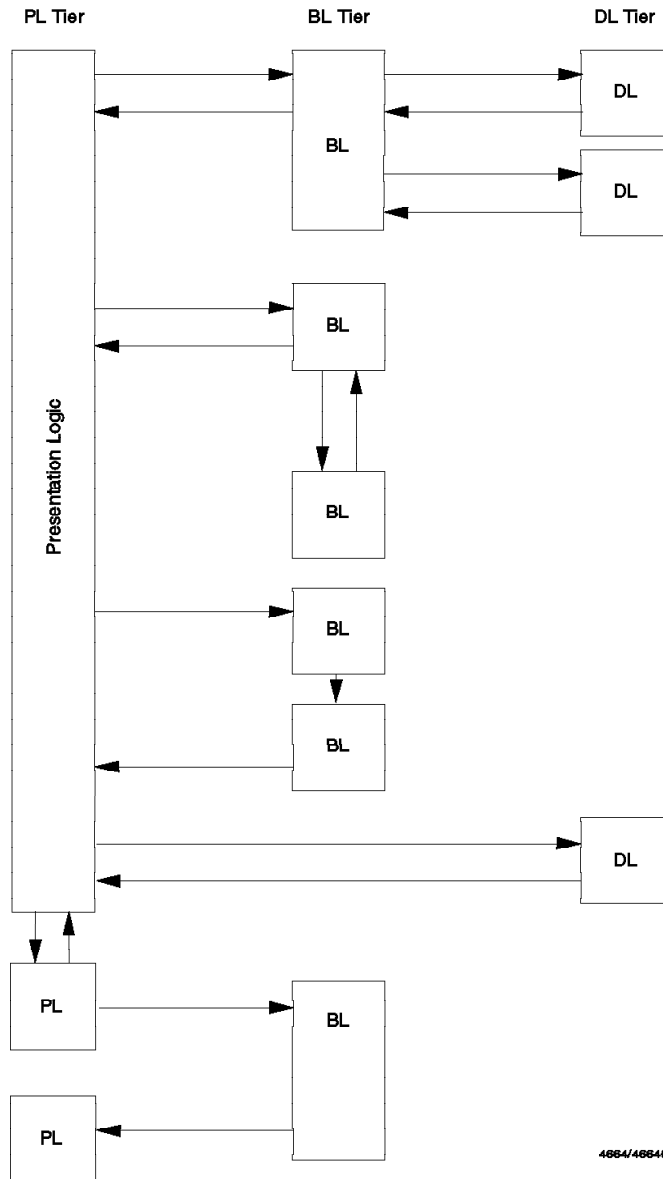


Figure 2. The Three Tiers of 3T

1.1.3 Message Flow

Figure 2 shows the way messages can flow between 3T classes. The classes can reside in the same machine (for example, the end user's OS/2 workstation), in different servers, or in main frames.

A PL can communicate with many BLs and DLs, and also with other PLs. A BL can invoke other BLs and/or DLs and PLs.

3T allows you to set up an application development environment in a local workstation without actually sending messages over the network.

Each class (PL, BL, and DL) can be regarded as an object. The number of classes (or objects) and messages in an application are not limited. This allows you to construct applications consisting of many small programs, each of them performing a specific task. You could write one BL for each message the PL sends, or have one BL processing all of the messages.

In most cases, each message has to be processed differently, you have to code a specific function for each message. In 3T, these functions or programs are known as *methods*.

At run-time, when a message arrives for a class, the PLM or BLM checks whether the class is allowed to receive it, and under what conditions. In 3T, those conditions are called *rules*. In the class file, you can define a set of rules for each message.

1.2 3T Facilities

3T is MQSeries. 3T provides features that aid in the design and implementation of applications. 3T makes application programming simple. When a program (class) sends a message to another program (class), the message is put on a queue. All the programmer has to know is the name of the queue; that is, the same name as the class. The PLM or BLM does the rest of the message routing. These 3T run-time services automatically trigger the program that has to receive and process the message.

Note: You must install MQSeries on your workstation.

1.2.1 Class Definition Compiler

For a 3T application, you must create a class source file. You use the Class Definition Language (CDL) to write it. The class compiler compiles the source file and creates a binary class file. This binary file is used during run time by the presentation and business logic managers. You need one class file for each class.

A class file contains the following information:

- The names and descriptions of all *messages* sent and received by the class. Usually, message descriptions are kept in a separate file, since they are referenced in at least two classes, sender and receiver. Such a message header file is then included in the class files.
- *Class descriptions* contain the external attributes of a class, such as class type (PL, BL, or DL), the names of the messages it can send and receive, and whether it can recover from a server failure. A class file must contain a class description of all the classes it communicates with. Therefore, it is advisable to place this information in a separate file and include it in the class files.
- The *class section* describes the specifics of the class the class source file is written for, such as the names of the classes it can send messages to, and the name of the file that contains message structures. Also, this section includes rules for all the incoming messages.
- *Rules* define what to do when a message arrives. There must be a rule for each message. A rule can be satisfied when one message arrives (on time or late), or when it doesn't arrive. Also, a rule can be satisfied when some,

none or all replies arrive (in time, late, or never). Each rule is associated with a method to process the event.

- *Methods* describe what program to invoke when a rule is satisfied. These programs are BLs or PLs, written in C, COBOL or PL/I.

Note: You cannot write a class file for a DL class since DLs are not 3T programs. However, you must define the DLs you request services from in the *class definition sections* of the appropriate class.

After the application design is completed, use the CDL to define the classes and the message flow between them. At that time you also know how many methods or programs are required to process the data and what their names are.

The class compiler provides three functions:

1. It compiles user-written class source files and creates class binary files for 3T's run-time programs, PLM and BLM.
2. It provides a *crosscheck function* that validates your overall application design by crosschecking all class source files. It checks whether:
 - A message has a destination.
 - There is a method to process the message.
 - A class receives the messages.
 - A request has a reply associated with it.
3. It creates the following files for each business logic:
 - A language-dependent skeleton (C, COBOL, PL/I) for each method
 - A make file (.MAK) for the compiler
 - A definition file (.DEF) for the compiler

Note: Use of the class compiler is described in detail in the following chapters.

1.2.2 Application Program Interface

3T contains a set of APIs for three languages:

- COBOL
- C
- PL/I

In your application programs use the APIs to call 3T functions of the PLM or BLM. These programs make the necessary calls to the underlying MQI.

There are three categories of API calls:

- *Base calls* (17) are used to:
 - Send requests, replies and information messages
 - Write to the log file
 - Set timeout values
 - Query information about a message, a class, a server, or an instance
- *PLM calls* (7) are used to:
 - Glue 3T to the GUI of a PL
 - Query information about an event

- Set the state of an instance
- End the Presentation Logic Manager
- *SDDM calls* (17) are used to invoke functions of the *Self-Defining Data Manager* (SDDM). These calls are used to manipulate variable-format data.

1.2.3 Application Simulator

The application simulator is for IBM MQSeries Three Tier for OS/2 only.

This feature allows:

- Selected parts of the application to be modelled prior to any code being written
- To validate the message flow
- To determine the likely system performance by allowing the system to be loaded artificially

Note: The Application Simulator is not covered in this publication.

1.2.4 Test Harness

The test harness simplifies testing of MQSeries applications by allowing each piece to be tested separately. This development tool speeds up thorough testing of individual application components. It can dramatically reduce development time.

Note: The Test Harness is not covered in this publication.

1.2.5 Job Viewer

A 3T job is the execution of a 3T application. Typically, an application has several windows on the screen at the same time. The Job Viewer is implemented as a 3T class and it:

- Provides a window list of all the windows in the job
- Helps in selecting the window needed to accomplish a task
- Can be used to minimize and maximize windows

The Job Viewer is the first PL started up in an application.

Note: The Job Viewer is not covered in this publication.

1.2.6 Self-Defining Data Manager

The SDDM provides a simple way for the application programmer to build and manipulate the data of a message. Typically, a method is required to unpack a received message, then perhaps using its instance data together with the unpacked data, to build a new message; this is essentially a re-packing operation.

The SDDM provides a set of APIs to manage variable and fixed length messages. The SDDM creates a *set* for each message. The set contains *elements* that are data plus header information containing an element identifier, the length of the element, and the data type (character string or integer). Data must be in multiples of four bytes.

Note: You may also define the structure of a fixed length message in a *structure file* and your application may refer to the fields defined in it.

Note: Some of the SDDM APIs are covered in the following chapters.

1.2.7 Visual Basic Support

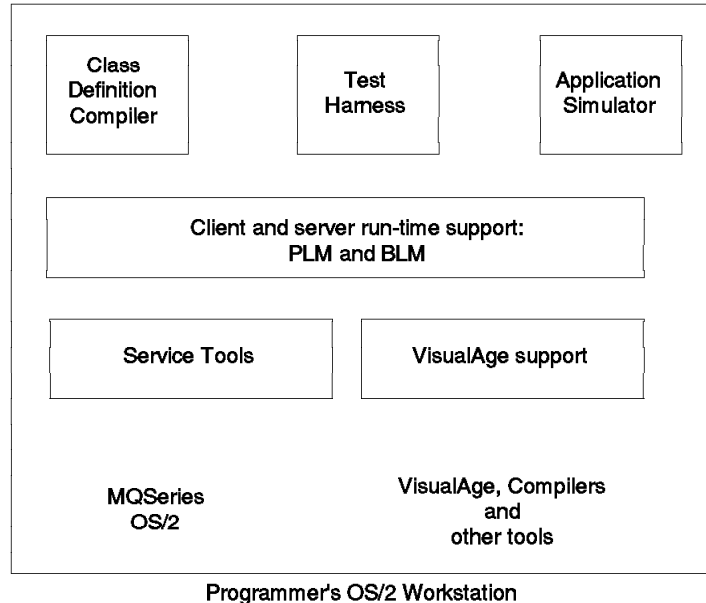
MQSeries 3T Visual Basic support for Windows client is provided as a SupportPac. It allows you to use Microsoft's Visual Basic product to develop Microsoft Windows 3.1 GUIs for IBM MQSeries Three Tier applications.

You develop presentation logics on the Windows system. Business logics and class definitions must be developed on either an OS/2 or AIX machine.

Note: How to use the Visual Basic support for 3T is described in Chapter 3, "Using Visual Basic" on page 43.

1.3 Application Design

The figure below shows what could be installed in an application developer's workstation:

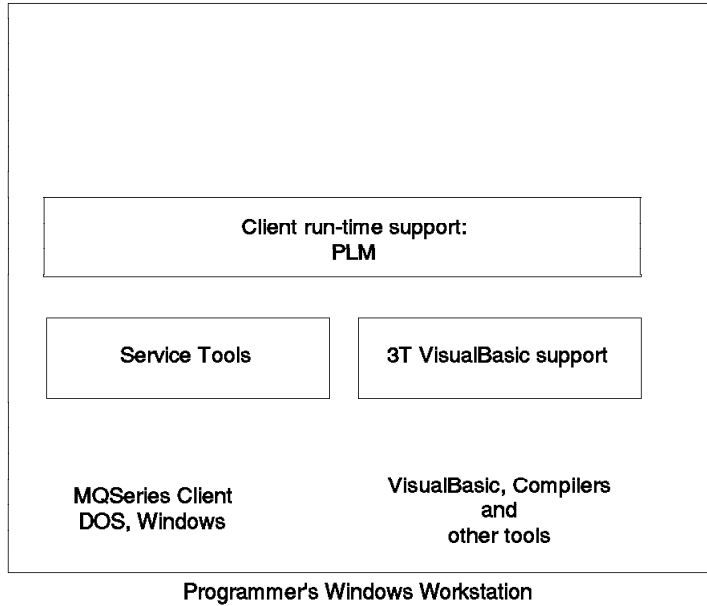


4864488402

Figure 3. Programmer's OS/2 Workstation

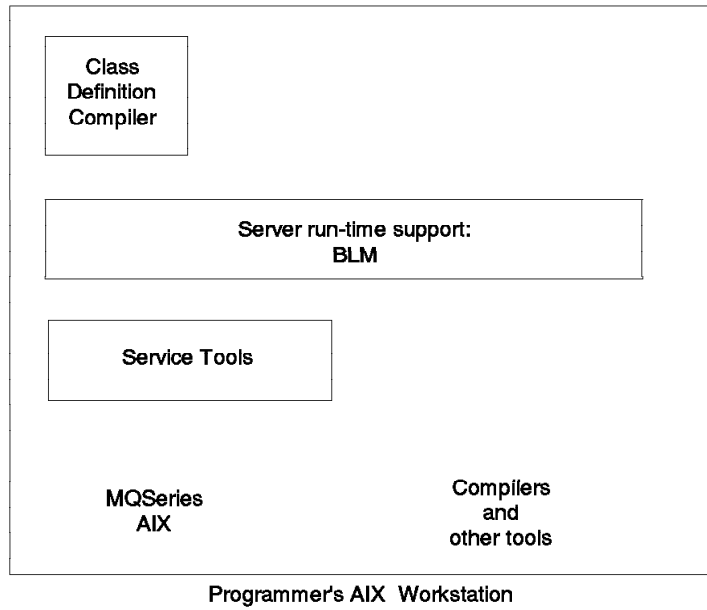
Notes:

1. Components shown boxed are supplied with 3T.
2. Test Harness and Application Simulator run under OS/2 only.
3. The job viewer runs under OS/2 and Windows.



4864486403

Figure 4. Programmer's Windows Workstation



4864486404

Figure 5. Programmer's AIX Workstation

1.3.1 The 3T Application Model

The IBM MQSeries Three Tier application model is a distributed model. Distributed applications are constructed from fragments of an application code. In 3T, these programs are called classes. The classes that together make up the application, can (and generally will) exist on different machines.

Most communication between classes is carried by MQSeries messaging. Named messages flow from one class to another.

One 3T class addresses another 3T class directly. Classes are not aware of the underlying queues or queueing mechanisms.

The distributed application design is defined in class definition files (one for each class). The files contain descriptions of classes, methods, and messages that flow between classes, and the rules for invoking methods.

3T knows three forms of classes:

- PL class (Presentation Logic)
- BL class (Business Logic)
- DL class (Data Logic)

The model contains the concept of a Data Logic class. This can be an existing application running on a platform that is accessible via MQSeries messaging. 3T has no code supporting Data Logic; the purpose of including Data Logic in the model is one of completeness.

Each class definition is compiled to produce a binary class file which is used by the 3T run-time components (PLM and BLM) during testing and execution.

BL classes are written in Cobol or C. PL classes are created using a GUI generator tool or C.

Classes that make up an application can run on machines in different locations.

A class contains one or more *methods*. Methods are small, self-contained pieces of code.

The flow of activity between classes is achieved using 3T events, where an event can be:

- A message arrival (at a PL or BL class) initiated by a MQSEND application programming interface (API) call in another class (method).
- A PL (GUI) event initiated by the user.
- A timeout maturing, which was originated by a MQTIME API call in the same class where it will be processed.

Note: The term 3T message refers to a message generated by a 3T application. This is a particular type of MQSeries message and uses standard MQSeries products to convey it to the destination class. The translation of class names to queue names is done by 3T.

When an event occurs, the appropriate method is scheduled by 3T. This is done by the 3T PLM on a workstation, or the 3T BLM on an application server machine. 3T makes the decision of what method to run based on three things:

- The class binary file (specifies conditions for scheduling methods)

- The content of the class' input queue
- The instance data for the class

When a method is scheduled it is given the following data:

- The content of the message received that caused the method to be scheduled
- The instance data

The method runs and can issue API calls to send messages to other classes or to log data, generate timeouts, etc.

When the method is finished, control is returned to the 3T PLM or 3T BLM. The class is only activated again if an appropriate 3T event occurs.

1.3.2 The 3T Application Development Process

The following outlines a standard process for the development of 3T applications:

1. The customer defines the requirements for his application, such as:
 - Business requirements. Some of them are:
 - High productivity GUI interfaces with multi-window capability.
 - Event driven in response to a business activity (for example, customer arrival, cross-selling opportunity).
 - Access to multiple environments (for example, CICS, IMS, DB2).
 - Access data from other organizations (for example, credit bureau, airline).
 - Ability to cope with partial network failure (for example, a host or server does not respond for any reason).
 - Ability to detect cross-selling opportunities.
 - Allow the user to switch tasks (avoid keeping the customer waiting; invoke another transaction).
 - Type all data into a GUI once and use it to request services from multiple applications (classes) on multiple platforms.
 - Application execution requirements. Some of them are:
 - Multiple simultaneous use of business logics
 - Modular reusable GUIs, presentation logics and business logics
 - Automatic late message processing
 - Ability to make multiple non-blocking parallel requests
 - Application development requirements. Some of them are:
 - Having an application model
 - Ability to use object-oriented or procedural design methods
 - Use of familiar languages
 - Compatibility with existing 4G languages
 - Suitability for large scale application production
 - Application and testing tools

- Systems management requirements. Some of them are:
 - Ability to deploy
 - Version control
 - Operational control
 - Security
- 2. Based on the requirements gathered, the customer:
 - Analyses requirements for the application
 - Produces specifications that reflect the customer's requirements
 - Makes a high level design (HLD)
- 3. The designer maps the HLD into MQSeries 3T by writing class definitions, using his favorite editor. 3T provides the Class Definition Language (CDL) to name:
 - Classes
 - Messages that flow between classes
 - Triggering rules that schedule the methods in the classes
- 4. The designer checks the definition for consistency using the 3T class compiler.
- 5. The designer uses the Application Simulator to investigate the likely performance of the finished application.
- 6. The GUI programmer uses the HLD and class definitions to design the GUIs. These are built by:
 - Using a visual programming tool such as VisualAge
 - Writing a PM program
 - Writing a Windows program
- 7. The BL programmer uses HLD and class definitions to design and build the Business Logics.
- 8. The DL programmer uses HLD and class definitions to design and build the Data Logics. This may be existing applications that require modification for the access by MQSeries.
- 9. The GUI and BL programmers use the Test Harness to debug their code.
- 10. The various parts of the application are now assembled on a single machine and debugged in combination.
- 11. The parts are copied to several machines and tested.
- 12. A system test is conducted.
- 13. A performance test is conducted.
- 14. The development library data is moved into a production library in anticipation of deployment.
- 15. A pilot test is done on life data.
- 16. A deployment test is conducted.
- 17. The application is deployed or phased in.

1.3.3 Structured Application Design

IBM MQSeries Three Tier for OS/2 encourages object-oriented programming. 3T objects are not objects as in OOA/OOD; however, they are a step in that direction.

First define what work shall be done in the three tiers:

- The PL is the front end of the application. Usually, there is one PL per application. Many instances of the PL can run in the same or different workstations.
- Typically, the BL is used to manipulate data. A BL does not interact with the user. However, you can log data to a file or display data on the Business Logic Manager's screen.

A BL receives requests from the PL or from other BLs. In order to perform its work, the BL can request services from a DL or another BL. A BL can be written for any platform that supports 3T.

- DLs are 3T classes; however, they run on platforms not supported by 3T. DLs are written using MQSeries APIs. DLs can be IMS or CICS programs running in the host or any program not using 3T APIs.

The second step is to define the class topology. Usually, the work assigned to the BL and DL tiers can be broken into functions, each function performing a specific task, such as validating input data, reading an address file, or updating an account.

For each function we write a separate program, called a *method*. A method processes one or more named messages or requests. A class can contain one or more methods.

By structuring the application, defining classes and messages, you can picture the connections between the various objects. 3T supplies the CDL to write down the class topology in the class source files. You can use the class compiler to crosscheck the message flow between classes.

Diskette

This publication comes with two diskettes. They contain the source code and all files necessary to run the two applications described in this book.

Chapter 2. Installation

This chapter describes the installation of the software needed to develop and run MQSeries Three Tier applications on AIX servers and Windows clients.

For this project we use four systems:

1. An AIX server for development

This system is used to develop business logic for the server and class files for the server and the Windows workstations.

2. An AIX server for test

This system is used as the receiver in the file transfer example between two AIX machines.

3. A Windows development workstation

This machine is used to develop and test presentation logic for Windows Version 3.1. The GUIs are created with Visual Basic Version 3.

4. A Windows production workstation

This machine is used to test the sample applications developed through the course of this book. We install only the software required for a production environment.

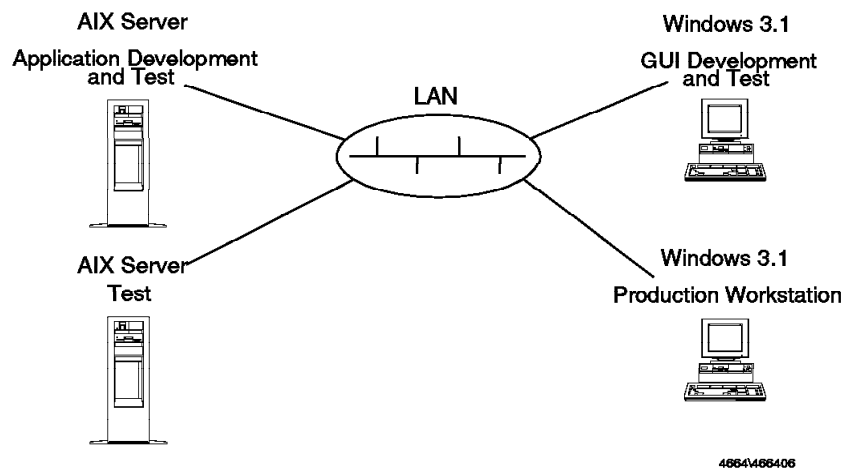


Figure 6. Project's Configuration

The following sections guide you through the installation process for configuration in Figure 6.

2.1 AIX Server

The AIX server is used to develop most of the 3T application. The following software is required:

- AIX Version 3.2.5 (base product)
- MQSeries for AIX Version 2.2
- IBM DCE Threads for AIX Version 1.1
- CSD 14 for MQSeries
- MQSeries Three Tier for AIX
- IBM CSet++ for AIX/6000 Version 2

It is assumed that you have an AIX server with all the software but MQSeries and MQSeries Three Tier installed.

2.1.1 MQSeries Base Product

As a pre-requisite to the use of MQSeries 3T running on any platform, the base MQSeries software product must be installed and running on the same target platform, MQSeries server for server functionality or MQSeries client for client functionality.

The complete installation procedure is given in the MQSeries publication *IBM MQSeries for AIX Version 2 Release 1 System Management Guide*, SC33-1373-00. Pages 19 through 27 deal specifically with this process and give a detailed description. A synopsis version follows with sizing and procedural suggestions.

Create a user and group called *mqm*. You may do this using SMIT as described in the *System Management Guide* page 21 or you may enter the commands directly as follows:

```
mkuser mqm
mkgroup mqm
```

Make *root* a member of the *mqm* group. Using SMIT for this is best.

Note: You will also have to create a user and group *mq3t* to install the MQSeries Three Tier product and make the *mq3t* user a part of the *mqm* group as well. You might wish to do this now as well or you can wait until you install 3T.

Now pick the device from which you are going to install the base MQSeries product:

- If you install from a tape insert the tape in the appropriate tape drive.
- If you install from a network software server be sure the appropriate software server file system is mounted.

To better illustrate, the following is an example of the commands to execute:

If the network file server, NFS, is a node called "earth" and the directory which contains the install image has the path `/usr/AIX325C` then the standard mount command to use so that SMIT will execute in its normal fashion to install the MQSeries Licensed Program Product (lpp) is as follows:

```
mount earth:/usr/AIX325C /usr/sys/inst.images
```

Note: Your system must be known to the file server system; that is, its name must be in the /etc/hosts file on the node "earth" in order for this to work.

You can verify that the mount command worked properly by executing the df command after which you should see the following line as part of the file systems displayed.

```
earth:/usr/AIX325C 409600 317724 22% - - /usr/sys/inst.images
```

Now to actually install the base MQSeries product use SMIT and select the menus' items in the order illustrated in the following cascading list of selections.

Software Installation & Maintenance

Install / Update Software

Install / Update Selectable Software (Custom Install)

Install Software Products at Latest Available Level

After pressing the Enter key on the final menu item selection the screen in Figure 7 should appear.

Note: This is a none Motif GUI screen image.

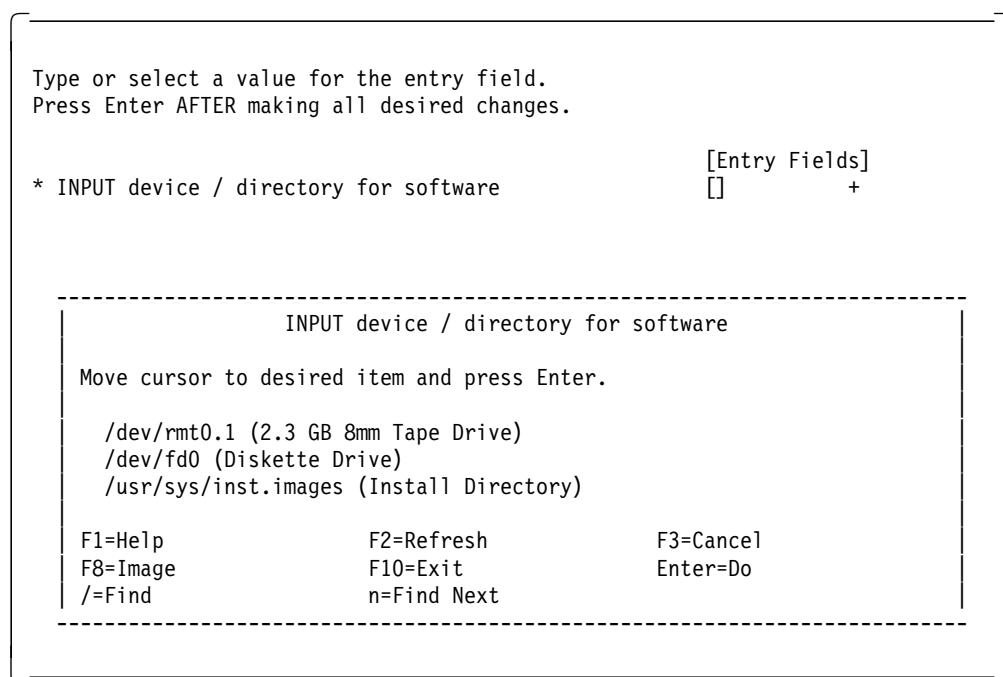


Figure 7. MQSeries for AIX Installation, Device Selection

Select the device to be used for input by pressing the PF4 key to get to the bottom half of the screen illustrated or type directly between the square brackets the device to be used. For example,

- For a tape type /dev/rmt0.1.
- If it is a network installed image accessed via the example mount command, type /usr/sys/inst.images.

In either case, after the device selection is made press the Enter key to proceed. Then the screen in Figure 8 on page 18 appears.

```

                                Install Software Products at Latest Available Level

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                [Entry Fields]
* INPUT device / directory for software      /usr/sys/inst.images
* SOFTWARE to install                        [2.2.0.0  mqm          > +
Automatically install PREREQUISITE software?  yes                +
COMMIT software?                             yes                +
SAVE replaced files?                         no                 +
VERIFY Software?                             no                 +
EXTEND file systems if space needed?         yes                +
REMOVE input file after installation?        no                 +
OVERWRITE existing version?                 no                 +
ALTERNATE save directory                     []

F1=Help          F2=Refresh      F3=Cancel      F4=List
F5=Reset         F6=Command     F7=Edit       F8=Image
F9=Shell        F10=Exit       Enter=Do

```

Figure 8. MQSeries for AIX Installation

In the area labeled "SOFTWARE to install" you can:

- Press the PF4 key again to get a complete list of the software modules that can be installed in the MQSeries LPP.

Then select the option **1.2.2.0.0 mqm ALL** by pressing the PF7 key when the cursor is positioned to the line (it will appear highlighted).
- Type in the selection manually but it must exactly match the LPP name.

Using the PF4 key and then PF7 key to select is best.

The complete list of LPPs for the base MQSeries product is given below. If you are tight on disk space or do not wish to clutter your disk file systems with files that likely will never be used one would individually select only those language modules that would be used. Once you custom select one LPP item you must custom select all your LPP items. So either select the "ALL" option or select all the individual items required.

```

2.2.0.0  mqm                                ALL
        2.2.0.0  mqm.De_DE
        2.2.0.0  mqm.Es_ES
        2.2.0.0  mqm.Fr_FR
        2.2.0.0  mqm.Ja_JP
        2.2.0.0  mqm.aix_client
        2.2.0.0  mqm.base
        2.2.0.0  mqm.books
        2.2.0.0  mqm.dos_client
        2.2.0.0  mqm.info
        2.2.0.0  mqm.os2_client
        2.2.0.0  mqm.samples
        2.2.0.0  mqm.server
        2.2.0.0  mqm.win_client

```

Once you have made your selection(s) press Enter and the installation process will begin. This may take some time so go have a coffee break or do a context switch to your next asynchronously performable task.

Note: The directory that will be expanded during this installation is /usr/lpp/mqm and the file system holding it will increase by approximately 35 MB.

The output from processing of this installation should look like the following:

```
---- start ----
installp -qacFNXd/usr/sys/inst.images \
        -f {File Containing Software} 2>&1

Contents of {File Containing Software}:
  mqm 2.2.0.0.all

Verifying requisites...done
Results...

SUCSESSES
-----
  Filesets listed in this section passed pre-installation verification
  and will be installed.

  Selected Filesets
  -----
  mqm.De_DE 2.2.0.0          # MQSeries Messages - German
  mqm.Es_ES 2.2.0.0          # MQSeries Messages - Spanish
  mqm.Fr_FR 2.2.0.0          # MQSeries Messages - French
  mqm.Ja_JP 2.2.0.0          # MQSeries Messages - Japanese
  mqm.aix_client 2.2.0.0     # MQSeries Client for AIX
  mqm.base 2.2.0.0           # MQSeries Base Kit for Client...
  mqm.books 2.2.0.0          # MQSeries BookManager Books
  mqm.dos_client 2.2.0.0     # MQSeries Client for DOS
  mqm.info 2.2.0.0           # MQSeries On-line Information
  mqm.os2_client 2.2.0.0     # MQSeries Client for OS/2
  mqm.samples 2.2.0.0        # MQSeries Samples
  mqm.server 2.2.0.0         # MQSeries Server
  mqm.win_client 2.2.0.0     # MQSeries Client for Windows

  << End of Success Section >>

The number of restored files is 2.

0503-376 installp: Applying software for the "usr" part of product
  mqm 2.2.0.0.
:
Restoring files, please wait.
:
The number of restored files is 210.
The files for package xxxxxx are being verified.
This may take several minutes, please wait.
:
:
0503-392 installp: The installation was SUCCESSFUL for the "root" part of the
  following software products:
  mqm.samples 2.2.0.0
  mqm.info 2.2.0.0
  mqm.books 2.2.0.0
  mqm.base 2.2.0.0
  mqm.win_client 2.2.0.0
  mqm.server 2.2.0.0
  mqm.os2_client 2.2.0.0
  mqm.dos_client 2.2.0.0
  mqm.aix_client 2.2.0.0
  mqm.Ja_JP 2.2.0.0
  mqm.Fr_FR 2.2.0.0
  mqm.Es_ES 2.2.0.0
  mqm.De_DE 2.2.0.0

Verifying requisites...done
Results...
```

SUCSESSES

Filesets listed in this section passed pre-commit verification and will be committed.

Selected Filesets

```
mqm.De_DE 2.2.0.0          # MQSeries Messages - German
mqm.Es_ES 2.2.0.0          # MQSeries Messages - Spanish
mqm.Fr_FR 2.2.0.0          # MQSeries Messages - French
mqm.Ja_JP 2.2.0.0          # MQSeries Messages - Japanese
mqm.aix_client 2.2.0.0      # MQSeries Client for AIX
mqm.base 2.2.0.0           # MQSeries Base Kit for Client...
mqm.books 2.2.0.0          # MQSeries BookManager Books
mqm.dos_client 2.2.0.0     # MQSeries Client for DOS
mqm.info 2.2.0.0           # MQSeries On-line Information
mqm.os2_client 2.2.0.0     # MQSeries Client for OS/2
mqm.samples 2.2.0.0        # MQSeries Samples
mqm.server 2.2.0.0         # MQSeries Server
mqm.win_client 2.2.0.0     # MQSeries Client for Windows
```

<< End of Success Section >>

0503-379 installp: Committing software for the "usr" part of product
mqm 2.2.0.0.

0503-400 installp: The commit operation was SUCCESSFUL for the "usr" part of
the following software products:
mqm.xxxxx 2.2.0.0

:

Finished processing all filesets. (Total time: 13 mins 43 secs).

Installation Summary

Name	Fix Id	Part	Event	Result	State
mqm.samples		USR	APPLY	SUCCESS	APPLIED
mqm.info		USR	APPLY	SUCCESS	APPLIED
mqm.books		USR	APPLY	SUCCESS	APPLIED
mqm.base		USR	APPLY	SUCCESS	APPLIED
mqm.win_client		USR	APPLY	SUCCESS	APPLIED
mqm.server		USR	APPLY	SUCCESS	APPLIED
mqm.os2_client		USR	APPLY	SUCCESS	APPLIED
mqm.dos_client		USR	APPLY	SUCCESS	APPLIED
mqm.aix_client		USR	APPLY	SUCCESS	APPLIED
mqm.Ja_JP		USR	APPLY	SUCCESS	APPLIED
mqm.Fr_FR		USR	APPLY	SUCCESS	APPLIED
mqm.Es_ES		USR	APPLY	SUCCESS	APPLIED
mqm.De_DE		USR	APPLY	SUCCESS	APPLIED
mqm.samples		ROOT	APPLY	SUCCESS	APPLIED
mqm.info		ROOT	APPLY	SUCCESS	APPLIED
mqm.books		ROOT	APPLY	SUCCESS	APPLIED
mqm.base		ROOT	APPLY	SUCCESS	APPLIED
mqm.win_client		ROOT	APPLY	SUCCESS	APPLIED
mqm.server		ROOT	APPLY	SUCCESS	APPLIED
mqm.os2_client		ROOT	APPLY	SUCCESS	APPLIED
mqm.dos_client		ROOT	APPLY	SUCCESS	APPLIED

Once the installation process completes the MQSeries queue manager(s) and queues and any other desired or default MQM objects must be created and configured. Refer to 2.1.3, "Creating MQM Objects" on page 22 for more information.

Also, you may have to install updates to the base product. When this book was written the correctional service diskette (CSD) 14 had to be applied for the use of MQSeries 3T. Refer to 2.1.2, "CSD 14 for MQSeries" on page 21 for more information.

2.1.2 CSD 14 for MQSeries

The corrective service diskette (CSD) must be applied for 3T. Download the CSD14 update file onto your AIX system into an appropriately sized directory with sufficient available disk space (about 8.5 MB). In the example the directory is /usr/csd. Remember, per the CSD update notice after binary transferring the file to disk, rename it to csd14.objbin.

Use the *SMIT menu selections* to install the updates. The following SMIT menu selections show the path and directory entry to make. Change the target directory, of course, if you loaded the file into a different target directory than the example.

Software Installation & Maintenance

Install / Update Software

Install ALL Software Updates on Installation Media

INPUT device / directory for software [/usr/csd/csd14.objbin]

The ending output from the update assuming your system was not previously updated would have output that looks like the following:

```
Starting SMIT
(Menu screen selected,
  FastPath = "top_menu",
  id_seq_num = "0",
  next_id = "top_menu",
  title = "System Management".)
(Menu screen selected,
  FastPath = "install",
  id_seq_num = "010",
  next_id = "install",
  title = "Software Installation & Maintenance".)
(Menu screen selected,
  FastPath = "install_update",
  id_seq_num = "010",
  next_id = "install_update",
  title = "Install / Update Software".)
(Selector screen selected,
  FastPath = "install_all",
  id = "install_all",
  next_id = "install_all.cmd_header",
  title = "Install ALL Software Updates On Installation Media".)
(Dialogue screen selected,
  FastPath = "install_all",
  id = "install_all.cmd_header",
  title = "Install ALL Software Updates On Installation Media".)
[Nov 19 1995, 16:42:54]
  Command_to_Execute follows below:
>> /usr/lib/instl/sm_inst installp_cmd -T ems -q -a -c -B -g -N -X -d '/usr/csd/csd14.objbin' -S all'
```

Installation summary:

Name	Fix Id	Part	Event	Result	State
mqm.aix_client	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.base	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.dos_client	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.os2_client	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.samples	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.server	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.win_client	U439048	USR	APPLY	SUCCESS	APPLIED
mqm.aix_client	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.base	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.dos_client	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.os2_client	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.samples	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.server	U439048	ROOT	APPLY	SUCCESS	APPLIED
mqm.win_client	U439048	ROOT	APPLY	SUCCESS	APPLIED

mqm.win_client	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.server	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.samples	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.os2_client	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.dos_client	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.base	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.aix_client	U439048	USR	COMMIT	SUCCESS	COMMITTED
mqm.win_client	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.server	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.samples	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.os2_client	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.dos_client	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.base	U439048	ROOT	COMMIT	SUCCESS	COMMITTED
mqm.aix_client	U439048	ROOT	COMMIT	SUCCESS	COMMITTED

2.1.3 Creating MQM Objects

The files and structures that are created for a queue manager will be placed by default into sub-directories of /var/mqm. It will be necessary to have at least 17 MB in the file system containing the directory /var/mqm.

It is wise at this point before going on to create a separate file system that has /var/mqm as its mount point and has at least 17 MB of free space in it. You should use SMIT once again to accomplish this choosing the menu items as follows:

Physical & Logical Storage

File Systems

Add / Change / Show / Delete File Systems

Journalled File Systems

Add a Journalled File System

```

                                     Journalled File Systems

Move cursor to desired item and press Enter.

Add a Journalled File System
Add a Journalled File System on a Previously Defined Logical Volume
Change / Show Characteristics of a Journalled File System
Remove a Journalled File System

-----
                                     Volume Group Name

Move cursor to desired item and press Enter.

rootvg

F1=Help           F2=Refresh       F3=Cancel
F8=Image          F10=Exit         Enter=Do
/=Find           n=Find Next
  
```

Figure 9. Select Root Volume Group (VG)

Select **rootvg** volume group and press Enter.

In the screen in Figure 10 on page 23 change the values for "SIZE of file system" and "MOUNT POINT" as shown.

```

                                Add a Journaled File System

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

                                [Entry Fields]
Volume group name                rootvg
* SIZE of file system (in 512-byte blocks) [34000]      #
* MOUNT POINT                    [/var/mqm]
Mount AUTOMATICALLY at system restart?   yes           +
PERMISSIONS                        read/write     +
Mount OPTIONS                      []             +
Start Disk Accounting?              no            +

F1=Help      F2=Refresh    F3=Cancel    F4=List
F5=Reset     F6=Command    F7=Edit     F8=Image
F9=Shell     F10=Exit      Enter=Do

```

Figure 10. Add a Journaled File System

After this file system has successfully been created you may now proceed to create a default Message Queue Manager and the other structures used by MQSeries.

We created a default queue manager, RS60001, with this command:

```
crtmqm -q RS60001
```

The parameter -q makes the queue manger the default queue manager.

Press Enter and after a few minutes a successful completion message should appear.

Note: The default queue manager in the second AIX system is RS60002.

Hint

The queue manager name can be any 48-character name but convention says to make your default queue manager name the same as your TCP/IP host name but all capitals and suffixed with ".MQM".

For example, if your system's TCP/IP host name is "rabbit" than you would name your default queue manager name RABBIT.MQM. So the command would be written as

```
crtmqm -q RABBIT.MQM
```

Now you need to create the standard default and model queues for your system. Use the IBM supplied script file *amqscoma.tst*. in the directory */usr/lpp/mqm/samp*. This file is run using the *runmqsc* command as follows:

```
runmqsc < amqscoma.tst > coma.log
```

The output of the command will be written into the file *coma.log*. You should review this file and make sure there are no errors in the creation of any of the MQSeries objects it was trying to create. A quick way to check if all is well is to look at the end of the output file:

```
⋮  
21 MQSC commands read.  
0 commands have a syntax error.  
0 commands cannot be processed.
```

Note:

The last three lines show the number of commands processed (21), the number of commands with syntax errors, and the number of commands that could not be processed. These should both be zero. If they are not investigate as to why and correct the problem. Do not continue, however, until you get a clean run on this file.

The most common reasons that the execution of the command fails are:

- There is not sufficient space allocated for */var/mqm*.
- There is a problem with the security/authorization features of MQSeries. This problem can be circumvented by removing the lines referring to the authorization services in the file */var/mqm/qmgrs/RABBIT.MQM/qm.ini*. This should only be done, however, if you are working in a protected and non-production environment.

```
#####  
#* Module Name: qm.ini *#  
#* Type : MQSeries queue manager configuration file *#  
# Function : Define the configuration of a single queue manager *#  
#####  
Service:  
  Name=AuthorizationService  
  EntryPoints=9  
ServiceComponent:  
  Service=AuthorizationService  
  Name=MQSeries.UNIX.auth.service  
  Module=amqzfu  
  ComponentDataSize=0  
Log:  
  LogPrimaryFiles=3  
  LogSecondaryFiles=2  
  LogFilePages=1024  
  LogType=CIRCULAR  
  LogBufferPages=17  
  LogPath=/var/mqm/log/RABBIT.MQM/
```

Figure 11. QM.INI File

2.1.4 MQSeries Three Tier for AIX

The procedure to install the MQSeries 3T product is identical to that of installing the base MQSeries product.

```

Install Software Products at Latest Available Level

Type or select values in entry fields.
Press Enter AFTER making all desired changes.

* INPUT device / directory for software      [Entry Fields]
* SOFTWARE to install                        /usr/sys/inst.images
Automatically install PREREQUISITE software? [1.0.0.0 mq3t
COMMIT software?                             yes +
SAVE replaced files?                         no +
VERIFY Software?                             no +
EXTEND file systems if space needed?         yes +
REMOVE input file after installation?         no +
OVERWRITE existing version?                  no +
ALTERNATE save directory                      []

F1=Help      F2=Refresh      F3=Cancel      F4=List
F5=Reset     F6=Command     F7=Edit       F8=Image
F9=Shell     F10=Exit       Enter=Do

```

Figure 12. MQSeries 3T for AIX Installation

Below is the list of MQ3T LPP's to select for a custom installation.

```

1.0.0.0 mq3t ALL
1.0.0.0 mq3t.books
1.0.0.0 mq3t.os2_client_En_US
1.0.0.0 mq3t.os2_client_Es_ES
1.0.0.0 mq3t.os2_client_Fr_FR
1.0.0.0 mq3t.os2_client_Ja_JA
1.0.0.0 mq3t.os2_client_dev
1.0.0.0 mq3t.samples
1.0.0.0 mq3t.server_Es_ES
1.0.0.0 mq3t.server_Fr_FR
1.0.0.0 mq3t.server_Ja_JA
1.0.0.0 mq3t.server_base
1.0.0.0 mq3t.tools
1.0.0.0 mq3t.win_client_En_US
1.0.0.0 mq3t.win_client_Es_ES
1.0.0.0 mq3t.win_client_Fr_FR
1.0.0.0 mq3t.win_client_Ja_JA
1.0.0.0 mq3t.win_client_Ja_JA
1.0.0.0 mq3t.win_client_dev

```

The output from your MQ3T installation process should be similar to the following:

```

---- start ----
installp -qacFNXd/usr/sys/inst.images \
-f {File Containing Software} 2>&1

Contents of {File Containing Software}:

```

```

mq3t 1.0.0.0.all

Verifying requisites...done
Results...

SUCSESSES
-----
  Filesets listed in this section passed pre-installation verification
  and will be installed.

  Selected Filesets
  -----
mq3t.books 1.0.0.0                # AIX Books
mq3t.os2_client_En_US 1.0.0.0    # OS/2 Client Runtime (US Engl...
mq3t.os2_client_Es_ES 1.0.0.0    # OS/2 Client Runtime (Spanish)
mq3t.os2_client_Fr_FR 1.0.0.0    # OS/2 Client Runtime (French)
mq3t.os2_client_Ja_JA 1.0.0.0    # OS/2 Client Runtime (Japanese)
mq3t.os2_client_dev 1.0.0.0      # OS/2 Client Development Tools
mq3t.samples 1.0.0.0             # AIX Server Samples
mq3t.server_Es_ES 1.0.0.0         # AIX Server Runtime messages ...
mq3t.server_Fr_FR 1.0.0.0         # AIX Server Runtime messages ...
mq3t.server_Ja_JA 1.0.0.0         # AIX Server Runtime messages ...
mq3t.server_base 1.0.0.0          # AIX Server Runtime Base
mq3t.tools 1.0.0.0                # AIX Server Development Tools
mq3t.win_client_En_US 1.0.0.0     # Windows Client Runtime (US E...
mq3t.win_client_Es_ES 1.0.0.0     # Windows Client Runtime (Span...
mq3t.win_client_Fr_FR 1.0.0.0     # Windows Client Runtime (French)
mq3t.win_client_Ja_JA 1.0.0.0     # Windows Client Runtime (Japa...
mq3t.win_client_dev 1.0.0.0       # Windows Client Development T...

<< End of Success Section >>

The number of restored files is 2.

0503-376 installp: Applying software for the "usr" part of product
mq3t 1.0.0.0.
:
Restoring files, please wait.
The number of restored files is 1115.
The files for package xxxxx are being verified.
This may take several minutes, please wait.
:
0503-391 installp: The installation was SUCCESSFUL for the "usr" part of the
following software products:
mq3t.win_client_dev 1.0.0.0
mq3t.win_client_Ja_JA 1.0.0.0
mq3t.win_client_Fr_FR 1.0.0.0
mq3t.win_client_Es_ES 1.0.0.0
mq3t.win_client_En_US 1.0.0.0
mq3t.server_base 1.0.0.0
mq3t.tools 1.0.0.0
mq3t.server_Ja_JA 1.0.0.0
mq3t.server_Fr_FR 1.0.0.0
mq3t.server_Es_ES 1.0.0.0
mq3t.samples 1.0.0.0
mq3t.os2_client_dev 1.0.0.0
mq3t.os2_client_Ja_JA 1.0.0.0
mq3t.os2_client_Fr_FR 1.0.0.0
mq3t.os2_client_Es_ES 1.0.0.0
mq3t.os2_client_En_US 1.0.0.0
mq3t.books 1.0.0.0

0503-377 installp: Applying software for the "root" part of product
mq3t 1.0.0.0.

0503-392 installp: The installation was SUCCESSFUL for the "root" part of the
following software products:
mq3t.server_base 1.0.0.0
Finished processing all filesets. (Total time: 19 mins 11 secs).

Verifying requisites...done
Results...

SUCSESSES
-----

```

Filesets listed in this section passed pre-commit verification and will be committed.

Selected Filesets

```
-----
mq3t.books 1.0.0.0 # AIX Books
mq3t.os2_client_En_US 1.0.0.0 # OS/2 Client Runtime (US Engl...
mq3t.os2_client_Es_ES 1.0.0.0 # OS/2 Client Runtime (Spanish)
mq3t.os2_client_Fr_FR 1.0.0.0 # OS/2 Client Runtime (French)
mq3t.os2_client_Ja_JA 1.0.0.0 # OS/2 Client Runtime (Japanese)
mq3t.os2_client_dev 1.0.0.0 # OS/2 Client Development Tools
mq3t.samples 1.0.0.0 # AIX Server Samples
mq3t.server_Es_ES 1.0.0.0 # AIX Server Runtime messages ...
mq3t.server_Fr_FR 1.0.0.0 # AIX Server Runtime messages ...
mq3t.server_Ja_JA 1.0.0.0 # AIX Server Runtime messages ...
mq3t.server_base 1.0.0.0 # AIX Server Runtime Base
mq3t.tools 1.0.0.0 # AIX Server Development Tools
mq3t.win_client_En_US 1.0.0.0 # Windows Client Runtime (US E...
mq3t.win_client_Es_ES 1.0.0.0 # Windows Client Runtime (Span...
mq3t.win_client_Fr_FR 1.0.0.0 # Windows Client Runtime (French)
mq3t.win_client_Ja_JA 1.0.0.0 # Windows Client Runtime (Japa...
mq3t.win_client_dev 1.0.0.0 # Windows Client Development T...
```

<< End of Success Section >>

0503-379 installp: Committing software for the "usr" part of product
mq3t 1.0.0.0.

0503-400 installp: The commit operation was SUCCESSFUL for the "usr" part of
the following software products:

```
mq3t.books 1.0.0.0
mq3t.os2_client_En_US 1.0.0.0
mq3t.os2_client_Es_ES 1.0.0.0
mq3t.os2_client_Fr_FR 1.0.0.0
mq3t.os2_client_Ja_JA 1.0.0.0
mq3t.os2_client_dev 1.0.0.0
mq3t.win_client_En_US 1.0.0.0
mq3t.win_client_Es_ES 1.0.0.0
mq3t.win_client_Fr_FR 1.0.0.0
mq3t.win_client_Ja_JA 1.0.0.0
mq3t.win_client_dev 1.0.0.0
mq3t.server_base 1.0.0.0
```

0503-380 installp: Committing software for the "root" part of product
mq3t 1.0.0.0.

0503-401 installp: The commit operation was SUCCESSFUL for the "root" part of
the following software products:
mq3t.server_base 1.0.0.0

Filesets processed: 12 of 17 (Total time: 20 mins 8 secs).

0503-379 installp: Committing software for the "usr" part of product
mq3t 1.0.0.0.

0503-400 installp: The commit operation was SUCCESSFUL for the "usr" part of
the following software products:

```
mq3t.server_Es_ES 1.0.0.0
mq3t.server_Fr_FR 1.0.0.0
mq3t.server_Ja_JA 1.0.0.0
mq3t.tools 1.0.0.0
mq3t.samples 1.0.0.0
```

Finished processing all filesets. (Total time: 20 mins 12 secs).

Installation Summary:

Name	Fix Id	Part	Event	Result	State
mq3t.win_client_dev		USR	APPLY	SUCCESS	APPLIED
mq3t.win_client_Ja_JA		USR	APPLY	SUCCESS	APPLIED
mq3t.win_client_Fr_FR		USR	APPLY	SUCCESS	APPLIED
mq3t.win_client_Es_ES		USR	APPLY	SUCCESS	APPLIED
mq3t.win_client_En_US		USR	APPLY	SUCCESS	APPLIED
mq3t.server_base		USR	APPLY	SUCCESS	APPLIED
mq3t.tools		USR	APPLY	SUCCESS	APPLIED
mq3t.server_Ja_JA		USR	APPLY	SUCCESS	APPLIED
mq3t.server_Fr_FR		USR	APPLY	SUCCESS	APPLIED
mq3t.server_Es_ES		USR	APPLY	SUCCESS	APPLIED
mq3t.samples		USR	APPLY	SUCCESS	APPLIED
mq3t.os2_client_dev		USR	APPLY	SUCCESS	APPLIED
mq3t.os2_client_Ja_JA		USR	APPLY	SUCCESS	APPLIED
mq3t.os2_client_Fr_FR		USR	APPLY	SUCCESS	APPLIED
mq3t.os2_client_Es_ES		USR	APPLY	SUCCESS	APPLIED
mq3t.os2_client_En_US		USR	APPLY	SUCCESS	APPLIED
mq3t.books		USR	APPLY	SUCCESS	APPLIED
mq3t.server_base		ROOT	APPLY	SUCCESS	APPLIED
mq3t.books		USR	COMMIT	SUCCESS	COMMITTED
mq3t.os2_client_En_US		USR	COMMIT	SUCCESS	COMMITTED
mq3t.os2_client_Es_ES		USR	COMMIT	SUCCESS	COMMITTED
mq3t.os2_client_Fr_FR		USR	COMMIT	SUCCESS	COMMITTED
mq3t.os2_client_Ja_JA		USR	COMMIT	SUCCESS	COMMITTED
mq3t.os2_client_dev		USR	COMMIT	SUCCESS	COMMITTED
mq3t.win_client_En_US		USR	COMMIT	SUCCESS	COMMITTED
mq3t.win_client_Es_ES		USR	COMMIT	SUCCESS	COMMITTED
mq3t.win_client_Fr_FR		USR	COMMIT	SUCCESS	COMMITTED
mq3t.win_client_Ja_JA		USR	COMMIT	SUCCESS	COMMITTED
mq3t.win_client_dev		USR	COMMIT	SUCCESS	COMMITTED
mq3t.server_base		USR	COMMIT	SUCCESS	COMMITTED
mq3t.server_base		ROOT	COMMIT	SUCCESS	COMMITTED
mq3t.server_Es_ES		USR	COMMIT	SUCCESS	COMMITTED
mq3t.server_Fr_FR		USR	COMMIT	SUCCESS	COMMITTED
mq3t.server_Ja_JA		USR	COMMIT	SUCCESS	COMMITTED
mq3t.tools		USR	COMMIT	SUCCESS	COMMITTED
mq3t.samples		USR	COMMIT	SUCCESS	COMMITTED

2.2 Windows Client for Development

For this project, we install the following software:

- IBM DOS Version 7.0
- Microsoft Windows Version 3.1
- IBM TCP/IP Version 2.1.1
- IBM MQSeries Windows Client
- IBM MQSeries Three Tier Client
- Microsoft Visual Basic Version 3
- MQSeries 3T Support Pack for Visual Basic

2.2.1 IBM DOS 7.0

Insert the first DOS installation diskette in the A-drive, make the A-drive your current drive and type:

```
setup
```

We installed the following optional tools:

- IBM AntiVirus/DOS
- REXX Language Support

- Central Point Backup (this is a default)

Note: We did not select the DOS Shell.

2.2.2 MS Windows Version 3.1

Insert first diskette in the A-drive, make the A-drive your current drive and type:
setup

The installation program asks you to make choices. We recommend the following:

- Select **Express Setup (Recommended)** and press Enter.
- You have to enter your name when prompted.
- Accept the default directory C:\WINDOWS.
- On the printer installation screen select the printer attached to your workstation. If you have none select **No printer installed** and press Enter.
- Reboot your system

After the installation of DOS and Windows your directory should look like this:

```
DOS          <DIR>      11-06-95   5:55p
COMMAND  COM      52,956 11-17-94   1:00p
WINA20   386       9,349 11-17-94   1:00p
CONFIG   OLD        83 11-06-95   6:00p
AUTOEXEC OLD      140 11-06-95   6:00p
WINDOWS  <DIR>      11-06-95   6:06p
CONFIG   SYS      144 11-06-95   6:22p
AUTOEXEC BAT    176 11-06-95   6:22p
          8 file(s)      62,848 bytes
```

The CONFIG.SYS contains:

```
FILES=30
BUFFERS=10
DOS=HIGH
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\SETVER.EXE
DEVICE=C:\WINDOWS\SMARTDRV.EXE /DOUBLE_BUFFER
STACKS=9,256
```

To allow more space for environment variables we recommend to add this line to the CONFIG.SYS file:

```
SHELL=C:\DOS\COMMAND.COM /E:2048 /P
```

The AUTOEXEC.BAT file contains:

```
C:\WINDOWS\SMARTDRV.EXE
@ECHO OFF
SET PATH=C:\WINDOWS;C:\DOS;%PATH%
SET TEMP=C:\DOS
C:\DOS\MOUSE.COM
C:\DOS\DOSKEY.COM
SET IBMAV=C:\DOS
CALL C:\DOS\IBMAVDR.BAT C:\DOS\
```

2.2.3 TCP/IP

For the communication with the AIX server we selected TCP/IP for DOS and Windows, Version 2.1.1 and applied the fixes from Corrective Service Diskette (CSD) Version 2.1.1.4.

2.2.3.1 TCP/IP Installation

Insert the first of four diskettes of the TCP/IP base code in the A-drive, make the A-drive your current drive and type:

```
install
```

We accepted the default directory C:\TCPDOS and selected all components listed in the *Customize Product Installation* window, totalling about 11.5 megabytes. When the install program asks if it shall update the CONFIG.SYS type y.

The *TELNET 3270 EMULATOR* screen lets you choose a default translation table. We accepted the default **F** (US).

Reboot the system before you apply the CSD!

The insert the first of the three CSD diskettes into the A-drive and type:

```
tcpcsd
```

Type Y when the following window appears:

```
IBM TCP/IP for DOS Version 2.1.1
Corrective Service Diskette
July 31, 1995

Your TCPBASE environment variable is set to: C:\TCPDOS\ETC\..

Do you want the Corrective Service applied to the TCP/IP fpr DOS
installed at C:\TCPDOS\ETC\.. (Y, N or Q)
?y
```

Another screen asks you if you want to save the previous versions. Answer n.

You may also choose to have the online books updated. There are four books for you to view:

- User's Guide
- Installation and Administration
- Command Reference
- Programmer's Reference

To read the books type `tcpread` and follow the directions on the subsequent screens.

2.2.3.2 TCP/IP Customization

Before you begin with the customization you have to obtain from your network administrator the following:

Table 1. Addresses for TCP/IP Customization

	Used in this project
IP address for your workstation	9.24.104.107
Host name of the workstation	OAKC1
Subnet mask, usually that is...	255.255.255.0
Address of the IP router	9.24.104.1
Domain name server address	9.24.104.108
Domain name	itso.ral.ibm.com
The address of the AIX server is	9.24.104.26

At a DOS prompt type:

custom

The first screen you see shows the directories TCP/IP uses. Click on **OK** or press Enter.

Then the IBM logo appears. Select **Configure** from the menu bar. The menu offers the following choices:

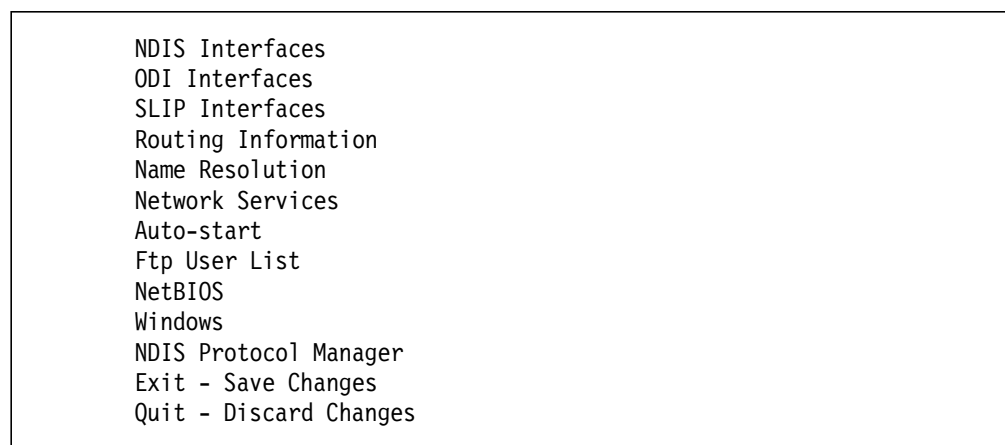


Figure 13. TCP/IP Configure Menu

To customize the workstation we followed these steps:

- Step 1.** Select **NDIS Interfaces** from the menu and fill in the window that will be displayed:
- Leave ND0 as the default for *Interface*.
 - Tab to *Options* and use the space bar to select **Enable interface**. An X will appear between the brackets.
 - Tab to *IP Address* and enter the address given to you by your LAN or systems administrator. We were given the IP address 9.24.104.107.
 - Tab to *Subnet mask* and enter 255.255.255.0.

- e. Tab to *Bound adapter*, click on the arrow and select an adapter from the pull-down. Our adapter is an IBM 16/4 Token Ring Network Adapter. Then press Enter.
 - f. Click on **OK**.
- Step 2.** Now you are prompted to insert the TCP/IP device driver diskette in the A-drive. Press OK.
- Step 3.** Click on **Routing Information** in the menu. In the window displayed enter two fields:
- a. Click on the arrow next to *Route Type* and double-click on **Default** in the pull-down.
 - b. Tab to *Router IP Address* and type 9.24.104.1. Then click on **OK**.
- Step 4.** Click on **Name Resolution** in the menu and enter three values in the window displayed:
- a. The host name, that is the name of the workstation that you configure, is OAKC1.
 - b. The domain name for our installation is its0.ral.ibm.com.
 - c. The domain name server address is 9.24.104.108.
 - d. Click on **OK**.
- Step 5.** Click on **Auto-start** in the menu and enable *TCP/IP* by pressing the space bar or use the mouse and click on the field between the brackets. Then click on **OK**.
- Step 6.** Click on **Windows** in the menu and then on **Yes** in the window displayed. This creates an icon for TCP/IP in the Program Manager window.
- In the subsequent *Confirm* window click on **Yes** to have the CUSTOM program update the SYSTEM.INI file in the Windows directory.
- Step 7.** When you select **NDIS Protocol Manager** from the menu the install program displays the adapter chosen in step 1. Click on **OK**.
- Step 8.** Select **Exit - Save Changes** from the menu. You are presented with some windows that ask you if you want the AUTOEXEC.BAT and CONFIG.SYS updated. It is recommended that you let the install program do this.
- Step 9.** Reboot your system.

After the installation of TCP/IP your directory contains the following new or updated entries:

CONFIG	SYS	293	11-16-95	6:54p	<-- changed
AUTOEXEC	BAT	315	11-16-95	6:51p	<-- changed
TCPDOS	<DIR>		11-16-95	5:29p	<-- new
AUTOEXEC	BK	176	11-16-95	5:42p	<-- backup
AUTOEXEC	BK1	279	11-16-95	6:51p	<-- backup
CONFIG	BK1	144	11-16-95	6:53p	<-- backup

The CONFIG.SYS contains the following changes:

```
FILES=30
BUFFERS=10
DOS=HIGH
DEVICE=C:\DOS\HIMEM.SYS
DEVICE=C:\DOS\SETVER.EXE
DEVICE=C:\WINDOWS\SMARTDRV.EXE /DOUBLE_BUFFER
STACKS=9,256
DEVICE = C:\DOS\ANSI.SYS
DEVICE = C:\TCPDOS\BIN\PROTMAN.DOS /I:C:\TCPDOS\ETC
DEVICE = C:\TCPDOS\BIN\DOSTCP.SYS
DEVICE = C:\TCPDOS\BIN\IBMTOK.DOS
SHELL=C:\DOS\COMMAND.COM /E:2048 /P
```

The AUTOEXEC.BAT contains the following changes:

```
C:\TCPDOS\BIN\NETBIND
SET ETC=C:\TCPDOS\ETC
C:\WINDOWS\SMARTDRV.EXE
@ECHO OFF
REM Old PATH statement
REM set path=c:\windows;c:\dos;%path%
set path=c:\windows;c:\dos;%path%;C:\TCPDOS\BIN;
SET TEMP=C:\DOS
C:\DOS\MOUSE.COM
C:\DOS\DOSKEY.COM
SET IBMAV=C:\DOS
CALL C:\DOS\IBMAVDR.BAT C:\DOS\
CALL TCPSTART
```

2.2.3.3 Test the Configuration

To verify your TCP/IP configuration select **Verify Data** from the Verify menu. A window confirms that you have specified all required items. However, the program cannot tell you if the values you entered during the customization are correct.

To test the configuration type at a DOS prompt:

```
tcpcheck
```

This program checks if:

- the INET daemon is up
- the ADAPTER works
- the GATEWAY is present
- the NAMESERVER is present
- the NAMESERVER works

At this time you should be able to "ping" the server, providing that machine is up and running. You can do that in two ways: by specifying the host name or the IP address of the machine you want to ping. The host name of our server is RS60001. At a DOS prompt type:

```
ping RS60001
```

and press Enter. The result should look like this:

```

C:\ping RS60001
PING rs60001.itso.ral.ibm.com (9.24.104.26): 56 data bytes
64 bytes from 9.24.104.26: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 9.24.104.26: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 9.24.104.26: icmp_seq=3 ttl=255 time=0 ms
^C

--- RS60001.itso.ral.ibm.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/max/avg = 0/0/0 ms

C:\ping 9.24.104.26
PING 9.24.104.26 (9.24.104.26): 56 data bytes
64 bytes from 9.24.104.26: icmp_seq=0 ttl=255 time=0 ms
64 bytes from 9.24.104.26: icmp_seq=1 ttl=255 time=0 ms
^C

```

Press Ctrl+C to stop the ping.

To ping from Windows, select the **TCPIP** icon and then the **Ping** icon. In the Host menu (shown below) enter the IP address or the server name and click on **OK**.

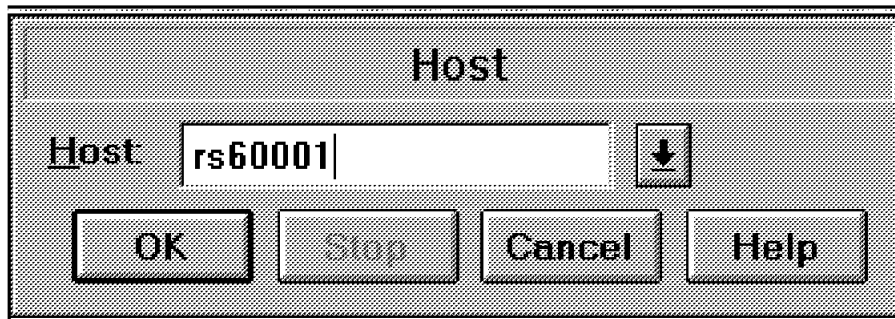


Figure 14. Ping a Host

To end the ping program select **Exit** from the File menu.

There should be no loss of data for the packets.

2.2.4 MQSeries Windows Client

For the installation of MQSeries for clients and servers the publication *MQSeries Clients*, SC33-1632 is very helpful.

Note: For MQSeries 3T to work the corrective service diskette CSD 14 must be applied to MQSeries on the AIX server system.

2.2.4.1 Installation

For the Windows 3.1 client you copy from either an OS/2 or AIX server the 15 files listed in Figure 15 on page 35.

AMQ9	MSG	30617
AMQCNBWB	DLL	74988
AMQCNBWB	LIB	2048
AMQCPMNW	DLL	40768
AMQCPMNW	LIB	1536
AMQCTCPW	DLL	89468
AMQCTCPW	LIB	2048
CMQC	H	50759
CMQCFC	H	23021
CMQXC	H	14905
MQIC	DLL	205300
MQIC	LIB	2560
RUNMQFMT	EXE	66273

Figure 15. MQSeries for Windows Clients: Files

There are two ways to move these files from the AIX server to the Windows workstation:

- On the AIX server, you may copy the files to diskette and then copy them from the diskette into a directory on the workstation's hard drive.
- Since TCP/IP is working, you may also "telnet" to the server and "ftp" (use the file transfer program) the files from the server to the workstation.

Where are the files? The files are in the following directories on your AIX system:

```
/usr/lpp/mqm/win_client/dll/*.exe
/usr/lpp/mqm/win_client/dll/*.dll
/usr/lpp/mqm/win_client/lib/*.lib
/usr/lpp/mqm/win_client/msg/*.msg
```

Transfer via diskette: Transferring the files via diskette requires more typing than using the file transfer program. You have to enter one command per file you want to copy, such as:

```
doswrite -D /dev/rfd0 /usr/lpp/mqm/win_client/bin/runmqfmt.exe runmqfmt.exe
-----
           diskette      AIX directory      filename      target file
```

Another way is to write a shell script to write files to a DOS diskette, such as:

```
cd /usr/lpp/mqm/win_client/dll
for i in *
do
  doswrite -D /dev/rfd0 $i $i
  echo writing $i
done
```

You have to repeat this for the other directories.

Then copy the files into a directory in your client machine:

```
C:\>md MQMWIN
C:\>cd MQMWIN
C:\MQMWIN>copy a:*.*
```

Transfer using FTP: A better way is to transfer the files using the file transfer program. You do this from the client workstation. Assume you want to copy the files for the MQ client into the directory mqmwin.

```
C:\>:cd mqmwin
C:\>:ftp rs60001
userid: mqm
password::
ftp>cd /usr/lpp/mqm/win_client/dll ftp>prompt off
ftp>mget *.dll
```

Repeat this for all directories. Change the path statement in your AUTOEXEC.BAT file to include the new directory:

```
set path=c:\windows;c:\dos;%path%;C:\TCPDOS\BIN;C:\MQMWIN;C:\
```

2.2.4.2 Establishing Communications

To establish MQSeries communications between workstation and server you have to create a channel and a queue in the AIX server. You must have a user ID on the AIX system. For this project, we use the user IDs root, mqm and mq3t. All belong to the two user groups mqm and mq3t.

On the server use the runmqsc command to add the channel and the queue:

```
runmqsc
```

The command to create a queue called RS60001.FREMOTE to receive messages from remote systems is as follows:

```
DEF QL(RS60001.FREMOTE) LIKE(SYSTEM.DEFAULT.LOCAL.QUEUE) DEFPSIST(YES)
  1 : DEF QL(HOSTNAME.FREMOTE) LIKE(SYSTEM.DEFAULT.LOCAL.QUEUE) DEFPSIST(YES)
AMQ8006: MQSeries queue created.
```

You can display the properties of the queue with the following command:

```
DIS Q(RS60001.FREMOTE) ALL
  2 : DIS Q(RS60001.FREMOTE) ALL
AMQ8409: Display Queue details.
DESCR( )
PROCESS( )
BOQNAME( )
INITQ( )
TRIGDATA( )
QUEUE(RS60001.FREMOTE)
CRDATE(1995-11-09)
CRTIME(16.29.58)
GET(ENABLED)
PUT(ENABLED)
DEFPRTY(0)
DEFPSIST(YES)
MAXDEPTH(5000)
MAXMSGL(4194304)
BOTHRESH(0)
SHARE
DEFSOPT(SHARED)
NOHARDENBO
MSGDLVSQ(PRIORITY)
RETINTVL(999999999)
USAGE(NORMAL)
NOTRIGGER
TRIGTYPE(FIRST)
```

```

TRIGDPTH(1)
TRIGMPRI(0)
QDEPTHHI(80)
QDEPTHLO(20)
QDPMAXEV(ENABLED)
QDPHIEV(DISABLED)
QDPLOEV(DISABLED)
QSVCINT(999999999)
QSVCIEV(NONE)
TYPE(QLOCAL)
DEFTYPE(PREDEFINED)
SCOPE(QMGR)
IPPROCS(1)
OPPROCS(0)
CURDEPTH(0)

```

The command to create the channel between the workstation (OAKC1) and the server (RS60001) is as follows:

```

def chl(OAKC1.TO.RS60001) CHLTYPE(SVRCONN) TRPTYPE(TCP) MCAUSER('mqm') +
  2 : def chl(OAKC1.TO.RS60001) CHLTYPE(SVRCONN) TRPTYPE(TCP) MCAUSER('mqm')
+
like(system.def.svrconn)
  : like(system.def.svrconn)
AMQ8014: MQSeries channel created.

```

Press Ctrl+D to exit RUNMQSC.

In the client workstation add the following environment variable to the AUTOEXEC.BAT file:

```
SET MQSERVER=OAKC1.TO.RS60001/TCP/9.24.104.26
```

The three parameters for the environment variable MQSERVER are:

- OAKC.TO.RS60002 is the name of the SVRCONN channel defined on the AIX MQSeries server.
- The protocol type of the channel is TCP.
- The AIX server is at IP address 9.24.104.26

For more information on environment variables refer to *MQSeries Clients*, SC33-1632.

Note: Unlike OS/2 environment variables have no effect until placed into the AUTOEXEC.BAT file and the system is rebooted.

2.2.4.3 Verify the Client/Server Connection

To check if communications between workstation and server works you may use the CL1T program. This program runs under Windows 3.1.

To invoke the program type CL1T in the *Run* window (providing the program resides in the root directory of the C drive) and click on **OK**.

Diskette

CL1T is supplied on diskette 1 distributed with this book.

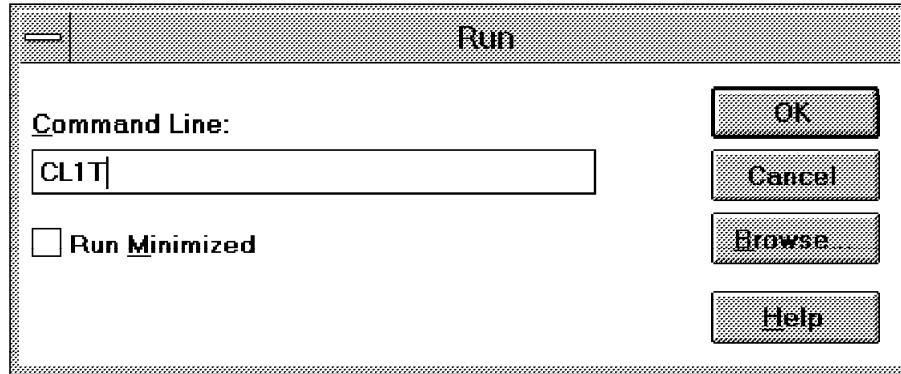


Figure 16. Verify the Host Connection

Ignore the text in the window *CL1 - Windows MQI Client Test*. When you click on MQI Test in the menu bar you see the menu of the six functions the program performs:

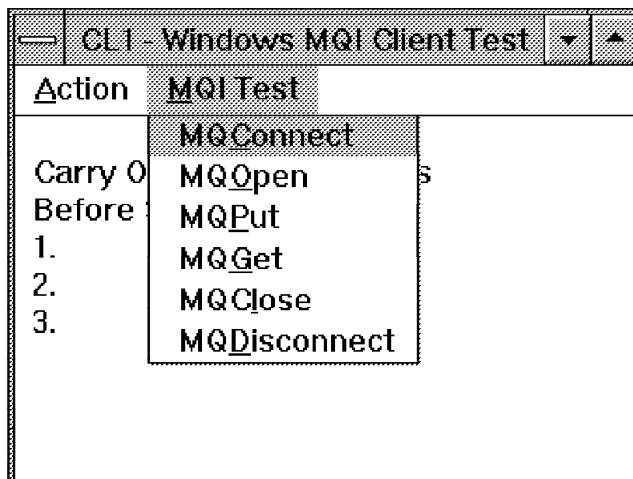


Figure 17. Issue Single MQI APIs

Perform the test by issuing the commands in the above order. When you click on one of the menu items a window appears in which you enter data. The window disappears when the function executed correctly. Otherwise an error message will be displayed. For error messages refer to the MQI return codes in *MQSeries for MVS/ESA Messages and Codes*, SC33-0819.

Notes:

1. If you do not type the name of a queue manager the program uses the default queue manager.
2. As queue name we use RS60001.FREMOTE. The queue name is case sensitive!
3. The program lets you specify the number of messages to send and a message text.
4. The MQGet function gets all messages in the queue, not only the ones you sent. Use the NOWAIT Option. WAIT causes the program to stay in a loop and you will not be able to close and disconnect.

2.2.5 3T Windows Client

The 3T client support for Windows 3.1 comes on two diskettes. To install it follow these steps:

- Step 1.** Bring up Windows
- Step 2.** Select **Run** from the File menu.
- Step 3.** Type a:\install
- Step 4.** Click on **Continue** when the *Installation* window appears.
- Step 5.** Click on **OK** in the *Install* window. Ensure that the option *Update CONFIG.SYS/AUTOEXEC.BAT* is checked.
- Step 6.** Since we install a Windows development workstation click on **Select all** in the *Install - directories* window. That installs the following functions:
 - MQSeries Three Tier - Windows Tools
 - MQSeries Three Tier - Windows Runtime
 - MQSeries Three Tier - Windows Samples

You may change the directory name, if you wish. Click on **Install** to begin with the installation.

- Step 7.** Reboot your system after the installation is completed.

Note: Though we do not plan to use the Windows sample programs they may be beneficial for reference purposes. To create executables for those C samples we would have to install Visual C++. In this project, however, we use Visual Basic to create programs running in the client workstations.

The AUTOEXEC.BAT contains the following changes:

```
C:\TCPDOS\BIN\NETBIND
SET ETC=C:\TCPDOS\ETC
C:\WINDOWS\SMARTDRV.EXE
@ECHO OFF
REM Old PATH statement
REM set path=c:\windows;c:\dos;%path%
set path=c:\windows;c:\dos;%path%;C:\TCPDOS\BIN;C:\MQMWIN;C:\;c:\3TIERW\BIN;
c:\3TIERW\SAMPLES\BIN
SET TEMP=C:\DOS
C:\DOS\MOUSE.COM
C:\DOS\DOSKEY.COM
SET IBMAV=C:\DOS
CALL C:\DOS\IBMAVDR.BAT C:\DOS\
CALL TCPSTART
SET MQSERVER=OAKC1.TO.RS60001/TCP/9.24.104.26
SET INCLUDE=c:\3TIERW\INCLUDE
SET LIB=c:\3TIERW\LIB
SET BMQLOCPATH=c:\3TIERW
```

You must add the following environment variable to the AUTOEXEC.BAT file or you get a conversion error (BMQ1332) when sending messages to the AIX server.

```
SET BMQCCSID=850
```

Make sure this environment variable setting matches the AIX system environment language settings. US English is IBM-850. This environment variable is needed because presently the AIX code translation does not support

the MQ3T "unibyte" implemented code page number 1004. This tells MQ3T to use a CCSID 850 translation.

The CONFIG.SYS remains unchanged.

The installation program adds the following directories:

```
C:3TIERW
+---INCLUDE
+---LIB
+---DATA
+---BIN
+---UCONVTAB
+---ICONV
\---SAMPLES
  +---C
    +---HELLO1
    +---HELLO2
    +---HELLO3
    +---HELLO4
    +---HELLO5
    +---HELLO6
    \---JOBVIEW
  \---BIN
```

The Program Manager windows contains three new icons:

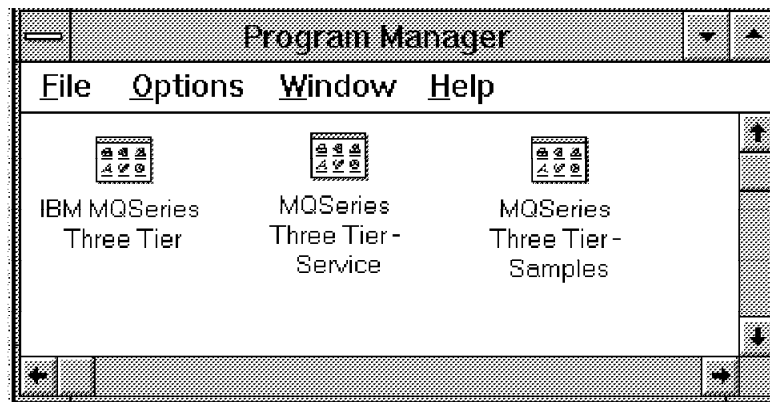


Figure 18. MQ3T Icons in Windows Program Manager

2.2.6 Visual Basic

For this project, we use Microsoft Visual Basic for Windows, Standard Edition, Version 3.0. To install it follow these steps:

- Step 1.** Bring up Windows
- Step 2.** Select **Run** from the File menu.
- Step 3.** Type a:\setup and press Enter.
- Step 4.** You have to enter your name when prompted.
- Step 5.** Accept the directory C:\VB; it will be created if it does not exist.
- Step 6.** Select **Complete Installation** in the Installation Option window.

The Visual Basic files are in the directory C:\VB. The installation program has no need to update the CONFIG.SYS and AUTOEXEC.BAT files.

To start Visual Basic double-click on the **Visual Basic** icon in the program manager window. This displays the following window:

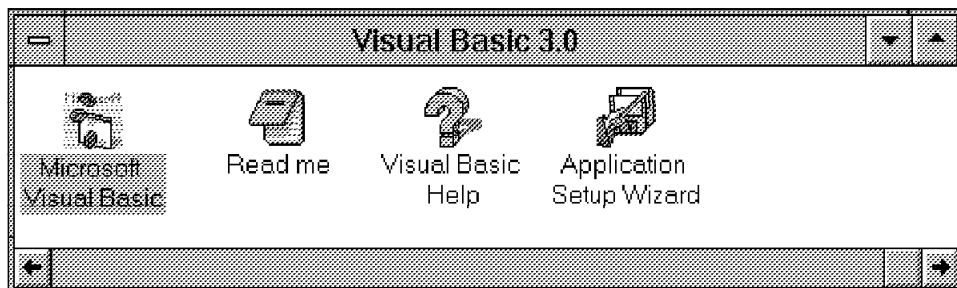


Figure 19. Visual Basic Icons

2.2.7 Visual Basic Support for Windows Clients

MQSeries Three Tier Visual Basic Support for Windows Client is provided as a SupportPac, MA3B. It contains Visual Basic source code to allow MQSeries Three Tier applications to be written using Visual Basic. Also included are program fragments illustrating how the MQSeries Three Tier API is invoked using Visual Basic. The SupportPac consists of four files:

MA3B PACKAGE	General Information
MA3B ANNOUNCE	Announce File
MA3B ZIPBIN	Visual Basic support and samples etc
LICENSE2 TXT	License agreement

This SupportPac extends the range of options for MQSeries Three Tier customers by enabling the use of Microsoft's Visual Basic product. Using this SupportPac you can use Visual Basic to develop Windows 3.1 Graphical User Interfaces for IBM MQSeries Three Tier applications.

To install the SupportPac follow these steps:

Step 1. Copy MA3B.ZIP to a temporary directory or diskette.

Step 2. To unzip the file type UNZIP MA3B. This creates the following files:

```
MA3B      ZIP      298862
LICENSE2  TXT       5586
SOURCE    <DIR>
INSTALL   BAT       701
README    WRI       50560
```

Note: README.WRI is a Windows Write format file that contains installation instructions. Open this file from the File Manager window.

Step 3. Assuming the files are on a diskette, insert the diskette in the A-drive, make the A-drive your current drive, and type:

```
install
```

The installation program adds the following subdirectories and files:

```
C:\3TIERW\VBSUPP\BMQB.BAS
C:\3TIERW\VBSUPP\CMQB.BAS
C:\3TIERW\VBSUPP\BMQVBX.BAS
C:\3TIERW\VBSUPP\OAKCD.BMP
C:\3TIERW\VBSUPP\OAKCU.BMP
C:\3TIERW\VBSUPP\OAKEU.BMP
C:\3TIERW\VBSUPP\OAKMU.BMP
C:\3TIERW\SAMPLES\VB\HELLO1
C:\3TIERW\SAMPLES\VB\HELLO2
C:\3TIERW\SAMPLES\VB\PCUST
C:\3TIERW\SAMPLES\VB\SPEEDUP
C:\3TIERW\SAMPLES\VB\TEMPLATE
C:\3TIERW\README.WRI
```

Step 4. Before you can use the SupportPac copy the following files into the WINDOWS\SYSTEM directory:

```
C:\3TIERW\VBSUPP\BMQNTFY.VBX
C:\3TIERW\VBSUPP\VBRUN300.DLL
```

Note: VBRUN300.DLL may already be in the directory WINDOWS\SYSTEM. During Visual Basic installation a copy is placed in that directory.

2.3 Windows Client for Production

A Windows client that executes presentation logic only requires the following software:

- IBM DOS Version 7.0
- Microsoft Windows Version 3.1
- IBM TCP/IP Version 2.1.1
- IBM MQSeries Windows Client
- IBM MQSeries Three Tier Client
- The run-time program of Visual Basic
- A subset of the SupportPac of 3T for Visual Basic
- The application

Chapter 3. Using Visual Basic

MQSeries Three Tier Visual Basic Support for Windows Clients is provided as a SupportPac, MA3B. This licensed material contains Visual Basic source code that allows you to use Microsoft's Visual Basic product to develop Microsoft Windows 3.1 GUIs for IBM MQSeries Three Tier applications.

Note: You develop Presentation Logics (PLs) on the Windows system. Business Logics (BLs) must be developed on either an OS/2 or AIX machine. Also, the 3T class compiler is only available for OS/2 and AIX.

The following sections include material from the SupportPac. How to install the package is described in section 2.2.6, "Visual Basic" on page 40.

3.1 Introduction

The MQSeries Three Tier (MQ3T) product announced on the 12th September 1995 provides a structured way of developing distributed client/server applications. MQ3T has been designed to be "AD Tool Neutral". This means that different application development tools can be used to produce the application parts for MQ3T applications.

One of the key areas of application development for MQ3T is that of producing the Graphical User Interface (GUI) parts of the distributed application. IBM has already announced IBM VisualAge V2 support for MQ3T and SupportPac MA3D extends this support to VisualAge V3. This SupportPac is designed to extend the range of options for MQ3T customers by enabling the use of Microsoft's Visual Basic product. Using this SupportPac you can use Visual Basic to develop Microsoft Windows 3.1 GUIs for IBM MQSeries Three Tier applications.

3.2 The SupportPac

This SupportPac contains Visual Basic source code to allow MQ3T applications to be written using Visual Basic. Also included are program fragments illustrating how the MQ3T API is invoked using Visual Basic.

These fragments have been extracted from a larger application that formed part of the first MQSeries Three Tier education class.

Also provided are two samples HELLO1 and HELLO2 which are analogous to the MQ3T C samples. These can be run and used to confirm that the installation has been successful.

A template program is provided which can form the basis of further code development. In its initial state, it provides the infrastructure which any MQ3T Visual Basic program will have, and unmodified, runs in a similar way to HELLO1. The user is prompted to write their own code.

A speed-up program provides a faster means to start PL programs. The program is provided which reloads the VBRUN300.DLL and can be used to start this during Windows startup.

3.3 The Visual Basic / MQ3T Interface

In MQ3T, the GUI parts of an application are called Presentation Logic classes (PLs). The purpose of the SupportPac MA3B is to produce PLs using Visual Basic.

Visual Basic PLs interface to the MQ3T Presentation Logic Manager (the PLM). It is the PLM that notifies the Visual Basic PL when 3T rules have been satisfied and the associated methods need to be invoked.

When it first starts, the Visual Basic PL needs to identify itself to the PLM, so the PLM knows how to direct events to the PL. This is achieved when the PL issues the MQ3T MQREG API registration call. The call supplies the PL's window handle and an event ID that the PL wants to associate with events from the PLM. The handle allows the PLM to send events to the PL through the Windows 3.1 windowing system and the event ID allows the PL to determine that it was the PLM (not some other part of Windows 3.1) that sent the event.

Whenever the PLM communicates with the Visual Basic application, it will send the specified event ID to the specified window handle.

The following code registers a PL with the PLM. The procedure is invoked when the form is loaded.

```
Sub Form_Load ()
    If Command = "" Then          ' Check for command-line parameter
        MsgBox "No ClassName passed to program.", 16, "PROGRAM TERMINATING"
    End
End If

vPLClass = Command              ' Convert string to classname

' Register class with 3T

MQREG ByVal vPLClass,
      1,
      ByVal OAK1.hWnd,
      ByVal BMQ_NOTIFY,
      ByVal MQRGO_REMOVE_LIST_ENTRIES,
      CompCode, Reason

DisplayCompCode "MQREG"

End Sub
```

Figure 20. Register a Window with 3T

Note: "Form_Load" is obtained from the HELLO1 sample. A parameter containing the class name must be passed to the PL. Otherwise, the program terminates.

Table 2 on page 45 lists the parameters for the MQREG API.

Table 2. MQREG Parameters

Parameter	Description
ByVal vPLClass (input)	The name of the 3T PL class that will be associated with the window handle. This is the first parameter of the STARTJOB command. The field is defined in HELLO1H.BAS.
1 (input)	The maximum number of instances the class can process at one time. The PLM creates one instance of the PL. However, several copies of the program may be active at the same time.
ByVal OAK1.hWnd (input)	The window handle associated with the class. The PLM sends PLM event messages for the PL to this window handle.
ByVal BMQ_NOTIFY (input)	The message ID that the PLM uses when it posts an event message to the window handle. This value is specified in the file BMQVBX.BAS as WM_USER + 1 (see page 51).
ByVal MQRGO_REMOVE_LIST_ENTRIES	This positional parameter has no effect under Windows.
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

When the PL instance ends it has to unregister itself. The MQUREG API is used for this purpose. The following code is obtained from the HELLO1 sample and shows how to end a PL instance and how to unregister it.

```

Sub Close_Click ()
    ' common exit path

    If vHInst <> 0 Then          ' issue MQSETS only if vHInst is valid 1
        MQSETS ByVal vHInst, ByVal MQSTATE_END, CompCode, Reason

        DisplayCompCode "MQSETS"
    End If

    ' unregister the previously registered class

    MQUREG ByVal vPLClass,
            ByVal OAK1.hWnd,
            ByVal MQRGO_FORCE,
            CompCode, Reason
    ' all on one line

    DisplayCompCode "MQUREG"

    End          ' end program execution
End Sub

```

Figure 21. Unregister a Window with 3T

The routine in Figure 21 is invoked when the user exits the application. The routine issues two 3T APIs:

- MQSETS (**MQ Set State**)
- MQUREG (**MQ UnREGister**)

If the window handle is valid a MQSETS API is issued. Setting the instance state to the value of MQSTATE_END causes the PLM to terminate the instance.

The window handle, vHInst, is not valid (**1**) when the program was run from within Visual Basic. Since no startjob command was issued there is no instance.

The parameters of the MQSETS API are:

Table 3. MQSETS Parameters

Parameter	Description
ByVal vPLClass (input)	The name of the class that is associated with the window handle. This is the first parameter of the STARTJOB command and it was saved when the form was loaded (see Figure 20 on page 44).
ByVal MQSTATE_END (input)	This value causes the instance to end; the GUI goes away. This and other values a state can be set to are defined in BMQB.BAS and explained on page 161 of the <i>Application Programming</i> manual.
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

The MQUREG API unregisters the PL. The parameters of the API are:

Table 4. MQUREG Parameters

Parameter	Description
ByVal vPLClass (input)	The name of the class that is associated with the window handle.
ByVal OAK1.hWnd (input)	The handle of the window to unregister.
ByVal MQURGO_FORCE (input)	This option terminates all instances associated with the class and window handle. The other option, MQURGO_NORMAL, does not unregister if the window handle is associated with any instances.
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

The routine DisplayCompCode in Figure 30 on page 54, also from the HELLO1 sample, displays the completion and reason codes of 3T APIs in a message box. The parameter handed over to the routine is the name of the 3T API call.

3.4 Parameter Passing

After the Visual Basic application (PL) has registered itself, the PLM will pass the application an *instance handle*, which the application then uses for any further communication with the PLM. The instance handle is passed by the PLM using the Windows LPARAM variable, with the WPARAM variable indicating what the PLM is communicating.

However, in Visual Basic, these parameters are not directly accessible, and so it has been necessary to develop an interface layer between Visual Basic and the PLM to intercept events, obtain the LPARAM and WPARAM values, and pass them to the Visual Basic application. This interface layer is implemented as a Visual Basic Custom Control, or .VBX file.

As well as intercepting the LPARAM and WPARAM variables, it also overcomes the problems that occur in Visual Basic when the application is displaying a Modal dialog Box. If a Modal dialog Box is being displayed, Visual Basic will only allow the application to detect the events that are associated with that Modal dialog Box (for instance, a Cancel_click, or an Enter_click if the Modal dialog Box is displaying Cancel and Enter). While Modal dialog Boxes are being displayed, incoming events from the PLM will be lost.

The MQ3T Custom Control can detect the incoming event from the PLM and will hold the event until the Visual Basic application has removed the Modal dialog Box, when the MQ3T Custom Control will pass the event to the application.

Figure 22 summarizes the above.

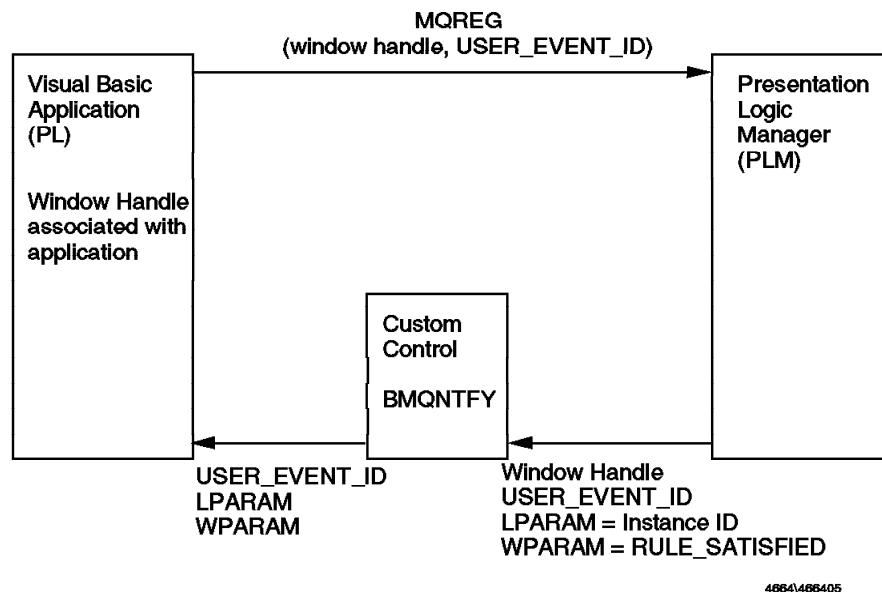


Figure 22. Parameter Passing between PL and PLM

Once the Visual Basic application (PL) has obtained its Instance ID, it can then use the other MQ3T API calls to send data (MQSEND) and query/obtain replies (MQQRYE/MQQRYM). Typically the application will be structured to issue MQSENDS when the user clicks buttons on the application screen.

The code in Figure 23 on page 48, also obtained from the HELLO1 sample, is invoked when a PL event occurs, that is when the PL receives a message.

The name of the procedure, OAK1_NewEvent, is hard-coded in the routine BMQNTFY.VBX. OAK1 is the name of the object that is created when the oaktree icon is dragged from the Visual Basic Toolbox into the form. NewEvent is the default name for the procedure associated with that object.

Three parameters are passed from the custom control BMQNTFY to the Windows event procedure:

1. The user event ID (message ID) as written in the MQREG call in Figure 20 on page 44. The value BMQ_NOTIFY is defined in the file BMQVBX.BAS as WM_USER + 1 (see page 51). When a message with that ID arrives the procedure OAK1_NewEvent is called.
2. There are three event message IDs (WPARAM):
 - MQPLM_RULE_SATISFIED
 - MQPLM_INSTANCE_DELETED
 - MQPLM_HWND_UNREGISTERED

A description can be found on page 226 of the *Application Programming* guide.

3. The instance handle (LPARAM) should be saved, since it will be used as an input parameter in other MQ3T API calls. It will tell the PLM which instance of a class issued the call. Here the instance handle is saved in the field vHInst. The field is defined in HELLO1H.BAS.

```
Sub OAK1_NewEvent (msg As Integer, wp As Integer, lp As Long)

    If wp = MQPLM_RULE_SATISFIED Then
        vHInst = lp
        ProcessPEvent ByVal lp ' make call to handle the event

    ElseIf wp = MQPLM_HWND_UNREGISTERED Then
        End ' already unregistered, so simply exit
    End If

End Sub
```

Figure 23. PL (GUI) Receives MQ3T Events

When a rule is satisfied then the instance ID is saved in the field *vHInst* (defined in HELLO1H.BAS) and another routine, ProcessPEvent, is called to process the event. That routine represents the input side of the Presentation Logic, the user code that analyses an incoming message and acts on it.

After describing what happens when the PLM sends a message to the PL (GUI) let us discuss what happens when the PL sends a message to a BL, DL or another PL.

Usually, such an action is initiated when the user clicks on a button or on a menu item in the GUI. The following procedure is from the HELLO1 sample. It is invoked when the user clicks on the menu item "Hello BL" and sends a message to the BL.

```

Sub MenuHelloBL_Click ()

  Dim BufferMsg As HELLO ' buffer - NB Don't define as String

  BufferMsg.message = "Hello BL Method"
  ' send request BLREQUEST to class BLCLASS

  MQSEND ByVal vHInst,
          ByVal BLCLASS,
          ByVal BLINSTANCE,
          ByVal BLREQUEST,
          0,
          BufferMsg,
          CompCode, Reason

  DisplayCompCode "MQSEND"
End Sub

```

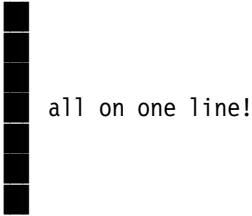


Figure 24. PL (GUI) Sends a Message

The MQSend API sends the message with the name BLREQUEST to the instance BLINSTANCE of the class BLCLASS. The following table describes the parameters:

Table 5. MQSEND Parameters

Parameter	Description
ByVal vPLClass (input)	The field vPLClass contains the name of the 3T PL class that sends the message. The name was obtained from the routine in Figure 20 on page 44. The field is defined in HELLO1H.BAS.
ByVal BLCLASS (input)	The name of the destination class, "hellobl1", is defined in the file HELLO1H.BAS.
ByVal BLINSTANCE (input)	The name of the destination instance, "Beech", is defined in the file HELLO1H.BAS.
ByVal BLREQUEST (input)	This is the name of the message to send. Its name "BLRequest" and its structure are defined in HELLO1.BAS.
0 (input)	This positional parameter is reserved for message attributes. Message attributes are not used in this example.
BufferMsg (input)	This is the buffer that holds the data of the message. The buffer is defined as HELLO and HELLO is defined in HELLO1.BAS as: Global Const HELLOLENGTH = 20 Type HELLO message As String * HELLOLENGTH End Type
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

3.5 SupportPac Content

The SupportPac code can be separated into five categories:

- Base functions
- Sample programs
- Sample READ ONLY code fragments
- A template for your own programs
- A run-time utility

The file README.WRI describes the SupportPac.

3.5.1 Base Functions

The following files are installed in the support directory:

C:\3TIERW\VBSUPP

3.5.1.1 BMQNTFY.VBX

This is the custom control used to intercept events from the Presentation Logic Manager destined for PLs. It is required so that the WPARAM and LPARAM parameters can be passed to the PL.


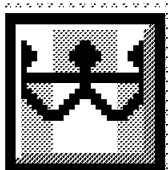
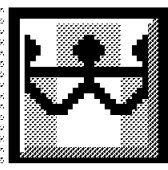
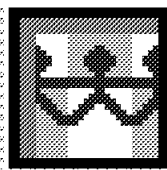
When developing Visual Basic applications, it is necessary to place this custom control somewhere on the form being developed. It is an invisible control, so is not seen at run time.

Also required on the form is the timer control, as this .VBX uses the timer to retrigger events.

Note: BMQNTFY.VBX must be copied to the \WINDOWS\SYSTEM directory for correct operation.

3.5.1.2 OAKCU.BMP OAKMU.BMP OAKEU.BMP OAKCD.BMP

These files are bit map support files for the custom control BMQNTFY.

<i>Table 6. Icons Used for Objects in Visual Basic Support</i>			
OAKCU	OAKMU	OAKEU	OAKCD
			

When the custom control is added to a Visual Basic application, the control appears as an Oak tree.

3.5.1.3 BMQVBX.BAS

This file defines the event/message ID that is generated by the custom control BMQNTFY when the PLM sends a message to a PL. It is used by the Visual Basic application when it registers with the PLM.

The file contains two lines. The event ID is called BMQ_NOTIFY.

```
Global Const WM_USER      = &H400
Global Const BMQ_NOTIFY   = WM_USER + 1&
```

Figure 25. BMQVBX.BAS File

3.5.1.4 BMQB.BAS

This is the Visual Basic equivalent of the *MQ3T* header file BMQC.H. It contains all the *MQ3T* structure definitions and API function prototypes and constants.

3.5.1.5 CMQB.BAS

This is the Visual Basic equivalent of the *MQ* header file CMQC.H. It contains all the *MQ* structure definitions and API function prototypes and constants.

3.5.2 Sample Programs

The SupportPac contains two sample programs, HELLO1 and HELLO2. The programs are similar to the HELLO programs supplied with MQ3T. The following files are installed in two directories:

```
C:\3TIERW\SAMPLES\VB\HELLO1
C:\3TIERW\SAMPLES\VB\HELLO2
```

Note: A detailed description on how to set up and run the HELLO1 sample can be found in 3.7, “Using the Visual Basic 3T Sample Programs” on page 67.

3.5.2.1 HELOGU1W.MAK

This file is the project file for a simple sample program that uses fixed-format messages. The program is similar to the HELLO1 C sample supplied with MQ3T.

This sample can be compiled and the executable file HELLOGU1W.EXE placed in the 3TIERW\SAMPLES\BIN directory.

To run the client part of HELLO1, start the PLM and issue a STARTJOB message. The sample will run using a Visual Basic PL instead of the usual C GUI program.

Alternatively with the PLM running, the GUI program can be debugged from within Visual Basic. Before sending a STARTJOB message, run the sample by single stepping within Visual Basic. The program will register and a point will be reached where control is given to the executing program. Notice how the menu contains a SEND command which is grayed out. Before this can be activated, the PL needs an instance. Send a STARTJOB message to the program. The Visual Basic program will receive a message and by single stepping through this code a handle will be retrieved. The grayed out menu item is then enabled. Clicking the SEND will send a message to the BL which will respond.

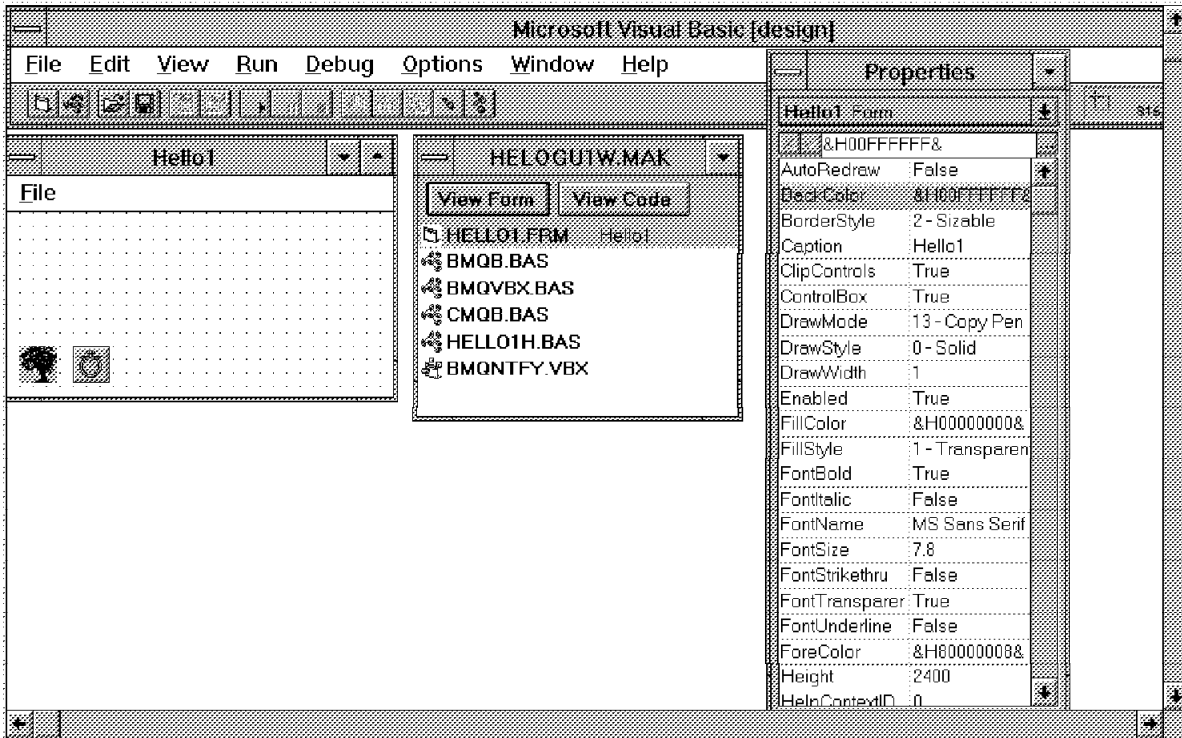


Figure 26. Sample HELLO1: HELOGU1W.MAK

Because the program is executing in Visual Basic, it can be single-stepped, or run and breakpoints can be set. Individual instructions can be executed and modified and rerun until correct. When the operation is complete, the program can be compiled and run in the traditional way with a STARTJOB message causing the program to be loaded and run. When the program loads, it will cause the custom control and Visual Basic run-time DLL to load.

You will find it useful to step through this working program to observe its operation and modify the program dynamically.

Note

How to compile and run the sample program is described in detail in 3.7.2, "Preparations on the Windows Client" on page 71.

3.5.2.2 HELLO1.FRM

This sample of code provides the dialog box and simple code to send a fixed length message to a BL. By comparing the code with the HELLO1 C sample direct comparisons between C and Visual Basic can be made.

HELLO1.FRM is similar to HELLOGU1.C in the C sample. The following figures contain the Visual Basic code.

```

VERSION 2.00
Begin Form Hello1
  BackColor      = &H00FFFFFF&
  Caption        = "Hello1"
  ClientHeight   = 1656
  ClientLeft     = 864
  ClientTop      = 1884
  ClientWidth    = 3480
  Height         = 2400
  Left           = 816
  ScaleHeight    = 1656
  ScaleWidth     = 3480
  Top            = 1188
  Width          = 3576
  Begin Timer Timer1
    Left          = 600
    Top           = 1200
  End
  Begin OAK OAK1
    Left          = 120
    Top           = 1200
  End
  Begin Menu MenuFile
    Caption       = "&File"
    Begin Menu MenuHelloBL
      Caption     = "&Hello BL"
      Enabled     = 0 'False
    End
    Begin Menu MenuExit
      Caption     = "E&xit"
    End
  End
End
End

```

Figure 27. Sample HELLO1: Visual Basic's Dialog Box Description

Form_Load is called when Visual Basic loads the form (displays the window). See also page 44.

```

Sub Form_Load ()
  ' if running from within Visual Basic, select Project...
  ' from Options menu and type the ClassName in the
  ' command line parameter - eg hellogu1
  If Command = "" Then ' check for command-line parameter
    MsgBox "No ClassName passed to program.", 16, "PROGRAM TERMINATING"
  End
  End If

  vPLClass = Command ' convert string to classname

  ' register class with MQ3T
  MQREG ByVal vPLClass, 1, ByVal OAK1.hWnd, ByVal BMQ_NOTIFY,
        ByVal MQRGO_REMOVE_LIST_ENTRIES, CompCode, Reason
  DisplayCompCode "MQREG"
End Sub

```

Figure 28. Sample HELLO1: Display the Window.
Object: Form, Procedure: Load

Declarations

```
Option Explicit
Dim CompCode As Long
Dim Reason As Long
```

Figure 29. Sample HELLO1: Declarations.

Object: General, Procedure: Declarations

DisplayCompCode is called to display MQ3T reason codes.

```
Sub DisplayCompCode (ByVal OakCall As String)
    Dim errtxt As String

    ' no previous instance ? - OK ( normal operation - not displayed )
    If CompCode = MQCC_WARNING And Reason = MQRC_NO_INSTANCE Then Exit Sub

    ' check return code and reason to determine if error should be displayed
    If CompCode <> MQCC_OK Then
        errtxt = "CC = " & CompCode & " RC = " & Reason
        MsgBox errtxt, 48, OakCall
    End If
End Sub
```

Figure 30. Sample HELLO1: Display Completion Codes.

Object: General, Procedure: DisplayCompCode

OAK1_NewEvent is called when the PL receives a message from the PLM. Note that this is a windows message and not the reply sent by the BL. The reply is retrieved in the procedure ProcessPLevent.

```
Sub OAK1_NewEvent (msg As Integer, wp As Integer, lp As Long)

    ' uncomment the next line to get notification of new events
    ' MsgBox "ClassName: " & vPLClass, 64, "NEW EVENT"

    If wp = MQPLM_RULE_SATISFIED Then
        vHInst = lp
        ProcessPLevent ByVal lp ' make call to handle the event

    ElseIf wp = MQPLM_HWND_UNREGISTERED Then
        End ' already unregistered, so simply exit
    End If

End Sub
```

Figure 31. Sample HELLO1: Receive Messages from BL.

Object: OAK1, Procedure: NewEvent

ProcessPLevent is called from *NewEvent* when a rule is satisfied. It retrieves the data of the reply message and displays it in a message box. A description of the APIs is on page 57.

```

Sub ProcessPLevent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure
    Dim MsgParams As MQMP            ' message parameters
    Dim BufferLen As Long             ' buffer length
    Dim BufferMsg As HELLO           ' buffer -
                                    ' NB Don't define as String

    ' rule clicked - enable "Hello BL" in "File" menu
    MenuHelloBL.Enabled = True

    ' query information about the current event
    MQQRYE ByVal HInst, MQevent, CompCode, Reason
    DisplayCompCode "MQQRYE"

    ' if the rule is RI_BLREPLY, retrieve the message data and display it
    If MQevent.RuleId = RI_BLREPLY Then
        BufferLen = MQevent.MaxBufferLength
        MQQRYM ByVal HInst, ByVal 1, MsgParams, BufferLen, BufferMsg, CompCode, Reason
        DisplayCompCode "MQQRYM"

        ' if the retrieve works, display the message from the BL Manager
        If CompCode = MQCC_OK Then
            MsgBox BufferMsg.message, 64, PLCLASS
        End If
    End If

    ' end the current event - this enables the PL Manager to post new events
    MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
    DisplayCompCode "MQENDE"
End Sub

```

Figure 32. Sample HELLO1: Process Fixed-length Messages from BL.
Object: General, Procedure: ProcessEvent

Close_Click is invoked when the user clicks on **Close** from the *File* menu. See also page 45.

```

Sub Close_Click ()
    ' common exit path
    If vHInst <> 0 Then           ' issue MQSETS only if vHInst is valid
        MQSETS ByVal vHInst, ByVal MQSTATE_END, CompCode, Reason
        DisplayCompCode "MQSETS"
    End If

    ' unregister the previously registered class
    MQUREG ByVal vPLClass, ByVal OAK1.hWnd, ByVal MQURGO_FORCE, CompCode, Reason
    DisplayCompCode "MQUREG"

    End                               ' end program execution
End Sub

```

Figure 33. Sample HELLO1: Menu Item Close.
Object: General, Procedure: CloseClick

MenuHelloBL_Click is invoked when the user clicks on the **Hello BL** item from the *File* menu. MQSEND sends the message BLRequest to the instance "Beech" of the class "hellobl1". The values are defined in Figure 37 on page 58.

```
Sub MenuHelloBL_Click ()
  Dim BufferMsg As HELLO ' buffer - NB Don't define as String

  BufferMsg.message = "Hello BL Method"
  ' send request BLREQUEST to class BLCLASS
  MQSEND ByVal vHInst, ByVal BLCLASS, ByVal BLINSTANCE, ByVal BLREQUEST,
    0, BufferMsg, CompCode, Reason
  DisplayCompCode "MQSEND"
End Sub
```

Figure 34. Sample HELLO1: Send a Fixed-length Message to the BL.
Object: MenuHelloBL, Procedure: Click

Form_Unload is called when the user clicks on **Close** from the system menu.

```
Sub Form_Unload (Cancel As Integer)
  Close_Click ' common exit path
End Sub
```

Figure 35. Sample HELLO1: Close the Window.
Object: Form, Procedure: Unload

MenuExit_Click is called when the user clicks on **Exit** from the *File* menu.

```
Sub MenuExit_Click ()
  Close_Click ' common exit path
End Sub
```

Figure 36. Sample HELLO1: Exit the Program.
Object: MenuExit, Procedure: Click

The procedure in Figure 32 on page 55 issues three MQ3T API calls:

- **MQQRYE (MQ Query Event)** stores the properties of the event in the event structure MQevent. The structure is explained on page 250 of the *Application Programming* manual.
- **MQQRYM (MQ Query Message)** stores the properties of the message in the MQMP structure MsgParams and the message itself in the buffer BufferMsg. You need this call to gain access to the message data. The MQMP structure is explained on page 255 of the *Application Programming* manual.
- **MQENDE (MQ End Event)** indicates that the program has processed the event and allows another event to occur.

The following tables describe the parameters of the API calls.

Table 7. MQQRYE Parameters

Parameter	Description
ByVal HInst (input)	The name of the instance that issues the call. The name was saved in the OAK1_NewEvent procedure, shown in Figure 23 on page 48, and passed to the ProcessPEvent procedure (Figure 40 on page 60).
MQevent (input)	This structure contains data that describe the <i>event</i> . This example refers to two fields in the structure: <ul style="list-style-type: none"> • MQevent.RuleId contains the ID of the rule that caused the event: RI_BLREPLY. • MQevent.MaxBufferLength contains the size of the buffer you need to store the longest message. Refer to page 250 of the <i>Application Programming Guide</i> .
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

Table 8. MQQRYM Parameters

Parameter	Description
ByVal HInst (input)	The name of the instance that issues the call. The name was saved in the OAK1_NewEvent procedure, shown in Figure 23 on page 48, and passed to the ProcessPEvent procedure (Figure 40 on page 60).
ByVal 1 (input)	The number of the message you want to query as defined in the <i>rule</i> that triggers the event.
MsgParams (input)	This structure describes the message properties. In this example we do not use any. Refer to page 255 of the <i>Application Programming Guide</i> .
BufferLen (input)	This is the length of the message. It was obtained with the MQQRYE call (see table above).
BufferMsg (output)	The buffer that holds the message. In this example we display the field "BufferMsg.message".
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

Table 9. MQENDE Parameters

Parameter	Description
ByVal HInst (input)	The name of the instance that issues the call. The name was saved in the OAK1_NewEvent procedure, shown in Figure 23 on page 48, and passed to the ProcessPEvent procedure (Figure 40 on page 60).
ByVal MQSTATE_USER (input)	The state of the instance. You may specify any value except MQSTATE_NEW.
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

3.5.2.3 HELLO1H.BAS

This file declares data structures used by the HELLO1.FRM sample application when communicating with a BL.

HELLO1.BAS, shown in Figure 37, is similar to HELLO1.H in the C sample.

```
'
' Rule id for STARTJOB
'
Global Const RI_STARTJOB = 0&

' Rule IDs for HELLO1
'
Global Const RI_BLREQUEST = 1&
Global Const RI_BLREPLY = 2&
'
' Classes
'
Global Const PLCLASS = "hellogu1"
Global Const BLCLASS = "hellobl1"
'
' Instances
'
Global Const PLINSTANCE = "Oak"
Global Const BLINSTANCE = "Beech"
'
' Messages
'
Global Const STRTJOB = "StartJob"
Global Const BLREQUEST = "BLRequest"
Global Const BLREPLY = "BLReply"
'
' Global variables
'
Global vHInst As Long           ' instance handle
Global vPLClass As String * 12  ' PL classname

===== The following code is for HELLO1H.BAS =====
'
' Hello structure
'
Global Const HELLOLENGTH = 20   ' message length

Type HELLO                       ' user defined type
  message As String * HELLOLENGTH ' message data
End Type

===== The following code is for HELLO2H.BAS =====
'
Global Const ID_HELLO_MSG = 1    ' hello message element id in
                                  ' variable-format message
```

Figure 37. HELLO1H.BAS and HELLO2H.BAS Files

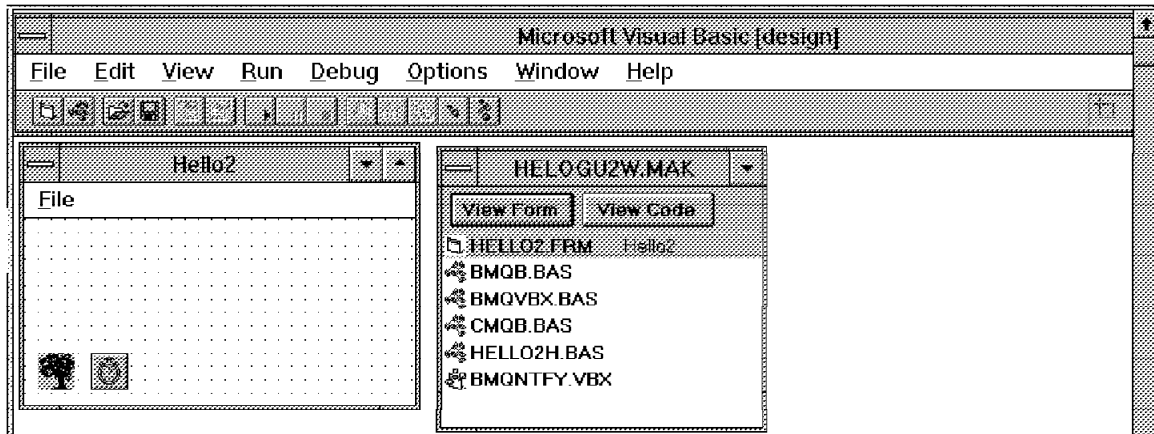


Figure 38. Sample HELLO2: HELOGU2W.MAK

3.5.2.4 HELOGU2W.MAK

This file is the project file for a simple sample program that uses variable-format messages. The program is similar to the HELLO2 C sample supplied with MQ3T.

This sample can be compiled and the executable file HELLOGU2W.EXE placed in the 3TIERW\SAMPLES\BIN directory.

To run the client part of HELLO2, start the PLM, and issue a STARTJOB message. The sample will run using a Visual Basic PL instead of the usual C GUI program.

The difference between HELLO1 and HELLO2 is that HELLO1 sends and receives fixed length messages while HELLO2 demonstrates how to send and receive variable length messages.

```

Sub MenuHelloBL_Click ()
  Dim hSet As Long      ' new for HELLO2
  Dim NL
  Dim Message As String * 16      ' length of string important in VB

  ' create a set for message - new for HELLO2
  MQCRTS hSet, ByVal MQSL_DEF_SET_LENGTH, CompCode, Reason
  DisplayCompCode "MQCRTS"

  NL = Chr(10)          ' new-line character required by BL
  Message = "Hello BL Method" & NL
  MQADDC ByVal hSet, ByVal ID_HELLO_MSG, ByVal 16,           on one
    ByVal MQRPLC_YES, CompCode, Reason                       line !
  DisplayCompCode "MQADDC"

  ' send request BLREQUEST to class BLCLASS - changed for HELLO2
  MQSEND ByVal vHInst, ByVal BLCLASS, ByVal BLINSTANCE,    on one
    ByVal BLREQUEST, ByVal 0, ByVal hSet, CompCode, Reason  line !
  DisplayCompCode "MQSEND"
End Sub

```

Figure 39. Sample HELLO2: Send a Variable-length Message to the BL

3.5.2.5 HELLO2.FRM

This sample of code provides the dialog box and simple code to send a variable-format message to a BL. By comparing the code with the HELLO2 C sample direct comparisons between C and Visual Basic can be made. HELLO2.FRM is similar to HELLOGU2.C in the C sample.

MenuHelloBL_Click is invoked when the user clicks on the **Hello BL** item from the *File* menu. The MQ3T API MQCRTS (MQ Create Set) creates a set for the variable length message. MQADDC (MQ ADD Character string) adds one character-string element that has the element ID 1. Its length is 16 bytes. Note that elements must be a multiple of four bytes. MQSEND sends the message BLRequest to the instance "Beech" of the class "hellobl1". The values are defined in Figure 37 on page 58.

ProcessPLevent is called from *NewEvent* when a rule is satisfied. It retrieves the data of the reply message and displays it in a message box. Here the MQQRYM refers to a set. MQCPYC copies the message element 1 into the message buffer HelloMsg.

```
Sub ProcessPLevent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure
    Dim MsgParams As MQMP            ' message parameters
    Dim BufferLen As Long             ' buffer length

    Static HelloMsg As String * 100 ' new for HELLO2
    Dim hSet As Long

    ' rule clicked - enable "Hello BL" in "File" menu
    MenuHelloBL.Enabled = True

    ' query information about the current event
    MQQRYE ByVal HInst, MQevent, CompCode, Reason
    DisplayCompCode "MQQYRE"

    ' if the rule is RI_BLREPLY, retrieve the message data and display it
    If MQevent.RuleId = RI_BLREPLY Then
        BufferLen = MQevent.MaxBufferLength

        ' changed for HELLO2
        MQQRYM ByVal HInst, 1, MsgParams, BufferLen, hSet, CompCode, Reason
        DisplayCompCode "MQQYRM"

        ' if the retrieve works, display the message from the BL Manager
        If CompCode = MQCC_OK Then
            BufferLen = MQevent.MaxBufferLength

            ' new for hello2
            MQCPYC hSet, ID_HELLO_MSG, BufferLen, ByVal HelloMsg, CompCode, Reason
            DisplayCompCode "MQCPYC"

            MsgBox HelloMsg, 64, PLCLASS ' changed for hello2
        End If
    End If

    ' end the current event - this enables the PL Manager to post new events
    MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
    DisplayCompCode "MQENDE"
End Sub
```

Figure 40. Sample HELLO2: Process Variable-length Messages from BL

3.5.2.6 HELLO2H.BAS

This file declares data structures used by the HELLO2.FRM sample application when communicating with a BL.

HELLO2.BAS is similar to HELLO2.H in the C sample. The file is shown in Figure 37 on page 58.

3.5.3 Sample Read Only Code Fragments

The following files are installed in two sample directories:

C:\3TIERW\SAMPLES\VB\PFCUST
C:\3TIERW\SAMPLES\VB\PCUST

The code is taken from a larger program, it is not intended to be run in its present form. Although it is easy to start a Visual Basic program while looking at the form, the code has been modified to cause a dialog box to be displayed before the program ends.

Note: DEMOH.BAS, DATADICT.BAS and TRAN.BAS are copied into both directories.

3.5.3.1 PFCUST.MAK

This project file, allows the PFCUST.FRM form to be loaded into Visual Basic. It provides the required infrastructure and includes files from the support directory C:\3TIERW\VBSUPP.

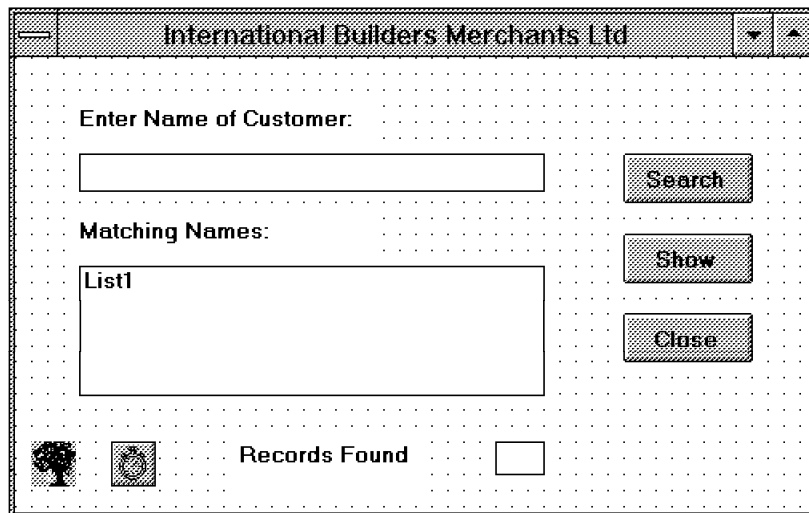


Figure 41. Sample PFCUST: Find Customer GUI

3.5.3.2 PFCUST.FRM

This sample code shows an example of a Find Customer PL form. The user can enter the name of a customer and then click on a search button.

When the button is clicked, the PL sends a message (using a PLM API) to a remote database. Subsequently when a reply is received, the PLM notifies that PL that the reply is available and the PL uses the PLM APIs to retrieve the reply data, and then displays it for the user.

This source file demonstrates how to invoke the main MQ3T API calls. It also shows the use of custom control BMQNTFY.VBX, which has been placed on the form with the name OAK1. The function OAK1_NewEvent is used to detect an event from the PL Manager.

3.5.3.3 PFCUST.BAS

This file declares data structures used by the Find Customer application when communicating with BLs and DLs.

3.5.3.4 DEMOH.BAS

This file contains examples of class and instance names, and declares global variables used in the sample code.

3.5.3.5 DATADICT.BAS

This file contains database constants used in the sample code.

3.5.3.6 TRAN.BAS

This file contains the transaction codes used between the PLs, BLs and DLs.

3.5.3.7 PCUST.MAK

This project file, allows the PCUST.FRM form to be loaded into Visual Basic. It provides the required infrastructure and includes files from the support directory C:\3TIERW\VBSUPP.

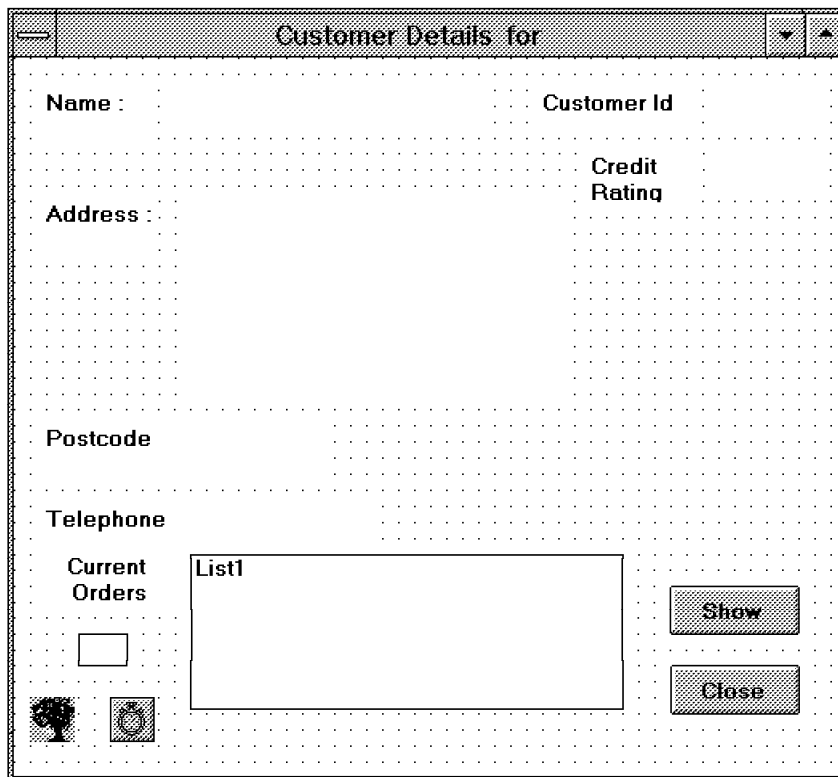


Figure 42. Sample PCUST: Customer Details GUI

3.5.3.8 PCUST.FRM

This is a second sample of code that relates to the PFCUST sample above. When the user highlights one of possibly several matching names identified on the PFCUST GUI and clicks on the **Show** button, then the PCUST PL is started. PCUST.FRM is a Customer Details PL (Visual Basic form) which sends a request message to recover details about the selected customer. When a reply is received, the PLM tells the PL (causes the appropriate event to occur) that a reply is available and the PCUST PL uses the PLM APIs to obtain the customer details. PCUST then paints the GUI with the received information.

This program demonstrates how to invoke the main MQ3T API calls.

3.5.3.9 PCUST.BAS

This file declares data structures used by the Customer Details application when communicating with BLs and DLs.

3.5.4 A Template for Your Own Program

The following files are installed in the TEMPLATE directory:

C:\3TIERW\SAMPLES\VB\TEMPLATE

The program can be used as a stepping-stone to writing other Visual Basic GUI programs. Before using the files in this directory, copy them to a new directory. Two places where the code needs modifying are displayed in a dialog box and these can be found easily in TEMPLATE.FRM

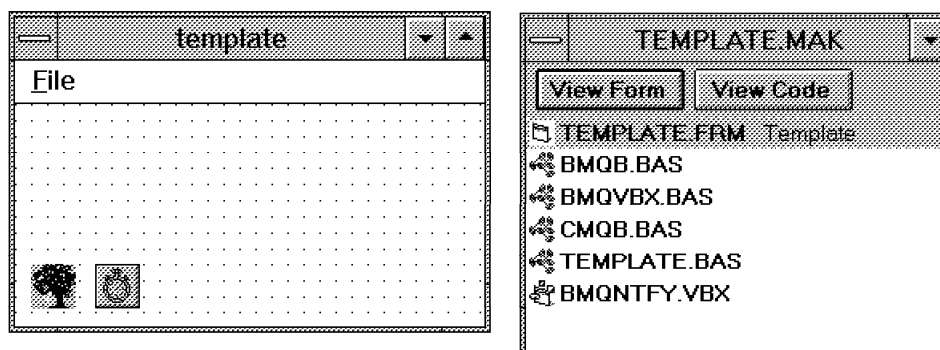


Figure 43. Template GUI Provided with SupportPac

3.5.4.1 TEMPLATE.MAK

This file is the project file for a simple template sample program. The program is similar to the HELLO1 Visual Basic program and can be used as the starting point for subsequent PL program development.

This sample can be compiled and the executable file TEMPLATE.EXE placed in the 3TIERW\SAMPLES\BIN directory.

To run the client part of TEMPLATE, start the PLM, and issue a STARTJOB message (for HELLO1). The sample will run using a Visual Basic PL but instead of the usual HELLO1 code, the user is requested to add their own code.

3.5.4.2 TEMPLATE.FRM

This sample of code provides the dialog box and simple code to send a fixed-format message to a BL. Initially the program requests the user to add their own code, but optionally, fragments from HELLO1 can be run to achieve a working program.

3.5.4.3 TEMPLATE.BAS

This file declares data structures used by the TEMPLATE.FRM sample application when communicating with a BL.

3.5.5 Run-Time Utility

The following files are installed in the SPEEDUP directory:

C:\3TIERW\SAMPLES\VB\SPEEDUP

Once this program is running, a STARTJOB messages will cause PLs to start with an appreciable delay. Providing SPEEDUP is left running, the Visual Basic Run-time support DLL will be available to start PLs more quickly.

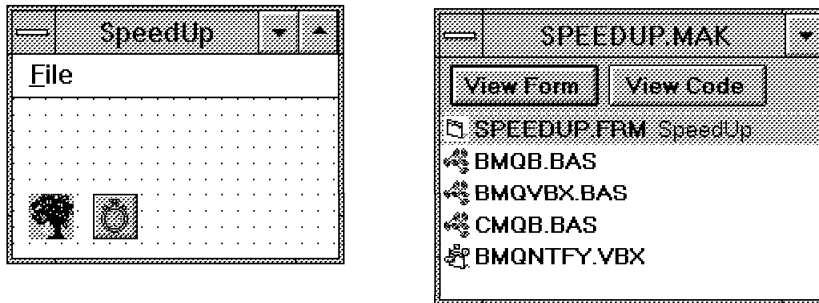


Figure 44. SPEEDUP Program: GUI

3.5.5.1 SPEEDUP.MAK

This file is the project file for a simple speed-up program. This sample can be compiled and the executable file SPEEDUP.EXE placed in the 3TIERW\SAMPLES\BIN directory. Optionally the program can be copied into the Windows Startup folder.

3.5.5.2 SPEEDUP.FRM

This sample of code provides a means to load and verify the Custom Control and Visual Basic run-time DLL. If HELLO1 is run on first starting WINDOWS, there will be a delay from issuing a STARTJOB message until the PL is visible. The delay is caused by the HELLO1 sample having to load the Visual Basic Run-time support DLL. The program executes in the minimized state.

3.6 Application Programming Interface Calls

This section describes the 3T application programming interface (API) calls; they are listed in alphabetic order. For a more detailed description of each call, refer to the book *MQSeries Three Tier Application Programming*, SC33-1452-00.

3.6.1 Types of API Calls

3T provides the following three types of API calls:

- Base
- Presentation Logic Manager
- Self-Defining Data Manager

3.6.1.1 Base Calls

MQSEND	Send a named request or inform message
MQXSEND	Send a fully-specified request or inform message
MQRPLY	Send a named reply message
MQXRPLY	Send a fully-specified reply message
MQTIME	Set a timeout
MQLOG	Write to the log file
MQQRY	Query the server, a class, or an instance
MQQRYM	Query a message

3.6.1.2 Presentation Logic Manager Calls

MQREG	Register a presentation logic (PL) program
MQUREG	Unregister a presentation logic program
MQQRYE	Query an event
MQENDE	End an event
MQSETS	Set the state of an instance
MQENDP	End the Presentation Logic Manager

3.6.1.3 Self-Defining Data Manager calls

MQCRTS	Create a set
MQADDB	Add an element to a set from a buffer
MQADDC	Add a character-string element to a set
MQADDI	Add an integer element to a set
MQCMPB	Compare element data with a buffer
MQCMPC	Compare character-string element data with a character string
MQCMPE	Compare set elements
MQCMPI	Compare integer-element data with an integer
MQCPYB	Copy an element into a buffer
MQCPYC	Copy character-string element data into a string
MQCPYE	Copy an element to another set
MQCPYI	Copy integer-element data to an integer
MQDELA	Delete all elements from a set
MQDELE	Delete an element from a set
MQDELS	Delete a set

MQQRYS	Query a set
MQVALS	Validate set data

The above MQ3T APIs for Visual Basic are defined in BMQB.BAS and are summarized in Appendix B, "Summary of MQ3T APIs" on page 251.

3.6.2 Notes to API Calls

We want to bring four topics regarding the Visual Basic API calls to your attention:

3.6.2.1 SDDM Validity Checking

Although the SDDM calls provide limited validity checking of the buffers passed on the MQADDB, MQCRTS, and MQVALS calls, primarily it is the responsibility of the application developer to ensure the validity of the buffer. The MQCPYB call issues a warning if the buffer you specify is too small, but the call will still complete.

3.6.2.2 SDDM Error Handling

In many error situations, the SDDM API calls attempt to complete, at least partially. To help them do this, 3T provides a null set handle (MQSH_NULL) which you can use in the SDDM calls in place of a valid set handle. When the calls use the null set handle, they return a warning.

3.6.2.3 Visual Basic Calls by Reference or by Value?

Visual Basic defines all calls by reference by default. The ByVal keyword is used to pass by value. Where a parameter is used only for input, it is declared ByVal. Strings are always declared ByVal which forces Visual Basic to convert the internal form of the string (length and string) to a null terminated string as used by the MQ3T DLL BMQAPICW.DLL which is provided as part of MQ3T for Windows.

Although the ByVal can be omitted, in the call, the keyword is used by the sample programs to enhance readability. Except for strings, calls with ByVal are calls by value, on return from the function, the parameter is not changed.

3.6.2.4 Declarations of Complex Data Types

Some of the API calls make use of complex definitions for PMQCHAR, MQEVENT, and MQMP which are declared in BMQB.BAS; otherwise all the parameters are declared as either Any, Long or String.

3.7 Using the Visual Basic 3T Sample Programs

The MQSeries Three Tier Visual Basic Support for Windows Client includes two sample programs that show how to use the 3T APIs from Visual Basic.

The samples HELLO1 and HELLO2 are analogous to the MQ3T C samples. These can be run and used to confirm that the installation has been successful.

The files are in the directory C:\3TIERW\SAMPLES\VB.

Note: There are HELLO1 and HELLO2 directories within C:\3TIERW\SAMPLES\C. Those examples are written in C. They must be compiled before you can run them.

3.7.1 Preparations on the AIX Server

The following instructions are applicable for the September 1995 released version of the MQSeries 3T product. Subsequent versions of MQSeries 3T will likely not require all the following steps/actions to be performed in order to make the HELLO1 sample.

Assumptions:

- MQSeries for AIX and MQSeries Three Tier for AIX is installed.
- The MQSeries 3T for AIX installation directory is /usr/lpp/mq3t.
- The MQ3T user directory is /home/mq3t.

Set up and Compile:

The following steps show how to compile the HELLO1 sample on the AIX server:

Step 1. Copy the sample code from the installation directories into your home directory:

```
cp /usr/lpp/mq3t/samples/c/hello1/* /home/mq3t
```

Step 2. Change the owner and group permission on the files. You will need to be a root user to do this.

```
su root
password: ???
chown mq3t *
chgrp mq3t *
chmod 755 *
exit
```

Notes:

- a. 755 changes the permission to read/write and execute for the owner, to read and write for group users, and read and execute for all other users.
- b. Make sure that you are still a root user when you execute the exit command. Type id and check if the user id is root:

```
id
uid=0(root) gid=... ..
```

Step 3. Ensure that the language environment variable, LANG, is set correctly to the appropriate user's language (not a programming language):

```
LANG=En_US; export LANG
```

Reboot the system if the change is needed.

- Step 4.** Ensure that a link is made in the /usr/include directory for the files bmqc.h and cmqc.h. To check this issue the ls commands below and see if the results match the line following the command.

```
ls -la /usr/include/bmqh
lrwxrwxrwx 1 root system ...
    /usr/include/bmqc.h -> /usr/lpp/mq3t/include/bmqc.h
```

```
ls -la /usr/include/cmqh
lrwxrwxrwx 1 root system ...
    /usr/include/cmqc.h -> /usr/lpp/mq3t/include/cmqc.h
```

Note: The cmqc.h link should have been set up as part of the MQSeries base installation.

If a link is not done issue one or both of the following commands:

```
ln -fs /usr/lpp/mq3t/include/bmqc.h /usr/include/bmqc.h
ln -fs /usr/lpp/mq3t/include/cmqc.h /usr/include/cmqc.h
```

Remember, the syntax for the ln command is:

```
ln "real file" "phantom file"
```

- Step 5.** Ensure that the INCLUDE environment variable is set:

```
INCLUDE=/usr/include ; export INCLUDE
```

- Step 6.** Copy or link the file heloms1x.ch to helloms1.ch. To link type:

```
ln -fs heloms1x.ch helloms1.ch
```

- Step 7.** Copy or link the file hello1x.h to hello1.h. To link type

```
ln -fs hello1x.h hello1.h
```

- Step 8.** Proceed to "make" all files needed on AIX for the HELLO1 sample code. You do this by executing the *make* utility on the file with *.mak* appended to it.

```
make -f hello1cx.mak
```

- Step 9.** If you see any errors it is better to start from scratch by deleting all files in the /home/mq3t/hello1 directory and repeating steps one through seven.

- Step 10.** Start the default queue manager:

```
strmqm
```

- Step 11.** Make sure that the queue manager default objects are created. To do this issue the following command:

```
runmqsc < amqscoma.tst > coma.log
```

The file amqscoma.tst is in the directory /usr/lpp/mqm/samp. You have to issue this command only once.

- Step 12.** Make sure that the queues for the HELLO samples are created. To do this issue the following command:

```
runmqsc < sampcoma.tst > coma1.log
```

The file is in the directory /usr/lpp/mq3t/samples. You have to issue this command only once.

Important

The queue manager must be running to execute RUNMQSC.

Note: Prior to step 8 you may wish to get a comparison of the files created to those prior to the "make" command in order to better understand what the "make" step accomplishes for one. One way to do this is as follows:

1. Prior to step 8 log the current directory file names:
`ls >foo.names`
2. During "make" step capture errors into log and view on screen at the same time:
`make -f hello1cx.mak | tee make1.log`
3. Immediately after step 8 log the current directory file names again:
`ls >foo1.names`
4. Compare the before and after files names and review.
`sdiff foo.names foo1.names`

Figure 45 on page 70 illustrates the output that one should see as a result of the "make" process in step 8. It would be captured in make1.log if the second step above was performed.

The following table shows the file comparison of names of files before step 8 and those created after step 8 as performed according to the note on page 69.

Table 10. Files for HELLO1 Sample

Before the make (step 8)	After the make (step 8)
	HELLO.MFF
	STARTJOB.MFF
	hello1.h
hello1cx.mak	hello1cx.mak
hello1st.h	hello1st.h
hello1x.c	hello1x.c
hello1x.h	hello1x.h
	hellobl1
	hellobl1.c
	hellobl1.exp
	hellobl1.mak
	hellobl1.map
	hellobl1.o
	hellobl1.u
	hellogu1.cb
hellogu1.cs	hellogu1.cs
	helloms1.ch
hellopr1.ch	hellopr1.ch
	helob1cx.cb
helob1cx.cs	helob1cx.cs
helob1cx.prf	helob1cx.prf
heloms1x.ch	heloms1x.ch
libmain.c	libmain.c
	libmain.o
	libmain.u

```

;bmqcc hellogu1.cs

Class Compiler.
Version 1.00.000. Sep 6 1995
(C) Copyright IBM Corporation 1994, 1995.
All rights reserved.

*** Parsing the class source file 'hellogu1.cs' ***
.....

*** Checking the msgin/msgout of class hellogu1 ***

*** CHECKED ***

*** Creating binary class file 'hellogu1.cb' ***

*** CREATED ***
    bmqcc /s helobl1cx.cs

Class Compiler.
Version 1.00.000. Sep 6 1995
(C) Copyright IBM Corporation 1994, 1995.
All rights reserved.

*** Parsing the class source file 'helobl1cx.cs' ***
.....

*** Checking the msgin/msgout of class hellobl1 ***

*** CHECKED ***

*** Creating binary class file 'helobl1cx.cb' ***

*** CREATED ***

*** Generating skeleton files ***

*** GENERATED 'hellobl1.c', 'hellobl1.mak' and 'hellobl1.exp' ***
    make -kf hellobl1.mak
    xlc_r -g -c -Dsigned= -Dvolatile= -D_Optlink -I. -M libmain.c
    xlc_r -g -c -Dsigned= -Dvolatile= -D_Optlink -I. -M hellobl1.c
    xlc_r -L. -lXm -lXt -lXl1 -L/usr/lpp/mq3t/lib -lbmqpic -e LibMain -bM:SRE -bE:hellobl1.exp
-bmap:hellobl1.map libmain.o hellobl1.o
    mv a.out hellobl1

```

Figure 45. Messages for MAKE of HELLO1 Sample

3.7.2 Preparations on the Windows Client

The following directories are of interest:

C:\3TIER\BIN contains 3T executables, such as STRPLMW.EXE, STRJOBW.EXE, and ENDPLMW.EXE.

C:\3TIER\VBSUPP contains custom controls and bit maps required for Visual Basic.

C:\3TIER\SAMPLES\BIN contains class binary files. This directory is in the search path, see AUTOEXEC.BAT on page 39.

Important

All executables and class binary files must be in the directory \3TIER\SAMPLES\BIN unless you add your own directory to the path.

C:\3TIER\SAMPLES\VB contains subdirectories for the samples, HELLO1 and HELLO2, and for the code fragments.

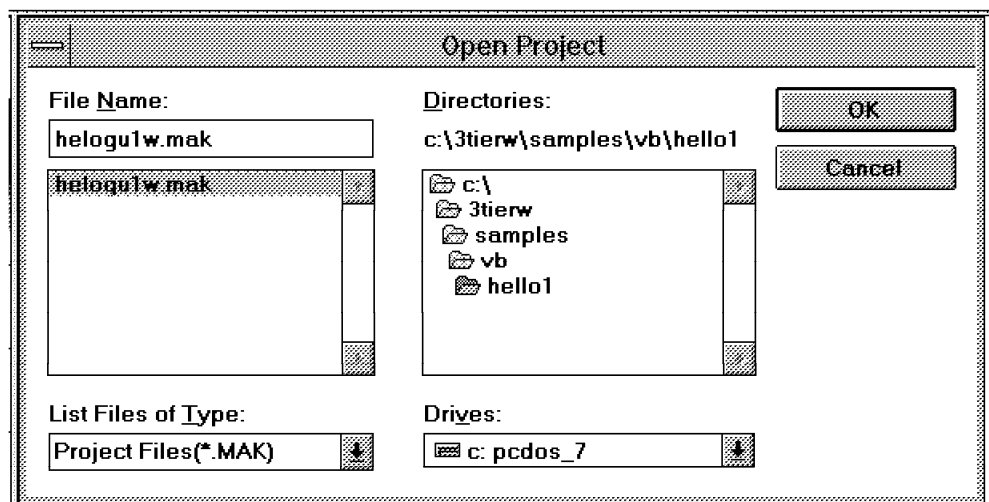
The HELLO1 files should be in a directory path as follows:

C:\3TIER\SAMPLES\VB\HELLO1

C:\3TIER\SAMPLES\C contains Windows sample programs written in C. We do not use them because we do not have a C compiler installed in our Windows workstation. However, we use other files in this directory to execute the Visual Basic programs.

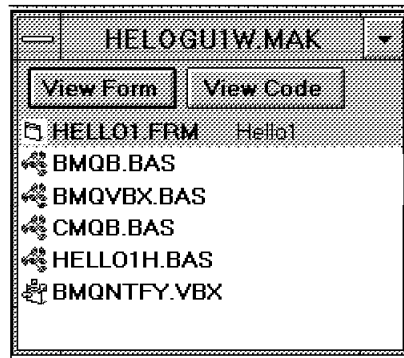
To look at the HELLO1 sample follow these steps:

- Bring up Visual Basic. You will see a "Form1" displayed. Ignore it.
- Select **Open Project** from the *File* menu.
- In the Open Project window select **hello1** from the directory c:\3tier\samples\vb.
- Click on **helogu1w.mak** and then on **OK**.

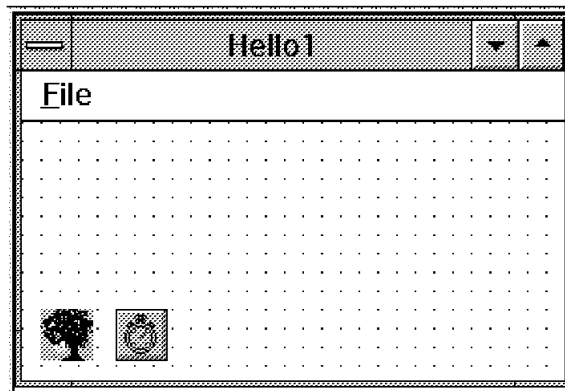


Note: You may minimize the Program Manager window to eliminate it from the Visual Basic window.

- Click on **Project** in the Window menu. This displays a window named HELOGU1W.MAK shown below:



- Click on the **View Form** push button and you will see the Hello1 window displayed.



The Hello1 form shows two icons for custom controls:

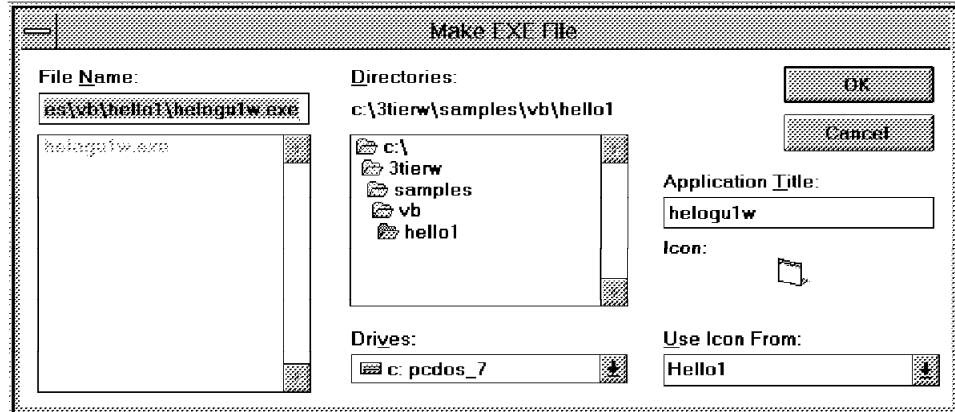
1. The icon for BMOQTFY.VBX appears as an oak tree. This is an IBM-created custom control for 3T.
2. The icon for the TIMER module appears as a stop watch. TIMER is a standard Microsoft Visual Basic module.

The two controls are only visible when the form is created, not at run time.

Important

If the two controls do not appear then make sure that they are in the standard Visual Basic directory for custom controls, usually c:\windows\system.

If in the HELLO1 subdirectory an executable file for the Visual Basic GUI does not exist by the name HELOGU1W.EXE then it needs to be created. Go into Visual Basic and select the **Make .EXE File** option from the File menu.



Make sure the target name is HELOGU1W.EXE. It needs to match this same name that is the hard coded name in the HELLO1 sample class definitions.

The EXE will be stored in the directory C:\3TIERW\SAMPLES\VB\HELLO1. Since this directory is not in a path move the EXE into the directory C:\3TIERW\SAMPLES\BIN.

To run the HELLO1 sample you can use the the class binary file HELOGU1W.CB that is in the directory C:\3TIERW\SAMPLES\BIN. You may also use the .CB file you created in the server. This file, however, requires that the name of the .EXE is HELLOGU1.EXE.

3.7.3 Running the HELLO1 Sample

The HELLO1 sample sends a request message to a business logic in the AIX server. The server responds with a reply message. To run this program follow these steps:

In the server:

- Start the (default) queue manager:

```
strmqm
```

- Start the BLM:

```
strblm heloblcx.prf
```

The profile referenced in the strblm command is shown below. The class name parameter in the profile points to the class binary file "helob1cx.cb". This file contains the class name for the BL, that is "helobl1", defined in "hello1x.h".

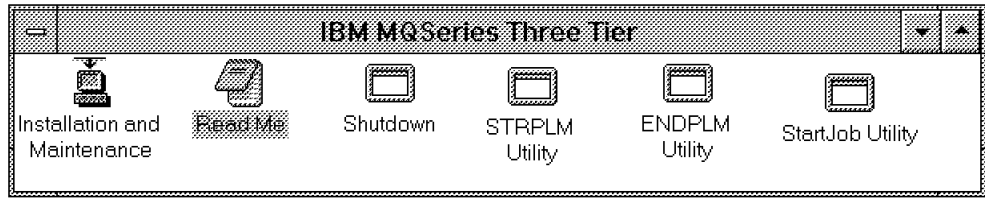
```
[SERVER]
```

```
ClassNames = heloblcx
```

Note: The class name is case sensitive!

In the client:

- Click on the **MQSeries Three Tier** icon in the Program Manager window.



- In the IBM MQSeries Three Tier window click on the **STRPLM Utility** icon.
- The utility should display the profile and a valid path, e.g.

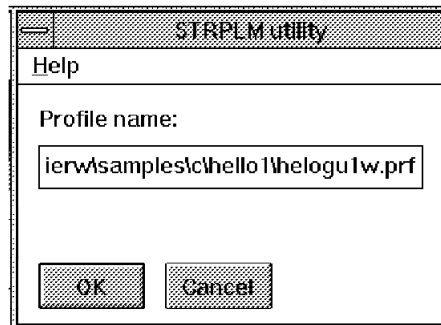
c:\3tierw\samples\c\hello1\hellogu1.prf

Though this profile belongs to the C sample it will work for the Visual Basic sample. It contains the following data:

```
[CLIENT]
```

```
ClassNames = hellogu1
```

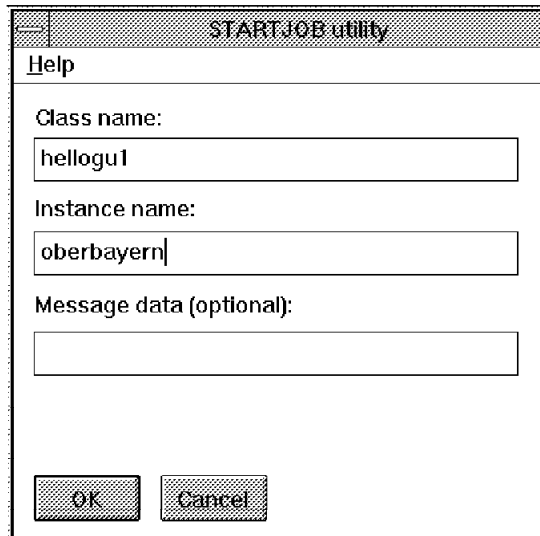
The class name parameter in the profile points to the class binary file "hellogu1.cb". This file contains the class name for the PL, that is "hellogu1", defined in "hello1.h".



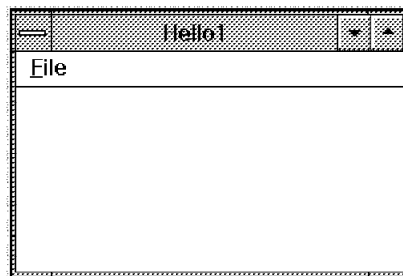
Press OK. A message will be displayed when the presentation logic manager has been started.



- Click on the **StartJob Utility** icon and enter hellogu1 as class name and any instance name. Then click on **OK**.



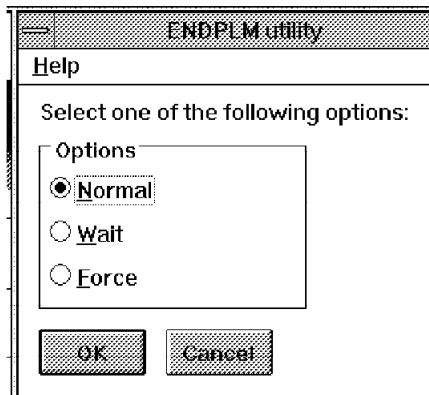
The hello1 GUI should appear.



If not, then the HELLOGU1.EXE is not in a path. Move it into the directory C:\3TIERW\SAMPLES\BIN.

You may check the error log BMQERROR.LOG in the server to find out what the name of the executable has to be.

- To send a message to the BL in the server click on **Hello BL** in the File Menu. You should get a response if the BL in the server is running. If not check the in BMQERROR.LOG for an error message.
- To end the PLM click on the **ENDPLM Utility** icon in the IBM MQSeries Three Tier window. Then select the option **Normal** or **Force** and click on **OK**.



Chapter 4. File Transfer Example

This chapter describes how an existing file transfer application that transfers files between two RS/6000 systems is integrated in an MQSeries Three Tier environment.

The existing sender program gets started with command line arguments. We want to use a GUI to start the program and reduce the sender logic code requirements by using MQ3T APIs. Therefore, we have to write a presentation logic that includes the GUI. In addition, we have to write some code that accepts the message sent by the presentation logic to start the file transfer. The figure below shows the options available to us:

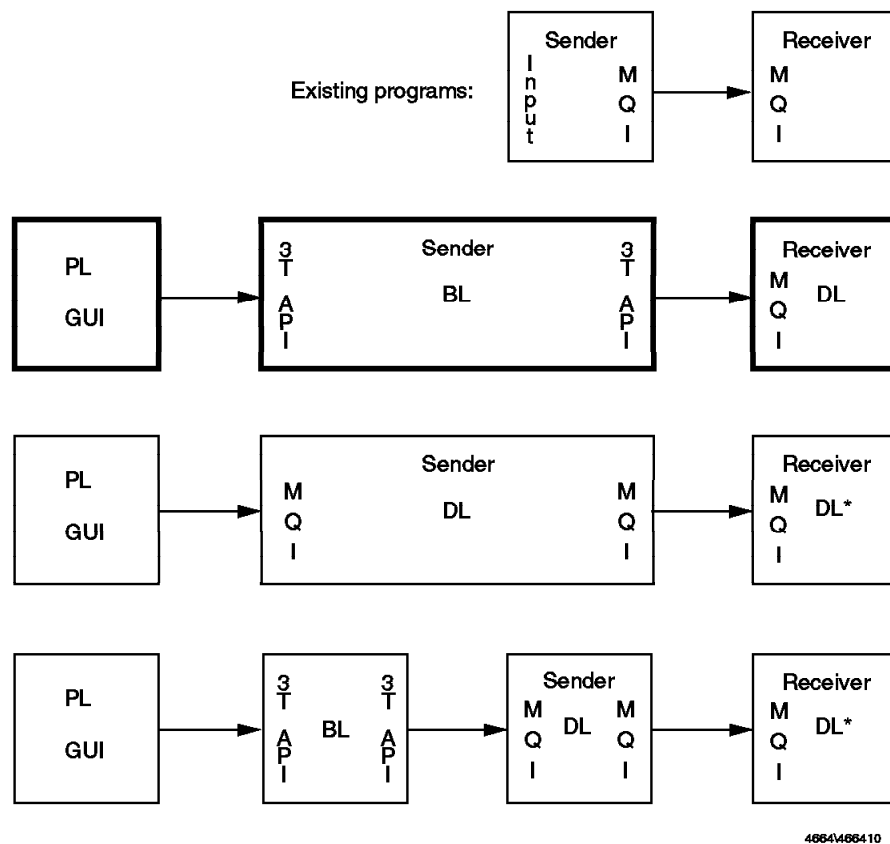


Figure 46. Options for Modification of the File Transfer Program

Note: DL* are MQI-enabled programs that do not require an external class description in 3T since they do not directly communicate with any PL or BL class.

The purpose of this chapter is to demonstrate the use and benefits of the MQSeries Three Tier product in conjunction with the base MQSeries software. By using an example application we hope to accomplish two primary objectives:

1. We demonstrate what a real live MQ3T application would look like that has some real world complexity but not too much that it would obscure the forest with the trees.

2. To show one major benefit of using the MQ3T APIs, namely simplicity via code reduction of 50% or more for business logic API implementation.

Note: In the following sections we describe the existing sender and receiver programs and the first modification option highlighted in Figure 46 on page 77.

Diskette

All example code and files for this demonstration application are included in diskette 1 shipped with this document.

4.1 Application Description

To illustrate the MQ3T API in conjunction with the MQI API two flavors of the demo application were created:

- A pure MQSeries MQI implementation of a file transfer application that uses command line arguments typed in on the AIX sender workstation to control what files to send and what target platform to send them to.
- The second flavor uses MQ3T APIs and is configured as a PL to BL to DL three tier application architecture. Here the MQSeries Three Tier nomenclature is used where PL stands for Presentation Logic, BL is Business Logic, and DL is the Data Logic components in a three level or three tier application architecture as used and described in the MQSeries Three Tier documentation.

For a demonstration of the full functionality of the MQSeries Three Tier product another demonstration application (Chapter 5, “The Bacon Lettuce and Tomato Sandwich” on page 111) was devised to show a pure MQ3T example. It illustrates the OO affinity of MQ3T; the powerful development tools of the product more completely as well as the development methodology. That being said this demonstration application should give an excellent feel of the nuts and bolts requirements for both and MQI and MQ3T application implementation.

Before continuing with the description of the file transfer utility, its set-up and use it would be best noted at this point that a detailed description (almost line by line) of the application is given in the book *Messaging & Queuing Using the MQI* by Burnie Blakeley, Harry Harris and Rhys Lewis. The example code given and described in this book in chapter 12, pages 315 to 353, was used as the base code for the MQI API example.

The sender program was slightly modified for the none-MQ3T version but the receiver program has extensive additions and modifications to implement features of the AIX/UNIX operating system. Of course, the MQ3T (BL) sender program is an extensive modification of the MQI sender version (albeit much simpler). It is highly recommended that this book and this chapter be read in conjunction with this demonstration application.

As the names of the application programs imply, *mqftp* and *mqftpctx*, the base application performs a similar function as the TCP/IP utility “ftp”.

Note: The McGraw Hill version of the receiver program is called “mqftp”, the “mqftpctx” refers to the added extensions and that it is triggered.

It does it, however, using the MQSeries MQI or MQ3T API. Though this application was devised primarily to illustrate the MQSeries APIs it does have over and above these two very real and *practical benefits*:

- It can be used between any two systems using the MQSeries software, even if they are not TCP/IP connected.
- Files can be sent and the target machine does not even have to be up and running as the files will eventually be transferred automatically when the MQSeries channels are reconnected.

With a little more imagination you could even add the capability to the receiver program to *fork* to the AIX/UNIX command line processor or *shell* and have it execute a "shell" script file that may have just been sent over by the sending process. The possibilities of what can now be done with this application becomes very open ended. This in fact is just what the *mqftp* receiver program does.

Available on diskette

Diskette 1 distributed with this book has on it several files you can use to set up this application. Among them are UNIX "shell" script files as well as bit maps and a bit map viewer.

The first flavor (none-3T version) in a nutshell: First you should get the none-MQ3T version of the application working between two AIX boxes. How to do this is described further on in this chapter.

- The receiver program is set up to be run when triggered by the MQSeries Trigger Monitor upon receipt of a message on the designated triggered queue.
- On the sender AIX platform, run the application using the command line format of:
`mqftp MQMname from_file_name to_file_name`
where MQMname is the name of the destination queue manager.
- With this command send over the following files:
 1. The bit map viewer "xv"
 2. The bit maps "challenger.gif" and "toucan.gif"
 3. A shell script file, for example, "foo1.cmd" in Figure 47 on page 80, that can execute the "xv" bit map viewer on the bit map files

The `to_file_name` in the `mqftp` command is special. If the receiving program, `mqftp.c`, finds the name "conan.cmd" it performs its "fork" function and executes this "shell" command file. It displays the bit maps to the screen or any screen that has X-windows connectivity capabilities in its domain.

```

clear
echo "\n\n"
banner MQM " & " MQ3t
echo "\n\n"
ls
pwd
id
export DISPLAY=hostname:0;xv $PWD/challenger.gif &
export DISPLAY=hostname:0;xv $PWD/toucan.gif &
export DISPLAY=rs60001:0;xv $PWD/challenger.gif &
id
echo done!

```

Figure 47. Shell Script File "foo1.cmd"

The second flavor (3T version) in a nutshell: The MQ3T version goes a step or two further accomplishing the same task but with a bit more polish and panash.

- The sender program is converted to an MQ3T BL application program. In the process you will see that the only C code remaining is almost entirely the required C code to open and read the file that is to be transferred to the target system.
- Also, instead of having command line input for the parameters of the sending process we now have an MQ3T PL process running under DOS/WINDOWS and implemented in MicroSoft Visual Basic sending the parameters as a request message to the BL process running on AIX.

The BL process in turn processes the PL message and replies with any and all error messages pertaining to the PL parameter message. If no errors are found it sends the appropriate files to the appropriate target process which is the DL receiver program as before and informs the PL process of the successful completion of this fact.

4.2 Set Up and Run the MQI Application

This section describes:

- How to set up the sender workstation
- How to set up the receiver workstation
- How to run the file transfer demonstration program

4.2.1 Set Up of the Sender Workstation

To load and set up the file transfer programs onto your AIX workstation from the distributed demonstration diskette, perform the following steps:

- Step 1.** Load diskette 1 supplied with this book into the AIX diskette drive.
- Step 2.** Login or sign on to AIX as the *mqm* user.
- Step 3.** Examine the diskette contents with the *dosdir* command.

```

mqm@rs60001 /usr/mqm> dosdir
CMQAIX
C3TAIX
VB3TWIN
README
:

```

- Step 4.** Create a holding sub-directory for the diskette sub-directory contents.
- Step 5.** Make this new directory the working directory.
- Step 6.** Using the dosread command read the cmqaix.z file onto AIX.
- Step 7.** Make sure it was read by doing an ls command.
- Step 8.** Uncompress the loaded file.
- Step 9.** Make sure the uncompress worked by using the ls command.
- Step 10.** Unpack with tar command the uncompressed file.
- Step 11.** List the directory contents to make sure all files were unpacked using the ls command.

The exact commands for the above steps are given below where the dollar sign preceding each command is the normal AIX/UNIX "shell" prompt.

```
$ mkdir cmqaix
$ cd cmqaix
$ dosread cmqaix/cmqaix.z cmqaix.Z
$ ls
cmqaix.Z
$ uncompress cmqaix.Z
$ ls
cmqaix
$ tar xvf cmqaix
```

At the end of these steps you will obtain the files in Table 11.

<i>Table 11. Files for MQI File Transfer Example</i>		
File name	File type	Description
Files to be transferred		
challenger.gif	image	bit map of the space shuttle
first.cmd	cmd file	simple script to test the connection
foo1.cmd	cmd file	script file, see Figure 47 on page 80
toucan.gif	image	bit map of a toucan
xv	executable file	bit map viewer
Files to run the example		
mqftp	executable file	sender MQI program
mqftpctx	executable file	receiver MQI program
Files to develop the example		
ccit.cmd	cmd file	to compile programs
mqftpctx.c	source file	receiver MQI program
mqftp.c	source file	sender MQI program
kshfork	executable file	included in mqftpctx
Utilities		
killmqm	cmd file	cancels MQM processes

The base MQSeries file transfer programs are now successfully installed.

If necessary, change the owner and group of all the files to *mqm* if not set. Do this by switching to the *root* user and using the *chown* and *chgrp* commands. Give all files "read" and "execute" permissions using *chmod* and finally exit from the root user ID.

```
$ su
root's Password: *****
$ chown mqm *
$ chgrp mqm *
$ chmod 755 *
$ exit
$
```

Two more set-up tasks must be performed on the sender AIX machine. Pointing to the receiving AIX machine we have to create:

1. An MQSeries transmission queue
2. An MQSeries channel

To create a transmission queue using the default naming conventions for MQSeries, start the MQSeries command line control server by typing:

```
$ runmqsc
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.
Starting MQSeries Commands.
```

If the target (receiver) system's Message Queue Manager's name is RS60002.MQM then enter the following command:

```
df ql(RS60002.MQM) type(xmit) like(system.default.local.queue)
```

This will create a transmission queue called RS60002.MQM.

If the sender system's Message Queue Manager's name is RS60001.MQM and the TCP/IP name of the receiver system is rs60002 then enter the following command.

```
df chl(RS60001.T0.RS60002) chltype(sdr) trptype(tcp) conname('rs60002') +
xmitq(RS60002.MQM) like(system.def.sender) descr('Sender side')
```

This will create the sender half of the sender/receiver channel between the two AIX machines.

```
ps -ef | grep amq
echo
ps -ef | grep -v grep | grep amq | cut -c10-14 | cat >killfoo
for i in cat killfoo
do
echo killing $i;kill $i
done
rm killfoo
```

Figure 48. Utility "killmqm.cmd"

4.2.2 Set Up of the Receiver Workstation

On the receiver side we need to create:

- The receiver half of the channel we just created for the sender
- A triggered target queue
- A process object

The "runmqsc" commands to create these objects are:

```
define chl(RS60001.TO.RS60002) chltype(RCVR) trptype(tcp) +  
    like(SYSTEM.DEF.RECEIVER) descr('Receiver side')
```

```
define ql('File.Transfer.Queue') like (SYSTEM.DEFAULT.LOCAL.QUEUE) +  
    process(MQFTRPT.P) trigger defpsist(yes) trigtype(DEPTH) +  
    initq(SYSTEM.DEFAULT.INITIATION.QUEUE)
```

```
define process(MQFTRPT.P) like(SYSTEM.DEFAULT.PROCESS) +  
    descr('triggered mqftrp') applcid('/usr/mqm/mqftrptx')
```

Alternatively, you may copy these definitions into two files:

- mqftp.def for the sender machine
- mqftrtx.def for the receiver machine

Modify the message queue manager names and channel names if necessary.

In the *process definition*, with the keyword "applcid", you must specify the directory in which the executable program "mqftrtx" is placed on the receiver AIX machine. In this example it is /usr/mqm.

Notes:

1. It is imperative that the Message Queue Manager, trigger monitor and receiver program "mqftrtx" are all started in the directory specified in the APPLICID field of the trigger process definition. In the examples given so far this would be /usr/mqm. Thus the following commands and program should all be started from this directory:
 - strmqm
 - runmqtrm
 - mqftrtx
2. In the *File.Transfer.Queue definition*, the keyword TRIGTYPE is equal to DEPTH (DEPTH=1 as a default) and this means that once the queue has been triggered, the queue manager disables triggering. For this reason, in the mqftrtx.c program, the trigger is rearmed at the end of the execution. If the program terminates abnormally the trigger is not rearmed and you need a MQSC command (ALTER QL) to set the TRIGGER keyword for the queue.
3. If the mqftrtx program is sent to the receiver AIX system with the *ftp* utility it will be necessary to change the mode of the file to be executable after it has arrived. Use the *chmod* command and type on the AIX command line:
\$ chmod +x mqftrtx

4.2.3 Running the File Transfer Example

Once the MQSeries objects for the proper operation of the file transfer program are in place it is time to start:

1. The message channel on the sender AIX system
2. The trigger monitor on the receiver system

Sender: To start the channel, type the following on the sender AIX command line. A successful completion message should appear once it has started.

```
$ runmqchl -c RS60001.TO.RS60002 &  
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.  
Channel program started.
```

```
$
```

Note: It is most important that the "&" (ampersand) be added to the previous command or the message channel agent program will be started in the foreground and will *not allow you* to get back to the AIX shell without killing it with a CTRL-C console interrupt.

Receiver: Make sure the MQM is started and then start the trigger monitor as follows:

```
$ runmqtrm  
.... MQSeries trigger monitor started.
```

```
.... Waiting for a trigger message
```

Watch out!

It is imperative that you do NOT run the trigger monitor in the background as the shell scripts to be sent across to be executed will not execute if the trigger monitor is started in the background.

Make sure that the File.Transfer.Queue has triggering enabled. To accomplish that use the "runmqsc" utility and modify the queue characteristic directly as follows:

```
$ runmqsc  
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.  
Starting MQSeries Commands.
```

```
alter ql('File.Transfer.Queue') trigger  
1 : alter ql('File.Transfer.Queue') trigger  
AMQ8008: MQSeries queue changed.
```

The receiver system should now be ready to receive files/messages from the sender AIX system. The sender program, mqftp, requires as parameters:

1. The destination queue manager
2. The name of the source file
3. The name of the target file

To actually send a file via the sender program "mqftp" to the triggered receiver program "mqftprtx" type the following on the sender AIX machine:

```

$ mqftp RS60002.MQM first.cmd conan.cmd
Sender: Processing Completed Normally. Reason was 0
Sender: Queue Close Reason Code was 0
Sender: Disconnect Reason Code was 0
$

```

The output that appears on the receiving AIX system trigger monitor's window look like the following:

```

/usr/mqm/mqftprtx "TMC      2File.Transfer.Queue      MQFTP
RT.P
                        /usr/mqm/mqftprtx

                        "      RS60002.MQM

Usage:
  mqftpr
Receiver: Transferring file to conan.cmd
Processing command file now - with fork
doing nothing
done

```

In the last two lines of the output you can see the execution of the script file "first.cmd" which performs two echo commands.

```

#this is a nothing command file
echo doing nothing
echo done

```

Figure 49. Shell Script File "first.cmd"

A bit later, if no further files are sent, the receiving program times out and terminates with the following message waiting to be awoken again by the trigger monitor on receipt of a new message/file.

```

Sender: Processing Completed. Reason was 2033
Sender: Queue Close Reason Code was 0
Sender: Disconnect Reason Code was 0
.... Error starting triggered application.

```

.... Waiting for a trigger message

An error message is displayed by trigger monitor because the receiver program exits with a reason code greater than zero, namely 2033-no more messages in the queue. However, this is not a real error.

Now a file named "conan.cmd" should exist on the receiver AIX system. It is in the user "mqm" home directory and has the following contents:

```

#processing completed

```

This is the result of the receiver program overwriting the command file once it has been executed, turning this file into a no-op command file so it will not be executed a second time mistakenly.

Now the remaining files should be sent across in the following order:

- `mqftp RS60002.MQM xv xv`
- `mqftp RS60002.MQM challenger.gif challenger.gif`
- `mqftp RS60002.MQM toucan.gif toucan.gif`
- `mqftp RS60002.MQM foo1.cmd conan.cmd`

The final file transfer of the `foo1.cmd` file should give a fairly neat effect of displaying several UNIX command outputs on the trigger monitor window and then display nice renditions of the challenger space shuttle and a multi-colored toucan to both AIX machines.

4.3 Set Up and Run the MQ3T Application

The MQI example was developed to execute a file transfer between two RS6000 machines. For the MQ3T version, we added a GUI program for entering parameters and starting the file transfer. The GUI program runs on a Windows 3.1 MQ client workstation. This section describes:

- How to set up the sender workstation
- How to set up the receiver workstation
- How to set up the Windows workstation
- How to run the file transfer demonstration program

4.3.1 Set Up the Sender Workstation

To load and set up the file transfer programs onto your AIX workstation from the distributed demonstration diskette perform the following steps:

Step 1. Load diskette 1 into the AIX diskette drive.

Step 2. Login or sign-on to AIX as the `mq3t` user.

Step 3. Examine the diskette contents with the `dosdir` command.

```
mq3t@rs60001 /homer/mq3t> dosdir
CMQAIX
C3TAIX
VB3TWIN
README
:
```

Step 4. Create a holding sub-directory for the diskette sub-directory contents.

Step 5. Make this new directory the working directory.

Step 6. Using the `dosread` command read the `c3taix.z` file onto AIX.

Step 7. Make sure it was read by doing an `ls` command.

Step 8. Uncompress the loaded file.

Step 9. Make sure the uncompress worked by using the `ls` command.

Step 10. Unpack with `tar` command the uncompressed file.

Step 11. List the directory contents to make sure all files were unpacked using the `ls` command.

The exact commands for the steps on page 86 are given below where the dollar sign preceding each command is the normal AIX/UNIX "shell" prompt.


```

$ mkdir c3taix
$ cd c3taix
$ dosread c3taix/c3taix.z c3taix.Z
$ ls
c3taix.Z
$ uncompress c3taix.Z
$ ls
c3taix
$ tar xvf c3taix

```

At the end of these steps you will obtain the files in Table 12 on page 88.

Note

The File Transfer example has been developed based on the *hello1* sample described in 3.7, “Using the Visual Basic 3T Sample Programs” on page 67.

The base MQ3T file transfer programs are now successfully installed.

If necessary, change the owner and group of all the files to *mqm*. Switch to the *root* user and use the *chown* and *chgrp* commands. Give all files read and execute permissions using *chmod* and finally exit from the root user ID. The commands are shown below:

```

$ su
root's Password: *****
$ chown mqm *
$ chgrp mqm *
$ chmod 755 *
$ exit
$

```

The definitions for the sender AIX machine are the same as for the MQI example. However, there is one more MQI definition:

You need a remote queue that points to the queue in the receiver machine. The name of that queue must match the DL class name, HELLODL1. To keep the definitions independent from those needed by MQ3T you can define an alias queue name for the remote queue. This allows you to change the destination of the messages without modifying the MQ3T application or the user input.

The “runmqsc” commands to create the remote queue in the sender AIX machine are:

```

define qremote(MQFTP) LIKE(SYSTEM.DEFAULT.REMOTE.QUEUE) +
    replace descr('remote queue pointing to File.Transfer.Queue') +
    rname('File.Transfer.Queue') rqmname(RS60002.MQM) xmitq(RS60002.MQM)

define qalias('hellodl1') targq(MQFTP) replace

```

Table 12. Files for MQ3T File Transfer Example

File name	File type	Description
Files to be transferred		
foo1.cmd	cmd file	script file, see Figure 47 on page 80
Files to run the example		
mqftprtx	executable file	receiver (DL) program
hellobl1	executable file	sender (BL) program
hellogu1.cb	class binary file	3T class file for PL
heloblcx.cb	class binary file	3T class file for BL
heloblcx.prf	profile file	startup profile for BLM
Files to develop the example		
hello1cx.mak	makefile	compiles BL class and program files
hellogu1.cs	class source file	PL class
hello1st.h	header file	defines message structures
hello1x.c	source file	BL program in C
hello1x.h	header file	defines constants
hellobl1.c	skeleton file	BL program in C created with bmqcc /s
helloms1.ch	class source file	messages
hellopr1.ch	class source file	external class descriptions
heloblcx.cs	class source file	BL class
libmain.c	dummy file	required for compilation
File output of compilation		
hellobl1.exp	exports	created with bmqcc /s
hellobl1.mak	makefile for hellobl1.c	created with bmqcc /s
hellobl1.map		output of compilation
hellobl1.o	object file	output of compilation
hellobl1.u		output of compilation
libmain.o	dummy file	output of compilation
libmain.u	dummy file	output of compilation
Utilities		
redo	cmd file	to redo MQ3T compilation

4.3.2 Set Up the Receiver Workstation

The definitions for the receiver AIX machine are the same as for the MQI example. You need the same receiver program as for the MQI example, *mqftprtx*. Remember that if the *mqftprtx* program has been sent to the receiver AIX system via the *ftp* utility it will be necessary to change the mode of the file to be executable after it has arrived. Use the *chmod* command and type on the command line:

```
$ chmod +x mqftprtx
```

The program must be in the directory that was defined in the keyword *APPLICID* of the MQI process definition. You can check this by issuing the following commands:

```
$ runmqsc
```

```
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.  
Starting MQSeries Commands.
```

```
display process (MQFTPRT.P) applicid  
1 : display process(MQFTPRT.P) applicid  
AMQ8407: Display Process details.  
  APPLICID(/usr/mqm/mqftprtx)  
  PROCESS(MQFTPRT.P)
```

4.3.3 Set Up the Windows Workstation

To load and set up the Windows 3.1 GUI for the file transfer program follow these steps:

- Step 1.** Load diskette 1 into the A-drive.
- Step 2.** Examine the diskette contents with the *dir* command.

```
C:\>dir  
CMQAIX  
C3TAIX  
VB3TWIN  
README  
:
```

- Step 3.** Create directory that will hold the contents of the *VB3TWIN* directory on the diskette, for example, *C:\VB\VB3TWIN*, and make it the current directory.
- Step 4.** Copy into this directory the files from *VB3TWIN* directory on the diskette.

The commands for the above steps follow:

```
C:\>cd vb  
C:\VB>md vb3twin  
C:\VB>cd vb3twin  
C:\VB\VB3TWIN>copy a:\vb3twin\*.*
```

The directory will contain the files in Table 13 on page 90.

The connection between Windows and AIX machine and the MQ definitions are the same as described in the "hello1" example.

Table 13. Client's Files for MQ3T File Transfer Example		
File name	File type	Description
Files to run the example		
RES3TGTC.EXE	executable file	Presentation Logic (GUI)
HELLOGU1.CB	class binary file	class file for PL
HELLOGU1.PRF	profile	startup profile for PL
Visual Basic files for development		
RES3TGTC.MAK	Visual Basic	project file
RES3TGTC.FRM	Visual Basic	form (GUI)
BMQNTFY.VBX	custom control	intercepts 3T events
BMQC.BAS	header file	3T definitions
BMQVBX.BAS	header file	for custom control
CMQC.BAS	header file	MQ definitions
HELLO1H.BAS	header file	definitions for GUI
Example		
SAMPLE.SET		shows environment

4.3.4 Running the MQ3T File Transfer Example

This application executes programs in three machines. The correct sequence of activation is:

1. Receiver machine (AIX)
2. Sender machine (AIX)
3. Client workstation (Windows)

4.3.4.1 Starting the Receiver

On the receiver AIX machine, start the queue manager and the trigger monitor, using the commands:

```
strmqm
runmqtrm
```

— Watch out! —

It is imperative that you do *not* run the trigger monitor in the background as the shell scripts to be sent across to be executed will not execute if the trigger monitor is started in the background.

4.3.4.2 Starting the Sender

On the sender AIX machine, start the queue manager and the channel between sender and receiver using following commands:

```
$ strmqm
Queue Manager started.
$ runmqchl -c RS60001.TO.RS60002 &
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.
Channel program started.

$
```

Watch out again!

It is most important that the "&" (ampersand) be added to the previous command or the message channel agent program will be started in the foreground and will not allow you to get back to the AIX "shell" without killing it with a CTRL-C console interrupt.

Make sure that the *File.Transfer.Queue* has triggering enabled. This is the queue from which the receiver program will get the file transferred. Use the "runmqsc" utility to enable triggering. Modify the queue characteristics as follows:

```
$ runmqsc
5765-115 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.
Starting MQSeries Commands.
```

```
alter ql('File.Transfer.Queue') trigger
1 : alter ql('File.Transfer.Queue') trigger
AMQ8008: MQSeries queue changed.
```

The sender program is not an MQI application any longer, using MQI APIs, but an MQ3T business logic (BL) using MQ3T APIs. A BL runs under control of a business logic manager (BLM). To start BLM issue the following command from the "c3taix" directory:

```
$ strblm helob1cx.prf
```

```
Business Logic Manager
Version 1.00.000. Sep 6 1995
(C) Copyright IBM Corporation 1994, 1995.
All rights reserved.
```

```
$
```

The "helob1cx.prf" file is the startup profile for BLM. If you wish, add the "&" (ampersand) to this command. However, the better choice is to run the program in foreground, even if it locks the window. This allows you to stop the BLM with a CTRL-C console interrupt instead of killing the process. The latter may cause the BLM to hang up and you may have to stop and restart the queue manager to reset the environment.

4.3.4.3 Starting the Client

In your Windows workstation, bring up Windows and double-click on **IBM MQSeries Three Tier** in the Program Manager's window. This displays the MQ3T window below:

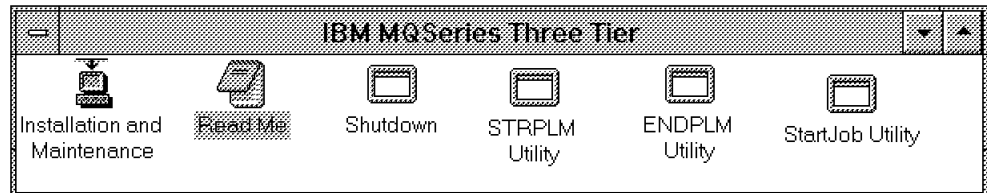


Figure 50. MQSeries Three Tier Window

Use the STRPLM utility to start the Presentation Logic Manager in your workstation. Double-click on that icon and enter path and name of the profile as shown in Figure 51 on page 92.

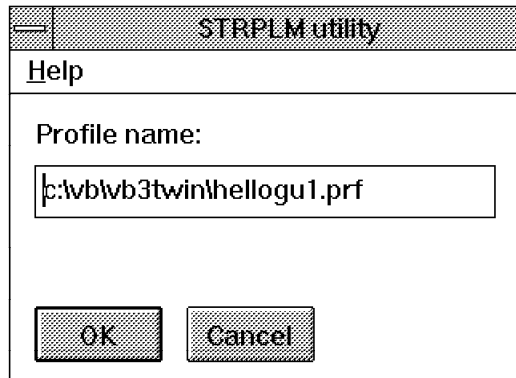


Figure 51. STRPLM Window

The profile tells the PLM what PL classes it has to serve. In this case it has only one class, hellogu1. The file contains two lines:

```
[CLIENT]
Classnames = hellogu1
```

In the STRPLM utility window, click on **OK**. If the queue manager is running in the AIX server machine it responds with this pop-up window:

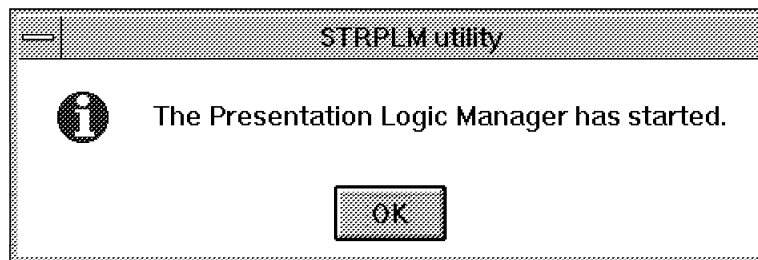


Figure 52. Pop-up: PLM Started

To start the job double-click on the **STARTJOB** utility in the MQ3T window, Figure 50 on page 91. Enter here the class name, hellogu1, and any instance name you like. When you click on **OK** the GUI for the file transfer example, Figure 53 on page 93, appears.

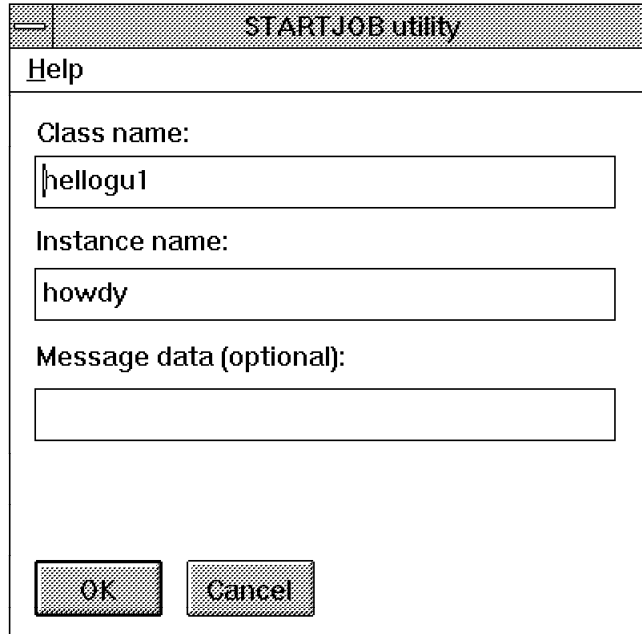


Figure 53. STARTJOB Window

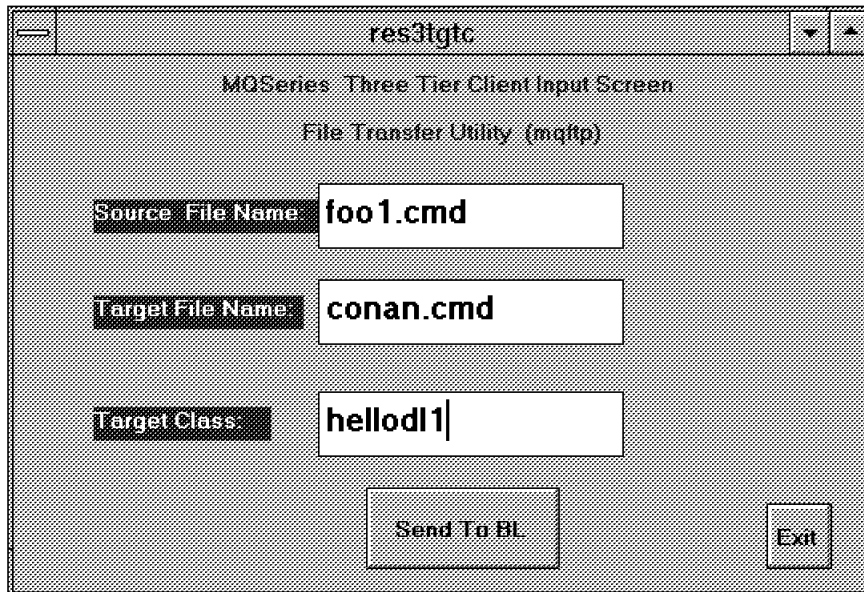


Figure 54. GUI for File Transfer Program

The source file, foo1.cmd is supplied on diskette 1 that accompanies this book. "hellodl1" is the name of the class (program) that receives the file. By default, the class name is also the queue name. Click on the **Send To BL** push button to start the file transfer.

When the PL message is received by the sender AIX machine you will see some messages displayed in the BLM's window. The messages are displayed by the BL program. They are the input parameters for the file transfer program that have been entered in the GUI and the return code of the MQXSEND call issued by the BL program.

```
PL message arguments:
dest_class ---hellod11---
from_file ---fool.cmd---
into_file ---conan.cmd---
comp code = 0
reason = 0
```

The output that appears on the receiving AIX system trigger monitor's window looks like the one displayed in the MQI example. This is because the receiver program, mqftprtx, has not been changed.

```
/usr/mqm/mqftprtx "TMC      2File.Transfer.Queue      MQFTP
RT.P
                        /usr/mqm/mqftprtx

                        "      RS60002.MQM

Usage:
  mqftpr
Receiver: Transferring file to conan.cmd
Processing command file now - with fork
doing nothing
done
```

4.4 Developing the MQ3T Application

In this section we discuss the three programs of the MQ3T file transfer application in detail. We explain:

- What information we defined in the 3T class files
- How 3T uses the profile to rout messages
- What modifications have been made to the sender program
- How the GUI was developed

4.4.1 Defining Class Source Files

The first step in developing an MQ3T application is to write the class source files. Each class describes one object involved in the process. A class definition includes:

- A description of each class the object can exchange messages with
- All messages the class can send and receive
- The names of the methods (programs) that process messages
- Rules that define what method to invoke when a message arrives

In this file transfer example there are two class source files:

- *hello1.cs* for the presentation logic
- *hello1cx.cs* for the business logic

The class source files are based on the "hello1" sample. However, there are some changes required, mostly to provide the 3T infrastructure for the DL class. These changes are marked in the figures on page 96. and explained in the following notes:

- 1** The file *bmqc.h* is the MQ3T product header file.
- 2** The file *hello1x.h* is the application header file. See also **11**.
- 3** The file *helloms1.ch* contains the message definitions. See also **12**, **13**.
- 4** The file *hello1r1.ch*, shown in Table 14, contains external class definitions.
- 5** The ProgName for the PL source file points to the GUI program.
- 6** SourceName in the HelloBLMethod points to the sender program *hello1x.c*.
- 7** The method sends the message DLREQUEST to the receiver AIX machine.
- 8** The new HelloDL1Method processes replies from the receiver AIX machine.
- 9** The BL sends messages to the PL and DL. The destination queue name for the messages defaults to the class name specified in the DESTINATION parameter, DLCLASS. The value for DLCLASS is defined in *hello1x.h*, see **11**.
- 10** The new DLReplyRule, with the ID 100, is satisfied when a reply message from the receiver AIX machine arrives. The rule invokes the method HelloDL1Method.
- 11** The header file *hello1x.h* contains additional definitions for the DL class.
- 12** Since the message is sent from a PC to an AIX machine, data conversion is necessary. Refer to page 302 of the *Application Programming* manual.
- 13** A pair of messages to be sent between BL and DL is added.

Table 14. Class Descriptions - *hello1r1.ch*

PLCLASS	BLCLASS	DLCLASS
CLASSDESC BEGIN ClassName PLCLASS ClassType PL MsgIn STRTJOB, BLREPLY MsgOut BLREQUEST END	CLASSDESC BEGIN ClassName BLCCLASS Harden YES ClassType BL MsgIn BLREQUEST, DLREPLY MsgOut BLREPLY, DLREQUEST END	CLASSDESC BEGIN ClassName DLCLASS ClassType DL MsgIn DLREQUEST MsgOut DLREPLY END

```

#include <bmqc.h>           1
#include "hello1.h"        2

HEADING
BEGIN
  Title "hellogu1 class file"
END
CSINCLUDE "helloms1.ch"   3
CSINCLUDE "hellopr1.ch"   4

METHOD
BEGIN
  MethodName HelloGu1Method
  MethodType PROGRAM
  ProgName RES3TGTC.EXE    5
  StartupTime 10
  Interface PULL
  MsgOut BLREQUEST
END

CLASS
BEGIN
  ClassType PL
  ClassName PLCLASS
  Destination BLCLASS
  RULE
  BEGIN /* startjob message */
    RuleId RI_STARTJOB
    RuleName StartJobRule
    MethodName HelloGu1Method
    MsgIn STRTJOB
  END
  RULE
  BEGIN /* reply from sender */
    RuleId RI_BLREPLY
    RuleName BLReplyRule
    MethodName HelloGu1Method
    MsgIn BLREPLY
  END
END

```

Figure 55. PL Class File "hellogu1.cs"

```

:
MESSAGE /* PL to BL */
BEGIN
  MsgName DLREQUEST
  MsgType REQUEST
  OperationCode OC_BLREQUEST
  Format FIXED
  StrucName HELLO
  StrucFile hello1st.h
  StrucLen 60
  ConversionDLL MQFMT_STRING 12
END
:
MESSAGE /* BL to DL */ 13
BEGIN
  MsgName DLREQUEST
  MsgType REQUEST
  OperationCode OC_DLREQUEST
  Format FIXED
  StrucName HELLOD
  StrucFile hello1st.h
  StrucLen 500
END
MESSAGE /* DL to BL */
BEGIN
  MsgName DLREPLY
  MsgType REPLY
  OperationCode OC_DLREPLY
  Format FIXED
  StrucName HELLOR
  StrucFile hello1st.h
  StrucLen 60
  ConversionDLL MQFMT_STRING
END

```

Figure 56. Class Header File "helloms1.ch"

```

#include <bmqc.h>           1
#include "hello1x.h"        2

HEADING
BEGIN
  Title "helobl1x class file"
END
CSINCLUDE "helloms1x.ch"   3
CSINCLUDE "hellopr1.ch"   4

METHOD
BEGIN
  MethodName HelloBLMethod
  MethodType C_LIBRARY
  ProgName hellobl1.Method1
  SourceName hello1x        6
  MsgOut BLREPLY, DLREQUEST 7
END
METHOD 8
BEGIN
  MethodName HelloDLMethod
  MethodType C_LIBRARY
  ProgName hellobl1.Method2
  SourceName hello1d
END

CLASS
BEGIN
  ClassName BLCLASS
  Harden YES
  ClassType BL
  Destination PLCLASS, DLCLASS 9
  PingTimeout 10
  RULE 10
  BEGIN /* request from BL */
    RuleId RI_BLREQUEST
    RuleName BLRequestRule
    MethodName HelloBLMethod
    MsgIn BLREQUEST
  END
  RULE
  BEGIN /* reply from DL */
    RuleId 100
    RuleName DLReplyRule
    MethodName HelloDLMethod
    MsgIn DLREPLY
  END
END

```

Figure 57. BL Class File "helobl1x.cs"

```

#include <bmqc.h>
#include "hello1st.h" /* message structure */

#define ID_WINDOW 256 /* resource IDs */
#define ID_FILE 300
#define ID_HELLOBL 301
#define ID_EXITPROG 302
#define BLCCLASS "hellobl1" /* classes */
#define PLCLASS "hellogu1"
#define DLCLASS "hellod1" 11
#define PLINSTANCE "Oak" /* instances */
#define BLINSTANCE "Beech"
#define BLREQUEST "BLRequest" /* messages */
#define BLREPLY "BLReply"
#define STRTJOB "StartJob"

#define WM_PLTEST (WM_USER + 1) /* event message ID */

#define RI_STARTJOB 0 /* rule IDs */
#define RI_BLREQUEST 1
#define RI_BLREPLY 2
#define OC_STARTJOB (MQOC_USER) /* operation codes */
#define OC_BLREQUEST (MQOC_USER + 1)
#define OC_BLREPLY (MQOC_USER + 1)
#define OC_DLREQUEST (MQOC_USER + 2) 11
#define OC_DLREPLY (MQOC_USER + 3)

```

Figure 58. Header File "hello1x.h"

4.4.2 Compiling Class Source Files

Compile class source files with the MQ3T class compiler *bmqcc*. This program parses a class source file (.CS) and generates a class binary file (.CB). The class binary file is used by the MQ3T run-time program, either a PLM or a BLM. You may also create skeletons to develop the business logic of the application.

Figure 59 illustrates the compilation of the class source file *hellogu1.cs*. and the creation of the skeleton files for the BL. The commands are in bold print.

```
;bmqcc hellogu1.cs

Class Compiler.
Version 1.00.000. Sep 6 1995
(C) Copyright IBM Corporation 1994, 1995.
All rights reserved.

*** Parsing the class source file 'hellogu1.cs' ***
.....

*** Checking the msgin/msgout of class hellogu1 ***

*** CHECKED ***

*** Creating binary class file 'hellogu1.cb' ***

*** CREATED ***
;bmqcc /s helobl1cx.cs

Class Compiler.
Version 1.00.000. Sep 6 1995
(C) Copyright IBM Corporation 1994, 1995.
All rights reserved.

*** Parsing the class source file 'helobl1cx.cs' ***

*** Checking the msgin/msgout of class hellobl1 ***

*** CHECKED ***

*** Creating binary class file 'helobl1cx.cb' ***

*** CREATED ***

*** Generating skeleton files ***

*** GENERATED 'hellobl1.c', 'hellobl1.mak' and 'hellobl1.exp' ***
```

Figure 59. Compiling a Class Source File

Notes:

1. Ensure that the language environment variable "LANG" is set correctly to the appropriate user's language, such as:
export LANG=En_US
2. Ensure that a link is made in the /usr/include directory for the files *bmqc.h* and *cmqc.h*.

The *cmqc.h* link should have been set up as part of the MQSeries base installation.

For the file *bmqc.h*, supplied with MQSeries 3T, execute the following command:

```
ln -fs /usr/lpp/mq3t/include/bmqc.h /usr/include/bmqc.h
```

3. Ensure that the "INCLUDE" environment variable is set:

```
export INCLUDE=/usr/include
```

4. Copy or link the file "heloms1x.ch" to "helloms1.ch":

```
ln -fs heloms1x.ch helloms1.ch
```

If you compile with the "/s" options the class compiler creates three files:

- hellobl1.c
- hellobl1.mak
- hellobl1.exp

hellobl1.c is a skeleton file that contains one entry point for each of the methods defined in the class file. In this file you will find two entry points, for the HelloBL1Method and one for the HelloDL1Method. For each method the compiler includes an #INCLUDE statement for the file that contains the source code for the method. This name is specified as *SourceName* in the source file.

hellobl1.mak is used to compile the BL program. Start the compilation by typing:

```
make -f hellobl1.mak
```

The above command compiles the C program only. To compile the class source file and then C program in one step use the make file supplied on the diskette, *hello1cx.mak*. Type the following command:

```
make -f hello1cx.mak
```

hellobl1.exp is used for the compilation and contains the following information:

```
#! hellobl1.a  
LibMain  
Method1  
Method2
```

The "redo" command file included in the diskette can be used to reset the environment and to run another compilation of all files.

```
rm *.cb  
touch *.h  
touch *.ch  
make -f hello1cx.mak  
echo "\n"  
echo "Ignore error messages of 'Cannot Open or Does not exist' on Log files\n"  
cat *.LOG  
rm BMQERROR.LOG  
echo "\n"  
echo "Type the following command to start the blm (if hellobl1 was created):\n"  
echo "strblm helob1cx.prf\n"
```

Figure 60. Command File "redo"

4.4.3 Routing Messages

The startup profile for BLM, helob1cx.prf, contains only the SERVER section that specifies the name of the class binary file (.cb) the BLM has to use.

```
[SERVER]
ClassNames=helob1cx
```

According to these definition files the following figure shows how MQ3T routes the messages (the file transfer in this case) to the destination class hellod11.

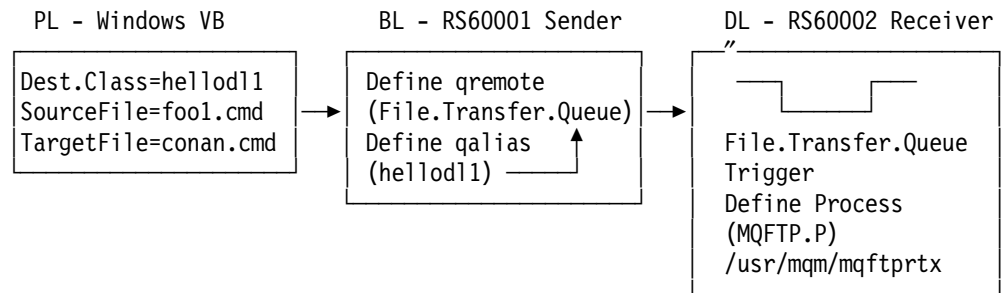


Figure 61. Routing Messages to "hellod11"

Notes:

1. On the windows client, you are asked to enter a destination class name for the file transfer, hellod11.
2. The receiver program gets messages from a fixed queue, named File.Transfer.Queue. This queue is triggered and the trigger initiates the program.
3. In the sender machine, you have to define a remote queue pointing to to the File.Transfer.Queue. and an alias queue name to match the name of the DL destination class. However, you may define only a remote queue called hellod11 instead. You can use the alias to keep MQI definitions independent from the MQ3T application.

If you don't want to define remote and alias queues in the sender machine, the other way to tell MQ3T how to route messages to destination classes is by using the startup profile. You can add a CLASS section to the "helob1cx.prf" file as shown below:

```
[SERVER]

ClassNames=helob1cx

[CLASS]

ClassName = hellod11
QName     = File.Transfer.Queue@RS60002.MQM
```

Figure 62. Profile with Server and Class Sections

With this entry you tell the BLM to route the messages for the destination class hellod11 on the queue File.Transfer.Queue owned by the Queue Manager RS60002.MQM.

4.4.4 Writing the Business Logic

The sender MQI program *mqftp.c* becomes in the MQ3T environment the business logic. This new sender program *hellobl1.c* is quite different.

First of all, this program is a skeleton file generated by the Class Compiler. It contains INCLUDE statements to include the real source code at compile time (*hello1x.c* and *hello1d.c*). Instead of a "main" section, this program begins with a MQENTRY call. In this call MQ3T passes some parameters to the BL program, such as *HInst*. This is the handle representing the instance that sent the message.

```
#include <bmqc.h>
#include "hello1st.h"

void MQENTRY Method1( MQHINST HInst,
                     MQLONG RuleId,
                     PMQLONG pState,
                     MQLONG fBLRequest,
                     HELLO *pBLRequest
                     )
{
    #include "hello1x.c"
}

void MQENTRY Method2( MQHINST HInst,
                     MQLONG RuleId,
                     PMQLONG pState,
                     MQLONG fDLREPLY,
                     HELLOR *pDLREPLY
                     )
{
    #include "hello1d.c"
}
```

Figure 63. Skeleton File "hellobl1.c"

The MQI program gets its input parameters (source file, target file, and queue manager name) from a command entered on the AIX machine. The program finds them in the argv and argc variables. The MQ3T program receives its input parameters from the PL in form of a request message. The pointer to that message is one of the parameters of the MQENTRY section. The structure of this message is defined in the *hellost1.h* header file as follows:

```
typedef struct _HELLO /* hello */
{
    MQCHAR mqmname[20];
    MQCHAR file_source[20];
    MQCHAR file_target[20];
} HELLO;
```

Figure 64 on page 101 and Figure 65 on page 102 shows the business logic. Compare it to the MQI example described in Chapter 12 of *Messaging & Queuing Using the MQI* by Burnie Blakeley, Harry Harris and Rhys Lewis.

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "hello1x.h"          /* sample header file */
/*****
/* (B) include definitions for MQSeries
*****/
#include <cmqc.h>

/*****
/* (C) Define constants used in this program
*****/
#define MAX_FILE_SIZE 150000

HELLOR hellor;              /* buffer for MQRPLY */
MQLONG CompCode;           /* completion code */
MQLONG Reason;             /* reason code */
HELLO My_Rec;              /* message structure */

/*****
/* (G) Declare other variables used in the program
*****/
FILE *fp;
int nbytes = 0;
char dest_class [48];
char from_file [128];
char into_file [128];
char *pc;

struct
{
    char to_file [128];
    unsigned int Data_Length;
    MQLONG Buffer [MAX_FILE_SIZE];
} F_Transfer_Msg ;

pc = strchr (pBLRequest->mqmname, ' '); *pc='\0'; 1
pc = strchr (pBLRequest->file_source, ' '); *pc='\0';
pc = strchr (pBLRequest->file_target, ' '); *pc='\0';

strcpy (dest_class, pBLRequest->mqmname);
strcpy (from_file, pBLRequest->file_source);
strcpy (into_file, pBLRequest->file_target);

printf("PL message arguments:\n"); 2
printf(" dest_class ---%s---\n", dest_class);
printf(" from_file ---%s---\n", from_file);
printf(" into_file ---%s---\n", into_file);

```

Figure 64. MQ3T File Transfer: Sender Program (BL) "hello1x.c" (Part 1)

1 The six C instructions copy the the three input parameters from the message into three work fields.

2 Their contents is then displayed on the BLM's screen.

```

/*****
/* (I) Open the file to be transferred */
/*****
fp = fopen(from_file,"r") ;
if (fp == NULL)
{
    printf("Could not open from_file %s\n",from_file) ;
    strcpy( hellor.message, "ERROR: Source File Could not be Opened ?");
    MQRPLY( HInst, BLREPLY, 0, &hellor.message, &CompCode, &Reason );
    *pState = MQSTATE_CLEAR;
    return ;
}

/*****
/* Start Building the Message content */
/*****
strcpy (F_Transfer_Msg.to_file, into_file);

/*****
/* (N) Read the data from the file into the message buffer */
/*****
nbytes = fread(F_Transfer_Msg.Buffer, 1, MAX_FILE_SIZE, fp);
if (nbytes == MAX_FILE_SIZE)
    printf("WARNING: Copy of file may have been truncated\n");

if (nbytes != 0)
{
    F_Transfer_Msg.Data_Length = nbytes ;

/*****
/* to send the message to the DL Class we use */
/* MQXSEND instead of MQSEND. */
/*****
    MQXSEND (HInst,dest_class, 0, MQMT_BMQ_REQUEST, MQOC_USER + 2,
             MQOV_DEFAULT, MQMA_FIXED_FORMAT, 0,
             (sizeof(F_Transfer_Msg) - MAX_FILE_SIZE + nbytes),
             &F_Transfer_Msg, NULL,&CompCode, &Reason );

    printf(" comp code = %d\n reason = %d\n",CompCode,Reason);
}

if (CompCode > 0 && Reason == 5069)
    strcpy( hellor.message, "ERROR: Invalid Destination/Target Class !" );
else
    strcpy( hellor.message, "SUCCESS: PL's Parameters Processed !");

MQRPLY( HInst, BLREPLY, 0, &hellor.message, &CompCode, &Reason );

/*****
/* (P) Close the file and clear the instance state */
/*****
fclose(fp) ;
*pState = MQSTATE_CLEAR;

```

Figure 65. MQ3T File Transfer: Sender Program (BL) "hello1x.c" (Part 2)

3 + 4 Both version of the sender program, MQI and MQ3T, have sections to open and read the file to be sent.

5 The difference between the MQI and MQ3T programs lies in the way the transfer is coded:

- The MQI program issues five MQI calls:
 - MQCONN to connect to the queue manager
 - MQOPEN to open the destination queue
 - MQPUT to send the file
 - MQCLOSE to close the queue
 - MQDISC to disconnect from the queue manager
- On the other side, the MQ3T program issues only one MQ3T call:
 - MQXSEND to send the file

```
MQXSEND ( HInst,                /* sending instance */
          dest_class,           /* destination class */
          0,                    /* destination instance */
          MQMT_BMQ_REQUEST,     /* message type */
          MQOC_USER + 2,       /* operation code */
          MQOV_DEFAULT,        /* operation version */
          MQMA_FIXED_FORMAT,   /* attribute */
          0,                    /* role */
          (sizeof(F_Transfer_Msg) - MAX_FILE_SIZE + nbytes), /*length*/
          &F_Transfer_Msg,     /* data buffer */
          NULL,                 /* conversion DLL */
          &CompCode, &Reason ); /* return codes */
```

In the MQI program, error and informational messages are displayed on the AIX screen using the “printf” function. For the MQ3T program, this makes no sense since the program has been started from a Windows workstation and the AIX machine is simply the server that holds the files to be transferred.

Therefore, output messages must be displayed in the GUI on the Windows machine. So error and informational messages are sent to the PL program using the MQ3T call MQRPLY. Also, since the PL program sends a REQUEST message to the BL, the BL *must* respond with a REPLY message. The GUI (PL) waits for a this reply message. No more requests can be sent from this interface until the reply has arrived.

6 This code sends the reply to the PL. The message contents depend on the return code from the MQXSEND call.

When a BL program sends a request message, MQ3T intercepts this message, keeps the MQ3T header, turns the MQ3T keyword MQMT_BMQ_REQUEST into the MQI keyword MQMT_REQUEST, and puts the message in the MQSeries queue used by the DL program.

7 The state MQSTATE_CLEAR allows the BL to accept more messages.

Receiver program

The receiver program is the same for both, the MQI and MQ3T example. It is not modified for a MQ3T environment since the receiver AIX machine does not have MQ3T installed. MQ3T considers this program a data logic (DL).

4.4.5 Writing the Presentation Logic

The Presentation Logic (and the Presentation Logic Manager) run in a Windows 3.1 workstation. We have chosen Visual Basic to develop the program. IBM provides the support to "glue" a Visual Basic program to MQ3T. This product is discussed, in detail, in Chapter 3, "Using Visual Basic" on page 43.

Since the MQ3T file transfer program is based on the "hello1" example, most of the code is shown in Chapter 3. The form (GUI) is shown in Figure 54 on page 93. The Visual Basic procedures for the form are:

<i>Close_Click</i>	Ends the program, see Figure 33 on page 55
<i>DisplayCompCode</i>	displays return codes, see Figure 30 on page 54
<i>Exit_cmd_Click</i>	Invoked when Exit button is clicked, calls <i>Close_Click</i>
<i>Form_Load</i>	Invoked when the form is loaded, see Figure 28 on page 53, note the modifications in Figure 68 on page 105.
<i>Form_Unload</i>	See Figure 35 on page 56
<i>FromFile_txt_Change</i>	Invoked when a source file name is typed
<i>MQMName_txt_Change</i>	Invoked when a target class name is typed
<i>OAK1_NewEvent</i>	Invoked when an event messages from the PLM arrives, see Figure 31 on page 54
<i>ProcessPLEvent</i>	Processes the reply message sent by the BL (sender program)
<i>SendMe_cmd_Click</i>	Invoked when the Send to BL button is clicked, sends the request message containing the three input parameters
<i>ToFile_txt_Change</i>	Invoked when a target file name is typed

```
Option Explicit
Dim fromfile As String * 20
Dim tofile As String * 20
Dim mqmname As String * 20
Dim REASON As Long
Dim COMPCODE As Long
Dim msg
Dim NL
```

Figure 66. MQ3T File Transfer: Declarations. This describes the fields for the three input parameters, a message area, and a field that will contain the "new line" characters.

```
Sub Exit_cmd_Click ()
    Close_Click 'common exit routine from hello1
End Sub
```

Figure 67. MQ3T File Transfer: Exit. This routine is called when the Exit button is clicked.

```

Sub Form_Load ()

    fromfile = ""           ' clear work fields
    tofile = ""
    mqmname = ""

    FromFile_txt.Text = "" ' clear input fields
    ToFile_txt.Text = ""
    MQMName_txt.Text = ""

    NL = Chr(13) + Chr(10) ' NEW LINE control character

    vPLClass = "hellogu1"   ' class name
                           ' register class with 3T

    MQREG ByVal vPLClass, 1, ByVal OAK1.hWnd, ByVal BMQ_NOTIFY,
          ByVal MQRGO_REMOVE_LIST_ENTRIES, COMPCODE, REASON

    DisplayCompCode "MQREG" ' display return codes

End Sub

```

Figure 68. MQ3T File Transfer: Display the Window. This routine is called when the form is loaded. It initializes the fields defined in Figure 66 before it "glues" the GUI to MQ3T.

Note: The MQREG statement must be written on one line!

```

Sub ProcessPLevent (ByVal HInst As Long)
    Dim MQevent As MQevent ' event structure
    Dim MsgParams As MQMP  ' message parameters
    Dim BufferLen As Long  ' buffer length
    Dim ReplyMsg As HELLO  ' buffer - NB Don't define as String

    ' query information about the current event
    MQQRYE ByVal HInst, MQevent, COMPCODE, REASON
    DisplayCompCode "MQQYRE"

    ' if the rule is RI_BLREPLY, retrieve the message data and display it
    If MQevent.RuleId = RI_BLREPLY Then
        BufferLen = MQevent.MaxBufferLength
        MQQRYM ByVal HInst, ByVal 1, MsgParams, BufferLen, ReplyMsg, COMPCODE, REASON
        DisplayCompCode "MQQYRM"

        ' if the retrieve works, display the message from the BL Manager
        If COMPCODE = MQCC_OK Then
            MsgBox ReplyMsg.message, 64, "Message from BL"
        End If
    End If

    ' end the current event -
    ' this enables the PL Manager to post new events
    MQENDE ByVal HInst, ByVal MQSTATE_USER, COMPCODE, REASON
    DisplayCompCode "MQENDE"
End Sub

```

Figure 69. MQ3T File Transfer: Process an Event Message. This procedure is called from OAK1_NewEvent when an event message from the PLM has arrived and a rule has been satisfied.

```

Sub FromFile_txt_Change ()           ' source file name
    fromfile$ = FromFile_txt.Text
End Sub

Sub MQMName_txt_Change (0)          ' destination class name
    mqmname$ = MQMName_text.Text
End Sub

Sub ToFile_txt_Change ()            ' target file name
    tofile$ = ToFile_txt.Text
End Sub

```

Figure 70. MQ3T File Transfer: User Input. One of these routines is invoked when the user types or changes one of the three input parameters.

```

Sub SendMe_cmd_Click ()
    If Trim(fromfile$) = "" Then
        msg = "Source file must be entered!"
        MsgBox msg, , "ERROR"
        FromFile_txt.SetFocus
        Exit Sub
    End If

    If Trim(tofile$) = "" Then
        msg = "Target file must be entered!"
        MsgBox msg, , "ERROR"
        ToFile_txt.SetFocus
        Exit Sub
    End If

    If Trim(mqmname$) = "" Then
        msg = "Target Class name must be entered!" & NL & NL & "( e.g. hellod1 )"
        MsgBox msg, 0, "ERROR"
        MQMName_txt.SetFocus
        Exit Sub
    End If

    Dim buffermsg As HELLO ' buffer - NB Don't define as String

    buffermsg.msg_mqmname = mqmname$
    buffermsg.msg_from_file = fromfile$
    buffermsg.msg_to_file = tofile$

    ' send request BLREQUEST to class BLCLASS

    MQSEND ByVal vHInst, ByVal BLCLASS, ByVal BLINSTANCE,
            ByVal BLREQUEST, 0, buffermsg, COMPCODE, REASON

    DisplayCompCode "MQSEND"
End Sub

```

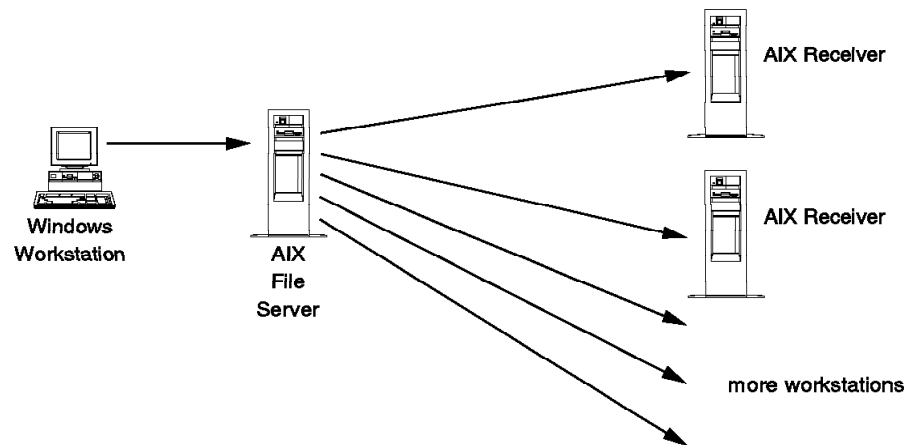
Figure 71. MQ3T File Transfer: Send Input Parameters to BL. This routine is invoked when the Send to BL button is clicked. If all parameters are entered the message is built and sent to the BL.

4.4.6 A Software Distribution Application

You could extend the application developed so far and use the BL as a skeleton for the development of a software distribution application:

- From the Windows workstation, the user initiates the file transfer from the file server (sender program) to several different target systems.
- After receiving a file, the target machines (receiver program) sends a confirmation message back to the sender.
- The confirmation message can then be written to a log file to keep track of the software level installed in each machine.

The log could be kept in either the sender AIX machine or in the Windows workstation.



4864486408

Figure 72. Software Distribution Application

We will not develop this program here, rather we discuss ways to route messages to several different systems. We will explore two possibilities:

- Define classes and queues for all possible target systems in the sender
- Use one target class and modify the queue name in the sender's profile

4.4.6.1 Using Destination Classes

For every new target machine you want to reach, you have to define one DL class to MQ3T. You may "hard code" the class names or define them in a header file, such as *hello1x.h*

```
⋮
#DEFINE DLCLASS2 "he11od12"
⋮
```

Then you have to write the external class descriptions for the new classes. You may use a class header file such as *hellopr1.ch*.

```

:
CLASSDESC
BEGIN
  ClassName  DLCLASS2
  ClassType  DL
  MsgIn      DLREQUEST
  MsgOut     DLREPLY
END
:

```

You also have to add the new classes to the destination parameters in the class definition of the sender's class source file, for example *helob1xc.cs*.

```

:
CLASS
BEGIN
  ClassName  BLCLASS
  Harden     YES
  ClassType  BL
  Destination PLCLASS, DLCLASS, DLCLASS2
  PingTimeout 10
:

```

The class source file must be compiled with `bmqcc`.

Next you have to define the remote queue names. You have two options:

- Define a remote queue name and a queue alias
- Insert two CLASS sections in the startup profile for the BLM, *helob1cx.prf*.

For the first option, use the following `runmqsc` commands:

```

define qremote(MQFTP2) LIKE(SYSTEM.DEFAULT.REMOTE.QUEUE) +
  replace descr('remote queue pointing to File.Transfer.Queue') +
  rname('File.Transfer.Queue') rqnname(RS60003.MQM) xmitq(RS60003.MQM)

define qalias('hellod12') targq(MQFTP2) replace

```

Note: By default, the queue name is the same as the class name. When the sender program `hellob1` sends a file to `hellod12`, the BLM inserts the file (message) into the remote queue `MQFTP2`.

The second way is to add to the sender program's profile class sections for all destinations, as shown in Figure 73 on page 109.

```

[SERVER]
ClassNames=helob1cx

[CLASS]

ClassName = hellod11
QName     = File.Transfer.Queue@RS60002.MQM

[CLASS]

ClassName = hellod12
QName     = File.Transfer.Queue@RS60003.MQM

```

Figure 73. Profile for Server Supporting Multiple Classes

4.4.6.2 Using the Profile

If you choose to add the destinations to your profile you do not have to change the sender's class source file. In the profile (*helob1cx.prf*), define:

- A CLASS entry for the destination class hellod11
- The fully qualified queue name in the target machine (QName@QMGRName).

In this case you need as many .prf files as there are target machines. You start the BLM with the appropriate profile, depending on the machine you want to reach.

With this method, you always enter the destination class hellod11 in the GUI, however, you have to start and stop the BLM each time you want to change the target machine, since you can activate only one Business Logic Manager.

```

[SERVER]
ClassNames=helob1cx

[CLASS]

ClassName = hellod11
QName     = File.Transfer.Queue@RS60002.MQM

```

Figure 74. Server Profile with Class and Queue Definition

On the receiver machines you have to replicate the set-up steps described on pages 83 and 89.

Chapter 5. The Bacon Lettuce and Tomato Sandwich

For a bacon-lettuce-and-tomato sandwich (BLT), you need two pieces of bread, two pieces of bacon, some lettuce, mayonaise, some tomato slices, and someone who manufactures it for you. Of course, you need some kitchen utensils, a toaster, and a microwave to cook the bacon.

You are surprised to read the recipe for a BLT in a book that deals with the second generation of client/server processing. This is because the construction of a BLT is easy to understand and yet complex enough to demonstrate the process of an objected application design that is required for 3T. Also, it may be considered an example of a home version of order entry and inventory control.

Let us make a first sketch of the scenario. You go home for lunch and you are hungry. Your wife is home and you ask her to prepare a BLT for you. From a system design point of view there are two "objects", you and your wife, and messages that flow between you.

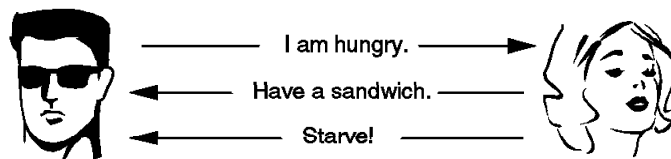


Figure 75. BLT: Message Flow between You and Your Wife

Figure 75 shows what messages may be exchanged between you and your wife:

- You send a request message to your wife to request the sandwich. However, you may try an inform message instead and inform her that you are hungry.
MQ3T knows three message types: the inform message that the recipient does not have to respond to, and the request message that must be answered with a reply message.
- The message coming back to you either carries the BLT or tells you to make the sandwich yourself or starve.

Usually, that's all what happens. However, there are two other possibilities to consider. Your wife may be mad at you and never respond (during your lunch break), or she got tied up with the telephone and prepares the sandwich too late for you to eat it.

When you design an application you have to anticipate the unusual. That's what programming is all about. You have to deal with *messages that are late or never arrive*. The MQ3T infrastructure helps you to do this without much coding.

You could set an alarm clock, and when ten minutes have passed you could go somewhere else for lunch or call Luigi and order a pizza. Luigi advertises he will always deliver. But is this true? What do you do if Luigi does not deliver? You go back to work hungry.

What you read so far is only the tip of the iceberg. The system we design has many more objects. What, for example, happens when you run out of tomatoes

or when the toaster breaks down? And what if your son, whom your wife sent for the bacon, forgets and plays with the dog instead? Also, what do you do when the electricity is shut off? You see, we deal with a rather complex system. Let us analyze this system, make an application design, and write it down in the form of a couple of MQ3T programs.

MQ3T Application Development Process

The following sections describe how to design and develop the application based on the scenario described in section 1.3.2, "The 3T Application Development Process" on page 12.

5.1 Requirements

Let us describe the demonstration program about the household of Konrad and his wife Karen as far as lunch is concerned.

The process is event-driven, initiated by Konrad. Konrad's lunch break is limited. Therefore, his lunch must be delivered within a specific time. If, for whatever reason, Karen does not prepare the sandwich, there must be enough time left to order a pizza from Luigi next door. Should Luigi not deliver then Konrad has to go back to work hungry.

Karen has a different set of requirements. When she does not have the food items she needs to prepare the BLT she has to notify Konrad immediately, so that he can order a pizza.

Missing food items must be written on a shopping list. This process should be automated; also consider that any item can either be used up or spoiled.

Once a day the shopping list is sent to the grocer. When the grocer delivers, the food is stocked and the inventory list is updated. Should the manufacturing process being held up because of a missing item, it resumes again.

Since Karen is interested in everything going on in the house she wants to be able to inquire about the status of the food supply and the condition of the kitchen equipment at any time.

Events, out of Karen's and Konrad's control, may spoil the food supply, cause some equipment to seize operation, or disable communications between "objects" involved. The objects may be Karen and Konrad themselves, or the refrigerator and the other appliances.

After this rough outline let us summarize the application. As said before, the BLT application is comprised of the manufacturing of a specific sandwich, order entry and inventory control. As backup for a malfunction in the manufacturing process an outside vendor is called (to deliver a pizza). The purpose of the application is threefold:

1. Lunch has to be prepared within a specified time (production).
2. The food supply in the household shall be monitored and replenished when depleted or spoiled (inventory control).
3. A process for detecting and repairing malfunctioning appliances has to be developed (maintenance).

So far we defined three users that can influence or control the business process. We have to build user interfaces for them so that they can interact with the system. The users and their interfaces are:

- *Konrad* starts the application when he enters the house and ends it when he goes back to work. His only purpose is to eat lunch. He initiates the process by asking Karen for a BLT. If he does not receive it within a specified time he relies on a vendor to deliver a pizza.
- *Karen* controls the fully automated manufacturing plant. When a piece of equipment breaks it is put on the repair list. When a food item is depleted it is put on the shopping list. She monitors two interfaces:
 - The electronic *shopping list* contains items that have to be bought in order to produce the BLT. Karen fills in the quantity to order and sends it to the grocer.
 - The *repair list* is used to monitor equipment and to initiate any repair.
- *Luigi* takes orders and delivers.

Note: The *grocer* is an entity outside our environment and, therefore, not a “user” that has a user interface. However, the grocer is an *object* in our business design.

To turn this application into a “computer game” we add one more user:

- A *Gremlin* creates events that causes the business to deal with abnormal situations, such as broken toasters and rotten tomatoes.

The job: For this demonstration, only Konrad initiates the application. He is the 3T job owner. Konrad owns all GUIs (presentation logics) in our scenario, and all business logics as well. To start the demonstration program we type:

```
startjob KONRAD any_instance_name
```

To end the demonstration program we close Konrad’s GUI. Closing Konrad’s GUI causes all other PLs and BLs that are associated with the job to end, too.

Classes: In this demonstration, we have PL and BL classes, each class representing one of the objects. The class KONRAD is the job owner.

Note: Class names are case sensitive.

Instances: 3T allows us to create many instances of a class. In this demonstration, however, we will use only one instance of each class. For example, there will be only one instance of the class KONRAD. When (the instance of) KONRAD ends then 3T ends all instances associated with the job. 3T sends a system message to the instances so that some clean up work can be done.

Note: We could call Konrad’s class “CUSTOMER” and use Konrad as an instance name. Other instances of that class could be Dagwood and Blondie.

5.2 Business Analysis

We will now analyze the requirements and produce specifications for the application. The specifications will then be used to make a high level design. Keep in mind that we are going to design a “computer game” that contains the “real” application and modules that simulate external events. In this section we define:

- The objects
- The functions performed by each object
- The message flow between objects
- The GUI interfaces

5.2.1 Objects and Their Functions

Table 15 defines the objects and their functions.













<i>Table 15 (Page 1 of 2). Objects and Their Functions</i>		
Symbol	Object	Function
	Konrad	<p>Konrad is the job owner and starts the process.</p> <ul style="list-style-type: none"> • He causes the GUIs for the other PLs to be displayed. • He can order BLTs and pizzas.
	Luigi	<ul style="list-style-type: none"> • He receives pizza orders from Konrad and delivers, providing he is not busy or otherwise distracted. • He responds to inquiries issued from the repair list. • The gremlin can keep him from working.
	Karen	<p>Karen controls the manufacturing process.</p> <ul style="list-style-type: none"> • She produces the BLT providing food is available and the kitchen equipment is in working order. • She informs Konrad when she cannot deliver. • She does not accept any BLT requests when she is on the phone. • She responds to inquiries issued from the repair list.
	Refrigerator	<ul style="list-style-type: none"> • It supplies and stocks bacon, lettuce and mayonaise and keeps inventory. • When an item is used up an order is placed in the shopping list. From that time on no requests are accepted until a new food shipment arrives. • The gremlin can disable the unit or spoil the food items. • It responds to inquiries issued from the shopping and repair lists.
	Vegetable basket	<ul style="list-style-type: none"> • It supplies and stocks tomatoes and keeps inventory. • It places an entry on the shopping list when it is out of tomatoes. It only honors requests when tomatoes are available. • The gremlin can hide the basket or spoil the tomatoes. • It responds to inquiries issued from the shopping and repair lists.

Table 15 (Page 2 of 2). Objects and Their Functions

Symbol	Object	Function
	Breadbox	<ul style="list-style-type: none"> • It supplies and stocks bread and keeps inventory. • It places an entry on the shopping list when it is out of bread. It only honors requests when bread is available. • The gremlin can spoil the bread or break the breadbox. • It responds to inquiries issued from the shopping and repair lists.
	Toaster	<ul style="list-style-type: none"> • It receives bread and returns toast to the requestor. • It can be disabled by the gremlin. • It responds to inquiries issued from the repair list.
	Microwave	<ul style="list-style-type: none"> • It receives bacon, cooks it and returns it to the requestor. • It can be disabled by the gremlin. • It responds to inquiries issued from the repair list.
	Shopping list	<p>The list contains the names of all food items.</p> <ul style="list-style-type: none"> • It indicates what items have to be ordered. • The user must enter a quantity before an order is sent to the grocer. • An inquiry function allows to monitor quantity of the food items in stock. • The gremlin can erase the shopping list.
	Repair list	<p>This GUI can perform two functions:</p> <ul style="list-style-type: none"> • Inquiries allow it to monitor if any kitchen equipment is working or not working. • A repair function initiates the repair of equipment out of order. <p>Some special functions allow it to reverse situations inflicted by the gremlin.</p>
	Grocer	<ul style="list-style-type: none"> • He receives the shopping list and fills the order, providing the store is open. • He delivers the items directly to the refrigerator, bread and vegetable baskets. • He responds to inquiries issued from the repair list. • The gremlin can close the store.
	Gremlin	<p>The gremlin causes abnormal situations, such as:</p> <ul style="list-style-type: none"> • Spoil the food supply • Disable kitchen equipment • Close the grocery store • Send Luigi on a break • Keep Karen on the phone forever • Erase the shopping list

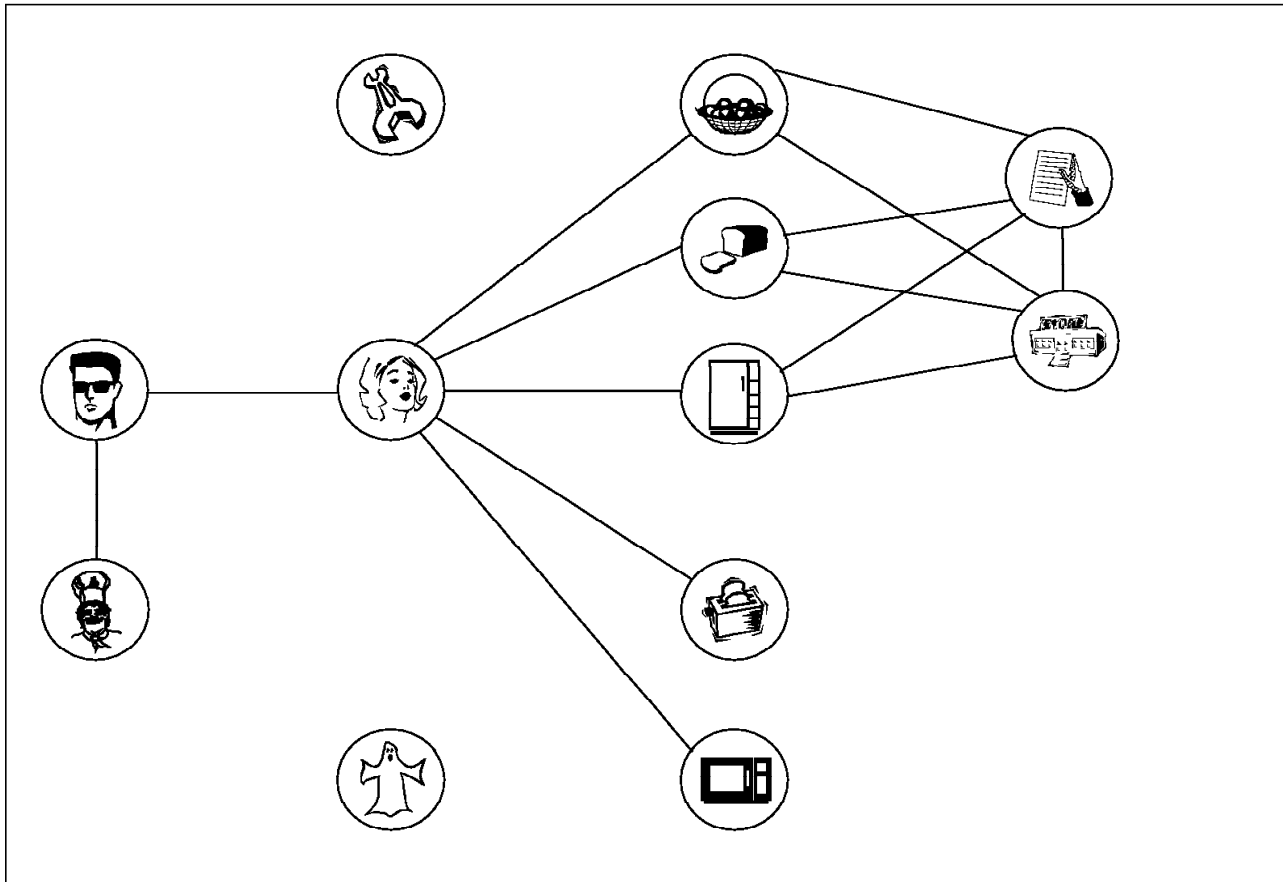


Figure 76. BLT: Objects and Message Flow, Production

5.2.2 Message Flow between Objects

Communications between the objects is shown in two figures:

- Figure 76 shows the message flow for:
 - The production of the BLT
 - The ordering process of food items
 - The delivery process of food items

You see that Karen is the central figure in this process. She is, however, not involved in ordering and delivering food items. The three storage facilities (breadbox, vegetable basket, refrigerator) deal directly with the shopping list for ordering and the grocer for deliveries.

- Figure 77 on page 117 shows the message flow for:
 - The exceptions that can occur in the process, inflicted by the gremlin. Equipment can break, food can be spoiled, the grocery may close, Luigi may be called away, or Karen may receive a phone call that keeps her busy.
 - The process that “repairs” equipment, opens grocery again, calls Luigi back from lunch, and ends Karen’s phone calls.

The solid lines represent the influence of the gremlin and the dotted lines show the connections to the repair facility.

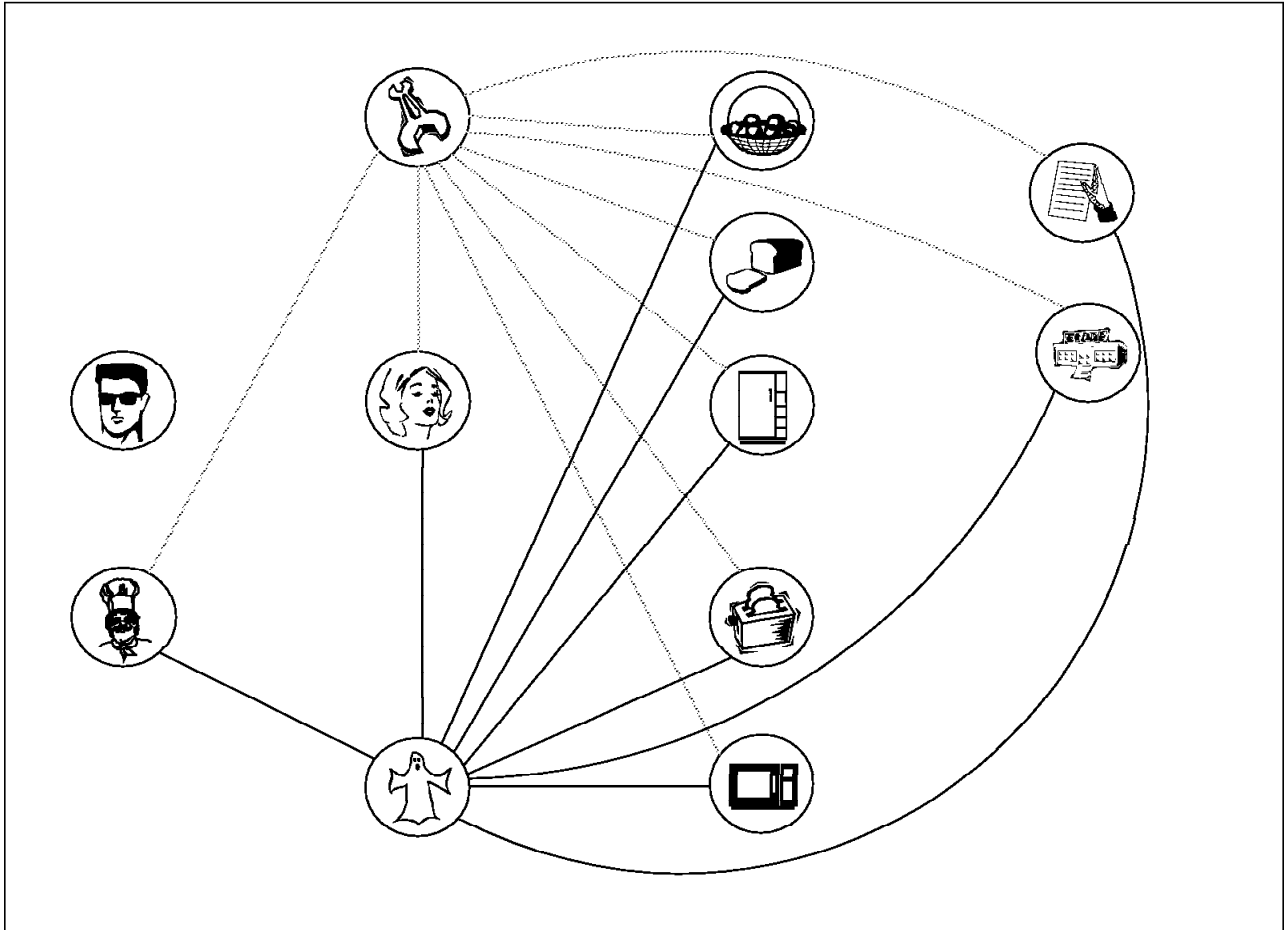


Figure 77. BLT: Objects and Message Flow, Maintenance

In Figure 76 on page 116 you can see that Karen sends a message to the breadbox to request bread. After she received the bread she sends it to the toaster to be toasted.

One could imagine the scenario in Figure 78. The bread request could be sent to the breadbox. The breadbox could then send the bread to the toaster which then sends toast to Karen.

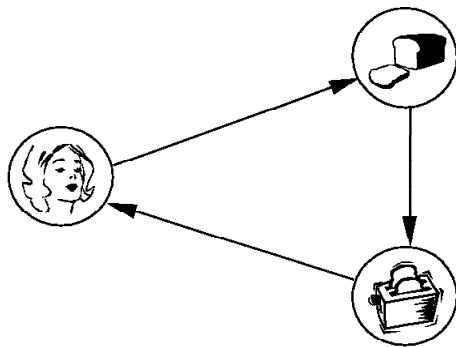


Figure 78. BLT: Circular Message Flow

However, 3T is not designed to work that way if you use request and reply messages. You could send inform messages around in a circle. Inform messages, however, cannot be timed. You would never find out if the breadbox or the toaster is broken.

Note: The lines between the objects indicate the message flow only, they do not represent a specific message. Several messages of different types may be exchanged between objects.

5.2.3 GUI Prototypes

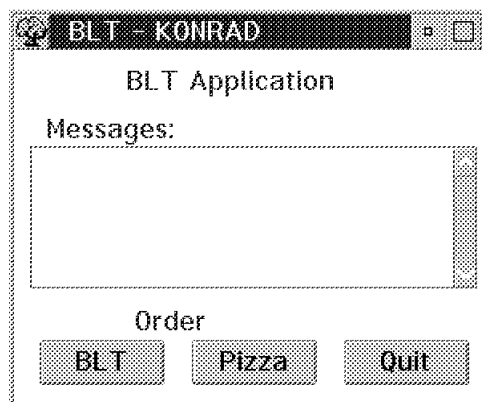
Now we have to decide what graphical user interfaces we need. We use GUIs to externally manipulate the execution of the application. The five objects that require a GUI are:

- Konrad
- Luigi
- Shopping list
- Repair list
- Gremlin

For Karen we do not need a GUI since she represents the automated production process.

Figure 79 through Figure 83 on page 120 show prototypes for the user interfaces. The GUIs are created with the OS/2 Presentation Manager. Their final appearance is determined by the GUI programmer. However, in the design phase we should have a good understanding what function the GUIs initiate and what kind of information they display:

- All GUIs contain a scrollable area to display messages.
- All actions are initiated with push buttons.
- If an action can be directed to more than one object then radio buttons are used to identify that object.



Messages

- BLT is served.
- Pizza is served.
- Too late for BLT (timed out).
- Too late for pizza (timed out).
- BLT arrives too late.
- Pizza arrives too late.

Actions

- Order a BLT
- Order a pizza
- Quit

Figure 79. GUI Prototype for Konrad

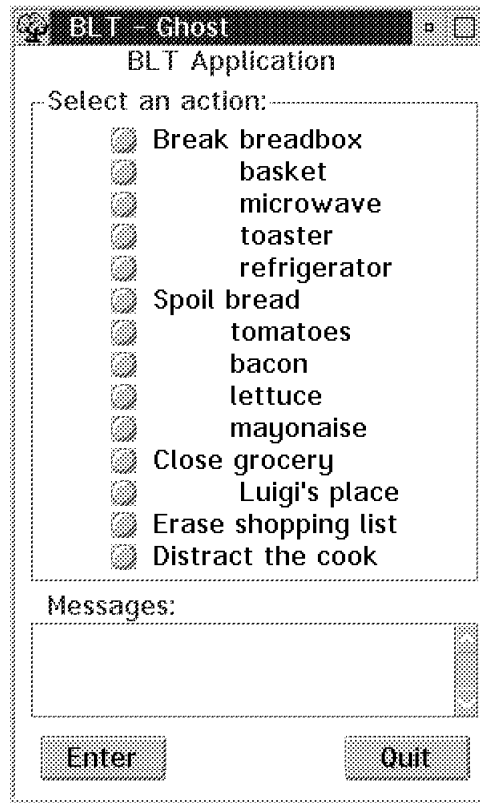


Figure 80. GUI Prototype for Gremlin

Messages

The message area could be used to keep an action log.

Actions

- Send a message to the object associated with the selected radio button.
- Quit.

Only one radio button can be selected at one time.

- "Spoil" sets the inventory to 0.
- "Break" disables the BL, it does not end the BLM.
- "Erase shopping list" sets all values to 0.
- The other functions disable the PL/BL but do not end the PLM/BLM.

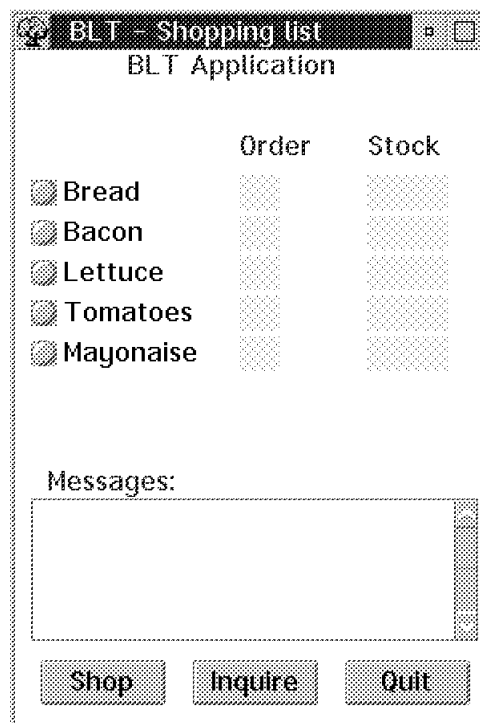


Figure 81. GUI Prototype for Shopping List

Messages

The message area could be used to keep an action log:

- Order sent to grocery
- Inquire food item
- No response to inquiry

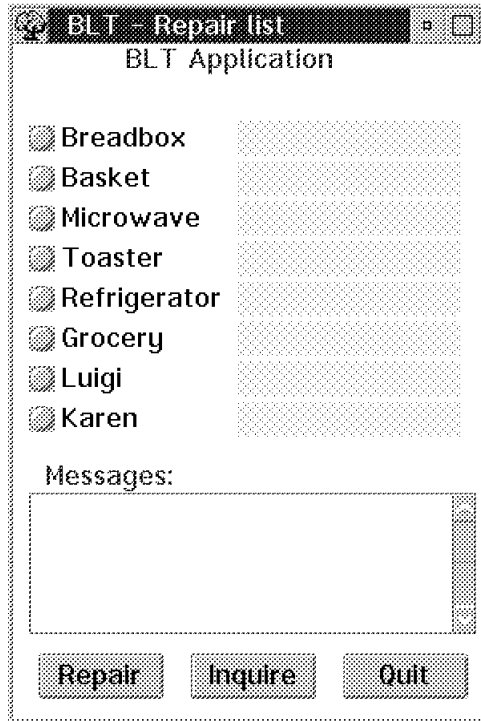
Input

A quantity must be typed before an order can be send to the grocery.

Actions

- "Shop" sends the shopping list to the grocery. The radio buttons have no effect for this function.
- "Inquire" checks the quantity of the item marked by the radio button. When the response to the inquiry comes back the value is displayed in the field "Stock".
- Quit.

Only one radio button can be selected at one time.



Messages

Message area could be used to keep an action log, for example:

- Get working
- Inquiring
- No response to inquiry
- Store is closed

Actions

- "Repair" tries to activate an object.
- "Inquire" checks the status an object.

A radio button must be selected for the action.

Status

The status of the object or the inquiry is displayed as follows:

- OK
- Working
- Not responding
- Inquiring ...
- Wait ...

Figure 82. GUI Prototype for Repair List



Messages

- Deliver a pizza (Konrad's order arrived)
- Luigi is taking a break (initiated by the gremlin)
- Luigi is back from his break (initiated from the repair list)

Actions

- Send the pizza to Konrad
- Quit

Figure 83. GUI Prototype for Luigi

Note: The GUI prototypes have been created using the OS/2 Presentation Manager.

5.3 3T Design

Figure 76 on page 116 and Figure 77 on page 117 illustrate the high level design of the application. We will now map this design into a 3T design.

- We identify and name the classes.
- We define the message flow and name the messages.
- We define the methods that process the messages.
- We specify the rules that invoke the methods.
- We name the methods and the files that contain the source code.

5.3.1 3T Classes

We designed the application having twelve classes, five presentation logics and seven business logics. We assign to them the names specified in Table 16.

Table 16. BLT: 3T Classes

Number	Object	Class name	Type	Hard/Soft
1	Konrad	KONRAD	PL	N/A
2	Luigi	LUIGI	PL	N/A
3	Karen	KAREN	BL	hard
4	Refrigerator	FRIDGE	BL	hard
5	Vegetable basket	BASKET	BL	hard
6	Bread box	BREADBOX	BL	hard
7	Toaster	TOASTER	BL	hard
8	Microwave oven	MICRO	BL	hard
9	Shopping list	SHOPPING	PL	N/A
10	Grocer	GROCER	BL	hard
11	Gremlin	GREMLIN	PL	N/A
12	Repair list	REPAIR	PL	N/A

We define all BLs as “hard” classes, meaning that all BLs can recover from a network failure.

For each class we have to create a 3T class source file. Each class source file contains besides other information “external descriptions” of all classes the class communicates with. Since our application has only twelve classes we can write the external attributes of all classes in one separate file. That file we will include into all class source files.

The external attributes of a class are:

- The *class name* as defined in Table 16
- The *class type* as specified in Table 16
- A list of all *messages* the class can *receive*
- A list of all *messages* the class can *send*
- A keyword that defines whether the class is *hard* or not

Before we can write the external class descriptions we have to analyze the message flow and name all messages.

5.3.2 Messages

Figure 76 on page 116 and Figure 77 on page 117 show the message flow between the twelve classes. Now let us look at each class individually, determine what messages each class can send and receive, and name them.

5.3.2.1 BLT Production Process

Figure 84 shows the message flow and the messages names used for the production of the BLT.

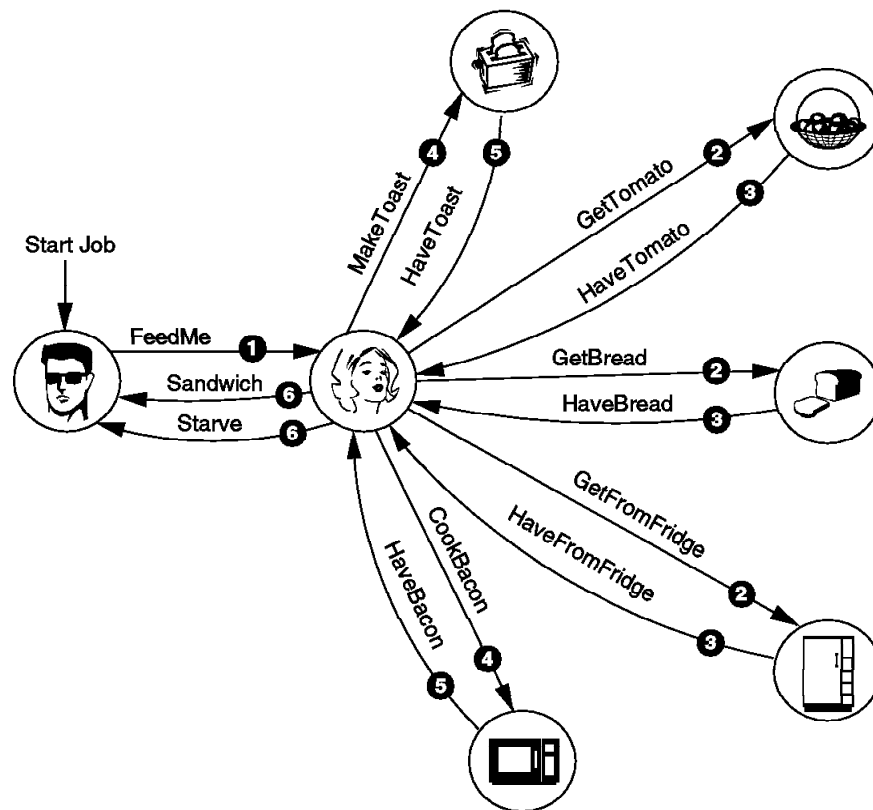


Figure 84. BLT: Messages in Production Process

1. Konrad sends the request message *FeedMe* to Karen.
2. Karen sends a wave of three request messages (*GetTomato*, *GetBread*, *GetFromFridge*) to the vegetable basket, breadbox and refrigerator.
3. The three classes respond with the reply messages *HaveTomato*, *HaveBread* and *HaveFromFridge*.
4. After all three reply messages have arrived in time, Karen sends a second wave of two request messages (*MakeToast* and *CookBacon*).
5. Toaster and microware send the reply messages *HaveToast* and *HaveBacon*.
6. To end the process Karen does one of the following:
 - When the replies from toaster and microwave arrived in time, she sends the reply message *Sandwich* to Konrad.

- If any one of the five objects do not respond in time she sends the reply message *Starve* to Konrad.

Note: If the replies to one of the waves (3 and 5) are incomplete *all* messages are ignored, the ones that arrived on time and the ones that arrive late.

What is a wave?

When a class sends request messages to several classes then it sends a “wave” of messages. The replies to a wave of messages are expected at the same time and usually processed by one method. A class can send only one wave of messages at a time.

5.3.2.2 Inventory Control Process

Figure 85 shows the message flow and the messages names used to control, order and deliver food items.

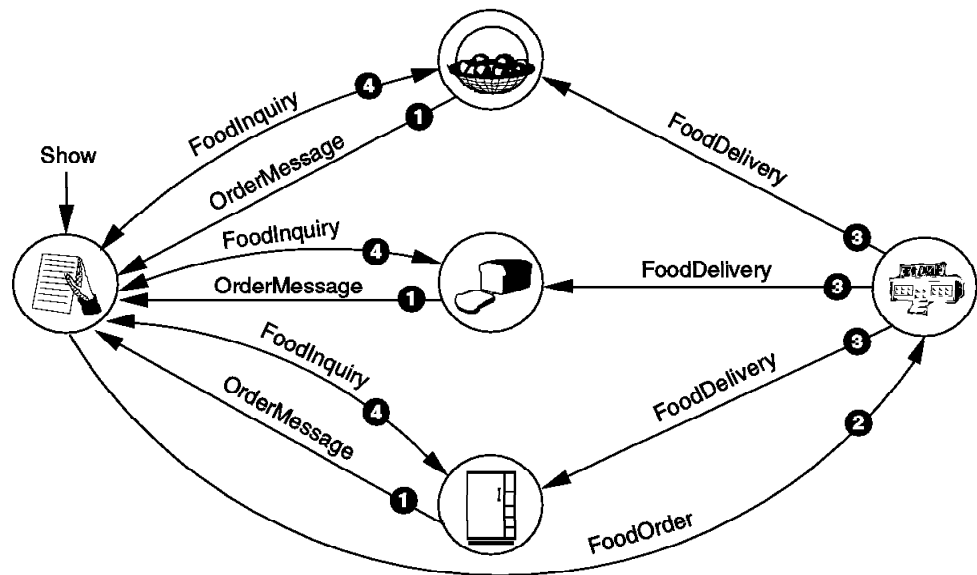


Figure 85. BLT: Messages in Inventory Control Process

1. The process is initiated when the inventory of one of vegetable basket, breadbox or refrigerator is depleted. The *OrderMessage* is sent to the shopping list. This is an *INFORM* message, no response is expected.
2. The user of the shopping list enters the quantity to order and sends a *FoodOrder* message to the grocer. This is an *INFORM* message, too.
3. Depending on the number of items ordered the grocer sends one or more *FoodDelivery* messages to the basket, breadbox or refrigerator.
4. At any time, one can send from the shopping list a *FoodInquiry* message to one of the classes, which respond with a message of the same name. These are *INFORM* messages.

Note: We use *INFORM* messages to show how the 3T timer function can be used to check if responses arrive in time.

5.3.2.3 Food Order Process

Figure 86 shows the messages that Konrad can send and receive. Konrad has two options: he can order a BLT or a pizza.

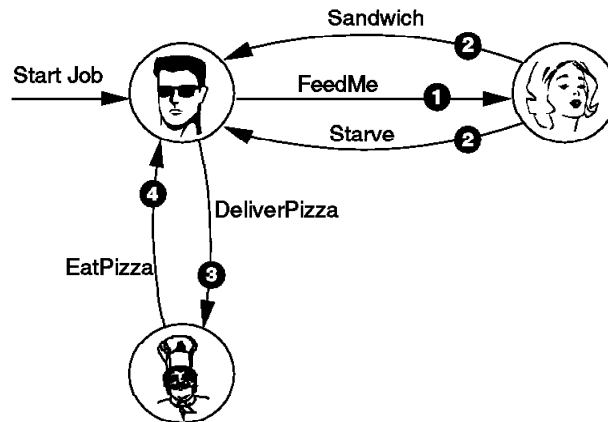


Figure 86. BLT: Messages in Food Order Process

1. Konrad sends a request message to Karen to order a BLT.
2. Karen replies with either a *Sandwich* or *Starve* message.

Konrad knows when he will not receive a BLT, he receives either the *Starve* message or the timer expires and he receives no message at all.

3. Except when he is waiting for a BLT, Konrad can send the *DeliverPizza* request to Luigi.
4. Luigi replies with a *EatPizza* message. To do this the user has to click on a push button.

5.3.2.4 Exceptions and Maintenance

All classes except Konrad are involved in this process.

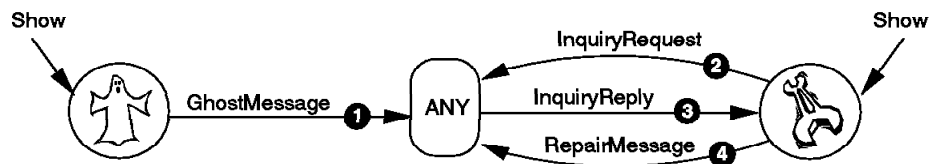


Figure 87. BLT: Messages in Exception/Maintenance Process

1. The gremlin can send a *GhostMessage* to any object. The action that class takes depends on the contents of the message. It either disables the object or sets the inventory to zero.
2. From the repair list one can inquire the status of any object by sending an *InquiryRequest* to it.
3. The object replies with an *InquiryReply*. It will be indicated in the list when the reply does not arrive in time.
4. A *RepairMessage* activates a disables object.

Note: We use REQUEST/REPLY messages to show how 3T controls messages that arrive in time or late.

5.3.2.5 Message Summary

Table 17 summarizes the messages.

<i>Table 17. BLT: Message Summary</i>					
Name	Type	Operation Code	Role	Format	Structure
StartJob	INFORM	OC_STARTJOB		FIXED	STARTJOB
Show	INFORM	OC_SHOW		FIXED	STARTJOB
FeedMe	REQUEST	OC_SANDWICH		FIXED	MSG100
Sandwich	REPLY	OC_SANDWICH		FIXED	MSG100
Starve	REPLY	OC_STARVE		FIXED	MSG100
DeliverPizza	REQUEST	OC_PIZZA		FIXED	MSG100
EatPizza	REPLY	OC_PIZZA		FIXED	MSG100
GetTomato	REQUEST	OC_TOMATO	1	FIXED	MSG100
HaveTomato	REPLY	OC_TOMATO	1	FIXED	MSG100
GetBread	REQUEST	OC_BREAD	2	FIXED	MSG100
HaveBread	REPLY	OC_BREAD	2	FIXED	MSG100
GetFromFridge	REQUEST	OC_FRIDGE	3	FIXED	MSG100
HaveFromFridge	REPLY	OC_FRIDGE	3	FIXED	MSG100
MakeToast	REQUEST	OC_TOAST	5	FIXED	MSG100
HaveToast	REPLY	OC_TOAST	5	FIXED	MSG100
CookBacon	REQUEST	OC_BACON	6	FIXED	MSG100
HaveBacon	REPLY	OC_BACON	6	FIXED	MSG100
GhostMessage	INFORM	OC_GHOST		FIXED	MSG100
RepairMessage	INFORM	OC_REPAIR		FIXED	MSG100
InquiryRequest	REQUEST	OC_INQUIRY		FIXED	MSG100
InquiryReply	REPLY	OC_INQUIRY		FIXED	MSG100
OrderMessage	INFORM	OC_ORDER		FIXED	MSG100
FoodInquiry	INFORM	OC_FOODINQ		FIXED	MSG100
FoodOrder	INFORM	OC_FOOD		VARIABLE	
FoodDelivery	INFORM	OC_FOOD		FIXED	MSG100
MQ_SYSTEM_OWNER_ENDED	INFORM	MQOC_SYSTEM_OWNER_ENDED		FIXED	N/
<p>Note:</p> <ul style="list-style-type: none"> • All fixed messages are 100 bytes long. • The message structures are in "bltstruc.h". • System messages are defined in "bmqsysms.ch". 					

Note: In this example we will also process one 3T system message: MQ_SYSTEM_OWNER_ENDED. This message is sent to all BLs when the job owner's GUI is closed.

System messages are defined in \3TIER2\INCLUDE\bmqsysms.ch. If you use one of them include this statement in your class source files:

```
CSINCLUDE "bmqsysms.ch" /* 3T system message descriptions */
```

After all messages are defined we have to write them in a form 3T can understand. As an example, we define the *GetBread* message as follows:

```
MESSAGE
BEGIN
  MsgName      GetBread      // Message name
  MsgType      REQUEST      // Message type (INFORM, REQUEST, REPLY)
  OperationCode OC_BREAD    // Each message must have one
  Role         2            // Used to collate the replies
  Format        FIXED       // FIXED or VARIABLE
  StrucLen     100         // Message length
  StrucName    MSG100      // Name of the structure
  StrucFile    bltstruc.h  // File that contains the structure
END
```

Figure 88. A Message Description

We define all messages that class can send or receive in the class source file for the class, also the reply messages. We create a header or include a file that contains all messages used in this application. This file, MESSAGES.CH, is in Appendix A, "Class Source Files for BLT Example" on page 213 and also on diskette 2 included with this book.

The operation codes are defined in the header file "bltdef.h" as follows:

```
/*
*****
/*      Operation codes for messages      */
*****
#define OC_STARTJOB      (MQOC_USER)
#define OC_SHOW          (MQOC_USER + 1)
#define OC_SANDWICH      (MQOC_USER + 2)
#define OC_STARVE        (MQOC_USER + 3)
#define OC_PIZZA         (MQOC_USER + 4)
#define OC_TOMATO        (MQOC_USER + 5)
#define OC_BREAD         (MQOC_USER + 6)
#define OC_FRIDGE        (MQOC_USER + 7)
#define OC_TOAST         (MQOC_USER + 8)
#define OC_COOK          (MQOC_USER + 9)
#define OC_GREMLIN       (MQOC_USER + 10)
#define OC_REPAIR        (MQOC_USER + 11)
#define OC_INQUIRY       (MQOC_USER + 12)
#define OC_ORDER         (MQOC_USER + 13)
#define OC_FOODINQ       (MQOC_USER + 14)
#define OC_FOOD          (MQOC_USER + 15)
*****
*/
```

Note: The operation codes for the system messages are defined in the header file *bmqc.h*.

Figure 89 on page 127 shows the contents of the file *bltstruc.h* that contains the two message structures used in this demonstration program.


```

typedef struct _STARTJOB /* startjob */
{
    MQCHAR Buffer[100]
} STARTJOB;

typedef struct _MSG100 /* standard message */
{
    MQCHAR message[20];
    MQLONG number;
    MQLONG value;
    MQCHAR filler[72];
} MSG100;

```

Figure 89. Message Structures

5.3.3 Class Descriptions

For the "external" class descriptions we create a header or include file CLASSES.CH. Each class description contains the information from Table 16 on page 121 and lists of all messages the class can send and receive.

Note: You do not have to include the names of reply messages in either the external class descriptions or the methods that send them. The cross reference check function of the class compiler will produce a warning, however.

For a better understanding, we include all messages in our class source files.

The following shows two class descriptions from the file CLASSES.CH. The complete listing is in Appendix A, "Class Source Files for BLT Example" on page 213 and also on diskette 2 accompanying this book.

```

CLASSDESC // PL: Konrad
BEGIN
  ClassName KONRAD
  ClassType PL
  MsgIn StartJob, Sandwich, Starve, EatPizza
  MsgOut FeedMe, DeliverPizza, Show
END

CLASSDESC // BL: Karen
BEGIN
  ClassName KAREN
  Harden YES
  ClassType BL
  MsgIn FeedMe,
        HaveTomato, HaveBread, HaveFromFridge,
        HaveToast, HaveBacon,
        GhostMessage, RepairMessage, InquiryRequest,
        MQ_SYSTEM_OWNER_ENDED
  MsgOut Sandwich, Starve,
        GetTomato, GetBread, GetFromFridge,
        MakeToast, CookBacon,
        InquiryReply
END

```

Figure 90. Class Descriptions

5.3.4 Rules and Methods

MQ3T requires us to define rules for all messages we want to process. There are three standard rules for REPLY messages and one rule for INFORM and REQUEST messages:

1. One INFORM, REQUEST or REPLY message arrives, or all replies of a wave arrive in time.
2. A single REPLY message or at least one of the replies of a wave does not arrive in time. This is a timed rule that is satisfied when a timer expires.
3. A REPLY message arrives late. Late messages are treated like INFORM messages. If you do not specify a rule for a late reply it is thrown away.

Furthermore, you can define rules that depend on the *state* the instance is in. A state is represented by a number. Initially, 3T sets the state of any instance to the equivalent of MQSTATE_NEW, that is 0. The programmer can change this state to any other value representing situations such as "busy", "waiting for host data", or "accept input". For example, a rule can be satisfied when a message arrives and the instance state is "not busy". If the state is "busy" the message remains in the queue.

Each rule is associated with a method. The method defines what piece of code shall be executed when a rule is satisfied. Several rules may share the same method.

Each method description contains either a PL program name, such as luigi.exe, or a BL library and procedure name, such as micro.Inquiry. For BLs, you also specify the name of the source file that contains the application code, for example deliver.c.

For both PLs and BLs you can define multiple methods. The code representing PL methods and the GUI is in one file that becomes the EXE. For BLs, 3T can create a skeleton for you that contains the entry point for each procedure or method, and an INCLUDE statement for the source code. All the programmer writes is the code to process the message(s) for which a rule is satisfied.

State: When an instance is created its state is MQSTATE_NEW (0). The programmer must set the state to another value before the method ends. To end an instance set its state to MQSTATE_END (-1). For a PL, the GUI will disappear (and reappear if a message is in the queue and a rule can be satisfied).






Note: There is no rule what to define first, rules or methods.

5.3.4.1 Rules and Methods for PLs

Since you can have only one executable for a PL (GUI) all messages the PL can receive are processed by the same windows procedure. You may define as many methods as you wish, however, each of them must specify the same program name.

Table 18 on page 129 contains the methods for the PLs of the BLT application.

For an example, let us look at the rules for the PL class KONRAD. We name eight rules for the four messages the class can receive. Refer to the class description in Figure 90 on page 127 and Figure 86 on page 124.

Table 18. BLT: Methods for Presentation Logics							
 KONRAD	 LUIGI	 GREMLIN	 SHOPPING	 REPAIR	Method Name	Program Name	Messages
X					BLTMethod	konrad.exe	from Karen
X					PizzaMethod	konrad.exe	from Luigi
	X				TheMethod	luigi.exe	all
		X			TheMethod	gremlin.exe	all
			X		TheMethod	shopping.exe	all
				X	TheMethod	repair.exe	all

StartJob This message is sent by the 3T executable STARTJOB.EXE which is invoked by the startjob command.

RuleName: StartJobRule

MethodName: BLTMethod

Sandwich This message is sent by Karen when the BLT is put together. We specify three rules for this message, depending on the time the rule is satisfied:

- In time
- When the timer expires (timeout)
No message arrives!
- After the timer expired (late)

RuleName: SandwichRule1 - in time
SandwichRule2 - timeout
SandwichRule3 - late

MethodName: BLTMethod

Starve This message is sent by Karen when she is unable to produce a BLT, for example, when she is out of food or when when one of the objects in the kitchen is not working.

RuleName: StarveRule

MethodName: BLTMethod

EatPizza This message represents a pizza delivery by Luigi. Just like the BLT the message can arrive in time or late. Another rule is satisfied when the timer expires.

RuleName: PizzaRule1 - in time
PizzaRule2 - timeout
PizzaRule3 - late

MethodName: PizzaMethod

For Konrad we defined rules that depend on the *time* a message arrives. For Luigi we write rules that depend on the *state* the instance (program) is in.

Luigi accepts pizza orders when he is in one of two states, MQSTATE_NEW (0) and MQSTATE_CLEAR (1). Luigi's instance is created when a *DeliverPizza* message arrives. When we process this message we change the state from MQSTATE_NEW (0) to MQSTATE_BUSY (30). Luigi remains in this state until the Deliver push button is pressed which causes the *EatPizza* message to be sent. Then the state is set to MQSTATE_CLEAR (1); MQSTATE_END (-1) would make the GUI disappear. While Luigi bakes the pizza his state is MQSTATE_BUSY. Pizza orders that arrive when the state is not new or clear remain in the queue.

For Luigi we define seven rules but only one method:

DeliverPizza This message is sent by Konrad to order a pizza. We specify two rules for this message, ensuring that Luigi bakes only one pizza at a time. The rules are satisfied when the state is:

- MQSTATE_NEW: Luigi's instance did not exist when the message arrived.
- MQSTATE_CLEAR: Luigi is waiting for work.

RuleName: LuigiRule1 - first order arrives (MQSTATE_NEW)
LuigiRule2 - another order arrives (MQSTATE_CLEAR)

MethodName: TheMethod

GhostMessage This message is sent by the Gremlin to send Luigi on a break. In the method, the state is set to either MQSTATE_DISABLED or MQSTATE_DISABLED_WHILE_BUSY.

RuleName: GremlinRule

MethodName: TheMethod

InquiryRequest This message is sent from the repair list to find out if Luigi is on a break (disabled), busy or waiting for work. The method responds with an *InquiryReply*.

RuleName: InquiryRule

MethodName: TheMethod

RepairMessage This message, sent from the repair list, reverses the state caused by the gremlin. In the method, the state is set to either MQSTATE_CLEAR or MQSTATE_BUSY. We define three rules:










RuleName: RepairRule1 - the state is MQSTATE_DISABLED
RepairRule2 - the state is MQSTATE_DISABLED_WHILE_BUSY
RepairRule3 - in any other state the message will be ignored

MethodName: TheMethod

3T passes the rule ID in one of the message parameters to the program. In the program, we check the rule ID to execute the appropriate code. The rule name is used to set a timer for the rule.

Table 19 on page 131 is the summary of all rules for the PLs.

Table 19. BLT: Rules for Presentation Logics

Class	Message	Rule Name	Rule ID	Remarks
 KONRAD	StartJob	StartJobRule	RI_STARTJOB	
	Sandwich	SandwichRule1	RI_SANDWICH1	in time
		SandwichRule2	RI_SANDWICH2	not in time
		SandwichRule3	RI_SANDWICH3	late
	Starve	StarveRule	RI_STARVE	
	EatPizza	PizzaRule1	RI_PIZZA1	in time
		PizzaRule1	RI_PIZZA2	not in time
PizzaRule3		RI_PIZZA3	late	
 LUIGI	DeliverPizza	LuigiRule1	RI_PIZZA1	MQSTATE_NEW
		LuigiRule2	RI_PIZZA2	MQSTATE_CLEAR
	RepairMessage	RepairRule1	RI_REPAIR1	MQSTATE_DISABLED
		RepairRule2	RI_REPAIR2	MQSTATE_DISABLED_ WHILE_BUSY
		RepairRule3	RI_REPAIR3	any other state
 REPAIR	InquiryReply	RepairRule1	RI_REPAIR_INQ	in time
		RepairRule2	RI_REPAIR_NO	not in time
		RepairRule3	RI_REPAIR_LATE	late
   GREMLIN SHOPPING REPAIR	Show	ShowRule	RI_SHOW	display GUI
 LUIGI	GhostMessage	GremlinRule	RI_GREMLIN	can occur at any time
	 SHOPPING	InquiryRequest	RepairInqRule	
 SHOPPING	OrderMessage	OrderRule	RI_ORDER	
	FoodInquiry	FoodRule	RI_FOOD_INQ	
	none	TimerRule	RI_TIMER	Timer set in program
<p>Note: In this example we use rules that:</p> <ul style="list-style-type: none"> • are time dependent • depend on the state of the instance <p>Refer to page 135.</p>				

5.3.4.2 Rules and Methods for BLs

BLs are designed different than PLs. While PLs have only one entry point, namely is the window procedure for the GUI, BLs have an entry point for each method. That allows you to write a separate routine for each rule. In this example, we try to re-use common methods (routines). Table 20 on page 133 contains the methods and the associated programs (SourceName) for the BLs.

The methods perform the following functions:

- Sandwich:** Karen received a *FeedMe* request from Konrad and sends a wave of three messages to request material for the BLT.
- MakeBLT:** Within a specified time, Karen received either all, some or none of the material she sent for. If all material is there, she sends a wave of two messages to have the bacon cooked and the bread toasted. Otherwise she sends the *Starve* message to Konrad.
- Note:** We demonstrate how to process, in one method, a wave of messages that arrive in time or when the timer expired. Here, two rules invoke the same method.
- ServeBLT:** After toast and bacon arrived in time Karen completes her task by sending the *Sandwich* message to Konrad.
- Note:** For the second wave of messages we use two methods. One is invoked when toast and bacon arrive in time, and the other is invoked when the timer expires.
- NoBLT:** Either the toaster or the microwave is not responding in time. Karen sends the *Starve* message to Konrad.
- Ignore:** This method displays all late messages Karen receives. It also displays *RepairMessages* if there is nothing to repair.
- Gremlin:** In this method, the class that receives a *GremlinMessage* gets disabled.
- Repair:** In this method, the class that receives a *RepairMessage* is put back in working order.
- Deliver:** This method replies to Karen's requests for material. For each of the three classes, basket, breadbox and refrigerator, we include different source code. Each class maintains its own inventory file.
- Delivery:** This method is invoked when the grocer delivers food to the refrigerator, the bread basket and the bread box.
- FoodInquiry:** This method processes *FoodInquiry* messages sent from the shopping list. It responds with a message of the same name.
- Note:** These are INFORM messages. We demonstrate how a timer can be used to check if an instance is responding.
- Cook:** The same code is used for the toaster and the microwave. We demonstrate how to find out for which class the message was intended.
- SellFood:** The grocer received an order from the shopping list and sends *FoodDelivery* messages to the basket, breadbox and refrigerator.
- ClearUp:** This method is invoked when a BL receives a system message saying that the owner, Konrad, has ended. Here the programmer can write some code to be executed before the BL ends.

Table 20. BLT: Methods for Business Logics












 KAREN	 BASKET	 BREADBOX	 FRIDGE	 MICRO	 TOASTER	 GROCER	Method and Program Name	Source Name	Messages
X							Sandwich	bltorder	FeedMe
X							MakeBLT	bltmake	HaveTomato HaveBread HaveFromFridge
X							ServeBLT	bltserve	HaveBacon HaveToast (in time)
X							NoBLT	bltnone	HaveBacon HaveToast (timeout)
X	X	X	X				Ignore	xIgnore	RepairMessage (when nothing to repair) all late Have...
X	X	X	X	X	X	X	Gremlin	xGremlin	GhostMessage
X	X	X	X	X	X	X	Repair	xRepair	RepairMessage
X	X	X	X	X	X	X	Inquiry	xInquiry	InquiryMessage
	X	X	X				Deliver	basket1 bbox1 fridge1	GetTomato GetBread GetFromFridge
	X	X	X				Delivery	delivery	FoodDelivery
	X	X	X				FoodInquiry	foodinq	FoodInquiry
				X	X		Cook	cook	CookBacon MakeToast
						X	SellFood	grocer1	FoodOrder
X	X	X	X	X	X	X	ClearUp	xClear	MQ_SYSTEM_ OWNER_ENDED

Table 21. BLT: Rules for Business Logics

Class	Message	Rule Name	Rule ID	Method	Remarks
 KAREN	Feedme	SandwichRule1	RI_SANDWICH1	Sandwich	MQSTATE_NEW
		SandwichRule2	RI_SANDWICH2		MQSTATE_CLEAR
	HaveTomato HaveBread HaveFromFridge	MakeRule1	RI_MAKEBLT1	MakeBLT	in time
		MakeRule2	RI_MAKEBLT2		timeout
		TomatoRule	RI_TOMATO	Ignore	late
		BreadRule	RI_BREAD		
	FridgeRule	RI_FRIDGE			
	HaveTomato HaveBacon	ServeRule1	RI_SERVEBLT1	ServeBLT	in time
		ServeRule2	RI_SERVEBLT2	NoBLT	timeout
		ToastRule	RI_TOAST	Ignore	late
BaconRule		RI_BACON			
ALL CLASSES	GremlinMessage	GremlinRule	RI_GREMLIN	Gremlin	
	InquiryRequest	InquiryRule	RI_REPAIR_INQ	Inquiry	
	RepairMessage	RepairRule1	RI_REPAIR1	Repair	MQ_STATE_DISABLED no condition for BASKET, BREADBOX, FRIDGE
		RepairRule2	RI_REPAIR2		DISABLED_WHILE_BUSY
		RepairRule3	RI_REPAIR3	Ignore	other
OWNER_ENDED	OwnerEndedRule	RI_SYS_OE	ClearUp		
 BASKET BREADBOX FRIDGE	GetTomato GetBread GetFromFridge	DeliverRule1	RI_DELIVER1	Deliver	MQSTATE_NEW
		DeliverRule2	RI_DELIVER2		MQSTATE_CLEAR
	FoodInquiry	FoodInquiry	RI_FOOD_INQ	FoodInquiry	
	FoodDelivery	FoodRule	RI_FOOD	Delivery	
 MICRO TOASTER	CookBacon MakeToast	CookRule	RI_COOK	Cook	not MQSTATE_DISABLED
 GROCCER	FoodOrder	SellRule	RI_SELL	SellFood	not MQSTATE_DISABLED

In the business logic we use rules that

- are timed
- depend on the state the instance is in
- use roles to correlate messages

Timed rules: For a time dependent message we define either two or three rules. 3T processes the rules in the order they appear in the class source file.

- The first rule is satisfied when the message arrives.
- The second rule is a “timed rule” that is satisfied when the timer expires.

Note: The time is set with the MQTIME API.

- The third rule is satisfied when a reply arrives after the timer has expired. This rule is optional. 3T discards late replies if no “late rule” is defined.

Karen sends two waves of messages, one with three and the other with two messages. Let us look at the rules defined for the replies to the wave with two messages. We expect replies from the toaster and the microwave.

```

RULE          1
BEGIN          // messages arrive (in time)
  RuleId      RI_SERVEBLT1
  RuleName    ServeRule1
  MethodName  ServeBLT
  MsgIn       HaveToast, HaveBacon
END
RULE          2
BEGIN          // timer expired
  RuleId      RI_SERVEBLT2
  RuleName    ServeRule2
  MethodName  NoBLT
  Timed      Yes
  MsgIn       HaveToast PLACEHOLDER,
              HaveBacon PLACEHOLDER
END
RULE          3
BEGIN          // toast arrives late
  RuleId      RI_TOAST
  RuleName    ToastRule
  MethodName  Ignore
  MsgIn       HaveToast LATE
END
RULE          4
BEGIN          // cooked bacon arrives late
  RuleId      RI_BACON
  RuleName    BaconRule
  MethodName  Ignore
  MsgIn       HaveBacon LATE
END
```

1 This is a “regular” rule without any dependencies. The rule is satisfied when both messages are present. It does not matter if this occurs after seconds or days. If you want to limit the time a method waits for a message (or several messages) you have to specify a second rule that specifies a timeout.

2 This rule is a “timed” rule. It is satisfied after the timer has expired, regardless whether a message is present or not. The timer is set in the method that sends the request messages:

MQTIME (ByVal HInst, “ServeRule2”, ByVal 10, CompCode, Reason);

The parameters for the MQTIME API mean:

Table 22. MQTIME Parameters

Parameter	Description
ByVal HInst	The name of the instance the rule is for.
“ServeRule2”	The name of the rule (not the ID) that is timed.
ByVal 10	The time in seconds.
CompCode, Reason	The completion and reason codes are listed in the <i>Application Programming</i> manual.

The method NoBLT is invoked when no message is present or when only one message is present. We specify PLACEHOLDER for both messages. In the method NoBLT, we can check which message (if any) arrived. 3T supplies this information as a parameter when the procedure is called.

3 and **4** A rule is satisfied when either the message *HaveToast* or *HaveBacon* arrives after the timer for *ServeRule2* (**2**) has expired. If you do not specify a rule for late replies 3T discards them. You need one rule for each late message you want to process.

State dependent rules: We can use the state of the instance to control when a method shall be invoked. For example, Karen accepts the *FeedMe* message under two conditions:

- The state is MQSTATE_NEW
- The state is MQSTATE_CLEAR

An instance is in the NEW state when it is created, that means when the first message arrives. The *FeedMe* message is not necessarily the first message Karen receives. The Gremlin could send the first message, or an inquiry could be initiated from the repair list. The method that processes the first message sets the state to CLEAR or to DISABLED, if it is a message from the gremlin. For the *FeedMe* message we write two rules:

```

RULE
  BEGIN          // first BLT request
    RuleId      RI_SANDWICH1
    RuleName    SandwichRule1
    MethodName  Sandwich
    State      MATCHSTATE MQSTATE_NEW
    MsgIn       FeedMe
  END
RULE
  BEGIN          // next BLT request
    RuleId      RI_SANDWICH2
    RuleName    SandwichRule2
    MethodName  Sandwich
    State      MATCHSTATE MQSTATE_CLEAR
    MsgIn       FeedMe
  END

```

SandwichRule1 is satisfied only when *FeedMe* is the first message Karen receives. For all other BLT requests SandwichRule2 applies.

Since the class compiler does not let us "or" together two conditions (MQSTATE_NEW | MQSTATE_CLEAR) we have to write two rules, one for each state, using the condition MATCHSTATE.

Reversibly, toaster and microwave use a rule that is satisfied when a message arrives and they are *not* disabled. Therefore we write a rule that is satisfied when the state does not match a certain condition:

```
RULE
  BEGIN                                // bread arrives
    RuleId      RI_COOK
    RuleName    CookRule
    MethodName  Cook
    State      NOTMATCHSTATE  MQSTATE_DISABLED
    MsgIn       MakeToast
  END
```

MakeToast messages that arrive while the toaster is disabled remain in the queue until the state changes.

For the RepairMessage, that is sent when the user clicks on the Repair button in the repair list window, we write three rules:

1. If the state is MQSTATE_DISABLED invoke a method that sets the state to MQSTATE_CLEAR.
2. If the state is MQSTATE_DISABLED_WHILE_BUSY invoke a method that sets the state to MQSTATE_BUSY.
3. If the instance is in any other state the message is ignored, since there is nothing to repair. In our BLT sample we invoke the *Ignore* method that displays that such a situation occurred.

```
RULE                                // Set state to CLEAR
  BEGIN
    RuleId      RI_REPAIR1
    RuleName    RepairRule1
    MethodName  Repair
    State      MATCHSTATE  MQSTATE_DISABLED
    MsgIn       RepairMessage
  END
RULE                                // Set state to BUSY
  BEGIN
    RuleId      RI_REPAIR2
    RuleName    RepairRule2
    MethodName  Repair
    State      MATCHSTATE  MQSTATE_DISABLED_WHILE_BUSY
    MsgIn       RepairMessage
  END
RULE                                // Ignore message  (No state!)
  BEGIN
    RuleId      RI_REPAIR3
    RuleName    RepairRule3
    MethodName  Ignore
    MsgIn       RepairMessage
  END
```

As said before, 3T scans the rules in the order they appear in the class source file. Therefore, the *ignore* method can only be invoked if the state of the instance does not match the states defined in the previously defined rules.

Correlating messages: When Karen sends the two waves of messages she wants 3T to do the correlation of the replies for her. To tell 3T which requests and replies belong together we use the *role* keyword in the message. Let us look at the wave of two messages. The rules are defined on page 135. The messages are:

```
MESSAGE          // KAREN to TOASTER: make toast
BEGIN
  MsgName        MakeToast
  MsgType        REQUEST
  OperationCode  OC_TOAST
  Role          5
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
MESSAGE          // TOASTER to KAREN: here is toast
BEGIN
  MsgName        HaveToast
  MsgType        REPLY
  OperationCode  OC_TOAST
  Role          5
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
MESSAGE          // KAREN to microwave: cook bacon
BEGIN
  MsgName        CookBacon
  MsgType        REQUEST
  OperationCode  OC_COOK
  Role          6
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
MESSAGE          // Microwave to KAREN: here is the bacon
BEGIN
  MsgName        HaveBacon
  MsgType        REPLY
  OperationCode  OC_COOK
  Role          6
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
```

We use two roles, 5 and 6. The first role ties *MakeToast* and *HaveToast* together, the second role connects *CookBacon* and *HaveBacon*. The roles are only used when the BLM receives the above messages for Karen.

The classes that receive the messages (toaster and microwave) do not need the roles. The class compiler displays warning messages telling you that the roles are ignored for those classes.

5.4 Design Crosscheck

After the design is completed and the class source files are written, we use the 3T class compiler to check out the design. The crosscheck function tells us errors, such as:

- A method sends a message but none receives it
- A rule is defined for a message but no method sends it
- A rule is defined but a method is not

To perform the crosscheck you have to create a file that contains a list of all classes the application uses. For the BLT example, we created the file *classes.lst* in Figure 91.

```
KONRAD
LUIGI
GREMLIN
SHOPPING
REPAIR
KAREN
BASKET
BREADBOX
FRIDGE
TOASTER
MICRO
GROCER
```

Figure 91. Input File for Design Crosscheck: "classes.lst"

To invoke the crosschecker type:

```
bmqcc -x classes.lst
```

The output will be in the file *classes.xck*. Figure 92 on page 140 shows examples of the output. During processing you will see warning messages like this:

```
*** Parsing the class source file 'fridge.cs' ***
fridge.cs(124): warning: BMQ1439: Role 3 is ignored. Role is ignored for MsgOut
REPLY messages.
fridge.cs(114): warning: BMQ1439: Role 3 is ignored. Role is ignored for MsgIn
REQUEST messages.
fridge.cs(114): warning: BMQ1439: Role 3 is ignored. Role is ignored for MsgIn
REQUEST messages.

*** Checking the msgin/msgout of class FRIDGE ***
*** CHECKED ***

BMQCC: 3 warning(s) detected

*** Creating binary class file 'fridge.cb' ***
*** CREATED ***
```

Roles are used to correlate reply messages in the receiver.

- The first warning (124) is produced because HaveFromFridge is a reply message that the refrigerator sends. The refrigerator does not need a role. 3T needs it when it receives the message from the refrigerator for Karen to correlate it with HaveBread and HaveTomato that are sent by from the breadbox and the vegetable basket.
- The second warning (114) is produced because GetFromFridge is a request message that the refrigerator receives. 3T needs the role for Karen to match GetFromFridge with HaveFromFridge.

Notes:

1. A single request/reply pair does not need a role.
2. Do not specify roles for INFORM messages.

```
* * * CROSS REFERENCE LISTING * * *  
  
MESSAGE NAME          ATTRIBUTES  
  
CookBacon             MessageType = REQUEST, OperationCode = 65545,  
                      OperationVersion = 1, Role = 6,  
                      Format = FIXED, StrucFile = bltstruc.h,  
                      StrucName = MSG100,  
                      StrucLen = 100, ConversionDLL = MSG100,  
                      Senders = KAREN,  
                      Receivers = MICRO.  
  
HaveBacon             MessageType = REPLY, OperationCode = 65545,  
                      OperationVersion = 1, Role = 6,  
                      Format = FIXED, StrucFile = bltstruc.h,  
                      StrucName = MSG100,  
                      StrucLen = 100, ConversionDLL = MSG100,  
                      Senders = MICRO,  
                      Receivers = KAREN.  
  
:  
* * * POTENTIAL REASON FOR NO MESSAGE FLOW * * *  
  
MESSAGE NAME          REASON  
  
CookBacon             SEND - This message may not be sent by a BL program.  
                      RECEIVE - NONE  
  
HaveBacon             SEND - This message may not be sent by a BL program.  
                      RECEIVE - NONE  
  
StartJob              SEND - This message cannot be sent by any class,  
                      because there are no methods defined to send it.  
                      RECEIVE - NONE
```

Figure 92. Output File from Design Crosscheck: "classes.xck"

Note: Compile each class file to eliminate syntax errors before you run the crosscheck.

5.5 Building the GUIs

We use Visual Basic to build the GUIs for the Windows clients. A description of the prototypes is in 5.2.3, “GUI Prototypes” on page 118. There are five GUIs for:



Konrad



Luigi



Gremlin



Shopping list



Repair list

Before we start designing the GUIs and writing the code some preparations have to be made:

- The class binary files (.CB) for the PLs that have been created on the AIX machine have to be copied into our Windows workstation. Copy the five files into a directory that is in the search path in the AUTOEXEC.BAT file on page 51, such as C:\3TIERW\SAMPLES\WIN. The binary files are:
 - Konrad.cb
 - Luigi.cb
 - Gremlin.cb
 - Shopping.cb
 - Repair.cb
- Create a directory that will hold all files that we need to develop and test the GUIs: C:\VB\BLT.
- For the GUI development we need a file that contains constants and global variables. Since we need most of the definitions from the file *BLTDEF.H* in A.15, “Definitions for Class Source Files” on page 247, we convert this file into a Visual Basic file, *BLTDEF.BAS*, and copy it into the new directory.
- To test a GUI we need a profile to start the PLM. We place this file, shown in Figure 93, also in the new directory. The class names are case sensitive. For each of the classes there must be a .CB file in the search path.

```
*****
*
* PLMS.PRF: Startup profile for the PL Manager *
*
*****

[CLIENT]

ClassNames = KONRAD LUIGI GREMLIN REPAIR SHOPPING
LogLevel = 300
```

Figure 93. Profile to Start Five PLs

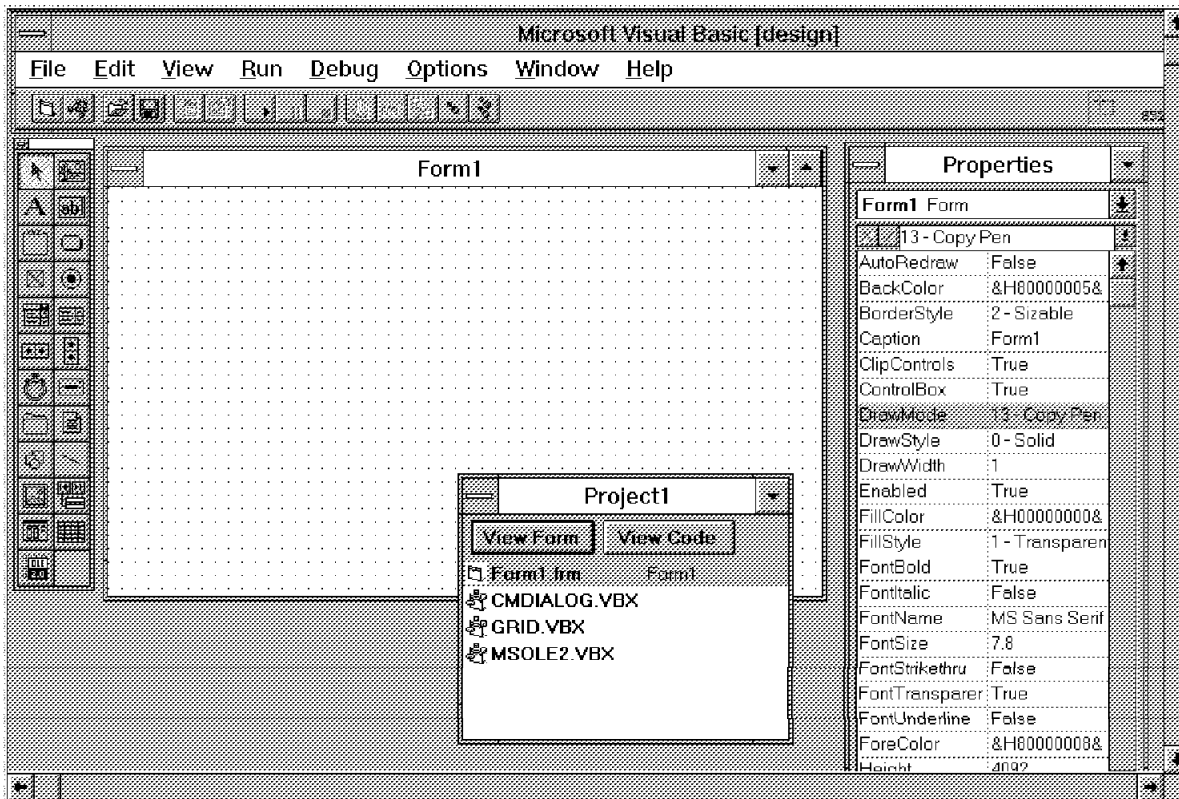


Figure 94. Microsoft Visual Basic (design) Window

When you bring up Visual Basic you will see what is shown in Figure 94.

The project window shows three custom controls (VBX files) that we do not need. Therefore, we delete them, one at a time. Click on one file name in the *Project1* window and then select **Remove File** from the File menu. The file name disappears instantly.

For our GUIs we need the 3T custom control BMQNTFY.VBX from the directory C:\WINDOWS\SYSTEM. To add the 3T custom control to the project select **Add file** from the File menu. In the Add File window (Figure 95 on page 143) choose

- **Custom Controls** from the *List Files of Type* list box
- **C:\WINDOWS\SYSTEM** from the Directories list box

When you click on **OK**, the oaktree icon appears on the bottom of the toolbox.

Next we add to the project three files that contain definitions MQ3T needs. The files have been installed in the directory C:\3tierw\vbsupp. Figure 96 on page 143 shows the Add File window for these files.

Using the same method, we add the file BLTDEF.BAS to the project. This file is in the new \BLT directory.

Note: For the BLT GUIs we create five Visual Basic projects, one for each GUI. When it compiles, Visual Basic creates one EXE for a project. We will create five GUI programs that can run in one or five different workstations. Therefore, we create one (Visual Basic) project for each GUI. This allows us to create five separate EXEs.

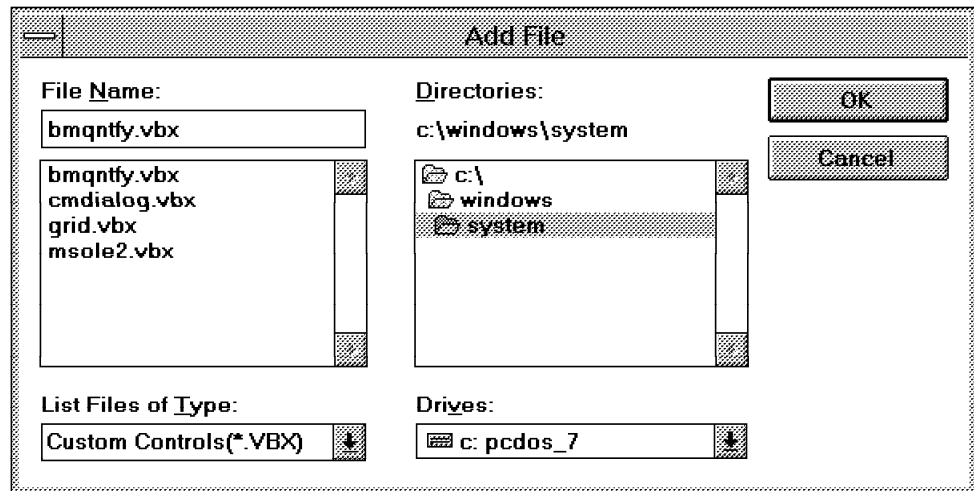


Figure 95. Add BMQNTFY.VBX to a Project

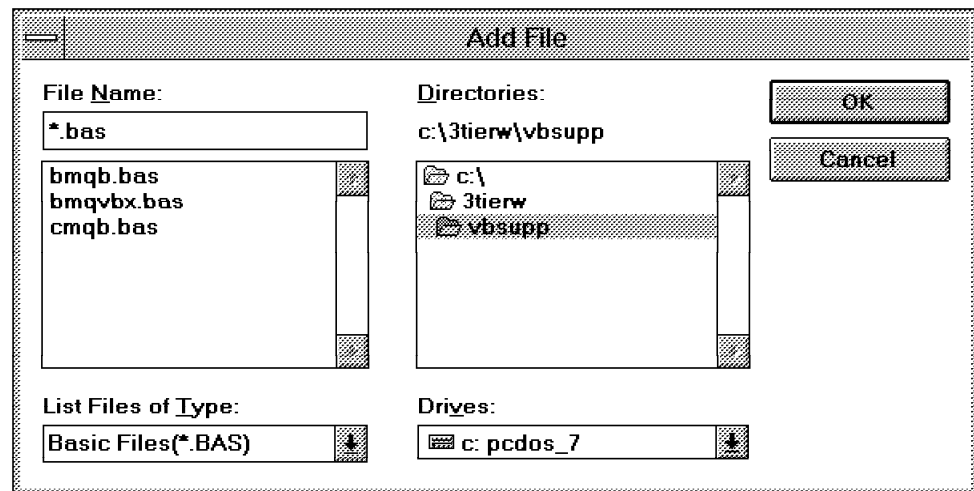


Figure 96. Add MQ3T Files to a Project

The following explains what has to be done to each of the forms:

- Size the form and make it smaller.
- Change the name of the form:
 1. Click on **Name** in the Properties window.
 2. Type the new name, for example Konrad, next to the checkmark in the Properties window.
 3. Click on the checkmark.

This changes the name in the project window instantly.
- Change the title of the form:
 1. Click on **Caption** in the Properties window.
 2. Type the new caption next to the checkmark, for example, BLT - KONRAD.
- Move timer and the oaktree objects into the form.

3T requires these objects. It does not matter where you place them since they are invisible during run time.

- Save the project under its new name, Konrad, in the directory C:\VB\BLT.

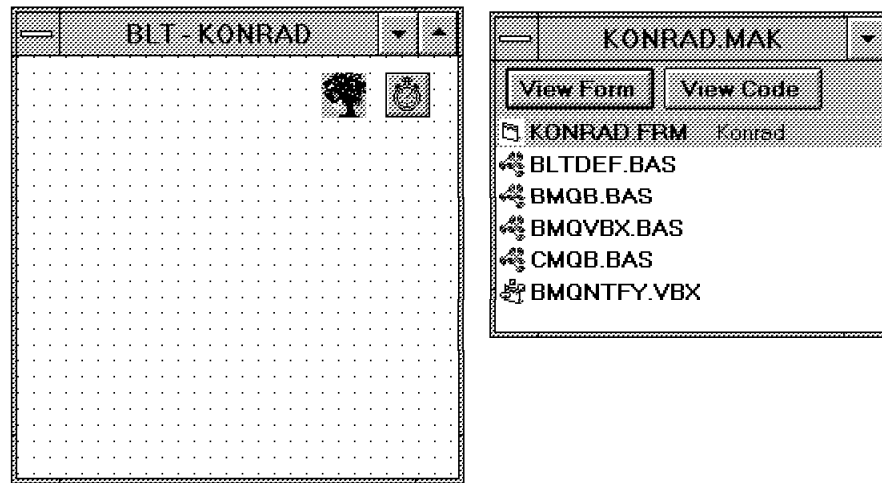


Figure 97. Generic Frame and Project Window for BLT

5.5.1 Project Konrad

To create this and the other four Visual Basic projects we have to:

- Design the GUI
- Write the Basic code for the events
- Test the program from within Visual Basic (during coding)
- Compile the project when completed
- Test the program

5.5.1.1 Create the GUI for Konrad

We use the frame in Figure 97 as a base and add the following controls:

- A label with the text BLT Application:
Double-click on the **Label** icon in the toolbox. Move the label from the center to the top of the frame. Then highlight *Caption* in the Properties window for the label and type, next to the checkmark, BLT Application. You may have to resize the label to make the text fit.
- A label with the text Messages:
This is done in the same fashion as described above.
- A text box to display messages:
This box will be used by the program to display messages.
 - Double-click on the **Text box** icon in the tool box.
 - Size the field so that it is wide enough for about 25 characters and high enough to display about eight lines.
 - Set the *FontSize* Property to 9.6.
 - Set the *MultiLine* Property to True.

- Set the *ScrollBars* Property to 2 - vertical.
- Set the *TabStop* Property to False.
- Make the *Text* Property blank.
- A label with the text "Order" above the command buttons
- Three command (push) buttons:
 - Double-click on the **Commandbutton** icon in the tool box.
 - Move the object from the center to the bottom of the form.
 - Change the *Caption* property to BLT, Pizza, and Quit.
 - Change the *Name* property to BLT, Pizza, and Quit.
 - Set the *TabIndex* property to 0, 1, and 2 to allow the user to use the keyboard instead of the mouse.

Use the mouse to align the buttons on the bottom of the form.

Figure 98 shows how the completed form appears in the Visual Basic design window (left) and at run-time (right). You could run the program. However, all it does is tab from one command button to the next.

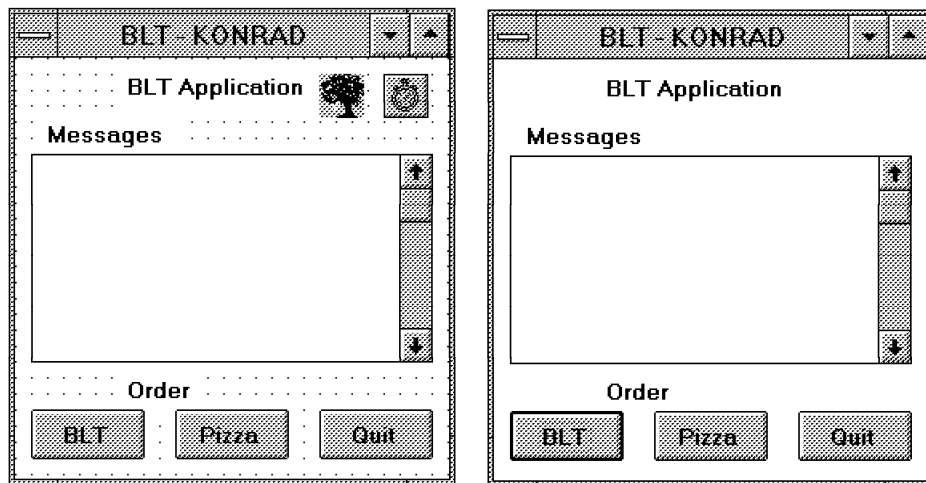


Figure 98. Konrad's Frame at Design and Run Time

5.5.1.2 Write Basic Code Used for all PLs

To make the Konrad's window do something we have to give it life. This is done by writing some code for the events that we want to handle. Such events occur when

- The form is loaded.
- A push button is clicked.
- A message arrives.

Form_Load is the procedure that is invoked when the form is loaded, that is when the GUI appears on the screen. To write code for this event double-click on the form (but not on a control). You see that this procedure does not contain any code.

The purpose of this routine is:

- To ensure that a class name was passed to the program
- To register the program with 3T

```
Sub Form_Load ()
  If Command = "" Then
    MsgBox "PL requires parameter Class Name.", 16, "Input Error"
  End
End If

vPLClass = Command      'Convert string to class name

' Register with 3T
MQREG ByVal vPLClass, 1, ByVal OAK1.hWnd, ByVal BMQ_NOTIFY,
      ByVal MQRGO_REMOVE_LIST_ENTRIES, CompCode, Reason
DisplayCompCode "MQREG"
End Sub
```

Figure 99. Form_Load Procedure.

Note: Write the MQREG call in one line!

Form_Unload is executed when all forms of an application are closed.

```
Sub Form_Unload (Cancel As Integer)
  Close_Click      'common exit path
End Sub
```

Figure 100. Form_Unload Procedure

Quit_Click is invoked when the third command button, Quit, is pressed. In this routine we call the procedure "Close_Click" by inserting one line:

```
Sub Quit_Click ()
  Close_Click      'common exit path
End Sub
```

Figure 101. Quit Procedure

The above programs call two subroutines or procedures, namely Close_Click and DisplayCompCode. To add a new procedure to the project click on **View Code** in the project window KONRAD.MAK and then on **New Procedure** in the View menu. In the *New Procedure* window (Figure 102) type the procedure name and click on **OK**. Write the code for the procedure in the window that appears next.

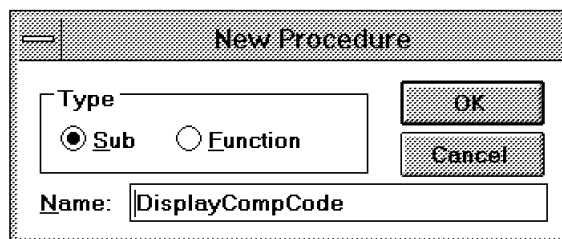


Figure 102. Visual Basic: Create a New Procedure

Note: You may copy and paste code from other projects, such as HELLO1.

Close_Click is used to end the instance and unregister the GUI from MQ3T. The code is shown in Figure 33 on page 55.

DisplayCompCode is used to display MQ3T return codes. The code is shown in Figure 30 on page 54.

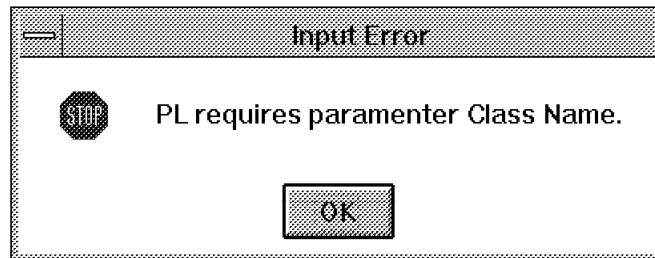
Declarations: We define CompCode and Reason as shown in Figure 29 on page 54. For Konrad we need a few more declarations, as you can see in Figure 99 on page 146.

```
Option Explicit
Dim CompCode As Long ' return code
Dim Reason As Long ' return code
Dim NL As String ' holds new line characters
Dim DSMsg As String ' buffer to display messages
Dim ij As Long ' work field
Dim szClass As String ' class name
Dim szInstance As String ' instance name
```

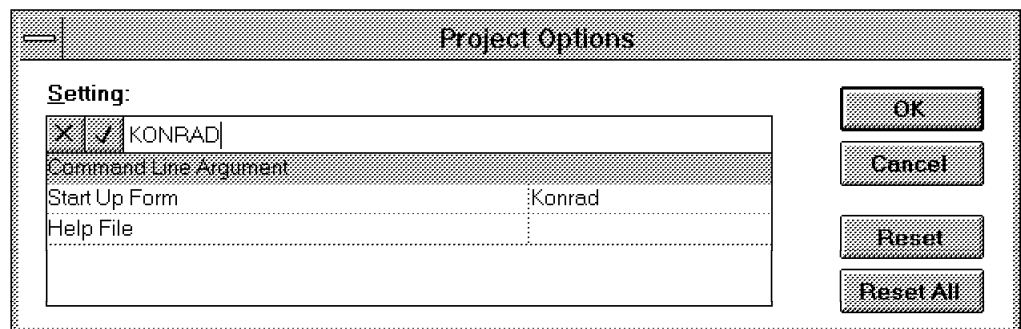
Figure 103. BLT: Declarations

5.5.1.3 Perform the First Test for Konrad

At this time you could test the program from within Visual Basic. To start Konrad click on **Start** in the Run menu, or press F5. However, instead of the form (window) the following error message will be displayed:



How do we provide this parameter? Click on **Project** in the Options menu and type the class name KONRAD in the subsequent window, shown below. The class name is case sensitive! Then press Enter.



Now you can run the program. You can use the Tab keys to tab from one push button to the next, and you can quit the program. For the other events we still have to write the code.

5.5.1.4 Write the Basic Code for Konrad

BLT_Click is invoked when the user clicks on the BLT push button. The request message FeedMe is sent to Karen.

- 1** If the user clicks the push button again before Karen replies, the second MQSEND will fail. You cannot send the same request message twice. You must wait for either the reply or a timeout.
- 2** The timeout value, in seconds, is set with the MQTIME call.
- 3** To prevent Konrad from sending a second request and to prevent the error message (**1**) we can simply disable the pushbuttons.
- 4** We display messages in the frame's text box, and not in message boxes.

```
Sub BLT_Click ()
  Dim msg100 As msg100          ' message buffer

  MQSEND ByVal vHInst, ByVal "KAREN", ByVal szInstance,
          ByVal "FeedMe", 0, msg100, CompCode, Reason
  1
  If CompCode = MQCC_FAILED And Reason = MQRC_REPLY_ALREADY_EXPECTED Then
    4
    DSMsg = "Don't ask for a BLT twice"
    DS_MLE
  Else
    DisplayCompCode "MQSEND"
  End If
  2
  MQTIME ByVal vHInst, ByVal "SandwichRule2", 10, CompCode, Reason
  DisplayCompCode "MQTIME"
  4
  DSMsg = "BLT requested. Wait..."
  DS_MLE
  3
  BLT.Enabled = False
  Pizza.Enabled = False
End Sub
```

Figure 104. Konrad: BLT Push Button Procedure

Pizza_Click is invoked when the user clicks on the Pizza push button. The request message DeliverPizza is sent to Luigi. The code is very similar to BLT_Click.

```
Sub Pizza_Click ()
  :
  :
  MQSEND ByVal vHInst, ByVal "LUIGI", ByVal szInstance,
          ByVal "DeliverPizza", 0, msg100, CompCode, Reason

  If CompCode = MQCC_FAILED And Reason = MQRC_REPLY_ALREADY_EXPECTED Then
    DSMsg = "Don't ask for a pizza twice"
  :
  :
  MQTIME ByVal vHInst, ByVal "PizzaRule2", 10, CompCode, Reason
  :
  :
  DSMsg = "Pizza requested. Wait..."
  :
  :
End Sub
```

Figure 105. Konrad: Pizza Push Button Procedure

DS_MLE displays messages in the text box in a GUI. The calling program has to prepare the message text in the global variable DSMsg. The routine ensures that the contents of the text box is limited to about 450 characters.

```

Sub DS_MLE ()
Dim length As Long

    NL = Chr(13) + Chr(10)           ' new line
    length = Len(Text1.Text)         ' current length
    If length > 450 Or length = 0 Then
        Text1.Text = DSMsg          ' overwrite text
    Else
        Text1.Text = Text1.Text & NL & DSMsg ' append text
    End If
End Sub

```

Figure 106. BLT: Display Messages in Text Box

OAK1_NewEvent is invoked when the Visual Basic run-time program receives a message from the PLM (via BMQNTFY.VBX). Refer to 3.4, “Parameter Passing” on page 47 for detailed explanations. We use the same routine in all GUIs.

- ProcessPLevent is called when a rule is satisfied. This routine processes the message.
- Close_Click is called when the instance has been deleted. This allows us to unregister and automatically close the window when the job owner ended.
- When the instance has been deleted we unregister it. This happens when the job owner’s (Konrad’s) window was closed.
- When the instance has been unregistered we simply end the program.

```

Sub OAK1_NewEvent (msg As Integer, wp As Integer, lp As Long)

    If wp = MQPLM_RULE_SATISFIED Then
        vHInst = lp
        ProcessPLevent ByVal lp ' make call to handle the event

    ElseIf wp = MQPLM_INSTANCE_DELETED Then
        MQUREG ByVal vPLClass, ByVal OAK1.hWnd, ByVal MQURGO_FORCE, CompCode, Reason
        DisplayCompCode "MQUREG"

    ElseIf wp = MQPLM_HWND_UNREGISTERED Then
        End ' already unregistered, so simply exit
    End If

End Sub

```

Figure 107. BLT: Events from MQ3T

ProcessPLEvent processes all messages the PL receives from other classes.

```
Sub ProcessPLEvent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure
    Dim BufferLen As Long             ' buffer length
    Dim msg100 As msg100             ' buffer - NB Don't define as String

    1
    ' query information about the current event
    MQQRYE ByVal HInst, MQevent, CompCode, Reason
    DisplayCompCode "MQQRYE"

    2
    Select Case MQevent.RuleId

    3
    Case RI_STARTJOB
        a szClass = RTrim(MQevent.Classname)
           szInstance = RTrim(MQevent.LocalInstanceName)
        b Konrad.Caption = szClass & " / " & szInstance
           MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
           DisplayCompCode "MQENDE"

        c MQSEND ByVal HInst, ByVal "REPAIR", ByVal szInstance,
           ByVal "Show", 0, msg100, CompCode, Reason
           DisplayCompCode "MQSEND - 1"
           MQSEND ByVal HInst, ByVal "SHOPPING", ByVal szInstance,
           ByVal "Show", 0, msg100, CompCode, Reason
           DisplayCompCode "MQSEND - 2"
           MQSEND ByVal HInst, ByVal "GREMLIN", ByVal szInstance,
           ByVal "Show", 0, msg100, CompCode, Reason
           DisplayCompCode "MQSEND - 3"
        Exit Sub

    4
    Case RI_SANDWICH1
        DSMsg = "The BLT is served."
    Case RI_SANDWICH2
        DSMsg = "It is too late for a BLT."
    Case RI_SANDWICH3
        DSMsg = "BLT arrived too late"
    Case RI_STARVE
        DSMsg = "Starve!"
    Case RI_PIZZA1
        DSMsg = "Pizza is delivered"
    Case RI_PIZZA2
        DSMsg = "It is too late for a pizza."
    Case RI_PIZZA3
        DSMsg = "Pizza arrived late."

    End Select

    5
    BLT.Enabled = True
    Pizza.Enabled = True

    MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
    DisplayCompCode "MQENDE"

End Sub
```

Figure 108. Konrad: Process Messages.

Note: Write the MQSEND instructions in one line!

1 We query the event to obtain the rule ID from the event structure MQevent.

2 The following statements process the incoming messages. What message arrived is determined by the rule ID that was defined in the 3T message descriptions, file *messages.ch*.

3 When the STARTJOB message arrives execute thee functions:

a We save class name and instance name of Konrad. The instance name, obtained from the STARTJOB command, is used for all classes in the job.

b We change the caption (header line) of the window to include class and instance names.

b To display the repair list, shopping list, and the window for the Gremlin, we send a *Show* message to each of the classes. The instance name is the same as for Konrad.

Note: Luigi's window is displayed when he receives the first pizza order.

4 When any of the other messages arrive we display a message.

5 Since the BLT or pizza order is completed, either in time or when the timer expired, we enable the two push buttons in the GUI to allow for more work to be sent to Luigi or Karen. Then we end the method.

5.5.2 Project Luigi

You can develop the GUI and the code for Luigi in the same fashion as you did for Konrad. However, since Konrad's and Luigi's forms are very similar, you may copy and change the following files with your favorite editor:

- KONRAD.MAK becomes LUIGI.MAK
- KONRAD.FRM becomes LUIGI.FRM

5.5.2.1 Create the GUI for Luigi

In Luigi's project file change Konrad to Luigi as marked in Figure 109.

```
LUIGI.FRM                                <=== changed
C:\WINDOWS\SYSTEM\BMQNTFY.VBX
C:\3TIERW\VBSUPP\BMQB.BAS
C:\3TIERW\VBSUPP\BMQVBX.BAS
C:\3TIERW\VBSUPP\CMQB.BAS
BLTDEF.BAS
ProjWinSize=105,459,252,204
ProjWinShow=2
Command="LUIGI"                          <=== changed
IconForm="Luigi"                         <=== changed
Title="LUIGI"                            <=== changed
ExeName="LUIGI.EXE"                      <=== changed
Path="C:"
```

Figure 109. Luigi's Project File

In the file LUIGI.FRM change the second and third line:

```
VERSION 2.00
Begin Form Luigi                          <=== changed
    Caption          = BLT - LUIGI        <=== changed
    :
    :
```

To work with Luigi's frame select **Open project** from Visual Basic's File menu and in the subsequent Open Project window select **luigi.mak** from the C:\VB\BLT

directory. Then click on **OK**. The following steps outline how to create Luigi's frame:

- Delete the push button in the middle:
 - Click on the button labelled **Pizza** and press the Del key. The push button disappears.
 - In the project window click on **View Code**
 - Select the object **general** and the procedure **Pizza_Click** This is the Basic code associated with the deleted push button.
 - Delete all the code to erase the procedure.
- Move the label "Order" to the left so that it positioned above the left push button and change the caption in the "Label3" properties window to "Action".
- Change the caption of the left push button to "Deliver".

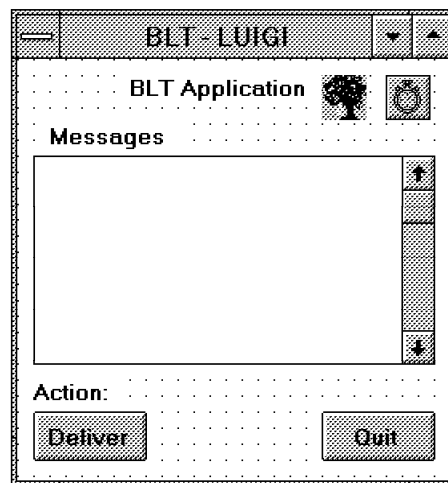


Figure 110. Luigi's Frame at Design Time

5.5.2.2 Write the Basic Code for Luigi

Most of the procedures are the same for all GUIs. For Luigi, we have to modify or re-code two routines:

Deliver_Click sends the reply message *EatPizza* to Konrad.

1 The MQQRY call is issued to obtain the instance description in the structure MQid. This structure contains the state the instance is in.

2 If the state is *clear* then Luigi is waiting for work. When a pizza order arrives the state is set to *busy*.

3 A message from the Gremlin sets Luigi's state to *disabled*. If in this state, Luigi did not receive a *DeliverPizza* message.

4 When Luigi is busy, that is he received a *DeliverPizza* message but did not deliver yet, and a message from the gremlin arrives the state is set to *disabled while busy* baking the pizza. In this state Luigi is not able to deliver.

5 Luigi delivers the pizza. The instance state is set to *clear*. In that state Luigi can receive more work.

```
Sub Deliver_Click ()
  Dim msg100 As msg100
  Dim MQid As MQid
  Dim BufferLength As Long
  1
  BufferLength = Len(MQid)
  MQQRY ByVal vHInst, "", ByVal MQQRYT_INSTANCE, BufferLength,
    MQid, CompCode, Reason
  DisplayCompCode "MQQRY"
  2
  If MQid.InstanceState = MQSTATE_CLEAR Then
    DSMsg = "No pizza order received!"
    DS_MLE
    Exit Sub
  End If
  3
  If MQid.InstanceState = MQSTATE_DISABLED Then
    DSMsg = "Luigi is resting!"
    DS_MLE
    Exit Sub
  End If
  4
  If MQid.InstanceState = MQSTATE_DISABLED_WHILE_BUSY Then
    DSMsg = "Luigi is called away"
    DS_MLE
    Exit Sub
  End If
  5
  MQRPLY ByVal vHInst, ByVal "EatPizza", 0, msg100, CompCode, Reason
  DisplayCompCode "MQRPLY"
  DSMsg = "Pizza is on the way."
  DS_MLE
  MQSETS ByVal vHInst, ByVal MQSTATE_CLEAR, CompCode, Reason
  DisplayCompCode "MQSETS"
End Sub
```

Figure 111. Luigi: Deliver Procedure

ProcessPLEvent processes all messages Luigi can receive:

- *DeliverPizza* makes Luigi busy.
- *GremlinMessage* sets Luigi's state to disabled or disabled while busy.
- *RepairMessage* resets Luigi's state to busy or clear.
- *RepairInquiry* reports Luigi's state to the repair list.

The following notes refer to Figure 112 on page 154.

6 The MQQRY call is issued to obtain the instance description in the structure MQid. This structure contains the state the instance is in.

7 If the state is *new* then Luigi's window just appeared on the screen. We update the caption in the window's header with the class and instance names and set the state to *clear*. Luigi's state is never *new* again.

8 We query the current event again and save its new state in wkfld. The state is used further on in the routine. The field wkfld contains or will contain the state the instance is in when the procedure exits.

```

Sub ProcessPLEvent (ByVal HInst As Long)
    Dim MQevent As MQevent          ' event structure
    Dim BufferLen As Long            ' buffer length
    Dim msg100 As msg100            ' buffer
    Dim wkfld As Long               ' to store instance state

6
MQQRYE ByVal HInst, MQevent, CompCode, Reason
DisplayCompCode "MQQRYE"

7
If MQevent.InstanceState = MQSTATE_NEW Then
    szClass = RTrim(MQevent.Classname)
    szInstance = RTrim(MQevent.LocalInstanceName)
    Luigi.Caption = szClass & " / " & szInstance
    MQSETS ByVal HInst, ByVal MQSTATE_CLEAR, CompCode, Reason
    DisplayCompCode "MQSETS"
End If

8
MQQRYE ByVal HInst, MQevent, CompCode, Reason
DisplayCompCode "MQQRYE"
wkfld = MQevent.InstanceState

9
Select Case MQevent.RuleId
a    Case RI_PIZZA1, RI_PIZZA2          ' state = NEW, CLEAR
        DSMsg = "Deliver a pizza!"
        wkfld = MQSTATE_BUSY
b    Case RI_GREMLIN
        DSMsg = "Luigi takes a break."
        If MQevent.InstanceState = MQSTATE_BUSY Then
            wkfld = MQSTATE_DISABLED_WHILE_BUSY
        Else
            wkfld = MQSTATE_DISABLED
        End If
c    Case RI_REPAIR1
        DSMsg = "Luigi is back ..."
        wkfld = MQSTATE_CLEAR
        Case RI_REPAIR2
        DSMsg = "Luigi is back ..."
        wkfld = MQSTATE_BUSY
        Case RI_REPAIR3
        DSMsg = "Ignore repair ..."
d    Case RI_REPAIR_INQ

        Select Case MQevent.InstanceState
            Case MQSTATE_NEW
                msg100.message = "NEW"
            Case MQSTATE_CLEAR
                msg100.message = "CLEAR"
            Case MQSTATE_BUSY
                msg100.message = "BUSY"
            Case MQSTATE_DISABLED
                msg100.message = "DISABLED"
            Case MQSTATE_DISABLED_WHILE_BUSY
                msg100.message = "DISABLED_WHILE_BUSY"
            Case MQSTATE_END
                msg100.message = "END"
        End Select

        DSMsg = "Inquiry:" & msg100.message
        msg100.number = MQevent.InstanceState
        MQRPLY ByVal HInst, ByVal "InquiryReply", 0, msg100, CompCode, Reason
        DisplayCompCode "MQRPLY"
    End Select

10
DSMsg = DSMsg & wkfld
DS_MLE
MQENDE ByVal HInst, ByVal wkfld, CompCode, Reason
DisplayCompCode "MQENDE"
End Sub

```

Figure 112. Luigi: Process PL Events

9 The rule ID tells us what message has arrived and what the state was when the message arrived.

a Two rules are defined for the *DeliverPizza* message:

- RI_PIZZA1 is satisfied when Luigi's state is MQSTATE_NEW
- RI_PIZZA2 is satisfied when Luigi's state is MQSTATE_CLEAR

The PLM does not invoke this procedure when the *DeliverPizza* message arrives and Luigi is in any other state.

b The *GremlinMessage* disables Luigi. The state is set to "disabled" or "disabled while busy". The latter is true when a *DeliverPizza* has been received but the delivery has not been taken place.

c Three rules are defined for the *RepairMessage*:

- RI_REPAIR1 is satisfied when Luigi is disabled.
- RI_REPAIR2 is satisfied when the gremlin disabled Luigi while he was baking a pizza.
- RI_REPAIR3 is satisfied when Luigi is in any other state. In this case we ignore the message.

d When an *InquiryMessage* arrives we respond with an *InquiryReply* that contains Luigi's state in two forms, as an integer and as an 20-byte character string.

10 We change the state to the value in *wkfld* and display the new state in Luigi's window.

5.5.3 Project Gremlin

The Gremlin is the simplest presentation logic of all. It sends only one INFORM message and receives no messages except for the *Show* message from Konrad that initially displays the window.

The form contains twelve radio buttons. The selected radio button determines where the message is sent to. The button number (1 to 12) is sent with the message. This number denotes what action the receiving program has to perform.

5.5.3.1 Create the GUI for Gremlin

To create the project for the Gremlin we copy Luigi's project file and change the names as shown below:

```
GREMLIN.FRM                                <=== changed
C:\WINDOWS\SYSTEM\BMQNTFY.VBX
C:\3TIERW\VBSUPP\BMQB.BAS
C:\3TIERW\VBSUPP\BMQVBX.BAS
C:\3TIERW\VBSUPP\CMQB.BAS
BLTDEF.BAS
ProjWinSize=105,459,252,204
ProjWinShow=2
Command="GREMLIN"                          <=== changed
IconForm="Gremlin"                          <=== changed
Title="GREMLIN"                             <=== changed
ExeName="GREMLIN.EXE"                       <=== changed
Path="C:"
```

Figure 113. The Gremlin's Project File

You may modify the file GREMLIN.FRM to change the form before you use Visual Basic to add the radio buttons:

- Change the names in the second and third line.
- Delete the block for the label "Action".
- Alter name and caption of the left command button. from "Deliver" to "Enter"

```
VERSION 2.00
Begin Form Gremlin                                <=== changed
  Caption          = BLT - GREMLIN                <=== changed
  :
  Begin CommandButton Enter                       <=== changed
    Caption        = "Enter"                      <=== changed
    Height         = 372
    Left          = 240
    TabIndex      = 14
    Top           = 7200
    Width         = 852
  End
```

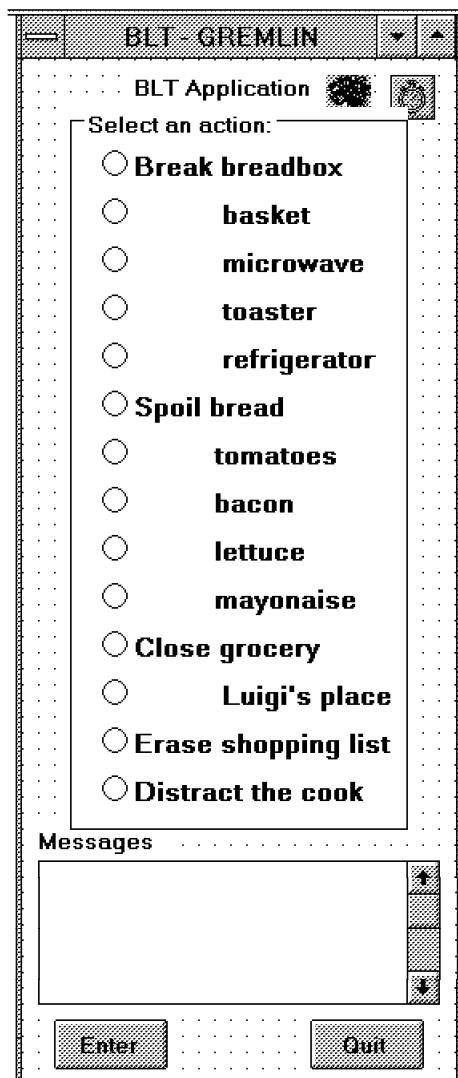


Figure 114. The Gremlin's Frame at Design Time

Now go into Visual Basic and resize the form to fit 14 radio buttons between the header line and the text box. You may have to make the text box shorter.

Create a *frame* for the radio buttons first. Change its caption to "Select an action". Inside this frame place the radio buttons as shown in Figure 114 on page 156.

To create a radio (or option) button:

- Double-click on the option button icon in the toolbox.
- Move the control to the top of the frame and size it to a height of 372. You see this number in the properties window.
- Change the caption to "Break breadbox"
- Change the font size to 9.6. You may have to resize the control to make the text fit.

You may copy the remaining radio buttons:

- Click on the previously created control.
- Press Ctrl+C and then Ctrl+V.
- Answer "no" when asked if you want to create a control array.
- Move the control below the previously created one.
- Change the values in the properties window.

Make sure that the tab order is in the order you desire. If not, change that value in the properties window for the radio and push buttons.

5.5.3.2 Write the Basic Code for Gremlin

Since most of the procedures are the same for all GUIs we have to write very little code.

To the definition we add the two fields below:

```
⋮  
Dim action As Integer      ' number of radio button selected  
Dim destin As String      ' destination class name
```

Option1_Click: Into the above fields we store values when the user clicks on a radio button. The following routine is called when the user chooses to bread the breadbox.

```
Sub Option1_Click ()  
    action = 1          ' number of radio button selected  
    destin = "BREADBOX" ' destination class  
End Sub
```

Figure 115. The Gremlin's Radio Button Procedure

Double-click on a radio button control and a window appears to what you add two lines. Table 23 on page 158 specified to what values action and destination have to be set to.

Table 23. Gremlin: Actions and Destinations for Radio Buttons			
Button	Procedure	Action	Destination
Break breadbox	Sub Option1_Click	1	BREADBOX
Break basket	Sub Option2_Click	2	BASKET
Break microwave	Sub Option3_Click	3	MICRO
Break toaster	Sub Option4_Click	4	TOASTER
Break refrigerator	Sub Option5_Click	5	FRIDGE
Spoil bread	Sub Option6_Click	6	BREADBOX
Spoil tomatoes	Sub Option6_Click	7	BASKET
Spoil bacon	Sub Option8_Click	8	FRIDGE
Spoil lettuce	Sub Option9_Click	9	FRIDGE
Spoil mayonaise	Sub Option10_Click	10	FRIDGE
Close grocery	Sub Option11_Click	11	GROCER
Close Luigi's	Sub Option12_Click	12	LUIGI
Erase shopping list	Sub Option13_Click	13	SHOPPING
Distract the cook	Sub Option14_Click	14	KAREN

Enter_Click is called when the user clicks on the Enter button. The routine sends an INFORM message to the destination class. The radio button number (action) is the only meaningful field in the message. The receiving classes use this number to determine what action they have to perform, for example, disable the breadbox (1) or set its inventory to zero (6).

Note: The instance name from the STARTJOB command is used for all classes.

```

Sub Enter_Click ()

  Dim msg100 As msg100          ' message structure

  msg100.number = action        ' radio button number

  DSMsg = "To " & destin & " button " & action
  DS_MLE          ' display message

  MQSEND ByVal vHInst, ByVal destin, ByVal szInstance,
          ByVal "GremlinMessage", 0, msg100, CompCode, Reason
  DisplayCompCode "MQSEND"

End Sub

```

Figure 116. The Gremlin's Push Button Procedure.

Note: Write the MQSEND call in one line!

ProcessPLEvent is called only once, when the *Show* message from Konrad arrives. We query the event to obtain the Gremlin's class and instance names. We use then to change the window header. The state remains MQSTATE_USER for the life of the instance.


```

Sub ProcessPLEvent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure

    MQQRYE ByVal HInst, MQevent, CompCode, Reason ' query event
    DisplayCompCode "MQQYRE"

    If MQevent.InstanceState = MQSTATE_NEW Then ' change caption
        szClass = RTrim(MQevent.Classname)
        szInstance = RTrim(MQevent.LocalInstanceName)
        Gremlin.Caption = szClass & " / " & szInstance
    End If

    MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
    DisplayCompCode "MQENDE"
End Sub

```

Figure 117. The Gremlin's Event Procedure

5.5.4 Project Repair List

The repair list PL is used to check the status of other classes and to reverse the status imposed by the Gremlin. This PL cannot replace spoiled food items, however. To replenish food items the shopping list has to be used.

You can create the project file in the same fashion as for the other projects, by copying the .MAK and .FRM files. Let us copy a .MAK file and change the names as indicated below.

Note: The IconForm is the *Name* property of the form. Since there will also be a push button with the name "Repair" we decided to give the form the name "Repairlist" and the button the name "Repair".

```

REPAIR.FRM                                <=== changed
C:\WINDOWS\SYSTEM\BMQNTFY.VBX
C:\3TIERW\VBSUPP\BMQB.BAS
C:\3TIERW\VBSUPP\BMQVBX.BAS
C:\3TIERW\VBSUPP\CMQB.BAS
BLTDEF.BAS
ProjWinSize=105,459,252,204
ProjWinShow=2
Command="REPAIR"                          <=== changed
IconForm="Repairlist"                     <=== changed !!!
Title="REPAIR"                             <=== changed
ExeName="REPAIR.EXE"                      <=== changed
Path="C:"

```

Figure 118. The Repair List's Project File

5.5.4.1 Create the GUI for the Repair List

You may also copy a .FRM file and delete all unwanted controls. If you do not do that and you open the project, Visual Basic displays an error message stating that it cannot find the file REPAIR.FRM. However, you can easily create it by selecting *New form* from the File menu.

In this window we will not display messages in a text box but in a list box.

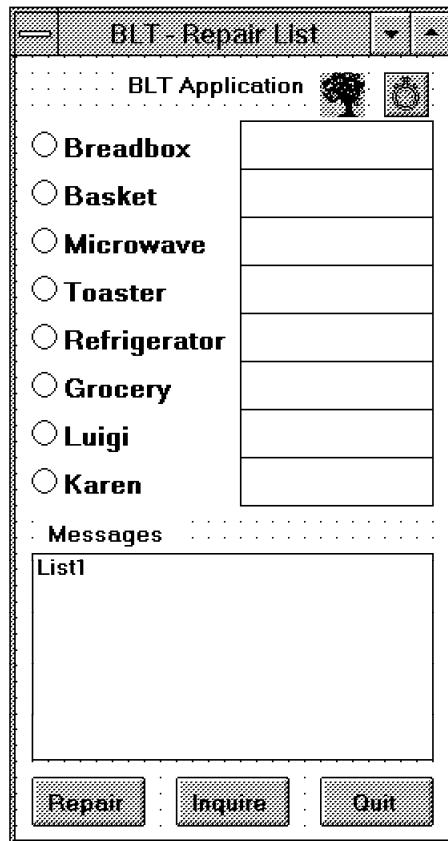


Figure 119. The Repair List's Frame at Design Time

If you decide to create the frame from scratch follow these steps:

1. Size the frame and change the caption as shown in Figure 119.
2. Create the label with the caption "BLT Application".
3. Move the stopwatch and oaktree controls into the frame.
4. Create the first radio button and change the following properties:
 - Caption: Breadbox
 - Font size: 9.6
 - Height: 372 (use the mouse)
5. Create the other radio buttons and change their captions. Use Ctrl+C and Ctrl+V to copy the control and its property.
6. Create the first text box at the right of the radio button and change the following properties:
 - Text: blank
 - Font size: 9.6
 - Height: 372 (use the mouse)
 - Tabstop: False
 - Enable: False
7. Create the other text boxes. Use Ctrl+C and Ctrl+V to copy the control and its property.

8. Create the label with the caption "Messages".
9. Create a list box to be used to display messages. Set the Tabstop property to "False".
10. Create three command buttons on the bottom of the screen and change their captions as shown in Figure 119 on page 160.

A vertical scrollbar appears automatically when the list contains more items (lines) than it can display.

5.5.4.2 Write the Basic Code for the Repair List

The following routines you can "copy and paste" from other projects.

1. Declarations (Figure 120)
2. Close_Click (Figure 33 on page 55)
3. DisplayCompCode (Figure 30 on page 54)
4. Form_Load (Figure 99 on page 146)
5. Form_Unload (Figure 100 on page 146)
6. OAK1_NewEvent (Figure 107 on page 149)
7. Quit_Click (Figure 101 on page 146)

Modify the declarations so that it includes all fields shown in Figure 120.

```
Option Explicit
Dim CompCode As Long
Dim Reason As Long
Dim NL As String ' new line characters
Dim DSMsg As String ' buffer for message to be displayed
Dim ij As Long ' work field
Dim szClass As String ' name of the class
Dim szInstance As String ' name of the instance
Dim action As Integer ' action (ratio button number)
Dim save_action As Integer ' save action
Dim destIn As String ' destination class
Dim SenderC As String ' class that sent the message (return address)
Dim field As Integer ' work field
```

Figure 120. The Repair List's Declarations

The following programs are new or different:

1. DS_MSG
2. OptionX_Click (1 through 8)
3. Inquire_Click
4. Repair_Click
5. ProcessPLEvent

DS_MSG displays messages in the list box "List1". In the previous projects we used DS_MLE to display messages in a text box. We allow up to 30 entries in the list. If the list contains more than that the first entry is removed. The calling program prepares the message and stores it in the buffer DSMsg.

Note: The scrollbar appears automatically when the list contains more entries than can be shown in the window.

```

Sub DS_MSG ()

    If List1.ListCount > 30 Then ' if more than 30 items
        List1.RemoveItem 0      ' delete the first
    End If
    List1.AddItem DSMsg        ' add to the end

End Sub

```

Figure 121. BLT: Display Messages in a List Box

Option1_Click through **Option8_Click** are called when the user clicks on one of the eight radio buttons. At this time we remember what button was clicked and to what class the inquiry or repair message has to be sent to. Table 24 shows the values for action and destin.

```

Sub Option1_Click ()

    action = 1
    destin = "BREADBOX"
    Text1.Text = ""

End Sub

```

Figure 122. Repair List: Radio Button Procedure

Table 24. Repair List: Actions and Destinations for Radio Buttons

Button	Procedure	Action	Destination
Breadbox	Sub Option1_Click	1	BREADBOX
Basket	Sub Option2_Click	2	BASKET
Microwave	Sub Option3_Click	3	MICRO
Toaster	Sub Option4_Click	4	TOASTER
Refrigerator	Sub Option5_Click	5	FRIDGE
Grocery	Sub Option6_Click	6	GROCER
Luigi	Sub Option7_Click	7	LUIGI
Karen	Sub Option8_Click	8	KAREN

Inquiry_Click is called when the user wants to inquire about the status of the class selected with the radio button. The following notes refer to the program shown in Figure 123 on page 163.

1 Before the message is sent we display the text "Inquiring..." in the text box next to the selected radio button. This shows the user that a message is on its way and we are waiting for a reply from the class. The field "action" contains the radio button number. It is put in there when a radio button is selected.

2 The *InquiryRequest* message is sent to the destination. The message itself does not contain any data. The receiving program knows what to do when this message arrives.

3 We could disabled the push button to avoid that a second inquiry is sent to the same destination. However, we can find out from the return code of the second MQSEND if there has been a request message sent and no reply arrived yet. If we are still waiting for a reply, we display a message in the list box.

4 After the request message has been sent we set the timer. When the timer expires and no reply has arrived the timeout rule, RepairRule2, is satisfied. The event procedure in Figure 125 on page 164 shows how this event is handled.

```

Sub Inquire_Click ()
  Dim msg100 As msg100
  1
  Select Case action          ' radio button number
    Case 1
      Text1.Text = "Inquiring..."
    Case 2
      Text2.Text = "Inquiring..."
    Case 3
      Text3.Text = "Inquiring..."
    Case 4
      Text4.Text = "Inquiring..."
    Case 5
      Text5.Text = "Inquiring..."
    Case 6
      Text6.Text = "Inquiring..."
    Case 7
      Text7.Text = "Inquiring..."
    Case 8
      Text8.Text = "Inquiring..."
    Case Else
      DSMsg = "No button selected "
      DS_MSG
      Exit Sub
  End Select
  2
  MQSEND ByVal vHInst, ByVal destin, ByVal szInstance,
          ByVal "InquiryRequest", 0, msg100, CompCode, Reason
  3
  If Reason = MQRC_REPLY_ALREADY_EXPECTED Then
    DSMsg = "Wait up to 30 sec for reply..."
    DS_MSG
    Exit Sub
  Else
    DisplayCompCode "MQSEND"
  End If
  4
  MQTIME ByVal vHInst, ByVal "RepairRule2", 30, CompCode, Reason
  DisplayCompCode "MQTIME"
  save_action = action          ' save value
End Sub

```

Figure 123. Repair List: Send Inquiry Request.

Note: Write the MQSEND instruction in one line!

```

Sub Repair_Click ()
  Dim msg100 As msg100

  DSMsg = "Get " & destin & " working"
  DS_MSG

  MQSEND ByVal vHInst, ByVal destin, ByVal szInstance,
          ByVal "RepairMessage", 0, msg100, CompCode, Reason
  DisplayCompCode "MQSEND"
End Sub

```

Figure 124. Repair List: Send Repair Message.

Note: Write the MQSEND instruction in one line!

Repair_Click is called when the user wants to send the *RepairMessage* to the class indicated by the radio button. The field "destin" contains the destination's class name. It is put in there by a radio button procedure, such as shown in Figure 122 on page 162. The routine displays a message in the list box.

ProcessPLEvent processes the two messages the PL can receive, namely *Show* and *InquiryReply*.

```

Sub ProcessPLEvent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure
    Dim MQmp As MQmp                 ' message parameters
    Dim BufferLen As Long             ' buffer length
    Dim msg100 As msg100             ' buffer
    Dim wkfld As Long                 ' value to set state to
    1 MQQRYE ByVal HInst, MQevent, CompCode, Reason
    DisplayCompCode "MQQRYE"
    2 If MQevent.RuleId = RI_SHOW Then ' the form displays
        szClass = RTrim(MQevent.ClassName)
        szInstance = RTrim(MQevent.LocalInstanceName)
        Repairlist.Caption = szClass & " / " & szInstance
        MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
        DisplayCompCode "MQENDE"
        Exit Sub
    End If
    3 If MQevent.RuleId = RI_REPAIR_NO Then ' timeout, no message arrives
        field = save_action ' field number in GUI
        DSMsg = "No response"
    4 Else ' a message arrived
        BufferLen = MQevent.MaxBufferLength
        MQQRYM ByVal HInst, ByVal 1, MQmp, BufferLen, msg100, CompCode, Reason
        DisplayCompCode "MQQRYM"
        SenderC = RTrim(MQmp.ClassName) ' class that sent message
        If SenderC = "BREADBOX" Then field = 1 ' field number in GUI
        If SenderC = "BASKET" Then field = 2
        If SenderC = "MICRO" Then field = 3
        If SenderC = "TOASTER" Then field = 4
        If SenderC = "FRIDGE" Then field = 5
        If SenderC = "GROCER" Then field = 6
        If SenderC = "LUIGI" Then field = 7
        If SenderC = "KAREN" Then field = 8
    End If

```

Figure 125. Repair List: Process PL Events (Part 1)

- 1 MQQRYE stores the properties of the current event in MQevent.
- 2 We change the window header, set the state to MQSTATE_USER and exit.
- 3 In case of a time out there is no message, however, a rule is satisfied. The saved radio button number tells us to what class the last message was sent.
- 4 When the reply arrives we query the message to obtain the message properties in MQmp. We find out what class sent the message, and store the corresponding radio button number in a field.
- 5 If the reply has arrived in time we display the state of the sending class in the text field next to the radio button. The state is in the message in the field "number". Instead of the number we display some text.

```

5
If MQevent.RuleId = RI_REPAIR_INQ Then ' reply to inquiry in time
  If msg100.number = MQSTATE_DISABLED Or
    msg100.number = MQSTATE_DISABLED_WHILE_BUSY Then
    If field = 1 Then DSMsg = "not usable" ' breadbox
    If field = 2 Then DSMsg = "hidden" ' basket
    If field = 3 Then DSMsg = "broken" ' microwave
    If field = 4 Then DSMsg = "unplugged" ' toaster
    If field = 5 Then DSMsg = "door jammed" ' refrigerator
    If field = 6 Then DSMsg = "closed" ' grocery
    If field = 7 Then DSMsg = "on a break" ' Luigi
    If field = 8 Then DSMsg = "on the phone" ' Karen
  ElseIf MQevent.InstanceState = MQSTATE_BUSY Then
    If field = 1 Then DSMsg = "empty"
    If field = 2 Then DSMsg = "empty"
    If field = 3 Then DSMsg = "unplugged"
    If field = 4 Then DSMsg = "unplugged"
    If field = 5 Then DSMsg = "cleaned out"
    If field = 6 Then DSMsg = "closed"
    If field = 7 Then DSMsg = "cooking"
    If field = 8 Then DSMsg = "working"
  Else ' class is working
    DSMsg = "OK"
  End If
6
ElseIf MQevent.RuleId = RI_REPAIR_LATE Then ' reply to inquiry is late
  DSMsg = "responded late"
End If
7
Select Case field
  Case 1
    Text1.Text = DSMsg
  Case 2
    Text2.Text = DSMsg
  Case 3
    Text3.Text = DSMsg
  Case 4
    Text4.Text = DSMsg
  Case 5
    Text5.Text = DSMsg
  Case 6
    Text6.Text = DSMsg
  Case 7
    Text7.Text = DSMsg
  Case 8
    Text8.Text = DSMsg
End Select
8
DSMsg = destin & ": " & msg100.message
DS_MSG
MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
DisplayCompCode "MQENDE"
End Sub

```

Figure 126. Repair List: Process PL Events (Part 2)

- 6** If the reply is late, we are not interested any more.
- 7** These instructions display a text representing the status in the appropriate text box next to the radio button.
- 8** The state is also logged in the list box.

5.5.5 Project Shopping List

The shopping list is used to control the ordering process. It is notified when a food item is depleted. The text "order" appears in the appropriate text box in the GUI. You can initiate two functions:

- Inquire how many tomatoes, bread, etc. are currently available.
- Type into one or more fields a quantity and send the list to the grocer.

To create the shopping list project we copy the .MAK and .FRM files from the repair list. In those file we change all occurrences of "repair" to "shopping". The form itself we change in Visual Basic.

5.5.5.1 Create the GUI for the Shopping List

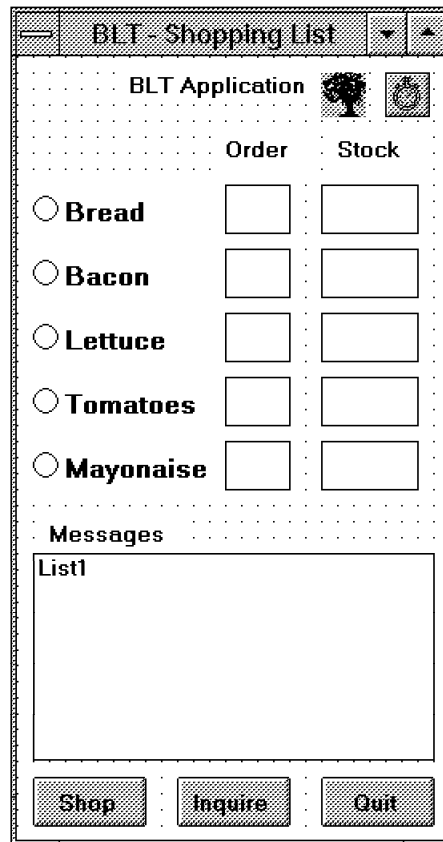


Figure 127. Shopping List's Frame at Design Time

To change the repair list GUI into the shopping list GUI follow these steps:

- Delete all but five radio buttons and change the caption as shown.
- Keep five of the text boxes but make them smaller.
- Add a second column of text boxes.
- Move the controls further down to make room for two labels above the text boxes.
- Add the two labels "Order" and "Stock".
- Change caption and name of the left push button to "Shop".
- Add the "Inquire" button between the two buttons already there.

Make sure that the TabIndex properties are set as you desire. Set the TabStop properties for the text boxes in the right column (stock) to "False".

The GUI contains two types of text boxes:

- Into the five *Order* fields the user types the quantity he wants to order. The data shall be right-justified and the input limited to two digits.
- The five *Stock* are used to display the number of food items in stock, as the result of an inquiry. This is a read-only field.
- Set the MultiLine property to True.
- Set the Alignment property to 1-Right Justify.
- Set the FontSize property to 9.6.
- Set the MaxLength property to 2 (order only).
- Set the Enabled property to False (stock only).

You may run the program from inside Visual Basic to test tabbing and keyboard input.

5.5.5.2 Write the Basic Code for the Shopping List

Since SHOPPING.FRM is a copy of REPAIR.FRM, you should delete the procedures Option6_Click, Option7_Click, Option8_Click, and Repair_Click. You find them in the object "general".

To the declarations (shown in Figure 120 on page 161) add one line:

```
Dim msgtxt as String      ' text to display in order column
```

The following routines remain unchanged:

1. DS_MSG (Figure 121 on page 162)
2. Close_Click (Figure 33 on page 55)
3. DisplayCompCode (Figure 30 on page 54)
4. Form_Load (Figure 99 on page 146)
5. Form_Unload (Figure 100 on page 146)
6. OAK1_NewEvent (Figure 107 on page 149)
7. Quit_Click (Figure 101 on page 146)

We have to modify or write routines that are invoked when:

- The user enters a quantity.
- A radio button is selected.
- The Inquire button is clicked.
- The Shop button is clicked.
- A message from another class arrives.

These routines are described below.

Text1_Change through Text5_Change get invoked when the user types into one of the text boxes to order a quantity. The number of digits he can enter is limited to 2. That is specified in the MacLength property for the text box. The only instruction blanks out the field when a not-numeric character was entered.

```
Sub Text1_Change ()
    If Not IsNumeric(Text1.Text) Then Text1.Text = ""
End Sub
```

Figure 128. Shopping List: Type a Quantity

Option1_Click through Option5_Click are invoked when the user selects a radio button. We store the number of the radio buttons (0 through 4) and the destination class name the inquiry message is sent to (providing the user selects the Inquire push button).

```
Sub Option1_Click ()
    action = 0           ' radio button number
    destin = "BREADBOX" ' destination class
    msgtxt = "bread"    ' used in other procedures
End Sub
```

Figure 129. Shopping List: Radio Button Procedure

The value stored in "action" is also the product ID, for example, 0 means bread and 4 means mayonaise.

Table 25. Shopping List: Actions and Destinations for Radio Buttons

Button	Procedure	Action	Destination
Bread	Sub Option1_Click	0	BREADBOX
Bacon	Sub Option2_Click	1	FRIDGE
Lettuce	Sub Option3_Click	2	FRIDGE
Tomatoes	Sub Option4_Click	3	BASKET
Mayonaise	Sub Option5_Click	4	FRIDGE

Inquire_Click sends a *FoodInquiry* message to either the basket, breadbox, or refrigerator. This message is an INFORM message. Contrary to REQUEST messages, such messages cannot be timed. This routine explains how the timer can be used to check if a response to an INFORM message arrived.

The following comments refer to Figure 130 on page 169.

- 1** The two fields are updated when a radio button is selected. The value stored in msg100.number tells the receiving class what product the inquiry is for.
- 2** This message is displayed in the list box. The radio button numbers are displayed as 1 through 5, not 0 through 4.
- 3** The INFORM message is sent to the destination.
- 4** We set the timer for 20 seconds. When the timer expires the TimerRule is satisfied and the PLM sends an event message to the PL. This message is processed in the ProcessPLevent procedure.

```

Sub Inquire_Click ()
  Dim msg100 As msg100          ' message buffer
  1
  msg100.number = action        ' radio button number (0-9)
  msg100.message = msgtxt      ' product name
  2
  DSMsg = "Inquiry to " & destin & " button " & action + 1
  DS_MLE
  3
  MQSEND ByVal vHInst, ByVal destin, ByVal szInstance,
          ByVal "FoodInquiry", ByVal 0, msg100, CompCode, Reason
  DisplayCompCode "MQSEND"
  4
  MQTIME ByVal vHInst, ByVal "TimerRule", 20, CompCode, Reason
  5
  If Reason = MQRC_TIMER_ALREADY_SET Then
    DSMsg = "Timer already set..."
    DS_MLE
  Else
    DisplayCompCode "MQTIME"
  End If
  6
  Inquire.Enabled = False
End Sub

```

Figure 130. Shopping List: Send Inquiry Message.

Note: Write the MQSEND instruction in one line!

5 You cannot reset the timer before it is expired. We display a message to inform the user that the inquiry he just sent will not be timed. This happens when all of these conditions are met:

- The response to the previous inquiry has arrived and enabled the Inquire button.
- The timer set for this inquiry did not expire yet.
- The user clicks the Inquire button.

6 This disables the push button until either a response to the inquiry arrives or the timer expires.

Shop_Click sends a *variable length* message to the grocer. The message can contain one to five fields, each field containing a quantity to order. A variable message is called a *set* that contains *elements*. We use API calls to the Self-defining Data Manager (SDDM) to work with sets and elements. A set can contain three types of elements:

- Integers
- Character strings
- The contents of a buffer

In this example, we work with integers.

An element consists of a header and data. The data length must be a multiple of four bytes. The header contains these fields:

- Element type
- Element ID
- Data length (for strings)

```

Sub Shop_Click ()
  Dim hSet As Long           ' variable length message
  Dim ik As Long            ' quantity
  Dim ij As Long            ' ID for item and loop counter
  Dim setlength As Long     ' set length
  Dim elements As Long      ' elements in set

  DSMsg = "Order from grocer"
  DS_MLE
  1 MQCRTS hSet, ByVal MQSL_DEF_SET_LENGTH, CompCode, Reason
  DisplayCompCode "MQCRTS"
  2 For ij = 0 To 4           ' check 5 entry fields
    ik = 0
    If ij = 0 And Text1.Text <> "" Then ik = Cdbl(Text1.Text)
    If ij = 1 And Text2.Text <> "" Then ik = Cdbl(Text2.Text)
    If ij = 2 And Text3.Text <> "" Then ik = Cdbl(Text3.Text)
    If ij = 3 And Text4.Text <> "" Then ik = Cdbl(Text4.Text)
    If ij = 4 And Text5.Text <> "" Then ik = Cdbl(Text5.Text)
    If ik > 0 Then          ' a quantity has been entered
      3 MQADDI ByVal hSet, ByVal ij, ByVal ik, ByVal MQRPLC_YES, CompCode, Reason
      DisplayCompCode "MQADDI"
    End If
  Next ij
  4 setlength = 0
  MQQRYS ByVal hSet, elements, setlength, 0, CompCode, Reason
  DisplayCompCode "MQQRYS"
  DSMsg = "Set length=" & setlength & " elements=" & elements
  DS_MLE
  5 If elements > 0 Then
    MQSEND ByVal vHInst, ByVal "GROCER", ByVal szInstance,
            ByVal "FoodOrder", ByVal 0, ByVal hSet, CompCode, Reason
    DisplayCompCode "MQSEND"
    DSMsg = elements & " items ordered"
  Else
    DSMsg = "No items ordered"
  End If
  DS_MLE
  6 MQDELS ByVal hSet, CompCode, Reason
  DisplayCompCode "MQDELS"
End Sub

```

Figure 131. Shopping List: Send an Order to the Grocer.

Note: Write the MQSEND instruction in one line!

In this example, we use the element ID to identify the five products:

0 = bread	2 = lettuce	4 = mayonaise
1 = bacon	3 = tomatoes	

1 MQCRTS creates a set with the default length. The length of the set is automatically adjusted when we insert (or remove) elements.

2 We check all input fields in the form. If they contain a number between 1 and 99 (the input is limited to two digits), we add an element to the set. Since the input is in character format the number has to be converted to an integer.

3 MQADDI add an integer element to the set. The element ID is 0 through 4 (in ij). The element data is in ik. MQRPLC_YES tells the SDDM to replace an element with the same ID.

4 MQQRYS returns the length and the number of elements in the set.

5 If the user did not enter any quantity in the GUI, the set is not sent to the grocer. The set must contain at least one element.

6 We delete the set before we exit the routine.

ProcessPLEvents is called when the PL receives a message from another class or when a timer expires.

```
Sub ProcessPLEvent (ByVal HInst As Long)
    Dim MQevent As MQevent           ' event structure
    Dim MQmp As MQmp                 ' message parameters
    Dim BufferLen As Long             ' buffer length
    Dim msg100 As msg100             ' buffer
    Dim wkfld As Long                ' value to set state to

    MQQRYS ByVal HInst, MQevent, CompCode, Reason
    DisplayCompCode "MQQRYS"

1
    If MQevent.RuleId = RI_SHOW Then      ' form displays
        szClass = RTrim(MQevent.ClassName)
        szInstance = RTrim(MQevent.LocalInstanceName)
        Shoppinglist.Caption = szClass & " / " & szInstance
        MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
        DisplayCompCode "MQENDE"
        Exit Sub
    End If

2
    If MQevent.RuleId = RI_GREMLIN Then    ' erase shopping list
        Text6.Text = ""
        Text7.Text = ""
        Text8.Text = ""
        Text9.Text = ""
        Text10.Text = ""
        DSMsg = "List erased..."
        DS_MLE
        MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
        DisplayCompCode "MQENDE"
        Exit Sub
    End If
```

Figure 132. Shopping List: Process PL Events (Part 1)

1 The *Show*, sent by Konrad, causes the GUI to be displayed.

2 The *GremlinMessage* causes the fields in the GUI to be erased.

3 MQQRYS retrieves the actual message sent by the other class. If the retrieve does not work we exit the routine.

4 The *OrderMessage* tells the program what food item is depleted.

5 The *FoodInquiry* message is the response to an inquiry sent out earlier.

6 The program is notified when the timer expires.

```

3
If MQevent.RuleId = RI_ORDER Or MQevent.RuleId = RI_FOOD_INQ Then

    BufferLen = MQevent.MaxBufferLength
    MQQRYM ByVal HInst, ByVal 1, MQmp, BufferLen, msg100, CompCode, Reason

    DisplayCompCode "MQQRYM"
    If Reason <> 0 Then
        MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
        DisplayCompCode "MQENDE"
    Exit Sub
    End If
End If

4
If MQevent.RuleId = RI_ORDER Then          ' order arrives
    DSMsg = "Order " & msg100.message
    DS_MLE
    Select Case msg100.number
        Case 0
            Text6.Text = "order"
            Text1.Text = ""
        Case 1
            Text7.Text = "order"
            Text1.Text = ""
        Case 2
            Text8.Text = "order"
            Text1.Text = ""
        Case 3
            Text9.Text = "order"
            Text1.Text = ""
        Case 4
            Text10.Text = "order"
            Text1.Text = ""
    End Select
End If

5
If MQevent.RuleId = RI_FOOD_INQ Then      ' response to inquiry
    DSMsg = "Response from " & RTrim(MQmp.ClassName)
    DS_MLE
    Inquire.Enabled = True
    Select Case msg100.number
        Case 0
            Text6.Text = msg100.value
        Case 1
            Text7.Text = msg100.value
        Case 2
            Text8.Text = msg100.value
        Case 3
            Text9.Text = msg100.value
        Case 4
            Text10.Text = msg100.value
    End Select
End If

6
If MQevent.RuleId = RI_TIMER Then
    DSMsg = "timer expired"
    DS_MLE
    Inquire.Enabled = True
End If

MQENDE ByVal HInst, ByVal MQSTATE_USER, CompCode, Reason
DisplayCompCode "MQENDE"
End Sub

```

Figure 133. Shopping List: Process PL Events (Part 2)

5.6 Building the Business Logic

We write the methods for the business logic in C language on an AIX machine.

The application uses seven BLs:



Karen

Refrigerator

Basket

Breadbox

Toaster

Microwave

Grocer

We already created the class source files as part of the 3T design, see 5.3, “3T Design” on page 121. MQ3T helps us in writing a business logic by creating skeleton files for all languages it supports. As a result, the programmer has to write only those programs named as *SourceName* in the methods.

5.6.1 Creating Skeleton Files

For each class definition file, invoke the class compiler with the */s* (skeleton) option, for example:

```
bmqcc /s karen.cs
```

The class compiler creates three files for Karen, that are:

karen.c The “C” skeleton file that contains INCLUDE statements for all methods.

karen.mak A make file to compile the business logic.

karen.exp An export file used to create the DLL.

Karen’s class source file, *karen.cs*, is listed in A.8, “Class Source File for KAREN” on page 232. The following figures show the files created for the BL Karen.

```
*****/
*
*   karen.exp: Source file generated by the Class Compiler   */
*           03/11/96    12:07:28 language = C              */
*
*****/
#! karen.a
LibMain
Sandwich
Gremlin
Repair
Ignore
Inquiry
MakeBLT
ServeBLT
NoBLT
ClearUp
```

Figure 134. Karen’s Export File.

Note: This file lists all methods used by the BL. The skeleton file *karen.c* contains nine INCLUDE statements, one for each routine.

Since Karen is the most versatile BL we use her skeleton file to explain some of the functions of the class compiler.

1 The name of the header file comes from the message definitions, see A.1, “Messages for The BLT Example” on page 213.

2 This is the entry point for the *FeedMe* message from Konrad. The name is obtained from the method description in the class file:

```
METHOD
  BEGIN          // order arrived
    MethodName   Sandwich
    MethodType   C_LIBRARY
    ProgName     karen.Sandwich
    SourceName   bltorder
    MsgOut       Starve, GetTomato, GetBread, GetFromFridge
  END
```

The parameters for MQENTRY are:

1. The name of Karen’s instance

The instance name is specified in Konrad’s MQSEND call.

```
MQSEND ByVal vHInst, ByVal "KAREN", ByVal szInstance, ByVal FeedMe, .
```

2. The ID of the rule that is satisfied

The rule is either RI_SANDWICH1 or RI_SANDWICH2, depending on the rule the instance is in. The two rules are specified as follows:

```
RULE
  BEGIN          // first BLT request
    RuleId       RI_SANDWICH1
    RuleName     SandwichRule1
    MethodName   Sandwich
    State        MATCHSTATE MQSTATE_NEW
    MsgIn        FeedMe
  END

RULE
  BEGIN          // next BLT request
    RuleId       RI_SANDWICH2
    RuleName     SandwichRule2
    MethodName   Sandwich
    State        MATCHSTATE MQSTATE_CLEAR
    MsgIn        FeedMe
  END
```

3. The state Karen’s instance is in when the message arrives

According to the rules the state can be one of:

- MQSTATE_NEW
- MQSTATE_CLEAR

4. A flag that indicates if a message is present or not

- MQMDF_MSG_IS_PRESENT
- MQMDF_MSG_NOT_PRESENT
- MQMDF_MSG_NOT_PRESENT_TIMEOUT

In Karen’s case a message is always present.

5. The pointer to the message

Since Karen uses a PUSH interface, 3T provides the message data in a buffer for the programmer ready to use. Data can be accessed with instructions such as

```
ij = pFeedMe->number;
```

3 We have to write the routine bltorder.c to process the *FeedMe* message.

Note: Most of the entry points look like the one described above, only the message name is different.


```

/*****
/*
/*      karen.c: Source file generated by the Class Compiler      */
/*      02-27-1996  12:02:06a  language = C                    */
/*                                                              */
/*****
#include <bmqc.h>
#include "bltstruc.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void MQENTRY Sandwich( MQHINST HInst,
                      MQLONG RuleId,
                      PMQLONG pState,
                      MQLONG fFeedMe,
                      MSG100 *pFeedMe
                      )
{
    #include "bltorder.c"

void MQENTRY Gremlin( MQHINST HInst,
                    MQLONG RuleId,
                    PMQLONG pState,
                    MQLONG fGremlinMessage,
                    MSG100 *pGremlinMessage
                    )
{
    #include "xgremlin.c"

void MQENTRY Repair( MQHINST HInst,
                   MQLONG RuleId,
                   PMQLONG pState,
                   MQLONG fRepairMessage,
                   MSG100 *pRepairMessage
                   )
{
    #include "xrepair.c"

void MQENTRY Ignore( MQHINST HInst,
                   MQLONG RuleId,
                   PMQLONG pState,
                   MQLONG fMessage1,
                   MSG100 *pMessage1
                   )
{
    #include "xignore.c"

void MQENTRY Inquiry( MQHINST HInst,
                    MQLONG RuleId,
                    PMQLONG pState,
                    MQLONG fInquiryRequest,
                    MSG100 *pInquiryRequest
                    )
{
    #include "xinquiry.c"
}

```

Figure 135. Karen's C Skeleton File (Part 1)

```

void MQENTRY MakeBLT( MQHINST HInst,      5
                    MQLONG  RuleId,
                    PMQLONG pState,
                    MQLONG  fHaveTomato,
                    MSG100 *pHaveTomato,
                    MQLONG  fHaveBread,
                    MSG100 *pHaveBread,
                    MQLONG  fHaveFromFridge,
                    MSG100 *pHaveFromFridge
                    )
{
    #include "bltmake.c"
}

void MQENTRY ServeBLT( MQHINST HInst,     6
                     MQLONG  RuleId,
                     PMQLONG pState,
                     MQLONG  fHaveToast,
                     MSG100 *pHaveToast,
                     MQLONG  fHaveBacon,
                     MSG100 *pHaveBacon
                     )
{
    #include "bltserve.c"
}

void MQENTRY NoBLT( MQHINST HInst,       7
                  MQLONG  RuleId,
                  PMQLONG pState,
                  MQLONG  fHaveToast,
                  MSG100 *pHaveToast,
                  MQLONG  fHaveBacon,
                  MSG100 *pHaveBacon
                  )
{
    #include "bltnone.c"
}

void MQENTRY ClearUp( MQHINST HInst,
                    MQLONG  RuleId,
                    PMQLONG pState,
                    MQLONG  fMQ_SYSTEM_OWNER_ENDED,
                    MQPTR   pMQ_SYSTEM_OWNER_ENDED
                    )
{
    #include "xclear.c"
}

```

Figure 136. Karen's C Skeleton File (Part 2)

4 This entry point is different since it does not contain flags and name of a specific message, rather of any message. The method is called when one of several rules is satisfied. The programmer has to issue an API call to inquire what message is present, if he cares. In Karen's case we issue an MQQRYM (query message) call to obtain the name of the message and then display it.

5 This entry point is generated for a method that expects three messages.

```
METHOD
  BEGIN          // material arrives (all or partially)
    MethodName  MakeBLT
    MethodType  C_LIBRARY
    ProgName    karen.MakeBLT
    SourceName  bltmake
    MsgOut      MakeToast, CookBacon, Starve
  END
```

For the three messages two rules apply:

1. All messages are present (left).
2. None or some of the messages are present and a timer expired (right).

```
RULE                                RULE
BEGIN // on-time arrival           BEGIN // timeout
  RuleId    RI_MAKEBLT1           RuleId     RI_MAKEBLT2
  RuleName  MakeRule1             RuleName   MakeRule2
  MethodName MakeBLT              MethodName MakeBLT
  MsgIn     HaveTomato,           Timed      Yes
            HaveBread,           MsgIn     HaveTomato    PLACEHOLDER,
            HaveFromFridge       HaveBread  PLACEHOLDER,
  END                                             HaveFromFridge PLACEHOLDER
                                END
```

When all messages are present the rule on the left is satisfied.

The rule to the right is *timed*. It is satisfied when the timer expires, regardless if a message has arrived or not. The timer is set with an MQTIME API call, such as:

```
MQTIME (HInst, "MakeRule2", 15, &CompCode, &Reason);
```

Note: Since the same program is invoked when all messages arrive in time *and* when the timer expires, the programmer has to write code (in bltmake.c) to find out what rule was satisfied. How to avoid that is demonstrated with the next methods.

If we want to process the messages that arrive after the timer has expired, we must specify an additional rule for *each* message, such as:

```
RULE
  BEGIN // tomato arrives late
    RuleId    RI_TOMATO
    RuleName  TomatoRule
    MethodName Ignore
    MsgIn     HaveTomato LATE
  END
```

In Karen's case we ignore late messages, however, for demonstration purposes we display a message, in the routine xignore, saying that a specific message arrived late.

6 + 7 Both methods expect two messages:

- ServeBLT is called when all messages arrive on time.
- NoBLT is called when the timer expires and no or only one message is present.

In this case the programmer does not have to check the rule, as in bltmake.c, to find out why to method has been called.

```

*****
#*                                                                    *
#*   karen.mak: Source file generated by the Class Compiler          *
#*           03/11/96      12:07:28 language = C                    *
#*                                                                    *
#*****
.SUFFIXES:
.SUFFIXES: .o .c

CC = xlc_r
CFLAGS = -g -c -Dsigned= -Dvolatile= -D_Optlink -I. -M
LFLAGS = -L. -lXm -lXt -lX11 -L/usr/lpp/mq3t/lib -lbnmqpic -e LibMain -bM:SRE

HEADERS = bltstruc.h

.c.o:
    $(CC) $(CFLAGS) $<

all: karen

karen.o: karen.c\
    bltorder.c\
    xgremlin.c\
    xrepair.c\
    xignore.c\
    xinquiry.c\
    bltmake.c\
    bltserve.c\
    bltnone.c\
    xclear.c\
    $(HEADERS)

libmain.o: libmain.c

karen: libmain.o karen.o karen.exp
    $(CC) $(LFLAGS) -bE:karen.exp -bmap:karen.map libmain.o karen.o
    mv a.out karen

```

Figure 137. Karen's Make File

Notes:

1. The class compiler includes the header file "bltstruc.h". It finds this name in the message definitions.
2. The linker uses libraries in mq3t/lib.
3. To create a DLL for a class type a command, such as:
 make -f karen.c

Use the class compiler to create similar skeleton files for all seven BL classes.

5.6.2 Creating The Business Logic

Table 20 on page 133 contains the names of all C procedures we need to write to make the business logic complete. Some routines are specifically for one class, others are common. A brief discussion follows below.

5.6.2.1 Karen's Business Logic

For the BLT production process we write the following programs:

- `bltorder.c` in Figure 138 on page 180
- `bltmake.c` in Figure 139 on page 181 and Figure 140 on page 182
- `bltserve.c` in Figure 141 on page 183
- `bltnone.c` in Figure 142 on page 183

Karen includes other routines, however, they are common.

bltorder processes the *FeedMe* message from Konrad. With a wave of three messages Karen asks for material to build the BLT. She waits up to 50 seconds for the material to arrive. When the timer expires Karen knows that she will not be able to produce the BLT in time for Konrad to eat it.

1 MQQRYM (MQ QueRY Message) obtains the message properties and, since the buffer length is 0, the length of the message. From the properties structure we will extract the names of the class and instance that sent the message.

2 When the API call was not successful we display an error message in the BLM's window and exit the routine. We use this approach for all API calls in all routines.

3 Class and instance names in the MQmp structure are padded with blanks. This routine extracts the significant characters.

4 The three MQSEND calls send request messages to the basket, breadbox, and refrigerator to request material.

5 Karen waits up to 50 seconds for the arrival of the three replies. If the timer expires and any reply is missing *MakeRule2* is satisfied.

6 After the request messages are sent Karen is busy; she waits for the replies before she can continue to make the BLT. We set the instance state to MQSTATE_BUSY to prevent her from accepting more requests. This is controlled by the BLM.

7 In this demonstration program we display messages to inform the user of the state the instance is in. This helps the user to understand how the 3T infrastructure works.

```

#include "bltdef.h"
MQMP      MQmp;
MQLONG    BufferLength, CompCode, Reason;
MSG100    msg100;
int        length;
char      *loc;
char      SenderC   [50];
char      SenderI   [100];
/*****
/*          receive request to make BLT
/*****
BufferLength = 0;
1    MQRYM (HInst, 1,
        &MQmp,
        &BufferLength, 0,
        &CompCode, &Reason );
2    if (CompCode != MQRC_NONE) {
        printf ("MQRYM: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
    }
/*****
/*          obtain sender's class and instance name (return address)
/*****
3    length = strlen MQmp.ClassName);
        loc   = strchr(MQmp.ClassName, ' ');
        if (loc != NULL) length = loc - MQmp.ClassName;
        strncpy (SenderC, MQmp.ClassName, length);
        SenderC[length] = '\0';
        length = strlen (MQmp.LocalInstanceName);
        loc   = strchr(MQmp.LocalInstanceName, ' ');
        if (loc != NULL) length = loc - MQmp.LocalInstanceName;
        strncpy (SenderI, MQmp.LocalInstanceName, length);
        SenderI[length] = '\0';
/*****
/*          Request material for BLT
/*****
4    MQSEND (HInst, "BASKET", SenderI, "GetTomato",
        0, &msg100, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
        }
        MQSEND (HInst, "BREADBOX", SenderI, "GetBread",
        0, &msg100, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
        }
        MQSEND (HInst, "FRIDGE", SenderI, "GetFromFridge",
        0, &msg100, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
        }
5    MQTIME (HInst, "MakeRule2", 50, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
        printf ("MQTIME CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
        }
6    *pState = MQSTATE_BUSY;
7    printf ("%s %s from %s (%s) State=BUSY\n",
        "BL processed ", MQmp.MsgName,
        SenderC, SenderI);
        printf ("Wait up to 50 seconds for material ....\n");
        return;

```

Figure 138. Karen's Method "bltorder"

```

#include "bltdef.h"
MQMP      MQmp;
MQLONG    BufferLength, CompCode, Reason;
MSG100    msg100;                /* message buffer      */
unsigned int length;
char      *loc;
char      SenderC   [50];        /* sending class
char      SenderI   [100];       /* sending instance
char      output    [120];       /* work/message buffer
char      wkfld     [50];
/*****
/*      Messages arrived, check if material complete      */
/*****
8      strcpy (output,"BL received");
        if (fHaveTomato    == MQMDF_MSG_IS_PRESENT)
            strcat (output," tomato,");
        if (fHaveBread     == MQMDF_MSG_IS_PRESENT)
            strcat (output," bread,");
        if (fHaveFromFridge == MQMDF_MSG_IS_PRESENT)
            strcat (output," bacon, lettuce, mayo");
        if ( fHaveTomato    == MQMDF_MSG_NOT_PRESENT_TIMEOUT ||
            fHaveBread     == MQMDF_MSG_NOT_PRESENT_TIMEOUT ||
            fHaveFromFridge == MQMDF_MSG_NOT_PRESENT_TIMEOUT )
        {
9          *pState = MQSTATE_CLEAR;
            MQRPLY (HInst, "Starve", 0, &msg100, &CompCode, &Reason);
            if (CompCode != MQRC_NONE) {
                printf ("MQRPLY: CC %d reason %d\n", CompCode, Reason);
                *pState = MQSTATE_END;
            }
            printf ("%s - timeout. State=CLEAR\n", output);
            return;
        }
10     else {
            switch (*pState) {
                case MQSTATE_NEW:                strcpy (wkfld,"NEW");
                                                break;
                case MQSTATE_CLEAR:             strcpy (wkfld,"CLEAR");
                                                break;
                case MQSTATE_BUSY:              strcpy (wkfld,"BUSY");
                                                break;
                case MQSTATE_DISABLED:          strcpy (wkfld,"DISABLED");
                                                break;
                case MQSTATE_DISABLED_WHILE_BUSY:
                                                strcpy (wkfld,"DISABLED_WHILE_BUSY");
                                                break;
                case MQSTATE_END:               strcpy (wkfld,"END");
            }
            printf ("%s. State=%s\n", output, wkfld);
        }
/*****
/*      Material complete      */
/*****
        BufferLength = 0;
1      MQRYM (HInst, 1,                /* instance & message number */
            &MQmp,                    /* MQMP structure (output)   */
            &BufferLength, 0,         /* data length (output)     */
            &CompCode, &Reason );    /* return codes              */
        if (CompCode != MQRC_NONE) {
            printf ("MQRYM: CC %d reason %d\n", CompCode, Reason);
            *pState = MQSTATE_END;
            return;
        }

```

Figure 139. Karen's Method "bltmake" (Part 1 of 2)

```

/*****
/*      obtain sender's instance name (return address)      */
/*****
length = strlen(MQmp.LocalInstanceName);    /* sending instance */
3   loc  = strchr(MQmp.LocalInstanceName, ' ');
    if (loc != NULL) length = loc - MQmp.LocalInstanceName;
    strncpy (SenderI, MQmp.LocalInstanceName, length);
    SenderI[length] = '\0';
/*****
/*      Toast bread and cook bacon                          */
/*****
11  MQSEND (HInst, "TOASTER", SenderI, "MakeToast",
           0, &msg100, &CompCode, &Reason);
    if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
    }
    MQSEND (HInst, "MICRO", SenderI, "CookBacon",
           0, &msg100, &CompCode, &Reason);
    if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
    }
12  MQTIME (HInst, "ServeRule2", 50, &CompCode, &Reason);
    if (CompCode != MQRC_NONE) {
        printf ("MQTIME CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
    }
    printf ("BL is cooking, wait for up to 10 seconds...\n");
    return;

```

Figure 140. Karen's Method "bltmake" (Part 2 of 2)

bltmake is called when Karen receives replies from the basket, breadbox, or refrigerator. This routine is called when the replies arrive in time *and* when the timer expires and no or some messages are present. We could write a separate methods for MakeRule1 and MakeRule2.. This approach is demonstrated in the next two figures, bltnone and bltserve.

8 We check the message data flag to find out if a message is present or not. If one of the replies is missing then we know that the timer has expired.

9 Since Karen knows that she cannot build the BLT she sends the *Starve* message to Konrad. The state is set to CLEAR so that she can accept future *FeedMe* messages.

10 We obtain the state the instance is in and display a message in the BLM's window to inform the user about the progress of the BLT production process.

11 Here Karen sends the second wave of messages to the toaster and microwave.

12 Karen waits up to 10 seconds for the replies from the kitchen equipment.


```

#include "bltdef.h"
MSG100  msg100;                               /* message buffer */
MQLONG  CompCode, Reason;

13  *pState = MQSTATE_CLEAR;
14  MQRPLY (HInst, "Sandwich", 0, &msg100, &CompCode, &Reason);

    if (CompCode != MQRC_NONE)
        printf ("MQRPLY: CC %d reason %d\n", CompCode, Reason);
    else
        printf ("BL delivers BLT and ends. State=CLEAR\n");

```

Figure 141. Karen's Method "bltserve"

bltserve is called when the replies from both microwave and toaster arrive.

13 Since the BLT production is completed we set the instance state to CLEAR. This enables Karen to accept more requests.

14 Karen sends the *Sandwich* to Konrad. If Konrad receives it on time or late is not of her concern. Since *FeedMe* is a request message she must respond with a reply.

```

#include "bltdef.h"
MSG100  msg100;                               /* message buffer */
MQLONG  CompCode, Reason;

15  MQRPLY (HInst, "Starve", 0, &msg100, &CompCode, &Reason);
    if (CompCode != MQRC_NONE)
        printf ("MQRPLY: CC %d reason %d\n", CompCode, Reason);
    else
        printf (" BL cannot make BLT, BL ends. State=%d\n", *pState);
    *pState = MQSTATE_CLEAR;

```

Figure 142. Karen's Method "bltnone"

bltnone is called when the timer expired and at least one of the replies from the microwave or toaster are not present.

15 Since the BLT production cannot be completed Karen sends the *Starve* message to Konrad. She must respond with a reply. The state is set to clear to allow for future *FeedMe* messages.

5.6.2.2 Business Logic for Basket, Breadbox and Refrigerator

The specific routines for the three objects that hold the material for the BLT look very similar. The routines are:

basket1.c processes requests from Karen

bbox1.c processes requests from Karen

fridge1.c processes requests from Karen

delivery.c processes deliveries from the grocer

foodinq.c replies to inquiries from the shopping list

Since the program for the refrigerator is more complex than the other (the refrigerator holds three items instead of one) we discuss it in more detail.

```

#include "bltdef.h"
MQMP      MQmp;
MQLONG    BufferLength, CompCode, Reason;
MSG100    msg100;                /* message buffer      */
char      wkfld[50];
char      SenderC   [50];        /* sending class      */
char      SenderI   [100];       /* sending instance   */
unsigned  int length;
char      *loc;
int       ik;
/*****
/*      File names for inventory      */
*****/
char      fnbasket[] = "basket.dat" ;      1
char      fnfridge[] = "fridge.dat";
char      fnbread[]  = "breadbox.dat";
char      fn[20];                /* file name      */
FILE      *fp;
long      inventory[5];
/*****
/*      receive request to deliver tomato      */
*****/
2      BufferLength = 0;
      MQRYM (HInst, 1, &MQmp, &BufferLength, 0, &CompCode, &Reason); /*
      if (CompCode != MQRC_NONE) {
          printf ("BL - MQRYM: CC %d reason %d\n", CompCode, Reason);
          *pState = MQSTATE_END;
          return;
      }
/*****
/*      obtain sender's class and instance name (return address)      */
*****/
3      length = strlen (MQmp.ClassName);
      loc = strchr(MQmp.ClassName, ' ');
      if (loc != NULL) length = loc - MQmp.ClassName;
      strncpy (SenderC, MQmp.ClassName, length);
      SenderC[length] = '\0';
      length = strlen (MQmp.LocalInstanceName);
      loc = strchr (MQmp.LocalInstanceName, ' ');
      if (loc != NULL) length = loc - MQmp.LocalInstanceName;
      strncpy(SenderI, MQmp.LocalInstanceName, length);
      SenderI[length] = '\0';

4      switch (*pState) {
          case MQSTATE_NEW:                strcpy (wkfld, "NEW");
                                          break;
          case MQSTATE_CLEAR:              strcpy (wkfld, "CLEAR");
                                          break;
          case MQSTATE_BUSY:                strcpy (wkfld, "BUSY");
                                          break;
          case MQSTATE_DISABLED:            strcpy (wkfld, "DISABLED");
                                          break;
          case MQSTATE_DISABLED_WHILE_BUSY:
                                          strcpy (wkfld, "DISABLED_WHILE_BUSY");
                                          break;
          case MQSTATE_END:                  strcpy (wkfld, "END");
      }
      printf ("%s %s from %s (%s) State=%s\n",
              "BL received ", MQmp.MsgName, SenderC, SenderI, wkfld);

```

Figure 143. Refrigerator's Method "fridge1" (Part 1)

```

/*****
/*      Check if file exists. If not, create it.      */
/*****
5   strcpy (fn, fnfridge);                          /* file name */
      if ( (fp = fopen(fn,"r")) == NULL) {
          if ( (fp = fopen(fn,"w")) == NULL) {
              printf("Cannot create file %s. Abort.\n", fn);
              *pState = MQSTATE_END;
              return;
          }
          for (ik = 0; ik < 5; ik++)
              inventory[ik] = 5;                      /* initial value */
          ik = fwrite (inventory, sizeof(long), 5, fp);
          printf("File %s created with 3 X 5 items.\n", fn);
      }
      fclose (fp);
/*****
/*      read file and check availability      */
/*****
6   if ( (fp = fopen(fn,"r+")) == NULL) {           /* for update */
          printf ("Problem opening file %s\n", fn);
          *pState = MQSTATE_END;
          return;
      }
      ik = fread (inventory, sizeof(long), 5, fp);
      inventory[1] = inventory[1] - 1;
      inventory[2] = inventory[2] - 1;
      inventory[4] = inventory[4] - 1;
      rewind (fp);
      ik = fwrite (inventory, sizeof(long), 5, fp);
      fclose (fp);

```

Figure 144. Refrigerator's Method "fridge1" (Part 2)

1 Each of the classes, refrigerator, basket, and breadbox keep the inventory in a file. To simplify the code each file contains five fields, even though they are not all used in every class. If applicable, the five fields contain in binary form:

```

inventory[0] - bread          inventory[1] - bacon
inventory[2] - lettuce       inventory[3] - tomatoes
inventory[4] - mayonaise

```

2 MQQRYM (MQ QueRY Message) obtains the message properties and (since the buffer length is 0) the length of the message. From the properties structure we will extract the names of the class and instance that sent the message.

3 Class and instance names in the MQmp structure are padded with blanks. This routine extracts the significant characters. We could hard-code the "return address" since only Karen could have sent the message. It is, however, necessary to get the instance name, since this name can change every time Konrad is started.

4 We obtain the status the instance is in to display it in the BLM's window.

5 If the inventory file does not exist we create it. Initially, there will be five of each item in stock. The routine sets all five fields to 5, even only one or, in the refrigerator's case, three are ever used.

6 This code decreases the inventory. For the refrigerator, three fields are updated, for the basket and breadbox it would be one.

```

/*****
/*          order when inventory depleted          */
/*****
7  if (inventory[1] == 0) {
        msg100.number = 1;                /* bacon */
        MQSEND ( HInst, "SHOPPING", SenderI, "OrderMessage",
                0, &msg100, &CompCode, &Reason );
        if (CompCode != MQRC_NONE) {
            printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
            *pState = MQSTATE_END;
            return;
        }
8    *pState = MQSTATE_BUSY;
        printf("Ordered bacon. State=%d\n", *pState);
    }
    if (inventory[2] == 0) {
        msg100.number = 2;                /* lettuce */
        MQSEND ( HInst, "SHOPPING", SenderI, "OrderMessage",
                0, &msg100, &CompCode, &Reason );
        if (CompCode != MQRC_NONE) {
            printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
            *pState = MQSTATE_END;
            return;
        }
        *pState = MQSTATE_BUSY;
        printf("Ordered lettuce. State=%d\n", *pState);
    }
    if (inventory[4] == 0) {
        msg100.number = 4;                /* mayonaise */
        MQSEND ( HInst, "SHOPPING", SenderI, "OrderMessage",
                0, &msg100, &CompCode, &Reason );
        if (CompCode != MQRC_NONE) {
            printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
            *pState = MQSTATE_END;
            return;
        }
        *pState = MQSTATE_BUSY;
        printf("Ordered mayonaise. State=%d\n", *pState);
    }
9    if (*pState != MQSTATE_BUSY) {
        *pState = MQSTATE_CLEAR;
        strcpy (wkfld, "CLEAR");
    }
    else strcpy (wkfld, "BUSY");
/*****
/*          return reply message          */
/*****
10   MQRPLY (HInst, "HaveFromFridge", 0, &msg100, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
            printf("MQRPLY: completion code %d reason %d\n",
                CompCode, Reason);
            *pState = MQSTATE_END;
            return;
        }
        printf("%s: %d bacon, %d lettuce, %d mayo, State=%state=%s\n",
            "Items delivered, portions left",
            inventory[1], inventory[2], inventory[4], wkfld)

```

Figure 145. Refrigerator's Method "fridge1" (Part 3)

7 Immediately after an item is delivered we check if the inventory is depleted. In case of the refrigerator we have to check three items. Into the *OrderMessage* we store the item number, 0 through 4, and send it to the shopping list.

8 If an item is depleted we set the instance state to BUSY. This prevents the instance from accepting more requests. The state is set to clear when a delivery arrives from the grocer.

9 If the object contains inventory we set the state to CLEAR and allow more requests to be processed.

10 The reply is sent to Karen. At the end a message is displayed to tell the user about the status of the instance.

foodinq is included in the business logic for all three objects. Many sections of the program are the same as in *fridge1.c*. The identical sections of the programs are not included in the figures below, however, references to the code in *fridge1.c* are made.

11 The shopping list sends one *FoodInquiry* message for one item at a time. The field "number" contains the item ID, that is 0 through 4. Depending on this number we select the inventory file to be read.

12 If the file does not exist we report back zero.

13 If the file exists we obtain the inventory and store it in the *FoodInquiry* message we send back.

14 We receive a *FoodInquiry* message and we send a message with the same name and structure back to the shopping list.

delivery is invoked when basket, breadbox, or refrigerator receive more food from the grocer. The grocer sends one *FoodDelivery* message for each item he delivers. If the shopping list contains bacon, lettuce and mayonaise, the refrigerator would receive three messages.

15 The field "number" in the message contains the item ID. The item ID determines what file to update.

16 The field "value" in the message contains the quantity the grocer delivers. The quantity is added to the inventory.

17 After a delivery the state is set to CLEAR. It may have been BUSY when there was no inventory left. For the refrigerator we have to take in account that it holds three items. Its state is set to CLEAR when all three items are available.

```

:
:
:                                     /* definitions */
:
: Same as 1 on page 184
:
int      ij, ik;
static char *szFood[] = { "bread", "bacon", "lettuce", "tomatoes", "mayonaise"}
/*****
/*      receive inventory inquiry      */
/*****
:
: Same as 2 on page 184
:
/*****
/*      obtain sender's class and instance name (return address)      */
/*****
:
: Same as 3 on page 184
:
/*****
/*      setup variables (message can be from one of three classes)      */
/*****
11    ij = pFoodInquiry->number;      /* food item 0 - 4); */
        if (ij == 0) strcpy (fn,fnbread);
        else
        if (ij == 3) strcpy (fn,fnbasket);
        else      strcpy (fn,fnfridge);
        if (*pState == MQSTATE_NEW) *pState = MQSTATE_CLEAR;

        switch (*pState) {      /* get current state */
:
:
: Same as 4 on page 184
:
        printf ("Received %s for %s State=%s\n",
                MQmp.MsgName, szFood[ij], wkfld);
/*****
/*      check if file exists      */
/*****
12    if ( (fp = fopen(fn,"r")) == NULL) {
        printf("Inventory file %s does not exist.\n", fn);
        msg100.value = 0;
        msg100.number = ij;
        }
/*****
/*      read file and check inventory      */
/*****
13    else {
        ik = fread (inventory, sizeof(long), 5, fp);
        msg100.value = inventory[ij];
        msg100.number = ij;
        fclose (fp);
        }
/*****
/*      send message to SHOPPING list      */
/*****
14    MQSEND ( HInst, SenderC, SenderI, "FoodInquiry",
                0, &msg100, &CompCode, &Reason );
        if (CompCode != MQRC_NONE) {
        printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        *pState = MQSTATE_END;
        return;
        }
        printf("Inventory: %d %s \n", msg100.value, szFood[ij]);

```

Figure 146. Food Inquiry Method "foodinq"

```

:
:
:                                     /*definitions*/
Same as 1 on page 184
:
/*****
/*      Setup variables (message can be from one of three classes)      */
/*****
15   ij = pFoodDelivery->number;      /* food item 0 - 4)      */
      if (ij == 0) strcpy (fn,fnbread);      /* select file name      */
      else
      if (ij == 3) strcpy (fn,fnbasket);
      else      strcpy (fn,fnfridge);
:
:
:
Same as 4 on page 184
:
      printf ("Received %d portions of %. State=%s\n",
              pFoodDelivery->value, pFoodDelivery->message, wkfld);
/*****
/*      Check if file exists. If not, create it.      */
/*****
:
:
:
Same as 5 on page 185
:
/*****
/*      Read file and update inventory      */
/*****
      if ( (fp = fopen(fn,"r+")) == NULL) {      /* for update      */
          printf ("Problem opening file %s\n", fn);
          return;
      }
16   ik = fread (inventory, sizeof(long), 5, fp);
      inventory[ij] = inventory[ij] + pFoodDelivery->value;
      rewind (fp);
      ik = fwrite (inventory, sizeof(long), 5, fp);
      fclose (fp);
/*****
/*      Change state from BUSY (= wait for food) to CLEAR      */
/*      Note: The refrigerator must have all three items!      */
/*****
17   il = 1;
      if (ij != 0 && ij != 3)      /* not basket and not breadbox */
          if (inventory[1] < 1) il = 0;      /* no bacon in fridge      */
          if (inventory[2] < 1) il = 0;      /* no lettuce in fridge      */
          if (inventory[4] < 1) il = 0;      /* no mayo in fridge      */
      if (il == 1)
          if (*pState == MQSTATE_BUSY) { *pState = MQSTATE_CLEAR;
                                          strcpy (wkfld, "CLEAR");
                                          }
      printf("File %s updated: %d portions of %s available. State=%s\n",
            fn, inventory[ij], pFoodDelivery->message, wkfld);

```

Figure 147. Food Delivery Method "delivery"

5.6.2.3 Business Logic for Microwave and Toaster

The two classes share one specific routine that is called when Karen sends the *MakeToast* or *CookBacon* message. Just as for the previous routines, much of the code is common.

There is one special situation to solve, however. The same routine *cook.c* is used in two different classes. Each class sends a different reply to Karen. Therefore, we have to find out who we are in order to send the correct message back.

We use the `MQQRYT_CLASS` parameter in the `MQQRY` (MQ QueRY) API to obtain the `MQCld` structure. This structure contains the class name. If the class name starts with a "T" it must be the toaster.

For this instance we set the state the `MQSTATE_END` after its work is done. The instance ends and is "new" when the next request arrives.

```

:
: Same definitions as before, except for:
:
MQCLD   MQCld;                               /* class parameters      */
/*****
/*      receive request to cook or toast      */
/*****
        *pState = MQSTATE_END;
:
: Same as 2 on page 184
:
/*****
/*      obtain sender's class and instance name (return address)      */
/*****
:
: Same as 3 on page 184
:
/*****
/*      reply to sender      */
/*****
        BufferLength = sizeof (MQCld);
        MQQRY (HInst, 0, MQQRYT_CLASS, &BufferLength,
                &MQCld, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
            printf ("BL - MQQRY: CC %d reason %d\n", CompCode, Reason);
            return;
        }
        if (MQCld.ClassName[0] == 'T')
            MQRPLY (HInst, "HaveToast", 0, &msg100, &CompCode, &Reason);
        else
            MQRPLY (HInst, "HaveBacon", 0, &msg100, &CompCode, &Reason);
        if (CompCode != MQRC_NONE) {
            printf ("MQRPLY: CC %d reason %d\n", CompCode, Reason);
            return;
        }
        printf ("%s from %s (%s) processed. State=END.\n",
                MQmp.MsgName, SenderC, SenderI);

```

Figure 148. Food Preparation Method "cook"

5.6.2.4 Business Logic for The Grocer

The grocer receives a *FoodOrder* message from the shopping list and sends one *FoodDelivery* message for each item he delivers. The *FoodOrder* is a variable length message set and can contain one to five elements. Each element contains a quantity. The element ID functions as the item ID (0 through 4).

```
#include "bltdef.h"
MQHSET hSet;
MQLONG BufferLength;
MQLONG elements;
MQMP MQmp;
MQLONG CompCode, Reason;
char SenderC [50];
char SenderI [50];
MQLONG DescLength=0;
PMQELIL pdStruct=NULL;
MQLONG eType;
MQLONG eLength;
MQLONG eId;
#define ID_ZERO 0
#define ID_ONE 1
#define ID_TWO 2
#define ID_THREE 3
#define ID_FOUR 4
MQLONG quantity;
MSG100 msg100;
MQLONG p;
unsigned int length;
char *loc;
char wkfld[25];
/* message set */
/* length of the set */
/* number of elements in the set */
/* message properties structure */
/* return codes */
/* class to deliver to */
/* instance name */
/* buffer length for an element */
/* SDDM - integer list element */
/* SDDM - type of element */
/* SDDM - length of element */
/* SDDM - ID of element */
/* SDDM - 0 = deliver bread */
/* SDDM - 1 = deliver bacon */
/* SDDM - 2 = deliver lettuce */
/* SDDM - 3 = deliver tomato */
/* SDDM - 4 = deliver mayonaise */
/* SDDM - value from element */
/* structure for delivery message */
/* work field for loop control */
/* work field */
/* work pointer */
/* work field */
/*****
/* Receive a set containing 1 to 5 food orders */
/*****
1 *pState = MQSTATE_END;
  BufferLength = 0;
  MQQRYM (HInst, 1, &MQmp, &BufferLength, 0, &CompCode, &Reason); /*
  if (CompCode != MQRC_NONE) {
    printf ("Error - MQRYM: CC %d reason %d\n", CompCode, Reason);
    return;
  }
/*****
/* Obtain sender's instance name (used in food delivery message) */
/*****
  length = strlen (MQmp.LocalInstanceName); /* sending instance */
  loc = strchr(MQmp.LocalInstanceName, ' ');
  if (loc != NULL) length = loc - MQmp.LocalInstanceName;
  strncpy(SenderI, MQmp.LocalInstanceName, length);
  SenderI[length] = '\0';
```

Figure 149. Grocer's Method "grocer1.c" (Part 1)

1 Since several new definitions are needed. Most of them are for the use of the Self-Defining Data Manager (SDDM) and referred to later in the text.

2 When the work is done we end the instance by setting its state to MQSTATE_END.

```

/*****
/*      Query the set
/*****
3   BufferLength = 0;
      MQRYS (hFoodOrder,          /* set handle          */
            &elements,           /* number of elements */
            &BufferLength,       /* buffer length      */
            NULL,                /* buffer (N/A)      */
            &CompCode, &Reason ); /* return codes      */
      if (CompCode != MQR_NONE) {
          printf ("Error - MQRYS: CC %d reason %d\n", CompCode, Reason);
          return;
      }
      printf("Received %s with %d item(s)\n",MQmp.MsgName, elements);
/*****
/*      Find the length of the structure and allocate space for it
/*****
4   DescLength = 0;
      MQCPYB ( hFoodOrder, MQEID_ELEMENT_LIST, &DescLength, NULL,
              &CompCode, &Reason );
      if (CompCode != MQR_NONE) {
          printf ("Error - MQCPYB(1): CC %d reason %d\n", CompCode, Reason);
          return;
      }
5   pdStruct = malloc(DescLength);
/*****
/*      Re-issue the call to get the structure
/*****
6   MQCPYB ( hFoodOrder, MQEID_ELEMENT_LIST, &DescLength, pdStruct,
              &CompCode, &Reason );
      if (CompCode != MQR_NONE) {
          printf ("Error - MQCPYB(2): CC %d reason %d\n", CompCode, Reason);
          free (pdStruct);
          return;
      }
      /* printf("Elements in Values array: %d\n",pdStruct->Count); */

```

Figure 150. Grocer's Method "grocer1.c" (Part 2)

3 We use the MQRYS (MQ QueRY Set) API to find out what the message (set) contains. The call returns two values:

- The buffer length
- The number of elements in the set

4 We use a special function of the MQCPYB (MQ CoPY element into Buffer) call to obtain a list of all elements in the set. The call returns the length of the list in DescLength. Since the buffer length (DescLength) is set to zero, the buffer is not referred to and we can specify NULL in its position.

5 The programmer must allocate memory for the element list.

6 The MQCPYB call is issued again, with buffer length and buffer address specified. The PMQELIL structure contains the element list. The structure is described on page 240 of the *Application Programming* manual. The *values* field of this structure contains the following information for each field in the set:

- Element type
- Element length
- Element identifier

```

/*****
/*      Parse the structure
/*****
7
    for ( p = 0; p < pdStruct->Count; p++)
    {
        eType  = pdStruct->Values[p++];
        eLength = pdStruct->Values[p++];
        eId    = pdStruct->Values[p];
8
        switch ( eId )
        {
            case ID_ZERO:
                strcpy (SenderC, "BREADBOX");
                strcpy (wkfld, "bread");
                break;

            case ID_ONE:
                strcpy (SenderC, "FRIDGE");
                strcpy (wkfld, "bacon");
                break;

            case ID_TWO:
                strcpy (SenderC, "FRIDGE");
                strcpy (wkfld, "lettuce");
                break;

            case ID_THREE:
                strcpy (SenderC, "BASKET");
                strcpy (wkfld, "tomatoes");
                break;

            case ID_FOUR:
                strcpy (SenderC, "FRIDGE");
                strcpy (wkfld, "mayonaise");
                break;

        }
        /*****
        /* Obtain quantity to deliver from element
        /*****
9
        MQCPYI (hFoodOrder, eId, &quantity, &CompCode,&Reason);
        if (CompCode != MQRC_NONE) {
            printf ("Error - MQCPYI: CC %d reason %d\n", CompCode, Reason);
            free (pdStruct);
            return;
        }
        /* printf("Deliver to %s\n", SenderC); */
        /*****
        /* Send delivery message
        /*****
10
        msg100.number = eId;           /* item number   */
        msg100.value  = quantity;      /* quantity     */
        strcpy (msg100.message, wkfld); /* item name    */
        MQSEND ( HInst, SenderC, SenderI, "FoodDelivery",
                0, &msg100, &CompCode, &Reason );
        if (CompCode != MQRC_NONE)
            printf ("MQSEND: CC %d reason %d\n", CompCode, Reason);
        else
            printf("Deliver to %s %d portions of %s\n",
                SenderC, quantity, wkfld);
11
    }

    free (pdStruct);

```

Figure 151. Grocer's Method "grocer1.c" (Part 3)

Note: We need the element identifier to tell us what item was ordered. We use the element ID as the product ID.

7 The element list is an array of "triplets". Each triplet describes the type, length and ID of each element in the set. This is described in Usage note 5 of the MQCPYB API call on page 139 of the *Application Programming Guide*.

8 The element ID (product ID) says what product is ordered and where to ship it.

9 MQCPYI (MQ CoPY Integer) copies the quantity from the set and into the field quantity.

10 Item ID and quantity are stored into the fixed length message *FoodDelivery* and sent to either the breadbox, basket, or refrigerator.

11 After all orders are filled we free the space allocated for the element structure.

5.6.2.5 Common Business Logic Programs

The following programs are used by (almost) all classes:

1. xclear
2. xrepair
3. xignore
4. xinqury
5. xgremlin

xclear is called when the BLM ends. MQ3T sends a system message that allows the user to do some cleanup.

```
#include "bltdef.h"
printf("BLM ended, clear up...\n");
```

Figure 152. Common Method "xclear.c"

xrepair process the *RepairMessage* sent by the repair list.

```
/*
 * xrepair: enable class
 */
#include "bltdef.h"
printf ("Working again ...\n");
if (*pState == MQSTATE_DISABLED_WHILE_BUSY) *pState = MQSTATE_BUSY;
else if (*pState == MQSTATE_DISABLED) *pState = MQSTATE_CLEAR;
else if (*pState == MQSTATE_NEW) *pState = MQSTATE_CLEAR;
```

Figure 153. Common Method "xrepair.c"

xignore displays all messages that the application does not care about. The purpose of this routine is to show the user what messages are ignored, such as late replies.

```

/*****
/*          xignore:  discard messages          */
/*****
MQMP      MQmp;                               /* message properties */
MQLONG    BufferLength;
char      SenderC   [50];                     /* sending class      */
char      SenderI   [100];                    /* sending instance   */
unsigned  int length;
char      *loc;
MQLONG    CompCode, Reason;                   /* completion codes   */
/*****
/*          get message properties             */
/*****
        BufferLength = 0;
        MQRYM  (HInst,                          /* instance of BL      */
                1,                               /* first message      */
                &MQmp,                          /* MQMP structure (output) */
                &BufferLength,                   /* data length (output) */
                0,                               /* data area (N/A)    */
                &CompCode, &Reason );           /* return codes       */

        if (CompCode != MQRC_NONE) {
            printf ("Ignore: MQRYM: CC %d reason %d\n", CompCode, Reason);
            return;
        }
/*****
/*          obtain sender's class and instance name          */
/*****
        length = strlen (MQmp.ClassName);        /* sending class      */
        loc    = strchr(MQmp.ClassName, ' ');
        if (loc != NULL) length = loc - MQmp.ClassName;
        strncpy (SenderC, MQmp.ClassName, length);
        SenderC[brk.length] = '\0';
        length = strlen (MQmp.LocalInstanceName); /* sending instance */
        loc    = strchr(MQmp.LocalInstanceName, ' ');
        if (loc != NULL) length = loc - MQmp.LocalInstanceName;
        strncpy(SenderI, MQmp.LocalInstanceName, length);
        SenderI[length] = '\0';
/*****
/*          display message and end          */
/*****
        printf ("%s %s from %s (%s)\n",
                "BL ignores ", MQmp.MsgName, SenderC, SenderI);
        printf ("  State=%d Rule=%d, MsgType=%d OpCode=%d\n",
                *pState, RuleId, MQmp.MsgType, MQmp.OperationCode);
        if (*pState == MQSTATE_NEW) *pState = MQSTATE_END;

```

Figure 154. Common Method "xignore.c"

xinquiry processes *InquiryRequest* messages from the repair list. It responds by reporting back the instance state.

```

MQMP      MQmp;
MQLONG   BufferLength, CompCode, Reason;
char     SenderC [50];
char     SenderI [100*rbk.];
char     wkfld [50];
unsigned int length;
char     *loc;
MSG100   msg100;
/*****
/*           Get properties of inquiry REQUEST message           */
/*****
    BufferLength = 0;
    MQRYM (HInst, 1, &MQmp, &BufferLength, 0, &CompCode, Reason); /*
    if (CompCode != MQRC_NONE) {
        printf ("BL - MQRYM: completion code %d reason %d\n",
                CompCode, Reason );
        *pState = MQSTATE_END;
        return;
    }
/*****
/*           Get sender's class and instance names           */
/*****
    length = strlen (MQmp.ClassName);           /* sending class */
    loc = strchr(MQmp.ClassName, ' ');
    if (loc != NULL) length = loc - MQmp.ClassName;
    strncpy (SenderC, MQmp.ClassName, length);
    SenderC[length] = '\0';
    length = strlen (MQmp.LocalInstanceName); /* sending instance */
    loc = strchr(MQmp.LocalInstanceName, ' ');
    if (loc != NULL) length = loc - MQmp.LocalInstanceName;
    strncpy(SenderI, MQmp.LocalInstanceName, length);
    SenderI[length] = '\0';
/*****
/* Change state IF this is first message the instance receives */
/*****
    if (*pState == MQSTATE_NEW) *pState = MQSTATE_CLEAR;
/*****
/*           Reply with state           */
/*****
    switch (*pState) {
        case MQSTATE_NEW:           strcpy (wkfld, "NEW");
                                    break;
        case MQSTATE_CLEAR:         strcpy (wkfld, "CLEAR");
                                    break;
        case MQSTATE_BUSY:          strcpy (wkfld, "BUSY");
                                    break;
        case MQSTATE_DISABLED:      strcpy (wkfld, "DISABLED");
                                    break;
        case MQSTATE_DISABLED_WHILE_BUSY:
                                    strcpy (wkfld, "DISABLED_WHILE_BUSY");
                                    break;
        case MQSTATE_END:           strcpy (wkfld, "END");
    }
    strcpy (msg100.message, wkfld);
    msg100.number = *pState;
    MQRPLY ( HInst, "InquiryReply", 0, &msg100, &CompCode, &Reason);
    if (CompCode != MQRC_NONE) {
        printf ("MQRPLY: completion code %d reason %d\n",
                CompCode, Reason );
        *pState = MQSTATE_END;
        return;
    }
    printf ("Reply to %s from %s (%s): State=%s\n",
            MQmp.MsgName, SenderC, SenderI, wkfld);

```

Figure 155. Common Method "xinquiry.c"

xgremlin is called when the Gremlin sends a message to disable a class or to set the inventory of an item to zero.

```

#include "bltdef.h"
int ij, ik;
long inventory[5];
char * fnbasket="basket.dat";
char * fnfridge="fridge.dat";
char * fnbread="breadbox.dat";
char fn[20];
char wkfld[50];
FILE *fp;
/*****/
/* Find out if disable or spoil, */
/* for spoil set file name and item number in file */
/*****/
ik = pGremlinMessage->number; /* ghost radio button 1 to 14 */
printf ("radio button is %d\n",ik);
ij = -1;
if (ik == 6 ) { strcpy (fn, fnbread); /* bread */
                ij = 0;
                strcpy (wkfld, "bread portions"); }
if (ik == 7 ) { strcpy (fn, fnbasket); /* tomatoes */
                ij = 3;
                strcpy (wkfld,"tomatoes"); }
if (ik == 8 ) { strcpy (fn, fnfridge); /* bacon */
                ij = 1;
                strcpy (wkfld, "bacon portions"); }
if (ik == 9 ) { strcpy (fn, fnfridge); /* lettuce */
                ij = 2;
                strcpy (wkfld, "lettuce portions");}
if (ik == 10 ) { strcpy (fn, fnfridge); /* mayonaise*/
                 ij = 4;
                 strcpy (wkfld, "mayonaise portions"); }
/*****/
/* Disable: radio buttons 1 - 5 and 11 - 14 */
/*****/
if (ij == -1) {
    printf ("The gremlin strikes again...\n");
    if (*pState == MQSTATE_BUSY) *pState = MQSTATE_DISABLED_WHILE_BUSY;
    else *pState = MQSTATE_DISABLED;
    return;
}
/*****/
/* Spoil: return of file does not exist */
/*****/
if ( (fp = fopen(fn,"r+") == NULL) {
    if (*pState == MQSTATE_NEW) *pState = MQSTATE_CLEAR;
    printf("File %s does not exist, nothing to spoil.\n", fn);
    return;
}
/*****/
/* Spoil: set inventory to 0 */
/*****/
ik = fread (inventory, sizeof(long), 5, fp);
printf("File %s, %s spoiled: %d\n", fn, wkfld, inventory[ij]);
inventory[ij] = 0;
rewind (fp);
ik = fwrite (inventory, sizeof(long), 5, fp);
fclose (fp);
/*****/
/* set state to "disabled" */
/*****/
if (*pState == MQSTATE_DISABLED) { *pState = MQSTATE_DISABLED_WHILE_BUSY;
                                   strcpy (wkfld, "DISABLED_WHILE_BUSY");
}
else { *pState = MQSTATE_BUSY;
       strcpy (wkfld,"BUSY");
}

```

Figure 156. Common Method "xgremlin.c"

5.7 System Test

The application consists of:

- Five Visual Basic programs in the Windows workstation
- Seven C programs in the AIX workstation

The C programs are compiled error free, and for the Visual Basic programs we made error free EXE files. To recompile all files on the AIX machine you may use the command file "redo" shown below. The routine clears all queues, too.

```
rm *.cb
rm *.exp
rm *.dat
touch *.h
touch *.ch
bmqcc konrad.cs
bmqcc luigi.cs
bmqcc gremlin.cs
bmqcc repair.cs
bmqcc shopping.cs
bmqcc /s karen.cs
bmqcc /s basket.cs
bmqcc /s breadbox.cs
bmqcc /s fridge.cs
bmqcc /s toaster.cs
bmqcc /s micro.cs
bmqcc /s grocer.cs
make -f konrad.mak
make -f luigi.mak
make -f gremlin.mak
make -f repair.mak
make -f shopping.mak
make -f karen.mak
make -f basket.mak
make -f breadbox.mak
make -f fridge.mak
make -f micro.mak
make -f toaster.mak
make -f grocer.mak
runmqsc < clearq.in
```

Figure 157. BLT: Redo all BLs

The queue manager resides in the AIX server. The Windows workstation has a "server connection" to the RS/6000.

On the server we executed the following runmqsc command to create a channel between the Windows client and the AIX server:

```
def chl (OAKC1.T0.RS60001) CHLTYPE(SVRCONN) TRPTYPE(TCP) MCAUSER('mqm') +
like(system.def.svrconn)
```

In the client workstation we added to the AUTOEXEC.BAT file:

```
SET MQSERVER=OAKC1.T0.RS60001/TCP/9.24.104.26
```



```

*****/
* File: BLTCOMA.TST */
* Description: Sample MQSC source defining MQM queues */
* For use with BLT sample */
*****/
* Default Oak Queues */
*****/
    DEFINE QMODEL('3TClient') REPLACE +
        SHARE +
        DESCR('Default 3T Client Model Queue')
    DEFINE QLOCAL('3TEXQ') REPLACE +
        DESCR('Default Oak Exception Queue')
    DEFINE QLOCAL('3TWAQ') REPLACE +
        DESCR('Default Oak WorkArea Queue')
*****/
* Queues for use with BLT sample */
*****/
    DEFINE QLOCAL('KAREN') REPLACE +
        DESCR('Queue for BL KAREN')
    DEFINE QLOCAL('LUIGI') REPLACE +
        DESCR('Queue for PL LUIGI')
    DEFINE QLOCAL('BASKET') REPLACE +
        DESCR('Queue for BL BASKET')
    DEFINE QLOCAL('BREADBOX') REPLACE +
        DESCR('Queue for BL BREADBOX')
    DEFINE QLOCAL('TOASTER') REPLACE +
        DESCR('Queue for BL TOASTER')
    DEFINE QLOCAL('MICRO') REPLACE +
        DESCR('Queue for BL MICRO')
    DEFINE QLOCAL('FRIDGE') REPLACE +
        DESCR('Queue for BL FRIDGE')
    DEFINE QLOCAL('GREMLIN') REPLACE +
        DESCR('Queue for PL GREMLIN')
    DEFINE QLOCAL('REPAIR') REPLACE +
        DESCR('Queue for PL REPAIR')
    DEFINE QLOCAL('SHOPPING') REPLACE +
        DESCR('Queue for PL SHOPPING')
    DEFINE QLOCAL('GROCER') REPLACE +
        DESCR('Queue for BL GROCER')
*****/
* END OF BLTCOMA */
*****/

```

Figure 158. BLT: Queue Definitions "bltcoma.tst"

The file *bltcoma.tst* contains all queues required for the BLT application. To add the queues issue the command:

```
runmqsc < bltcoma.tst
```

Note: We work with local queues only, BLs and PLs use the same queue manager in the AIX server.

Before we can begin our system test we have to prepare some more files:

- One profile for all PLs
- A profile for each of the BLs

```
*****
*
* PLMS.PRF: Startup profile for the PL Manager *
*
*****

[CLIENT]

ClassNames = KONRAD LUIGI GREMLIN REPAIR SHOPPING
LogLevel = 300
```

Figure 159. BLT: Profile for Presentation Logic

Note: The class names are case sensitive!

```
*****
*
* KAREN.PRF: Startup profile for the BLM for KAREN *
*
*****

[SERVER]

ClassNames = KAREN
LogLevel = 300
```

Figure 160. BLT: Profile for Business Logic

For each BL we need a file as the one shown above. The only difference is the class name.

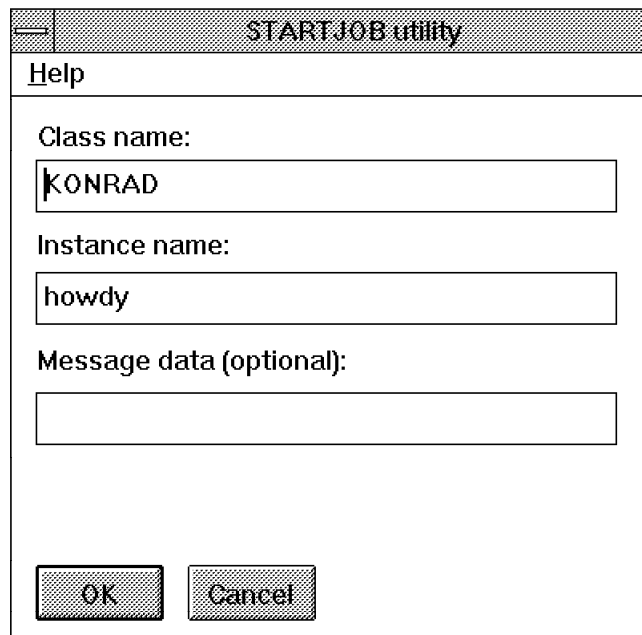
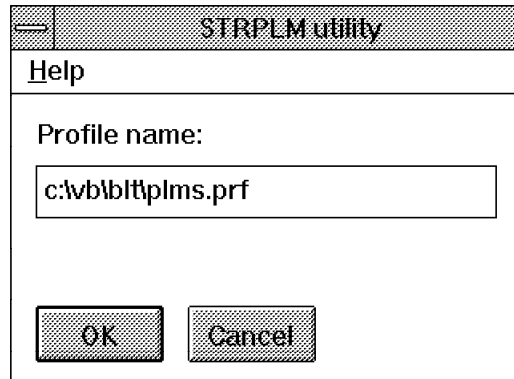
Note: When log level is set to 300 all error messages will be recorded in the file BMQERROR.LOG.

In the server, you can start the BLMs in the foreground or in the background. If you start them in the background, all messages will be displayed in one window. The "&" at the end of the command starts the BLM in the background. You may use the following commands:

```
strblm KAREN.prf &
strblm FRIDGE.prf &
strblm BASKET.prf &
strblm BREADBOX.prf &
strblm TOASTER.prf &
strblm MICRO.prf &
strblm GROCER.prf
```

In the client, go into windows and use the MQ3T icons to start the BLMs. You have to execute two functions as shown in the following two figures:

1. strplm
2. startjob



Four GUIs should appear on the screen. Remember, Luigi's window appears when he receives the first pizza order. Arrange the windows and explore MQ3T!

Diskette

All files are included on the second diskette.

Chapter 6. Data Conversion

An MQ Three Tiers application works in a multiplatform client/server environment. Hence, the programs that comprise your application should be able to run:

- In its entirety in a stand-alone machine
- Distributed on different machines of different types

Executing parts of an application on a different platform raises the data conversion issue. Between different platforms, you can find different code pages and different number formats. On the RS/6000 and PC platforms (that are currently the only platforms supported by MQ3T) the number formats are different.

MQSeries uses the following fields in conjunction with data conversion:

- In the Message Descriptor is a parameter called *Format* that identifies the format of the message. The format field can have one of the following three values:
 1. MQFMT_NONE if you don't mind the format of the message
 2. MQFMT_STRING as the standard MQ format for string messages
 3. The name of a user-defined format that must be no longer than 8 characters and must match the name of the user data exit routine (upper case).
- In the MQGET API you can specify the MQGMO_CONVERT option if your message needs a data conversion.

MQ3T exploits MQSeries data conversion method shielding the coding steps and requires that in the definition of the messages the parameter ConversionDLL is filled with the name of the library to use to convert the code pages and number format of the data.

The library name must contain eight characters or fewer. You need to specify this parameter when you send fixed-format messages (from a MQ3T point of view) between two classes on different types of computers.

If you don't specify a conversion DLL for a fixed-format message, the Class Compiler assumes the name to be the first 8 characters of StructName field in the message description.

```
MESSAGE
BEGIN
  MsgName      BLREQUEST
  MsgType      REQUEST
  OperationCode OC_BLREQUEST
  Format        FIXED
  StructName    HELLO
  StructFile    hello1st.h
  StructLen    60
  ConversionDLL MQFMT_STRING
END
```

Figure 161. Message Description with Conversion DLL

For a variable-format message, you don't need a conversion library. The Self-Defining Data Manager (SDDM) does the conversion for you.

Data conversion is needed for the messages between the BL program on the AIX machine and the PL program on the Windows machine.

In the File Transfer example, we use the standard MQSeries format, MQFMT_STRING, since the message contains only three character strings, namely the names of the source file, target file, and destination class. Each field is 20 bytes long.

Data conversion is executed in the AIX machine since it is the server and that is where the Queue Manager runs. The MQFMT_STRING format conversion is standard in MQSeries. It is performed by the Queue Manager without any exit.

For messages between BL program and DL program (DLREQUEST) no conversion is necessary, since each program runs on a RS/6000 machine. Figure 162 shows the description of the message used to transfer a file from one AIX machine to another. However, you would need data conversion if the DL program were on a PC.

```
MESSAGE
BEGIN
  MsgName      DLREQUEST
  MsgType      REQUEST
  OperationCode OC_DLREQUEST
  Format        FIXED
  StrucName     HELLOD
  StrucFile     hello1st.h
  StrucLen     500
END
```

Figure 162. Message Description without Conversion DLL

If you have a MQ3T application that sends or receives messages that contain other data types than strings, and you want to make it portable add the **ConversionDLL** parameter to the message definition section.

How does the data conversion work?

- Outgoing messages:

When MQ3T sends a fixed-length message, the 3T run-time program (PLM or BLM) puts the name of the conversion DLL in the message descriptor. No conversion is done on the sender's side.

- Incoming messages:

PLM and BLM use the MQGET call to get messages for the PLs and BLs they support. This API performs the conversion if:

1. The message came from another platform.
2. The message descriptor contains a conversion DLL.
3. The conversion DLL is available.

MQ3T always puts the DLL name in the message descriptor. You get an error message if the DLL cannot be found.

Note: When the environment doesn't require any conversion no action is taken.

6.1 Creating a Conversion DLL for AIX

In the BLT example, almost all messages are fixed-format messages with a user-defined structure that contains strings and integers. Therefore, a data conversion exit has to be written by the user.

This is the user-defined structure MSG100:

```
typedef struct _MSG100
{
    MQCHAR messages [20]
    MQLONG number;
    MQLONG value;
    MQCHAR filler [72]
} MSG100;
```

Figure 163. Message Structure that Needs Data Conversion

The steps to create a data conversion exit routine on an AIX machine (we use a RS/6000 as the server) are as follows:

- Step 1. Copy the "C" file structure in a separate file.
- Step 2. Use the *Create Data Conversion Exit Utility*, `crtmqcvx`, to generate a subroutine that uses standard libraries to convert the C structure. The command to invoke the utility is:

```
crtmqcvx <struct file> <output file>
```

If the structure is in the file `MSG100.str` and you type the command below then you create the file `MSG100.out` that is shown in Figure 164.

```
crtmqcvx MSG100.str MSG100.out
```

```
MQLONG Convert_MSG100(
    PMQBYTE *in_cursor,
    PMQBYTE *out_cursor,
    PMQBYTE in_lastbyte,
    PMQBYTE out_lastbyte,
    MQHCONN hConn,
    MQLONG opts,
    MQLONG MsgEncoding,
    MQLONG ReqEncoding,
    MQLONG MsgCCSID,
    MQLONG ReqCCSID,
    MQLONG CompCode,
    MQLONG Reason)
{
    MQLONG ReturnCode = MQRC_NONE;
    ConvertChar(20); /* message */
    AlignLong();
    ConvertLong(1); /* number */
    AlignLong();
    ConvertLong(1); /* value */
    ConvertChar(72); /* filler */

    Fail:
        return(ReturnCode);
}
```

Figure 164. Data Exit Source File

Step 3. Copy the MQSeries sample program amqsvfcx.c in a file named MSG100.c. The name of the file must be in uppercase! Use the following command:

```
cp /usr/lpp/mqm/samp/amqsvfcx.c MSG100.c
```

Step 4. Edit the MSG100.c file and follow the instructions in the file. Comments guide you in where to insert the subroutine in Figure 164 on page 205 and where to make changes.

The following notes refer to the conversion routine in Figure 165 on page 207 and Figure 166 on page 208.

1 Insert here the functions prototypes for the functions produced by the data conversion utility program.

2 Change the entry point name to EntryPointMSG100.

3 Change the entry point name to EntryPointMSG100.

4 Change the call to the subroutine.

5 Change function name.

6 Insert the functions produced by the data conversion exit utility program.

Step 5. Make a copy of the sample makefile helloxx.mak and call it MSG100.mak. Use the following command:

```
cp /usr/lpp/mq3t/samples/c/helloxx.mak MSG100.mak
```

Step 6. Edit the new "C" makefile and change:

- The name of the source file to MSG100.c
- The name of the object file to MSG100.o
- The name of the export file to MSG100.exp
- The entry point function name to EntryPointMSG100

The changes are marked in the Figure 167 on page 209.

Step 7. Copy the sample file hellox.exp in a file called MSG100.exp using the command:

```
copy /usr/lpp/mq3t/samples/c/hellox.exp MSG100.exp
```

This file is required to compile the data exit.

Step 8. In the .exp file, replace the EntryPointFunctionName with the name of the subroutine. MSG100.exp contains only this line:

```
EntryPointMSG100
```

Step 9. Run the compilation using the command

```
make -f MSG100.mak
```

Step 10. Copy the generated files MSG100 and MSG100_r into the /usr/lib directory using the commands:

```
cp MSG100 /usr/lib  
cp MSG100_r /usr/lib
```



```

#include <cmqc.h> /* For MQI datatypes */
#include <cmqxc.h> /* For MQI exit-related definitions */
#include <amqsvmha.h> /* For sample macro definitions */
/*****
/* Insert the function prototypes for the functions produced by
/* the data conversion utility program.
*****/
MQLONG Convert_MSG100( 1
    PMQBYTE *in_cursor,
    PMQBYTE *out_cursor,
    PMQBYTE in_lastbyte, part 1 of MSG100.out
    PMQBYTE out_lastbyte, on page 205
    MQHCONN hConn,
    MQLONG opts,
    MQLONG MsgEncoding,
    MQLONG ReqEncoding,
    MQLONG MsgCCSID,
    MQLONG ReqCCSID,
    MQLONG CompCode,
    MQLONG Reason);
MQDATACONVEXIT EntryPointMSG100; 2
/*****
/* The name of the function is not actually used to call the
/* conversion exit BUT it must be marked as the entry point using
/* the -e option during linking.
*****/
void MQENTRY EntryPointMSG100( 3
    PMQDXP pDataConvExitParms, /* Data-conversion exit parameter */
    /* block */
    PMQMD pMsgDesc, /* Message descriptor */
    MQLONG InBufferLength, /* Length in bytes of InBuffer */
    PMQVOID pInBuffer, /* Buffer containing the unconverted */
    /* message */
    MQLONG OutBufferLength, /* Length in bytes of OutBuffer */
    PMQVOID pOutBuffer) /* Buffer containing the converted */
    /* message */
{
    MQLONG ReturnCode = MQRC_NONE;
    MQHCONN hConn = pDataConvExitParms->Hconn;
    MQLONG opts = pDataConvExitParms->AppOptions;
    PMQBYTE in_cursor = pInBuffer;
    PMQBYTE out_cursor = pOutBuffer;
    PMQBYTE in_lastbyte = (PMQBYTE)pInBuffer + InBufferLength - 1;
    PMQBYTE out_lastbyte = (PMQBYTE)pOutBuffer + OutBufferLength - 1;
    MQLONG MsgEncoding = pMsgDesc->Encoding;
    MQLONG ReqEncoding = pDataConvExitParms->Encoding;
    MQLONG MsgCCSID = pMsgDesc->CodedCharSetId;
    MQLONG ReqCCSID = pDataConvExitParms->CodedCharSetId;
    MQLONG CompCode = pDataConvExitParms->CompCode;
    MQLONG Reason = pDataConvExitParms->Reason;
/*****
/* Insert calls to the code fragments to convert the format's
/* structure(s) here.
*****/
    ReturnCode = Convert_MSG100( 4
        &in_cursor,
        &out_cursor,
        in_lastbyte,
        out_lastbyte,
        hConn,
        opts,
        MsgEncoding,
        ReqEncoding,
        MsgCCSID,
        ReqCCSID,
        CompCode,
        Reason);
/*****
/* Check whether the conversion succeeded or failed and return
/* the values required by the caller.
*****/
}

```

Figure 165. C Source Program for Conversion Conversion Exit (Part 1)

```

if (ReturnCode == MQRC_NONE)
{
    pDataConvExitParms->ExitResponse = MQXDR_OK;
    pDataConvExitParms->CompCode      = MQCC_OK;
    pDataConvExitParms->Reason        = ReturnCode;
    /*****
    /* If the message had not been truncated then return its new */
    /* length. */
    /* Warning - this assumes that out_cursor has been set up to */
    /* point to the end of the message. Routines produced by the */
    /* data conversion exit utility will do this BUT if you are */
    /* writing your own routines ensure it is updated or the */
    /* message could end up with a zero length and appear not to */
    /* get converted! */
    /*****
    if (Reason != MQRC_TRUNCATED_MSG_ACCEPTED) {
        pDataConvExitParms->DataLength = out_cursor
            - (PMQBYTE)pOutBuffer;
    } /* end if */
    }
    /*****
    /* If conversion failed for lack of data or lack of output buffer */
    /* but the message had been truncated then indicate success but */
    /* do not adjust the values of Reason, CompCode or DataLength. */
    /*****
else if ((Reason == MQRC_TRUNCATED_MSG_ACCEPTED) &&
        ((ReturnCode == MQRC_TRUNCATED_MSG_ACCEPTED) ||
         (ReturnCode == MQRC_CONVERTED_MSG_TOO_BIG)))
{
    pDataConvExitParms->ExitResponse = MQXDR_OK;
}
/*****
/* Otherwise indicate that conversion of the message data failed. */
/*****
else {
    pDataConvExitParms->ExitResponse = MQXDR_CONVERSION_FAILED;
    pDataConvExitParms->CompCode      = MQCC_WARNING;
    pDataConvExitParms->Reason        = ReturnCode;
} /* end if */
return;
}
/*****
/* Insert the functions produced by the data conversion exit */
/* utility program. */
/*****
MQLONG Convert_MSG100(
    PMQBYTE *in_cursor,
    PMQBYTE *out_cursor,
    PMQBYTE in_lastbyte,
    PMQBYTE out_lastbyte,
    MQHCONN hConn,
    MQLONG opts,
    MQLONG MsgEncoding,
    MQLONG ReqEncoding,
    MQLONG MsgCCSID,
    MQLONG ReqCCSID,
    MQLONG CompCode,
    MQLONG Reason)
{
    MQLONG ReturnCode = MQRC_NONE;
    ConvertChar(20); /* message */
    AlignLong();
    AlignLong();
    ConvertLong(1); /* number */
    AlignLong();
    ConvertLong(1); /* value */
    ConvertChar(72); /* filler */
Fail:
    return(ReturnCode);
}

```

Figure 166. C Source Program for Conversion Conversion Exit (Part 2)

```

*****
PROJECT      = MSG100
BUILDPATH    = .
LIBPATH      = $(BUILDPATH)
COMMONPATH   = $(BUILDPATH)
OSSPATH      = $(BUILDPATH)
PROJECTPATH  = $(BUILDPATH)

*****

INCLUDEPATH  = -I/usr/lpp/mqm/inc

LFLAGSST     = -L$(LIBPATH) -lXm -lXt -lX11 -bM:SRE \
              -e EntryPointMSG100 -lmqm \
              -bE:$(PROJECTPATH)/MSG100.exp \
              -bmap:$(PROJECT).map

LFLAGSMT     = -L$(LIBPATH) -lXm -lXt -lX11 -bM:SRE \
              -e EntryPointMSG100 -lmqm_r \
              -bE:$(PROJECTPATH)/MSG100.exp \
              -bmap:$(PROJECT).map

CC           = xlc_r

DEFINES      = -D_XOPEN_SOURCE -DXTFUNCPROTO -DAIX -D_AIX -DPOSIX_SOURCE -DUNIX \
              -DNOTCX_ -DLAKES_AIX -DLAKES -DCOMMON \
              -DOAK_SKEL -DOAK_XCHECK -DOAK_AIX

CFLAGS       = -c -Dsigned= -Dvolatile= -D_Optlink

*****

HEADERS      =

SOURCES      = MSG100.c

OBJECTS      = MSG100.o

*****

all: $(PROJECT) $(PROJECT)_r

$(PROJECT): $(OBJECTS)
rm -rf $(PROJECT)
$(CC) $(LFLAGSST) $(OBJECTS)
mv a.out $(PROJECT)
chmod g+w $(PROJECT)

$(PROJECT)_r: $(OBJECTS)
rm -rf $(PROJECT)_r
$(CC) $(LFLAGSMT) $(OBJECTS)
mv a.out $(PROJECT)_r
chmod g+w $(PROJECT)_r
.c.o:
$(CC) $(CFLAGS) $(DEFINES) $(INCLUDEPATH) $<

```

Figure 167. Make File MSG100.mak for AIX

The files MSG100 and MSG100_r are the data conversion exit. There are two programs:

- MSG100 is loaded in a basic environment.
- MSG100_r is loaded in a DCE threaded environment.

If the data conversion exits are in a mixed DCE and non-DCE environment the queue manager will detect the calling environment and load the appropriate object.

It is important to copy the two files into the /usr/lib directory in order to make them visible and executable by every application that needs this kind of conversion, even if it is a client application.

In the BLT environment described in this book the data exit has to be on the AIX machine, because the client exploits the MQ services of the server. However, if the application is split between different machines a data exit is necessary on every machine that receives messages with the MSG100 format.

6.2 Creating a Conversion DLL for OS/2

The steps to build a data conversion exit on a OS2 machine are similar to those required for the AIX machine. There are, however, differences in the directory names.

Step 1. Copy the "C" file structure into a separate file.

Step 2. Use the *Create Data Conversion Exit Utility*, `crtmqcvx`, to generate a subroutine that uses standard libraries to convert the C structure.

If the structure is in the file `MSG100.str` and you type the command below than you create the file `MSG100.out` that is shown in Figure 164 on page 205.

```
crtmqcvx MSG100.str MSG100.out
```

The output file is the same as for AIX, shown in Figure 164 on page 205.

Step 3. Copy the sample file `amqsvfc2.c` from the MQSeries C samples into a file named `MSG100.c`. Use the following command:

```
copy d:\mqm\tools\c\samples\amqsvfc2.c MSG100.c
```

Step 4. Edit `MSG100.c` and follow the guidelines in the file. These comments show how to insert the subroutine that has been generated before.

Step 5. Copy the sample makefile `hellox2.mak` into a file called `MSG100.mak`. Use the following command:

```
copy d:\3tier2\samples\hellox2.mak MSG100.mak
```

Step 6. Edit the new makefile and change the file names as shown in Figure 168 on page 211.

Step 7. Copy the file `hellox.def` from the MQ3T samples into a new file and call it `MSG100.def`. Use the following command:

```
copy d:\3tier2\samples\hellox.def MSG100.def
```

This file is required to compile the data exit.

Step 8. Change in the `.def` file the `EntryPointFunctionName` as shown in Figure 169 on page 211.

Step 9. Compile the user exit using this command:

```
nmake MSG100.mak
```

Step 10. Copy the generated file `MSG100.dll` into `mqm's dll` directory:

```
copy MSG100.dll d:\mqm\dll
```

```

.SUFFIXES:
.SUFFIXES: .obj .c

CC = icc /c /Ge- /Ms /Q /Ti+
LINK = LINK386 /DEBUG /NOE /NOD /NOI /ALIGN:16 /EXEPACK /M
/BASE:0x10000

.c.obj:
    $(CC) -Fo$.obj $.c

all: MSG100.d11

MSG100.lnk: MSG100.mak
    echo MSG100.obj > $.lnk
    echo $.dll >> $.lnk
    echo $.map >> $.lnk
    echo mqmvx.lib+ >> $.lnk
    echo dde4mbs.lib+ >> $.lnk
    echo os2386.lib >> $.lnk
    echo MSG100.def >> $.lnk
MSG100.obj: MSG100.C\
    $(HEADERS)

MSG100.d11: $.lnk MSG100.obj
    $(LINK) @$*.lnk

```

Figure 168. Make File MSG100.mak for OS/2

```

LIBRARY MSG100 INITINSTANCE TERMINSTANCE

PROTMODE

DESCRIPTION 'Sample Data Conversion Exit for MSG100 Structure'

CODE SHARED
DATA NONSHARED MULTIPLE

HEAPSIZE 4096
STACKSIZE 8192

EXPORTS
    EntryPointMSG100 @1

```

Figure 169. MSG100.def File for OS/2

Note: For more information about data conversion exits refer to pages 270 through 280 of the *Distributed Queue Management Guide for MQSeries* and to the *MQ Three Tier Application Design* manual for the ConversionDLL parameter.

Appendix A. Class Source Files for BLT Example

This appendix contains the class source files and definitions for the BLT example.

A.1 Messages for The BLT Example

```

File name: messages.ch
/*****
/*
/* Messages.CH: Include for Class Source Files - Messages
/*
/*
*****/
MESSAGE          // StartJob message
BEGIN
    MsgName      StartJob
    MsgType      INFORM
    OperationCode OC_STARTJOB
    Format        FIXED
    StrucLen     100
    StrucName     STARTJOB
    StrucFile    bltstruc.h
    ConversionDLL MSG100
END
MESSAGE          // Show windows
BEGIN
    MsgName      Show
    MsgType      INFORM
    OperationCode OC_SHOW
    Format        FIXED
    StrucLen     0
END
MESSAGE          // KONRAD to KAREN: ask for BLT
BEGIN
    MsgName      FeedMe
    MsgType      REQUEST
    OperationCode OC_SANDWICH
    Format        FIXED
    StrucName     MSG100
    StrucFile    bltstruc.h
    ConversionDLL MSG100
END
MESSAGE          // KAREN to KONRAD: here is the BLT
BEGIN
    MsgName      Sandwich
    MsgType      REPLY
    OperationCode OC_SANDWICH
    Format        FIXED
    StrucName     MSG100
    StrucFile    bltstruc.h
    ConversionDLL MSG100
END
MESSAGE          // KAREN to KONRAD: cannot make BLT
BEGIN
    MsgName      Starve
    MsgType      REPLY
    OperationCode OC_STARVE
    Format        FIXED
    StrucName     MSG100
    StrucFile    bltstruc.h
    ConversionDLL MSG100
END

```

```

MESSAGE          // KONRAD to LUIGI: order pizza
BEGIN
  MsgName        DeliverPizza
  MsgType        REQUEST
  OperationCode  OC_PIZZA
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
MESSAGE          // LUIGI to KONRAD: here is the pizza
BEGIN
  MsgName        EatPizza
  MsgType        REPLY
  OperationCode  OC_PIZZA
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
END
MESSAGE          // KAREN to vegetable basket
BEGIN
  MsgName        GetTomato
  MsgType        REQUEST
  OperationCode  OC_TOMATO
  Role           1
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Vegetable basket to KAREN
BEGIN
  MsgName        HaveTomato
  MsgType        REPLY
  OperationCode  OC_TOMATO
  Role           1
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // KAREN to bread box: get bread
BEGIN
  MsgName        GetBread
  MsgType        REQUEST
  OperationCode  OC_BREAD
  Role           2
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // bread box to KAREN: have bread
BEGIN
  MsgName        HaveBread
  MsgType        REPLY
  OperationCode  OC_BREAD
  Role           2
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END

```



```

MESSAGE          // KAREN to FRIDGE: get mayo, bacon, lettuce
BEGIN
  MsgName        GetFromFridge
  MsgType        REQUEST
  OperationCode  OC_FRIDGE
  Role           3
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // FRIDGE to KAREN: here is it
BEGIN
  MsgName        HaveFromFridge
  MsgType        REPLY
  OperationCode  OC_FRIDGE
  Role           3
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // KAREN to TOASTER: make toast
BEGIN
  MsgName        MakeToast
  MsgType        REQUEST
  OperationCode  OC_TOAST
  Role           5
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // TOASTER to KAREN: here is toast
BEGIN
  MsgName        HaveToast
  MsgType        REPLY
  OperationCode  OC_TOAST
  Role           5
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // KAREN to microwave: cook bacon
BEGIN
  MsgName        CookBacon
  MsgType        REQUEST
  OperationCode  OC_COOK
  Role           6
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Microwave to KAREN: cooked
BEGIN
  MsgName        HaveBacon
  MsgType        REPLY
  OperationCode  OC_COOK
  Role           6
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END

```

```

MESSAGE          // Message to/from gremlin
BEGIN
  MsgName        GhostMessage
  MsgType        INFORM
  OperationCode  OC_GREMLIN
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Message to/from repair list
BEGIN
  MsgName        RepairMessage
  MsgType        INFORM
  OperationCode  OC_REPAIR
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Message from repair list
BEGIN
  MsgName        InquiryRequest
  MsgType        REQUEST
  OperationCode  OC_INQUIRY
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Message to repair list
BEGIN
  MsgName        InquiryReply
  MsgType        REPLY
  OperationCode  OC_INQUIRY
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // to shopping list: Order more
BEGIN
  MsgName        OrderMessage
  MsgType        INFORM
  OperationCode  OC_ORDER
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
MESSAGE          // Inquiry to/from shopping list
BEGIN
  MsgName        FoodInquiry
  MsgType        INFORM
  OperationCode  OC_FOODINQ
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END

```

```
MESSAGE          // Shopping list to grocer
BEGIN
  MsgName        FoodOrder
  MsgType        INFORM
  OperationCode  OC_FOOD
  Format          VARIABLE
END
MESSAGE          // Grocer to others
BEGIN
  MsgName        FoodDelivery
  MsgType        INFORM
  OperationCode  OC_FOOD
  Format          FIXED
  StrucName      MSG100
  StrucFile      bltstruc.h
  ConversionDLL  MSG100
END
```

A.2 Class Descriptions for The BLT Example

```

File name: classes.ch
/*****
/*
/* External class descriptions for the BLT example
/*
/*
*****/
CLASSDESC          // PL: Konrad
BEGIN
  ClassName    KONRAD
  ClassType    PL
  MsgIn        StartJob, Sandwich, Starve, EatPizza
  MsgOut       FeedMe, DeliverPizza, Show
END
CLASSDESC          // PL: Luigi's pizza place
BEGIN
  ClassName    LUIGI
  ClassType    PL
  MsgIn        DeliverPizza,
  GhostMessage, RepairMessage, InquiryRequest
  MsgOut       EatPizza, InquiryReply
END
CLASSDESC          // BL: Karen
BEGIN
  ClassName    KAREN
  Harden       YES
  ClassType    BL
  MsgIn        FeedMe, HaveFromFridge,
  HaveTomato, HaveBread, HaveToast, HaveBacon,
  GhostMessage, RepairMessage, InquiryRequest,
  MQ_SYSTEM_OWNER_ENDED
  MsgOut       Sandwich, Starve, GetFromFridge,
  GetTomato, GetBread, MakeToast, CookBacon,
  InquiryReply
END
CLASSDESC          // BL: vegetable basket
BEGIN
  ClassName    BASKET
  Harden       YES
  ClassType    BL
  MsgIn        GetTomato, FoodInquiry,
  GhostMessage, RepairMessage, InquiryRequest, FoodDelivery,
  MQ_SYSTEM_OWNER_ENDED
  MsgOut       HaveTomato, InquiryReply, OrderMessage, FoodInquiry
END
CLASSDESC          // BL: bread box
BEGIN
  ClassName    BREADBOX
  Harden       YES
  ClassType    BL
  MsgIn        GetBread, FoodInquiry,
  GhostMessage, RepairMessage, InquiryRequest, FoodDelivery,
  MQ_SYSTEM_OWNER_ENDED
  MsgOut       HaveBread, InquiryReply, OrderMessage, FoodInquiry
END

```

```

CLASSDESC          // BL: refrigerator
BEGIN
  ClassName    FRIDGE
  ClassType    BL
  Harden       Yes
  MsgIn        GetFromFridge, FoodInquiry,
               GhostMessage, RepairMessage, InquiryRequest, FoodDelivery,
               MQ_SYSTEM_OWNER_ENDED
  MsgOut       HaveFromFridge, InquiryReply, OrderMessage, FoodInquiry
END
CLASSDESC          // BL: toaster
BEGIN
  ClassName    TOASTER
  Harden       YES
  ClassType    BL
  MsgIn        MakeToast,
               GhostMessage, RepairMessage, InquiryRequest,
               MQ_SYSTEM_OWNER_ENDED
  MsgOut       HaveToast, InquiryReply
END
CLASSDESC          // BL: microwave
BEGIN
  ClassName    MICRO
  Harden       YES
  ClassType    BL
  MsgIn        CookBacon,
               GhostMessage, RepairMessage, InquiryRequest,
               MQ_SYSTEM_OWNER_ENDED
  MsgOut       HaveBacon, InquiryReply
END
CLASSDESC          // PL: gremlin
BEGIN
  ClassName    GREMLIN
  ClassType    PL
  MsgIn        Show
  MsgOut       GhostMessage
END
CLASSDESC          // PL: repair list
BEGIN
  ClassName    REPAIR
  ClassType    PL
  MsgIn        Show, InquiryReply
  MsgOut       RepairMessage, InquiryRequest
END
CLASSDESC          // PL: shopping list
BEGIN
  ClassName    SHOPPING
  ClassType    PL
  MsgIn        Show, OrderMessage, FoodInquiry, GhostMessage
  MsgOut       FoodInquiry, FoodOrder
END
CLASSDESC          // DL: grocery
BEGIN
  ClassName    GROCER
  ClassType    BL
  Harden       Yes
  MsgIn        FoodOrder,
               GhostMessage, RepairMessage, InquiryRequest,
               MQ_SYSTEM_OWNER_ENDED
  MsgOut       FoodDelivery, InquiryReply
END

```

A.3 Class Source File for BASKET

```

File name: basket.cs
/*****
/*
/* Class Source file for BL: Vegetable Basket
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for vegetable basket"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "messages.ch" // message descriptions
CSINCLUDE "classes.ch" // class descriptions

METHOD
BEGIN
  MethodName Deliver
  MethodType C_LIBRARY
  ProgName basket.Deliver
  SourceName basket1
  MsgOut HaveTomato, OrderMessage
END
METHOD
BEGIN
  MethodName Delivery
  MethodType C_LIBRARY
  ProgName basket.Delivery
  SourceName delivery
END
METHOD
BEGIN
  MethodName Ignore
  MethodType C_LIBRARY
  ProgName basket.Ignore
  SourceName xIgnore
END
METHOD
BEGIN // Gremlin message
  MethodName Gremlin
  MethodType C_LIBRARY
  ProgName basket.Gremlin
  SourceName xGremlin
END
METHOD
BEGIN // repair message
  MethodName Repair
  MethodType C_LIBRARY
  ProgName basket.Repair
  SourceName xRepair
END
METHOD
BEGIN // inquiry message
  MethodName Inquiry
  MethodType C_LIBRARY
  ProgName basket.Inquiry
  SourceName xInquiry
  MsgOut InquiryReply
END

```

```

METHOD
  BEGIN          // inquiry message
    MethodName  FoodInquiry
    MethodType  C_LIBRARY
    ProgName    basket.FoodInquiry
    SourceName  foodinq
    MsgOut      FoodInquiry
  END
METHOD
  BEGIN          // owner ended message
    MethodName  ClearUp
    MethodType  C_LIBRARY
    ProgName    basket.ClearUp
    SourceName  xClear
  END

CLASS           // BL: Vegetable basket
  BEGIN
    ClassType   BL
    ClassName   BASKET
    Destination KAREN, REPAIR, SHOPPING
    Harden      Yes
    PingTimeout 10
  RULE
    BEGIN          // give a tomato
      RuleId     RI_DELIVER1
      RuleName   DeliverRule1
      MethodName Deliver
      State      MATCHSTATE MQSTATE_NEW
      MsgIn      GetTomato
    END
  RULE
    BEGIN          // give a tomato
      RuleId     RI_DELIVER2
      RuleName   DeliverRule2
      MethodName Deliver
      State      MATCHSTATE MQSTATE_CLEAR
      MsgIn      GetTomato
    END
  RULE
    BEGIN          // Gremlin message arrives
      RuleId     RI_GREMLIN
      RuleName   GremlinRule
      MethodName Gremlin
      MsgIn      GhostMessage
    END
  RULE           // INFORM message
    BEGIN
      RuleId     RI_REPAIR1
      RuleName   RepairRule1
      MethodName Repair
      State      MATCHSTATE MQSTATE_DISABLED
      MsgIn      RepairMessage
    END
  RULE           // INFORM message
    BEGIN
      RuleId     RI_REPAIR2
      RuleName   RepairRule2
      MethodName Repair
      State      MATCHSTATE MQSTATE_DISABLED_WHILE_BUSY
      MsgIn      RepairMessage
    END
  END

```

```
RULE          // ignore message when any other state
BEGIN
  RuleId      RI_REPAIR3
  RuleName    RepairRule3
  MethodName  Ignore
  MsgIn       RepairMessage
END
RULE
BEGIN
  RuleId      RI_REPAIR_INQ
  RuleName    InquiryRule
  MethodName  Inquiry
  MsgIn       InquiryRequest
END
RULE
BEGIN
  RuleId      RI_FOOD_INQ
  RuleName    FoodInqRule
  MethodName  FoodInquiry
  MsgIn       FoodInquiry
END
RULE
BEGIN          // more tomatos arrived
  RuleId      RI_FOOD
  RuleName    FoodRule
  MethodName  Delivery
  MsgIn       FoodDelivery
END
RULE
BEGIN
  RuleId      RI_SYS_OE
  RuleName    OwnerEndedRule
  MethodName  ClearUp
  MsgIn       MQ_SYSTEM_OWNER_ENDED
END
END
```


A.4 Class Source File for BREADBOX

```

File name: breadbox.ch
/*****
/*
/* Class Source file for BL: bread box
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for the bread box"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
  MethodName Deliver
  MethodType C_LIBRARY
  ProgName breadbox.Deliver
  SourceName bbox1
  MsgOut HaveBread, OrderMessage
END
METHOD
BEGIN
  MethodName Delivery
  MethodType C_LIBRARY
  ProgName breadbox.Delivery
  SourceName delivery
END
METHOD
BEGIN
  MethodName Ignore
  MethodType C_LIBRARY
  ProgName breadbox.Ignore
  SourceName xIgnore
END
METHOD
BEGIN // Gremlin message
  MethodName Gremlin
  MethodType C_LIBRARY
  ProgName breadbox.Gremlin
  SourceName xGremlin
END
METHOD
BEGIN // repair message
  MethodName Repair
  MethodType C_LIBRARY
  ProgName breadbox.Repair
  SourceName xRepair
END
METHOD
BEGIN // inquiry message
  MethodName Inquiry
  MethodType C_LIBRARY
  ProgName breadbox.Inquiry
  SourceName xInquiry
  MsgOut InquiryReply
END

```

```

METHOD
  BEGIN          // inquiry message
    MethodName  FoodInquiry
    MethodType  C_LIBRARY
    ProgName    breadbox.FoodInquiry
    SourceName  foodinq
    MsgOut      FoodInquiry
  END
METHOD
  BEGIN          // owner ended message
    MethodName  ClearUp
    MethodType  C_LIBRARY
    ProgName    breadbox.ClearUp
    SourceName  xClear
  END

CLASS           // BL: bread box
  BEGIN
    ClassType   BL
    ClassName   BREADBOX
    Destination KAREN, REPAIR, SHOPPING
    Harden      Yes
    PingTimeout 10
  RULE
    BEGIN       // request for bread
      RuleId    RI_BREAD1
      RuleName  BreadRule1
      MethodName Deliver
      State     MATCHSTATE MQSTATE_NEW
      MsgIn     GetBread
    END
  RULE
    BEGIN       // request for bread
      RuleId    RI_BREAD2
      RuleName  BreadRule2
      MethodName Deliver
      State     MATCHSTATE MQSTATE_CLEAR
      MsgIn     GetBread
    END
  RULE
    BEGIN       // Gremlin message arrives
      RuleId    RI_GREMLIN
      RuleName  GremlinRule
      MethodName Gremlin
      MsgIn     GhostMessage
    END
  RULE
    BEGIN
      RuleId    RI_REPAIR1
      RuleName  RepairRule1
      MethodName Repair
      State     MATCHSTATE MQSTATE_DISABLED
      MsgIn     RepairMessage
    END
  RULE
    BEGIN
      RuleId    RI_REPAIR2
      RuleName  RepairRule2
      MethodName Repair
      State     MATCHSTATE MQSTATE_DISABLED_WHILE_BUSY
      MsgIn     RepairMessage
    END
  END

```

```
RULE
  BEGIN
    RuleId    RI_REPAIR3
    RuleName  RepairRule3
    MethodName Ignore
    MsgIn     RepairMessage
  END
RULE
  BEGIN
    RuleId    RI_REPAIR_INQ
    RuleName  InquiryRule
    MethodName Inquiry
    MsgIn     InquiryRequest
  END
RULE
  BEGIN
    RuleId    RI_FOOD_INQ
    RuleName  FoodInqRule
    MethodName FoodInquiry
    MsgIn     FoodInquiry
  END
RULE
  BEGIN                                // bread delivery
    RuleId    RI_FOOD
    RuleName  FoodRule
    MethodName Delivery
    MsgIn     FoodDelivery
  END
RULE
  BEGIN
    RuleId    RI_SYS_OE
    RuleName  OwnerEndedRule
    MethodName ClearUp
    MsgIn     MQ_SYSTEM_OWNER_ENDED
  END
END
```

A.5 Class Source File for FRIDGE

```

File name: fridge.ch
/*****
/*
/* Class Source file for BL: Refrigerator
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for refrigerator"
END

CSINCLUDE "bmqsyms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
  BEGIN
    MethodName Deliver
    MethodType C_LIBRARY
    ProgName fridge.Deliver
    SourceName fridge1
    MsgOut HaveFromFridge, OrderMessage
  END
METHOD
  BEGIN
    MethodName Delivery
    MethodType C_LIBRARY
    ProgName fridge.Delivery
    SourceName delivery
  END
METHOD
  BEGIN
    MethodName Ignore
    MethodType C_LIBRARY
    ProgName fridge.Ignore
    SourceName xignore
  END
METHOD
  BEGIN // Gremlin message
    MethodName Gremlin
    MethodType C_LIBRARY
    ProgName fridge.Gremlin
    SourceName xGremlin
  END
METHOD
  BEGIN // repair message
    MethodName Repair
    MethodType C_LIBRARY
    ProgName fridge.Repair
    SourceName xrepair
  END
METHOD
  BEGIN // inquiry message
    MethodName Inquiry
    MethodType C_LIBRARY
    ProgName fridge.Inquiry
    SourceName xinquiry
    MsgOut InquiryReply
  END
END

```

```

METHOD
  BEGIN          // inquiry message
    MethodName  FoodInquiry
    MethodType  C_LIBRARY
    ProgName    fridge.FoodInquiry
    SourceName  foodinq
    MsgOut      FoodInquiry
  END
METHOD
  BEGIN          // owner ended message
    MethodName  ClearUp
    MethodType  C_LIBRARY
    ProgName    fridge.ClearUp
    SourceName  xClear
  END

CLASS           // BL: Microwave
  BEGIN
    ClassType   BL
    ClassName   FRIDGE
    Destination KAREN, REPAIR, SHOPPING
    Harden      Yes
    PingTimeout 10
  RULE
    BEGIN          // deliver 3 items
      RuleId      RI_FRIDGE1
      RuleName    FridgeRule1
      MethodName  Deliver
      State       MATCHSTATE MQSTATE_NEW
      MsgIn       GetFromFridge
    END
  RULE
    BEGIN          // deliver 3 things
      RuleId      RI_FRIDGE2
      RuleName    FridgeRule2
      MethodName  Deliver
      State       MATCHSTATE MQSTATE_CLEAR
      MsgIn       GetFromFridge
    END
  RULE
    BEGIN          // Gremlin message arrives
      RuleId      RI_GREMLIN
      RuleName    GremlinRule
      MethodName  Gremlin
      MsgIn       GhostMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR1
      RuleName    RepairRule1
      MethodName  Repair
      State       MATCHSTATE MQSTATE_DISABLED
      MsgIn       RepairMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR2
      RuleName    RepairRule2
      MethodName  Repair
      State       MATCHSTATE MQSTATE_DISABLED_WHILE_BUSY
      MsgIn       RepairMessage
    END
  END

```

```
RULE
  BEGIN
    RuleId      RI_REPAIR3
    RuleName    RepairRule3
    MethodName  Ignore
    MsgIn       RepairMessage
  END
RULE
  BEGIN
    RuleId      RI_REPAIR_INQ
    RuleName    InquiryRule
    MethodName  Inquiry
    MsgIn       InquiryRequest
  END
RULE
  BEGIN
    RuleId      RI_FOOD_INQ
    RuleName    FoodInqRule
    MethodName  FoodInquiry
    MsgIn       FoodInquiry
  END
RULE
  BEGIN                                // more food is delivered
    RuleId      RI_FOOD
    RuleName    FoodRule
    MethodName  Delivery
    MsgIn       FoodDelivery
  END
RULE
  BEGIN
    RuleId      RI_SYS_OE
    RuleName    OwnerEndedRule
    MethodName  ClearUp
    MsgIn       MQ_SYSTEM_OWNER_ENDED
  END
END
```

A.6 Class Source File for GREMLIN

```

File name: gremlin.ch
/*****
/*
/* Class Source file for PL: GREMLIN
/*
/*****
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
    Title "Class File for Gremlin"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
    MethodName TheMethod
    MethodType PROGRAM
    ProgName gremlin.exe
    StartupTime 10
    Interface PULL
    MsgOut GhostMessage
END

CLASS // PL: GREMLIN
BEGIN
    ClassType PL
    ClassName "GREMLIN"
    Destination "BASKET", "BREADBOX", "FRIDGE", "TOASTER", "MICRO",
                "LUIGI", "KAREN", "GROCER", "SHOPPING"

    RULE
    BEGIN // start Gremlin
        RuleId RI_SHOW
        RuleName ShowRule
        MethodName TheMethod
        MsgIn Show
    END
END

```

A.7 Class Source File for GROCER

```

File name: grocer.ch
/*****
/*
/* Class Source file for BL: Grocer
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
    Title "Class File for grocer"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
    MethodName SellFood
    MethodType C_LIBRARY
    ProgName   grocer.SellFood
    SourceName grocer1
    MsgOut     FoodDelivery
END
METHOD
BEGIN // Gremlin message
    MethodName Gremlin
    MethodType C_LIBRARY
    ProgName   grocer.Gremlin
    SourceName xGremlin
END
METHOD
BEGIN // repair message
    MethodName Repair
    MethodType C_LIBRARY
    ProgName   grocer.Repair
    SourceName xRepair
END
METHOD
BEGIN // inquiry message
    MethodName Inquiry
    MethodType C_LIBRARY
    ProgName   grocer.Inquiry
    SourceName xInquiry
    MsgOut     InquiryReply
END
METHOD
BEGIN // owner ended message
    MethodName ClearUp
    MethodType C_LIBRARY
    ProgName   grocer.ClearUp
    SourceName xClear
END

```



```

CLASS          // BL: Grocer
BEGIN
  ClassType    BL
  ClassName    GROCER
  Destination  REPAIR, FRIDGE, BASKET, BREADBOX
  Harden       Yes
  PingTimeout  10
  RULE
    BEGIN          // Order arrives
      RuleId      RI_SELL
      RuleName    SellRule
      MethodName  SellFood
      State       NOTMATCHSTATE MQSTATE_DISABLED
      MsgIn       FoodOrder
    END
  RULE
    BEGIN          // Gremlin message arrives
      RuleId      RI_GREMLIN
      RuleName    GremlinRule
      MethodName  Gremlin
      MsgIn       GhostMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR1
      RuleName    RepairRule1
      MethodName  Repair
      MsgIn       RepairMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR_INQ
      RuleName    InquiryRule
      MethodName  Inquiry
      MsgIn       InquiryRequest
    END
  RULE
    BEGIN
      RuleId      RI_SYS_OE
      RuleName    OwnerEndedRule
      MethodName  ClearUp
      MsgIn       MQ_SYSTEM_OWNER_ENDED
    END
END

```

A.8 Class Source File for KAREN

```

File name: karen.ch
/*****
/*
/* Class Source file for BL: KAREN
/*
/*****
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for KAREN - Main BL"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
  BEGIN // order arrived
    MethodName Sandwich
    MethodType C_LIBRARY
    ProgName karen.Sandwich
    SourceName bltorder
    MsgOut Starve, GetTomato, GetBread, GetFromFridge
  END
METHOD
  BEGIN // material arrives (all or partially)
    MethodName MakeBLT
    MethodType C_LIBRARY
    ProgName karen.MakeBLT
    SourceName bltmake
    MsgOut MakeToast, CookBacon, Starve
  END
METHOD
  BEGIN // all material cooked
    MethodName ServeBLT
    MethodType C_LIBRARY
    ProgName karen.ServeBLT
    SourceName bltserve
    MsgOut Sandwich
  END
METHOD
  BEGIN // some material cooked (timeout)
    MethodName NoBLT
    MethodType C_LIBRARY
    ProgName karen.NoBLT
    SourceName bltnone
    MsgOut Starve
  END
METHOD
  BEGIN
    MethodName Ignore
    MethodType C_LIBRARY
    ProgName karen.Ignore
    SourceName xignore
  END

```

```

METHOD
  BEGIN          // Gremlin message
    MethodName Gremlin
    MethodType  C_LIBRARY
    ProgName    karen.Gremlin
    SourceName  xGremlin
  END
METHOD
  BEGIN          // repair message
    MethodName  Repair
    MethodType  C_LIBRARY
    ProgName    karen.Repair
    SourceName  xRepair
  END
METHOD
  BEGIN          // inquiry message
    MethodName  Inquiry
    MethodType  C_LIBRARY
    ProgName    karen.Inquiry
    SourceName  xInquiry
    MsgOut      InquiryReply
  END
METHOD
  BEGIN          // owner ended message
    MethodName  ClearUp
    MethodType  C_LIBRARY
    ProgName    karen.ClearUp
    SourceName  xClear
  END

CLASS           // BL: KAREN
  BEGIN
    ClassType   BL
    ClassName   KAREN
    Destination KONRAD, BREADBOX, FRIDGE, MICRO, TOASTER, BASKET, REPAIR
    Harden      Yes
    PingTimeout 10
  RULE
    BEGIN          // first BLT request
      RuleId      RI_SANDWICH1
      RuleName    SandwichRule1
      MethodName  Sandwich
      State       MATCHSTATE MQSTATE_NEW
      MsgIn       FeedMe
    END
  RULE
    BEGIN          // next BLT request
      RuleId      RI_SANDWICH2
      RuleName    SandwichRule2
      MethodName  Sandwich
      State       MATCHSTATE MQSTATE_CLEAR
      MsgIn       FeedMe
    END
  RULE
    BEGIN          // Gremlin message arrives
      RuleId      RI_GREMLIN
      RuleName    GremlinRule
      MethodName  Gremlin
      MsgIn       GhostMessage
    END

```

```

RULE          // INFORM message
BEGIN
  RuleId      RI_REPAIR1
  RuleName    RepairRule1
  MethodName  Repair
  State       MATCHSTATE MQSTATE_DISABLED
  MsgIn       RepairMessage
END
RULE          // INFORM message
BEGIN
  RuleId      RI_REPAIR2
  RuleName    RepairRule2
  MethodName  Repair
  State       MATCHSTATE MQSTATE_DISABLED_WHILE_BUSY
  MsgIn       RepairMessage
END
RULE          // ignore message when any other state
BEGIN
  RuleId      RI_REPAIR3
  RuleName    RepairRule3
  MethodName  Ignore
  MsgIn       RepairMessage
END
RULE          // reply when in any state
BEGIN
  RuleId      RI_REPAIR_INQ
  RuleName    InquiryRule
  MethodName  Inquiry
  MsgIn       InquiryRequest
END
RULE
BEGIN          // material arrives in time
  RuleId      RI_MAKEBLT1
  RuleName    MakeRule1
  MethodName  MakeBLT
  MsgIn       HaveTomato, HaveBread, HaveFromFridge
END
RULE
BEGIN          // material arrives incomplete (timeout)
  RuleId      RI_MAKEBLT2
  RuleName    MakeRule2
  MethodName  MakeBLT
  Timed       Yes
  MsgIn       HaveTomato    PLACEHOLDER,
              HaveBread     PLACEHOLDER,
              HaveFromFridge PLACEHOLDER
END
RULE
BEGIN          // tomato arrives late
  RuleId      RI_TOMATO
  RuleName    TomatoRule
  MethodName  Ignore
  MsgIn       HaveTomato    LATE
END
RULE
BEGIN          // bread arrives late
  RuleId      RI_BREAD
  RuleName    BreadRule
  MethodName  Ignore
  MsgIn       HaveBread     LATE
END

```

```

RULE
  BEGIN                                // fridge delivers late
    RuleId    RI_FRIDGE
    RuleName   FridgeRule
    MethodName Ignore
    MsgIn     HaveFromFridge LATE
  END
RULE
  BEGIN                                // BLT is ready to serve
    RuleId    RI_SERVEBLT1
    RuleName   ServeRule1
    MethodName ServeBLT
    MsgIn     HaveToast, HaveBacon
  END
RULE
  BEGIN                                // toast / bacon incomplete (timeout)
    RuleId    RI_SERVEBLT2
    RuleName   ServeBLT
    MethodName NoBLT
    Timed     Yes
    MsgIn     HaveToast  PLACEHOLDER,
              HaveBacon  PLACEHOLDER
  END
RULE
  BEGIN                                // toast arrives late
    RuleId    RI_TOAST
    RuleName   ToastRule
    MethodName Ignore
    MsgIn     HaveToast LATE
  END
RULE
  BEGIN                                // cooked bacon arrives late
    RuleId    RI_BACON
    RuleName   BaconRule
    MethodName Ignore
    MsgIn     HaveBacon LATE
  END
RULE
  BEGIN
    RuleId    RI_SYS_OE
    RuleName   OwnerEndedRule
    MethodName ClearUp
    MsgIn     MQ_SYSTEM_OWNER_ENDED
  END
END

```

A.9 Class Source File for KONRAD

```

File name: konrad.ch
/*****
/*
/* Class Source file for PL: KONRAD
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
    Title "Class File for KONRAD"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
    MethodName BLTMethod
    MethodType PROGRAM
    ProgName konrad.exe
    StartupTime 10
    Interface PULL
    MsgOut FeedMe, DeliverPizza, Show
END
METHOD
BEGIN
    MethodName PizzaMethod
    MethodType PROGRAM
    ProgName konrad.exe
    StartupTime 10
    Interface PULL
END

CLASS // PL for KONRAD
BEGIN
    ClassType PL
    ClassName KONRAD
    Destination KAREN, LUIGI, GREMLIN, REPAIR, SHOPPING
    RULE
        BEGIN // startjob
            RuleId RI_STARTJOB
            RuleName StartJobRule
            MethodName BLTMethod
            MsgIn StartJob
        END
    RULE
        BEGIN // BLT arrives on time
            RuleId RI_SANDWICH1
            RuleName SandwichRule1
            MethodName BLTMethod
            MsgIn Sandwich
        END
    RULE
        BEGIN // timer for BLT expired
            RuleId RI_SANDWICH2
            RuleName SandwichRule2
            MethodName BLTMethod
            MsgIn Sandwich PLACEHOLDER
            Timed YES
        END
    END
END

```

```

RULE
  BEGIN          // BLT arrives late
    RuleId      RI_SANDWICH3
    RuleName    SandwichRule3
    MethodName  BLTMethod
    MsgIn      Sandwich LATE
  END
RULE
  BEGIN          // cannot make a BLT
    RuleId      RI_STARVE
    RuleName    StarveRule
    MethodName  BLTMethod
    MsgIn      Starve
  END
RULE
  BEGIN          // pizza arrives on time
    RuleId      RI_PIZZA1
    RuleName    PizzaRule1
    MethodName  PizzaMethod
    MsgIn      EatPizza
  END
RULE
  BEGIN          // timer for pizza expired
    RuleId      RI_PIZZA2
    RuleName    PizzaRule2
    MethodName  PizzaMethod
    MsgIn      EatPizza PLACEHOLDER
    Timed      YES
  END
RULE
  BEGIN          // pizza arrives late
    RuleId      RI_PIZZA3
    RuleName    PizzaRule3
    MethodName  PizzaMethod
    MsgIn      EatPizza LATE
  END
END

```

A.10 Class Source File for LUIGI

```

File name: luigi.ch

/*****
/*
/* Class Source file for PL: LUIGI Pizza Place
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for LUIGI's Pizza Place"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
  MethodName TheMethod
  MethodType PROGRAM
  ProgName luigi.exe
  StartupTime 10
  Interface PULL
  MsgOut EatPizza, InquiryReply
END

CLASS // PL: LUIGI's pizza place
BEGIN
  ClassType PL
  ClassName "LUIGI"
  Destination "KONRAD", "REPAIR"
  RULE
  BEGIN // first pizza order arrives
    RuleId RI_PIZZA1
    RuleName LuigiRule1
    MethodName TheMethod
    State MATCHSTATE MQSTATE_NEW
    MsgIn DeliverPizza
  END
  RULE
  BEGIN // next pizza order arrives
    RuleId RI_PIZZA2
    RuleName LuigiRule2
    MethodName TheMethod
    State MATCHSTATE MQSTATE_CLEAR
    MsgIn DeliverPizza
  END
  RULE
  BEGIN // Gremlin message arrives
    RuleId RI_GREMLIN
    RuleName GremlinRule
    MethodName TheMethod
    MsgIn GhostMessage
  END
END

```



```

RULE
  BEGIN          // repair message arrives
    RuleId      RI_REPAIR1
    RuleName    RepairRule1
    MethodName  TheMethod
    State       MATCHSTATE MQSTATE_DISABLED
    MsgIn       RepairMessage
  END
RULE
  BEGIN          // repair message arrives
    RuleId      RI_REPAIR2
    RuleName    RepairRule2
    MethodName  TheMethod
    State       MATCHSTATE MQSTATE_DISABLED_WHILE_BUSY
    MsgIn       RepairMessage
  END
RULE            // in all other states ignore the message
  BEGIN
    RuleId      RI_REPAIR3
    RuleName    RepairRule3
    MethodName  TheMethod
    MsgIn       RepairMessage
  END
RULE            // inquiry request message arrives
  BEGIN
    RuleId      RI_REPAIR_INQ
    RuleName    RepairInqRule
    MethodName  TheMethod
    MsgIn       InquiryRequest
  END
END

```

A.11 Class Source File for MICRO

```

File name: micro.ch
/*****
/*
/* Class Source file for BL: Microwave
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
    Title "Class File for microwave"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
    MethodName Cook
    MethodType C_LIBRARY
    ProgName micro.Cook
    SourceName cook
    MsgOut HaveBacon
END
METHOD
BEGIN // Gremlin message
    MethodName Gremlin
    MethodType C_LIBRARY
    ProgName micro.Gremlin
    SourceName xGremlin
END
METHOD
BEGIN // repair message
    MethodName Repair
    MethodType C_LIBRARY
    ProgName micro.Repair
    SourceName xRepair
END
METHOD
BEGIN // inquiry message
    MethodName Inquiry
    MethodType C_LIBRARY
    ProgName micro.Inquiry
    SourceName xInquiry
    MsgOut InquiryReply
END
METHOD
BEGIN // owner ended message
    MethodName ClearUp
    MethodType C_LIBRARY
    ProgName micro.ClearUp
    SourceName xClear
END

```

```

CLASS          // BL: Microwave
BEGIN
  ClassType    BL
  ClassName    MICRO
  Destination  KAREN, REPAIR
  Harden       Yes
  PingTimeout  10
  RULE
    BEGIN          // raw bacon arrives
      RuleId      RI_COOK
      RuleName    CookRule
      MethodName  Cook
      State       NOTMATCHSTATE MQSTATE_DISABLED
      MsgIn       CookBacon
    END
  RULE
    BEGIN          // Gremlin message arrives
      RuleId      RI_GREMLIN
      RuleName    GremlinRule
      MethodName  Gremlin
      MsgIn       GhostMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR1
      RuleName    RepairRule1
      MethodName  Repair
      MsgIn       RepairMessage
    END
  RULE
    BEGIN
      RuleId      RI_REPAIR_INQ
      RuleName    InquiryRule
      MethodName  Inquiry
      MsgIn       InquiryRequest
    END
  RULE
    BEGIN
      RuleId      RI_SYS_OE
      RuleName    OwnerEndedRule
      MethodName  ClearUp
      MsgIn       MQ_SYSTEM_OWNER_ENDED
    END
END

```

A.12 Class Source File for REPAIR

```

File name: repair.ch
/*****
/*
/* Class Source file for PL: REPAIR
/*
/*
/*****
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
    Title "Class File for repairman"
END
CSINCLUDE "bmqsyms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
    MethodName TheMethod
    MethodType PROGRAM
    ProgName repair.exe
    StartupTime 10
    Interface PULL
    MsgOut RepairMessage, InquiryRequest
END

CLASS // PL: REPAIR
BEGIN
    ClassType PL
    ClassName "REPAIR"
    Destination "BASKET", "BREADBOX", "FRIDGE", "TOASTER", "MICRO",
                "LUIGI", "KAREN", "GROCER"

    RULE
        BEGIN // start program
            RuleId RI_SHOW
            RuleName ShowRule
            MethodName TheMethod
            MsgIn Show
        END
    RULE
        BEGIN // repair response
            RuleId RI_REPAIR_INQ
            RuleName RepairRule1
            MethodName TheMethod
            MsgIn InquiryReply
        END
    RULE
        BEGIN // no repair response
            RuleId RI_REPAIR_NO
            RuleName RepairRule2
            MethodName TheMethod
            MsgIn InquiryReply PLACEHOLDER
            Timed YES
        END
    RULE
        BEGIN // late repair response
            RuleId RI_REPAIR_LATE
            RuleName RepairRule3
            MethodName TheMethod
            MsgIn InquiryReply LATE
        END
END
END

```

A.13 Class Source File for SHOPPING

```

File name: shopping.ch
/*****
/*
/* Class Source file for PL: Shopping list
/*
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for shopping list"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
  MethodName TheMethod
  MethodType PROGRAM
  ProgName shopping.exe
  StartupTime 10
  Interface PULL
  MsgOut FoodInquiry, FoodOrder
END

CLASS
BEGIN
  ClassType PL
  ClassName SHOPPING
  Destination BASKET, FRIDGE, BREADBOX, GROCER
  RULE
  BEGIN // start program
    RuleId RI_SHOW
    RuleName ShowRule
    MethodName TheMethod
    MsgIn Show
  END
  RULE
  BEGIN
    RuleId RI_ORDER
    RuleName OrderRule
    MethodName TheMethod
    MsgIn OrderMessage
  END
  RULE
  BEGIN
    RuleId RI_FOOD_INQ
    RuleName FoodRule
    MethodName TheMethod
    MsgIn FoodInquiry
  END
  RULE
  BEGIN
    RuleId RI_TIMER
    RuleName TimerRule
    MethodName TheMethod
    Timed YES
  END
END

```

```
RULE
  BEGIN          // Gremlin message arrives
    RuleId      RI_GREMLIN
    RuleName    GremlinRule
    MethodName  TheMethod
    MsgIn       GhostMessage
  END
END
```

A.14 Class Source File for TOASTER

```

File name: toaster.ch

/*****
/*
/* Class Source file for BL: toaster
/*
*****/
#include <bmqc.h>
#include "bltdef.h"

HEADING
BEGIN
  Title "Class File for the toaster"
END

CSINCLUDE "bmqsysms.ch" // 3T system message descriptions
CSINCLUDE "Messages.ch" // message descriptions
CSINCLUDE "Classes.ch" // class descriptions

METHOD
BEGIN
  MethodName Cook
  MethodType C_LIBRARY
  ProgName toaster.Cook
  SourceName cook
  MsgOut HaveToast
END
METHOD
BEGIN // Gremlin message
  MethodName Gremlin
  MethodType C_LIBRARY
  ProgName toaster.Gremlin
  SourceName xGremlin
END
METHOD
BEGIN // repair message
  MethodName Repair
  MethodType C_LIBRARY
  ProgName toaster.Repair
  SourceName xRepair
END
METHOD
BEGIN // inquiry message
  MethodName Inquiry
  MethodType C_LIBRARY
  ProgName toaster.Inquiry
  SourceName xInquiry
  MsgOut InquiryReply
END
METHOD
BEGIN // owner ended message
  MethodName ClearUp
  MethodType C_LIBRARY
  ProgName toaster.ClearUp
  SourceName xClear
END

CLASS // BL: toaster
BEGIN
  ClassType BL
  ClassName TOASTER
  Destination KAREN, REPAIR
  Harden Yes
  PingTimeout 10

```

```
RULE
  BEGIN                                // bread arrives
    RuleId    RI_COOK
    RuleName   CookRule
    MethodName Cook
    State      NOTMATCHSTATE MQSTATE_DISABLED
    MsgIn      MakeToast
  END
RULE
  BEGIN                                // Gremlin message arrives
    RuleId    RI_GREMLIN
    RuleName   GremlinRule
    MethodName Gremlin
    MsgIn      GhostMessage
  END
RULE
  BEGIN
    RuleId    RI_REPAIR1
    RuleName   RepairRule1
    MethodName Repair
    MsgIn      RepairMessage
  END
RULE
  BEGIN
    RuleId    RI_REPAIR_INQ
    RuleName   InquiryRule
    MethodName Inquiry
    MsgIn      InquiryRequest
  END
RULE
  BEGIN
    RuleId    RI_SYS_OE
    RuleName   OwnerEndedRule
    MethodName ClearUp
    MsgIn      MQ_SYSTEM_OWNER_ENDED
  END
END
```


A.15 Definitions for Class Source Files

File name: bltdef.h

```

/*****
/*
/* Definitions for PL and BL
/*
/*****
/*      User states for rules
/*****
#define MQSTATE_BUSY          30
#define MQSTATE_DISABLED      31
#define MQSTATE_DISABLED_WHILE_BUSY 33
/*****
/*      Rule IDs for classes
/*****
#define RI_STARTJOB          0  /* start job owner KONRAD          */
#define RI_SHOW              1  /* KONRAD: initiate other PLs      */
/* PLs : shows GUI
#define RI_TIMER            5  /* SHOP : invokes timer method    */

#define RI_SANDWICH1        11  /* KONRAD: BLT arrives on time     */
/* KAREN : BLT order arrives, state=NEW
#define RI_SANDWICH2        12  /* KONRAD: BLT does not arrive, timed out */
/* KAREN : BLT order arrives, state=CLEAR
#define RI_SANDWICH3        13  /* KONRAD: BLT arrives late
#define RI_PIZZA1           15  /* KONRAD: pizza arrives on time   */
/* LUIGI : pizza order arrives, state=NEW
#define RI_PIZZA2           16  /* KONRAD: pizza does not arrive, timed out */
/* LUIGI : pizza order arrives, state=CLEAR
#define RI_PIZZA3           17  /* KONRAD: pizza arrives late
#define RI_STARVE           19  /* KONRAD: BLT cannot be made

#define RI_MAKEBLT1         20  /* KAREN: all material arrived in time */
#define RI_MAKEBLT2         21  /* KAREN: material arrived incomplete */
#define RI_TOMATO           22  /* KAREN: tomato arrived late
#define RI_BREAD            23  /* KAREN: bread arrived late
#define RI_FRIDGE           24  /* KAREN: items from fridge arrived late */
#define RI_SERVEBLT1        25  /* KAREN: everthing cooked in time
#define RI_SERVEBLT2        26  /* KAREN: not everthing cooked in time
#define RI_TOAST            27  /* KAREN: toast arrived late
#define RI_BACON            28  /* KAREN: bacon arrived late

#define RI_DELIVER1         30  /* BASKET: tomato requested, state=NEW */
#define RI_DELIVER2         31  /* BASKET: tomato requested, state=CLEAR */
#define RI_BREAD1           32  /* BBOX : bread requested, state=NEW
#define RI_BREAD2           33  /* BBOX : bread requested, state=CLEAR
#define RI_FRIDGE1          34  /* FRIDGE: items requested, state=NEW
#define RI_FRIDGE2          35  /* FRIDGE: items requested, state=CLEAR

#define RI_COOK             40  /* MICRO : received bacon to cook
/* TOASTER: received bread to toast

#define RI_GREMLIN          50  /* all: disable the instance
#define RI_REPAIR1          51  /* all: enable when state=DISABLED
#define RI_REPAIR2          52  /* all: enable when state=BUSY+DISABLED
#define RI_REPAIR3          53  /* all: ignore message
#define RI_REPAIR_INQ       55  /* REPAIR: reply to inquiry arrived
/* others: inquiry request arrived
#define RI_REPAIR_NO        56  /* REPAIR: no response to inquiry (timeout)
#define RI_REPAIR_LATE      57  /* REPAIR: late response to inquiry

```

```

#define RI_ORDER      60  /* SHOP : received item to put on list */
#define RI_FOOD_INQ  61  /* SHOP : received response to inquiry */
                        /* FRIDGE: received food inquiry request */
                        /* BASKET: same */
                        /* BBOX : same */
#define RI_FOOD      62  /* FRIDGE: food items are delivered */
                        /* BASKET: same */
                        /* BBOX : same */
#define RI_SELL      63  /* GROCER: message to deliver food arrived */
#define RI_SYS_OE    111 /* BLs : Konrad ended */
/*****
/*      Operation codes for messages      */
*****/
#define OC_STARTJOB  (MQOC_USER)
#define OC_SHOW      (MQOC_USER + 1)
#define OC_SANDWICH  (MQOC_USER + 2)
#define OC_STARVE    (MQOC_USER + 3)
#define OC_PIZZA     (MQOC_USER + 4)
#define OC_TOMATO    (MQOC_USER + 5)
#define OC_BREAD     (MQOC_USER + 6)
#define OC_FRIDGE    (MQOC_USER + 7)
#define OC_TOAST     (MQOC_USER + 8)
#define OC_COOK      (MQOC_USER + 9)
#define OC_GREMLIN   (MQOC_USER + 10)
#define OC_REPAIR    (MQOC_USER + 11)
#define OC_INQUIRY   (MQOC_USER + 12)
#define OC_ORDER     (MQOC_USER + 13)
#define OC_FOODINQ   (MQOC_USER + 14)
#define OC_FOOD      (MQOC_USER + 15)
*****/

```

A.16 Definitions for Visual Basic

File name: bltdef.bas

```

' *****
'
'   Definitions for PL and BL
'
' *****
'       User states for rules
' *****
Global Const MQSTATE_BUSY = 30&
Global Const MQSTATE_DISABLED = 31&
Global Const MQSTATE_DISABLED_WHILE_BUSY = 33&
' *****
'       Rule IDs for classes
' *****
Global Const RI_STARTJOB = 0&      ' start job owner KONRAD      */
Global Const RI_SHOW = 1&         ' KONRAD: initiate other PLs  */
                                ' PLs : shows GUI              */
Global Const RI_TIMER = 5&        ' SHOP : invokes timer method */

Global Const RI_SANDWICH1 = 11&   ' KONRAD: BLT arrives on time  */
                                ' KAREN : BLT order arrives, state=NEW */
Global Const RI_SANDWICH2 = 12&   ' KONRAD: BLT does not arrive, timed out */
                                ' KAREN : BLT order arrives, state=CLEAR */
Global Const RI_SANDWICH3 = 13&   ' KONRAD: BLT arrives late    */
Global Const RI_PIZZA1 = 15&      ' KONRAD: pizza arrives on time */
                                ' LUIGI : pizza order arrives, state=NEW */
Global Const RI_PIZZA2 = 16&      ' KONRAD: pizza does not arrive, timed out */
                                ' LUIGI : pizza order arrives, state=CLEAR */
Global Const RI_PIZZA3 = 17&      ' KONRAD: pizza arrives late   */
Global Const RI_STARVE = 19&      ' KONRAD: BLT cannot be made   */

Global Const RI_GREMLIN = 50&     ' all:  disable the instance   */
Global Const RI_REPAIR1 = 51&     ' all:  enable when state=DISABLED */
Global Const RI_REPAIR2 = 52&     ' all:  enable when state=BUSY+DISABLED */
Global Const RI_REPAIR3 = 53&     ' all:  ignore message        */
Global Const RI_REPAIR_INQ = 55&   ' REPAIR: reply to inquiry arrived */
                                ' others: inquiry request arrived */
Global Const RI_REPAIR_NO = 56&   ' REPAIR: no response to inquiry (timeout) */
Global Const RI_REPAIR_LATE = 57& ' REPAIR: late response to inquiry */
Global Const RI_ORDER = 60        ' SHOP:  received item to put on list
Global Const RI_FOOD_INQ = 61     ' SHOP:  received response to inquiry
' *****
' Global variables
' *****
Global vPLClass As String * 12    ' PL class name
Global vHInst As Long             ' instance handle

Type MSG100
    message As String * 20
    number As Long
    value As Long
    filler As String * 72
End Type

```


Appendix B. Summary of MQ3T APIs

Table 26 (Page 1 of 6). MQSeries 3T APIs for Visual Basic

API	Parameter	Declared	Description
MQADDB	ByVal HSet	Long	Set handle
	ByVal BufferLength	Long	Length (in bytes) of the Buffer area
	Buffer	Any	Buffer that holds the element
	ByVal Replace	Long	Replace existing element
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQADDC	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	ByVal StringLength	Long	Length of the character-string data
	pString	String	The character-string data
	ByVal Replace	Long	Replace existing element
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQADDI	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	ByVal XInteger	Long	The integer data
	ByVal Replace	Long	Replace existing element
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCMPB	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	ByVal BufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	Buffer for comparison
	pResult	Long	Result of comparison
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCMPC	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	ByVal StringLength	Long	Length (in bytes) of the string
	ByVal pString	Long	Character string for comparison
	pResult	Long	Result of comparison
	pCompCode	Long	Completion code
	pReason	Long	Reason code

<i>Table 26 (Page 2 of 6). MQSeries 3T APIs for Visual Basic</i>			
API	Parameter	Declared	Description
MQCMPE	ByVal HSet1	Long	Handle of the first set
	ByVal HSet2	Long	Handle of the second set
	ByVal ElementId1	Long	ID of element in first set
	ByVal ElementId2	Long	ID of element in second set
	pResult	Long	Result of comparison
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCMPI	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	ByVal XInteger	Long	The integer
	pResult	Long	Result of comparison
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCPYB	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	pBufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	The buffer to which the element is to be copied
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCPYC	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	pStringLength	Long	Length (in bytes) of the string
	ByVal pString	Long	The buffer to which the element is to be copied
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCPYE	ByVal HSetSource	Long	Handle of source set
	ByVal HSetDestination	Long	Handle of destination set
	ByVal SourceElement	Long	ID of source element
	ByVal DestinationElement	Long	ID of destination element
	ByVal Replace	Long	Replace existing element
	pCompCode	Long	Completion code
	pReason	Long	Reason code

<i>Table 26 (Page 3 of 6). MQSeries 3T APIs for Visual Basic</i>			
API	Parameter	Declared	Description
MQCPYI	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	ID of the element
	pInteger	Long	Integer copied from the element
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQCRTS	pHSet	Long	Set handle
	ByVal SetLength	Long	Initial length (in bytes) of the set
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQDELA	ByVal HSet	Long	Set handle
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQDELE	ByVal HSet	Long	Set handle
	ByVal ElementId	Long	Identifier of the element
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQDELS	ByVal HSet	Long	Set handle
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQENDE	ByVal HInst	Long	Instance handle
	ByVal InstanceState	Long	Instance state
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQENDP	ByVal Options	Long	Options
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQLOG	ByVal HInst	Long	Instance handle
	ByVal LogLevel	Long	Logging level for the log message
	ByVal BufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	Buffer that holds the log message
	ByVal InsertCount	Long	Number of insert strings
	pInsertTable	Long	Table of pointers to insert strings
	pCompCode	Long	Completion code
	pReason	Long	Reason code

<i>Table 26 (Page 4 of 6). MQSeries 3T APIs for Visual Basic</i>			
API	Parameter	Declared	Description
MQQRY	ByVal HInst	Long	Instance handle
	pClassName	String	Class name
	ByVal QueryType	Long	Type of query
	pBufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	The buffer to which the information is to be copied
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQQRYE	ByVal HInst	Long	Instance handle
	pEventData	MQEVENT	Event data
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQQRYM	ByVal HInst	Long	Instance handle
	ByVal MsgNumber	Long	Message number
	pMsgProperties	MQMP	Message properties
	pMsgDataLength	Long	Length (in bytes) of the MsgData area
	MsgData	Any	Buffer to hold the retrieved message or handle
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQQRYS	ByVal HInst	Long	Instance handle
	pElementCount	Long	Number of elements in the set
	pBufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	Buffer to hold the set
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQREG	pClassName	String	Name of the class to associate with the window handle
	ByVal MaxInstances	Long	Maximum instances
	ByVal HWnd	Long	Window handle to associate with the class
	ByVal MsgId	Long	Message identi
	ByVal Options	Long	Options
	pCompCode	Long	Completion code
	pReason	Long	Reason code

<i>Table 26 (Page 5 of 6). MQSeries 3T APIs for Visual Basic</i>			
API	Parameter	Declared	Description
MQRPLY	ByVal HInst	Long	Instance handle
	pMsgName	String	Name of the reply message
	ByVal MsgAttrs	Long	Message attributes
	MsgData	Any	Buffer that holds the reply message
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQSEND	ByVal HInst	Long	Handle of the sending instance
	pClassName	String	Name of the destination class
	pInstanceName	String	Name of the destination instance
	pMsgName	String	The name of the message to be sent
	ByVal MsgAttrs	Long	Message attributes
	MsgData	Any	Buffer that holds the message data to be sent
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQSETS	ByVal HInst	Long	Instance handle
	ByVal InstanceState	Long	Instance state
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQTIME	ByVal HInst	Long	Instance handle
	pRuleName	String	Name of the timed rule
	ByVal Timeout	Long	Timeout period in seconds
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQUREG	pClassName	String	Name of the class to unregister
	ByVal HWnd	Long	Handle of the window to unregister
	ByVal Options	Long	Options
	pCompCode	Long	Completion code
	pReason	Long	Reason code

<i>Table 26 (Page 6 of 6). MQSeries 3T APIs for Visual Basic</i>			
API	Parameter	Declared	Description
MQVALS	pHSet	Long	Set handle
	ByVal BufferLength	Long	Length (in bytes) of the buffer area
	Buffer	Any	Buffer containing the set data
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQXRPLY	ByVal HInst	Long	Instance handle
	ByVal OperationCode	Long	Type of reply
	ByVal OperationVersion	Long	User-defined value
	ByVal MsgAttrs	Long	Message attributes
	ByVal MsgDataLength	Long	Length (in bytes) of the MsgData area
	MsgData	Any	Buffer that holds the reply message
	pConversionDLL	String	Name of the data-conversion DLL
	pCompCode	Long	Completion code
	pReason	Long	Reason code
MQXSEND	ByVal HInst	Long	Instance handle
	pClassName	String	Name of the destination class
	pInstanceName	String	Name of the destination instance
	ByVal MsgType	Long	Type of message
	ByVal OperationCode	Long	Purpose of the message
	ByVal OperationVersion	Long	User-defined value
	ByVal MsgAttrs	Long	Message attributes
	ByVal Role	Long	Role
	ByVal MsgDataLength	Long	Length (in bytes) of the MsgData area
	MsgData	Any	Buffer that holds the message to be sent
	pConversionDLL	String	Name of the data-conversion DLL
	pCompCode	Long	Completion code
	pReason	Long	Reason code

Appendix C. Diskette Contents

SG24-4664
IBM MQSeries Three Tier
Examples for Windows and AIX

Diskette 1

1. *README* contains the same information as this appendix.
2. *CL1T.EXE* can be used to verify a client/server connection. Its use is demonstrated in Chapter 2, "Installation" on page 15.
3. *cmqaix* is a directory that contains a compressed tar format file with the programs and definition files to run the MQI file transfer example described in Chapter 4, "File Transfer Example" on page 77. The chapter explains how to install and run the sample.

The commands to uncompress the file are:

- `dosread cmqaix.tz cmqaix.tar.Z` (copy the file on a AIX machine)
 - `uncompress cmqaix.tar.Z`
 - `tar -xvf cmqaix.tar`
4. *c3taix* is a directory that contains a compressed tar format file with the programs and definition files to run the MQ3T file transfer example described in Chapter 4, "File Transfer Example" on page 77. The chapter explains how to install and run the sample.

The commands to uncompress the file are:

- `dosread c3taix.tz c3taix.tar.Z` (copy the file on a AIX machine)
 - `uncompress c3taix.tar.Z`
 - `tar -xvf c3taix.tar`
5. *vb3twin* is a directory that contains all files required in the Windows client workstation, the class definitions and the Visual Basic files.
 6. *classes* is a directory that contains the class definition for MQ3T in the AIX server.

Diskette 2

1. *bltaix* is a directory that contains all files for the BLT example in Chapter 5, "The Bacon Lettuce and Tomato Sandwich" on page 111 that are required in the server.
2. *bltwin* is a directory that contains all files for the BLT example in Chapter 5, "The Bacon Lettuce and Tomato Sandwich" on page 111 that were created in the Windows workstation.

Diskette

List of Abbreviations

APA	all points addressable	MQ	message queuing
API	application program interface	MQ3T	MQSeries Three Tier
ASCII	American National Standard Code for Information Exchange	MQI	message queuing interface
BL	Business Logic	NFS	network file server
BLM	Business Logic Manager	OOA	Object-Oriented Analysis
CDL	Class Definition Language	OOD	Object-Oriented Design
CM/2	IBM Communications Manager/2	OS/2	IBM Operating System/2
CRLF	carriage return / line feed	PL	Presentation Logic
CUA	Common User Access	PLM	Presentation Logic Manager
CSD	Corrective Service Diskette	PM	Presentation Manager
DDL	dynamic link library	SAA	System Application Architecture
EBCDIC	extended binary coded decimal interchange code	SDDM	Self-defining Data Manager
GUI	graphical user interface	SMIT	System Management Interface Tool
IBM	International Business Machines Corporation	SNA	System Network Architecture
ITSO	International Technical Support Organization	SVGA	Super Video Graphics Adapter
LPP	licensed program product	TCP/IP	Transmission Control Protocol/Internet Protocol
MLE	multiline entry field	VB	Visual Basic
		VGA	Video Graphics Adapter
		WYSIWYG	what you see is what you get

Index

Special Characters

#INCLUDE 100

Numerics

3T 5

- API calls for Visual Basic 251
- application model 11
- icons 50
- message flow 5
- run-time components 3
- Visual Basic interface 44
- what tiers are 2
- Windows directories 71

3T design 14

- application model 7
- BLT example 121
- file transfer options 77
- software distribution 107

3T for Windows 39

3T samples

- BLT application 111
- file transfer 77
 - develop MQ3T version 94
 - using MQ3T 86
 - using MQI 80
- for Visual Basic 50
 - for quick start of PLs 64
 - HELLO1 51
 - HELLO1 set up and run 67
 - HELLO2 59
 - read-only fragments 61
 - template 63
 - software distribution 107

A

abbreviations 259

acronyms 259

AIX 1

- redo BLT 198
- redo file transfer 98

AIX server 16

AIX workstation 10, 15

alter queue 84

amqscoma.tst 68

AntiVirus/DOS 28

API 7

calls

See MQ...

list of MQ3T calls 64

MQ3T calls for Visual Basic 251

MQTIME 11, 136

notes to Visual Basic 66

API types 65

base calls 65

PLM calls 65

SDDM calls 65

application

design 9, 14

development process 12

development requirements 12

execution requirements 12

model 11

portable 204

simulator 8

applcid 83

auto-start 32

AUTOEXEC.BAT

3T Windows client 39

BMQCCSID 39

MQSERVER 37

TCP/IP 33

Windows 3.1 29

B

basket

business logic 183

procedure "delivery.c" 189

procedure "foodinq.c" 188

binary class file 11

bit map viewer 79

BLM

in background and foreground 91

BLT

3T design 121

analysis 113

building GUIs 141

business logic 173

class descriptions 127

classes 121

description 111

design crosscheck 139

GUI prototypes 118

make file 178

message flow 116

messages 122

exceptions 124

inventory control process 123

maintenance 124

operation codes 126

order process 124

production process 122

structure 127

summary 125

objects and functions 114

profile for BLM 200

profile for PLMs 141, 200

- BLT (*continued*)
 - queue definitions 199
 - requirements 112
 - rules and methods 128
 - server connection 198
 - skeleton files 173
 - start BLMs 200
 - test 198
- BLT sample program 111
- bltcoma.tst 199
- bltdef.bas 249
- bltdef.h 247
- bltmake.c 181
- bltnone.c 183
- bltorder.c 180
- bltserve.c 183
- BMQ_NOTIFY 44, 48, 53, 105, 146
 - declared 51
 - explained 45
- BMQB.BAS 51
 - installed 41
- BMQC.H 51, 68
- bmqcc example (AIX) 70
- BMQERROR.LOG 75, 98, 200
- bmqh link 68
- BMQLOCPATH 39
- BMQNTFY.VBX 47, 48, 50
 - between PL and PLM 47
 - install 42
- bmqsysms.ch 125
- BMQVBX.BAS 51
- breadbox
 - business logic 183
 - procedure "delivery.c" 189
 - procedure "foodinq.c" 188
- Business Logic 2, 14
- Business Logic Manager 3
- business requirements 12
- ByVal 66

C

- CBbl 170
- CDL 4, 6
- challenger.gif 79
- channel 36, 82, 84
 - trigger monitor 84
- channel definition 37
- chgrp command 67, 82, 87
- chmod command 67, 82, 87
- chown command 67, 82, 87
- circular message flow 118
- cl1t 37, 38
- class 2, 113
 - application model 11
 - description 6
 - section 6
 - types 11

- class compiler 6
 - functions 7
- class file contents 6
- class name 46
- class source files
 - BLT: basket 220
 - BLT: BL rules and methods
 - explained 131
 - summary 133
 - BLT: breadbox 223
 - BLT: class descriptions 218
 - BLT: definitions 247
 - BLT: fridge 226
 - BLT: gremlin 229
 - BLT: grocer 230
 - BLT: Karen 232
 - BLT: Konrad 236
 - BLT: Luigi 238
 - BLT: messages 213
 - correlation 138
 - operation codes 126
 - role 138
 - structure 127
 - BLT: microwave 240
 - BLT: PL rules and methods
 - explained 128
 - summary 130
 - BLT: repair list 242
 - BLT: shopping list 243
 - BLT: toaster 245
 - crosscheck 139
 - file transfer example 96
 - late messages 135
 - messages
 - explained 126
 - state depended rules 136
 - system messages 125
 - timed rules 135
- client 1
 - 3T functions 3
 - channel in server 37
 - check MQ connection to server 37
 - compile HELLO1 72
 - installation 28
 - local queue in server 36
 - MQSERVER environment variable 37
 - ping the server 33
 - run HELLO1 74
 - setup server for BLT 198
 - Visual Basic support 43
- CMQB.BAS 51
 - installed 41
- CMQC.H 51, 68
- cmqh link 68
- code fragments 61
- code page 39, 203
- command button 145

- communication
 - MQ server connection 36
 - test TCP/IP connection 33
 - verify client/server connection 37
- CompCode (defined) 54
- CompCode (displayed) 54
- compile
 - class source files 97
 - HELLO1 sample 67
- complex data types 66
- CONFIG.SYS
 - environment variable space 29
 - TCP/IP 33
 - Windows 3.1 29
- configure TCP/IP 31
- ConversionDLL 203
- convert
 - character to long 170
- cook.c 190
- correlate messages 138
- create (Visual Basic)
 - command button 145
 - label 144
 - option button 157
 - radio button 157
 - text box 144, 160
- create remote queue 36
- crosscheck 7
- crtmqm 23
- CSINCLUDE 125
- custom command 31
- custom control 47, 50
- customization
 - TCP/IP 31

D

- data conversion 203
 - make file 209
- data conversion program 205
- Data Logic 2, 7, 14, 78, 103
- DEF file 7
- default queue manager 23
- define channel 37, 82, 83
- define process 83
- define queue alias 87
- define remote queue 87
- define transmission queue 82
- define trigger queue 83
- delivery.c 189
- DEPTH 83
- design
 - application 9, 14
- design crosscheck 139
- developer's workstation 9
- development process 12
- device selection 17
- display
 - CompCode, Reason 54

- display (*continued*)
 - return codes 54
- display queue 36
- documentation
 - TCP/IP 30
- dosdir command 80
- dosread command 81, 87
- doswrite command 35

E

- element list 192
- elements (set) 169
- endplm utility 75
- enhancements 1
- entry point 100, 174
 - entry point
 - See MQENTRY
- environment variable
 - BMQCCSID 39
 - INCLUDE 68, 98
 - LANG 97
 - language 67
 - MQSERVER 37
 - space (CONFIG.SYS) 29
 - TCPBASE 30
- error log 75, 200
- event 11
 - and rules 7
 - custom control 47
 - direct to PL 44
 - end 56
 - in MQREG 45
 - instance deleted 48
 - message ID 48
 - OAK1_NewEvent 48, 149
 - procedure 48, 54
 - procedure to process 55
 - query 56
 - rule satisfied 48
 - structure MQEVENT 56
 - what it is 11
 - window unregistered 48
- export file 173

F

- facilities 6
- file transfer 77
- file transfer sample
 - 3T version 80
 - business logic 100
 - C skeleton 100
 - class source files 94
 - compile class source files 97
 - file structure 100
 - files 88, 90
 - possible extensions 107
 - presentation logic 104
 - run it 90

- file transfer sample (*continued*)
 - 3T version (*continued*)
 - sender program (BL) 101
 - set-up receiver station 89
 - set-up sender station 86
 - set-up Windows station 89
 - skeleton files 98
 - description 78
 - MQI version 79
 - files 81
 - run it 84
 - set-up receiver station 83
 - set-up sender station 80
- first.cmd 85
- font size 144
- foo1.cmd 80
- foodinq.c 188
- fridge1.c 184
- ftp command 36

G

- Gremlin
 - Visual Basic project 155
- grocer
 - business logic 191
 - procedure "grocer1.c" 191
- grocer1.c 191

H

- HELLO1
 - compile (Windows) 72
 - compile on AIX 68
 - files 69
 - profile 73, 74
 - run it 73
 - set up on AIX 67
 - view GUI (Windows) 71
- HELLO1.FRM 52
- HELLO1H.BAS 58
- HELLO1x.BAS 58
- HELLO2.FRM 60
- HELOGU1W.MAK 51
- HELOGU2W.MAK 59
- home directory 67

I

- icons used by 3T 50
- include statement 100
- inform message 4
- installation 15
 - AIX server 16
 - 3T for AIX 25
 - MQSeries base 16
 - MQSeries CSD 21
 - MQSeries objects for AIX
 - windows client (development) 28
 - 3T 39

- installation (*continued*)
 - windows client (development) (*continued*)
 - 3T Visual Basic Support 41
 - DOS 7.0 28
 - MQSeries client 34
 - TCP/IP for Windows 30
 - Visual Basic 40
 - Windows 3.1 29
 - windows client (production) 42
- instance 113
- instance handle 48
- instance state 46, 128
- IP address 31

J

- job 113
- job viewer 8

K

- Karen
 - business logic 179
 - C skeleton file 175
 - export file 173
 - make file 178
 - procedure "bltmake.c" 181
 - procedure "bltnone.c" 183
 - procedure "bltorder.c" 180
 - procedure "bltserve.c" 183
- killmqm command file 82
- Konrad
 - class description 127
 - explained 127
 - Visual Basic project 144

L

- label 144
- language 67
- late message 128, 136
- license agreement 41
- In command 68
- log level 200
- LPARAM 47, 48, 50
- ls command 68
- Luigi
 - Visual Basic project 151

M

- MA3B SupportPac 41
- make command 68
- make example (AIX) 70
- make EXE (Visual Basic) 72
- make file 7
- make file (data conversion) 209
- make file example (AIX) 178

- malloc 192
- MATCHSTATE 136
- message 4, 6
 - data flag 174, 182
 - descriptor 203
 - ID 48
 - routing 99
 - set 59
 - types 4
- message flow 5
- method 7, 11
 - for PL 128
- microwave
 - business logic 190
 - procedure "cook.c" 190
- mkdir command 81, 87
- mkgroup command 16
- mkuser command 16
- mount command 16
- MQ_SYSTEM_OWNER_ENDED 125
- MQ3T
 - directories 71
 - icons 74
 - program manager icons 40
 - Visual Basic API calls 251
- MQADDC 59
 - explained 60
- MQADDI 170
 - explained 171
- MQClose 38
- MQConnect 38
- MQCPYB 192
 - explained 192
- MQCPYC 60
 - explained 60
- MQCRTS 59, 170
 - explained 60, 170
- MQDELS 170
- MQDisconnect 38
- MQENDE 55, 60, 105, 150
 - explained 56
 - parameters 57
- MQENTRY 100
 - file transfer skeleton 100
 - for any message 176
 - for one specific message 174
 - for three messages 177
 - when timer expires 177
- MQFMT_NONE 203
- MQFMT_STRING 203
- mqftp command format 79
- mqftp 78
- mqftpctx 78
- MQGet 38, 203
- MQM objects 22
- MQOpen 38
- MQPLM_HWND_UNREGISTERED 48

- MQPLM_INSTANCE_DELETED 48
- MQPLM_RULE_SATISFIED 48
- MQPut 38
- MQQRYE 55, 60, 105, 150
 - explained 56
 - parameters 57
- MQQRYM 55, 60, 105
 - explained 56
 - parameters 57
- MQQRYS 170, 192
 - explained 171, 192
- MQREG 44, 53, 105, 146
 - explained 44
 - parameters 45
- MQRPLY 181, 183
- MQSEND 49, 56, 106, 150
 - explained 48, 148
 - parameters 49
- MQSEND (set) 59, 170
 - explained 60
- MQSeries
 - base product 16
 - for AIX 16
 - updates for MQ3T 21
- MQSeries 3T for AIX
- MQSERVER 39
- MQSETS 45, 55
 - explained 46
 - parameters 46
- MQTIME 11, 136, 148, 163, 169, 182
 - explained 136, 163, 168, 169
 - parameters 136
- MQUREG 45, 55
 - explained 45
 - parameters 46
- MQXSEND 102, 103
- MSG100.def (OS/2) 211
- MSG100.mak (AIX) 209
- MSG100.mak (OS/2) 211
- multi line 144

N

- name resolution 32
- NDIS interface 31
- new line characters 149
- NOTMATCHSTATE 137
- number format 203

O

- OAK1_NewEvent 48, 149
- objects for MQM 22
- OO 14
- option button 157
- OS/2 1
 - data conversion 210
- OS/2 workstation 9

overview 1

P

- parameter passing 47
- parameters of API calls 251
- PCUST.FRM 63
- PCUST.MAK 62
- permission 67
- PFCUST.FRM 61
- PFCUST.MAK 61
- ping
 - DOS 33
 - Windows 34
- PL and BMQNTFY 47
- PLACEHOLDER 135, 136
- PLM and BMQNTFY 47
- PLM and Visual Basic 44
- PLM API calls 7
- PMQELIL 192
- portable application 204
- pre-requisites 6
- Presentation Logic 2, 14
- Presentation Logic Manager 3
- procedures
 - business logic (C)
 - bltmake (Karen) 181
 - bltnone (Karen) 183
 - bltorder (Karen) 180
 - bltserve (Karen) 183
 - cook 190
 - delivery 189
 - file transfer program 101
 - fooding 188
 - fridge1 184
 - grocer1 191
 - xclear 194
 - xgremlin 197
 - xignore 195
 - xiquiry 196
 - xrepair 194
 - data conversion 205
 - Visual Basic
 - BLT_Click (Konrad) 148
 - Close_Click 45, 55
 - Deliver_Click (Luigi) 153
 - DS_MLE (display) 149
 - DS_MSG (display) 162
 - Enter_Click (Gremlin) 158
 - Exit_cmd_Click 104
 - Form_Load 44, 53, 105, 146
 - Form_Unload 56, 146
 - input (shopping list) 168
 - Inquire_Click (repair list) 163
 - Inquire_Click (shopping list) 169
 - MenuExit_Click 56
 - MenuHelloBL_Click 49, 56
 - OAK1_NewEvent 48, 54, 149
 - Option1_Click (Gremlin) 157
 - Option1_Click (repair list) 162

procedures (*continued*)

Visual Basic (*continued*)

- Option1_Click (shopping list) 168
- Pizza_Click (Konrad) 148
- ProcessPLEvent 55
- ProcessPIEvent (file transfer) 105
- ProcessPLEvent (Gremlin) 159
- ProcessPLEvent (Konrad) 150
- ProcessPLEvent (Luigi) 154
- ProcessPIEvent (repair list) 164
- ProcessPLEvent (shopping list) 171
- Quit_Click 146
- Repair_Click (repair list) 163
- SendMe_cmd_Click (file transfer) 106
- Shop_Click (shopping list) 170
- user input (file transfer) 106

process object 83

profile

- BLM in BLT example 200
- for file transfer 91, 92
- for HELLO1 73, 74
- for messages routing 99, 107
- PLMs in BLT example 141, 200

Q

- QLOCAL 36, 82, 83
- queue (alter) 84
- queue definitions for BLT 199
- queue manager
 - create 23
 - directories 22
 - naming convention 23
 - objects 24

R

- radio button 157
- Reason (defined) 54
- Reason (displayed) 54
- redo command file (AIX)
 - for file BLT example 198
 - for file transfer example 98
- refrigerator
 - business logic 183
 - procedure "delivery.c" 189
 - procedure "fooding.c" 188
 - procedure "fridge1.c" 184
- remote queue 36
- repair list
 - Visual Basic project 159
- request message 4
- response message 4
- return address 48
- return codes 54
- role 138
- rootvg 22
- routing information 32

- routing messages 99
- rule 6
 - correlate messages 138
 - define for ... 128
 - dependency 128
 - state dependent 136
 - timed 135
- run-time components 3
- run-time utility SPEEDUP 64
- runmqchl 84, 90
- runmqsc 24, 36, 68, 82
- runmqtrm 83, 84, 90
- running HELLO1 73

S

- sampcoma.tst 68
- scroll bar 145
- SDDM 4, 8, 169
 - data conversion 204
- SDDM API calls 8, 65
- SDDM error handling 66
- SDDM validity checking 66
- sdiff command 69
- Self-Defining Data Manager
 - See SDDM
- sending class, instance 48
- server 1
 - 3T functions 3
 - channel for client 37
 - check MQ connection from client 37
 - create MQM objects 22
 - installation (AIX) 16
 - local queue for client 36
 - run HELLO1 73
 - setup and compile HELLO1 67
 - setup for BLT 198
- set 59, 169
- set-up
 - MQ3T file transfer 86
 - MQI file transfer 80
- setup
 - DOS 7.0 28
 - Windows 3.1 29
- shell script first.cmd 85
- shell script foo1.cmd 80
- shopping list
 - Visual Basic project 166
- simulator 8
- skeleton files 7, 98, 173
- SMIT 17
- software distribution application 107
- SPEEDUP 64
- start
 - channel 84
 - client for MQ3T file transfer 91
 - receiver for MQ3T file transfer 90
 - receiver for MQI file transfer 84
 - sender for MQ3T file transfer 90

- start (*continued*)
 - sender for MQI file transfer 84
 - trigger monitor 84
- startjob utility 75, 92
- state 46, 128
- state and rule 136
- strblm 73, 91
- strmqm 68, 73, 90
- strplm utility 74, 92
- structured design 14
- Sub Close_Click 45, 55
- Sub DisplayCompCode 54
- Sub Form_Load 44, 53
- Sub Form_Unload 56
- Sub MenuExit_Click 56
- Sub MenuHelloBL_Click 49, 56, 59
- Sub OAK1_NewEvent 48, 54
- Sub ProcessPLEvent 55, 60
- subnet mask 31
- SupportPac 41
 - content 50
 - MA3B for Visual Basic 43
- SVRCONN 37
- system messages (3T) 125
- system test 198
- systems management requirements 13

T

- tabstop 145
- tar command 81, 87
- TCP/IP 30
 - customization 31
 - installation (DOS) 30
 - ping 33
 - test connection 33
- TCPBASE 30
- TCPCHECK 33
- TCPREAD 30
- template 63
- test 198
- test configuration 33
- test harness 8
- text box 144
- text button 160
- three tiers 2
- tier 78
 - purpose 14
 - what it is 2
- timed rules 135
- timeout value 148
- toaster
 - business logic 190
 - procedure "cook.c" 190
- toucan.gif 79
- transmission queue 82
- trigger 6, 57, 79, 83, 91
- trigger monitor 83, 84, 90

trigger monitor error 85
triggered target queue 83
triggering rules 13, 57
TRIGTYPE 83
Trim 106
types of API calls 65

U

uncompress command 81, 87
using VB 3T samples 67
using Visual Basic 77

V

validate design 7
variable length message 59
VBRUN300.DLL 43
 install 42
VBX file 47, 50
verify client/server connection 37
Visual Basic
 3T interface 44
 3T samples 43, 51
 using the samples 67
 3T support 41, 43
 add file 143
 and PLM 44
 CBbl 170
 endplm 75
 generic frame and project 144
 HELLO1 sample 51
 HELLO2 sample 59
 installation 40
 make EXE 72
 MQ3T API calls 251
 new line 149
 new project 142
 notes to API calls 66
 open project 71
 program manager icons 41
 startjob 75
 strplm 74
 SupportPac 43
Visual Basic calls by reference/value 66

W

wave 123
window message ID 51
Windows 3.1 29
Windows client 34
Windows workstation 10, 15, 28
WM_USER 51, 96
 Visual Basic 51
workstation 9, 10
WPARAM 47, 48, 50

X

xclear.c 194
xgremlin.c 197
xignore.c 195
xinquiry.c 196
xrepair.c 194

**International Technical Support Organization
MQSeries Three Tier
Examples for Windows Clients and AIX Servers
March 1996**

Publication No. SG24-4664-00

Your feedback is very important to help us maintain the quality of ITSO Bulletins. **Please fill out this questionnaire and return it using one of the following methods:**

- Mail it to the address on the back (postage paid in U.S. only)
- Give it to an IBM marketing representative for mailing
- Fax it to: Your International Access Code + 1 914 432 8246
- Send a note to REDBOOK@VNET.IBM.COM

**Please rate on a scale of 1 to 5 the subjects below.
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)**

Overall Satisfaction	_____		
Organization of the book	_____	Grammar/punctuation/spelling	_____
Accuracy of the information	_____	Ease of reading and understanding	_____
Relevance of the information	_____	Ease of finding information	_____
Completeness of the information	_____	Level of technical detail	_____
Value of illustrations	_____	Print quality	_____

Please answer the following questions:

- a) If you are an employee of IBM or its subsidiaries:
- | | | |
|--|----------|---------|
| Do you provide billable services for 20% or more of your time? | Yes_____ | No_____ |
| Are you in a Services Organization? | Yes_____ | No_____ |
- b) Are you working in the USA? Yes_____ No_____
- c) Was the Bulletin published in time for your needs? Yes_____ No_____
- d) Did this Bulletin meet your needs? Yes_____ No_____

If no, please explain:

What other topics would you like to see in this Bulletin?

What other Technical Bulletins would you like to see published?

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

Name

Address

Company or Organization

Phone No.



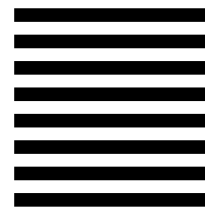
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM International Technical Support Organization
Department HZ8, Building 678
P.O. BOX 12195
RESEARCH TRIANGLE PARK NC
USA 27709-2195



Fold and Tape

Please do not staple

Fold and Tape



Printed in U.S.A.

SG24-4664-00

