FactoryLink ECS

Programmer's Access Kit (PAK)

FactoryLink ECS 6.0 / March / 1996

©Copyright 1984 - 1996 United States Data Corporation. All rights reserved.

- NOTICE -

The information contained herein is confidential information of United States Data Corporation, a Delaware corporation, and is protected by United States copyright and trade secret law and international treaties. This document may refer to United States Data Corporation as “USDATA.”

Information in this document is subject to change without notice and does not represent a commitment on the part of United States Data Corporation (“USDATA”). Although the software programs described in this document (the “Software Programs”) are intended to operate substantially in accordance with the descriptions herein, USDATA does not represent or warrant that (a) the Software Programs will operate in any way other than in accordance with the most current operating instructions available from USDATA, (b) the functions performed by the Software Programs will meet the user's requirements or will operate in the combinations that may be selected for use by the user or any third person, (c) the operation of the Software Programs will be error free in all circumstances, (d) any defect in a Software Program that is not material with respect to the functionality thereof as set forth herein will be corrected, (e) the operation of a Software Program will not be interrupted for short periods of time by reason of a defect therein or by reason of fault on the part of USDATA, or (f) the Software Programs will achieve the results desired by the user or any third person.

U.S. GOVERNMENT RESTRICTED RIGHTS. The Software is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the government of the United States is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or in subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights clause at 48 CFR 52.227-19, as applicable. Contractor/Manufacturer is United States Data Corporation, 2435 North Central Expressway, Suite 100, Richardson, TX 75080-2722. To the extent Customer transfers Software to any federal, state or local government agency, Customer shall take all acts necessary to protect the rights of USDATA in Software, including without limitation all acts described in the regulations referenced above.

The Software Programs are furnished under a software license or other software agreement and may be used or copied only in accordance with the terms of the applicable agreement. It is against the law to copy the software on any medium except as specifically allowed in the applicable agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of USDATA.

Trademarks. USDATA, FactoryLink and FactoryLink ECS are registered trademarks of United States Data Corporation. Open Software Bus is a registered trademark licensed to United States Data Corporation.

All other brand or product names are trademarks or registered trademarks of their respective holders.

Table of Contents . . . .

PAK User Manual

Preface . . . 11

About this Manual . . . 11
How this Manual is Organized . . . 11
How to Use this Manual . . . 11
Organization of this Manual . . . 11
Important Terms . . . 14
Referencing the Operating System Notes . . . 15

1 Introduction to the Programmer's Access Kit (PAK) . . . 17

Requirements . . . 18
Required Hardware . . . 18
Required Software . . . 18
Installing the Programmer's Access Kit . . . 19
Operating System Notes . . . 20
For OS/2 Users . . . 20
For UNIX Users . . . 20
For Windows/NT Users . . . 20

2 FactoryLink Architecture . . . 21

FactoryLink Operation . . . 21
Components of FactoryLink . . . 23
FactoryLink Domain (Multi-operator) Coding Considerations . . . 39
Domains: User and Shared (Per-User Shared Memory Regions) . . . 39
Application Example . . . 43
Configuring FactoryLink . . . 45
How FactoryLink Tasks Transfer Data . . . 46
How FactoryLink Architecture Affects New Task Development . . . 48
FactoryLink Triggers . . . 52
FactoryLink Files and Directories . . . 53
FactoryLink Environment Variables . . . 53
Use of Environment Variables in FactoryLink Path Names . . . 54

Path Name Format 56
FactoryLink Directory Organization 62
Operating System Notes 63
Example 6. Get File Information 72
FactoryLink Directory Organization 73

3 *Constructing a Task* 77

Task Design Guidelines 77
Setting up the Configuration Environment 78
Converting the Database Tables to CTs 80
Writing the Task's Program 81
Overview 82
Operating System Notes 84

4 *Setting up the Configuration Environment* 87

About this Chapter 87
Design the Database Table(s) 89
TYPE 90
OBJECT 90
XREF 90
Task-Specific 90
Create the Attribute Catalog(s) 92
AC File Format 92
Sample AC File 102
Executing an Editor Program from the Configuration Manager 106
Create the KEY Files 108
Construction of a Key File 108
Sample KEY File 108
Test the Configuration Environment 109
Informing FactoryLink about the Task 109
Testing the Configuration Environment 110
Operating System Notes 112

5 *Converting Database Tables to CTs* 115

Creating the CTG Conversion Scripts 116
Conversion Overview 116

Conversion Script Format 117
Sample Conversion Script 124
Creating FactoryLink Configuration Tables (CTs) 124
Adding CT Information to the CM System Table 126
Testing the Conversion Process. 126
Operating System Notes 127

6 *Using the Run-Time Manager* 129

Interaction With Other Tasks 130
Design Conventions 131
Run-Time Requirements 132
Initialization 132
Kernel check (Conditional) 133
Error Handling 133
Termination Notification 133
Orderly Shutdown 134
Domain Selection 134
Sample Task Program Skeleton 135

7 *FactoryLink Kernel and Library* 147

FactoryLink Kernel 148
FactoryLink Library 149
Kernel Multi-User Environment (MUE) Extensions 150
Domains: User and Shared (Per-User Shared Memory Regions) 150
Conventions 152
Return Reference List 153
System Shutdown 155
Kernel and Library Services 156
Process Management 156
Database Access 159
Tag List Registration and Notification 164
Mailbox 165
Memory Management 166
Signals 167
Environment Access 169
CT Access 169

Path Manipulation . . . 171
Format Version Number . . . 171
Message Translation . . . 171
Sleep . . . 175
Miscellaneous . . . 176
Object CT Overview . . . 178
Overview of Object CT Services . . . 178
Overview of the Object CT API . . . 178
Normalized Tag Reference Overview . . . 181

8 FactoryLink API Reference Guide . . . 183

ct_close . . . 184
ct_close_obj . . . 185
ct_create . . . 186
ct_find_index . . . 187
ct_find_obj . . . 188
ct_get_hdrlen . . . 189
ct_get_name . . . 190
ct_get_ncts . . . 191
ct_get_nrecs . . . 192
ct_get_reclen . . . 193
ct_get_type . . . 194
ct_open . . . 195
ct_read_hdr . . . 196
ct_read_index . . . 197
ct_read_rec . . . 198
ct_read_recs . . . 199
ct_update . . . 200
ct_write_hdr . . . 201
ct_write_index . . . 202
ct_write_rec . . . 203
ct_write_recs . . . 204
fl_access_mem . . . 205
fl_alloc_mem . . . 207
fl_change_read . . . 208
fl_change_read_tag_list . . . 210
fl_change_wait . . . 212

fl_change_wait_tag_list 214
fl_clear_chng 216
fl_clear_wait 217
fl_count_mbx 218
fl_create_rtdb 220
fl_dbfmtt 222
fl_delete_rtdb 224
fl_errno 225
fl_exit_app 226
fl_forced_write 227
fl_free_mem 229
fl_get_app_dir 230
fl_get_app_globals 231
fl_get_cmd_line 232
fl_get_copyrt 233
fl_get_ctrl_tag 234
fl_get_env 235
fl_get_msg_tag 236
fl_get_nprocs 237
fl_get_pgm_dir 238
fl_get_stat_tag 239
fl_get_tag_info 241
fl_get_tag_list 243
fl_get_tick 245
fl_get_title 246
fl_get_version 247
fl_getvar 248
fl_global_tag 249
fl_hold_sig 251
fl_id_to_name 252
fl_init 253
fl_init_app 254
fl_lock 256
fl_name_to_id 257
fl_path_access 258
fl_path_add 259
fl_path_add_dir 260

fl_path_alloc 261
fl_path_closedir 263
fl_path_create 264
fl_path_cwd 265
fl_path_date 266
fl_path_get_size 267
fl_path_get_type 268
fl_path_info 269
fl_path_mkdir 270
fl_path_norm 272
fl_path_opendir 273
fl_path_readdir 274
fl_path_remove 276
fl_path_rmdir 277
fl_path_set_dir 278
fl_path_set_device 279
fl_path_set_extension 280
fl_path_set_file 281
fl_path_set_node 282
fl_path_set_pattern 283
fl_path_sys 284
fl_path_time 286
fl_proc_exit 287
fl_proc_init 288
fl_proc_init_app 290
fl_query_mbx 292
fl_read 294
fl_read_mbx 296
fl_read_app_mbx 298
fl_recv_sig 300
fl_reset_app_mem 301
fl_send_sig 302
fl_set_chng 304
fl_set_owner_mbx 305
fl_set_tag_list 306
fl_set_term_flag 307
fl_set_wait 308

fl_sleep . . . 309
fl_test_term_flag . . . 310
fl_unlock . . . 311
fl_wait . . . 312
fl_wakeup . . . 313
fl_wakeup_proc . . . 315
fl_write . . . 316
fl_write_mbx . . . 318
fl_write_app_mbx . . . 320
fl_xlate . . . 322
fl_xlate_init . . . 324
fl_xlate_load . . . 326
fl_xlate_get_tree . . . 329
fl_xlate_set_progpath . . . 331
fl_xlate_set_tree . . . 332
make_full_path . . . 334
spool . . . 336
tsprintf . . . 341
Operating System Notes . . . 342

9 *Normalized Tag References* . . . 347

Normalized Tag Reference Overview . . . 348
Overview of FLNTAG Services . . . 349
Overview of the FLNTAG API . . . 349
Normalized Tag Reference API Guide . . . 352
flntag_calc_base . . . 353
flntag_calc_tag . . . 355
flntag_create . . . 357
flntag_destroy . . . 358
flntag_find_def . . . 359
flntag_find_tag . . . 360
flntag_gen_objname . . . 362
flntag_gen_ref . . . 362
flntag_gen_str . . . 362
flntag_get_dimen . . . 364
flntag_get_member . . . 364
flntag_get_name1 . . . 364

flntag_get_node 364
flntag_parse_brkt4dims 365
flntag_parse_comma4dims 365
flntag_parse_ref 367
flntag_set_dimen 368
flntag_set_member 368
flntag_set_name 368
flntag_set_node 368

10 Object Definitions 371

Object CT Overview 372
Overview of Object CT Services 372
Overview of the Object CT API 372
Object CT API Reference Guide 375
-ct_close_obj 376
ct_find_obj 377
ct_nrecs_obj 378
ct_open_obj 379
ct_read_objs 380
flobjrec_get_chgbits 381
flobjrec_get_descr 381
flobjrec_get_dimen 381
flobjrec_get_domain 381
flobjrec_get_perwhen 381
flobjrec_get_tag 381
flobjrec_get_type 381

Preface

About this Manual

The *Programmer's Access Kit User Manual* covers FactoryLink IV version 4.1.3 and the following operating systems: OS/2, UNIX, VMS and Windows.

Use this manual as a guide when performing the following functions:

- Installing the Programmer's Access Kit
- Creating a FactoryLink-compatible task using the Programmer's Access Kit

How this Manual is Organized

The *Programmer's Access Kit User Manual* contains generic information that applies to the Programmer's Access Kit (PAK), regardless of the operating system. Each chapter in this manual contains an operating system specific section at the end of the chapter. This section contains information about the PAK that is unique to a each operating system.

How to Use this Manual

You are not required to read this manual from cover to cover before attempting to use the PAK. However, Chapters 3-6 contain information relevant to task design that should be covered prior to program design.

Organization of this Manual

This manual presents the following topics:

- Chapter 1, “Introduction to the Programmer's Access Kit (PAK)
  - Required Hardware
  - Required Software
  - Installing the Programmer's Access Kit

- Chapter 2, "FactoryLink Architecture
  - FactoryLink Operation
  - FactoryLink Triggers
  - FactoryLink Multi-User Environment (MUE) Design and Coding Considerations
  - FactoryLink Files and Directories (including Normalized Path Names)
- Chapter 3, "Constructing a Task
  - Guidelines for Task Design
  - Task Construction Procedure
- Chapter 4, "Setting up the Configuration Environment
  Details Phase 1 (Steps 1-4) of the task construction procedure outlined in Chapter 3:
  - Step 1. Design the Database Table(s).
  - Step 2. Create the Attribute Catalog(s).
  - Step 3. Create the KEY Files.
  - Step 4. Test the Configuration Environment.
- Chapter 5, "Converting Database Tables to CTs
  Details Phase 2 (Steps 5-6) of the task construction procedure outlined in Chapter 3:
  - Step 5. Create the CTG Conversion Scripts.
  - Step 6. Test the Conversion Process
- Chapter 6, "Using the Run-Time Manager
  - Interaction With Other Tasks
  - Design Conventions
  - Run-Time Requirements
  - Examples of Correct Coding Techniques
- Chapter 7, "FactoryLink Kernel and Library
  - FactoryLink Kernel
  - FactoryLink Library
  - Calling and Return Conventions
  - System Shutdown
  - Multi-User (Domain) Considerations
  - Kernel and Library Services

- Chapter 8, “FactoryLink API Reference Guide
  Lists and describes each FactoryLink API function, and provides the following information about each function:
  - **Call format:** Syntax to use for this function
  - **Arguments:** List containing the following information about each argument:

    Type

    Name

    Description

    Method used to pass argument (by reference or by value)
  - **Returns:** Possible values returned by the function. Where applicable, the value is listed as the symbolic representation, known as a “keyword,” of the error number returned.
  - **Additional information:** Additional information about the function
- Index
  - Notational Conventions

The following paragraphs describe the text and function representation conventions used in this manual.

Notational Conventions in Text

This manual uses the following conventions to distinguish elements of text:

Convention	Use
bold	Value of an element, valid entries for a field
italic	Name of a manual
`monospace`	Sample command lines and program code and examples
[ ]	(Brackets) Indication of optional command-line or file entry

Representation of Functions

When a FactoryLink function is referred to in the text, it is referred to by its symbolic representation containing all capital letters and no parentheses. For Example, the FactoryLink function that locks the database is referred to as FL_LOCK.

Important Terms

Before you proceed, an important distinction must be made between the following terms used in this manual:

- **Programmer** — refers to the person who is developing the new task, including task design and writing the task's program This manual also refers to the programmer as “you”.
- **Developer** (or Application Developer) — refers to the person who is configuring an application.
- **Operator** — refers to the person who will operate the finished application.
- **User** — has a special meaning in the multi-user environment. See discussion of domains and the concept of the USER/SHARED domains.
- **Database tables** — Configuration database tables that store information about the elements. You must design these tables as part of task development. When the operator selects a task from the Configuration Manager Main Menu and enters data in the task's panels, the database tables store this data.
- **Configuration tables (CT)** — Binary files produced by the CTGEN utility at run time containing data extracted from the database tables.

For term definitions, refer to the Glossary in *FactoryLink Fundamentals*.

Referencing the Operating System Notes

When operating system-specific information is relevent to a topic, small icons representing the specific operating system will appear in the margins next to the relevent text. The following list describes the use of these icons.

W Represents Windows-specific information.

C Represents OS/2-specific information.

U Represents Unix-specific information.

If one or more of these icons appear in the margin of a topic, refer to the end of the chapter under “Operating System Notes” for the appropriate operating system-specific information.

Chapter 1 . . . .

Introduction to the Programmer's Access Kit (PAK)

The FactoryLink Programmer's Access Kit (PAK) is a collection of optional FactoryLink software tools and related documentation for use in the design and construction of FactoryLink-compatible programs.

The PAK allows you to take full advantage of FactoryLink's open architecture. This manual describes the C-language calling conventions. However, you can develop programs using any other language that supports these calling conventions. Customer-written programs have full access to FactoryLink's real-time database and operate in conjunction with other FactoryLink programs.

Use the FactoryLink Programmer's Access Kit to:

- Create new FactoryLink tasks that perform functions not performed by standard FactoryLink tasks
- Develop communications interfaces to computers or devices for which standard interface tasks are not already developed
- Develop a program to replace a standard task to meet current application needs

Unlike any other open architecture system, FactoryLink fully integrates a customer-written task into the FactoryLink environment. The FactoryLink PAK allows customer-written tasks to become an integral part of FactoryLink. FactoryLink's Configuration Manager fully supports the new task with full-screen editing, context-sensitive help, and documentation utilities.

This section contains information about the following topics:

- Required Hardware
- Required Software
- Installing the Programmer's Access Kit

REQUIREMENTS

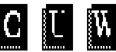
Required Hardware

The PAK is a FactoryLink option used in conjunction with a FactoryLink Development System. The hardware requirements specified in the *FactoryLink Product Matrix* also apply to the Programmer's Access Kit option.

Required Software

The PAK requires specific versions of software tools not included with the PAK.

Refer to the *FactoryLink Product Matrix* for a list of FactoryLink software requirement.



Refer to “Operating System Notes” on page 20 for a list of language-specific compiler requirements.

Installing the Programmer's Access Kit

Perform a selective installation to install the Programmer's Access Kit. Refer to the FactoryLink *Installation Guide* for more information about performing a selective installation. Follow the installation procedure. Install the Programmer's Access Kit software as an option.

Operating System Notes

For OS/2 Users

Required Software (page 18)

For C-language programing, use one of the folowing compilers:

- IBM C/2 Compiler version 1.1
- Microsoft C Compiler version 5.10 or 6.0
- Other versions of C compilers that operate under and generate code for OS/2 and that can link to object libraries and Dynamic Link Libraries (DLLs) created using one of the compilers listed above

For updated compiler requirements, please contact Customer Support.

For UNIX Users

Required Software (page 18)

For C-language programing, use an ANSI C-compliant compiler.

For Windows/NT Users

Required Software (page 18)

For C-language programing, use Borland C for Windows, version 3.0 or 3.1.

For updated compiler requirements, please contact Customer Support.

Chapter 2 . . . .

FactoryLink Architecture

You must know about the FactoryLink architecture to construct a FactoryLink-compatible task.

This chapter contains information about the following topics:

- FactoryLink Operation
- FactoryLink Domain (Multi-operator) Coding Considerations
- Configuring FactoryLink
- FactoryLink Triggers
- FactoryLink Files and Directories
- Operating System Notes

FactoryLink Operation

FactoryLink allows developers to create custom applications by selecting, configuring, and linking different programs so all these programs can freely exchange information in real time. All programs share the FactoryLink global real-time database which uses the "Open Software Bus®" architecture. This real-time database is an array of information composed of individual data items (elements) stored in high-speed memory at run time. Each element contains either a numeric value, a text message, or a "mailbox" entry. The developer specifies the data type of each element and identifies each element by assigning it a unique element name such as **press1**, **temp2**, **valve1**, **time8am**, **action1**.

FactoryLink is a modular system. It is structured as a set of individual programs, or tasks that perform separate functions. For example, one task handles all mathematical computations and logical operations. Other tasks handle timing operations or alarms display. These tasks communicate and share data with one another through the real-time database managed by the FactoryLink kernel and run-time system.

FactoryLink also includes tasks such as

- operator interfaces (implemented as animated color graphics displays)
- historical trending (charting and graphing) functions

- reporting and logging functions
- communications with programmable controllers, and networks

The developer can choose up to 31 different tasks from the FactoryLink library to run concurrently, communicating with each other through the real-time database.

> This limit is only on FactoryLink tasks themselves. Other operating system-specific programs that do not directly access the real-time database can also coexist on the system. The number of non-FactoryLink programs that can operate concurrently with the 31 (maximum) FactoryLink programs is limited only by the amount of available memory and similar operating system constraints. Consult the system manager for information about how to calculate available memory on the system in use.

FactoryLink tasks communicate through a common database rather than directly with each other. This system provides the following advantages over systems relying on real-time inter-process communications (IPC) through passing buffers or sharing files.

Tasks maintain their independence and inherent compatibility with one another.

Data formats for interfaces will not change unpredictably as the maintenance programmer changes.

Tasks can hand off the inter-process communication to the database function which acts as an intermediary, meaning less time is spent waiting for another task to acknowledge error-free receipt of data.

Functions, conditions, or events can be related through the use of a common element since each task has access to the same global elements.

FactoryLink applications can be designed for shared use or private use. This flexibility is accomplished through the use of domains. A domain is a public (shared) or private (user) data area reserved in the real-time database. This database design feature allows the application developer to define which portions of the database are available. The database may be used privately or shared among all user in various parts of the application at run time. During application development, the developer associates specific tasks, tables, and elements with particular shared (public) or user (private) domains.

Components of FactoryLink

FactoryLink consists of the following components:

- FactoryLink kernel
- FactoryLink real-time database
- FactoryLink tasks
- FactoryLink Configuration Manager
- FactoryLink Run-Time Manager
- FactoryLink API
- Recommended optional programs (operating system-specific)

The following sections discuss the relationships among these components and the way the components function in FactoryLink.

FactoryLink Kernel

The FactoryLink kernel is a software module that creates the real-time database when the operator starts the FactoryLink Run-Time system. The kernel creates the database by allocating blocks of memory for elements of each data type. The size of the real-time database is determined by the number of elements used in the application. The kernel also contains a set of callable software services providing FactoryLink tasks with a common utility library. This library allows for uniform and controlled access to the database. These services provided in the utility library include:

- access to and maintenance of the real-time database
- process management
- translation tree management
- directory/path translation and normalization
- miscellaneous utilities

FactoryLink Real-Time Database

The real-time database, that exists only at run time is a memory-resident array of information that acts as an in-memory storage device and interprocess communication mechanism for the tasks. All FactoryLink tasks communicate through the real-time database, sharing the information in the database by reading from or writing to elements.

Elements in the real-time database consist of any data computed or collected by a FactoryLink task as well as operator-entered values. For example, the Math and

Logic task stores the results of mathematical calculations in the real-time database. A communications interface task stores data collected from a PLC. The Event and Interval Timer task stores indications that various timed events have been scheduled or have occurred and records when defined intervals have elapsed.

When planning to create disk files and history data storage, keep in mind the transient nature of real-time database elements. Because the real-time database is memory-resident, it exists only when the Run-Time system is operating. Historical information can be stored only if it is written to disk before the Run-Time system is shut down.

Structure of the Real-Time Database

The real-time database is made up of arrays and pointers. There are six arrays of elements, one array for each supported data type. Whenever the developer defines an element, the system prompts the developer to identify the data type of the element by choosing one of FactoryLink's predefined data types. The following chart shows the storage capacities, ranges, and accuracies of each data type:

Table 2-1

Data Type	Value	Storage in User Area	Storage in Kernel Area	Value Range and Accuracy
Digital (Boolean)	1 bit	* 2 bytes	16 bytes	1 (ON) or 0 (OFF)
Analog (Short integer)	2 bytes	* 2 bytes	18 bytes	-32,768 to 32,767 (signed)
Long Analog (Long integer)	4 bytes	4 bytes	20 bytes	-2^{31} to 2^{31} -1
Floating-point (IEEE standard/double precision)	8 bytes	8 bytes	24 bytes	+/-10^{-308} to +/-10^{308}
Message (String)	0 to 65535 bytes	0 to 65535 bytes	24 bytes per message + storage of data	ASCII or arbitrary binary data

* In HP 9000/800, IBM AIX RS6000, and OSF/1 implementations, digital and analog elements occupy 4 bytes
each of user area storage and 20 bytes of kernel storage.

Table 2-1

Data Type	Value	Storage in User Area	Storage in Kernel Area	Value Range and Accuracy
Mailbox (Variable-length data, organized as a queue)	0 to 65535 bytes	0 to 65535 bytes	24 bytes per message + storage of data	Arbitrary binary data
* In HP 9000/800, IBM AIX RS6000, and OSF/1 implementations, digital and analog elements occupy 4 bytes each of user area storage and 20 bytes of kernel storage.				

Use the following formulas to determine memory requirements:

- For each element in the shared domain:

```
(N-8) + (8 x I)
```

where

N is the Storage in Kernel Area from the table above.

I is the number of instances (shared + user)

Example: A shared domain long analog element in a system with a maximum of two user instances:

```
[20 bytes (from table) - 8] + {8 x [1(shared) + 2 (user)]} = 40
bytes
```

- For each element in the user domain:

```
(N x I)
```

Example: A user domain floating-point element in a system with a maximum of two user instances:

```
24 bytes (from table) x [1 (shared) + 2(user)] = 72 bytes
```

The largest number of elements that can be created per array varies according to operating system:

- In Windows NT and UNIX, an array can have up to 65,535 elements.
- In Windows and OS/2, an array can have as many elements as will fit into a 65,535 bytes memory segment. This number varies by data type. Use the following formula or refer to the following table to determine the number of elements an array can have:

```
max. # of array elements = 65535/N
```

where

N is the Kernel Storage Area from the chart above.

The following table shows the maximum number of elements per array for each data type:

Table 2-2

Data Type of Element	Maximum # of Elements per Array
DIGITAL	4,095
ANALOG	3,640
LONG ANALOG	3,276
FLOATING-POINT	2,730
MESSAGE	2,730 + storage of data
MAILBOX	2,730 + storage of data

The FactoryLink kernel uses data types and element numbers to access (read from or write to) elements in the real-time database.

Structure of an element: An element consists of the following items:

- One or more bits containing the element's value
- Set of change-status bits (change-status word)
- Set of wait bits (wait-status word)

In each of these status words, FactoryLink assigns a single bit to each potential client process. FactoryLink supports up to 31 separate processes per domain instance. A process does not officially become a FactoryLink client process until it registers with the kernel by initializing the FactoryLink calling process, but the bits are still allocated for all 31 possible processes per domain instance.

Change-Status Bits

For each element, one change-status bit exists for each potential client process. The read-call and write-call functions use the change-status bits within the FactoryLink kernel to indicate changes in an element's value. The value of the change-status bit can be either ON (equal to 1) or OFF (equal to 0).

FactoryLink tasks write information to elements either through write calls or forced-write calls. There is an important distinction between a write and a forced write.

If a write call updates a value of an element, the writing task uses the write-call function within the FactoryLink kernel. This function first determines whether the element's value has changed. If the new value is different from the old value, the write-call function sets each of the element's change-status bits to 1 (ON) and stores the new value in the element. However, if the comparison determines that the new value for the element is identical to the old value, nothing is changed. This method can save processing time during repeated updates to the same elements. Use write calls most of the time, except when the forced-write is specifically needed. If a value of an element is updated by a forced-write call, the writing task uses the forced-write call function within the kernel. This function does not compare old and new values. Instead, the forced-write call function assumes the element has changed and sets all of the element's change-status bits to ON as it stores the new value, even if the updated value being assigned to the element is the same as the old value. Forced writes are useful when you need to trigger processes by changing the value of a trigger element but does not wish to change the actual contents of the element, or when an element needs to be processed a second time, even if its value has not changed.

FactoryLink tasks read information from elements through

- read calls
- change-read calls
- and change-wait calls

The read-call function always returns the current value of the element to the calling process regardless of the value of the element's change-status bit assigned to that process.

When a task makes a change-read-call, the reading task requests change-status information about specific elements. If the function finds that the change-status bit of an element has changed since it was last read, the function informs the calling task that it has found a changed element and returns the value of the first changed element found. If the change-status bits have not changed since the last read for any of the specified elements, the change-read-call function returns a code indicating this to the calling task. This method blocks the calling routine's processing for less time than would checking or rereading all the elements.

When a task makes a change-wait call, the reading task uses the change-wait-call function within the kernel to request change-status information about specific elements. Once a task makes its call, the task then hibernates while waiting for an element to change. When a task is asleep, it uses no CPU cycles. The Run-Time

Manager monitors the real-time database and wakes any task whose specified elements have changed and/or have had their change-status bits set to 1 (ON) by a writing task since the last reading by the hibernating task. This call blocks the calling process until at least one of the specified elements' change-status bits are toggled.

Once a task has read an element, the functions for all of the read tasks reset the element change-status bit associated with that task to OFF by writing a 0 to the task change-status bit in the element. As successive tasks read an element, then toggle the change-status bits in the element, one by one, to OFF.

The kernel maintains the change-status bits in a manner transparent to the tasks; however, developers can use these bits in the Math and Logic task. For example, a developer can write a Math and Logic procedure that uses the "?" operator to determine whether the value of an element has changed and then take an action.

Change-status bits optimize system performance in the multi-tasking environment. FactoryLink tasks use change-status bits as exception flags and the kernel acts as an exception processor. Exception processor terminology is uniform across all FactoryLink documentation and should not be confused with the traditional use within the industry of the term exception processing to mean error handling or CPU exception recovery. This allows FactoryLink tasks to perform functions on an exception-only basis (only on those elements whose values have changed) rather than continually reprocessing unchanged information. This method results in a noticeable increase in software performance.

Wait bits: For each element, one wait bit exists for each possible client process. When the client is currently waiting to read or to write the specified element, the value of the bit is 1 (ON); otherwise, the value is 0 (OFF).

Element names

Each element has a unique, developer-defined, symbolic element name. The developer assigns these element names, one for each element used in the application during system configuration and the system stores them in a separate predefined FactoryLink data file. Element names are linked at run time with pointers to their associated elements, allowing their use by any FactoryLink task. Because the FactoryLink real-time database is completely memory-resident and is organized as arrays and pointers when loaded at run time, FactoryLink does not have to keep track of element names stored as text strings. This, in part, accounts for FactoryLink's high processing speed.

This section supplements "Concepts" in the *User Manual.*

Element name assignment: During configuration the developer creates and assigns an element name to an element by entering its name, data type, and a short (optional) description in a configuration table associated with a specific FactoryLink task. If the element referenced by the developer does not exist, the system automatically creates it. Once an element has been defined in this way, other tasks refer to this element using its element name, reading or writing data to or from the element at run time.

You can use a FactoryLink element name to define a single element or it may be used to define an array (or matrix) of elements. If an element name specifies a single element, the element name consists of an alphanumeric string that does not contain brackets. This is known as a one-part element name. One-part element names have the following format.

1-32 characters
First character: A-Z, a-z, @, $, or _
Remaining characters: A-Z, a-z, @, $, _,., or 0-9
No embedded spaces

> **Note:** The (.) character can be used in version 4.1.3; however, if it is used incorrectly, future software updates may produce undesirable results. In future releases of FactoryLink, the (.) character will be used to represent an object that has its own members. Therefore, we recommend that developers not use the (.) in element names in version 4.1.3.

A sample one-part element name is as follows:

```
set_temp
```

An element array is a group of related elements that the developer defines with one two-part element name rather than with individual element names. Element arrays prevent the developer from having to define large numbers of scalar elements separately. Certain FactoryLink tasks, such as Math and Logic and Database Browser, perform operations on an entire element array using only one reference to the array rather than using separate references to each element in the array.

Defining Element Arrays

Define element arrays in a configuration table by using a two-part element name. A two-part element name consists of an alphanumeric string (the array name) followed by one set of square brackets containing an integer (array index) for each dimension of the array. Dimension refers to the capacity or size of the array. If an

array has more than one dimension, it is called multi-dimensional, and each element will have as many array indices as the array has dimensions. Think of the Cartesian coordinate system in which both vertical and horizontal indices are specified to pinpoint a position on the spatial grid. A two-dimensional array is an array in which each of the elements is also an array, and its elements are referenced by [row,column] pairs.

The array dimensions defined within the square brackets are the sizing factors that determine the number of elements in an array. For example, if the developer specifies that the size of a one-dimensional element array is 5, the real-time database creates five separate elements that can be referenced individually or as a group. Their array element names are the same, but they have different array indices. (Refer to “Example 1: Defining a one-dimensional element name” on page 32.)

Two-part element names have the following format:

Part 1: 1-32 characters

First character: A-Z, a-z, @, $, or _
Remaining characters: A-Z, a-z, @, $, _,., or 0-9

Do not embed any spaces between the first and second parts of an element name.

Part 2: 3-16 characters

> **Note:** The (.) character can be used in version 4.1.3; however, if it is used incorrectly, future software updates may produce undesirable results. In future releases of FactoryLink, the (.) character will be used to represent an object that has its own members. Therefore, we recommend that developers not use the (.) in element names in version 4.1.3.

First character: [

Second character: 0-9

Remaining characters: any combination of brackets (**[or]**), and single digits (**0-9**)

Last character: **]**

No embedded spaces

The following is a sample two-part element array name:

```
set_temp[2][1]
```

Each set of square brackets represents a dimension or spatial characteristic of the element array, such as length, width, or height. When a developer defines an element array in a configuration panel, a dialog is displayed, showing default sizes for each dimension that the developer can modify before pressing the Enter key.

The size of a default dimension is always one larger than the integer specified in the element name brackets because C arrays are indexed from zero to the specified dimension. For example, if the element name is tagname[0][0], the default dimension sizes appearing in the dialog box are 1,1. If the element name is tagname[4][6][3], the default dimension sizes are 5,7,4.

This also means that arrays are indexed beginning at zero; for example, tagname[0] is the first element of the element array tagname.

CAUTION

Once the dialog is displayed and you have pressed Enter, you cannot change the number of or the size of the dimensions in an array. If either value is incorrect, delete all elements defined in the array before redefining the element array with different dimensions.

The number of array dimensions and the number of elements allowed in an array are constrained in two ways:

- The element name field limits the size of the Array dimensions. Sixteen characters are allowed in the dimension portion (second field) of an array element name. For example, the array element name tempset[3][1][10][1][1] contains 7 characters in the element name portion (first field) and 16 characters (the maximum) in the dimension portion. The 16-character limit in the dimension field allows the developer to specify up to five dimensions, assuming most of the dimensions are single-digit.
- The largest number of elements that can be created per array is an important limit. This number varies according to data type and operating system.

To determine the number of elements created in an array with a given set of dimensions, use the following formula:

a x b x c

where

a is the size of the first dimension.

b is the size of the second dimension.

c is the size of the third dimension.

If an array contains more than three dimensions, use the same method to multiply by the sizes of the additional dimensions.

Example 1: Defining a one-dimensional element name

If the developer defines a one-dimensional element name

tagarray[0]

then the default size of the dimension that is displayed in the dialog is

1

If the developer modifies the size of the dimension to

5

then five elements are created with the following element names:

tagarray[0]
tagarray[1]
tagarray[2]
tagarray[3]
tagarray[4]

To reference individual elements in the array, enter tagarray[index number] in the panel for each element referenced.

Example 2: Defining a two-dimensional element name

If the developer defines a two-dimensional element name as:

msg_tag[1][1]

the default sizes of the two dimensions that appear in the dialog box are:

2,2

If the developer modifies the sizes of the dimensions to:

3,2

then six elements are created with the following element names:

msg_tag[0][0]

msg_tag[0][1]

msg_tag[1][0]

msg_tag[1][1]

msg_tag[2][0]

msg_tag[2][1]

To reference individual elements in this array, enter msg_tag[index1][index2] in the panel for each element referenced.

Example 3: Using a one-dimensional array

Suppose that a FactoryLink developer needs to define message elements whose values indicate the colors the cars on one conveyor belt are painted. If 300 cars are painted on the conveyor belt and element arrays are not used, the developer must define 300 elements individually. However, with element arrays the developer defines only one two-part element name and specifies that the name contains 300 elements by entering 299 (creating elements 0 to 299) inside the brackets:

color[299]

Set the integer ([299]) sizing the element's single dimension to one less than the desired number, since elements are numbered beginning at index zero (**0**). Here, the desired number (299 + 1, or 300) of elements are created under one element name, as illustrated below:

color[0]
color[1]
color[2]
.
.
.color[299]

Example 4: Using a two-dimensional array

Suppose that the developer wants to define elements whose values indicate the colors that cars on three conveyor belts are painted. The developer can define a two-dimensional element array by entering one element name, as illustrated below:

paint[299][2]

The integer ([299]) for the first dimension indicates one fewer than the number of cars on each conveyor. The integer ([2]) for the second dimension indicates one fewer than the number of conveyors. As a result, the correct number [(299 + 1) X (2 + 1), or 900] of elements are created under one element name, as illustrated below:

paint[0][0]	paint[0][1]	paint[0][2]
paint[1][0]	paint[1][1]	paint[1][2]
paint[2][0]	paint[2][1]	paint[2][2]
paint[3][0]	paint[3][1]	paint[3][2]
.	.	.
.	.	.
.	.	.
paint[299][0]	paint[299][1]	paint[299][2]

Example 5: Using a three-dimensional array

Suppose that the developer must define elements whose values indicate the colors that cars on three conveyor belts and trucks on three conveyor belts are painted. The developer can define a three-dimensional element array using one element name, as illustrated:

car_truck[299][2][1]

The third dimension ([1]) indicates the type of automobile. In this example, 1,800 elements are created under one element name:

(299 + 1) X (2 + 1) X (1 + 1) = 1800

The three-dimensional array may be thought of as a cube of indexed elements, each element having an (x,y,z) coordinate reference (such as depth, vertical position, and horizontal position).

Element Descriptions

A description includes optional information about the element defined or referenced with an element name in a data entry panel. When the developer assigns an element name to an element and choose Enter, the system prompts for a description of that element as well as for a data type. Enter a description of no more than 40 alphanumeric characters.

Once an element has been defined, we recommend developers not change the original description. However, two methods for changing descriptions are available:

- Open the Element Tag List (refer to Chapter 5, “Converting Database Tables to CTs) and modify the description field.
- On the panel where the element has been defined, place the cursor in the element name field and press <Ctrl-T>. The Element Tag Definition box is displayed, allowing the developer to change information about this element, including its description.

Element name storage

Element names are stored in the /{**FLAPP**} directory in the OBJECT database table. Refer to “Design the Database Table(s)” on page 89 in Chapter 4, “Setting up the Configuration Environment” for details about the OBJECT database table.

The information stored in these tables enables the Configuration Manager to translate an element name into its associated element number and data type which are stored in a binary configuration table file (CT file). At run time, the typical FactoryLink application program reads its configuration table files (CT files) and makes calls to read and/or write elements identified by the element numbers and data types listed in the CT files.

Predefined elements

The FactoryLink starter application FLNEW comes with a set of predefined elements for internal system use. These also serve as a convenience to application developers who can choose from these predefined elements when configuring tasks. Predefined elements in the USER domain contain a unique identifier of _U. Predefined elements are listed alphabetically by element name in the current FactoryLink *Release Notes*.

FactoryLink Tasks

FactoryLink tasks are programs that read from and write to the real-time database, allowing for an exchange of information among the tasks. The developer

uses the configuration tables to specify the elements read or written by each task. A task is only aware of the elements that it is reading and writing, not of other tasks and who owns the element. Therefore, the task does not know which task wrote the data it is reading from the database nor can it know which task is reading the data it is writing to the database.

Although some tasks may only read or write to elements, most tasks have both read and write access to the elements. Remember that all elements are global so any task can use any element as long as that element's data type matches the type required for the indicated operation. It is possible for two tasks to use the same element, sharing responsibility for updating the information stored in the element.

It is important to plan and configure the system so read and write operations to the real-time database are predictable. For instance, if two tasks are configured to write simultaneously to the same element in the real-time database, it is impossible to know which task is responsible for the data in the element. No inherent mechanism for preventing this situation exists. Even though it is sometimes desirable to have such a capability, most applications should be designed so there is only one writer task and one or more reader tasks for any particular element. Otherwise, the LOCK and UNLOCK functions may provide some protection so tasks do not undo a desired update.

A FactoryLink application task can call functions to perform the following operations involving the real-time database:

- Lock the database
- Unlock the database
- Read one or more database elements
- Write one or more database elements
- Determine whether a database element has been modified by another task since it last read the element
- Sleep until any one member of a list of specific database elements is modified
- Access miscellaneous utilities

A variety of pre-written tasks are available for use. Refer to Chapter 1, “Introduction to the Programmer's Access Kit (PAK)” for information about available FactoryLink tasks.

Configuration Manager

The Configuration Manager is a development system program used to set up or configure a FactoryLink application. It is used only for system configuration and application development.

Developers create an application by specifying the elements created for each task and the actions each task performs on these elements. In the Configuration Manager, the developer enters data in one or more database tables for each task. The Configuration Manager normally invokes a default Panel Editor that provides a data entry panel for each of the database tables. However, the Configuration Manager may invoke specialized editors, such as the Application Editor, to handle non-textual configuration.

Although the Configuration Manager is not necessary to enter and maintain task configuration information, it reduces the development time for an application. More importantly, it means that developer-written tasks have the same developmental look and feel as standard FactoryLink tasks.

Run-Time Manager

The Run-Time Manager is a program that monitors and controls other FactoryLink tasks using the services of the real-time database. The Run-Time Manager performs the following functions:

- Task startup
- Task shutdown
- Task status display

The Run-Time Manager has a Configuration Manager-maintained table known as the System Configuration table. This table specifies startup criteria for each task started by the Run-Time Manager. The Run-Time Manager communicates with each task by sending commands to and reading status information from the real-time database. Each task must monitor the command objects so the task shuts down when instructed by the Run-Time Manager. Refer to “Design the Database Table(s)” on page 89 in Chapter 4, “Setting up the Configuration Environment” for additional information about the System Configuration table.

It is not essential that a FactoryLink task be started by the Run-Time Manager. A FactoryLink custom-written application can be started automatically by the Run-Time Manager maintaining a consistent operator interface with the built-in FactoryLink tasks, or it can be designed to start manually. Should you choose to start a task manually, it will be up to the operator to set the appropriate environment variables. The following environment variables should always be set before starting the task manually:

- {FLINK} base directory of FactoryLink installation
- {**FLAPP**} base directory of configuration files
- {FLLANG} language environment variable
- {FLNAME} invocation name
- {FLDOMAIN} domain name
- {FLUSER} user instance name

FactoryLink API (FLIB)

FactoryLink tasks use the FactoryLink API, a library of C functions, to read and write data contained in the real-time database. The library also includes additional support functions. Where possible, the API is consistent across operating systems and platforms. This permits programs written in ANSI C to be compiled without source modification on any of the FactoryLink platforms. The developer can upgrade the hardware platform by physically porting the application over to a different platform and recompiling it.

Recommended Optional Programs

Refer to Chapter 3, “Constructing a Task” for a list of recommended options in a FactoryLink system.

FactoryLink Domain (Multi-operator) Coding Considerations

This section discusses coding considerations required to handle multi-operator FactoryLink operation.

Domains: User and Shared (Per-User Shared Memory Regions)

Data areas or groups of elements may be grouped and marked with protection to be opened in a particular way. Some elements may need to be the same value for all users and change whenever any operator modifies them. Because of the nature of the multi-operator environment (MUE), other elements may need to be different dynamically for each operator. One operator or a set of users may need to have private copies of particular data items in the same Real Time Database. If data items are separated into public (shared simultaneously) and private (duplicated at startup of the task, but dynamically modified as the application progresses), the MUE can function optimally. The group of shared data items is the same across instances of the application while each user has copies made of the private data items. These general concepts in the FactoryLink system are termed shared and user domains.

User domain

FactoryLink tasks use elements as a means of task control. These elements must be duplicated for each operator. The subset of the real-time database duplicated on a per-user basis is known as the User domain. In other words, each operator can have data in the Real Time Database that only he can open. The instance of the application will behave uniquely as he modifies these values during each user's instance.

The multi-user extensions allow duplicate-named element areas (tag areas) by allocating an array of pointers to database segments for each user. This allows the tag numbers (indices into the pointer array) themselves to remain the same for each operator while allowing the tag number to reference a private data area for each operator.

Shared Domain

FactoryLink tasks also use elements to monitor and control the state of the manufacturing process. These database elements must be the same for all users. The subset of the real-time database shared by all users is known as the SHARED domain.

To share specific elements among all authorized users, each user's pointers must map database segments for shared elements to the same physical memory. When a new operator area is created, the common element pointers are copied from a master copy associated with the first user who logged on to the application.

Application Instances/ Identification

Since multiple copies of a FactoryLink task may be running concurrently, a task must identify itself to the kernel. The task must specify the application, domain and user instance as well as the task name.

Each application instance is uniquely specified by its invocation name. The invocation name is used to locate the shared memory segment containing the kernel's global data structure; it uniquely identifies an instance of the FactoryLink real-time database. The invocation name is stored by the Run-Time Manager in the environment variable {FLNAME}.

The domain an application instance is associated with is specified by the domain name. The kernel uses the domain name to determine which real-time database segments the task owns and which should be shared; the task uses the domain name to determine which CT files should be processed. The domain name is stored by the Run-Time Manager in the environment variable {FLDOMAIN}.

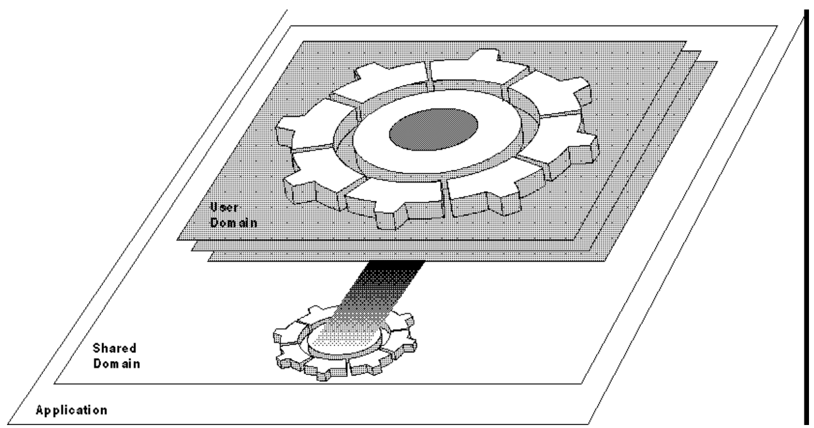
The user instance is specified by the user name. The user name determines which instance of the domain-specific real-time database segments the task is to use. The user name is stored by the Run-Time Manager in the environment variable {FLUSER}.

When a FactoryLink application is started, the Run-Time Manager must be supplied with the invocation name, domain name and user name. These may be supplied on the command line at run time by an operator or automatically set through environment variables. The three required environment variables are created by the Run-Time Manager if they do not exist. The values passed to the Run-Time Manager are stored in the environment of any sub-processes created by the Run-Time Manager.

Application Design Considerations

FactoryLink supports the concept of shared and private data areas by the separation of data access into domains. Every application has access to the shared domain but only the set of users specifically designated by the developer may open a particular USER domain. An application developer associates tasks, tables, and elements with a specific domain. These domain associations determine how an application functions at run time.

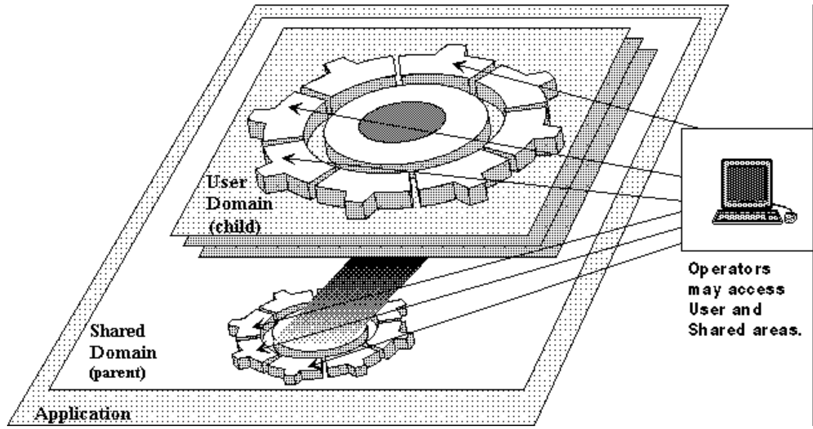
Domains may be nested, resulting in a parent-child hierarchical domain set. Any information or task existing in a shared domain (parent) is available to all operators within a USER domain (child). A graphical representation of the relation between these sets is shown on the following page.



The application developer chooses the shared domain to associate globally accessible tasks, tables, and elements. An operator running an application in a shared domain has access to the shared tasks and data only.

This global functionality can reduce configuration time since many items can be defined in the shared area. In addition, changes made to data in the shared domain by any task are instantly known to all tasks accessing the shared data.

The USER domain specifies those tasks, tables, and elements that will be defined as private to an individual operator. These elements are explicitly associated together under a domain designation (analogous to a user name) and are marked as the private property of that operator. As previously noted, domains are structured hierarchically. This parent-child structure allows an operator running an application copy (domain instance) in the User domain to open tasks and data within that domain and its parent, the shared domain. The following illustration is a graphical representation of this relationship.



During the installation process, default domain associations are made in the System Configuration Table. These defaults determine which domain a task is associated with at run time. Some tasks are associated with both shared and operator domains. During application planning, the developer should review the default domain associations to verify that they are compatible with current needs. If an application has special requirements, these default associations may be changed.

In addition to these run-time domain associations, the Configuration Manager has a domain selection feature that must be set before the tasks are configured. If the run-time domain associations do not match the configuration domain associations, the application will not run as intended.

Refer to Chapter 5, “Converting Database Tables to CTs” for information about changing a task's default domain association and selecting a domain for a task.

Domain Associations

Domain designations are critically important. If an application is not associated properly with defined domains, the application will not function properly at run time. Before an application begins to run, the associated domain is defined as an environment variable. This variable must match one of the previously defined domain values. Two types of domain relationships are possible:

- An application may be designed to run as shared. All tasks, tables, and database elements are associated with the shared domain. At run time, the application runs as a shared entity with one set of tasks and elements used by all users.
- An application may be designed to run as user. All operators have a private copy (domain instance) of the application. This domain instance appears to the operator as the only copy of the application running on the system; in other words, the mechanics of this data sharing are transparent to the operator. Tasks, tables, and elements may be associated with both the shared and operator domains. At run time, all domain instances are identical when generated; however, individual operator actions are independent, and the data created is stored in a real-time database unique to each operator.

> **Note:** Currently, there may be only one instance of the shared domain while there are usually several occurrences of user domains.

Refer to Chapter 7, “FactoryLink Kernel and Library” for further information and a discussion of the FactoryLink kernel's handling of shared and user domain data.

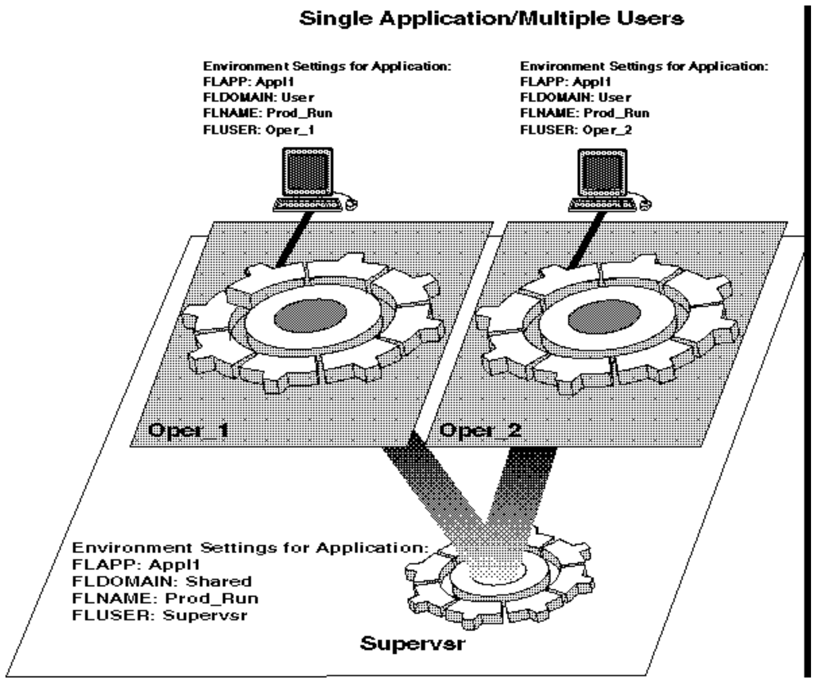
Application Example

Numerous application configurations are possible. The following example illustrates one way in which a FactoryLink application may be configured.

The environment settings shown in the following illustrations are used at run time. Refer to “FactoryLink Environment Variables” on page 53 for additional information on these settings.

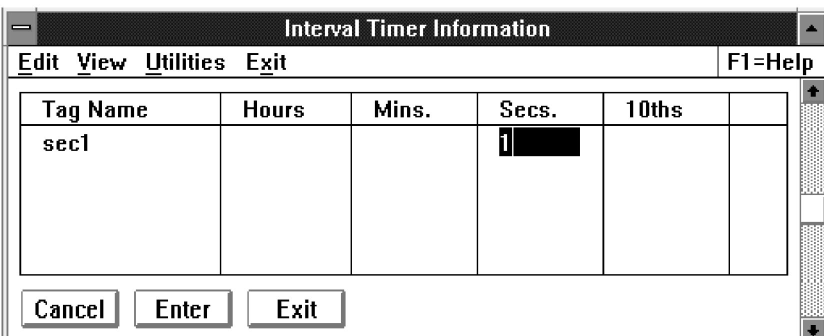
Single Application/Multiple Users

In the following example, an application developer has created an application that allows multiple operators to run separate invocations of the same FactoryLink application in which each operator has his own tasks and real-time database.



Configuring FactoryLink

Developers configure FactoryLink by filling in configuration tables associated with different FactoryLink tasks. The developer defines one or more elements, assigning each a symbolic element name and enters other static information required to define the function of the element(s). For instance, in the Interval Timer Information panel that follows, the developer defined a element with the symbolic element name sec1 and a value specifying the length of time between intervals. The element name serves to link the interval timer element to other FactoryLink tasks by making the state of the timer element accessible by other tasks.

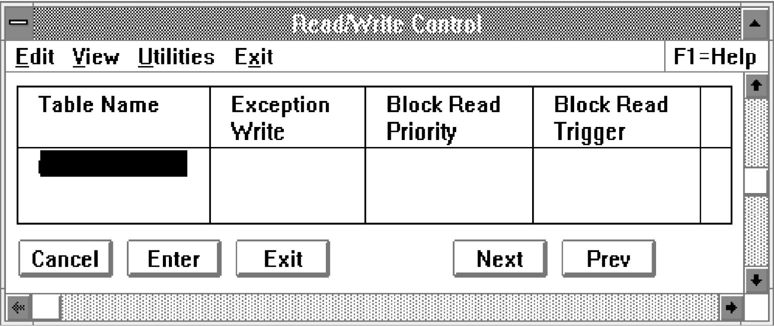


This information allows FactoryLink tasks to read data from and write data to the real-time database, providing developer-configurable communication links between FactoryLink tasks. Establishing these links is the essence of creating a FactoryLink application.

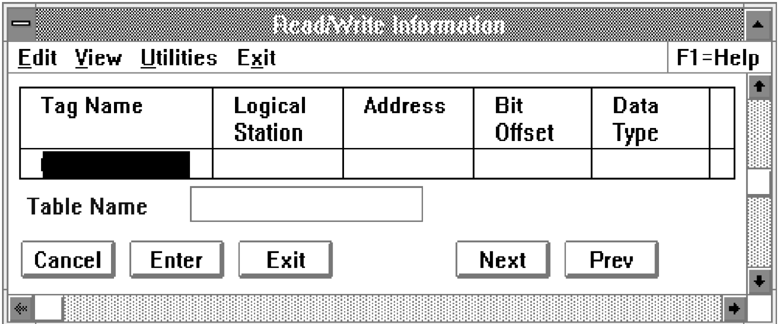
A developer enters information in a table by opening one or more screen displays, called panels, that provide predefined entry fields for information required for the task to function.

In general, there are two types of panels: control and information. In a control panel, the application developer enters information that identifies and/or initiates an operation, such as a read or write. In an information panel, the application developer enters specific information about the values used in the operation defined in the control panel. This is how the control and information panels for a particular task are related. A task may require both control and information panels or it may need only an information panel.

For instance, in the sample control panel that follows, the developer will specify types and priorities of read and/or write operations performed.



Then, in the following sample Information panel, which is associated with the previous Read/Write control panel, the developer will enter the PLC registers read or written and the names of elements that will receive or supply information from or to the PLC.



A task may require both Control and Information panels, or just an Information panel. A help screen can be selected with a single keystroke.

How FactoryLink Tasks Transfer Data

Data transfer in FactoryLink is based on reads, writes, and change reads of information in the real-time database. All information used by FactoryLink tasks, even if it is collected from or transmitted to outside sources, goes through the real-time database on its way to its final destination.

Any task can use any element in the FactoryLink real-time database as long as the data type of that element matches the type required for the indicated operation. It is possible— even beneficial— for two tasks to use the same element. When this occurs, both tasks share the information stored in the element.

Transfer Methods

FactoryLink transfers data in three ways:

- **Reading data from the real-time database:** A task may read the value of an element in the real-time database to display it on a screen, transmit it to an external device such as a PLC, or to send it along to another task.
- **Writing data to the real-time database:** A task may write information to an element in the real-time database as the result of an operator input, a read of an external device such as a PLC, or an input from another FactoryLink task.
- **Change-read operations:** A change-read operation is a scattered block read of the real-time database that checks only those elements whose values have changed since the last read operation. This is known as exception processing. Exception processing is possible because of the flags contained in the structure of FactoryLink elements. Because large blocks of data can be transferred between tasks in this way and because only the changed values are processed, change-reads optimize system performance.

Data Transfer Examples

Reads, writes, and change-reads are generally performed in combinations.

Example 1: When an operator enters a new value from the keyboard, the Graphics task reads the value and writes it to the element associated with the input object.

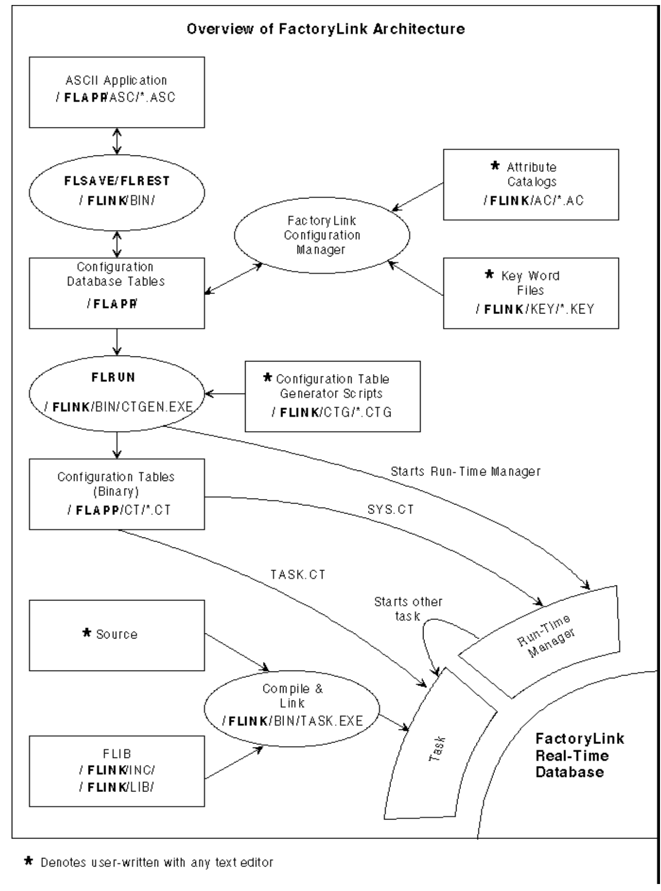
Example 2: The Graphics task uses a change-read function which executes in a loop to determine whether element values linked to a screen display have changed. If a value of the element has not changed, the function informs the task of that fact. If a value of the element has changed, the function returns the new value from the real-time database. At the completion of the execution of the loop, the task relinquishes any unused time to the next task.

Example 3: The Batch Recipe task can be configured to read values from a file and write them to associated elements in the real-time database or read elements from the database and write their values to a file on the disk.

Example 4: A programmable controller task reads values from the programmable controller and writes them to the real-time database. It can also read values from the database and write them to the programmable controller.

How FactoryLink Architecture Affects New Task Development

The following figure illustrates FactoryLink architecture.



The following chart describes each part of the illustration, beginning with the upper left portion:

Table 2-3

Item in Illustration:	Description:
ASCII Application	Any FactoryLink application can be saved to enable the importing/exporting of configuration data from one application directory to another (typically, from one platform to another).
FLSAVE/FLREST	Utilities used to save/restore a FactoryLink application. For additional information about these utilities, refer to "FactoryLink Utilities" in the *FactoryLink Fundamentals* guide.
Configuration Database	Tables that store information about the real-time data-**Table** base elements. You must design these tables as part of task development. In this manual, "configuration database tables" are referred to simply as "database tables."
FactoryLink Configuration Manager	Development system tool used to set up, or configure, a FactoryLink application. Used by programmer to test configuration.
Attribute Catalogs	Programmer-created ASCII text files that describe the database tables and the editing criteria for operator-entered data.
Key Word Files	ASCII text files that tell the Configuration Manager how to translate text table entries into binary values. You must create these files if an appropriate one does not already exist.
FLRUN	Command typed by the operator to execute the FactoryLink Software System, including the custom application. This command starts the Run-Time Manager task.
Configuration Table Generator Scripts	Programmer-created script files that tell the CTGEN utility (generate binary CT files) how to extract data from the database tables and combine it to produce a binary Configuration Table (CT) file at run time.

Table 2-3

Item in Illustration:	Description:
Configuration Tables (Binary)	Binary files produced by the CTGEN utility at run time containing data extracted from the database tables.
Source	Programmer-developed source code for a task.
Compile and Link	Step you perform to compile source code into object code and link object modules.
FLIB	FactoryLink library. Collection of utility functions serving primarily to interface application and system programs to the FactoryLink kernel.
Task	FactoryLink task that performs a required operation. Up to 32 FactoryLink tasks, including the Run-Time Manager, may be active per domain instance at once.
Run-Time Manager	FactoryLink task that monitors and controls other FactoryLink tasks that are using the services of the real-time database.
FactoryLink Real-Time Database	Memory-resident array of information that exists only at run time.

You are directly affected only by portions of the FactoryLink architecture. You must design the task's database tables; these tables contain all information required to completely and unambiguously describe the application with the possible exception of operator-entered menu selections and/or commands.

Once the design is final, you must create an attribute catalog file for one or more database tables. The attribute catalog file represents one menu option on the Configuration Manager Main Menu. If the attribute catalog file references a KEY file, the developer must create a KEY file that provides the Configuration Manager with information on developer-entered key words placed in the database tables. Refer to Chapter 4, “Setting up the Configuration Environment” in this manual for details about creating attribute catalog files and KEY files.

Then, you must open the Configuration Manager and use it to test whether the attribute catalog file(s) and, if used, the KEY files, accurately reflect the desired database table design and editing requirements.

At run time, the Run-Time Manager starts and the database tables are converted into one or more sorted binary files known as a CT file(s). When a FactoryLink task initially begins running, it reads its CT file to determine the elements to open and the actions to perform on those elements. You must write a configuration table generator script that describes how to extract data from the database tables and produce the CT file at run time.

The custom-developed program describes how the custom-developed task functions in relation to other tasks, including the Run-Time Manager.

Refer to Chapter 3, “Constructing a Task” for a list of the steps involved in task construction.

FactoryLink Triggers

Many FactoryLink tasks use digital elements to trigger certain actions; events such as read or write operations can be configured to occur as the result of a combination of bit value and change-status flag value in a trigger element. A trigger element can be used as a trigger by more than one task.

For an element to trigger an event, two conditions must exist:

- The value of the element must be 1.
- The value of the change-status flag being read by the task must be 1.

When a task reads a trigger element, if the element's value is 1 and has changed since the last time it was read, the reading task sets its change-status flag to **0** and begins the operation designated by the trigger.

Force-writing a 1 to a digital element, even though it may not change the actual value of the element (if the element was already equal to **1)**, causes the events tied to that trigger to be triggered during the next read operation. If multiple tasks are to use the same element as a trigger, use of the forced-write technique simplifies inter-task handshaking requirements.

Trigger elements can be created and used in many ways:

- Interval and event timer digital elements can be defined in the real-time database for use as triggers.
- Function keys can be configured as triggers.
- The Math and Logic task can create triggers of any data type to initiate a Math and Logic Procedure. The procedure is executed when the element's value changes to 1 (for a digital element) or whenever the element's value changes (for all other types of elements).
- A trigger can generate a report or data log.
- A trigger can be used to read and write recipes.
- A trigger can be used to execute another program.
- A trigger can cause a screen to be printed.
- A trigger can initiate a network transfer.
- A trigger can cause a new setpoint to be downloaded to a programmable controller.
- Each polled read function used by a programmable controller task can be triggered separately.

FactoryLink Files and Directories

This section discusses the variables used to set up the FactoryLink environment, the path name format used in this manual, and the FactoryLink directory organization.

FactoryLink Environment Variables

FactoryLink tasks depend on the values of various environment variables to determine many aspects of their interactions with files and Real-Time Database data structures. Whenever a task is started, whether manually or automatically by the Run-Time Manager, the following environment variables must always be set to appropriate values.

```
{FLINK}    base directory of FactoryLink installation
{FLAPP}    base directory of configuration files
{FLLANG}   language environment
{FLNAME}   invocation name
{FLDOMAIN} domain name
{FLUSER}   user instance name
```

Several details about the variables follow.

Two environment variables are used to define the two basic directories containing FactoryLink files. The first variable, designated as /{FLINK} in this manual, corresponds to the name of the directory containing the FactoryLink program files. The second variable, designated as /{**FLAPP**} in this manual, corresponds to the name of the directory containing the application-related files.

During installation, these variables are set to default values, except where noted for domain-setting-related variables.

Any references to these two variables in this manual are indicated by boldface type, such as /**{FLINK}**.

- **{FLINK}** is the name of the directory containing the FactoryLink program files. If this variable is not defined, it defaults to /**{FLINK}**.
- **{FLAPP}** is the name of the directory containing the application-related files. If this variable is not set, it defaults to /**{FLAPP}**.

During installation, the developer can specify values for these variables. Otherwise, standard defaults are used, except as noted. Refer to "Operating System Notes" on page 63 for operating system-specific notes.

- **{FLLANG}** is an optional environment variable that defaults to the English language.
- **{FLNAME}** is the name of the application.
- **{FLDOMAIN}** is the name of the domain under which the application is designed to run.
- **{FLUSER}** is the name of the domain instance.
- **{FLOPT}** is an optional environment variable which points to the license directory. It defaults to **{FLOPT}**\opt.

> **Note:** There are no defaults defined for these last three variables. The values are left blank to ensure you invoke the correct domain instance. You must ensure that these variables are set to valid values before tasks are started, either through operator interaction or by means of reading files.

Use of Environment Variables in FactoryLink Path Names

Additional Functionality

The FL_PATH library functions have been modified since the previous release to allow specification of environment variables in a multi-platform path. An environment variable is indicated by placing the environment variable name inside braces {}. For example, assuming that {**FLAPP**} is set to /users/{FLINK}/{**FLAPP**} and {FLDOMAIN} is set to shared, the multi-platform path {**FLAPP**}/{FLDOMAIN}/ct will be expanded to /users/{FLINK}/{**FLAPP**}/shared/ct. This macro expansion takes place automatically whenever braces are used around the name of a valid environment variable within a path name. The following API functions will recognize and expand environment variables. (Refer to Chapter 8, "FactoryLink API Reference Guide" for the specifications of these functions.)

```
fl_path_norm
fl_path_add_dir
fl_path_set_dir
fl_path_set_file
fl_path_set_pattern
fl_path_set_node
fl_path_set_device
```

Effects on Existing Tasks

The existing functionality of the FL_PATH functions is completely compatible with that of previous versions, with the exception that braces are now considered meta-characters. Refer to the discussion of environment variable name expansion in "Additional Functionality" on page 54. Existing FactoryLink custom-developed tasks which are already running correctly need not be changed. However, use of the environment variable expansion capability should be incorporated into existing tasks at the developer's convenience to ensure continued compatibility with future releases of the system software.

Conversions should be fairly obvious and straightforward after a bit of study of the new functionality. For example, most tasks presently use sprintf() to include the domain name in the path name for CT files. The code that combines the domain name with the CT file path can be removed and a string of the form {FLDOMAIN}/ct/file.ct can be passed to CT_OPEN instead.

Path names read from CT files may also include environment variable specifications. This allows the operator to specify a path name relative to some directory that varies from system to system or operator to operator. FactoryLink tasks should not attempt to parse path names. The FL_PATH functions should be used to build and modify path names.

Path Name Building and Representation

Paths are represented as either a system-dependent character string, or as a data structure containing the parts of the path as individual strings. The data structure form is called a normalized path name.

A normalized path contains the following components:

- Node name
- Device
- Directory
- File name
- Wild card pattern
- File date and time
- File size
- File type
- {System-dependent information}

The C structure for a normalized path is predefined as:

```
typedef struct _npath
{
    char    node[MAX_NODE_NAME];
    char    device[MAX_DEVICE_NAME];
    char    dir[MAX_DIR_NAME];
    char    file[MAX_FILE_NAME];
    char    wild[MAX_FILE_NAME];
    char    version[MAX_VERSION];
    char    verwild[MAX_VERSION];
    long    dt;
    long    size;
    int     type;
    int     magic;
    void   *sysdata;
} NPATH;
```

Path Name Format

The complete path name for a file has the following format in this manual:

```
DISK:/DIRECTORY/SUBDIRECTORY/FILENAME
```

Partial path names for a file follow the same format:

```
/DIRECTORY/SUBDIRECTORY/FILENAME
```

Normalized paths are new in the current release of FactoryLink. The new path functions listed below and documented in “FactoryLink API Reference Guide” on page 183 of this manual have been added to FLIB to eliminate system-specific path name dependencies and allow porting of code to different platforms without changes. FactoryLink Multi-platform code should use only these functions to generate path names, open file information, and search directories. The FactoryLink path functions are alphabetically listed below.

```
FL_PATH_ACCESS
```

```
FL_PATH_ADD
FL_PATH_ADD_DIR
FL_PATH_ALLOC
FL_PATH_CLOSEDIR
FL_PATH_CREATE
FL_PATH_CWD
FL_PATH_DATE
FL_PATH_FREE
FL_PATH_GET_SIZE
FL_PATH_GET_TYPE
FL_PATH_INFO
FL_PATH_MKDIR
FL_PATH_NORM
FL_PATH_OPENDIR
FL_PATH_READDIR
FL_PATH_REMOVE
FL_PATH_RMDIR
FL_PATH_SET_DIR
fl_path_set_device
FL_PATH_SET_EXTENSION
FL_PATH_SET_FILE
FL_PATH_SET_NODE
FL_PATH_SET_PATTERN
FL_PATH_SET_VERSION
FL_PATH_SYS
FL_PATH_TIME
```

It is essential that you call these functions instead of attempting to hard-code or build (using string functions) path names into their programs if you plan on porting the applications to other systems and in order to retain compatibility with future releases of FactoryLink. Refer to “Operating System Notes” on page 63 for operating system-specific notes. For specific examples (for informational purposes only) of how this formula translates to the path name format accepted by a particular operating system, The following examples briefly explain the purpose of each function and any special information about each. Here, the functions are grouped by functionality and purpose. Refer to Chapter 8, “FactoryLink API Reference Guide” for reference material on each function.

Example 1. Allocating a Normalized Path

```
NPATH *fl_path_alloc(void)
```

The function FL_PATH_ALLOC allocates and returns a pointer to a normalized path buffer. This function should be called rather than allocating the NPATH structure directly so that a buffer for system-dependent information can be added to the path buffer.

```
void   fl_path_free(NPATH *p)
```

The function FL_PATH_FREE releases the space allocated by a call to FL_PATH_ALLOC. The NPATH structure should not be accessed after FL_PATH_FREE is called; such an attempt will lead to unpredictable behavior of the system.

Example 2. Converting To/From a Normalized Path

```
char *fl_path_sys(NPATH *p, char *path, size_t length)

NPATH *fl_path_norm(NPATH *p, char *path)

NPATH *fl_path_set_dir(NPATH *p, char *dir)

NPATH *fl_path_cwd(NPATH *p)
```

The function FL_PATH_SYS converts a normalized path into a system specific path string. If the path argument is null, FL_PATH_SYS calls malloc**()** to allocate memory for the resulting path. The caller should call free to release the memory when it is no longer in use.

FL_PATH_NORM converts a system-specific path string into a normalized path. Alternatively, the FL_PATH_SET_DIR function can be used if the path is known to refer to a directory.

FL_PATH_SET_DIR replaces the directory portion of the path. The directory argument is converted to normalized form. If the NPATH argument is NULL, FL_PATH_SET_DIR first calls FL_PATH_ALLOC to allocate a NPATH buffer. The file name, extension and version are not modified by the FL_PATH_SET_DIR function.

FL_PATH_CWD builds a normalized path for the current working directory. If the NPATH argument is NULL, FL_PATH_CWD first calls FL_PATH_ALLOC to allocate a NPATH buffer.

Example 3. Modify an Existing Path

```
void fl_path_add(NPATH *p1, NPATH *p2)
void fl_path_add_dir(NPATH *p, char *dir)
void fl_path_set_file(NPATH *p, char *file)
void fl_path_set_pattern(NPATH *p, char *pattern)
void fl_path_set_node(NPATH *p, char *node)
void fl_path_set_device(NPATH *p, char *drive)
void fl_path_set_extension(NPATH *p, char *extension)
void fl_path_set_version(NPATH *p, char *version)
```

FL_PATH_ADD catenates two paths. Any missing component of the second path p2 is taken from the first path p1 or from the current directory if the first path is null.

FL_PATH_ADD_DIR adds a subdirectory specification to the end of the directory portion of a path. Only one subdirectory can be added to a path during each call to FL_PATH_ADD_DIR. The subdirectory name should not contain any path-separator characters.

FL_PATH_SET_FILE replaces the file name portion of the path.

FL_PATH_SET_PATTERN sets the wild card pattern for subsequent directory search.

FL_PATH_SET_NODE replaces the node name portion of the path.

fl_path_set_device replaces the drive portion of the path.

FL_PATH_SET_EXTENSION replaces the extension of the current file name portion of the path.

Example 4. Create/Delete Paths

```
int fl_path_mkdir(NPATH *p)
```

```
int fl_path_rmdir(NPATH *p)
int fl_path_create(NPATH *p)
int fl_path_remove(NPATH *p)
```

FL_PATH_MKDIR creates the directory given by the directory portion of the path.

FL_PATH_RMDIR deletes the directory given by the directory portion of the path.

FL_PATH_CREATE creates an empty file using the complete path.

FL_PATH_REMOVE removes the file specified by the complete path.

.Example 5. Search for Matching Files

```
int fl_path_opendir(NPATH *p)
int fl_path_readdir(NPATH *p)
void fl_path_closedir(NPATH *p)
```

FL_PATH_OPENDIR begins a directory search operation. The current directory information contained in NPATH is used as the directory to search and the current wildcard pattern is used to choose files. FL_PATH_OPENDIR returns GOOD if the directory could be opened for search, or ERROR if it could not. FL_PATH_OPENDIR reads the first entry in the directory.

FL_PATH_READDIR reads the next matching file in the directory and places the name of the file into the file name component of the path. The file type, date, time, and size are also stored in the NPATH structure. FL_PATH_ READDIR returns GOOD if a matching file was found or ERROR if not.

FL_PATH_CLOSEDIR ends a directory search. The following code fragment in C demonstrates how to use directory search functions to print a directory listing.

```
NPATH *p;
char date[80];
char time[80];

char fullpath[MAX_PATH_NAME];

   p = fl_path_norm(NULL, “*.*”);
   if (fl_path_opendir(p) == ERROR)
   {
   printf(“Directory Not found\n”);
   return;
}
```

```
   do
   {
        fl_path_date(p, date,80);
        fl_path_time(p, time,80);
        fl_path_sys(p, fullpath,MAX_PATH_NAME);
          printf("%s %s %s\n", date, time, fullpath);
   } while ( fl_path_readdir(p) != ERROR );
   fl_path_closedir(p);
   fl_path_free(p);
```

Example 6. Get File Information

```
int fl_path_info(NPATH *p)
long fl_path_date(NPATH *p, char *buf,size_t length)
long fl_path_time(NPATH *p, char *buf,size_t length)
int fl_path_access(NPATH *p)
char *fl_path_get_device(NPATH *p)
char *fl_path_get_node(NPATH *p)
char *fl_path_get_dir(NPATH *p)
char *fl_path_get_file(NPATH *p)
long fl_path_get_dt(NPATH *p)
long fl_path_get_size(NPATH *p)
int fl_path_get_type(NPATH *p)
```

FL_PATH_INFO initializes the date, time, size, and type for the path. If the file does not exist, FL_PATH_INFO returns ERROR. Otherwise, it returns GOOD. This function is called automatically by FL_PATH_OPENDIR and FL_PATH_READDIR.

FL_PATH_DATE formats the date of a file into the caller's buffer and returns the date and time (concatenated) as a long integer.

FL_PATH_TIME formats the time of the file into the caller's buffer. This function is operating system-dependent.

FL_PATH_SIZE returns the size of a file in bytes.

FL_PATH_TYPE returns the type of the file. A string constant is returned from among the valid types; these are operating-system dependent.

FL_PATH_ACCESS returns NPATH_READ, NPATH_WRITE, or NPATH_READ | NPATH_WRITE, depending on the file access mode. If the file does not exist, FL_PATH_ACCESS returns ERROR.

FactoryLink Directory Organization

Refer to "Operating System Notes" on page 63.

Operating System Notes

The following sections provide operating system specific information relevant to this chapter.

For Windows/NT Users

FactoryLink Environment Variables (page 53)

Unless otherwise specified by the developer, the default values for the FactoryLink environment variables follow:

Table 2-4

Environment Variable	Default Directory
{**FLINK**}	\{**FLINK**}
{**FLAPP**}	\{**FLAPP**}

Path Name Format (page 56)

For Microsoft Windows, the platform-independent path name format of

```
DISK:/DIRECTORY/SUBDIRECTORY/FILENAME
```

translates to the following format:

```
DRIVE:\DIRECTORY\SUBDIRECTORY\FILENAME
```

For example, if a file named TIMER.CT is located in a subdirectory named CT in the **{FLAPP}** directory, the file name in the *Programmer's Access Kit* guide is specified as **{FLAPP}**/CT/TIMER.CT. The Microsoft Windows file name would have the format, **{FLAPP}**\CT\TIMER.CT. The boldfaced type on **{FLAPP}** indicates this refers to the default or operator-specified name of the application-related directory specified during installation.

(2-33)

The FL_PATH functions operate under Microsoft Windows as stated with the following additional rules:

- If a Microsoft Windows-style path string is supplied to FL_PATH_NORM, FactoryLink assumes that any name following the last slash character is a file name.

- If a path refers to a directory, add a trailing slash to the path name for Microsoft Windows-style names.
- If the path is known to refer to a directory, the FL_PATH_SET_DIR function can be used instead.

For OS/2 Users

FactoryLink Environment Variables (page 53)

Unless otherwise specified by the operator, the default values for the FactoryLink environment variables are listed below:

Table 2-5

Environment Variable	Default Directory
{FLINK}	flos2
{**FLAPP**}	**flapp**

Use of Environment Variables in FactoryLink Path Names (page 54)

The FL_PATH functions operate under OS/2 as stated, with the following additional rules: when an OS/2-style path string is supplied to FL_PATH_NORM, it is assumed that any name following the last slash character is a file name. If it is known that the path refers to a directory, a trailing slash can be added to the path name for OS/2-style names. As stated, the FL_PATH_SET_DIR function can be used instead if the path is known to refer to a directory.

Path Name Format (page 56)

For OS/2, the platform-independent path name format of

```
DISK:/DIRECTORY/SUBDIRECTORY/FILENAME
```

translates to the following format:

```
DRIVE:\DIRECTORY\SUBDIRECTORY\FILENAME
```

For example, if a file named TIMER.CT is located in a subdirectory named CT in the /**{FLAPP}** directory, the file name in the *Programmer's Access Kit Manual* is specified as /**{FLAPP}**/CT/TIMER.CT. The OS/2 file name would have the format, **{FLAPP}**\CT\TIMER.CT. The boldfaced type on **{FLAPP}** indicates that this

refers to the default or operator-specified name of the application-related directory specified during installation.

Example 6. Get File Information (page 61)

The function FL_PATH_TIME formats the time of the file into the caller's buffer. This function is operating system-dependent. Under OS/2, the format is controlled by the country code set in the CONFIG.SYS file.

FactoryLink Directory Organization (page 62)

The files are organized functionally, as follows:

FactoryLink Program Directory (\{FLINK})

All of the following files are located in the **{FLINK}** directory. The subdirectories containing program files are organized as follows:

Table 2-6

Subdirectory	File(s)	Description of File(s)
\AC	*.AC	Text files that function as attribute catalogs to inform the Configuration Manager of the format of the database tables. They also determine editing entry criteria.
\BIN	*.CMD	FactoryLink command files
	*.EXE	Executable program files for each FactoryLink task
\BLANK		A blank **{FLAPP}** directory. Used by the FLBLANK utility to create a new application.
\CML		Compiled Math and Logic files
\CTGEN	*.CTG	CT script files
\DRW	*.G	Files used by Gedant and run-time graphics
	*.GP	

Table 2-6

Subdirectory	File(s)	Description of File(s)
\EDI		Subdirectory for External Device Interfaces (PLC drivers)
\INC	*.H	Header files for C programs
\INSTALL		Files used during FactoryLink installation
\KEY	*.KEY	Text files that tell the Configuration Manager how to translate text table entries into binary values to be placed in a configuration table (CT)
\LIB	*.LIB	Library files
	*.DLL	Dynamic link library files
\MPS	*.MPS	Multiplatform save files
\MSG	*.TXT	Message files for FactoryLink tasks
\OPT	FL.OPT	File containing FactoryLink options information
\SRC		Programmer's Access Kit files

FactoryLink Application Directory (\{FLAPP})

All of the following files are located in the \{FLAPP} directory. The subdirectories containing program files are organized as follows:

Table 2-7

Subdirectory	File(s)	Description of File(s)
	*.CDB	Database tables that store information about the elements such as name, type, number of definitions (number of writes specified by the defining task), and number of references

Table 2-7

Subdirectory	File(s)	Description of File(s)
	*.MDX	Indexes used by the Configuration Manager in conjunction with the CDBs to translate element names to element numbers at run time
\ASC	*.ASC	ASCII database tables that store information about the real-time database elements. They are used to import/export configuration data from one application directory to another (typically, from one platform to another platform).
	*.EXP	Output of CM export
\CT	*.CT	Binary configuration tables. Each FactoryLink program employs one or more configuration tables.
\DRW	*.DRW	Graphics files created with the Application Editor
	*.G	Run-time graphics files in U.S.DATA format
\LOG		Error log files produced by FactoryLink processes at run time containing debug information
\NET	GROUPS	Groups on this node
	LOCAL	FL/LAN information
\PROCS	*.PRG	Math and Logic procedures
\RCP		Files created by the Batch Recipe task
\RPT	*.RPT	Report files generated by the Report Generator task
\SPOOL		Subdirectory used by the FactoryLink Print Spooler task

For UNIX Users

Element names (page 28)

Under UNIX, the developer can create element arrays containing up to 64K (65,534) in elements per element array, depending on available memory.

FactoryLink Environment Variables (page 53)

Unless otherwise specified by the operator, the default values for the FactoryLink environment variables follow.

Table 2-8

Environment Variable	Default Directory
{FLINK}	/usr/flink
{**FLAPP**} }	/usr/**flapp**

Note: Unix default directories may vary according to the way that the Unix system administrator has configured the file system. On some systems, /usr may be /users or another variant; on other systems, it may be different for each operator. Check with the system administrator to determine the exact operator account directory set up for FactoryLink users; the system defaults to that operator account directory.

Use of Environment Variables in FactoryLink Path Names (page 54)

The FL_PATH functions operate under OS/2 with the following additional rules: when a UNIX-style path string is supplied to FL_PATH_NORM, any name following the last slash character is a file name. If it is known that the path refers to a directory, a trailing slash can be added to the path name for UNIX-style names. As stated, the FL_PATH_SET_DIR function can be used instead if the path is known to refer to a directory.

Example 6. Get File Information (page 61)

FL_PATH_TIME formats the time of the file into the caller's buffer. This function is operating system-dependent. Under Unix, the date/time string is formatted using the standard library function strftime. The format is thus modified by the setlocale() function.

Path Name Format (page 56)

For UNIX, the platform-independent path name format of

```
DISK:/DIRECTORY/SUBDIRECTORY/FILENAME
```

translates to the following format:

```
/directory/subdirectory/filename
```

For example, if a file named TIMER.CT is located in a subdirectory named CT in the /{**FLAPP**} directory, the file name in the *Programmer's Access Kit* guide is specified as /{**FLAPP**}/CT/TIMER.CT. The UNIX file name has the format /{**FLAPP**}/ct/timer.ct. The boldfaced type on {**FLAPP**} indicates that this refers to the default or operator-specified name of the application-related directory specified during installation.

Although colons are valid characters in file names under most UNIX installations, FactoryLink PAK modules should not use file names that include colons. Due to the multi-platform nature of FactoryLink and the need for portability, colons (":") in file names cause the FactoryLink system to interpret the portion of the name preceding the colon as a device name (currently ignored under UNIX); for example, "/tmp/ava:balt" will be seen as "/tmp/balt". The system will always assume anything preceding a ":" in a file name is a device name and will skip it. Therefore, do not place colons in file names.

Note specifically that in calls to make_full_path(), slash_to_norm(), and fl_path_norm() the returned path and file names will not be as expected if passed a file name containing a colon; they work as expected when file names without colons are passed in.

FactoryLink Directory Organization (page 62)

The files are organized functionally as follows:

FactoryLink Program Directory (/{FLINK})

All of the following files are located in the /{FLINK} directory. The subdirectories containing program files are organized as shown in the following chart.

Table 2-9

Subdirectory	File(s)	Description of File(s)
\AC	*.AC	Text files that function as attribute catalogs to inform the Configuration Manager of the format of the database tables. They also determine editing entry criteria.
\BIN	*.CMD	FactoryLink command files
	*.EXE	Executable program files for eachFactoryLink task
\BLANK		A blank **{FLAPP}** directory. Used by the FLBLANK utility to create a new application.
\CTGEN	*.CTG	CT script files
\CML		Compiled Math and Logic files
\DRW	*.G	Files used by Gedant and run-time graphics
	*.GP	
\EDI		Subdirectory for External Device Interfaces (PLC drivers)
\INC	*.H	Header files for C programs
\INSTALL		Files used during FactoryLink installation
\KEY	*.KEY	Text files that tell the Configuration Manager how to translate text table entries into binary values to be placed in a configuration table (CT)

Table 2-9

Subdirectory	File(s)	Description of File(s)
\LIB	*.LIB	Library files
	*.DLL	Dynamic link library files
\MPS	*.MPS	Multiplatform save files
\MSG	*.TXT	Message files for FactoryLink tasks
\OPT	FL.OPT	File containing FactoryLink options information
\SRC		Programmer's Access Kit files

FactoryLink Application Directory (/{FLAPP})

All of the following files are located in the **{FLAPP}** directory and/or the **{FLAPP}**\{DOMAIN} directory where the {DOMAIN} is either SHARED or USER. The subdirectories containing program files are organized as follows:

Table 2-10

Subdirectory:	File(s):	Description of File(s):
	*.CDB	Database tables that store information about the elements, such as name, type, number of definitions (number of writes specified by the defining task), and number of references
	*.MDX	Indexes used by the Configuration Manager in conjunction with the CDBs to translate element names to element numbers at run time
\ASC	*.ASC	ASCII database tables that store information about the elements. Used to import/export configuration data from one application directory to another (typically, from one platform to another).
	*.EXP	Output of CM export

Table 2-10

Subdirectory:	File(s):	Description of File(s):
\CML		Compiled Math and Logic files
\CT	*.CT	Binary configuration tables. Each FactoryLink program employs one or more configuration tables.
\DCT		External Device Interface CT files
\DRW	*.DRW	Graphics files created with the Application Editor
	*.G	Run-time graphics files in FactoryLink format
\LOG		Error log files produced by FactoryLink processes at run time containing debug information
\NET	GROUPS	Groups on this node
	LOCAL	FLLAN information
\PROCS	*.PRG	Math and Logic procedures
\RCP		Files created by the Batch Recipe task
\RPT	*.RPT	Report files generated by the Report Generator task.
\SPOOL		Subdirectory used by the FactoryLink Print Spooler task

The question marks (?) in the /{FLINK}/log log files for the Alarm Supervisor denote that a number is to be substituted. Use any number from 1 through 999.

Example 6. Get File Information

The function FL_PATH_TIME formats the time of the file into the caller's buffer. This function is operating system-dependent. Under Microsoft Windows, the format is controlled by the country code returned by the MS-DOS function 38h.

FactoryLink Directory Organization

Files are organized functionally as follows:

FactoryLink Program Directory ({FLINK})

The following files are located in the \{FLWIN} directory. Subdirectories containing program files are organized as follows:

Table 2-11

Subdirectory:	File(s):	Description of File(s):
\AC	*.AC	Text files that function as attribute catalogs to inform the Configuration Manager of the format of the database tables. Also determine editing entry criteria
\BIN	*.CMD	FactoryLink command files
	*.EXE	Executable program files for each FactoryLink task.
\BLANK		A blank **{FLAPP}** directory. Used by the FLBLANK utility to create a new application.
\CTGEN	*.CTG	CT script files
\CML		Compiled Math and Logic files
\DRW	*.G	Files used by Gedant and run-time graphics
	*.GP	
\EDI		Subdirectory for External Device Interfaces
\INC	*.H	Header files for C programs
\INSTALL		Files used during FactoryLink installation
\KEY	*.KEY	Text files that tell the Configuration Manager how to translate text table entries into binary values to be placed in a configuration table (CT)
\LIB	*.LIB	Library files

Table 2-11

Subdirectory:	File(s):	Description of File(s):
	*.DLL	Dynamic link library files
\MPS	*.MPS	Multiplatform save files
\MSG	*.TXT	Message files for FactoryLink tasks
\OPT	FL.OPT	File containing FactoryLink options information
\RPT	*.FRM	Report formats
\SRC		Programmer's Access Kit files

FactoryLink Application Directory ({FLAPP})

All of the following files are located in the **{FLAPP}** directory and/or the {**FLAPP**}\{DOMAIN} where the {DOMAIN} is either SHARED or USER. The subdirectories containing program files are organized as follows:

Table 2-12

Subdirectory:	File(s):	Description of File(s):
	*.CDB	Database tables that store information about the elements, such as name, type, number of definitions (number of writes specified by the defining task), and number of references
	*.MDX	Indexes used by the Configuration Manager in conjunction with the CDBs to translate element names to element numbers at run time
\ASC	*.ASC	ASCII database tables that store information about the elements. Used to import/export configuration data from one application directory to another (typically, from one platform to another).
	*.EXP	Output of CM export

Table 2-12

Subdirectory:	File(s):	Description of File(s):
\CT	*.CT	Binary configuration tables. Each FactoryLink program employs one or more configuration tables.
\DRW	*.DRW	Graphics files created with the Application Editor
	*.G	Run-time graphics files in FactoryLink format
\LOG		Error log files produced by FactoryLink processes at run time containing debug information
\NET	GROUPS	Groups on this node
	LOCAL	FLLAN information
\PROCS	*.PRG	Math and Logic procedures
\RCP		Files created by the Batch Recipe task
\RPT	*.RPT	Report files generated by the Report Generator task.
\SPOOL		Subdirectory used by the FactoryLink Print Spooler task
\DCT		External Device Interface CT files
\CML		Compiled Math and Logic files

Chapter 3 . . . .

Constructing a Task

This chapter suggests a three-phase procedure for constructing FactoryLink tasks and integrating them with the rest of the FactoryLink system.

Chapter 3 contains the following topics:

- Guidelines for Task Design
- Task Construction Procedure

Task Design Guidelines

We recommend you design FactoryLink tasks according to the architecture presented in this manual. Proper design enables the task to coexist efficiently with other FactoryLink programs. In addition, designing to the PAK guidelines provides an upgrade path to other operating systems.

The following guidelines enhance your FactoryLink design process:

- Tasks should use the FactoryLink configuration tools to describe work.

A FactoryLink task relies on the FactoryLink Configuration Manager in conjunction with the FactoryLink Application Editor to generate a file or files describing the work to be done at run time. This means you must:

  - Design one or more database tables.
  - Create one or more Attribute Catalogs (AC file) to describe the table(s).

- Tasks should be designed to run concurrently with FactoryLink system configuration.

A custom program runs concurrently and in conjunction with the FactoryLink system configuration we supplied as part of FactoryLink.

- Tasks should not interfere with other tasks.

A custom program should not adversely affect or otherwise interfere with concurrently running tasks, directly access system resources such as I/O devices which are under the control of the operating system, or severely degrade the performance of FactoryLink.

- Tasks should be designed for use with the FactoryLink Run-Time Manager.

Design the program for run-time use with the FactoryLink Run-Time Manager. The program must conform to certain requirements imposed by the Run-Time Manager in order for the latter to carry out its supervisory duties effectively. Refer to Chapter 6, "Using the Run-Time Manager" for a list of these requirements.

Task Construction Procedure

Although you are not required to follow any particular process when constructing tasks, we recommend you follow the top-down software design model as follows.

The top-down software design model for constructing and integrating a FactoryLink-compatible task consists of 3 major tasks:

- Setting up the Configuration Environment

  Refer to Chapter 4, "Setting up the Configuration Environment" for detailed information about setting up the configuration environment.
- Converting Database Tables to CTs

  Refer to Chapter 5, "Converting Database Tables to CTs" for detailed information about converting database tables to CTs.
- Writing the Task's Program

  Refer to the following chapters in this manual for detailed information about writing the task's program:

  Chapter 6, "Using the Run-Time Manager"

  Chapter 7, "FactoryLink Kernel and Library"

  Chapter 8, "FactoryLink API Reference Guide"

Setting up the Configuration Environment

Setting up the Configuration Environment includes 4 steps:

1. Design the Database Tables(s)
2. Create the Attribute Catalog(s)
3. Create the Key Files
4. Test the Configuration Environment

Design the Database Table(s)

Design the database table(s) for the task. This process involves laying out the panel(s), accessed from the Configuration Manager Main Menu in which the user enters configuration information for the task.

With the possible exception of operator selections and/or commands entered at run time, the database table contains all of the information needed by the task program to do its job. For example, it serves as the storage area for the user-entered inputs when the user selects this task from the Configuration Manager Main Menu. A table consists of one or more fixed-length database records. Each table is associated with an index to control the logical order of the records.

The task may require more than one table to describe the work to be performed. These database tables can be related (in a database sense) or unrelated to each other. Designing these tables involves mapping the work performed into a set of relational databases.

Create the Attribute Catalog(s).

Create at least one Attribute Catalog (with extension .AC) for each database table. The AC file describes the structure of the database table. In addition, it determines the user view of the table; that is, it lists the panel(s) displayed when the user chooses this task from the Configuration Manager Main Menu.

An AC file is an ASCII text file that may be created and edited with an ordinary text editor. It tells the Configuration Manager the format of the database table (the structures of the records) and precisely how the operator is allowed to edit it; that is, how to interpret operator input and fill in the fields of the records appropriately.

Place the AC file(s) in the /{FLINK}/AC directory. This directory contains AC files for all FactoryLink tasks; you can copy any AC file and edit it to create a new one.

Create the KEY Files.

If necessary, create any KEY files referenced by the Attribute Catalog(s) or the Panel file(s). Create the KEY file, which is a text file, with any text editor. This file tells the Configuration Manager how to validate operator-entered key words to be placed in the database table. After the KEY file(s) have been created, place them in the /{FLINK}/KEY directory which includes many KEY files useful for a FactoryLink system.

Test the Configuration Environment.

Inform FactoryLink about the new task by adding the name of the AC file to the TITLES file and adding the task to the System Configuration Table.

Using the Configuration Manager, test the configuration environment, including the Attribute Catalog.

> **Note:** To avoid problems, before beginning this test, save the application using the FLSAVE utility program. Upon completion of testing, restore the application using the FLREST utility. For details about these utilities, refer to the *FactoryLink Fundamentals* manual.

Once any problems are analyzed and corrected, repeatedly choose the task from the Configuration Manager Main Menu, each time entering a variety of simulated data in the panel(s) to be stored in the newly-created database table(s).

Converting the Database Tables to CTs

Converting the Database Tables to CTs involves the following two steps:

1. Create the CTG Conversion Scripts
2. Test the Conversion Process

Create the CTG Conversion Scripts.

Create a CTG conversion script for each of the database tables. The CTG conversion script tells the CTGEN utility (which generates binary CT files) how to extract data from the database tables and combine it to produce a binary Configuration Table (CT) file.

Place the CTG file(s) in the /{FLINK}/CTGEN directory which includes sample CTG files.

Test the Conversion Process.

Using a binary file dump program or the CTLIST utility included with FactoryLink, examine the CT file and verify that the conversion process properly creates the binary CT file.

Writing the Task's Program

Once you complete the design and conversion processes, you are ready to write the program itself. Writing the Task's Program consists of the following steps:

1. Compile the Source Modules
2. Link the Object Modules
3. Debug the Program
4. Create the Installation Medium

Create the Source Modules

Using a general-purpose text editor, create the source modules for the program, paying particular attention to the coding guidelines listed in this chapter. The PAK software includes sample source code modules. In addition, refer to Chapter 6, “Using the Run-Time Manager” for an example of properly documented and designed source code.

Define the operating system you will use with the task.

> **Note:** The platform may be defined in the compile-time environment instead of the source code.

Compile the Source Modules

Using one of the supported compilers, compile all source modules to form object files.

Link the Object Modules

Link all object modules to form an executable program file. Before attempting to link, read and understand the linker documentation. The linker recognizes several useful options. Correct any link errors before trying to execute the program.

Debug the program.

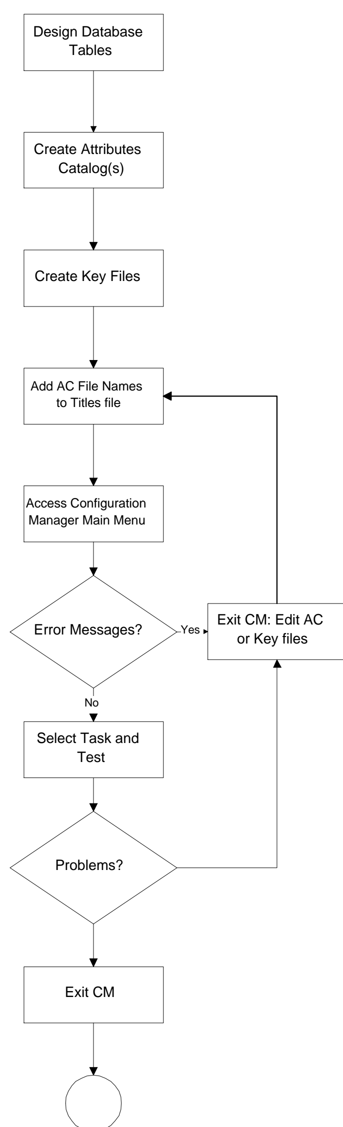
Debug the program or as many features of it as possible in the FactoryLink run-time environment.

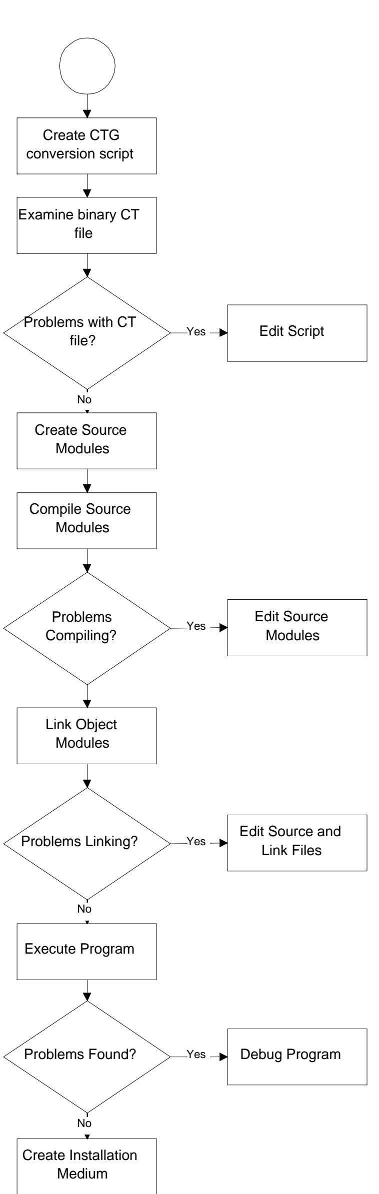
Create the Installation Medium

Create the installation medium (such as diskette or tape) and the appropriate command files for the end user to use when installing the custom module.

Overview

The following flow chart graphically illustrates the top-down design methodology we recommend for task construction.


Design Database Tables
Create Attributes Catalog(s)
Create Key Files
Add AC File Names to Titles file
Access Configuration Manager Main Menu
Error Messages?
Yes
Exit CM: Edit AC or Key files
No
Select Task and Test
Problems?
Exit CM


Create CTG conversion script
Examine binary CT file
Problems with CT file?
Yes
Edit Script
No
Create Source Modules
Compile Source Modules
Problems Compiling?
Yes
Edit Source Modules
Link Object Modules
Problems Linking?
Yes
Edit Source and Link Files
No
Execute Program
Problems Found?
Yes
Debug Program
No
Create Installation Medium

Operating System Notes

The following section contains operating system specific information related to this chapter.

For Windows/NT Users

Create the Source Modules (page 81)

Within the code, define the platform on which the task is to execute. The following statement is a sample definition in the C language:

```
#define WIN
```

Note: This can be defined in the compile environment instead of in the source code.

Compile the Source Modules (page 81)

Example: The name of the module is PROG1.C, located in the current directory. Using Borland C, from the command prompt, enter the following commands:

```
BCC -ml -DWIN -WE -C PROG1.C
```

Note: Use large-model to compile FactoryLink tasks.

Link the Object Modules (page 81)

Example: Create the program PROG.EXE from object modules PROG1.OBJ and PROG2.OBJ. Using Borland C, from the command prompt, enter the following commands:

```
TLINK /Twe /c /C [options] PROG1.OBJ,prog,,\FL-
WIN\LIB\FLIB.LIB+\FLWIN\LIB\FLIBW.LIB
```

For OS/2 Users

Create the Source Modules (page 81)

Within the code, define the platform on which the task is to execute. The following statement is a sample definition in the C language:

```
#define OS2
```

Note: This definition may be done in the compile environment instead of the source code.

Compile the Source Modules (page 81)

Example: The name of the module is PROG1.C, located in the current directory. Using Microsoft C, from the CMD prompt, enter the following commands:

```
SET INCLUDE = C:\MSC\INC;C:\FLOS2\PAK\INC;
CL -AL[options] PROG1.C
```

> **Note:** Compile FactoryLink tasks using large-model.

Link the Object Modules (page 81)

Example: Create the program PROG.EXE from object modules PROG1.OBJ and PROG2.OBJ. Using Microsoft C, enter the following commands from the CMD prompt:

```
SET LIB=C:\MSC\LIB;C:\FLOS2\PAK\LIB
LINK [options] PROG1 PROG2,PROG;
```

For UNIX Users

Create the Source Modules (page 81)

Within the code, define the platform on which the task is to execute. The following statement is a sample definition in the C language:

```
#define UNIX
```

> **Note:** This definition may be done in the compile environment instead of the source code.

Compile the Source Modules (page 81)

Example: The name of the module is PROG1.C, located in the current directory. Enter the following commands:

```
cc -c -I${FLINK}/inc
```

Link the Object Modules (page 81)

Example: Create the program PROG from object modules PROG1.O and PROG2.O. Enter the following commands:

```
cc -o prog prog1.o prog2.o ${FLINK}/lib/flib.a
${FLINK}/lib/flker.a
```

Chapter 4 . . . .

Setting up the Configuration Environment

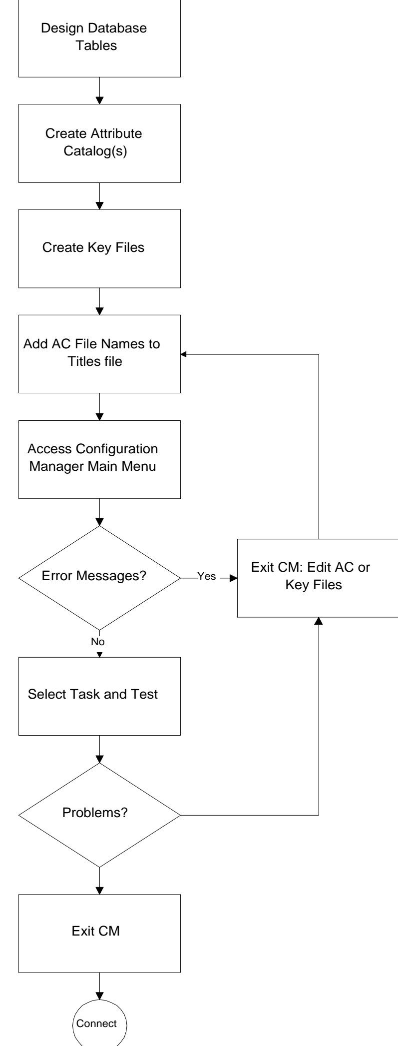
About this Chapter

This chapter provides a detailed look into setting up the configuration environment.

Chapter 4 covers the following tasks:

- Design the Database Table(s)
- Create the Attribute Catalog(s)
- Create the KEY Files
- Test the Configuration Environment

The figure on the following page illustrates the portion of the Task Construction Flowchart (refer to page 83) involved in setting up the configuration environment.


Design Database Tables
Create Attribute Catalog(s)
Create Key Files
Add AC File Names to Titles file
Access Configuration Manager Main Menu
Error Messages?
Yes
Exit CM: Edit AC or Key Files
No
Select Task and Test
Problems?
Exit CM
Connect

Design the Database Table(s)

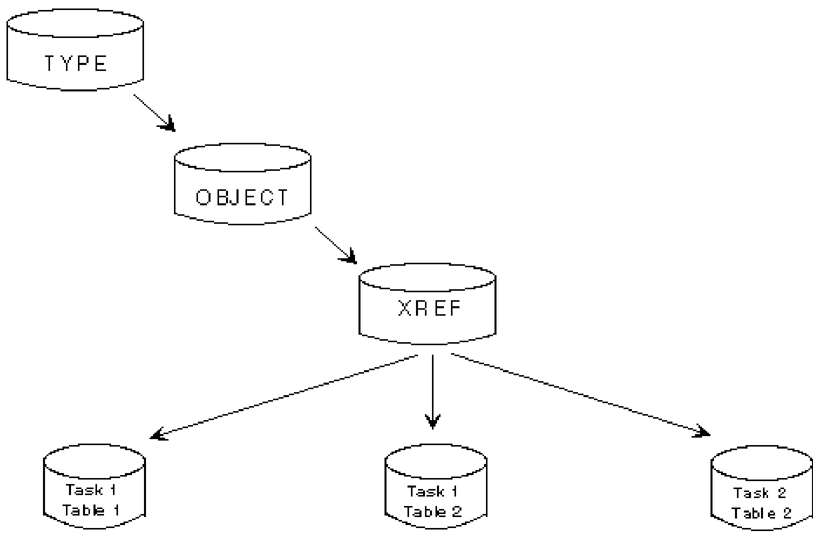
When you construct a task, you must design one or more database tables for the custom task.

The model for data entered and managed by the Configuration Manager is a set of relational database tables:

- TYPE
- OBJECT
- XREF
- Task-specific

The TYPE, OBJECT, and XREF database tables are inherent in FactoryLink and are not developer-definable. The task-specific database tables, however, are developer-defined and are only present when that task is used in the application.

The database tables form a set of relational databases as follow:



TYPE

The TYPE database table defines the six data types used in FactoryLink. Each of these types corresponds to a record in the TYPE table:

Type	Boolean
ANALOG	16-bit signed integer
LONGANA	32-bit signed integer
FLOAT	IEEE double floating-point
MESSAGE ASCII data	Variable length binary or
MAILBOX	Variable length data, organized as a queue

OBJECT

The Configuration Manager maintains the real-time database definition and all other configuration information for each application in the OBJECT database table. The particular database table format selected varies according to the platform. Knowledge of the format is generally not required to add a new FactoryLink task. The OBJECT table contains a list of real-time elements defined by the developer, but it does not contain task-specific information. Task-specific information is kept in other tables related solely to that task.

XREF

The Configuration Manager maintains a cross-reference table. The XREF database table contains a record for each occurrence of an element in any task-specific table. This information allows the Configuration Manager to quickly locate all occurrences of a particular element in the task-specific tables.

Task-Specific

A task consists of one of more developer-definable task-specific database tables. There is no limit on the number of tables per task. If there are more than two tables, the tables may be related either serially or in a one-to-many fashion.

Task-Specific Example

Suppose a PLC task consists of a Write table and a Read table. The Write and Read tables each consist of a Control panel and an Information panel. The Control panel contains header records specifying the trigger information for a group of data records. The control and information records are stored in different tables because they have different structures. Although it is possible to store the write and read information in the same table, storing them in different tables minimizes data entry and allows different validation criteria to be applied.

Designing the Task-Specific Database Table(s)

Designing the task-specific database table(s) is the first step in task construction. The design process should include the following considerations:

- What type of data is required for this task?
- What data must be stored in this task-specific database table, and what data can be gathered from other tasks, such as operator input from a screen designed with the Application Editor?
- What other tasks require information supplied by this task?
- Are both Control and Information panels required or just an Information panel?
- What is the panel design, including the following items:
  - Name of the panel
  - Panel layout, such as column headings and order of information
  - Data type of any element to be entered on the panel
  - Editing validation required for a field, such as data type limitation or value-range check

Relationship Between OBJECT and Task-Specific Tables

Task-specific tables contain information for the Configuration Manager to add to the other run-time tables on behalf of the task. For example, the Alarm Supervisor configuration allows the specification of an element to be monitored for an alarm condition. In addition to the element, the developer can also specify alarm messages and limits. The Configuration Manager adds the element to the OBJECT table if it has not already been created. In addition, the Configuration Manager adds the element name and task-specific information to the Alarm Supervisor table.

Create the Attribute Catalog(s)

An Attribute Catalog (AC) file represents one menu option on the Configuration Manager Main Menu; the AC file or option may or may not represent an entire task. For example, to configure the Batch Recipe task, the developer chooses only one option on the Configuration Manager Main Menu, Recipe, and fills in the associated panels. This option corresponds to the RECIPE.AC file. However, to configure the Interpreted Math and Logic task, the developer selects multiple options on the Configuration Manager Main Menu and fills in the associated panels. These options and their associated AC files are listed below:

Configuration Manager Main Menu Option	AC File Name
Math and Logic Variables	IMLV.AC
Math and Logic Triggers	IMLT.AC
Math and Logic Procedures	IMLP.AC

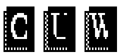
When the developer selects an option from the Configuration Main Menu, one or more panels are displayed. Each panel corresponds to a database table whose characteristics are specified in the AC file that corresponds to the option chosen.

An AC file indicates the name of the menu option, the order in which database tables are to be edited, and the relationships between the database tables. In addition, validation information can also be specified. Each field in a database table record editable by the developer must be specified in the AC file.

Because they are ASCII text files, the AC files for standard tasks can be easily copied and altered to create an AC for a new task.

AC File Format

In an AC file, each statement begins with a keyword followed by one or more parameters separated by commas. Comments may be included in the AC file by beginning the line with an asterisk (*). The comment continues to the end of the line. Blank lines and leading spaces are ignored. A statement may be split over multiple lines.



The general format of an AC file follows. Brackets ([]) indicate optional entries. Refer to “Sample AC File” on page 102 for an AC file example.

```
TASK name, title
```

```
CT table_name, file_name, title
[EDIT type [, editor]]
[VALIDATE type]
PANEL panel_file, x, y, width, height
FIELD name, type, width, field_prec, key_file, default,
low, high, flags
    [HEADING string, width ]
    [RELATE CDB_name, field_name, index_file ]
TYPE field_name [HEADING string, width]
DESC field_name [HEADING string, width]
DOMAIN "DOMAIN", 8
SELECT field_name , width [HEADING string, width]
SEQ field_name , width
INDEX index_file, key_expression, key_length
END
```

The following sections describe each statement in the AC file.

> **Note:** The keyword DELETE may appear in an .AC file. It is reserved for an internal function and is not available to the programmer.

TASK

The TASK statement defines the task name and title for the task list. Only one TASK statement appears in an AC file.

Parameter	Description	Valid Entries
name	Name of task as it will appear in the Task Name field on the System Configuration Information panel	Alphanumeric string of up to 8 characters
title	Name of the task as it will appear on the Configuration Manager Main Menu	Alphanumeric string

CT

The CT statement defines a database table. A task may have as many CT statements as needed, one per panel. If more than one panel is displayed when this task is chosen from the Configuration Manager Main Menu, place the CT statements in reverse order from the order in which the panels are displayed; that is, the first CT statement corresponds to the back panel, the last CT statement corresponds to the front panel, and so on.

Parameter	Description	Valid Entries	
table_name	Name of the database table	Valid table name.	
file_name	Name of the database table or file specification	If name refers to: Database table File spec.	Then enter: Same name as specified above in table_name TEXT or EXECUTE type in EDIT statement: File name(s) to edit.
title	Title of the panel	Alphanumeric string	

Refer to “Operating System Notes” on page 112 for valid name and table formats specific to each operating system

EDIT

The EDIT statement defines a module or program used to edit this database table. If the EDIT statement is not included, the Configuration Manager uses the default edit procedure (refer to the **DEFAULT** valid entry).

Parameter	Description	Valid Entries
type	Determiner of the module or program used for editing	**DEFAULT** (Standard panel edit procedure, which means that the developer completes the columns and rows of the panel, using the Tab key or arrow keys to move from field to field or row to row.) **TEXT** (Text panels, where developer presses the Enter key to move to the next line of text. Example: *Math and Logic Procedures.)* **EXECUTE** (For information about this entry, refer to "Executing an Editor Program from the Configuration Manager" on page 106 in this chapter.) **EXTERNAL** C L W
editor	Editor to be used on this database table; depends on the type parameter	**If the type is:** DEFAULT TEXT EXECUTE EXTERNAL

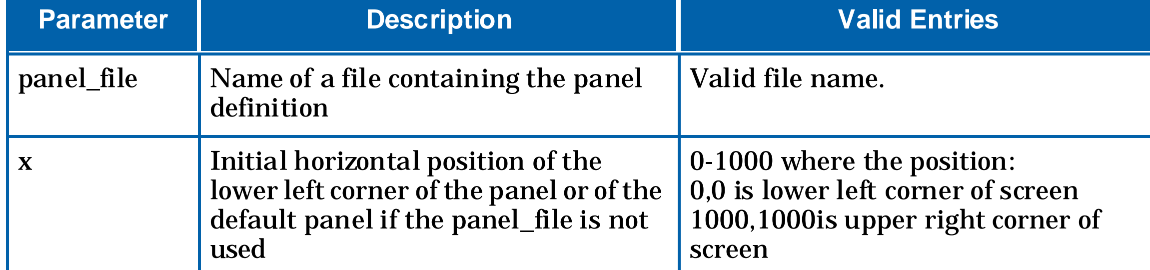
VALIDATE

The VALIDATE statement defines the level of editing a developer is allowed when completing a configuration table and the program or module that performs the editing.

Parameter	Description	Valid Entries	
type	Type and level of editing that can be performed on a configuration table	DEFAULT	Validation done internally by the Configuration Manager
		EXTERNAL	Validation performed by an external function
		READONLY	Developer can only view table; editing not allowed.
		NOEDIT	Do not use this entry: it is reserved. NOEDIT means developer cannot edit or view the table.

PANEL

The PANEL statement defines the display/edit window(s) for the database table. If the PANEL statement is not included, the developer cannot edit records in the table with the Configuration Manager default edit functions.

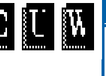


Parameter	Description	Valid Entries
panel_file	Name of a file containing the panel definition	Valid file name.
x	Initial horizontal position of the lower left corner of the panel or of the default panel if the panel_file is not used	0-1000 where the position: 0,0 is lower left corner of screen 1000,1000is upper right corner of screen

Parameter	Description	Valid Entries
y	Initial vertical position of the lower left corner of the panel or of the default panel if the panel_file is not used	0-1000 where the position: 0,0 is lower left corner of screen 1000,1000is upper right corner of screen
width	Width of the panel	0-1000
height	Height of the panel	0-1000

FIELD

The FIELD statement defines one editable field in the database.



Parameter	Description	Valid Entries
name	Field name exactly as it is stored in the database table definition	Valid field name.
type	Type of field. The Configuration Manager performs validation on the field based on the type.	CHARACTER NUMBER KEY TAG The developer can place any string in this parameter. If the string is not recognized, the Configuration Manager uses CHARACTER.
width	Maximum field length	0-11 characters or database limit for SQL
field_prec	Numeric precision of field	0
key_file	KEY or TAG types only: Name of a keyword file that is used to validate the value in the field.	Name of the key file; do not use the extension; no entry.
default	Default value for the field.	String. If the type is KEY, then the entry must be included in the key_file. If the type is TAG, then the entry must be a valid tag type.

Parameter	Description	Valid Entries
low	NUMBER type only: Lowest value allowed in this field.	Number; must match radix indicated in flags parameter.
high	NUMBER type only: Highest value allowed in this field.	Number; must match radix indicated in flags parameter.
flags	Edit options	b allow blank field v validate using min/max r read-only field o octal field x hexadecimal field u force field to upper case

HEADING

The HEADING option defines the text used as a field heading and the character width for the column. If a heading is not specified, the field name is used as the field heading and the width is calculated based on the number of characters in the field. The Configuration Manager allows an unlimited number of lines of text for the HEADING parameter. Delimit lines with the vertical bar on the first line and Heading on the second line.

Parameter	Description	Valid Entries
string	Field heading as it appears on the panel	Text string
width	Width of the heading in pixels	Number of pixels

RELATE

A RELATE entry indicates that this field is used as a relational field. The current value of the field is used to select records in another database table. The index file is currently unused.

Parameter	Description	Valid Entries
CDB_name	Name of the database table from which related records are to be selected (for example, associated data in another table about a specified table).	Table_name parameter entry of another CT statement in this AC file
field_name	Name of field to set using data from this field.	Name parameter entry of a FIELD statement in related CT in this AC file
index_file	Name of index to use for ordering; not currently implemented.	Index_file parameter entry of an INDEX statement in related CT in this AC file

TYPE

The TYPE statement causes the TAG type of a TAG field to be displayed.

Parameter	Description	Valid Entries
field_name	TAG field associated with the type	String containing the name of a field of type TAG in this CT
HEADING string	Field heading as it appears on the panel	Text string
HEADING width	Width of the heading in pixels	Number of pixels

DESC

The DESC statement indicates that the description of an OBJECT field should be displayed.

Parameter	Description	Valid Entries
field_name	TAG field associated with this description	String containing the name of a field of type TAG in this CT
HEADING string	Heading for this column on display	Text string
HEADING width	Width of the heading in pixels	Number of pixels

DOMAIN

Enter the DOMAIN statement exactly as shown below for each configuration panel in each .AC file:

```
DOMAIN “DOMAIN”, 8
```

After the developer configures a panel, the task-specific .CDB file's DOMAIN statement will contain SHARED or USER, depending on the domain the developer selected in the Domain Selection box. The OBJECT.CDB will associate each element with the domain defined by the developer in the Tag Definition pop-up panel.

For FactoryLink to operate correctly, this statement must be included for each configuration table defined in an .AC file.

SELECT

The SELECT statement defines a selection field. Each CT may have one or more select fields. The select fields determine whether or not a record in the CT is displayed and edited by matching the current value of the selection with the value contained in the database. Normally, select fields are not listed in the FIELD statements and cannot be changed by the developer. The first select field, however, is displayed at the bottom of the table and may be changed by the developer. The HEADING option allows a string to be defined that is used to label the select input box. Any other select fields should be set by a RELATE statement in another CT.

Parameter	Description	Valid Entries
field_name	Name of the field	String; valid database field name. C L W
width	Field length	Maximum database field width. C L W
HEADING string	Heading; displayed as prompt for select input field	Text string
HEADING width	Width of the heading in pixels	Number of pixels

SEQ

The SEQ statement defines a field that contains a sequence number for the records. Sequence numbers can be included in the primary key for the database so that records are sorted in the same order that they were entered. The Configuration Manager automatically generates the sequence number.

Parameter	Description	Valid Entries
field_name	Name of the field containing a sequence number for the records	Valid database field name. C L W
width	Field length	0 - maximum field width.

INDEX

The INDEX statement defines an index for the database. The first INDEX statement defines the primary index and controls the order of records on the screen. If needed, define additional indices for use by external programs, such as CTGEN. The Configuration Manager updates all indices whenever a database record is updated.

Parameter	Description	Valid Entries
index_file	Name of the index	Valid index name.
key_expression	List of fields that are used to form the key for the database. A field contained in the key should not be listed in the FIELD Statements	Entry format is: "FIELD+FIELD2+...FIELDN"
key_length	Length of the key	Integer equal to the sum of the lengths of the fields contained in the key

END

The END statement terminates a CT definition; therefore, there is one END statement corresponding to each CT statement in the AC file.

Sample AC File

Once the AC file is completed, it should resemble the sample AC file below (included in the PAK software as the file SKEL.AC):

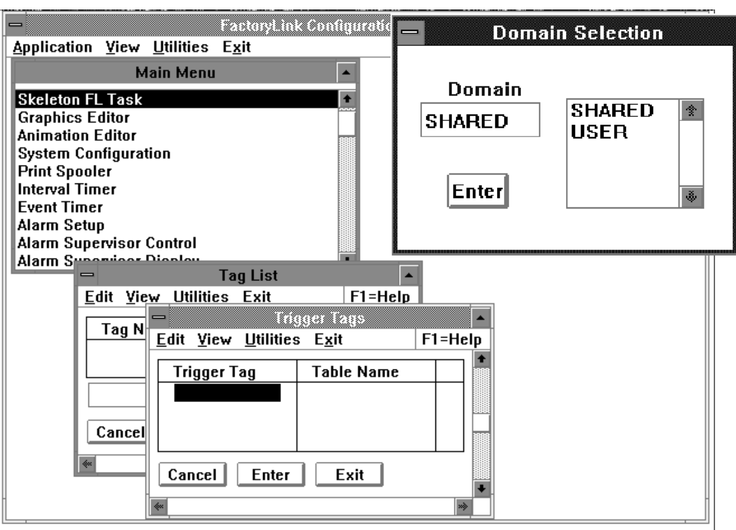
```
* Attribute Catalog for Skeleton Task
TASK "SKEL", "Skeleton FL Task"* Task name and Title

*   CT Name     Database   Title
CT "skeltags", "skeltags", "Tag List"
  EDIT DEFAULT * Default Editing
  VALIDATE DEFAULT * Default validation
  PANEL "", 100, 100, 500, 500 * Initial position
  FIELD "TAG",    "TAG", 16, 0, "", "", 0, 0, ""
     HEADING "Tag Name", 96
   DOMAIN"DOMAIN",           8
  SELECT "TABLE_NAME", 16 * Primary key
```

```
  SEQ “TABLE_NBR”, 10 * Sequence Number
     INDEX “skeltags”, “DOMAIN+TABLE_NAME+TABLE_NBR”, 34
*Indexf,key
END

*   CT Name     Database   Title
CT “skeltrig”, “skeltrig”, “Trigger Tags”
  EDIT DEFAULT                * Default Editing
  VALIDATE DEFAULT            * Default validation
  PANEL “”, 200, 0, 500, 500  * Initial position
  FIELD “TRIGGER”, “TAG”, 16, 0, “TYPED”, “DIGITAL”, 0, 0, “”
     HEADING “Trigger Tag”, 96
  FIELD “TABLE_NAME”, “CHARACTER”, 16, 0, “”, “”, 0, 0, “”
     HEADING “Table Name”, 96
     RELATE “skeltags”, “TABLE_NAME”, “skeltags”
   DOMAIN“DOMAIN”,8
  SEQ “TABLE_NBR”, 10                     * Sequence Number
   INDEX “skelhdr”, “DOMAIN+TABLE_NBR”, 18* Index file, key
END
```

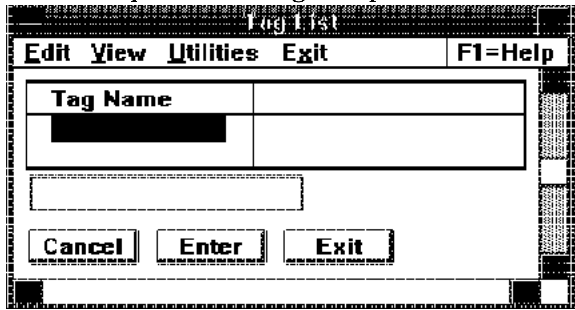
To configure the task, select SKeleton FL Task from the Configuration Manager Main Menu. As described in the CT statements, the panels, Tag List and Trigger Tags, are displayed on the Main Menu along with the Domain Selection box:



The second CT definition describes the Trigger Tags panel; therefore, it is the front panel in the display. Compare the PANEL statements for both panels: the lower left corner of the Tag List panel (position 100) appears further left than the lower left corner of the Trigger Tags panel (position 200). Also, the lower left corner of the Tag List is vertically higher (position 100) than the lower left corner of the Trigger Tags panel (position 0). The example runs in the User domain.

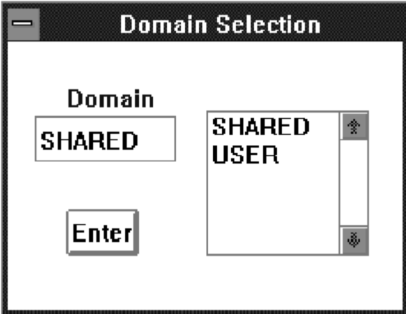
As described in the FIELD statements for the Trigger Tags panel, the panel contains two fields, Trigger Tag and Table Name. In the Trigger Tag field, since the type parameter in the FIELD statement is TAG and the key_file parameter is TYPED (digital), a developer configuring the task enters the name of a digital real-time database element. In the Table Name field, since the type parameter in the FIELD statement is CHARACTER and the width parameter is 16, a developer configuring the task enters a string of 1-16 characters.

As described in the RELATE statement for the Trigger Tags panel, the two panels are related to each other by the TABLE_NAME field. Once the Trigger Tags panel is complete, the developer selects the Table Name for which the developer wanted to complete a tag list, and then completes the Tag List panel for that table name.



After the developer completes the Trigger Tags panel, chooses the desired Table Name, and opens the Tag List panel, the chosen Table Name appears in the field above Cancel on the Tag List panel (refer to “SELECT” on page 101).

As described in the FIELD statement for this panel, the developer completes only one field, Element Name, entering the name of an element (TAG type in FIELD statement) that can be of any data type.



Next, the developer chooses the domain in which the external editor program is to run. Since an editor program normally runs an individual job for each system user, the developer in this example would choose USER to indicate the USER domain. To end the configuration, choose ENTER.

Executing an Editor Program from the Configuration Manager

The Configuration Manager can load any executable program as an editor; therefore, to run an editor from the Configuration Manager, create an AC file for the external editor program.

> **Note:** You must also add the name of the AC file to the TITLES file. Refer to "Testing the Configuration Environment" on page 110 for more information.

An AC file for an external program has the following format:

```
TABLE task name, title string
CT ct name, file spec, “”
EDIT EXECUTE executable, cm arguments, format string
VALIDATE DEFAULT
END
```

where

Parameter	Description
task name	Name of the FactoryLink run-time task
title string	Text that is to appear in the Configuration Manager Main Menu window
ct name	Name of the configuration table
file spec	File name to pass to the executable
executable	Path of the executable file
cm arguments	List of Configuration Manager arguments to be passed
format string	Format string for command arguments. If "%s" appears in the format string, it will be replaced with the file spec option and passed to the editor as a command line argument.

The following sample AC file loads the system E.EXE to edit FactoryLink Math and Logic files.

```
TASK “IML”, “Edit Math Procedures with System Editor”
```

```
CT “iml”, “procs/*.prg”, “”
EDIT EXECUTE “e.exe”, “”, “%s”
VALIDATE DEFAULT
END
```

Since the file specification contains a wild card character, the screen displays a list box of all .PRG files in the /**{FLAPP}**/PROCS directory. The developer selects a file name to be passed to the editor. If the file spec contains wild card characters and is not a full path name, the current application directory is added to the front of the file name to make a full path name.

> **Note:** The string specified as the format string is passed to the C-language function SPRINTF. Only one %s can be specified. Also, if a percent sign must be passed as an argument, denote this by using two percent signs.

If the file spec option does not contain any wild card characters, then the display does not contain the file list box, and the file spec option is used exactly as it appears in the AC file.

> **Note:** This is slightly inconsistent. The format string should be used in either case, but it is used only if the file spec contains wild card characters. There is currently no easy way to pass wild card characters on to the external editor.

If the developer selects MYPROC.PRG from the file list box, the following command is executed:

```
e.exe /{FLAPP}/PROCS/MYPROG.PRG
```

If the format string is given as “argument %s”, the following command is executed:

```
e.exe argument /{FLAPP}/PROCS/MYPROG.PRG
```

Create the KEY Files

KEY files are used to validate ASCII text strings entered by the developer in the Configuration Manager. They are also used by the CTGEN utility that generates the binary CT files to translate ASCII values into binary values used by the application task. A variety of KEY files can be found in the /{FLINK}/KEY directory on a FactoryLink system. If a KEY file does not exist that contains the desired values, create a new KEY file with an ordinary text editor.

Construction of a Key File

Each active line of a KEY file specifies a text-to-binary value conversion. Comment lines, which are not used by the system and are for information and documentation purposes only, begin with an asterisk (*).

By convention, the text and its associated value are displayed on the same line and are separated from each other by a vertical bar (|). Any leading and trailing blanks surrounding the text or the value are ignored by the Configuration Manager. The binary values in the KEY file must be entered in decimal notation.

Sample KEY File

A sample KEY file is MONTH.KEY, used for translating abbreviations of the names of the months to numeric values:

```
* MONTH.KEY

* NAME    | NUMBER

NULL      |-1
JAN       | 1
FEB       | 2
MAR       | 3
APR       | 4
MAY       | 5
JUN       | 6
JUL       | 7
AUG       | 8
SEP       | 9
OCT       | 10
NOV       | 11
DEC       | 12
```

Test the Configuration Environment

Prior to testing the configuration environment, inform FactoryLink that the new task exists.

Informing FactoryLink about the Task

To fully integrate the new task with the FactoryLink, the developer must configure FactoryLink to load and start up the task.

- Add the name of the AC file to the TITLES file to make the task accessible from the Configuration Manager Main Menu.
- Add the task to the System Configuration Table.

Note: The System Configuration Table defines reserved entries. A task can run without an entry; however, it cannot be automatically started by the Run Manager task, and the task cannot report its status to the Run-Time Manager using status and message elements.

- Add the task to the Run-Time Manager display to view status and message information on the Run-Time Manager display.

Adding the Name of the AC File to the TITLES File

When the Configuration Manager starts, it reads the file /{FLINK}/AC/TITLES which contains a list of all AC files to be loaded by the Configuration Manager. Therefore, the name of the AC file must be added to the TITLES file. Perform the following steps to allow access to the task configuration table(s) from the Configuration Manager Main Menu:

1. Put the AC file in the /{FLINK}/AC directory.
2. Using a text editor, add the name of the new AC file to the TITLES file, located in the /{FLINK}/AC directory. The entry location of the new AC file in the TITLES file corresponds to the location of that task selection on the Configuration Manager Main Menu.
3. Access the Configuration Manager Main Menu.

If the Configuration Manager finds anything wrong with the AC file or if it cannot locate a KEY file referenced by the AC file, it displays an error panel stating the nature of the problem.

If there are no problems or once any problems have been resolved, the name specified in the title parameter of the TASK statement in the AC file should appear as a selection on the Configuration Manager Main Menu.

Adding the Task to the System Configuration Table

To add a task to the System Configuration Table, open the System Configuration Information panel and complete the fields for the new task. For details about opening and entering information in the System Configuration Information panel, refer to "Using System Configuration" in the *FactoryLink Fundamentals* guide.

Adding the Task to the Run-Time Manager Display

So status and message information will be visible on the Run-Time Manager Display, add the task to the Run-Time Manager Display drawing and animate the appropriate fields:

1. Choose Application Editor from the Configuration Manager Main Menu.
2. Select the RUNMGR drawing.
3. On a line not in current use by a defined task, animate the Task field, the Status field, and the Message field as Output Text fields.
   - The name of the element entered for the Task field should match exactly the name entered in the Display Name field on the System Configuration Information panel.
   - The name of the element entered for the Status field should match exactly the name entered in the Display Status field on the System Configuration Information panel.
   - The name of the element entered for the Message field should match exactly the name entered in the Task Message field on the System Configuration Information panel.

Refer to the *Application Editor Guide* for detailed instructions about using the Application Editor.

Testing the Configuration Environment

Test the Attribute Catalog using the Configuration Manager.

Choose the new task from the Configuration Manager Main Menu. If the Configuration Manager finds anything wrong with the AC file or if it cannot locate a KEY file referenced by the AC file, it displays an error panel stating the nature of the problem.

Once these problems are analyzed and corrected, invoke the Configuration Manager repeatedly, each time entering a variety of simulated data in the panels, and, therefore, in the developer-created database tables. Using a compatible database manager or the CDBLIST utility included with the PAK, examine the resulting database table(s) to ensure that the Configuration Manager is generating the correct and expected data and placing it in the proper location within the table.

OPERATING SYSTEM NOTES

The following sections provide operating system specific information relevant to this chapter.

For Windows/NT Users

AC File Format (page 92)

This information corresponds to all references to valid name formats.

The PAK for Microsoft Windows currently uses a dBASE-compatible database library which dictates the following parameters:

Entries:	Maximum length/width:
All table and index names	Length of a file name
Field names	11 characters
Field width	255 characters

For OS/2 Users

AC File Format (page 92)

This information corresponds to all references to valid name formats.

The PAK for OS/2 uses a dBASE-compatible database library which dictates the following parameters:

Entries	Maximum length/width
All table and index names	Length of a file name
Field names	11 characters

Entries	Maximum length/width
Field width	255 characters

AC File Format (page 92)

For OS/2, no additional information applies.

Testing the Configuration Environment (page 110)

Use a dBASE-compatible database manager.

For UNIX Users

AC File Format (page 92)

This information corresponds to all references to valid name formats.

The Programmer's Access Kit under UNIX currently uses a dBASE-compatible database library which dictates the following parameters:

Entries	Maximum length/width
All table and index names	Length of a file name
Field names	11 characters
Field width	255 characters

EDIT (page 95)

The EXTERNAL entry for the type parameter of the EDIT statement is not valid in the UNIX environment.

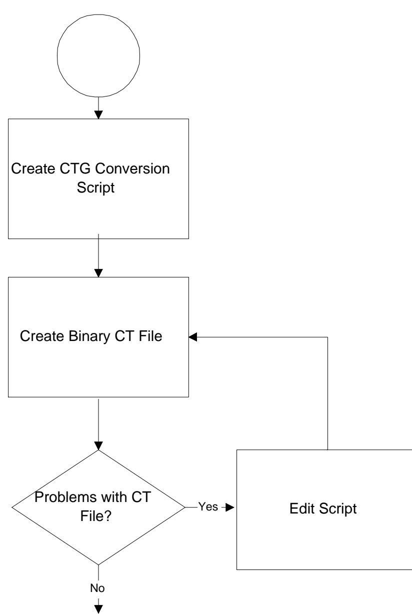
Testing the Configuration Environment (page 110)

Use a dBASE-compatible database manager.

Chapter 5 . . . .

Converting Database Tables to CTs

This chapter provides detailed instructions for the task construction procedure: converting database tables to CTs. The figure below illustrates the portion of the Task Construction Flowchart (see page 83) involved in converting database tables to CTs.



5

Converting Database Tables to CTs

When converting database tables to CTs, you must write a configuration table generator (CTG) script that tells the CTGEN utility (generate binary CT files) how to extract data from the database tables and combine it to produce a binary Configuration Table (CT) file. You must also verity that the script performs the desired conversion.

This section contains the following topics:

- Creating the CTG Conversion Scripts
- Testing the Conversion Process

Creating the CTG Conversion Scripts

Conversion Overview

The conversion process translates database tables into run-time, binary configuration tables. This enables FactoryLink run-time tasks to load the tables with little or no additional processing,

All run-time CTs use a common archive format. Each CT file begins with an archive header indicating the number of CTs in the archive. The header is followed by an index entry for each CT. Each CT consists of an optional header section followed by zero or more records. Each record within a CT has the same format.

> **Note:** The term archive refers to the binary CT file containing data for more than one database table. For example, the TIMER.CT file contains information for the event timer and interval timer database tables.

[CT archive header]

[CT index 0]

[CT index 1]

[CT index n]

[CT 0 header]

[CT 0 records]

[CT 1 header]

[CT 1 records]

[CT n header]

[CT n records]

The format of the CT archive header is shown below:

```
typedef struct _CTARC/* entire CT archive header*/
   {u16 magic;     /* magic number for CT file       */
   u16 version;    /* version number (0x0100 = V 1.0)*/
   u16 ncts;       /* number of CTs within archive   */
   /* (equals number of CTNDXs)                      */
] CTARC;
```

The format of a CT index is shown below:

```
typedef struct _CTNDX/* single CT index record       */
{
   u16 type;         /* type of entry (numeric)       */
   char name[10];    /* name of entry (ASCII)         */
   u32 offset;       /* file offset to header         */
   u16 hdrlen;       /* header length, in bytes       */
   u16 reclen;       /* record length, in bytes       */
   u16 nrecs;        /* number of records             */
CTNDX;
```

These structures can be found in the FLIB.H file. Refer to Chapter 7, “FactoryLink Kernel and Library” for additional information about this file.

The format of the CT header and CT records varies according to the database table.

Conversion Script Format

A conversion script, which is processed by the CTGEN utility, controls conversion of database table information into CT files. Generally, design the conversion script so the output from CTGEN matches the run-time task's data structures. This

allows the run-time task to read the binary data from the CT archive directly into memory. The /{FLINK}/CTGEN directory contains sample CT conversion scripts.

The format of a conversion script is shown below:

```
TABLE type, name, hdrlen, reclen
    HEADER database, namefield

SORT "FIELD1","FIELD2"
    DOMAIN "DOMAIN", S, 8
        FIELD name, format, storage, [options]
    RECORD database, namefield, indexfile

        SORT "FIELD1","FIELD2","FIELD3"
        DOMAIN "DOMAIN", S, 8
        FIELD name, format, storage, [options]
        SKIP count, value
```

The following paragraphs describe this format.

TABLE

The TABLE statement defines one CT type in the archive. The values of the parameters are placed in the appropriate fields in the CT index record. A single TABLE definition may result in multiple CT index entries. The HEADER database determines the number of times the table is repeated.

Multiple TABLE definitions may exist in the script. Each one is processed in order.

Parameter	Description	Valid Entries
type	Table type ID	0-65536 (Integer value)
name	Table name	String of up to 10 characters
hdrlen	Number of bytes to be written into the header entry in the archive	0 CTGEN automatically calculates this value.

HEADER

The HEADER statement defines the format of the header section of the CT. No more than one header record is written for a CT; however, the header record may be non-existent.

Parameter	Description	Valid Entries
database	Name of the database table to be used	Valid database table name (string of characters enclosed in quotes).
namefield	Controller of the repeated record section.	Null or non-null string If the value is:Then: null stringAll records in the repeated section are written. non-null stringThe value of the indicated field is used to fill in the name member of the CT index record and to select repeated records.

RECORD

The RECORD statement defines the format of the repeated record entries in a CT. Multiple RECORD statements are allowed. Each is processed in order.

Parameter	Description	Valid Entries
database	Name of the database table to be used	Valid database table name.

Parameter	Description	Valid Entries
namefield	Field in the RECORD database that must match the namefield field in the current record of the HEADER. If this parameter is not specified, the HEADER namefield is not specified, or the HEADER is non-existent, then all records in the RECORD database are written to the CT. This field should be the primary key.	Valid field name.
indexfile	Name of the index file that should be used to read the records	Index containing namefield as the primary key.

DOMAIN

Enter the DOMAIN statement exactly as shown below for each configuration panel in each .CTG file:

```
DOMAIN “DOMAIN”, S, 8
```

where DOMAIN is the name of the domain directory specified in the corresponding .AC file. This name must be the same as the name in the DOMAIN statement of the corresponding .AC file. For information about .AC files, refer to Chapter 4, “Setting up the Configuration Environment.

S Indicates that the domain directory name consists of a character string.

8 Is the number of characters in the domain directory name. This number must be the same as the number in the DOMAIN statement of the corresponding .AC file. For information about AC files, refer to Chapter 4, “Setting up the Configuration Environment.”

The DOMAIN statement in the .CTG file places the .CT file in the domain-specific path specified in this parameter, which is **{FLAPP}**/{FLDOMAIN}/*.CT.

For FactoryLink to operate correctly, this statement must be included for each configuration table included in a .CTG file.

FIELD

The FIELD statement defines the translation of one field in the database table to bytes in the CT entry. Include as many FIELD statements as necessary.

Parameter	Description	Valid Entries
name	Field name of the database field to be used	Valid field name.
format	Type of translation to be performed on the database field. The format is specified by a single character.	T Use only for TAG fields. Data is a two-word structure containing segment offset. S SCII string. The database field is copied directly to the output record. D Decimal number. The database field is considered to be a string of numerical digits representing a binary value. X Hexadecimal number O Octal number F Floating-point number
storage	Number of bytes in the output record that the value will occupy. The database value is truncated or null-padded to fit the output field width. All ASCII strings should be null-terminated. The storage width includes the terminating byte.	Number of bytes

Parameter	Description	Valid Entries
options	Specifier of additional information about how the data in the input field is converted. Use as many FIELD statements as required.	DEFAULT (Use alone or with any other option.) Default value to be used if the database table field contains a NULL. TAG (Do not use with TYPE, DIM or KEY.) Input field contains a tag name
		If format is:Then: S Tag name is output. D Tag segment and offset output as a numerical value T Tag segment and offset are written in binary with the offset in the low two bytes and the segment in the upper two bytes. DIM Do not use with TAG, KEY or TYPE.) The dimensions of the TAG contained in the database field is used as the output data.
		S The number of dimensions as a character string D The number of dimensions as a binary value TYPE (Do not use with TAG, DIM or KEY.) The name parameter is the name of a TAG field. The type of the TAG contained in the database field is used as the output data.
		If the output format is — Then the output is: S — The type name
		D — The type numeric ID

Parameter	Description	Valid Entries
		KEY — Do not use with TYPE, DIM or TAG.) The input field contains a string that must be converted using a key file. The name of the key file is listed after the KEY option.
		If the format is:Then: S — The replacement text is copied to the output record. D, X, or O — The replacement text is a string of digits to be converted to a binary numeric value.

SKIP

A SKIP statement causes one or more bytes to be written to the output. The SKIP statement allows padding of fields to match the task data structures as well as insertion of specific binary data into the CT file. Include as many SKIP statements as needed. FIELD and SKIP statements may be mixed in any order.

Parameter	Description	Valid Entries
count	Number of bytes to output	Integer value greater than 0
value	(Optional) Byte value used to fill the output record	Value of the byte (default = null)

SORT

The SORT statement defines the sort order of the repeated header or record entries in a CT. The SORT immediately follows the HEADER or RECORD to which it applies and may contain one or more field names to be used to sort the entries. Only one SORT statement is allowed per HEADER or RECORD entry.

Parameter	Description	Valid Entries
Sort field	Field names	"TABLE_NBR", "DOMAIN", or any field name in the header or record being sorted.

Sample Conversion Script

The following file is a sample conversion script included in the PAK software as SKEL.CTG.

```
TABLE 0, "", 4, 4
 HEADER "SKELTRIG.CDB", "TABLE_NAME"
 FIELD "TRIGGER", T, 4, TAG
 DOMAIN "DOMAIN", S, 8


RECORD "SKELTAGS.CDB", "TABLE_NAME", "SKELTAGS,CDX"
 FIELD "TAG",      T, 4, TAG
 DOMAIN "DOMAIN", S, 8
```

Creating FactoryLink Configuration Tables (CTs)

The binary CT file contains the information specified in the task database table(s). The CTGEN utility binds (converts) the tag names specified in the database tables to tag numbers maintained by the real-time database. At run time, the task loads the CT file and builds any internal structures required to perform the job. To improve performance, tasks use the tag number from the CT instead of the tag names in the database table(s) to access the real-time database.

After creating a *.CTG file and placing it into the {FLINK}/CTGEN directory, add it to the list of .CT files to be generated. Using any text editor, edit the CTLIST file on the {FLINK}/CTGEN directory. Add the CT name and all database tables that make up the CT.

```
ctname: databases . . .
```

After adding the script to the CTLIST file, generate a *.CT file using either of the following methods:

- Execute the Run-Time Manager by entering the following command at the system prompt:

```
FLRUN <Enter>
```

- Execute the CTGEN utility by entering the command shown below at the system prompt.

CTGEN uses the CTLIST file to build CTs and rebuild all CTs whose database tables have changed. CTGEN can be run stand-alone, or with a combination of parameters. To run CTGEN in stand-alone mode, enter the following command at the system prompt:

```
ctgen <Enter>
```

To run CTGEN with parameters, enter the following command at the system prompt:

```
ctgen [-i(name.ctg) -a(application dir) -o(ouput path)
-c -r -v(#) <Enter>
```

where

-i indicates that only those CTs referenced within the specified .CTG file are to be rebuilt. Next to -i, enter the file name of the .CTG file to be used. ({FLINK}/CTGEN is the default directory.)

-a are the CTs in alternate application directories.

-o indicates that the output is to be redirected to the specified file. (The default is **{FLAPP}**/CT/NAME.CT.)

-c indicates that all element numbers (segments and locations) are to be cleared before the CT is generated. (Must be followed by **-r**)

-r indicates that all files are to be rebuilt. (Usually preceded by **-c**)

-v(#) indicates verbose mode and level. Each verbose level displays cumulative messages. For example, v2 displays general activity messages and historian and client task messages. For general viewing, choose v1. For debugging, use v2-v4.

Valid entries:	Descriptions:
v1	displays general activity messages

5
Converting Database Tables to CTs

Valid entries:	Descriptions:
v2	displays historian and client task messages
v3	displays the parsing of .CTG file tokens
v4	displays tag names as they are written to the database.

If the environment variables (**{FLAPP}**/{FLNAME}/{FLDOMAIN}/{FLUSER}) have already been set and the developer has logged into the appropriate FactoryLink user account, **-n** and **-u** do not need to be specified.

Adding CT Information to the CM System Table

Refer to "Operating System Notes" on page 127 for operating system-specific notes on adding CT information to the CM System Table.

Testing the Conversion Process.

The following utilities provided with the PAK aid in testing the conversion process:

- CDBLIST - Lists a FactoryLink database table (located on the application directory).

```
cdblist *.cdb [*.cdx]
```

- CTLIST - Lists FactoryLink binary configuration tables (CTs) created by the CT Generator (CTGEN) located on the /**{FLAPP}**/CT directory. More than one CT file may be listed at a time.

```
ctlist name1.ct name2.ct ...
```

Operating System Notes

The following sections contain operating system specific information relevant to this chapter.

For OS/2 Users

Conversion Script Format (page 117)

This information corresponds to all references to formats for valid entries

The PAK for OS/2 currently uses a dBASE-compatible database library which uses the following parameters:

Entries	Maximum length/width
All table and index names	Length of a file name
Field names	11 characters
Field width	255 characters

Creating FactoryLink Configuration Tables (CTs) (page 124)

PAK for OS/2 does not have any additional CTGEN parameters.

Adding CT Information to the CM System Table (page 126)

This step is not necessary for OS/2

For Windows/NT Users

Conversion Script Format (page 117)

This information corresponds to all references to valid name formats.

The PAK for Microsoft Windows uses a dBASE-compatible database library which uses the following parameters:

Entries:	Maximum length/width:
All table and index names	Length of a file name
Field names	11 characters
Field width	255 characters

Creating FactoryLink Configuration Tables (CTs) (page 124)

PAK for Microsoft Windows does not have any additional CTGEN parameters.

Adding CT Information to the CM System Table (page 126)

This step is not necessary for PAK for Windows.

For UNIX Users

Conversion Script Format (page 117)

This information corresponds to all references to valid name formats.

The Programmer's Access Kit under UNIX currently uses a dBASE-compatible database library which dictates the following parameters:

Entries	Maximum length/width
All table and index names	Length of a file name
Field names	11 characters
Field width	255 characters

Adding CT Information to the CM System Table (page 126)

Do not perform this step in a UNIX environment.

Chapter 6 . . . .

Using the Run-Time Manager

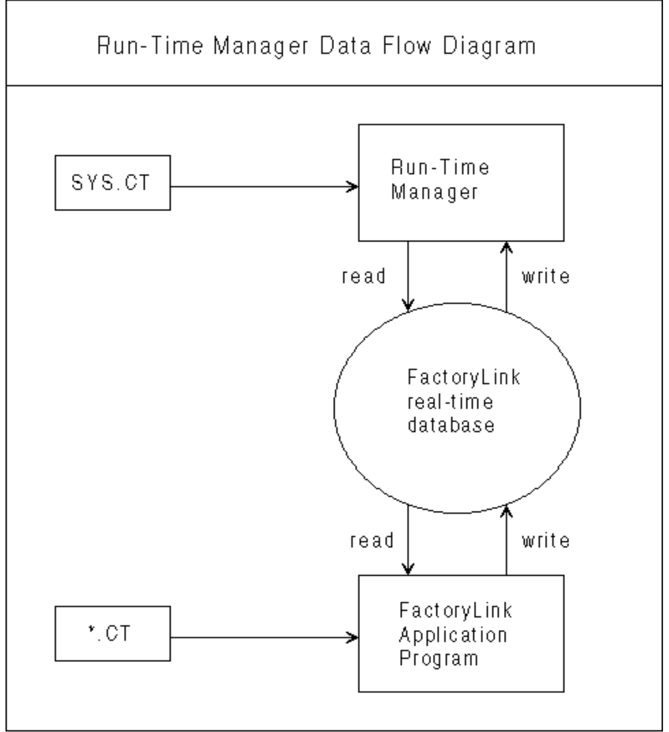
The Run-Time Manager is a program supplied with the FactoryLink Foundation package. It starts, monitors, and controls the concurrent execution of all FactoryLink tasks. The Run-Time Manager is itself a FactoryLink task that runs concurrently with the other FactoryLink tasks.

This section contains information about the following topics:

- Interaction With Other Tasks
- Design Conventions
- Run-Time Requirements
- Sample Task Program Skeleton

INTERACTION WITH OTHER TASKS

The Run-Time Manager interacts with the other tasks within a specific domain through the FactoryLink API and the real-time database as shown in the following figure.



Design Conventions

Design a custom FactoryLink task using the guidelines listed in Chapter 3, "Constructing a Task", Adherence to the following design conventions provides maximum portability and protects against software obsolescence.

- The program should access the real-time database and Run-Time Task Table only through the FactoryLink library functions provided expressly for this purpose. Refer to Chapter 7, "FactoryLink Kernel and Library," in this manual for an overview of these functions. Refer to Chapter 8, "FactoryLink API Reference Guide," in this manual for details about using these functions.
- If the program accesses window-management functions for display and input, it should use only standard operating system procedures, such as calling provided OS utilities.
- User domain instances of the Run-Time Manager should be started up after the Run-Time Manager instance in the shared domain. If a user instance of the Run-Time Manager attempts to start when there is no running instance of the Run-Time Manager in the shared domain, the user domain instance will be refused initialization by the FactoryLink kernel, and an error message to that effect is displayed. Should this error message occur, check to ensure that user domain instances are not started until after the shared domain Run-Time Manager is running.

Run-Time Requirements

For the Run-Time Manager to effectively perform FactoryLink task management duties, each FactoryLink task must adhere to certain run-time requirements as outlined in Chapter 3, “Constructing a Task“, of this manual. Most of these requirements are predicated on the fact that the Run-Time Manager monitors specific database elements in the real-time database on a per-task basis. The requirements are described below:

> **Note:** Refer to Chapter 8, “FactoryLink API Reference Guide” in this manual for information about FactoryLink functions included in the following discussion.

Initialization

First, the task attaches with the real-time database (or Kernel) by calling FLIB function FL_PROC_INIT() to obtain a task id (*taskid*). Function FL_PROC_INIT() uses environment variables {FLNAME}, {FLDOMAIN}, and {FLUSER} to determine which real-time database to attach.

> **Note:** Should the task need to attach elsewhere, FLIB function FL_PROC_INIT_APP() can be called.

If the call fails (*taskid* == **ERROR**), the task writes an appropriate error message to STDOUT and calls EXIT.

If the call succeeds (taskid!= ERROR), the task obtains the task's environment elements by calling FL_GET_ENV. The environment elements, which are used to communicate with the Run-Time Manager, include the following information:

- Status element for reporting task status as an ANALOG value
- Message element for reporting task status as a MESSAGE
- Application path specification
- Program path specification
- Command line arguments

Then the task writes FLS_ACTIVE to its status element and running to its message element. This causes Active to be displayed in the STATUS column and running to be displayed in the MESSAGE column of the Run-Time Manager display.

Kernel check (Conditional)

If proper execution of the task depends on the version of the Kernel installed on the system, obtain the Kernel's version and release number by calling FL_GET_VERSION. Check the values obtained against your own list of compatible values. As a rule an unacceptable version or release number should be considered a fatal error. If this occurs, write an appropriate error message to STDOUT and call EXIT.

Error Handling

If the task encounters any errors, failures, or other problems during execution, it reports appropriate error messages to its environment STATUS and MESSAGE elements.

- In case of a fatal error, the task exits following the shutdown procedure described in the Orderly Shutdown requirement.
- In case of a non-fatal error, the task sets its environment STATUS element to FLS_ERROR. This indicates to the Run-Time Manager the task encountered problems but is continuing execution, and the Run-Time Manager displays ERROR in the STATUS column of its display. The task writes a description of the problem to its environment MESSAGE element.

Termination Notification

A task must shut down when notified by the Run-Time Manager to do so. To check the current status of the task termination flag, a running task periodically calls FL_TEST_TERM_FLAG.

- If the value of the flag is 1 (ON), the task goes through the shutdown procedure in and writes normal shutdown to its environment MESSAGE element.
- If the value of the flag is 0 (OFF), the task continues execution.

Tasks must check the flag often. The operator should be able to terminate the task from the Run-Time Manager, and the task must terminate along with its associated FactoryLink session.

Orderly Shutdown

Just before calling EXIT for any reason, the task performs the following actions:

- Writes a message explaining the reason for termination to its environment MESSAGE element which is then displayed in the MESSAGE column of the Run-Time Manager display
- Sets its environment STATUS element to FLS_INACTIVE, which causes Inactive to appear in the STATUS column of the Run-Time Manager display

The task then terminates execution via FL_PROC_EXIT.

Domain Selection

In conformance with the requirements for the Run-Time manager, the task must respect the domain starting sequence and set its own environment variables. Set {FLDOMAIN} to shared or user as required; set {FLNAME} to the application name and fluser to the current user instance name.

Refer to Chapter 2, “FactoryLink Architecture,” for an overview, the “Domains: User and Shared (Per-User Shared Memory Regions)” on page 39 for additional information about deciding which domain the task should run in, and Chapter 7, “FactoryLink Kernel and Library” for details of domain instantiating.

Sample Task Program Skeleton

A sample FactoryLink program written in the C language follows. This sample program demonstrates proper programming practices, standards, and conventions. The example illustrates the interaction of the program with the Run-Time Manager.

> **Note:** Refer to Chapter 8, "FactoryLink API Reference Guide" of this manual for API function references.

```
/*

*************************************************************************
 * Copyright 1984-1992 United States Data Corporation. All Rights Reserved.

*************************************************************************
 *                              - NOTICE -
 *
 *  The information contained herein is confidential information of United
 *   States Data Corporation, a Delaware corporation, and is protected by
 * United States copyright and trade secret law and international treaties.
 * This information is not to be disclosed, used or copied by or transferred
 *  to any individual, corporation, company or other person without the
 *  express written permission of United States Data Corporation.

*************************************************************************
 *
 * FactoryLink Skeleton Task
 *
 * This file contains a skeleton for a generic FactoryLink task.
 *
 * A FactoryLink real-time task performs the following steps:
 *
 *    1) register with the FactoryLink kernel
 *    2) load any configuration information
```

```
 *    3) scan input values and process the data
 *    4) when the task termination flag is set,
 *       perform an orderly shutdown.
 *
 * Normally, the task will be started by the Run Time Manager task
 * using information in the process configuration table. Each task
 * is given a status TAG, a message TAG, and a control TAG. The
 * status and message TAGs are used by the task to communicate
 * internal processing errors to other tasks and the operator.
 * The control TAG is used by other tasks to start and stop this
 * task.
 *
 * At startup, a FactoryLink task must first register with the
 * kernel to obtain a task id. The task id is used in most real-time
 * database access calls.
 *
 * After successfully registering with the kernel, the task loads
 * any configuration information. The standard configuration file
 * is in an archive format. This archive format allows multiple
 * tables to be included in one file.
 *
 * After initialization is complete, the task should issue a
 * fl_change_wait() call to block until one or more database elements
 * have changed. The fl_change_wait() function will return GOOD
 * if elements have changed, or, ERROR if the termination flag has
 * been set for the task. When the termination flag is set, the
 * task should call fl_exit() and then shutdown.
 */
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
```

```
#include<flib.h>        /* FactoryLink definitions */


#defineXBUFSIZE1024           /* fl_xlate buffer size */

/*
 * CT header structure. This structure acts as a template
 * for reading the header portion of a configuration table.
 * The header is optional, and, its structure is task-specific.
 */
typedef structcthdr
{
      TAG   trigger;
} CTHDR;

/*
 * CT record structure. This structure acts as a template
 * for reading the configuration file records. The structure
 * is task-specific.
 */
typedefstructctrec
{
      TAG   value;
} CTREC;

TAG   *Triggers;              /* trigger tags */
uint  Trigcount;              /* # of triggers */

/*
 * Task global variables
 */
```

```
id_t  Task_id = -1;           /* Id for this task */
char  *App_dir = "";          /* Application dir. */
char  *Pgm_dir = "";          /* Program dir. */
char  *Cmd_arg = "";          /* Command argument */
TAG   Task_stat = {0xFFFF};/* Task status TAG */
TAG   Task_msg  = {0xFFFF};/* Task message TAG */
char  Task_name[] = "SKEL";/* Process name */
char  Task_desc[] = "Skeleton FL Task";/*  and Description */

/*
 * Task function prototypes
 */
void  shutdown(int);
void  status(char *, ANA);
void  ctload(void);
void  process(void);


/*
 * Task main function
 */
int   main(int argc, char *argv[])
{
KENV  env;
staticchar  xlbuf[XBUFSIZE];

      /* Set up for message translation */

      fl_xlate_init(Task_name, xlbuf, XBUFSIZE);

      /* Acquire a task id */
```

```
    Task_id = fl_proc_init(Task_name, Task_desc);
    if (Task_id < 0)
          exit(1);

    fl_get_env(Task_id, &env);     /* get task environment */
    Task_stat = env.e_stat;        /* task status (ANALOG) */
    Task_msg  = env.e_msg;         /* task message (MSG) */
    App_dir   = env.e_adir;        /* application directory */
    Pgm_dir   = env.e_pdir;        /* program data directory */
    Cmd_arg   = env.e_cmd;         /* task arguments/options */

    status("START", FLS_STARTING);/* indicate startup state */
    ctload();                      /* load task configuration */
    status("RUN", FLS_ACTIVE);     /* indicate running state */

    /*
     * Continuously loop, processing all input until
     * a terminate message has been sent to this task.
     */
    while ( fl_test_term_flag(Task_id) == OFF )
          process();               /* perform processing */

    status("STOP", FLS_INACTIVE); /* indicate normal stop */
    shutdown(0);                   /* exit to OS */
    return 0;
}


/*
 * shutdown the task.
```

```
 */
void  shutdown(int error)
{
      fl_proc_exit(Task_id);         /* terminate FL access */
      exit(error);                   /* exit to OS */
}

/*
 * status writes a message and a status value to the task's status tags.
 * The fl_xlate function is used to convert from the key string to a
 * language dependent message.
 */
void  status(char *s, ANA n)
{
MSG   m;

      if ( Task_id < 0 )
            return;
      m.m_ptr = fl_xlate(s);
      m.m_len = strlen(m.m_ptr);
      m.m_max = MAX_MSG;
      fl_write(Task_id, &Task_msg, 1, &m);
      fl_write(Task_id, &Task_stat, 1, &n);
}


/*
 * Load task Configuration Table file.
 *
 * This file will typically be named:
 *    <app_dir>/ct/<task>.ct
```

```
 *
 * This code demonstrates how to use the standard FactoryLink
 * configuration table archive format.
 */
void  ctload(void)
{
CT    ct_buf;                 /* Configuration Archive buffer */
int   num_cts;                /* number of tables in the archive */
int   num_recs;               /* number of records in one table */
int   ct_count;               /* counter for processing tables */
int   rec_count;              /* counter for processing records */
i16   info;                   /* TAG information */
CTHDR hdr_buf;                /* buffer for CT header - task defined */
CTREC rec_buf;                /* buffer for CT record - task defined */
char  ctfile[MAX_FILE_NAME];

                              /* Build the path name to the CT file */
      fl_getvar("FLDOMAIN", ctfile, sizeof(ctfile));
      strcat(ctfile, "/ct/skel.ct");

      /* Open the CT archive file */

      if ( ct_open(&ct_buf, App_dir, ctfile) != GOOD )
      {
            status("NOCT", FLS_INACTIVE);
            shutdown(1);
      }

      /* Determine the number of tables */

      num_cts = ct_get_ncts(&ct_buf);
```

```
if ( num_cts < 1 )
{
      status("NOTRIGGERS", FLS_INACTIVE);
      shutdown(1);
}

/* There is one trigger for each configuration table. */

Triggers = malloc(num_cts * sizeof(TAG));
if ( Triggers == (TAG *)0 )
{
      status("NOMEMORY", FLS_INACTIVE);
      shutdown(1);
}

Trigcount = 0;

/* Loop through the tables, processing each in turn */

for ( ct_count = 0; ct_count < num_cts; ct_count++ )
{
      /* Read the index for this table */

      if ( ct_read_index(&ct_buf, ct_count) != GOOD )
      {
            status("CTINDEX", FLS_INACTIVE);
            shutdown(1);
      }

      /* Read the table header */
```

```
if ( ct_read_hdr(&ct_buf, &hdr_buf) != GOOD )
{
      status("CTHEADER", FLS_INACTIVE);
      shutdown(1);
}

/*
 * Get information about the tag and verify
 * that the tag is of the correct type
 */
fl_get_tag_info(&hdr_buf.trigger, 1, &info, (u16 *)0);
if ( info != FL_DIGITAL )
{
      status("BADTAG", FLS_INACTIVE);
      shutdown(1);
}

/*
 * Store the trigger tag in the triggers array.
 * Note: Some pre-ANSI compilers cannot perform
 * the in-line structure assignment. Use:
 * memcpy(&Triggers[Trigcount++], hdr_buf.trigger,
sizeof(TAG))
 * instead.
 */
Triggers[Trigcount++] = hdr_buf.trigger;

/*
 * Read the records for this table.
 * Note: If no per-record processing is needed,
 * the ct_read_nrecs() function can be used to
```

```
             * read all records in one call.
             */
            num_recs = ct_get_nrecs(&ct_buf);

            for ( rec_count = 0; rec_count < num_recs; rec_count++ )
            {
                  if ( ct_read_rec(&ct_buf, &rec_buf, rec_count) != GOOD )
                  {
                        status("CTRECORD", FLS_INACTIVE);
                        shutdown(1);
                  }
                  /* Add code here to process the record */
            }
      }
      ct_close(&ct_buf);
}


/*
 * process is where the main processing takes place.
 *
 * Normally, a task waits on one or more triggers, then, processes
 * the data associated with each trigger as it changes.
 *
 * When the task termination flag is set, fl_change_wait returns
 * an error indication.
 */
void  process(void)
{
DIG   state;
uint  index = 0;
```

```
int    e;

      while ( (e = fl_change_wait(Task_id, Triggers, Trigcount, &index,
&state)) == GOOD )
      {
            if ( state )
            {
                  /* Process values for the trigger here */
            }
            if ( ++index >= Trigcount )
                  index = 0;
      }
      if ( e != GOOD )
      {
            /* issue a status message to indicate the error */
            if ( fl_errno(Task_id) != FLE_TERM_FLAG_SET )
                  status("FLREAD", FLS_ERROR);
      }
}
```

Chapter 7 . . . .

FactoryLink Kernel and Library

This chapter provides an overview of the FactoryLink kernel and library. Refer to Chapter 8, “FactoryLink API Reference Guide” for details about the use of the kernel’s services.

This section contains information about the following topics:

- FactoryLink Kernel
- FactoryLink Library
- Kernel Multi-User Environment (MUE) Extensions
- Calling and Return Conventions
- System Shutdown
- Kernel and Library Services

FactoryLink Kernel

The FactoryLink kernel is a software module that provides basic services to FactoryLink tasks. These services include process management, access to the real-time database, access to the CT archives, access to the environment, and mailbox services.

The name of the FactoryLink kernel library is flkernel.

FactoryLink Library

The FactoryLink library is a collection of utility functions serving primarily to interface application and system programs to the FactoryLink kernel.

All FactoryLink library functions are contained in the file FLIB.LIB, located in the directory {FLINK}/LIB.

Programs linking with FactoryLink library functions must be large-model C programs.

A C-language source program must include the header file FLIB.H to use any of these FactoryLink functions.

Platform-specific header files also exist; however, since you define the specific operating system in the source code or the compile environment, the inclusion of FLIB.H automatically includes the appropriate platform-specific header file. Refer to the "Create the Source Modules" on page 81 in Chapter 3, "Constructing a Task" for details about defining the operating system in the source code or compile environment.

KERNEL MULTI-USER ENVIRONMENT (MUE) EXTENSIONS

This section describes changes made to the FactoryLink kernel in order to support multi-user extensions. Current users need not worry about these changes; those who are upgrading or modifying an existing task may find this information of interest.

In order to allow multiple independent users of a FactoryLink application, the following aspects of the FactoryLink kernel have been modified since the previous releases:

- Allow more than 31 tasks so each user may run a collection of tasks.
- Provide per-user shared memory areas for inter-task communication.
- Provide common shared memory areas for real-time data.
- Allow specification of the user so a task is attached to the correct per-user shared memory areas.
- Allow multiple applications to run simultaneously.

Increased Task Handling Capability

The number of tasks available under FactoryLink is limited by the number of change/wait bits per database element and by the number of entries in the FactoryLink task table. The multi-user extensions allow additional change/wait bits by allocating a separate memory area for change/wait bits on a per-user basis. In addition to a per-user set of change bits, each user is also allocated a task table and a set of semaphores (e.g., LOCK/UNLOCK, P[x]/V[x] flags) for inter-task signaling.

Domains: User and Shared (Per-User Shared Memory Regions)

User Domain

FactoryLink tasks use real-time database elements to control tasks. These database elements are duplicated for each FactoryLink user. The subset of the real-time database duplicated on a per-user basis is known as the USER domain.

Multi-user extensions allow duplicate named real-time database element areas by allocating an array of pointers to database segments for each user. This allows the element numbers (indices into the pointer array) themselves to remain the same for each user while allowing the element number to reference a private data area for each user.

Shared Domain

FactoryLink tasks also use real-time database elements to monitor and control the state of the process. These database elements must be the same for all users. The subset of the real-time database shared by all users is known as the SHARED domain.

In order for database elements to be shared in common among all authorized users, it is only necessary that each user's pointers to database segments for shared elements map to the same physical memory. When a new user area is created, the common element pointers are copied from a master copy associated with the first user who logged on to the application.

Application Instances/ Identification

Since multiple copies of a FactoryLink task may be running concurrently, a task must identify itself to the kernel. The task must specify the application, domain and user instance as well as the task name.

Each application instance is specified by its invocation name. The invocation name is a character string of up to MAX_USR_NAME (currently set to 16) characters long and is used to locate the shared memory segment containing the global data structure of the kernel. The invocation name identifies an instance of the FactoryLink real-time database. The invocation name is stored in the environment variable {FLNAME} by the run-time manager.

The domain within an application is specified by the domain name. The domain name is a character string of up to MAX_USR_NAME (16) characters long and is used to determine which real-time database segments the task owns and which should be shared. In addition, the domain name is used by the task to determine which CT files should be processed. The domain name is stored in the environment variable {FLDOMAIN} by the run-time manager.

The user instance is specified by the user name. The user name is a character string of up to MAX_USR_NAME (16) characters long and is used to determine which instance of the domain specific real-time database segments the task is to use. The user name is stored in the environment variable {FLUSER} by the run-time manager.

When a FactoryLink application is started, the run-time manager must be supplied with the invocation name, domain name, and user name. Each may be supplied on the command line or through environment variables. The three required environment variables are created by the run-time manager if they do not exist. The values passed to the run-time manager are stored in the environment of any sub-processes created by the Run-Time Manager.

Refer to Chapter 2, "FactoryLink Architecture" in this manual for further information and a graphical representation of the concept of shared and user domain data handling.

Calling and Return Conventions

This section discusses the calling and return conventions and includes a reference list of error numbers.

Conventions

The following calling and return conventions apply to application programs that call any of the FactoryLink Library functions:

- Most functions in the Library return an item of C data type "int" (16-bit signed integer). These integers are returned in the AX register. A few return an item of data type long (32-bit signed integer).
- A return value of -1 (int or long) invariably indicates an error indicating the function failed. The reason for such a failure depends on the function and the circumstances under which it is called. The kernel returns an error code filled in whenever an error occurs but is unchanged by successful kernel calls.

To access the error, the calling task must call FL_ERRNO with its task id. FL_ERRNO returns the error text. The calling task accesses this variable to determine the nature of the error and takes appropriate action.

In the source code, the error numbers may be referred to by the integer value or the symbolic representation of the number. USDATA recommends using the symbolic representation. For example, the symbolic representation of 0 is GOOD. A line of code might read as follows:

```
if (ct_open(&ct_buf, app_dir, "ct/skel.ct") != GOOD)
```

This code checks whether the return value from a request to open a CT archive file indicates an error occurred. If the symbolic representation of GOOD is used, this code does not require any changes if, for some reason, the integer value of GOOD later changed to a value other than 0. If the integer value is used in the code and it changes, the code must be changed and recompiled.

The symbolic representations of these error numbers are found in the header file FLIB.H.

Return Reference List

Use the following lists (excerpts from FLIB.H) as references for error numbers.

Table 7-13

FactoryLink Error	Numbers *(Returned by Kernel Services)*
FLE_INTERNAL	1
FLE_OUT_OF_MEMORY	2
FLE_OPERATING_SYSTEM	3
FLE_NO_{FLINK}_INIT	4
FLE_NO_PROC_INIT	5
FLE_BAD_FUNCTION	6
FLE_BAD_ARGUMENT	7
FLE_BAD_DATA	8
FLE_BAD_TAG	9
FLE_NULL_POINTER	10
FLE_NO_CHANGE	11
FLE_PROC_TABLE_FULL	12
FLE_BAD_PROC_NAME	13
FLE_BAD_USER_NAME	14
FLE_BAD_OPTION	15
FLE_BAD_CHECKSUM	16
FLE_NO_OPTIONS	17
FLE_NO_KEY	18
FLE_BAD_KEY	19

Table 7-13

FactoryLink Error	Numbers *(Returned by Kernel Services)*
FLE_NO_PORT	20
FLE_PORT_BUSY	21
FLE_ALREADY_ACTIVE	22
FLE_NOT_LOCKED	23
FLE_LOCK_FAILED	24
FLE_LOCK_EXPIRED	25
FLE_WAIT_FAILED	26
FLE_TERM_FLAG_SET	27
FLE_QSIZE_TOOBIG	28
FLE_QSIZE_CHANGED	29
FLE_NO_TAG_LIST	30
FLE_TAG_LIST_CHANGED	31
FLE_WAKEUP_FAILED	32
FLE_NO_SIGNALS	33
FLE_SIGNALLED	34
FLE_NOT_MAILBOX	35
FLE_NO_MESSAGES	36
FLE_ACCESS_DENIED	37

Table 7-14

CT Access Function Error	Numbers
CT_CANNOT_OPEN_FILE	1 file is missing

Table 7-14

CT Access Function Error	Numbers	
CT_CANNOT_CLOSE_FILE	2	program bug
CT_FILE_NOT_OPEN	3	program bug
CT_SEEK_ERROR	4	wrong file size
CT_READ_ERROR	5	wrong file size
CT_WRITE_ERROR	6	drive not ready or disk full
CT_BAD_MAGIC	7	file corrupted
CT_BAD_DATA	8	file corrupted
CT_NULL_POINTER	9	program bug
CT_BAD_INDEX	10	CT does not exist
CT_BAD_RECORD	11	CT record does not exist

FactoryLink Client Process Status

Table 7-15

FLS_INACTIVE	0	inactive/not running
FLS_ACTIVE	1	active/running
FLS_ERROR	2	non-fatal error encountered
FLS_STARTING	3	starting/initializing
FLS_STOPPING	4	stopping/exiting

System Shutdown

A smart run-time task can initiate a shutdown of the FactoryLink system. To cause the FactoryLink Run-Time Manager to begin an immediate system

shutdown, the task writes a value of 1 (ON) value to the analog element **RTMCMD**.

Kernel and Library Services

The kernel and library services can be divided into the following categories:

- Process management
- Database access
- Tag list registration and notification
- Mailbox
- Memory management
- Signaling
- Environment access
- CT access
- Path manipulation
- Format version number
- Message translation
- Expression Processing
- Sleep
- Miscellaneous

This manual assumes that you develop client processes in the C Programming Language or in a language that provides a calling sequence compatible with the C language. This section includes information about the service functions and how to use them.

Function names are case-sensitive. All API function names are entirely lower-case. For example, a client process calls FL_PROC_INIT in the following way:

```
id = fl_proc_init(“RUNMGR”, “Run-Time Manager version 1.0");
```

Process Management

The process management services provide the means for the following activities:

- Identifying client processes and prospective clients to the kernel
- Preparing for real-time database access

- Informing other client processes that they should exit (quit running)
- Determining whether to continue running or to prepare for exiting
- Exiting and renouncing all further access to the database

In general, these functions aid in the control of client processes by the kernel and by other client processes, notably the FactoryLink Run-Time Manager.

Since a prospective client must make itself known to the kernel before doing anything, it must successfully call FL_PROC_INIT (which assigns a FactoryLink ID) before calling any other kernel service. There are a few exceptions to this rule; namely, the functions that do not have the FactoryLink ID as one of the arguments. A client process may not call any other kernel service after calling FL_PROC_EXIT which releases to the system, or “renounces,” the caller's FactoryLink ID.

Note: The FL_PROC_INIT function has been removed from the kernel and placed in the flib library in the current release of FactoryLink. This should not affect the functionality at the application developer level.

Using the Process Management Functions

The process management functions are C functions callable by C-language application and system programs. The library functions in turn make calls to the FactoryLink kernel. It is the duty of the kernel to maintain a task table. The task table holds all information relevant to running tasks and furnishes services allowing access to and limited manipulation of the data stored in this table.

The process management services consist of the following functions. Refer to Chapter 8, “FactoryLink API Reference Guide” for details about each of these functions.

Table 7-16

Function	Description
FL_SET_TERM_FLAG	Set the termination flag of a client process.
FL_TEST_TERM_FLAG	Ask the kernel the current status of current task's termination flag.
FL_PROC_EXIT	Exit the calling process.
FL_NAME_TO_ID	Translate a process name to a FactoryLink ID.

Table 7-16

Function	Description
FL_ID_TO_NAME	Translate a FactoryLink ID to a process name

.

> **Note:** The FL_PROC_INIT function is no longer in the kernel; it resides in the FLIB library. Refer to “fl_proc_init” on page 288 in Chapter 8, “FactoryLink API Reference Guide.”

Sample Process Management Function

The following example illustrates the use of the process-management functions. It does not illustrate the interaction of a task with the Run-Time Manager.

```
#include “FLIB.H”

char name[] = “MYNAME”;    /*my name(name of this task)*/

int  taskid;          /*task number to be assigned*/

main()                /*PROGRAM ENTRY POINT       */
{
     /*————————-WITHIN START-UP CODE—————————*/
     if                ((taskid = fl_proc_init(name,desc)) ==
ERROR)
                       /* first library call:      */
{                      /* tell the kernel who I am */
     printf(“Cannot get task number\n”);
     exit(1);          /*abort for stated reason   */
     }
     while (fl_test_term_flag() == OFF)
                       /* continue execution until */
     {                 /* flag is turned ON        */
     /*——————-MAIN LOOP OF TASK-DEPENDENT CODE———*/
     /*————————————-GOES HERE——————————-*/
     }

/*————————-WITHIN CLEAN-UP CODE————————*/
fl_proc_exit(taskid);/* tell kernel I'm exiting  */
```

```
        */
exit(0)/* exit to OS */
}
```

Database Access

The real-time database access functions allow the following activities:

- Reading from the real-time database
- Writing to the real-time database
- Testing to see whether real-time database values have changed
- Waiting for real-time database values to change

A given real-time database element may be read by any task; however, it should be written by only one task. This task is called the defining task. This restriction is not enforced, but it is a good design technique to avoid possible ambiguity in the real-time database.

Using the Database Access Functions

The database access functions are functions callable by C-language application and system programs.

The Library functions make calls to the FactoryLink kernel. The kernel maintains the real-time database shared among all client processes. The kernel permits client processes to access it only through these functions.

The database access services consist of the following functions. Refer to Chapter 8, "FactoryLink API Reference Guide" for details about each of these functions.

Table 7-17

Function	Description
FL_READ	Read specified elements from the real-time database.
FL_WRITE	Write specified elements into the real-time database.
FL_FORCED_WRITE	Force-write a specified element into the real-time database.

Table 7-17

Function	Description
FL_CHANGE_WAIT	Read the first real-time database element that has changed since it was last read; if no change, go to sleep until a change occurs.
FL_CHANGE_READ	Read the first real-time database element that has changed since it was last read.
FL_SET_CHNG	Set the calling task's change-status flags for specified real-time database elements.
FL_CLEAR_CHNG	Clear the calling task's change-status flags for specified real-time database elements.
FL_SET_SYNC	Set the calling task's sync flags for specified real-time database elements.
FL_CLEAR_SYNC	Clear the calling task's sync flags for specified real-time database elements.
FL_SET_WAIT	Set the calling task's wait flags for specified real-time database elements.
FL_CLEAR_WAIT	Clear the calling task's wait flags for specified real-time database elements.

FactoryLink Real-Time Database

The real-time database is organized as arrays and pointers. There are six arrays of elements, one array for each data type, and a separate storage area for messages and mailbox data:

Table 7-18

Data Type	Description
Digital	Boolean (logical)
Analog	Short integer
Long analog	Long integer

Table 7-18

Data Type	Description
Floating-point	IEEE standard double-precision
Message	String
Mailbox	Variable length data, organized as a queue

The corresponding typedefs for each data type are found in FLIB.H.

Database Elements: A real-time database element consists of the following items:

- One or more bits containing the element's value
- Set of change-status bits
- Set of wait bits
- Set of sync bits

Each set of bits consists of a single bit for each client process or, more properly, for each potential client process. A process does not officially become a FactoryLink client process until it registers with the kernel by initializing the FactoryLink calling process through a call to FL_PROC_INIT, but the bits still exist. Refer to Chapter 2, “FactoryLink Architecture,” in this manual for additional details.

Change-status bits: For each database element, each possible client process has one change-status bit. If the value of a bit is 0 (OFF), the value has not changed since client (or task) number N last read the element, where N = 0 through 30. The value of the bit is set to 1 (ON) when a new value is written to the element.

Wait bits: Each possible client process has one wait bit. When the client is currently waiting to read or to write the specified element, the value of the bit is **1** (ON); otherwise, the value is 0 (OFF). A client can set the value of a wait bit to 1 (ON) by performing one of the following actions:

- Call FL_CHANGE_WAIT. This sets the value to 1 (ON) only if none of the specified elements has changed (it sleeps until someone writes a new value into one or more of the elements).
- Set the wait bits directly by calling FL_SET_WAIT.

Sync bits: There is one sync bit for each possible client process. A value of 1 (ON) indicates that the specified element is synchronous for that client. A value of 0 (OFF) indicates it is asynchronous. A client process uses FL_SET_SYNC to

create synchronous elements and FL_CLEAR_SYNC to undo this; that is, to make them revert to asynchronous (default) status.

Tag Number: In the kernel, a tag number completely specifies a database element because it includes the data type as well as the array index. All database access service operates on mixed types; that is, on elements of various data types defined and allocated at run time.

Locking/Unlocking the Database: The kernel automatically locks the database during execution of any of these database access functions and unlocks it upon completion. This ensures that service calls are atomic operations in the sense that, once begun, they cannot be interrupted by service calls from other clients. The only exception is when the calling process may block another process waiting to write synchronous elements.

VAL Union Structure

All database types besides message and mailbox are read and written the same way, accessing the correct member of the union.

Table 7-19

Database Type	Union
DIGITAL	val.dig
ANALOG	val.ana
LONGANA	val.lana
FLOAT	val.flp
MESSAGE	val.msg
MAILBOX	val.mbxmsg

where val is a VAL union structure located in FLIB.H.

Sample Database Access Functions

The following examples illustrate the use of the database access functions in the FactoryLink Library. These examples are not complete C programs. None of the examples illustrates the interaction of the task with the Run-Time Manager.

Example A: Example A demonstrates how to use FL_READ to read messages from the database.

```
.
.
int  task_id
TAG  t;
VAL  v;
char buffer[100];

     v.msg.m_ptr = buffer;
     v.msg.m_len = 0;
     v.msg.m_max = sizeof(buffer);
     fl_read(task_id, &t, 1, &v);
.
.
```

Example B: Example B demonstrates how to use FL_WRITE to write messages into the database.

```
.
.
int    task_id;
TAG    t;
VAL    v;
char   buffer[100];

       strcpy( buffer, “The temperature is”);
       v.msg.m_ptr = buffer;
       v.msg.m_len = strlen(buffer);
       v.msg.m_max = sizeof(buffer);
       fl_write(task_id, &t, 1, &v);
.
.
```

Example C: Example C demonstrates how to use FL_READ to read analog values from the database.

```
.
.
int    task_id;
TAG    t;
VAL    v;
```

```
        fl_read(task_id, &t, 1, &v);
        printf( "The ANALOG value is %d\n", v.ana);
.
.
```

Example D: Example D demonstrates how to use FL_WRITE to write analog values to the database.

```
.
.
int     task_id;
TAG     t;
VAL     v;

        v.ana = 100;
        fl_write(task_id, &t, 1, &v);
.
.
```

Tag List Registration and Notification

The tag list registration and notification services allow one FactoryLink process to establish a list of database elements that serve as trigger elements for another FactoryLink process referred to as the target process.

The tag list services use the memory management services described later in this chapter to allocate, access, and free tag lists.

The registration and notification services consist of the following functions. Refer to Chapter 8, "FactoryLink API Reference Guide" for details about each of these functions.

Table 7-20

Function	Description
FL_SET_TAG_LIST	Register the tag list (a list of real-time database elements) to a target process.

Table 7-20

Function	Description
FL_GET_TAG_LIST	Retrieve the tag list (a list of real-time database elements) for the calling process.
FL_CHANGE_WAIT_TAG_LIST	Wait for a change in value of one or more elements in the list of real-time database elements in the tag list for the calling process.
FL_CHANGE_READ_TAG_LIST	Read a change in value of one or more elements in the list of real-time database elements in the tag list for a calling process.

Mailbox

A mailbox is a real-time database element that holds a queue of mailbox messages consisting of structures (typedef MBXMSG) and some associated message data. Refer to FLDEFS.H to view the MBXMSG structure. Since mailboxes are real-time database elements, a FactoryLink process can read and write values to them just as with other types of database elements; that is, FL_READ, FL_WRITE, FL_FORCED_WRITE, FL_CHANGE_READ, and FL_CHANGE WAIT work when passed to a mailbox element.

When messages are present in a mailbox, the change bit for the associated real-time database element is set to 1. When the mailbox is empty, the change bit for the associated real-time database element is clear (value of 0).

Mailbox services provide an inter-process communication (IPC) mechanism to FactoryLink processes. The services allow one FactoryLink process to send a mailbox message (typedef MSG), a structure that can hold arbitrary data of variable length, to another FactoryLink process.

Signal services, discussed later in this chapter, also provide an IPC mechanism to FactoryLink processes. Compared with signals, mailboxes allow data passing and are more flexible; however, they are slower. Also, messages sent to a given

mailbox are placed in a queue so they are read in the same order as they are sent. Signals are prioritized but not queued.

> **Note:** FL_READ_MBX allows messages to be read in a different order than the messages were sent.

The message queue associated with a mailbox contains a head and a tail. The head of the queue is the oldest message in the mailbox; the tail is the newest. Mailbox message reads occur at the head or relative to the head of the queue; writes always occur at the tail.

The mailbox services consist of the following functions. Refer to Chapter 8, "FactoryLink API Reference Guide" for details about each of these functions.

Table 7-21

Function	Description
FL_COUNT_MBX	Determine the number of messages in a mailbox, validate a mailbox, or monitor a mailbox.
FL_QUERY_MBX	Query a mailbox for a range of queued messages.
FL_READ_MBX	Read and dequeue a message from a mailbox.
FL_WRITE_MBX	Write and queue a message into a mailbox.
FL_SET_OWNER_MBX	Set the owner of a mailbox.

Memory Management

Memory management services provide functions that allocate, access, and free large blocks of contiguous memory sharable among all FactoryLink processes. The memory management group contains its own memory merging and compaction functions (garbage-collection functions) and handles this aspect of its management duties in a manner totally transparent to FactoryLink processes.

Memory blocks managed by these services are assigned a virtual pointer structure (typedef VPTR) that can be used to reference them.

The memory management services consist of the following functions. Refer to Chapter 8, "FactoryLink API Reference Guide" for details about each of these functions.

Table 7-22

Function	Description
FL_ALLOC_MEM	Allocate a specified amount of memory.
FL_ACCESS_MEM	Access memory block.
FL_FREE_MEM	Free memory.

Signals

Signals are notifications of events, particularly change of status, sent by one FactoryLink process to another, referred to as the target process. Signals are a form of inter-process communication (IPC). In some cases, the kernel itself may send a signal to a target process, but this is still a form of IPC since the kernel always executes in the context of the calling process and acts on its behalf. The target process, rather than the caller, is notified of the event or what status has changed.

Mailbox services, discussed earlier in this chapter, also provide an IPC mechanism to FactoryLink processes. Compared with signals, mailboxes allow data passing and are more flexible; however, they are slower. Also, messages sent to a given mailbox are placed in a queue so they are read in the same order as they are sent. Signals are prioritized but not queued.

Signals provide a primitive, but very fast, form of IPC; therefore, they are ideally suited for process synchronization. Each signal is assigned a numerical value in the range 0-31, so only 32 different events can be described by signals. The following events are predefined by the kernel and have special meaning:

Table 7-23

Signal	Symbolic Name	Meaning
0	FLC_SIG_TERMINATED	Process has been terminated (as
1	FLC_SIG_TERM_FLAG_SET	Termination flag is set.
2	FLC_SIG_TAG_LIST_CHANGED	Tag list has been changed.
3	FLC_SIG_MESSAGE_RECEIVED	Message has been received.

The signal services consist of the following functions. Refer to Chapter 8, “FactoryLink API Reference Guide” for details about each of these functions.

Table 7-24

Function	Description
FL_SEND_SIG	Send a signal to a target process.
FL_RECV_SIG	Receive a signal for the calling process.
FL_HOLD_SIG	Prevent or allow signal delivery for the calling process.

Environment Access

The kernel provides read-only access to the environment (KENV) structure through the environment access services.

The environment access services consist of the following functions. Refer to Chapter 8, “FactoryLink API Reference GuideFor details about each of these functions.

Table 7-25

Function	Description
FL_GET_CTRL_TAG	Return the control tag, which refers to a digital or analog real-time database element, for the specified process.
FL_GET_STAT_TAG	Return the value of the real-time database analog status element for the specified process.
FL_GET_MSG_TAG	Return value of the real-time database message element for the specified process.
FL_GET_PGM_DIR	Return the program directory for the specified process.
FL_GET_APP_DIR	Return the application directory for the specified process.
FL_GET_CMD_LINE	Return the command line for the specified process.

CT Access

The CT access functions allow client processes to access the FactoryLink CT archives. A typical FactoryLink application has a function that opens the task's CTs and reads them into memory.

Note: The term “archive” refers to the binary CT file containing data for more than one database table. For example, the TIMER.CT file contains information for the event timer and interval timer database tables.

The write CT access functions are the complement of the CT read functions. Use them to create and/or modify the CT archives.

The CT access services consist of the following functions. For details about each of these functions, refer to Chapter 8, "FactoryLink API Reference Guide."

Table 7-26

Function	Description
CT_OPEN	Open a CT archive file.
CT_READ_INDEX	Read an index from a CT archive.
CT_READ_HDR	Read the header for ctp into buffer.
CT_READ_REC	Read a record from the current CT into memory.
CT_READ_RECS	Read records from the current CT into memory.
CT_CALC_OFFSET	Calculate a CT offset.
CT_CLOSE	Close a CT archive.
CT_CREATE	Create/truncate/open a CT for update.
CT_UPDATE	Open a CT for update.
CT_WRITE_INDEX	Write a CT index record.
CT_WRITE_HDR	Write the CT header from buffer to the current CT header.
CT_WRITE_REC	Write a CT record to buffer.
CT_WRITE_RECS	Write CT records into buffer.
CT_FIND_INDEX	Find an index.

Path Manipulation

In the current release of FactoryLink, several API functions dealing with path name manipulation, construction, and normalization have been added. The functions beginning with the name "FL_PATH_" are all concerned with allowing developers to write portable code with generic path name structures which can be automatically converted to be system-specific. The path manipulation function MAKE_FULL_PATH still exists, for downward compatibility, to combine the directory and file name into a full path name. For a general explanation of these new functions, refer to "Path Name Building and Representation" in Chapter 3 of this manual; for full specifications on the new functions, refer to the appropriate pages in Chapter 8, "FactoryLink API Reference Guide."

Format Version Number

The format version number function, FL_DBFMTT, prepares a formatted string from a set of real-time database element values. For details about this function, refer to Chapter 8, "FactoryLink API Reference Guide," in this manual.

Message Translation

The message translation functions provide services so that run-time tasks can translate and print messages stored in external disk files (message files). Several enhancements to these functions have taken place since the previous release.

The message translation services consist of the following functions. For details about each of these functions, refer to Chapter 8, "FactoryLink API Reference Guide."

Table 7-27

Functions:	Descriptions:
FL_XLATE_INIT	Message translation initialization; initializes a message file and establishes a buffer to it (use of the buffer enhances performance). The message translation data structure is a tree.
FL_XLATE	Translate a key to its associated message.

Table 7-27

Functions:	Descriptions:
FL_XLATE_LOAD	Load the specified file into the current translation tree, replacing any duplicate keys. The function returns the number of entries loaded from this file or returns **ERROR**.
FL_XLATE_GET_TREE	Returns the address of the current translation tree or **NULL** if no translation files have been loaded.
FL_XLATE_SET_TREE	Sets the current translation tree to the tree at the specified address.
SPOOL	Spool a file or line.
TSPRINTF	Create a target string according to a format string using the given argument values.

Overview of Message Translation Functions

Keyword/translation pairs in FactoryLink are kept in a tree data structure. The translation for a keyword is retrieved using the fl_xlate() function. This tree is automatically loaded with a set of default translations for FactoryLink tasks. The translations may be overridden or supplemented by loading another translation file into the current tree.

Each task may maintain several translation trees.

The loading of translation files may be from a particular library of language translation files.

In order for all tasks to be language-independent, all tasks should use the fl_xlate functions for all message output.

If file names or other data need to be imbedded into a message, use fl_xlate() to retrieve the format string and then sprintf using that format, as in the following example.

Example 1

```
The file test.txt contains:
```

```
BADFILE“Could not open the file: %s”
```

```
The file test.c contains:
```

```
char buff[MAX_MSG_SIZE] ;
sprintf(buff, fl_xlate(“BADFILE”), filename);
```

Some fl_xlate functions, as noted in the following paragraphs, have been added or modified since the previous release to allow more flexibility and efficiency when translating FactoryLink keywords. Multi-language support is included.

Loading Translation Files

```
int fl_xlate_init(char FAR *file, char FAR *buff, uint len)
int fl_xlate_load(char FAR *file)
```

When loading translation files, the environment variable FLLANG is examined. If it is defined to be anything besides “C”, it will be appended to the /{FLINK}/MSG path used to load the master file, as well as user-defined files.

If the user-specified file contains path information, it will be folded into the /{FLINK}/MSG{/FLLANG} path when loading the file.

The function **fl_xlate_init** will start a fresh translation tree and load the default translation file **master.txt**. The user-specified translation file is then loaded on top of the existing definitions. Duplicate definitions are superseded by the last file loaded. The function returns the total number of translations loaded from both the master.txt file as well as the user-specified file, or it returns **ERROR** if there has been an error.

The **buff** and **len** parameters of **fl_xlate_init** are not used by the function, but were retained to stay compatible with existing code.

The function **fl_xlate_load** will load the specified file into the current translation tree, replacing any duplicate keys. The function returns the number of entries loaded from this file or returns **ERROR**.

Example 2.

If the FLLANG environment variable is not defined, the following call:

```
fl_xlate_init(“iml”, NULL, 0)
```

will load the **/{FLINK}/MSG/master.txt** file into the tree and then load the **/{FLINK}/MSG/iml.txt** file into the tree. The return value is the total number of translations loaded from both files (duplicates are counted only once.)

The call

```
fl_xlate_load("iml")
```

will load the file **/{FLINK}/MSG/iml.txt** into the tree and return the number of translations.

The call

```
fl_xlate_load("/temp/test")
```

will load the file **/TEMP/test.txt** into the tree and return the number of translations.

Example 3.

If the FLLANG environment variable is defined to be "GERMAN", the following call:

```
fl_xlate_init("iml", NULL, 0)
```

will load the **/{FLINK}/MSG/GERMAN/master.txt** file into the tree and then load the **/{FLINK}/MSG/GERMAN/iml.txt** file into the tree. The return value is the total number of translations loaded from both files (duplicates are counted only once.)

The call

```
fl_xlate_load("iml")
```

will now load the file **/{FLINK}/MSG/GERMAN/iml.txt** into the tree and return the number of translations.

The call

```
fl_xlate_load("/temp/test")
```

will still load the file **/TEMP/test.txt** into the tree and return the number of translations.

Translating

```
char FAR *fl_xlate(char FAR *key)
```

The function **fl_xlate** will return a pointer to the translation text for the requested key if the key is found in the tree. The pointer to the original key is returned if no translation is found.

Managing Multiple Translation Trees

```
void FAR *fl_xlate_get_tree( void )

void FAR *fl_xlate_set_tree( void *p )
```

The function **fl_xlate_get_tree()** will return the address of the current translation tree or **NULL** if no translation files have been loaded.

The function **fl_xlate_set_tree()** will set the current translation tree to the specified address. Future file loads and translations will be done using this tree.

Example 4.

```
void *tree1, *tree2 ; /* pointers to 2 trees */

fl_xlate_load(“file1") ;
tree1 = fl_xlate_get_tree() ;

fl_xlate_load(“file2") ;
tree2 = fl_xlate_get_tree() ;

fl_xlate_set_tree(tree1) ;
printf(“%s\n”, fl_xlate(“token”)) ;

fl_xlate_set_tree(tree2) ;
printf(“%s\n”, fl_xlate(“token”)) ;
```

fl_xlate_set_tree may also be used to start a fresh tree, as in the following example.

Example 5.

```
fl_xlate_set_tree(NULL) ; /* start new tree */
```

Sleep

The sleep function, FL_SLEEP, delays execution of the task for a specified amount of time (in milliseconds). For details about this function, refer to Chapter 8, “FactoryLink API Reference Guide.”

Miscellaneous

The kernel provides service functions to do miscellaneous “odd jobs” that fall into none of the other categories. These include database lock/unlock functions (semaphores), sleep/wake functions, initialization functions, and system data item retrieval functions.

The miscellaneous services consist of the following functions. For details about each of these functions, refer to Chapter 8, “FactoryLink API Reference Guide.”

Table 7-28

Function	Description
FL_LOCK	Lock the real-time database on behalf of the calling process.
FL_UNLOCK	Unlock the real-time database for the calling process.
FL_WAIT	Wait to read, write, or access the real-time database
	or certain elements in the database.
FL_WAKEUP	Awaken a mask of FactoryLink client processes.
FL_WAKEUP_PROC	Awaken a specified FactoryLink process.
FL_ERRNO	Return the last FactoryLink error number generated by the calling process.
FL_GET_VERSION	Get the kernel version number.
FL_GET_TAG_INFO	Get the information associated with a specified list of
	real-time database elements.
FL_GET_NPROCS	Get the number of client processes permitted to run concurrently.
FL_GET_ENV	Return the KENV structure of the client process.
FL_INIT	Initialize the FactoryLink kernel and its global data area.
FL_GET_TITLE	Return a pointer to the name of the product (“FactoryLink”).
FL_GET_COPYRT	Return a pointer to a copyright message for FactoryLink.

Table 7-28

Function	Description
FL_GET_TICK	maintained and reported by the operating system.
FL_GLOBAL_TAG	Retrieve the tag number for one or more global elements.

Many of these functions may be called internally (that is, by other service functions) as well as externally. The following chart provides some examples.

Table 7-29

Function	May be called by
FL_LOCK and FL_UNLOCK	All of the real-time database access services
FL_WAKEUP	FL_READ, FL_WRITE, FL_FORCED_WRITE, FL_CHANGE_WAIT, FL_CHANGE_READ, and FL_CLEAR_CHNG
FL_WAIT	FL_WRITE, FL_FORCED_WRITE, and FL_CHANGE_WAIT

OBJECT CT OVERVIEW

A binary representation of a configuration database file, the object CT is not unlike any other, task-specific CT. The main difference is that the CT contains the contents of the application's object database, which is not tied to any one particular task. Another difference is that a large application with many objects requires special handling in order to fit within a standard CT.

Written using the PAK's *ct_...()* primitives, the object CT API hides the details of how the object database contents are coerced into a CT.

Overview of Object CT Services

The Object CT API mirrors the existing *ct_...()* API and provides the following services:

Opening and closing of the object CT.

Object definition retrieval based on an object's name.

Random access, block retrieval for object definitions.

Overview of the Object CT API

Ignoring some organizational overhead, FactoryLink CTs equate to on-disk arrays of C-structures. The object CT is no different, and consists of many arrays of following structure:

```
typedef struct _flobjrec

char  tagname[MAX_TAG_NAME+1];

char  tagdomain[MAX_USR_NAME+1];

char  tagtype[MAX_TYPE_NAME+1];

char  tagdescr[MAX_PROC_DESC+1];

char  tagdimen[MAX_DIM_LENGTH+1];

u16   tagperwhen;

u16   tagchgbits;

TAG   tagno;
```

```
} FLOBJREC;
```

This structure reflects the current attributes that define a FactoryLink object. Applications should treat this structure as opaque and not access its members directly. An function-based interface, included with the Object CT API, should be used to query FLOBJREC's values. Using the API shields the PAK task from future changes that alter the object's structure and its members, yet leave the interface alone.

The Object CT API is a data abstracted set of functions. Its usage generally follows some for variant of the following sequence:

- Open the application's object table.
- Search for definitions for one or more objects.
- Close the object table.

Code Scrap: Printing all objects in a particular domain

```
#include <objct.h>


/*
 *  Function print_objs4dom writes to standard output all objects
 *  configured for a particular domain.
 */
int print_objs4dom(char *flapp, char *tgt_dom)
{
FLOBJREC  rec;
u32       nrecs;
u32       k;
CT        objct;


    if (ct_open_obj(&objct, flapp) != GOOD)
```

```
            return ERROR

        nrecs = ct_nrecs_obj(objct);
        for (k = 0; k < nrecs; k++)
        {
            ct_read_objs(objct, &rec, k, 1);

            if (strcmp(tgt_dim, flobjrec_get_domain(rec)) == 0)
            {
                printf("Object %s in domain %s\n",
                        flobjrec_get_name(rec), tgt_dom);
            }
        }
        ct_close_obj(&objct);
        return nrecs;
    }
```

Normalized Tag Reference Overview

A *FactoryLink object* is a named instantiation of a data type: be it digital, analog, float, long analog, message, or mailbox. The object may be a single instance of its data type, or it may be an array containing multiple instances of the same data type. The name of a FactoryLink object is referred to as the *object name*.

To connote a structured view, a FactoryLink object may be said to be a *member* of another object. A member FactoryLink object has equal standing with its parent object, save that it cannot have members of its own. A member tag can be arrayed, but these usually parallel the dimensions of its parent object.

Finally, FactoryLink object may have a local or remote value source. This source is known as its *node*. This *node* is sometimes referred to as an *external domain*.

A *tag* refers a single location for within the FactoryLink real-time database. A FactoryLink object equates to one or more tags, depending on whether it is an arrayed object and/or whether it has member objects associated with it.

Given these parameters, a reference to a FactoryLink object conforms to the following syntax:

[{***node***}:]{***name***}[***dimension***][...][***.member***]

where:

node	(optional) The source node for the given tag.
name	The base name for the tag.
dimension	(optional) The particular tag element for an arrayed tag.

member	(optional) The sub-component identifier for the base tag.

For example, the object reference “plc1:tagx[4][5].raw” has a *node* of “plc1”, an *id* of “tagx”, the *dimensions* of “[4][5]”, and a *member* of “raw”.

The above syntax provides the precision required to resolve an object reference to a single, real-time database location (tag, for short).

Also given this syntax, the combination of a reference’s ***node***, ***name***, and ***member*** equates to the object’s name, as seen through FLCM’s object list panel. Please keep in mind that the object’s name component may not be sufficient to uniquely identify a tag. It must be accompanied by its associated source, dimension, or member attributes.

Chapter 8 . . . .

FactoryLink API Reference Guide

This chapter provides the following information about each API function:.

- **Call format:** Valid format and syntax for this function
- **Arguments:** List containing the following information about each argument:
  - Type
  - Name
  - Description
  - Method used to pass the argument (by reference or by value)
- **Returns:** Symbolic representation, known as a keyword, of possible values returned by the function, such as ERROR, GOOD, or NULL. Keywords (except NULL) are defined in FLIB.H. NULL is defined by the include files supplied with the compiler in use.

Examples of other keywords representing return codes are listed below:

GOOD
CT_NULL_POINTER
CT_FILE_NOT_OPEN
CT_BAD_INDEX

- **Remarks:** Additional information about the function, such as code fragments in the C language

API functions are listed in alphabetical order.

Refer to "Operating System Notes" on page 342 for operating system-specific notes concerning file name limitations.

CT_CLOSE

Close a CT archive.

Call Format:

```
int  ct_close(ctp)
```

Arguments:

Table 8-30

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

GOOD

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_CANNOT_CLOSE_FILE

CT_CLOSE_OBJ

Object CT API.

31220 Prototype:

```
#include <objct.h>
int ct_close_obj(CT *objct)
```

31220 Arguments:

CT*	objct	(i/o)	Object CT handle.

31220 Returns:

GOOD	CT closed without error.
CT_NULL_PTR	Null pointer passed in for CT.
CT_FILE_NOT_OPEN	CT currently not opened.
CT_CANNOT_CLOSE _FILE	Error occurred closing file.

31220 Description:

Function *ct_close_obj()* closes the object CT. The CT handle should not be referenced after being closed.

31220 See Also:

`ct_open_obj().`

CT_CREATE

Create/truncate/open for update.

Call Format:

```
int ct_create(ctp, dirp, namep)
```

Arguments:

Table 8-31

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
char FAR	*dirp	Directory where file is created	Reference
char FAR	*namep	Name of file created	Reference

Returns:

GOOD

CT_NULL_POINTER

CT_CANNOT_OPEN_FILE

CT_WRITE_ERROR

Remarks:

Write the CT archive header from the caller's ctp→ctarc structure. The caller must fill in all of ctp→ctarc structure except for “magic.”

CT_FIND_INDEX

Find an index.

Call Format:

```
int ct_find_index(ctp, ndx)
```

Arguments:

Table 8-32

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
char FAR	*ndx	Name field	Reference

Returns:

GOOD

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_SEEK_ERROR

CT_READ_ERROR

CT_BAD_DATA

CT_BAD_INDEX

Remarks:

Find an index by matching the name field. The index is returned in ctp→ctndx.

CT_FIND_OBJ

Object CT API.

31220 Prototype:

```
#include <objct.h>
int ct_find_obj(CT *objct, char *objname, FLOBJREC *rec)
```

31220 Arguments:

CT*	objct	(i)	Object CT handle.
char*	objname	(i)	Name of the object to find.
FLOBJREC*	rec	(o)	Buffer for object's definition.

31220 Returns:

GOOD	Object found.
ERROR	Object not found.

31220 Description:

Function *ct_find_obj()* searches the given object CT for the given *objname* and returns its definition.

Function *ct_find_obj()* employs a binary search.

31220 See Also:

`ct_open_obj().`

CT_GET_HDRLEN

Get the length of the current CT table header.

Call Format:

```
int ct_get_hdrlen(ctp)
```

Arguments:

Table 8-33

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

Length of the header. If the CT archive is not opened, the return value is undefined.

Remarks:

Determine the length of the table header of the currently selected CT table.

CT_GET_NAME

Get the name of the current CT table.

Call Format:

```
char *ct_get_name(ctp)
```

Arguments:

Table 8-34

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

Table name. If the CT archive is not opened, the return value is undefined.

Remarks:

Returns a pointer to the name field of the currently selected CT table. Do not modify the memory pointed to by the return value.

CT_GET_NCTS

Determine the number of CT tables in the archive.

Call Format:

```
int ct_get_ncts(ctp)
```

Arguments:

Table 8-35

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

Number of CT tables. If the CT archive is not opened, the return value is undefined.

CT_GET_NRECS

Determine the number of records in the last selected CT table.

Call Format:

```
int ct_get_nrecs(ctp)
```

Arguments:

TypeNameDescriptionPassed By

CT FAR*ctpCT file bufferReference

Returns:

Number of records. If the CT archive is not opened, the return value is undefined.

CT_GET_RECLEN

Determine the record length of records in current CT table.

Call Format:

```
int ct_get_reclen(ctp)
```

Arguments:

Table 8-36

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

Length of individual records. If the CT archive is not opened, the return value is undefined.

Remarks:

All tables have fixed-length records.

CT_GET_TYPE

Get the type field from the current CT table.

Call Format:

```
int ct_get_type(ctp)
```

Arguments:

Table 8-37

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference

Returns:

Table type number. If the CT archive is not opened, the return value is undefined.

CT_OPEN

Open a CT archive file.

Call Format:

int ct_open(ctp, dirp, namep)

Arguments:

Table 8-38

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
char FAR	*dirp	Base path name	Reference
char FAR	*namep	CT file name	Reference

Returns:

GOOD, if the file was successfully opened; otherwise, one of the following:

CT_NULL_POINTER

CT_CANNOT_OPEN_FILE

CT_READ_ERROR

CT_BAD_MAGIC

Remarks:

CT_OPEN places information about the CT archive into the buffer pointed to by ctp. The archive is positioned at the first table in the archive.

CT_READ_HDR

Read the header for ctp into buffer.

Call Format:

```
int  ct_read_hdr(ctp, hdrp)
```

Arguments:

Table 8-39

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	*hdrp	CT header buffer	Reference

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_SEEK_ERROR

CT_READ_ERROR

Remarks:

Call CT_READ_INDEX before calling CT_READ_HDR.

CT_READ_INDEX

Read an index from a CT archive.

Call Format:

```
int  ct_read_index(ctp, ndx)
```

Arguments:

Table 8-40

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
uint	ndx	Index to read	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_SEEK_ERROR

CT_READ_ERROR

Remarks:

Each table in the archive has an associated index containing information about the table. The index is read into the CT file buffer pointed to by ctp. Reading an index selects a specific CT for reading.

CT_READ_REC

Read a record from the current CT into memory.

Call Format:

```
int  ct_read_rec(ctp, recp, rec)
```

Arguments:

Table 8-41

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	recp	Record buffer	Reference
uint	rec	Record number	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_BAD_RECORD

CT_SEEK_ERROR

CT_READ_ERROR

Remarks:

Read the indicated record from the currently selected CT into the memory pointed to by the buffer.

CT_READ_RECS

Read records from the current CT into memory.

Call Format:

```
int  ct_read_recs(ctp, recp, rec, recs)
```

Arguments:

Table 8-42

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	*recp	Record buffer	Reference
uint	rec	Starting record number	Value
uint	recs	Number of records to read	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_BAD_RECORD

CT_SEEK_ERROR

CT_READ_ERROR

Remarks:

Read the indicated records from the currently selected CT into the memory pointed to by the buffer. Reading begins at record number rec.

CT_UPDATE

Open a CT for update.

Call Format:

```
int  ct_update(ctp, dirp, namep)
```

Arguments:

Table 8-43

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
char FAR	*dirp	Directory name	Reference
char FAR	*namep	CT file name	Reference

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_CANNOT_OPEN_FILE

CT_WRITE_ERROR

Remarks:

Write the CT archive header from the caller's ctp→ctarc structure. The caller must fill in all fields of the ctp→ctarc structure except “magic” prior to calling this function.

CT_WRITE_HDR

Write the CT header from buffer to the current CT header.

Call Format:

```
int  ct_write_hdr(ctp, hdrp)
```

Arguments:

Table 8-44

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	*hdrp	CT header buffer	Reference

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_SEEK_ERROR

CT_WRITE_ERROR

Remarks:

Write the CT header from the caller's buffer to the currently selected CT's header.

CT_WRITE_INDEX

Write a CT index record.

Call Format:

```
int  ct_write_index(ctp, ndx)
```

Arguments:

Table 8-45

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
uint	ndx	Index record to write	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_BAD_RECORD

CT_SEEK_ERROR

CT_WRITE_ERROR

Remarks:

Write the specified CT index record from the caller's ctp→ctndx structure. This implicitly selects a CT as the current one.

CT_WRITE_REC

Write a CT record to buffer.

Call Format:

```
int  ct_write_rec(ctp, recp, rec)
```

Arguments:

Table 8-46

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	*recp	Record buffer to write	Reference
uint	rec	Record index to write	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_BAD_RECORD

CT_SEEK_ERROR

CT_WRITE_ERROR

Remarks:

Write the specified CT record from the caller's buffer. This only affects the currently selected CT.

CT_WRITE_RECS

Write CT records into buffer.

Call Format:

```
int  ct_write_recs(ctp, recp, rec, recs)
```

Arguments:

Table 8-47

Type	Name	Description	Passed By
CT FAR	*ctp	CT file buffer	Reference
void FAR	*recp	Records to write	Reference
uint	rec	Record index to start writing	Value
uint	recs	Number of records to write	Value

Returns:

GOOD, if successful; otherwise, one of the following:

CT_NULL_POINTER

CT_FILE_NOT_OPEN

CT_BAD_INDEX

CT_BAD_RECORD

CT_SEEK_ERROR

CT_WRITE_ERROR

Remarks:

Write multiple CT records from the caller's buffer. This only affects the currently selected CT.

FL_ACCESS_MEM

Access memory block.

Call Format:

```
int fl_access_mem(id, vptr, pptrp, sizep)
```

Arguments:

Returns:

Table 8-48

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
VPTR	vptr	Virtual pointer	Reference
void FAR	**pptrp	Pointer to physical pointer to be filled in by kernel	Reference
u32 FAR	*sizep	Pointer to size of memory block, in bytes, to be filled in by kernel	Reference

GOOD or ERROR

- If GOOD, *pptrp and *sizep are filled in.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_ARGUMENT
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_ACCESS_MEM obtains access to the memory block referenced by the specified virtual pointer for the calling process. The virtual pointer must have been previously obtained via FL_ALLOC_MEM. This function translates the virtual pointer (VPTR) to a physical pointer (void *) and a size (u32). Since the

physical pointer returned remains valid only while the database is locked, lock the database before calling this function. It is legal to pass NULL pointers in place of pptrp and/or sizep.

The following example displays how to allocate and access a one K-byte block of sharable memory:

```
id_t    id;
VPTR    vptr;
void   *pptr;
fl_alloc_mem(id, 1024, &vptr);
...
fl_lock(id);
fl_access_mem(id, vptr, &pptr, NULL);
...
fl_unlock(id);
```

FL_ALLOC_MEM

Allocate a specified amount of memory.

Call Format:

```
int fl_alloc_mem(id, size, vptrp)
```

Arguments:

Table 8-49

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
u32	size	Size of memory block, in bytes	Value
VPTR FAR	*vptrp	Pointer to virtual pointer to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *vptrp is filled in.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_ARGUMENT
  - FLE_NULL_POINTER
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_ALLOC_MEM allocates the specified amount of memory and returns a virtual pointer to the caller. The memory allocated is contiguous, sharable with other FactoryLink processes, initialized to all zeros, and is suitably aligned for storage. The virtual pointer may be subsequently used as a reference to the memory. The memory remains allocated until freed with FL_FREE_MEM. The size of the memory block requested must not exceed MAX_MEM_SIZE.

FL_CHANGE_READ

Read the first real-time database element that has changed since it was last read.

Call Format:

```
int  fl_change_read(id, tp, n, ip, vp)
```

Arguments:

Table 8-50

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are to be examined	Reference
uint	n	Number of elements involved	Value
uint FAR	*ip	Pointer to index into tag array to be used and updated, if necessary, by kernel	Reference
void FAR	*vp	Pointer to area to receive the value of the first changed element, if r == GOOD	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_ARGUMENT
  - FLE_NO_CHANGE
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

Test for a change in value of one or more elements in the real-time database. The calling process is immediately informed (it is never blocked) as to whether any of the specified real-time database elements changed since last read. ip specifies the first element to examine. However, in contrast to FL_CHANGE_WAIT, ip does not wrap around the tag array.

In the case of a mailbox element, the value passed back is the MBXMSG for the head message without the associated message data.

FL_CHANGE_READ_TAG_LIST

Read a change in value of one or more elements in a list of real-time database elements.

Call Format:

```
int fl_change_read_tag_list(id, tp, ip, vp)
```

Arguments:

Table 8-51

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG FAR	*tp	Pointer to a single trigger element to be filled in by kernel	Reference
uint FAR	*ip	Pointer to index to be filled in by kernel	Reference
void FAR	*vp	Pointer to value buffer to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tp, *ip, and *vp are filled in.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_NO_TAG_LIST
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_CHANGE_READ_TAG_LIST looks for a change in value of one or more elements in the tag list registered to the calling process. It is similar in operation to FL_CHANGE_READ in that it does not block the caller.

FL_CHANGE_WAIT

Read the first real-time database element that has changed since it was last read; if no change, go to sleep until a change occurs.

Call Format:

```
int  fl_change_wait(id, tp, n, ip, vp)
```

Arguments:

Table 8-52

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are to be examined	Reference
uint	n	Number of elements involved	Value
uint FAR	*ip	Pointer to index into tag array to be used and updated, if necessary, by kernel	Reference
void FAR	*vp	Pointer to area to receive the value of the first changed element, if r == GOOD	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_ARGUMENT
  - FLE_NO_CHANGE

Remarks:

Wait on a change in value of one or more elements in the real-time database. If any of these elements have changed, control is returned immediately to the caller (just as in FL_CHANGE_READ). Otherwise, the calling process is put to sleep (blocked) until one or more of the specified real-time database elements changes, at which point it is awakened. While sleeping, the calling task is also awakened if another process wakes it up (this can be done either directly via FL_WAKEUP or indirectly via FL_SET_TERM_FLAG). Although all n elements are waited upon, only one value is read and returned in vp. This is the one detected as the first one to change. In deciding which element is first, ip is used in wraparound fashion within the tag array.

In the case of a mailbox element, the value passed back is the MBXMSG for the head message without the associated message data.

FL_CHANGE_WAIT_TAG_LIST

Wait for a change in value of one or more elements in the list of real-time database elements.

Call Format:

```
int fl_change_wait_tag_list(id, tp, ip, vp)
```

Arguments:

Table 8-53

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG FAR	*tp	Pointer to a single trigger element to be filled in by kernel	Reference
uint FAR	*ip	Pointer to index to be filled in by kernel	Reference
void FAR	*vp	Pointer to value buffer to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tp, *ip, and *vp are filled in.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_NO_TAG_LIST
  - FLE_SIGNALLED
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_CHANGE_WAIT_TAG_LIST waits for a change in value of one or more elements in the tag list registered to the calling process. It is similar in operation to FL_CHANGE_WAIT in that it blocks the caller until a change occurs or some other event awakens the caller (such as receipt of a signal).

FL_CLEAR_CHNG

Clear the calling task's change-status flags for specified real-time database elements.

Call Format:

```
int  fl_clear_chng(id, tp, n)
```

Arguments:

Table 8-54

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying	Reference
		which elements are involved	
uint	n	Number of elements involved	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

Clear (to **0**) the change state of the specified real-time database elements for the calling process only. This function undoes the action of FL_SET_CHNG. It is also useful for establishing initial conditions for a programming loop.

FL_CLEAR_WAIT

Clear the calling task's wait flags for specified real-time database elements.

Call Format:

```
int  fl_clear_wait(id, tp, n)
```

Arguments:

Table 8-55

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying	Reference
		which elements are involved	
uint	n	Number of elements involved	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

Clear (to **0**) the wait flag of the specified real-time database elements for the calling process only. Also, use this function to establish initial conditions.

FL_COUNT_MBX

Determine the number of messages in a mailbox, validate a mailbox, or monitor a mailbox.

Call Format:

```
int fl_count_mbx(id, mbx, np)
```

Arguments:

Table 8-56

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG	mbx	Mailbox to be accessed	Reference
uint FAR	*np	Message count to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, also returns *np.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

Use FL_COUNT_MBX to determine how many messages are present in a mailbox. Generally, call FL_COUNT_MBX prior to allocating space for an array of MBXMSGs and calling FL_QUERY_MBX. Also, use this function to validate a mailbox TAG and to monitor a mailbox.

FL_CREATE_RTDB

Create an instance of the FactoryLink Real-Time Database. This function is called by the run-time manager to create a formatted, initialized (empty) instance of the RTDB for each domain.

Call Format:

```
int fl_create_rtdb(pgm_dir,app_dir,name,users,ucnt,utype,tcnt)
```

Arguments:

Table 8-57

Type	Name	Description	Passed By
char	*pgm_dir	FactoryLink program directory ({FLINK})	Reference
char	*app_dir	Application directory ({FLAPP})	Reference
char	*name	Application invocation name ({FLNAME})	Reference
KUSR	ucnt	Number of elements in **user** structure	Value
KTYPE	*utype	Database types information	Reference
int	tcnt	Number of elements in utype	Value

Returns:

GOOD or ERROR

Remarks:

Normally, this function is only called by the run-time manager.

The PGM_DIR and APP_DIR arguments contain the program and application directory names. This information is also placed into the process environment by the run-time manager.

The NAME argument contains a pointer to a string containing the invocation name for this application. This name is used by tasks to connect to the correct instance of the application.

The USERS argument points to an array of KUSR structures. The KUSR structure type is a new structure that contains information about the number of users to allocate.

```
typedef struct _KUSR
{
    char    u_name[MAX_USER_NAME];
    char    u_parent[MAX_USER_NAME];
    int     u_instance;
} KUSR;
```

The PARENT member is for use in domain processing. The INSTANCE member defines the number of users that can attach to the database with this domain name.

The UCNT argument specifies the number of elements in the array pointed to by USERS.

The UTYPE argument points to an array of KTYPE structures that define the segments contained in the real-time database.

The TCNT argument specifies the number of elements in the array pointed to by UTYPE.

FL_DBFMTT

Prepare a formatted string from a set of real-time database element values.

Call Format:

```
int  fl_dbfmtt(id,maxlen,bp0,fp,args)
```

Arguments:

Table 8-58

Type	Name	Description	Passed By
id_t	id	Task ID used to access real-time database	Value
int	maxlen	Maximum number of characters that can be stored in output buffer	Value
char FAR	bp0	Pointer to the memory buffer that stores formatted output	Reference
char FAR	fp	Format string	Reference
TAG	*args	Tag array (to be read from the real-time database)	Reference

Returns:

Length of string

Remarks:

This function is similar to the C-language function SPRINTF.

The format string can contain a format specifier of the following form:

```
%m.nc
```

where

m Represents the width of the output (optional). If **m** is preceded by a hyphen (-), the output is left justified. By default, the output is right justified.

n Represents the precision of the output (optional)

cIs one of the following characters:

Character	Output
B	String representing a DIGITAL value as **ON** or **OFF**
b	Number representing a DIGITAL value as **0** or **1**
d	Integer value in base 10
u	Unsigned integer value in base 10
x	Integer value in base 16 (hexadecimal)
o	Integer value in base 8 (octal)
c	Single character
s	Character string
f	Floating-point number
g	Floating-point number
%	A percent sign

For each format specifier, the next element in the args array is used to read a value from the real-time database. The element value is converted to the appropriate type for the format specifier and then formatted into the output buffer.

FL_DELETE_RTDB

Delete (remove and destroy) an instance of the FactoryLink Real-Time Database. This function is called by the run-time manager to release all memory associated with a particular instance of the RTDB in a domain.

Call Format:

```
int fl_delete_rtdb(rtid)
```

Arguments:

Table 8-59

Type	Name	Description	Passed By
int	rtid	FactoryLink ID value of this RTDB instance (which was returned by fl_create_rtdb when this RTDB instance was created)	Value

Returns:

GOOD or ERROR

Remarks:

This function should only be called by the Run-Time Manager.

FL_ERRNO

Return the last FactoryLink error number generated by the calling process.

Call Format:

```
int  fl_errno(id)
```

Arguments:

TypeNameDescriptionPassed By

id_tidCaller's FactoryLink IDValue

Returns:

Last error number or GOOD if none

Remarks:

Return the last FactoryLink error number generated by the calling process (during the last call to the kernel that resulted in an error).

FL_EXIT_APP

Exit an instance of an application.

Call Format:

```
int  fl_exit_app(id)
```

Arguments:

Table 8-60

Type	Name	Description	Passed By
id_t	id	FactoryLink task ID	Value

Returns:

GOOD or ERROR

Remarks:

Normally, only the Run-Time Manager should call this function.

FL_EXIT_APP exits an instance of an application in the multi-user environment. This function should be called by the task that called FL_INIT_APP before that task exits.

FL_FORCED_WRITE

Force-write a specified element into the real-time database.

Call Format:

```
int  fl_forced_write(id, tp, n, vp)
```

Arguments:

Table 8-61

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are to be force-written	Reference
uint	n	Number of elements to be force-written	Value
void FAR	*vp	Pointer to area holding the values	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG
  - FLE_OUT_OF_MEMORY

Remarks:

This function operates similarly to FL_WRITE except that all change states are set (to **1**), regardless of whether the new values written into the database are the same as the previous values stored there. Upon completion of writing, those client processes waiting on any of the elements involved are awakened. As before, any attempt by the writing process to write a value into a synchronous element that is still unread causes the writer to block until it has been read. Blocking does not occur until an attempt has been made to write all of the specified elements. All asynchronous elements are written on the first pass.

FL_FREE_MEM

Free memory.

Call Format:

```
int fl_free_mem(id, vptr)
```

Arguments:

Table 8-62

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
VPTR	vptr	Virtual pointer	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_ARGUMENT
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_FREE_MEM frees the memory referenced by the specified virtual pointer, that must have been previously obtained by a call to FL_ALLOC_MEM. FL_FREE_MEM undoes the action of FL_ALLOC_MEM. The memory is returned to the free memory pool and the virtual pointer can no longer be used as a reference to it.

FL_GET_APP_DIR

Return the application directory for the specified process.

Call Format:

```
int fl_get_app_dir(id, dir)
```

Arguments:

Table 8-63

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
char FAR	*dir	Pointer to directory string to be	Reference
		filled in by kernel	

Returns:

GOOD or ERROR

- If GOOD, *dir is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_APP_DIR returns the application directory for the process specified by the FactoryLink ID (usually, but not necessarily, the ID of the caller). It returns the application directory as a full path name, including a drive letter, if applicable. This directory is the root directory for all "dynamic" FactoryLink files; that is, for all files that are part of a particular FactoryLink application.

FL_GET_APP_GLOBALS

Read the global flags for a specific instance of an application.

Call Format:

```
int fl_get_app_globals(id, ugp)
```

Arguments:

Table 8-64

Type	Name	Description	Passed By
id_t	id	FactoryLink task ID	Value
KGLOBALS	*ugp	Pointer to global information return buffer to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, buffer to which *ugp points is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_APP_GLOBALS replaces the FL_GET_GLOBALS function from previous releases. It returns the global flags for a specific instance of an application as specified by the FactoryLink task ID passed.

FL_GET_CMD_LINE

Return the command line for the specified process.

Call Format:

```
int fl_get_cmd_line(id, line)
```

Arguments:

Table 8-65

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
char FAR	*line	Pointer to command line to be filled in by kernel	Reference

Returns:

GOOD or ERROR.

- If GOOD, *dir is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_CMD_LINE returns the command line for the process specified by the FactoryLink ID (usually, but not necessarily, the ID of the caller). The command line for a process is an ASCII string that generally contains instructions to the process to alter its normal behavior, and additional environmental information.

FL_GET_COPYRT

Return a pointer to a copyright message for FactoryLink.

Call Format:

```
char *fl_get_copyrt(void)
```

Arguments:

None

Returns:

Pointer to copyright message

Remarks:

Do not modify the pointer that is returned by this function.

FL_GET_CTRL_TAG

Return the control tag, which refers to a digital or analog real-time database element, for the specified process.

Call Format:

```
int fl_get_ctrl_tag(id, tag)
```

Arguments:

Table 8-66

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
TAG FAR	*tag	Pointer to control tag to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tag is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_CTRL_TAG returns the control tag, which refers to a digital or analog real-time database element, for the process specified by the FactoryLink ID (which is not necessarily, and usually is not, the ID of the caller). The control tag for a process is used to control starting and stopping of the process. Any process may write a value of **1** (or any other nonzero value) to the control tag to tell the Run-Time Manager to start the process. Likewise, it may write a value of **0** to tell the Run-Time Manager to stop the process by setting the termination flag of the process.

FL_GET_ENV

Return the KENV structure of the client process.

Call Format:

```
int  fl_get_env(id, uep)
```

Arguments:

Table 8-67

Type	Name	Description	Passed By
id_t	id	Client FactoryLink ID (**0**-**30**), used as an index into the array of KENVs kept by the kernel	Value
KENV FAR	*uep	Pointer to KENV structure to be returned	Reference

Returns:

GOOD or ERROR

Remarks:

Return a KENV structure describing the environment of the given client process, usually the calling process. The Run-Time Manager calls FL_SET_ENV to define this structure before the client process is started. This function returns all fields of the KENV structure.

FL_GET_MSG_TAG

Return the value of the real-time database message element for the specified process.

Call Format:

```
int fl_get_msg_tag(id, tag)
```

Arguments:

Table 8-68

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
TAG FAR	*tag	Pointer to message element to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tag is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_MSG_TAG returns the value of a real-time database message element for the process specified by the FactoryLink ID (usually, but not necessarily, the ID of the caller). The MSG database element is written by the process and read by the Run-Time Manager. Its ASCII value is assumed to continually describe what the process is doing, what problems it is encountering, and other similar information (another self-reporting mechanism). It is passed directly from the Run-Time Manager to the Real-Time Graphics process for display on the Run-Time Manager screen.

FL_GET_NPROCS

Get the number of client processes permitted to run concurrently.

Call Format:

```
int  fl_get_nprocs()
```

Arguments:

None

Returns:

Number of client processes

FL_GET_PGM_DIR

Return the program directory for the specified process.

Call Format:

```
int fl_get_pgm_dir(id, dir)
```

Arguments:

Table 8-69

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
char FAR	*dir	Pointer to directory string to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *dir is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_PGM_DIR returns the program directory for the process specified by the FactoryLink ID (usually, but not necessarily, the ID of the caller). The program directory is returned as a full path name, including a drive letter, if applicable. This directory is the root directory for all “static” FactoryLink files, that is, for all files that are part of an installed FactoryLink system, but are not part of any FactoryLink application.

FL_GET_STAT_TAG

Return the value of the real-time database analog status element for the specified process.

Call Format:

```
int fl_get_stat_tag(id, tag)
```

Arguments:

Table 8-70

Type	Name	Description	Passed By
id_t	id	FactoryLink ID	Value
TAG FAR	*tag	Pointer to status element to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tag is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_GET_STAT_TAG returns the value of the status element for the process specified by the FactoryLink ID (not necessarily the ID of the caller). The ANA status real-time database element is written by the process and read by the Run-Time Manager. Its numeric value is assumed to continually reflect the (self-reported) status of the process, as follows:

Table 8-71

Status	Symbolic Name	Meaning
0	FLS_INACTIVE	Process is inactive (not running)
1	FLS_ACTIVE	Process is active (running)
2	FLS_ERROR	Error (non-fatal error occurred)
3	FLS_STARTING	Starting (initialization in progress)
4	FLS_STOPPING	Stopping (shutdown in progress)
other	—-	Error (non-fatal error occurred)

The Run-Time Manager converts the status value to an ASCII string and passes it to the Real-Time Graphics process for display on the Run-Time Manager screen.

FL_GET_TAG_INFO

Get the information associated with a specified list of real-time database elements.

Call Format:

```
int fl_get_tag_info(tp, n, dp, op)
```

Arguments:

Table 8-72

Type	Name	Description	Passed By
TAG FAR	*tp	Pointer to tag array specifying which elements are involved	Reference
uint	n	Number of elements involved	Value
i16 FAR	*dp	Pointer to description array to be filled in by kernel and returned to caller	Reference
u16 FAR	*op	Pointer to offset array to be filled in by kernel and returned to caller	Reference

Returns:

GOOD or ERROR

Remarks:

The dp array is filled in with one of the following values, which correspond to the elements passed to it:

Table 8-73

Value	Description
FL_BAD_DATA	Element is out-of-range (bad t_data field)

Table 8-73

Value	Description
FL_BAD_TYPE	Element is a bad t_type field
FL_UNDEFINED	Element is undefined
FL_DIGITAL	Element is a digital (DIG)
FL_ANALOG	Element is an analog (ANA)
FL_MESSAGE	Element is a message (MSG)
FL_LANALOG	Element is a long analog (LONGANA)
FL_FLOAT	Element is floating point (FLP)
FL_MAILBOX	Element is mailbox (holds MBXMSGs)

The kernel fills in the op array with the offset into the vp buffer, where the value would be stored if FL_READ, FL_WRITE, or FL_FORCED_WRITE were called with the argument list (id, tp, n, vp). The caller may set either or both of the pointers dp or op to NULL if it does not wish to receive the corresponding information. FL_GET_TAG_INFO returns GOOD if all n elements in the tp array are valid and ERROR if one or more of them is bad.

FL_GET_TAG_LIST

Retrieve the tag list (a list of real-time database elements) for the calling process.

Call Format:

```
int fl_get_tag_list(id, tp, n, np)
```

Arguments:

Table 8-74

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG FAR	*tp	Pointer to array of trigger elements to be filled in by kernel	Reference
uint	n	Maximum number of trigger elements	Reference
uint FAR	*np	Actual number of trigger elements filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *tp and *np are filled in.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_NO_TAG_LIST
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

Assuming the tag list (list of real-time database elements) has been registered using the FL_SET_TAG_LIST function, FL_GET_TAG_LIST retrieves the tag list for the calling process.

FL_GET_TICK

Get the current clock tick and/or current date and time maintained and reported by the operating system.

Call Format:

```
int fl_get_tick(tickp, datetimep)
```

Arguments:

Table 8-75

Type	Name	Description	Passed By
u32 FAR	*tickp	Pointer to place to store clock tick	Reference
KDT FAR	*datetimep	Pointer to place to store data and time	Reference

Returns:

Always GOOD

Remarks:

The clock tick is graduated in millisecond units and is non-decreasing until wraparound time (about once a month). The chief use is in measuring elapsed time. The date and time are stored in a DATETIME structure familiar to the operating system. Either tickp or datetimep may be NULL, which the kernel interprets as a lack of interest in the corresponding value; therefore, the kernel does not store the value.

FL_GET_TITLE

Return a pointer to the name of the product (“FactoryLink”).

Call Format:

```
char *fl_get_title(void)
```

Arguments:

None

Returns:

Pointer to name of product

Remarks:

Do not modify the pointer that is returned by this function.

FL_GET_VERSION

Get the kernel version number.

Call Format:

```
uint fl_get_version()
```

Arguments:

None

Returns:

FactoryLink kernel version number. Numerically returns version x.y as (x * 256) + y.

FL_GETVAR

Returns the lookup value of an environment variable.

Call Format:

```
char *fl_getvar(name, buf, len)
```

Arguments:

Table 8-76

Type	Name	Description	Passed By
char	*name	Pointer to environment variable name	Value
char	*buf	Buffer to hold returned value	Reference
int	len	Length of the value buffer 'buf'	Value

Returns:

If successful in finding the specified variable, the function returns a pointer to the buf argument.

If the variable is not found, the function returns NULL.

Remarks:

This function is similar to GETENV from the standard C library. The purpose of this special API function is to allow the kernel to read the environment under OS/2.

FL_GLOBAL_TAG

Retrieve the tag number for one or more global elements.

Call Format:

```
int fl_global_tag(tagp, n, ids)
```

Arguments:

Table 8-77

Type	Name	Description	Passed By
TAG FAR	*tagp	Array where tag numbers are returned	Reference
uint	n	Number of elements to read	Value
uint	*ids	Array of identification numbers	Reference

Returns:

Number of elements read from the global CT file.

Remarks:

GLOBAL.CT contains global elements that are available to all tasks. For each member of the ids array, GLOBAL.CT is searched for a matching value. The associated element is then copied into the corresponding member of the tagp array.

Note: The id values are defined in FLIB.H.

The following code illustrates an example of the use of this function:

```
#include <flib.h>
  ...
uint ids[3] = {GT_SECONDS, GT_MINUTES, GT_HOURS};
TAG gtags[3];
```

```
ANA vals[3];

/* Find the tag numbers for the elements where the
   timer task is updating the seconds, minutes, and hours */
fl_global_tag(&gtags[0], 3, &ids[0]);
/* Read the time tags */
fl_read(taskid, &gtags[0], 3, &vals[0]);
seconds = vals[0];
minutes = vals[1];
hours   = vals[2];
```

FL_HOLD_SIG

Prevent or allow signal delivery for the calling process.

Call Format:

```
int fl_hold_sig(id, sig, hold)
```

Arguments:

Table 8-78

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
int	sig	Signal (**0-31**) to be affected	Value
int	hold	Hold value: **1** = prevent signal delivery **0** = allow signal delivery	Value

Returns:

The previous hold value (**1** or **0**) for the signal or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns FLE_BAD_ARGUMENT.

Remarks:

Initially, all signals except **0** and **1** are held; that is, by default, signals **0** and **1** are deliverable to the calling process, but others are not. A FactoryLink process wishing to be notified when its tag list of real-time database elements has changed must therefore execute the following function to allow delivery of this signal:

```
fl_hold_sig(id, FLC_SIG_TAG_LIST_CHANGED, 0);
```

It is legal to hold any signal with FL_HOLD_SIG, including signals **0** and **1**.

FL_ID_TO_NAME

Translate a FactoryLink ID to a process name.

Call Format:

```
int fl_id_to_name(id, name)
```

Arguments:

Table 8-79

Type	Name	Description	Passed By
id_t	id	FactoryLink ID to be translated	Value
char FAR	*name	Pointer to process name to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If GOOD, *name is filled in.
- If ERROR, an invalid FactoryLink ID is assumed.

Remarks:

FL_ID_TO_NAME checks the FactoryLink ID and, if the ID is valid, returns the associated process name from the KPROC array.

FL_INIT

Initialize the FactoryLink kernel and its global data area. (OBSOLETE; replace calls with calls to FL_INIT_APP for MUE.)

Call Format:

```
int fl_init(bp)
```

Arguments:

Table 8-80

Type	Name	Description	Passed By
char FAR	*bp	Pointer to command specifying kernel's initial parameters, options, and so on (currently ignored)	Reference

Returns:

ERROR (see Remarks)

Remarks:

Only the Run-Time Manager should call this function, and only once during run-time initialization.

FL_INIT has been retained from previous releases in order to maintain the same entry points into the kernel for compatibility with older applications, but will always return ERROR when called from the multi-user environment.

FL_INIT_APP

Initialize the FactoryLink kernel and its global data area.

Call Format:

```
int fl_init_app(name, domain, user, proc)
```

Arguments:

Table 8-81

Type	Name	Description	Passed By
char	*name	Application invocation name ({FLNAME})	Reference
char	*domain	Domain name ({FLDOMAIN})	Reference
char	*user	User name ({FLUSER})	Reference
KPROC	*proc	Process table	Reference

Returns:

GOOD or ERROR

Remarks:

Normally, only the Run-Time Manager should call this function, and only once during run-time initialization.

FL_INIT_APP uses the supplied name to locate the shared memory area containing that instance of the real-time database. The NAME argument points to a string that uniquely identifies the application. All possible shared memory areas are searched for the one containing the NAME string. If no match is found, FL_INIT_APP returns ERROR.

If this real-time database instance is located, fl_init_app searches the kernel instance table for an unused entry with a domain name the same as the domain argument. The domain argument must be one of the names passed to FL_CREATE_RTDB in the KUSR structure.

When a free entry is located, it is marked as in use and the user argument is copied into the instance table entry. All tasks which pass the same combination of {FLNAME},{FLDOMAIN},{FLUSER} to the FL_PROC_INIT_APP function will use this instance table entry and share the same USER domain data areas.

The index of the instance table entry is returned as the application ID. The ID is used by tasks as the upper byte of the task ID.

The previous API for database initialization, FL_INIT, will be retained for compatibility with previous versions so as to maintain the same entry points into the kernel, but will always return ERROR.

FL_LOCK

Lock the real-time database on behalf of the calling process.

Call Format:

```
int fl_lock(id)
```

Arguments:

Table 8-82

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value

Returns:

GOOD or ERROR

Remarks:

Only one client process may have the real-time database locked at any given time. If the calling process calls FL_LOCK and the real-time database is already locked by that same process, a counter is incremented and the lock remains in effect for the caller. This counter allows calls to FL_LOCK and FL_UNLOCK to be nested.

If another client process has already locked the real-time database, the caller is put to sleep (blocked) until his lock request can be honored. Upon return from FL_LOCK, only the calling process is granted access to the real-time database until it makes a corresponding call to FL_UNLOCK, provided that it does not execute FL_WAIT, either directly or indirectly. (FL_WAIT releases its lock and puts it to sleep. When it is reawakened, the lock is reinstated).

If the caller wants to keep the real-time database locked and thereby retain exclusive access to it, it must not call FL_CHANGE_WAIT or write any synchronous elements via FL_WRITE or FL_FORCED_WRITE.

FL_NAME_TO_ID

Translate a process name to a FactoryLink ID.

Call Format:

```
id_t fl_name_to_id(id, name)
```

Arguments:

Table 8-83

Type	Name	Description	Passed By
char FAR	*name	Pointer to process name to be translated	Reference
id_t	id	Task ID of calling task	Value

Returns:

FactoryLink ID or ERROR

- If ERROR, an invalid process name is assumed.

Remarks:

FL_NAME_TO_ID searches the KPROC array for the specified process name and, if the process name is found, returns the associated FactoryLink ID.

FL_PATH_ACCESS

Check file access mode for a given file.

Call Format:

```
int fl_path_access(path)
```

Arguments:

Table 8-84

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- The file access mode of the file as one of the following character strings:
  - NPATH_READ
  - NPATH_WRITE
  - NPATH_READ | NPATH_WRITE
- If specified file does not exist, returns ERROR.

Remarks:

The function FL_PATH_ACCESS returns a string informing the calling program of the mode(s) (read-only, write-enable, or read/write) in which the calling program is authorized to access the specified file, if available.

See also the other path functions (FL_PATH and related calls).

FL_PATH_ADD

Catenates two normalized paths.

Call Format:

```
void fl_path_add(path1,path2)
```

Arguments:

Table 8-85

Type	Name	Description	Passed By
NPATH	*path1	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
NPATH	*path2	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

N/A

Remarks:

FL_PATH_ADD catenates two paths. Any missing component of the second path **p2** is taken from the first path **p1** or from the current directory if the first path is null.

See also the other path functions (FL_PATH and related calls).

FL_PATH_ADD_DIR

Adds one subdirectory specification per call to the end of the directory portion of a path.

Call Format:

```
void fl_path_add_dir(path, dir)
```

Arguments:

Table 8-86

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*dir	Directory name in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_ADD_DIR adds a subdirectory specification to the end of the directory portion of a path. Only one subdirectory can be added to a path during each call to FL_PATH_ADD_DIR. The subdirectory name should not contain any path-separator characters.

See also the other path functions (FL_PATH and related calls).

FL_PATH_ALLOC

Allocate a normalized path name buffer.

Call Format:

```
NPATH *fl_path_alloc(void)
```

Arguments:

None

Returns:

- If successful, returns a pointer to a normalized path buffer. (The pointer is returned in the variable named in the call; e.g., **PATH = fl_path_alloc**)
- If unsuccessful, returns NULL; an invalid process name is assumed.

Remarks:

The function FL_PATH_ALLOC allocates and returns a pointer to a normalized path buffer. Programmers should call this function, rather than allocate the NPATH structure directly, so that a buffer for system-dependent information can be added to the path buffer.

The C structure NPATH for a normalized path is:

```
typedef struct _npath
{
    char    node[MAX_NODE_NAME];
    char    device[MAX_DEVICE_NAME];
    char    dir[MAX_DIRECT_NAME];
    char    file[MAX_FILE_NAME];
    char    wild[MAX_FILE_NAME];
    char    version[MAX_VERSION];
    char    verwild[MAX_VERSION];
    long    dt;
    long    size;
```

```
    int     type;
    int     magic;
    void   *sysdata;
} NPATH;
```

See also the other path functions (FL_PATH and related calls).

FL_PATH_CLOSEDIR

Ends a directory search for a file. (See also FL_PATH_OPENDIR and FL_PATH_READDIR.)

Call Format:

```
void fl_path_closedir(path)
```

Arguments:

Table 8-87

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

N/A

Remarks:

FL_PATH_CLOSEDIR ends a directory search.

See also the other path functions (FL_PATH_OPENDIR and related calls).

FL_PATH_CREATE

Create an empty file using the complete path specified.

Call Format:

```
int fl_path_create(path)
```

Arguments:

Table 8-88

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_CREATE creates an empty file using the complete path specified in the call by the pointer **p**. The file may then be opened, closed, copied, or referenced by any task with the proper file access privileges.

FL_PATH_CWD

Build a normalized path for the current working directory.

Call Format:

```
NPATH *fl_path_cwd(path)
```

Arguments:

Table 8-89

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns a pointer to a normalized path buffer. (The pointer is returned in the variable named in the call; e.g., **PATH = fl_path_cwd**)
- If unsuccessful, returns NULL.

Remarks:

FL_PATH_CWD builds a normalized path for the current working directory. If the NPATH argument is NULL, FL_PATH_CWD first calls FL_PATH_ALLOC to allocate a NPATH buffer. Either way, the working directory may now be accessed using the path name built.

FL_PATH_DATE

Places formatted system date and time stamp from a specified file's header into specified buffer.

Call Format:

```
long fl_path_date(path, buf, length)
```

Arguments:

Table 8-90

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*buf	Pointer to a buffer for receiving date	Reference
size_t	length	Length of output buffer	Value

Returns:

- If successful, returns file's date-and-time stamp.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_DATE formats the date and time stamp on a file (the date/time the file was last updated) into the caller's buffer and returns the date and time (concatenated) as a long integer.

FL_PATH_GET_SIZE

Returns the size in bytes of the specified file.

Call Format:

```
long fl_path_get_size(path)
```

Arguments:

Table 8-91

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns size of file in bytes.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_GET_SIZE returns the size of a specific file in bytes.

FL_PATH_GET_TYPE

Returns the file type of the specified file.

Call Format:

```
long fl_path_get_size(path)
```

Arguments:

Table 8-92

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns file type as one of the constants listed below.

  NPATH_REGULAR
  NPATH_DIRECTORY
  NPATH_FIFO (UNIX only)
  NPATH_DEVICE (UNIX only)

- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_GET_TYPE returns the type of the file.

FL_PATH_INFO

Initialize date, time, size, and type of files allowed for the specified path.

Call Format:

```
int fl_path_info(path)
```

Arguments:

Table 8-93

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_INFO initializes the date, time, size, and type for the path. If the path does not exist, FL_PATH_INFO returns ERROR. Otherwise, it returns GOOD.

This function is called automatically by FL_PATH_OPENDIR and FL_PATH_READDIR.

FL_PATH_MKDIR

Creates the directory specified by the directory portion of the indicated path. (See also FL_PATH_RMDIR.)

Call Format:

```
int fl_path_mkdir(path)
```

Arguments:

Table 8-94

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_MKDIR creates the directory given by the directory portion of the path, if the directory does not already exist. It will create all directories and subdirectories necessary for the path, up to and including the last subdirectory specified in the NPATH structure.

For example, if none of the following directories exist:

```
/test
/test/mystuff
/test/mystuff/log
```

the following code fragment in C would create them all, in hierarchical order:

```
NPATH *np;
np = fl_path_set_dir(NULL, “/test/mystuff/log”);
fl_path_mkdir(np);
```

This function returns GOOD if the directories already exist. It returns ERROR only if at least one of the directories cannot be created because of a system error, insufficient privilege levels on the part of the caller, or the like.

See also the other path functions (FL_PATH_RMDIR and related calls).

FL_PATH_NORM

Convert part or all of a system-specific path string into a normalized path.

Call Format:

```
NPATH *fl_path_norm(path, path_buf)
```

Arguments:

Table 8-95

Type	Name	Description	Passed By
NPATH	*p	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*path_buf	Source string	Reference

Returns:

- If successful, returns pointer to the NPATH structure; also, if the pointer argument **p** passed in was NULL, a NPATH buffer has been allocated and the pointer value is now set.
- If unsuccessful, returns NULL.

Remarks:

FL_PATH_NORM converts a system-specific path string into a normalized path. Any or all components of the path may be left out. If the NPATH argument is NULL, FL_PATH_NORM first calls FL_PATH_ALLOC to allocate a NPATH buffer.

FL_PATH_OPENDIR

Begins a directory search for a file. (See also FL_PATH_CLOSEDIR and FL_PATH_READDIR.)

Call Format:

```
int fl_path_opendir(path)
```

Arguments:

Table 8-96

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful in opening directory, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_OPENDIR begins a directory search operation. The current directory information contained in NPATH is used as the directory to search, and, the current wild card pattern is used to select files. FL_PATH_OPENDIR returns GOOD if the directory could be opened for search, or ERROR if it could not.

FL_PATH_OPENDIR reads the first entry in the directory.

See also the other path functions (FL_PATH_CLOSEDIR and related calls).

FL_PATH_READDIR

Reads next matching file from directory during a directory search for a file. (See also FL_PATH_OPENDIR and FL_PATH_READDIR.)

Call Format:

```
int fl_path_readdir(path)
```

Arguments:

Table 8-97

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_READDIR reads the next matching file in the directory and places the name of the file into the file name component of the path. The file type, date, time, and size are also stored in the NPATH structure. FL_PATH_ READDIR returns GOOD if a matching file was found or ERROR if not.

The following code fragment in C demonstrates how to use directory search functions to print a directory listing.

```
NPATH *p;
char  date[80];
char  time[80];
char  fullpath[MAX_PATH_NAME];


p = fl_path_norm(NULL, “*.*”);
if ( fl_path_opendir(p) == ERROR )
```

```
{
printf("Directory Not found\n");
return;
}


do
{
fl_path_date(p, date);
fl_path_time(p, time);
fl_path_sys(p, fullpath);
printf("%s %s %s\n", date, time, fullpath);
} while ( fl_path_readdir(p) != ERROR );
fl_path_closedir(p);
fl_path_free(p);
```

8

FactoryLink API Reference Guide

See also the other path functions (FL_PATH_ OPENDIR, FL_PATH_CLOSEDIR and related calls).

FL_PATH_REMOVE

Remove (delete) the file specified by the complete path given. (See also FL_PATH_CREATE.)

Call Format:

```
int fl_path_remove(path)
```

Arguments:

Table 8-98

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_REMOVE removes the file specified by the complete path given.

See also the other path functions (FL_PATH_CREATE and related calls).

FL_PATH_RMDIR

Remove (delete) the directory specified by the directory portion of the indicated path. (See also FL_PATH_MKDIR.)

Call Format:

```
int fl_path_rmdir(path)
```

Arguments:

Table 8-99

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference

Returns:

- If successful, returns GOOD.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_RMDIR deletes the directory given by the directory portion of the path.

See also the other path functions (FL_PATH_MKDIR and related calls).

FL_PATH_SET_DIR

Replaces the directory portion of the specified path with the directory argument specified converted to normalized form.

Call Format:

```
NPATH *fl_path_set_dir(path, dir)
```

Arguments:

Table 8-100

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*dir	Directory name in system-specific format	Reference

Returns:

- If successful, returns pointer to NPATH structure.
- If unsuccessful, returns NULL.

Remarks:

FL_PATH_SET_DIR replaces the directory portion of the path with the specified directory argument after converting the argument to normalized form. If the NPATH argument is NULL, FL_PATH_SET_DIR first calls FL_PATH_ALLOC to allocate a NPATH buffer. The file name, extension and version are not modified by the FL_PATH_SET_DIR function.

The FL_PATH_SET_DIR function can be used to convert a system-specific path string into a normalized path if the path is known to refer to a directory.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SET_DEVICE

Replaces the drive (device) name portion of the specified path with the argument specified, after converting the argument to normalized form.

Call Format:

```
void fl_path_set_device(path, drive)
```

Arguments:

Table 8-101

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*drive	Device (drive) name in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_SET_DEVICE replaces the drive (device) name portion of the path with the specified argument after converting the argument to normalized form.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SET_EXTENSION

Replaces the file extension portion of the specified path with the argument specified after converting the argument to normalized form.

Call Format:

```
void fl_path_set_extension(path, extension)
```

Arguments:

Table 8-102

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*extension	File name extension in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_SET_EXTENSION replaces the file extension portion of the path with the specified argument after converting the argument to normalized form.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SET_FILE

Replaces the file name portion of the specified path with the specified argument after converting the argument to normalized form.

Call Format:

```
void fl_path_set_file(path, file)
```

Arguments:

Table 8-103

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*file	File name in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_SET_FILE replaces the file name portion of the path with the specified argument after converting the argument to normalized form.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SET_NODE

Replaces the node name portion of the specified path with the specified argument after converting the argument to normalized form.

Call Format:

```
void fl_path_set_node(path, node)
```

Arguments:

Table 8-104

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*node	Node name in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_SET_NODE replaces the node name portion of the path with the specified argument after converting the argument to normalized form.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SET_PATTERN

Sets a “wild card” pattern for subsequent directory searches.

Call Format:

```
void fl_path_set_pattern(path, pattern)
```

Arguments:

Table 8-105

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*pattern	Wild card pattern in system-specific format	Reference

Returns:

N/A

Remarks:

FL_PATH_SET_PATTERN sets a wild card pattern in the specified portion of the path in normalized form for subsequent directory searches.

See also the other path functions (FL_PATH and related calls).

FL_PATH_SYS

Convert a normalized path into a system-specific path string

Call Format:

```
char *fl_path_sys(path, syspath, length)
```

Arguments:

Table 8-106

Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*syspath	Destination string	Reference
size_t	length	string length	Value

Returns:

- If successful, returns a pointer to a system-specific converted path string.
- If unsuccessful, returns NULL.

Remarks:

FL_PATH_SYS converts a normalized path into a system-specific path string. If the path argument is null, FL_PATH_SYS calls the C function **malloc** to allocate memory for the resulting path. The caller should call **free** to release the memory when it is no longer needed.

Example:

This example opens a specified AC file in the {FLINK}/ac directory.

```
NPATH *np = (NPATH *)NULL;
FILE  *ac_file;
char  *flink;/* Buffer containing FLINK path */
char  *filename;/* Buffer containing AC file name */
np = fl_path_alloc();
if (np == NULL)
```

```
return ERROR;
fl_path_sys(path, flink, sizeof(flink));
np = fl_path_add_dir(np, "ac");
fl_path_set_file(np, filename);
fl_path_set_extension(np, "ac");
fl_path_set_version(np, "");
ac_file = fl_path_fopen(np, "r");
fl_path_free(np);
if (ac_file == NULL)
return ERROR;
```

FL_PATH_TIME

Places formatted system time stamp from a specified file's header into specified buffer.

Call Format:

```
long fl_path_time(path, buf, length)
```

Arguments:

Table 8-107

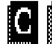
Type	Name	Description	Passed By
NPATH	*path	Pointer to a previously allocated NPATH struct containing a normalized path name buffer	Reference
char	*buf	Buffer to contain returned time stamp	Reference
size_t	length	length of output buffer	value

Returns:

- If successful, returns path's date and time stamp.
- If unsuccessful, returns ERROR.

Remarks:

FL_PATH_TIME formats the time stamp on a file (the time, not including day or date, when the file was last updated) into the caller's buffer. The format of the result from this function is operating-system dependent.



FL_PROC_EXIT

Exit the calling process.

Call Format:

```
int fl_proc_exit(id)
```

Arguments:

Table 8-108

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value

Returns:

GOOD or ERROR

Remarks:

This function renounces all further access to the real-time database.

FL_PROC_INIT

Initialize the FactoryLink calling process. (See also FL_PROC_ INIT_APP.)

Call Format:

```
int fl_proc_init(char *task, char *desc)
```

Arguments:

Table 8-109

Type	Name	Description	Passed By
char	*task	Pointer to process name, 32 chars.max, null-terminated	Reference
char	*desc	Pointer to process description, 80 chars max, null-terminated	Reference

Returns:

If successful at registering the starting-up process with the kernel, function returns the calling routine's FactoryLink task ID.

If unsuccessful, function returns one of the following negative (sign bit on) error codes:

- FLE_NO_{FLINK}_INIT
- FLE_BAD_PROC_NAME
- FLE_ALREADY_ACTIVE
- FLE_NULL_POINTER
- FLE_NO_PROC_INIT
- FLE_PROC_TABLE_FULL

Remarks:

This API has been retained from previous releases of FactoryLink to maintain compatibility for users to upgrade with no changes to existing startup code. However, new tasks should be written to use FL_PROC_INIT_APP to register with the kernel. Any task, even one written for a previous release, may now

override the environment values (i.e., use command arguments), if desired, by calling FL_PROC_INIT_APP. This API now calls the new version.

The current FL_PROC_INIT is now implemented as:

```
int fl_proc_init(char *task, char *desc)
{
char flname[MAX_USR_NAME] ;
char fluser[MAX_USR_NAME] ;
char fldomain[MAX_USR_NAME];
  fl_getvar("FLNAME", flname, sizeof(flname)) ;
  fl_getvar("FLDOMAIN", fldomain, sizeof(fldomain)) ;
  fl_getvar("FLUSER", fluser, sizeof(fluser)) ;
 return fl_proc_init_app(task,desc,flname,fldomain,fluser);
}
```

The calling process must pass a process name and description to FL_PROC_INIT. After validating the name, the FactoryLink kernel returns a small number (in the range **0-30**), called the FactoryLink ID, for use in subsequent kernel calls to identify the caller. The kernel uses FactoryLink IDs to keep track of client processes in much the same way as the operating system uses file handles to keep track of open files.

Multiple threads of a single process can execute separate FL_PROC_INIT_ APPs. In these cases, the kernel regards the threads as different client processes, and assigns distinct IDs, KPROC and KENV entries, change bits, sync bits, wait bits, and so on.

FL_PROC_INIT_APP

Initialize the calling process and register it with the FactoryLink kernel for a specific application/domain. (Replaces FL_PROC_INIT.)

Call Format:

```
fl_proc_init_app(task, desc, {FLNAME}, {FLDOMAIN}, {FLUSER}) ;
```

Arguments:

Table 8-110

Type	Name	Description	Passed By
char	*task	Pointer to process name, 32 chars. max, null-terminated	Reference
char	*desc	Pointer to process description, 80 chars max, null-terminated	Reference
char	*{FLNAME}	Pointer to application invocation name ({FLNAME})	Reference
char	*{FLDOMAIN}	Pointer to domain name ({FLDOMAIN})	Reference
char	*{FLUSER}	Pointer to user name ({FLUSER})	Reference

The **{FLNAME}** argument specifies the name of the invocation with which the task is registering.

The **{FLDOMAIN}** argument specifies the domain for which the task is registering (the domain with which it is to be associated at run time).

The **{FLUSER}** argument specifies which instance of the specified domain the task is registering for.

Returns:

If successful, returns the calling routine's FactoryLink task ID. The task ID contains two fields: the high-order byte contains the instance ID, while the lower byte contains the index into the instance-specific KPROC array.

If unsuccessful at registering the start-up process with the kernel, returns one of the following negative (sign bit on) error codes:

- FLE_NO_{FLINK}_INIT
- FLE_BAD_PROC_NAME
- FLE_ALREADY_ACTIVE
- FLE_NULL_POINTER
- FLE_NO_PROC_INIT
- FLE_PROC_TABLE_FULL

Remarks:

The calling process must pass a process name and description to FL_PROC_INIT_APP. After validating the name, the FactoryLink kernel returns a small number (in the range **0-30**), called the FactoryLink ID, for use in subsequent kernel calls to identify the caller. The kernel uses FactoryLink IDs to keep track of client processes in much the same way as the operating system uses file handles to keep track of open files.

Multiple threads of a single process can execute separate FL_PROC_INIT_ APPs. In these cases, the kernel regards the threads as different client processes, and assigns distinct IDs, KPROC and KENV entries, change bits, sync bits, wait bits, and so on.

This API is the replacement for FL_PROC_INIT. Use this routine wherever FL_PROC_INIT was previously used to register with the kernel for a specific application and/or domain.

FL_PROC_INIT has been retained for compatibility, but in current releases of FactoryLink, it now merely sets up and calls FL_PROC_INIT_APP.

FL_QUERY_MBX

Query a mailbox for a range of queued messages.

Call Format:

```
int fl_query_mbx(id, mbx, mmp, i, n, np)
```

Arguments:

Table 8-111

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG	mbx	Mailbox to be accessed	Reference
MBXMSG FAR	*mmp	Pointer to array of mailbox messages to be filled in by kernel	Reference
uint	i	Index relative to head of queue	Reference
uint	n	Requested number of mailbox messages	Value
uint FAR	*np	Actual number of mailbox messages to be filled in by kernel	Value

Returns:

GOOD or ERROR

- If GOOD, also returns *mmp and *np.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_NO_MESSAGES
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

Use this function to read one or more mailbox messages without reading their message data and without dequeueing them. The MBXMSG holds information about the message such as its type, who sent it, and (especially useful) the length of its data. Therefore, call FL_QUERY_MBX prior to allocating space for the message data and calling FL_READ_MBX. The argument i specifies where, relative to the head of the message queue, reading is to begin (i = **0** means start at the head itself, i = **1** means skip the first message and start with the second, and so on). The message at the head of the queue is the oldest message in the mailbox. The argument n specifies how many mailbox messages (maximum) are requested to be read, and the kernel fills in *np with the actual number that were read.

FL_READ

Read specified elements from the real-time database.

Call Format:

```
int fl_read(id, tp, n, vp)
```

Arguments:

Table 8-112

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are to be read	Reference
uint	n	Number of elements to be read	Value
void FAR	*vp	Pointer to area to receive the values	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

The elements may have mixed data types, as with all database access calls. The values of the elements are read from the real-time database and placed in the private data area of the calling process pointed to by vp. After each value transfer, vp is incremented by the size of the element. After each element is read, its change bit for the calling process is cleared (to **0**). The change state for other client processes are unaffected.

> **Note:** vp is incremented by the actual size of each element. This is important when reading blocks of mixed tag types. Use fl_get_tag_info ( ) to find out how much memory to prepare before calling fl_read when reading blocks of mixed tag types.

FL_READ_MBX

Read and dequeue a message from a mailbox. (Obsolete. See FL_READ_APP_MBX.)

Call Format:

```
int fl_read_mbx(id, mbx, mmp, i)
```

Arguments:

Table 8-113

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG	mbx	Mailbox to be accessed	Reference
MBXMSG FAR	*mmp	Pointer to a single mailbox message to be filled in by kernel	Reference
uint	i	Index relative to head of queue	Reference

Returns:

GOOD or ERROR

- If GOOD, also returns *mmp and message data.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_NO_MESSAGES
  - FLE_ACCESS_DENIED
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

This API has been retained from previous releases of FactoryLink to maintain compatibility for users to upgrade with no changes to existing startup code. However, new tasks should be written to use FL_READ_APP_MBX (with additional capabilities for multiuser systems). FL_READ_MBX can only be used to send messages to tasks in the same domain instance.

Use FL_READ_APP_MBX to read a mailbox by application instance. The additional argument in the current API allows the caller to specify the ID of the owner of the mailbox. The owner's ID is used to determine which instance of the mailbox should be read.

FL_READ_MBX reads a single mailbox message together with its message data. The message is then deleted from the mailbox (that is, it is dequeued). If the buffer provided by the caller is not large enough, message data may be lost. Just as in FL_QUERY_MBX, the argument ***i*** specifies which mailbox message, relative to the head of the message queue, is to be read. However, FL_READ is less flexible, in that it always reads from the head message. Indeed, the following function call:

```
fl_read_mbx(id, &mbx, mmp, 0) ;
```

is equivalent to the function call shown below:

```
fl_read(id, &mbx, 1, mmp) ;
```

FL_READ_APP_MBX

Read and dequeue a message from a mailbox in a specific application instance. (Replaces FL_READ_MBX.)

Call Format:

```
int fl_read_app_mbx(id, rid, mbx, msg, i)
```

Arguments:

Table 8-114

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
id_t	rid	FactoryLink ID of process owning the mailbox	Value
MBX	mbx	Mailbox to be accessed	Reference
MBXMSG	*msg	Pointer to a single mailbox message to be filled in by kernel	Reference
uint	i	Index relative to head of queue	Reference

Returns:

GOOD (on success) or ERROR (on failure)

- If GOOD, also returns *msg and message data.
- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_NO_MESSAGES
  - FLE_ACCESS_DENIED
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

Use FL_READ_APP_MBX to read a single mailbox message together with its message data. After this is done, the message is deleted from the mailbox (that is, it is dequeued). If the buffer provided by the caller is not large enough, message data may be lost.

Just as in FL_QUERY_MBX, the argument ***i*** specifies which mailbox message, relative to the head of the message queue, is to be read. However, FL_READ is less flexible, in that it always reads from the head message. Indeed, the following call:

```
fl_read_mbx(id, &mbx, mmp, 0);
```

is equivalent to the call:

```
fl_read(id, &mbx, 1, mmp);
```

FL_RECV_SIG

Receive a signal for the calling process.

Call Format:

```
int fl_recv_sig(id)
```

Arguments:

Table 8-115

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value

Returns:

The signal (**0-31**) received or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NO_SIGNALS
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

If two or more signals are active and deliverable when FL_RECV_SIG is called, the lowest numbered active signal is delivered to the caller. With the exception of signals **0** and **1**, once a signal has been delivered, it is deactivated (that is, it is no longer present). This means that signals **0** and **1**, once activated, can never be deactivated.

FL_RESET_APP_MEM

Reset application memory for a specific instance of an application by clearing all change bits and setting all data values to zero.

Call Format:

```
int fl_reset_app_mem(id)
```

Arguments:

Table 8-116

Type	Name	Description	Passed By
id_t	id	FactoryLink application task ID	Value

Returns:

GOOD or ERROR

Remarks:

Normally, only the Run-Time Manager should call this function.

FL_RESET_APP_MEM clears all change bits and sets all data values to zero for a specific instance of an application.

FL_SEND_SIG

Send a signal to a target process.

Call Format:

```
int fl_send_sig(id, name, sig)
```

Arguments:

Table 8-117

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
char FAR	*name	Name of FactoryLink process to	Reference
		whom signal is to be sent	
int	sig	Signal (**0**-**31**) to be sent	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_PROC_NAME
  - FLE_NO_PROC_INIT
  - FLE_BAD_ARGUMENT
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

It is legal to send any signal to any active FactoryLink process. If the intended recipient of the signal is asleep in the kernel (waiting to lock or to access the database) and the signal is not being held, the recipient is immediately awakened and returned the error code FLE_SIGNALLED. At this point, the recipient should call FL_RECV_SIG to see which signal was sent. Note that the following function:

```
fl_set_term_flag(id, name);
```

does precisely the same thing as the function shown below:

```
fl_send_sig(id, name, FLC_SIG_TERM_FLAG_SET);
```

FL_SET_CHNG

Set the calling task's change-status flags for specified real-time database elements.

Call Format:

```
int fl_set_chng(id, tp, n)
```

Arguments:

Table 8-118

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying	Reference
		which elements are involved	
uint	n	Number of elements involved	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

Set (to **1**) the change state of the specified real-time database elements for the calling process only. This is useful for establishing initial conditions prior to entering a programming loop, particularly those that use FL_CHANGE_READ or FL_CHANGE_WAIT to read real-time database values.

FL_SET_OWNER_MBX

Set the owner of a mailbox.

Call Format:

```
int fl_set_owner_mbx(id, mbx, onoff)
```

Arguments:

Table 8-119

Type	Name	Description	Passed By
id_t	id	Task ID	Value
TAG	mbx	Mailbox to modify	Reference
uint	onoff	Flag to indicate operation	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_ACCESS_DENIED

Remarks:

Use this function to set the owner of a mailbox. If the value of onoff is **TRUE**, the owner of the mailbox is set to the calling task. If the value of onoff is **FALSE**, the mailbox ownership is removed.

When a write is performed on the mailbox, the owner of a mailbox is signaled with FLC_SIG_MESSAGE_RECEIVED. For additional information about signals, refer to “fl_send_sig” on page 302.

FL_SET_TAG_LIST

Register the tag list (a list of real-time database elements) to a target process.

Call Format:

```
int fl_set_tag_list(id, name, tp, n)
```

Arguments:

Table 8-120

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
char FAR	*name	Name of target process	Reference
TAG FAR	*tp	Pointer to array of trigger elements	Reference
uint	n	Number of trigger elements	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_ARGUMENT
  - FLE_BAD_PROC_NAME
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

FL_SET_TAG_LIST establishes the tag list (list of real-time database elements) for a target process. If a tag list is already present when this call is made, it is replaced with the newly specified tag list. If successful, this call sends the signal FLC_SIG_TAG_LIST_CHANGED to the target process.

FL_SET_TERM_FLAG

Set the termination flag of a client process.

Call Format:

```
int fl_set_term_flag(id, name)
```

Arguments:

Table 8-121

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
char FAR	*name	Pointer to name of process whose termination flag is to be set	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_PROC_NAME
  - FLE_NO_PROC_INIT

Remarks:

Set the termination flag of a client process, possibly different from the calling process. Normally, only the Run-Time Manager uses this service.

FL_SET_WAIT

Set the calling task's wait flags for specified real-time database elements.

Call Format:

```
int fl_set_wait(id, tp, n)
```

Arguments:

Table 8-122

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are involved	Reference
uint	n	Number of elements involved	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG

Remarks:

Set (to **1**) the wait flag of the specified real-time database elements for the calling process only. This function is used in conjunction with FL_WAIT to wait on a set of real-time database elements.

FL_SLEEP

Delay execution of the task for the indicated number of milliseconds.

Call Format:

```
void fl_sleep(msecs)
```

Arguments:

Table 8-123

Type	Name	Description	Passed By
ulong	msecs	Number of milliseconds to delay	Value

Returns:

None

Remarks:

The actual resolution of the delay is system-specific; however, at most, the delay is one second. To determine the actual amount of time delay, use FL_GET_TICK. For information about FL_GET_TICK, refer to “fl_get_tick” on page 245.

FL_TEST_TERM_FLAG

Ask the kernel the current status of current task's termination flag.

Call Format:

```
int fl_test_term_flag(id)
```

Arguments:

Table 8-124

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value

Returns:

OFF, ON, or ERROR

Remarks:

Test the termination flag of the calling process and return its state (**OFF** or **ON**). If the flag is **ON**, another client process (usually the Run-Time Manager) has requested that the caller exit via FL_PROC_EXIT.

FL_UNLOCK

Unlock the real-time database for the calling process.

Call Format:

```
int fl_unlock(id)
```

Arguments:

Table 8-125

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns FLE_NOT_LOCKED.

Remarks:

Calls to FL_LOCK and FL_UNLOCK nest, so one call to FL_UNLOCK undoes one previous call to FL_LOCK.

FL_WAIT

Wait to read, write, or access the real-time database or certain elements in the database.

Call Format:

```
int fl_wait(id, req)
```

Arguments:

Table 8-126

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
int	req	Command; must have one of the following symbolic values: FLC_WAIT_READ, FLC_WAIT_WRITE FLC_WAIT_ACCESS	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_BAD_ARGUMENT
  - FLE_WAIT_FAILED
  - FLE_TERM_FLAG_SET

Note: Ensure the fl_wait() function call is embedded between calls to functions fl_lock() and fl_unlock(), otherwise, the run-time database will remain locked by the task.

FL_WAKEUP

Awaken a mask of FactoryLink client processes.

Call Format:

```
int fl_wakeup(id, mask, req)
```

Arguments:

Table 8-127

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
mask_t FAR	*mask	Pointer to bit mask of client processes to be awakened, updated by kernel to reflect those that were actually asleep and have been awakened	Reference
int	req	Command; must have one of the following symbolic values: FLC_WAIT_READ, FLC_WAIT_WRITE FLC_WAIT_ACCESS	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns FLE_BAD_ARGUMENT.

Remarks:

The calling process must pass a pointer to a bit mask of processes to be awakened. In the mask, bit 0 corresponds to FactoryLink ID number 0, bit 1 to ID 1, and so on through bit 30. The kernel modifies this bit mask to reflect the processes that were actually sleeping at the time the call to FL_WAKEUP occurred and were awakened.

The caller must also pass a symbolic command by setting req to one of the following symbolic commands:

- FLC_WAIT_READ
- FLC_WAIT_WRITE
- FLC_WAIT_ACCESS

Setting req to FLC_WAIT_READ wakes up only those processes waiting to read the database (which are those that have previously done FL_WAIT, directly or indirectly, with req to FLC_WAIT_READ). Similarly, setting req to FLC_WAIT_WRITE wakes up only those waiting to write to the database. Finally, setting req to FLC_WAIT_ACCESS wakes up all of them waiting to do anything to the database (which is all of them sleeping while waiting to do anything except lock the database).

FL_WAKEUP_PROC

Awaken a specified FactoryLink process.

Call Format:

```
int fl_wakeup_proc(id, name, req)
```

Arguments:

Table 8-128

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
char FAR	*name	Name of the task to wake	Reference
int	req	Wake-up request	Value

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns FLE_BAD_ARGUMENT.

Arguments:

FL_WAKEUP_PROC is identical to FL_WAKEUP except that the caller specifies a task name instead of a mask of tasks to wake up. For information about FL_WAKEUP, refer to “fl_wakeup” on page 313.

FL_WRITE

Write specified elements into the real-time database.

Call Format:

```
int fl_write(id, tp, n, vp)
```

Arguments:

Table 8-129

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
TAG FAR	*tp	Pointer to tag array specifying which elements are to be written	Reference
uint	n	Number of elements to be written	Value
void FAR	*vp	Pointer to area holding the values	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_LOCK_FAILED
  - FLE_BAD_TAG
  - FLE_OUT_OF_MEMORY

Remarks:

The elements may have mixed data types. The values of the elements are read from the private data area pointed to by vp and written into the kernel database. After each value transfer, vp is incremented by the size of the element. After each element is written, if its new value is different from its previous value, the change state for all client processes is set to TRUE. If any other client processes are waiting for changes of the element, those processes are awakened. Any attempt by the writing process to write a different value to a synchronous element that is still unread causes the writer to block until it has been read. However, blocking does not occur until an attempt has been made to write all of the specified elements. As a consequence, all asynchronous elements are written on the first pass.

If the tag being written to is a message tag, and the *m_prt* member of the MSG structure is set to NULL, and the *m_max* field is set to a non-zero value, and it is the first write to the tag, then the kernel will allocate space for the message based on the value of *m_max* but will not set the change flag for the tag.

FL_WRITE_MBX

Write and queue a message into a mailbox. (Obsolete. See FL_WRITE_APP_MBX.)

Call Format:

```
int fl_write_mbx(id, mbx, mmp)
```

Arguments:

Table 8-130

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
TAG	mbx	Mailbox to be accessed	Reference
MBXMSG FAR	*mmp	Pointer to a single mailbox message	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_NO_MESSAGES
  - FLE_ACCESS_DENIED
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

This API has been retained from previous releases of FactoryLink to maintain compatibility for users to upgrade with no changes to existing startup code. However, new tasks should be written to use FL_WRITE_APP_MBX (with additional capabilities for multiuser systems). FL_WRITE_MBX can only be used to send messages to tasks in the same domain instance.

Use FL_WRITE_MBX to write a single mailbox message together with its message data. The message being written is added to the queue (that is, it is queued) at the tail of the queue. Prior to making this call, the caller must fill in the MBXMSG and the message data.

Note that within the same domain instance (non-multi-user environments), FL_WRITE, when passed a mailbox TAG, does the same thing as FL_WRITE_MBX. That is, the function call

```
fl_write_mbx(id, mbx, mmp);
```

is equivalent to the function call

```
fl_write(id, &mbx, 1, mmp) ;
```

FL_WRITE_APP_MBX

Write and queue a message into a mailbox. (Replaces FL_WRITE_MBX.)

Call Format:

```
int fl_write_app_mbx(id, wid, mbx, msg);
```

Arguments:

Table 8-131

Type	Name	Description	Passed By
id_t	id	FactoryLink ID of caller	Value
id_t	wid	FactoryLink ID of process owning the mailbox	Value
TAG	mbx	Mailbox to be accessed	Reference
MBXMSG FAR	*msg	Pointer to a single mailbox message to be filled in by kernel	Reference

Returns:

GOOD or ERROR

- If ERROR, call the FL_ERRNO function with the caller's FactoryLink ID, and it returns one of the following errors:
  - FLE_NULL_POINTER
  - FLE_BAD_TAG
  - FLE_NOT_MAILBOX
  - FLE_NO_MESSAGES
  - FLE_ACCESS_DENIED
  - FLE_OUT_OF_MEMORY
  - FLE_LOCK_FAILED
  - FLE_LOCK_EXPIRED

Remarks:

Use FL_WRITE_APP_MBX to write, to a mailbox by application instance, a single mailbox message together with its message data. The message being written is added to the queue (that is, it is queued) at the tail of the queue. Prior to making this call, the caller must fill in the MBXMSG and the message data. The additional arguments in the current API allow the caller to specify the ID of the owner of the mailbox and an index into the queue. The owner's ID is used to determine which instance of the mailbox to write this message into.

Note that within the same domain instance (non-multi-user environments), FL_WRITE, when passed a mailbox TAG, does the same thing as FL_WRITE_APP_MBX. That is, the function call

```
fl_write_app_mbx(id, mbx, mmp);
```

is equivalent to the function call

```
fl_write(id, &mbx, 1, mmp);
```

FL_XLATE

Translate a key to its associated message.

Call Format:

```
char FAR *fl_xlate (key);
```

Arguments:

Table 8-132

Type	Name	Description	Passed By
char FAR	*key	Pointer to key for which to search	Reference

Returns:

Pointer to message if key found; equal to keyptr if key not found

Remarks:

This function, along with FL_XLATE_INIT, comprises the message-translation facility. These two functions allow run-time tasks to print messages stored in external disk files called “message files.” For details about initialization of the message-translation facility, refer to “fl_xlate_init” on page 324.

The following examples illustrate the use of the message-translation facility:

Example 1

Allocate a message array, such as message[], and call the translation function to access messages in the message file and put the result in message[]. In the following example, the function copies the string **normal shutdown** (the message associated with the key SHUTDOWN) into message[]**.**

```
char message[100];
strcpy(message, fl_xlate(“SHUTDOWN”));
```

Example 2

The following function puts the string **Run Time Manager: errno = 3** into message**[]**.

```
sprintf(message, fl_xlate ("errno"), 3);
```

The errno key translates to "Run Time Manager: errorno = %d," and in this example, %d has the value of 3.

Example 3

The following function prints the message string **Cannot open file SYS.CT** on the screen:

```
printf(fl_xlate("cantopen"), "SYS.CT");
```

The cantopen key translates to "Cannot open file %s," and in this example, %s has the value of SYS.CT.

FL_XLATE_INIT

Message translation initialization; initializes a message file and establishes a buffer for it (use of the buffer enhances performance).

Call Format:

```
int fl_xlate_init (name, bp, blen);
```

Arguments:

Table 8-133

Type	Name	Description	Passed By
char FAR	*name	Name of message file	Reference
char FAR	*bp	Pointer to buffer to be used to store keys and their associated messages	Reference
uint	blen	Size of buffer, in bytes	Value

Returns:

Number of messages read and buffered, or ERROR

Remarks:

This function, along with FL_XLATE, comprises the basis of the FactoryLink message-translation facility. These two functions allow run-time tasks to print messages stored in external disk files called "message files." For details about message translation, refer to "fl_xlate" on page 322.

A task that uses the message translation facility can be written so that it is independent of the messages that it prints; it is dependent on the mnemonic keys, but not the messages. In particular, it may be written so that it is independent of the language in which the messages are written. Achievement of this independence is the main purpose of message translation.

Message files are ASCII text files that can be edited and changed with an ordinary text editor. On a FactoryLink system, the current working copies of these files are in the directory /**{FLINK}**/MSG, and all are assumed to have the extension .TXT. Message files must conform to the following rules:

- The message translation functions assume that lines beginning with an asterisk (*) are comments and ignores these lines. They also ignore blank lines.
- Non-comment, non-blank lines serve to translate a key to an associated message. They must have the following form:

```
KEY white space MESSAGE
```

- The message may be enclosed within double quote marks (") for clarity and to avoid the stripping of leading and trailing white space by the translation function. It may contain %, the substitution character used by PRINTF and SPRINTF.
- A vertical bar (|) is allowed as a separator within the white space, for compatibility with the format of FactoryLink .KEY files.
- When editing a message file, place the most commonly used messages near the beginning of the file, so that access to the commonly used messages is quicker than to other messages, particularly when the buffer in use is small.

To use the message-translation functions, a task first sets up a buffer, such as buffer[], and then calls the initialization function, which opens the message file /**{FLINK}**/MSG/RUNMGR.TXT, as shown in the following example:

```
char buffer[500];

fl_xlate_init("RUNMGR", buffer, sizeof(buffer));
```

Refusal to designate a buffer is also legal:

```
fl_xlate_init("RUNMGR", NULL, 0);
```

This function provides unbuffered access to /**{FLINK}**/MSG/RUNMGR.TXT. When doing unbuffered access, messages are read into a static buffer in the Library function. Subsequent reads overwrite this buffer, so the contents must be used before the next access to the file.

/**{FLINK}**/MSG/RUNMGR.TXT provides an example of a message file.

See also the related message translation functions fl_xlate, fl_xlate_set_tree, and fl_xlate_get_tree.

FL_XLATE_LOAD

Load the specified file (passed as a parameter) into the current translation tree, replacing any duplicate keys.

Call Format:

```
int fl_xlate_load (fname);
```

Arguments:

Table 8-134

Type	Name	Description	Passed By
char FAR	*fname	File name from which to load tree data	Reference

Returns:

If successful, returns the number of entries loaded from this file

If unsuccessful, returns ERROR

Remarks:

This function adds to the functionality of the kernel's message-translation facility (FL_XLATE). When a new translation file is loaded using this function, duplicate keys are overridden.

In order for a new tree to be loaded, an alternate tree to the default tree should have been created previously using the fl_xlate_init function. The new tree will become the default into which key files are loaded.

Tasks may maintain more than one translation tree. Translation files may be kept in libraries, one per language used, for ease of use. This guarantees that all tasks remain language-independent, and allows run-time tasks to use the fl_xlate functions for all message output.

Example 1

```
int num_msg;
num_msg = fl_xlate_init("runmgr", NULL, 0);
if (num_msg == 0)
```

```
return ERROR;
num_msg = fl_xlate_load("iml");
if (num_msg == 0)
return ERROR;
printf("%s\n", fl_xlate("token"));

 * TSPRINTF: Format a string into a buffer. This is a "tiny" sprintf()
 * that handles %s, %c, %d, %u, %x, %X, %o, %ld, %lu, %lx, %lX, %lo, and
* %%. It recognizes width and pad specifications (%3d, %03d, etc.), long
 * values, and left/right justification. It does not understand floating
 * point formats (%e, %f, %g) and minimum width.
```

When loading translation files, the environment variable FLLANG is examined. If it is defined to be anything besides "C", it will be appended to the /{FLINK}/MSG path used to load the master file, as well as user-defined files.

If the user-specified file contains path information, it will be folded into the /{FLINK}/MSG{/FLLANG} path when loading the file.

The function **fl_xlate_init** will start a fresh translation tree and load the default translation file **master.txt**. The user-specified translation file is then loaded on top of the existing definitions. Duplicate definitions are superseded by the last file loaded. The function returns the total number of translations loaded from both the MASTER.TXT file as well as the user-specified file, or it returns **ERROR** if there has been an error.

The **buff** and **len** parameters of **fl_xlate_init** are not used by the function, but were retained to stay compatible with existing code.

The function **fl_xlate_load** will load the specified file into the current translation tree, replacing any duplicate keys. The function returns the number of entries loaded from this file or returns **ERROR**.

Example 2

If the FLLANG environment variable is not defined, the following call:

```
num_msg = fl_xlate_init("iml", NULL, 0)
```

will load the **/{FLINK}/MSG/master.txt** file into the tree and then load the **/{FLINK}/MSG/iml.txt** file into the tree. The return value is the total number of translations loaded from both files (duplicates are only counted once.)

The call

```
num_msg = fl_xlate_load("iml")
```

will load the file **/{FLINK}/MSG/iml.txt** into the tree and return the number of translations.

The call

```
num_msg = fl_xlate_load("/temp/test")
```

will load the file **/TEMP/test.txt** into the tree and return the number of translations.

Example 3

If the FLLANG environment variable is defined to be "german", the following call:

```
num_msg = fl_xlate_init("iml", NULL, 0)
```

will load the **/{FLINK}/MSG/GERMAN/master.txt** file into the tree and then load the **/{FLINK}/MSG/GERMAN/iml.txt** file into the tree. The return value is the total number of translations loaded from both files (duplicates are counted only once.)

The call

```
num_msg = fl_xlate_load("iml")
```

will now load the file **/{FLINK}/MSG/GERMAN/iml.txt** into the tree and return the number of translations.

The call

```
num_msg = fl_xlate_load("/temp/test")
```

will still load the file **/TEMP/test.txt** into the tree and return the number of translations.

See also the related message translation functions fl_xlate, fl_xlate_set_tree, and fl_xlate_get_tree.

FL_XLATE_GET_TREE

Returns the address of the current translation tree or the string **NULL** if no translation files have been loaded.

Call Format:

```
void FAR *fl_xlate_get_tree( void ) ;
```

Returns:

Address of the current translation tree; returns **NULL** if no translation files have been loaded

Remarks:

This function adds to the functionality of the kernel's message-translation facility (FL_XLATE.) When a program is maintaining multiple translation trees, the address of the current translation tree (default translations for FactoryLink tasks) is returned by calling this function.

Tasks may maintain more than one translation tree. Translation files may be kept in libraries, one per language used, for ease of use. This guarantees that all tasks remain language-independent, and allows run-time tasks to use the fl_xlate functions for all message output.

Example 1

```
void *english, *french; /* pointers to 2 trees */
int  num_msg1, num_msg2;
num_msg1 = fl_xlate_init("english/iml", NULL, 0);
english = fl_xlate_get_tree();
num_msg2 = fl_xlate_init("french/iml", NULL, 0);
french = fl_xlate_get_tree();
fl_xlate_set_tree (english); /* Switch to English tree */
printf ("%s\n", fl_xlate("token");
fl_xlate_set_tree (french); /* Switch to French tree */
printf ("%s\n", fl_xlate("token");
```

You may switch translation trees at any point in the program, and switch back when ready, without losing any data.

See also the related message translation functions fl_xlate, fl_xlate_init, fl_xlate_set_tree, and fl_xlate_load.

FL_XLATE_SET_PROGPATH

Overrides the environment variable **{FLINK}**, allowing programs to support the -p command parameter for overriding the default program directory.

Call Format:

```
int fl_xlate_set_progpath(progname);
```

Arguments:

Table 8-135

Type	Name	Description	Passed by
char FAR	*progname	Pointer to path of program directory to be used for translation	Reference

Returns:

The number of characters in the program path name.

FL_XLATE_SET_TREE

Sets the current translation tree to the tree at the specified address. Ensuing file loads and translations will be done using this tree.

Call Format:

```
void FAR *fl_xlate_set_tree( void *p ) ;
```

Arguments:

Table 8-136

Type	Name	Description	Passed by
void FAR	*tree	pointer to tree to be used for translation	Reference

Returns:

If successful, returns the pointer sent by the programmer

If no tree exists at the specified address, returns **NULL**

Remarks:

This function adds to the functionality of the kernel's message-translation facility (FL_XLATE.) When a program is maintaining multiple translation trees, this function allows changing the current translation tree (default translations for FactoryLink tasks.)

This function may also be used to start a fresh translation tree file.

Tasks may maintain more than one translation tree. Translation files may be kept in libraries, one per language used, for ease of use. This guarantees that all tasks remain language-independent, and allows run-time tasks to use the fl_xlate functions for all message output.

Example 1

```
void *english, *french; /* pointers to 2 trees */

int  num_msg1, num_msg2;

num_msg1 = fl_xlate_init("english/iml", NULL, 0);
```

```
english = fl_xlate_get_tree();
num_msg2 = fl_xlate_init("french/iml", NULL, 0);
french = fl_xlate_get_tree();
fl_xlate_set_tree (english); /* Switch to English tree */
printf ("%s\n", fl_xlate("token");
fl_xlate_set_tree (french); /* Switch to French tree */
printf ("%s\n", fl_xlate("token");
```

You may switch translation trees at any point in the program, and switch back when ready, without losing any data.

fl_xlate_set_tree may also be used to start a fresh tree, as in the following example.

Example 5.

```
fl_xlate_set_tree(NULL) ; /* start new tree */
```

See also the related message translation functions fl_xlate, fl_xlate_init, fl_xlate_get_tree, and fl_xlate_load.

MAKE_FULL_PATH

Combine the directory and file name into a full path name. (See also FL_PATH and related functions.)

Call Format:

```
void make_full_path(fpathp, dpathp, rpathp)
```

Arguments:

Table 8-137

Type	Name	Description	Passed By
char FAR	*fpathp	Buffer where full path is returned	Reference
char FAR	*dpathp	Base directory	Reference
char FAR	*rpathp	Name of file to be added to the directory	Reference

Returns:

No values

Remarks:

The name of the file to be added to the directory may contain a relative path.

To maintain backward compatibility with previous releases, MAKE_FULL_PATH has been retained in FLIB, but it is currently implemented using the fl_path functions. For new development, the fl_path functions should be used. Refer to Chapter 2, “FactoryLink Architecture” for the names of the functions, and to the definitions of those functions in this chapter.

```
void make_full_path(pathp, dirp, filep)

{

NPATH *p1;

NPATH *p2;
```

```
p1 = fl_path_alloc();
p2 = fl_path_alloc();
if ( dirp == NULL )
fl_path_cwd(p1);
else
fl_path_set_dir(p1, dirp);
fl_path_norm(p2, filep);
fl_path_add(p1, p2);
fl_path_sys(p1, pathp, MAX_PATH);
fl_path_free(p1);
fl_path_free(p2);
}
```

SPOOL

Spool a file or line.

Call Format:

```
int spool (id, flags, message, length)
```

Arguments:

Table 8-138

Type	Name	Description	Passed By
id_t	id	Caller's FactoryLink ID	Value
char FAR	*flags	Pointer to a zero-terminated string. The string specifies what is to be printed (a file or a line), what output device is to be used (Devices 1 through 5), what type of data (text or binary) is to be expected, and whether to delete the file (assuming that a file, not a line, is specified) after successful printing. The following recognized characters determine these actions and may or may not appear in the string: **Char.** **Action** **B** Use binary mode in reading and printing the file or the line. Send the Binary ON command sequence before beginning and the Binary OFF sequence upon completion of the print job. **D** Delete the file after successful printing. **L** Print the line that follows (“message” specifies a line to be printed). **#** Use output Device number # (where # stands for a digit from 1-5, the default being 1).	Reference
char FAR	*message	Pointer to a character string.	Reference

Note: Do not specify both the L and D flags in the same job request. Any other combination is legal. The order of characters in the flags string is immaterial.

Table 8-139

Type	Name	Description		Passed By
		If the flags argument is:	**Then the message argument is:**	
		L	Line to be printed	
		B, **D**, or #	Path name of file to be printed	
int	length	Length, in bytes, of the message Value string. Used if both the L and B flags are specified, which means that the message string is not necessarily zero-terminated. (It may contain any ASCII characters, including 00 hex.) string.		Value

Returns:

A signed integer (an int) that indicates the status of the job request. A value of zero indicates that the request has been accepted and queued by the SPOOL task. Any non-zero value indicates that some sort of error has occurred. If the return value is negative, the request was not received by the SPOOL task. If it is positive, it was received by SPOOL but could not be processed. Specifically, the return values have the following meanings:

Table 8-140

Return	Value	Meaning
-3		The request was too long (the flags and the message strings combined exceed 128 bytes; a program error is the likely cause).
-2		The request was not sent (caused by another task repeatedly tying up the channel to the SPOOL task). This can only be caused by a program error in one of the tasks running on the system. The task waits a few seconds (at most) before retrying the request. Should subsequent retry attempts fail, the calling task should print an informative error message, such as **Print Spooler is temporarily unavailable**, and take appropriate action.

Table 8-140

Return	Value	Meaning
-1		The request was sent to the SPOOL task, but no reply was received, the likely cause of which is that the SPOOL task is not running. The task prints an error message, such as **Print Spooler not running**, and either quits or finds some alternate way to output its data.
0		The request was accepted and queued by the SPOOL task (all is OK).
1		The request had a bad flags argument and was therefore rejected, which could be caused by either a non-existent output device or a program error. If the output device does not exist, there is no entry in the Print Spooler Configuration table for the given device number.
2		The request could not be processed because the spool queue is full; the requesting task may wish to try again later.

Remarks:

The SPOOL function is related to the FactoryLink Print Spooler task.

FactoryLink Print Spooler Task

The FactoryLink Print Spooler is a FactoryLink task called SPOOL that processes job requests from other FactoryLink tasks running on the system. These job requests specify either a file to be printed on a printer or other output device or file, or a line to be output. For details about the FactoryLink Print Spooler task, refer to “Print Spooler” in *the Core Tasks Configuration Guide*.

The task initiating the job request must be an integral part of the request itself or specify what file or line is to be printed, which output device is to be used, and what type of data (text or binary) printed.

When the SPOOL task receives a job request, it checks to see whether the designated output device is busy (presumably processing requests received earlier). If the output device is busy, it queues, or spools, the current request. Otherwise, it processes the request immediately. Requests that are queued for

later processing are handled strictly on a first-come, first-served basis (there is no prioritization). In effect, printing occurs on all output devices simultaneously.

Examples

The following examples in C illustrate how to use the SPOOL function:

Example 1:

```
int spool(id, “2", ”C:/CONFIG.SYS", 0);
```

Meaning: Print the indicated text file on Device 2.

Example 2:

```
int spool(id, “L”, “** WARNING: Line pressure LOW. *”, 0);
```

Meaning: Print the indicated line on Device 1 (the default printer).

Example 3:

```
int spool(id, “B3", ”C:/{FLINK}/USR/TASK.DAT", 0);
```

Meaning: Print the indicated binary data file on Device 3.

Example 4:

```
int spool(id, “D”, “C:/SOURCE/TEST.LOG”, 0);
```

Meaning: Print the indicated text file on Device 1 and delete the file afterwards.

In all of these examples, the value of **int** should be checked. The only possible return values in these examples are **-2** through **2**.

TSPRINTF

Create a target string according to a format string using the given argument values (...).

Operating System Notes

The following section contains operating system specific notes relevant to the API Reference Guide.

For OS/2 Users

fl_path_get_type (page 268)

FL_PATH_GET_TYPE returns the file type of the file specified. One of the following constants is returned.

- NPATH_REGULAR
  NPATH_DIRECTORY

tsprintf (page 341)

Create a target string according to a format string using the given argument values.

Call Format:

```
int tsprintf (bp, fp, ...);
```

Arguments:

Table 8-141

Type	Name	Description	Passed By
char	*bp	Pointer to target string in which to put result	Reference
char	*fp	Pointer to format string to be used	Reference
	...	Argument values. This function operates in much the same way as its namesake, SPRINTF, except that it recognizes only the following substitution subset: **Value** **Description** **% c** Character %s String, zero-terminated %d Signed decimal int %u Unsigned decimal int %o Unsigned octal int %x Unsigned hexadecimal int using A-F %X Unsigned hexadecimal int using A-F %ld Signed decimal long	

Returns:

Length of resulting target string, not including the zero terminator. This function handles width and padding. For example, **%3d** prints a decimal value in a field of width at least 3, and **%03d** pads the field with zeros instead of blanks.

Remarks:

Use this function only for OS/2 multithreaded programs.

For UNIX Users

This chapter provides details about the FactoryLink functions.

This reminder is repeated from the architecture chapter for the convenience of those using only the reference guide.

Although colons are valid characters in file names under most UNIX installations, FactoryLink PAK modules should not use file names that include colons. Due to the multi-platform nature of FactoryLink and the need for portability, colons (":") in file names cause the FactoryLink system to interpret the portion of the name preceding the colon as a device name (currently ignored under UNIX); for example, "/tmp/ava:balt" will be seen as "/tmp/balt". The system will always assume that anything preceding a ":" in a file name is a device name, and will skip it. Therefore, do not place colons in file names.

Note specifically that in calls to **make_full_path()**, **slash_to_norm()**, and **fl_path_norm()** the returned path and file names will not be as expected if passed a file name containing a colon; they work as expected when file names without colons are passed in.

fl_sleep (page 309)

When using the fl_sleep function under SCO UNIX, you must link in libx.a using the following command line argument:

```
-lx
```

> **Note:** If you do not link libx.a when using the fl_sleep function, the nap() function will be unresolved.

fl_path_get_type (page 268)

FL_PATH_GET_TYPE returns the file type of the file specified. One of the following constants is returned.

- NPATH_REGULAR
  NPATH_DIRECTORY
  NPATH_FIFO (Unix only)
  NPATH_DEVICE (Unix only)

tsprintf (page 341)

Create a target string according to a format string using the given argument values.

Remarks:

Do not use this command in the UNIX environment. Instead, use the C-language SPRINTF. For details, refer to the appropriate C-language documentation.

For Windows/NT Users

fl_path_get_type (page 268)

FL_PATH_GET_TYPE returns the file type of the file specified. One of the following constants is returned:

- NPATH_REGULAR
  NPATH_DIRECTORY

tsprintf (page 341)

To create a target string according to a format string, use the following argument values:

Call Format:

```
int tsprintf (bp, fp, ...);
```

Arguments:

Table 8-142

Type	Name	Description	Passed By
char	*bp	Pointer to target string in which to put result	Reference
char	*fp	Pointer to format string to be used	Reference
	...	Argument values. This function operates similarly to, SPRINTF, except that it recognizes only the following substitution subset: **Value** **Description** %c Character %s String, zero-terminated **%d** Signed decimal int **%u** Unsigned decimal int **%o** Unsigned octal int **%x** Unsigned hexadecimal int using A-F **%X** Unsigned hexadecimal int using A-F **%ld**Signed decimal long	

Returns:

Length of resulting target string, not including the zero terminator. This function handles width and padding. For example, **%3d** prints a decimal value in a field width of at least 3, and **%03d** pads the field with zeros instead of blanks.

Chapter 9 . . . .

Normalized Tag References

Normalization alludes to the deconstruction of a complex entity into its base components. Once an entity has been decomposed, its base components become exposed and can be manipulated individually. Eventually, these base components may be recombined to reform the complex entity.

Tag references equate to complex strings comprised of one or more base components. The base components for a tag reference are: node, name, dimension, and member. Normalization of tag references encompasses the services that:

- Decompose a tag reference into its base components.
- Allow the base components to be obtained and modified.
- Reconstruct the base components back into a tag reference.

This section describes the FactoryLink PAK interface for the normalization of tag references and covers writing a program incorporating the FactoryLink Normalized Tag (FLNTAG) API. Topics include:

- Normalized Tag Reference Overview
- Normalized Tag Reference API Guide

The order of these topics reflects a top-down approach.

Normalized Tag Reference Overview

A FactoryLink object is a named instantiation of a data type of digital, analog, float, long analog, message, or mailbox. The object may be a single instance of its data type or it may be an array containing multiple instances of the same data type. The name of a FactoryLink object is referred to as the object name.

To connote a structured view, a FactoryLink object may be said to be a member of another object. A member FactoryLink object has equal standing with its parent object, save that it cannot have members of its own. A member tag can be arrayed, but these usually parallel the dimensions of its parent object.

Finally, FactoryLink object may have a local or remote value source. This source is known as its node. This node is sometimes referred to as an external domain.

A tag refers a single location for within the FactoryLink real-time database. A FactoryLink object equates to one or more tags, depending on whether it is an arrayed object and/or whether it has member objects associated with it.

Given these parameters, a reference to a FactoryLink object conforms to the following syntax:

```
[{node}:]{name}[dimension][...][.member]
```

where:

node	(optional) The source node for the given tag
name	The base name for the tag
dimension	(optional) The particular tag element for an arrayed tag
member	(optional) The sub-component identifier for the base tag

For example, the object reference “plc1:tagx[4][5].raw” has a node of “plc1”, an id of “tagx”, the dimensions of “[4][5]”, and a member of “raw”.

The above syntax provides the precision required to resolve an object reference to a single, real-time database location (tag, for short).

Also given this syntax, the combination of a reference node, name, and member equates to the name of the object, as seen through FLCM object list panel. Please keep in mind that the object name component may not be sufficient to uniquely identify a tag. It must be accompanied by its associated source, dimension, or member attributes.

Overview of FLNTAG Services

Obviously, the syntax for a tag reference becomes complex when associated attributes are prefixed and/or appended onto it. Even a simple reference, such as x[5], requires a significant amount of parsing and interpretation.

The FLNTAG API consolidates this processing into a single interface. References can be decomposed into the their base components, have these component modified individually, and ultimately recombined into a reference string. The API also provides hooks for obtaining object definitions and RTDB locations.

The services provided by the FLNTAG API are:

- Creation and destruction of an FLNTAG instance.
- Decomposition of a tag reference string into a FLNTAG.
- Object definition and RTDB location retrieval based on the FLNTAG.
- Component level manipulation of a FLNTAG.
- Generation of a string (reference, object name, ...) from a FLNTAG.

Overview of the FLNTAG API

The FLNTAG API is a data-abstracted set of functions. Its operation generally follows some for variant of the following sequence:

1. Open the application's object table (table containing all known objects),
2. Create an FLNTAG instance.
3. Load a tag reference string into the FLNTAG.
4. Obtain the RTDB location for the reference, using the FLNTAG API.
5. Destroy the FLNTAG instance
6. Close the object table.

Hence, the reference's location within the RTDB can be determined in a hands-off manner. Additionally, should the syntax for a reference change, code based on the API need not change to support it.

Code Scrap: Converting a Tag Reference to a TAG

```
#include <flntag.h>

/*
 *  Function convert_ref2tag() returns the RTDB location for the
 *  given reference.
 */
int convert_ref2tag(char *flapp, char *tagref, TAG *tag)
{
FLNTAG  *ntag;
CT      objct;
int     rv;

      if (ct_open_obj(&objct, flapp) != GOOD)
          return ERROR

      if ((ntag = flntag_create()) != NULL)
      {
          if ((rv = flntag_parse_ref(ntag, tagref)) ==
FLNTAG_E_GOOD)
              rv = flntag_find_tag(ntag, &objct, tag);

          rv = (rv == FLNTAG_E_GOOD) ? GOOD : ERROR;
          flntag_destroy(ntag);
      }
      else
      {
```

```
            rv = ERROR;
        }

        ct_close_obj(&objct);
        return rv;
}
```

FLNTAG Return Codes

The following chart lists the return codes defined in FLNTAG.H. The meaning of these codes are described in the description of the functions that may return it.

Return Code	Value
FLNTAG_E_GOOD	0
FLNTAG_E_GENERAL	-1
FLNTAG_E_HANDLE	-2
FLNTAG_E_INVARG	-3
FLNTAG_E_NOMEM	-4
FLNTAG_E_REFSYN	-5
FLNTAG_E_INVCHR	-6
FLNTAG_E_LENGTH	-7
FLNTAG_E_DIMSYN	-8
FLNTAG_E_DIMLEN	-9
FLNTAG_E_DIMNUM	-10
FLNTAG_E_DIMRNG	-11
FLNTAG_E_MBRREF	-12

Normalized Tag Reference API Guide

Presented in alphabetical order, the following pages describe the Normalized Tag API.

flntag_calc_base-

flntag_calc_tag PAGEREF

flntag_create PAGEREF

flntag_destroy PAGEREF

flntag_find_def PAGEREF

flntag_find_tag PAGEREF

flntag_gen_objname PAGEREF

flntag_gen_ref PAGEREF

flntag_gen_str PAGEREF

flntag_get_dimen PAGEREF

flntag_get_member PAGEREF

flntag_get_name PAGEREF

flntag_get_node PAGEREF

flntag_parse_brkt4dims PAGEREF

flntag_parse_comma4dims PAGEREF

flntag_parse_ref PAGEREF

flntag_set_dimen PAGEREF

flntag_set_member PAGEREF

flntag_set_name PAGEREF

flntag_set_node PAGEREF

-

FLNTAG_CALC_BASE

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_calc_base(FLNTAG *ntag, TAG *tag,
                     char *def_dims, TAG *base)
```

Arguments:

FLNTAG*	ntag	(i)	FLNTAG for which to obtain definition.
TAG*	tag	(i)	The RTDB location (tag) for the reference contained within the given *ntag*.
char*	def_dims	(i)	The dimensions defined for the given object.
TAG*	base	(o)	The base RTDB location (tag) for the object referenced by *ntag*.

Returns:

FLNTAG_E_GOOD	Definition found and RTDB location returned.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_DIMNUM	The number of dimensions in the *ntag*'s reference differs from the number of dimensions specified by the object's definition.
FLNTAG_E_DIMSIZ	The size of the dimensions in the *ntag*'s reference exceeds from the maximum dimension sizes specified in the object's definition.

Description: Given a normalized tag reference, its RTDB location, and its dimension definition, function *flntag_calc_base()* calculates the RTDB location for the base of the given *ntag*. For nonarrayed objects, the location of a tag reference always equals the base tag location obtained from the object's definition.

This function is commonly called during the loading of a task's CT. CT's often contain the reference string, the RTDB location for the reference string, and the dimension definition for the object being referenced. This function uses these three pieces of information to obtain the base location for reference.

See Also: flntag_create(), flntag_find_def().

FLNTAG_CALC_TAG

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_calc_tag(FLNTAG *ntag, TAG *base,
                    char *def_dims, TAG *tag)
```

Arguments :

FLNTAG*	ntag	(i)	FLNTAG for which to obtain definition.
TAG*	base	(i)	The RTDB location (tag) for the base of the reference contained within the given *ntag*.
char*	def_dim s	(i)	The dimensions defined for the given object.
TAG*	tag	(o)	RTDB location (tag) for the object referenced by *ntag*.

Returns :

FLNTAG_E_GOOD	Definition found and RTDB location returned.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_DIMNUM	The number of dimensions in the *ntag*'s reference differs from the number of dimensions specified by the object's definition.
FLNTAG_E_DIMSIZ	The size of the dimensions in the *ntag*'s reference exceeds from the maximum dimension sizes specified in the object's definition.

Description: Given a normalized tag reference, its base RTDB location, and its dimension definition, function *flntag_calc_tag()* calculates the RTDB location for the given *ntag*. For nonarrayed objects, the location of a tag reference always equals the base tag location obtained from the object's definition.

Code Scrap: Converting a FLNTAG to a TAG

```
#include <flntag.h>

/*
 *  Function convert_ntag2tag() returns the RTDB location for
the
 *  given reference.
 */
int convert_ntag2tag(FLNTAG *ntag, CT *objct, TAG *tag)
{
FLOBJREC def
int      rv;

      if ((rv = flntag_find_def(ntag, objct, &def) ==
FLNTAG_E_GOOD)
      {
          rv = flntag_calc_tag(ntag,
                              flobjrec_get_tag(&def),
                              flobjrec_get_dimen(&def),
                              tag);
      }
      return rv;
}
```

See Also: **flntag_create(), flntag_find_def().**

FLNTAG_CREATE

Normalized Tag API.

Prototype:

```
#include <flntag.h>
FLNTAG* flntag_create(void)
```

Returns :

FLNTAG*	Normalized tag instance handle.
NULL	Memory allocation failure.

Description: Function *flntag_create()* allocates a normalized tag instance. This handle is subsequently passed to all other *flntag_...()* functions.

See Also: flntag_destroy().

FLNTAG_DESTROY

Normalized Tag API.

Prototype:

```
#include <flntag.h>
FLNTAG* flntag_destroy(FLNTAG *ntag)
```

Arguments:

FLNTAG*	ntag	(i/o)	Handle to release.

Description: Function *flntag_destroy()* releases all resources associated with the given handle. This handle should not be referenced after being destroyed.

See Also: flntag_create().

FLNTAG_FIND_DEF

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_find_def(FLNTAG *ntag, CT *objct, FLOBJREC *rec)
```

Arguments:

FLNTAG*	ntag	(i)	FLNTAG for which to obtain definition.
CT*	objct	(i)	Object CT handle.
FLOBJREC*	rec	(o)	Definition for the object referenced by *ntag*.

Returns :

FLNTAG_E_GOOD	Definition found and returned.
FLNTAG_E_GENERAL	Tag definition not found.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.

Description: Function *flntag_find_def()* returns the definition for the object referenced by the given *ntag*. This definition is obtained by searching the application's object CT.

A required argument for this function, parameter *objct* can be obtained via function *ct_open_obj().*

See Also: flntag_create(), flntag_parse_ref(), ct_open_obj().

FLNTAG_FIND_TAG

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_find_tag(FLNTAG *ntag, CT *objct, TAG *tag)
```

Arguments:

FLNTAG*	ntag	(i)	FLNTAG for which to obtain RTDB location.
CT*	objct	(i)	Object CT handle.
TAG*	tag	(o)	RTDB location (tag) for the object referenced by *ntag*.

Returns:

FLNTAG_E_GOOD	Definition found and RTDB location returned.
FLNTAG_E_GENERAL	Tag definition not found.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_DIMNUM	The number of dimensions in the *ntag*'s reference differs from the number of dimensions specified by the object's definition.
FLNTAG_E_DIMSIZ	The size of the dimensions in the *ntag*'s reference exceeds from the maximum dimension sizes specified in the object's definition.

Description: Function *flntag_find_tag()* returns the RTDB location (the tag) for the object referenced by the given *ntag*. This location is obtained by searching the application's object CT for the object definition, obtaining the base RTDB location from that definition, and. finally, calculating the offset from the base if the tag is an arrayed element.

For example, if the *ntag* equates to reference **x[3]**, function *flntag_find_tag()* returns RTDB location for **x[3]**, not **x[0]**.

A required argument for this function, parameter *objct* can be obtained via function *ct_open_obj().*

See Also: **flntag_create(), flntag_calc_tag(), flntag_parse_ref(), ct_open_obj().**

FLNTAG_GEN_OBJNAME
FLNTAG_GEN_REF
FLNTAG_GEN_STR

Normalized Tag API.

Prototype:

```
#include <flntag.h>

int flntag_gen_str(FLNTAG *ntag, u16 excl,
                   char *outstr, int maxlen)

#define flntag_gen_objname(ntag, outstr, maxlen) \
        flntag_gen_str(ntag, FLNTAG_S_DIMEN, outstr,
maxlen)

#define flntag_gen_ref(ntag, tagref, maxlen) \
        flntag_gen_str(ntag, 0, tagref, maxlen)
```

Arguments:

FLNTAG*	ntag	(i)	FLNTAG for which to generate string.
u16	incl	(i)	Bit-wise OR-flags for the components to include within the generated string. FLNTAG_S_NODE FLNTAG_S_NAME FLNTAG_S_DIMEN FLNTAG_S_MEMBER
char*	outstr	(o)	Target buffer for the generated string. If NULL, then the string is not generated.
int	maxlen	(i)	Maximum number of characters to write to the *outstr* buffer.

Returns :

rtn >= 0	Full length of the requested string.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.

Description: Function *flntag_gen_str()* generates the reference string to which the given *ntag* equates. When nonzero, parameter *incl*, a bit mask, includes one or more components from the resulting string.

Normally, the entire reference is desired, and the macro *flntag_gen_ref()* can be used to build it.

For cases where the object name is needed, macro *flntag_gen_objname()* can be used.

See Also: flntag_create(), flntag_parse_ref().

FLNTAG_GET_DIMEN
FLNTAG_GET_MEMBER
FLNTAG_GET_NAME1
FLNTAG_GET_NODE

Normalized Tag API.

Prototype:

```
#include <flntag.h>
char* flntag_get_dimen(FLNTAG *ntag)
char* flntag_get_member(FLNTAG *ntag)
char* flntag_get_name(FLNTAG *ntag)
char* flntag_get_node(FLNTAG *ntag)
```

Arguments:

FLNTAG*	ntag	(i)	FLNTAG whose components are being obtained.

Returns:

Value string	Pointer to the requested value.

Description: Normalized tags consist of four components: node, name, dimension, and member. This suite of *flntag_get_...()* functions allows the caller to obtain the current value of an individual component. All values are returned as null-terminated strings.

Currently, these entry points are implemented as macros which return a pointer to the private members of the FLNTAG structure. These addresses must be treated in a read-only manner.

See Also: flntag_create(), flntag_gen_str(), flntag_set_dimen(), flntag_set_member(), flntag_set_name(), flntag_set_node().

FLNTAG_PARSE_BRKT4DIMS
FLNTAG_PARSE_COMMA4DIMS

Normalized Tag API.

Prototype:

```
#include <flntag.h>
i16 flntag_parse_brkt4dims(char *dimstr,
                           u16 *dims, u16 dimslen);
i16 flntag_parse_comma4dims(char *dimstr,
                            u16 *dims, u16 dimslen);
```

Arguments:

char*	dimstr	(i)	Dimension string to parse.
u16*	dims	(o)	Size of each dimension found in the parsed string returned within a array of *u16*s. If equal to NULL, this information is not returned.
u16	dimslen	(i)	Length of the given *dims* array. This prevents overwrites should the number of dimensions exceed the size of the array.

Returns:

>= 0	Number of dimensions found in the given string.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_REFSYN	Reference's syntax illegal for the dimension.

Description: Functions *flntag_parse_brkt4dims()* and *flntag_parse_comma4dims()* parses the given dimension string and returns the number of dimension contained within it. These functions can also return the value of the each individual dimension as a u16 array.

Function *flntag_parse_brkt4dims()* expects dimension strings such as “[2][4]”.

Function *flntag_parse_comma4dims()* expects dimension strings such as “2,4”.

See Also: flntag_parse_ref().

FLNTAG_PARSE_REF

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_parse_ref(FLNTAG *ntag, char *ref)
```

Arguments:

FLNTAG*	ntag	(i/o)	FLNTAG to load according to the given reference.
char*	ref	(i)	Reference to parse.

Returns:

FLNTAG_E_GOOD	Reference successfully parsed and loaded.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_REFSYN	Reference's syntax illegal for a tag.

Description: Function *flntag_parse_ref()* parses the given object reference string and loads its components into the given normalized tag handle. Each component is validated to ensure that it has a legal syntax.

The previous contents of the given *ntag* are overwritten by this operation.

See Also: `flntag_create(), flntag_gen_ref().`

FLNTAG_SET_DIMEN
FLNTAG_SET_MEMBER
FLNTAG_SET_NAME
FLNTAG_SET_NODE

Normalized Tag API.

Prototype:

```
#include <flntag.h>
int flntag_set_dimen(FLNTAG *ntag, char *dimen)
int flntag_set_member(FLNTAG *ntag, char *member)
int flntag_set_name(FLNTAG *ntag, char *name)
int flntag_set_node(FLNTAG *ntag, char *node)
```

Arguments:

FLNTAG*	ntag	(i/o)	FLNTAG whose components are being set.
char*	value	(i)	Target set value.

Returns:

FLNTAG_E_GOOD	Reference successfully parsed and loaded.
FLNTAG_E_HANDLE	Invalid FLNTAG handle.
FLNTAG_E_INVARG	Invalid function arguments.
FLNTAG_E_REFSYN	Reference's syntax illegal for a component.

Description: Normalized tags consist of four components: node, name, dimension, and member. This suite of flntag_set_...() allows an individual component of the FLNTAG to be modified.

Valid syntax is enforced by these functions. The enforced rules are as follows:

flntag_set_node flntag_set_name flntag_set_member	Allowed characters are "_@$".and alphanumeric. The first character cannot be numeric.
flntag_set_dimen	A string containing dimensions delineated by brackets (i.e. [4][5]).

See Also: `flntag_create()`, `flntag_gen_str()`, `flntag_get_dimen()`, `flntag_get_member()`, `flntag_get_name()`, `flntag_get_node()`.

- **NORMALIZED TAG REFERENCES**
- *flntag_set_node*
-
-

Chapter 10 . . . .

Object Definitions

A *FactoryLink object* is a user-defined name associated with a set of attributes. Examples of an object's attributes are its type, dimension, and domain, but other properties exist as well. This set of attributes equates to the object's definition.

A FactoryLink application consists of many objects. To view the objects defined within an application, execute the configuration manager (FLCM) and choose "View - Object List" from its main menu. The resulting list shows all objects, along with their definitions, known by the application.

This chapter describes the FactoryLink PAK interface for accessing these tag definitions in a run-time environment.

This section covers the Object CT API. Topics include:

- Object CT Overview
- Object CT API Reference Guide

Object CT Overview

A binary representation of a configuration database file, the object CT is not unlike any other task-specific CT. The main difference is that the CT contains the contents of the application's object database, which is not tied to any one particular task. Another difference is that a large application with many objects requires special handling in order to fit within a standard CT.

Written using the PAK's *ct_...()* primitives, the object CT API hides the details of how the object database contents are coerced into a CT.

Overview of Object CT Services

The Object CT API mirrors the existing *ct_...()* API and provides the following services:

- Opening and closing of the object CT.
- Object definition retrieval based on an object's name.
- Random access, block retrieval for object definitions.

Overview of the Object CT API

Ignoring some organizational overhead, FactoryLink CTs equate to on-disk arrays of C-structures. The object CT is no different, and consists of many arrays of following structure:

```
typedef struct _flobjrec
{
char  tagname[MAX_TAG_NAME+1];
char  tagdomain[MAX_USR_NAME+1];
char  tagtype[MAX_TYPE_NAME+1];
char  tagdescr[MAX_PROC_DESC+1];
char  tagdimen[MAX_DIM_LENGTH+1];
u16   tagperwhen;
u16   tagchgbits;
TAG   tagno;
} FLOBJREC;
```

This structure reflects the current attributes that define a FactoryLink object. Applications should treat this structure as opaque and not access its members

directly. A function-based interface, included with the Object CT API, should be used to query FLOBJREC's values. Using the API shields the PAK task from future changes that alter the object's structure and its members, yet leave the interface alone.

The Object CT API is a data abstracted set of functions. Its usage generally follows some for variant of the following sequence:

1. Open the application's object table.
2. Search for definitions for one or more objects.
3. Close the object table.

Code Scrap: Printing all objects in a particular domain

```
#include <objct.h>


/*
 *  Function print_objs4dom writes to standard output
all objects
 *  configured for a particular domain.
 */
int print_objs4dom(char *flapp, char *tgt_dom)
{
FLOBJREC  rec;
u32       nrecs;
u32       k;
CT        objct;

      if (ct_open_obj(&objct, flapp) != GOOD)
          return ERROR

      nrecs = ct_nrecs_obj(objct);
      for (k = 0; k < nrecs; k++)
      {
```

```
            ct_read_objs(objct, &rec, k, 1);

            if (strcmp(tgt_dim,
flobjrec_get_domain(rec)) == 0)
            {
                printf("Object %s in domain %s\n",
                      flobjrec_get_name(rec), tgt_dom);
            }
        }
        ct_close_obj(&objct);
        return nrecs;
}
```

Object CT API Reference Guide

Presented in alphabetical order, the following pages describe the Object CT API.

-

ct_find_obj PAGERE

ct_nrecs_obj PAGEREF

ct_open_obj PAGEREF

ct_read_objs PAGEREF

flobjrec_get_chgbits PAGEREF

flobjrec_get_descr PAGEREF

flobjrec_get_dimen PAGEREF

flobjrec_get_domain PAGEREF

flobjrec_get_perwhen PAGEREF

flobjrec_get_tag PAGEREF

flobjrec_get_type PAGEREF

-CT_CLOSE_OBJ

Object CT API.

Prototype:
```
#include <objct.h>
int ct_close_obj(CT *objct)
```

Arguments :

CT*	objct	(i/o)	Object CT handle.

Returns :

GOOD	CT closed without error.
CT_NULL_PTR	Null pointer passed in for CT.
CT_FILE_NOT_OPEN	CT currently not opened.
CT_CANNOT_CLOSE_FILE	Error occurred closing file.

Description: Function *ct_close_obj()* closes the object CT. The CT handle should not be referenced after being closed.

See Also: ct_open_obj().

CT_FIND_OBJ

Object CT API.

Prototype: #include <objct.h>

int ct_find_obj(CT *objct, char *objname, FLOBJREC *rec)

Arguments:

CT*	objct	(i)	Object CT handle.
char*	objname	(i)	Name of the object to find.
FLOBJREC*	rec	(o)	Buffer for object's definition.

Returns:

GOOD	Object found.
ERROR	Object not found.

Description: Function *ct_find_obj()* searches the given object CT for the given *objname* and returns its definition.

Function *ct_find_obj()* employs a binary search.

See Also: ct_open_obj().

CT_NRECS_OBJ

Object CT API.

Prototype:

```
#include <objct.h>
int ct_nrecs_obj(CT *objct, u32 *nrecs)
```

Arguments:

CT*	objct	(i)	Object CT handle.
u32*	nrecs	(o)	Number of objects in the given CT.

Returns :

GOOD	Number of objects returned.
ERROR	Unable to determine number of objects.

Description: Function *ct_nrecs_obj()* returns the total number of objects contained within the given object CT.

See Also: ct_open_obj(), ct_read_objs().

CT_OPEN_OBJ

Object CT API.

Prototype: #include <objct.h>

int ct_open_obj(CT *objct, char *flapp)

Arguments:

CT*	objct	(i/o)	Object CT handle.
char*	flapp	(i)	Application directory.

Returns:

GOOD	CT closed without error.
CT_NULL_PTR	Null pointer passed in for CT.
CT_CANNOT_OPEN_FILE	Error occurred opening file.
CT_READ_ERROR	Error occurred reading file.
CT_BAD_MAGIC	Given file is not an object CT.

Description: Function *ct_open_obj()* opens the object CT. This CT handle is passed to all subsequent object CT API calls.

See Also: ct_close_obj().

CT_READ_OBJS

Object CT API.

Prototype:

```
#include <objct.h>
int ct_read_objs(CT *objct, FLOBJREC *recs,
            u32 srec, u32 nrecs)
```

Arguments:

CT*	objct	(i)	Object CT handle.
FLOBJREC*	recs	(o)	Buffer for object definitions.
u32	srec	(i)	Starting record number.
u32	nrecs	(i)	Number of records to read.

Returns:

GOOD	Objects read into buffer.
ERROR	Error occurred reading objects into buffer. Caller might have attempted to read past end of the CT.

Description: Function *ct_read_obsj()* reads the given number of definitions (*nrecs*) into target buffer *recs*, beginning at record *srec*.

See Also: ct_open_obj(), ct_nrecs_obj().

FLOBJREC_GET_CHGBITS
FLOBJREC_GET_DESCR
FLOBJREC_GET_DIMEN
FLOBJREC_GET_DOMAIN
FLOBJREC_GET_PERWHEN
FLOBJREC_GET_TAG
FLOBJREC_GET_TYPE

Object CT API.

Prototype:

```
#include <objct.h>
u16   flobjrec_get_chgbits(FLOBJREC *rec)
char* flobjrec_get_descr(FLOBJREC *rec)
char* flobjrec_get_dimen(FLOBJREC *rec)
char* flobjrec_get_domain(FLOBJREC *rec)
u16   flobjrec_get_perwhen(FLOBJREC *rec)
TAG*  flobjrec_get_tag(FLOBJREC *rec)
char* flobjrec_get_type(FLOBJREC *rec)
```

Arguments:

FLOBJREC*	rec	(i)	FLOBJREC whose components are being obtained.

Returns:

Value	Requested value.

Description: This suite of *flobjrec_get_...()* functions allows the caller to obtain an attribute's value from an object's definition.

While most of the return values are self-explanatory, a few require more detail:

Function *flobjrec_get_chgbits()* refers to a persistence attribute, namely whether to set the change bit on when restoring the persistent value. The value is 1 for set-the-change-bit, and 0 for not.

Function *flobjrec_get_perwhen()* refers to a persistence attribute, namely whether to save the object's value based on time, on value change, or according its domain's persistence settings. The legal values for this attribute are:

0	No persistence of value.
1	Time-based persistence.
2	Exception-based persistence.
3	Time- & Exception-based persistence.
4	Domain settings persistance

Function *flobjrec_get_tag()* returns the base RTDB location. For arrayed objects, this is the RTDB location for the object whose dimensions are all equal to zero.

Currently these entry points are implemented as macros which, in some cases, return a pointer to the private members of the FLOBJREC structure. These addresses must be treated in a read-only manner.

See Also: ct_find_obj(), ct_open_obj().

Index

A

AC file
    comments in 92
    creation 79
    CT statement 93
    DESC statement 93, 100
    description 49, 50, 79, 92
    EDIT statement 93, 95
    END statement 93, 102
    example (external editor program) 106
    FIELD statement 93, 97
    format 92
    format (external editor program) 106
    HEADING statement 93, 98
    INDEX statement 93, 102
    PANEL statement 93, 96
    RELATE statement 93, 99
    sample 102
    SELECT statement 93, 101
    SEQ statement 93, 101
    TASK statement 92
    TYPE statement 93, 99
    VALIDATE statement 93, 96
AC files
    description of 65, 70, 73
    EDIT statement 113
    format 112
ac files
    EDIT statement 113
    format 113
ANALOG 24
API functions
    called by Run-Time Manager only 220, 224, 226, 301
    OBSOLETE 253
API, FactoryLink 38
application
    exiting an instance of 226
    reset data areas of 301
application directory (FLAPP) 66, 71, 74
application directory retrieval 169, 230
application global flags retrieval 231
application-related files 53
architecture
    affect on new task development 48
    of FactoryLink 21
archive (CT) 116, 169
array
    real-time database element 29
arrays 28
ASC files 67, 71, 74
attribute catalog
    AC file 92

C

calling and return conventions 152
CDB (database tables) 66, 71, 74
CDBLIST utility 111, 126
CDX (index) 67, 71, 74
change-read call 27, 160, 208

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

change-status bits
  clear 160, 216
  description 26, 161
  set 160, 304
change-status flags
  change-status bits 160
change-wait call 27, 160, 212
client processes
  retrieve number of 176, 237
  status 155
clock tick 245
CM System Table 128
CMD files 65, 70, 73
command line retrieval 169, 232
components of FactoryLink Software System 23
configuration database tables 49
configuration environment
  testing 113
configuration environment setup 78
Configuration Manager 17, 37, 49, 91, 106
configuration tables (CT)
  CT file 14
  definition for manual 14
control panels 45, 91
control tag retrieval 169, 234
conventions
  calling 152
  notational 13
  return 152
  task design 131
conversion script
  description 117
  format 117
  sample 124
conversion script format 127, 128
converting database tables to CTs 78, 80, 115
  testing 126
copyright message, pointer to 176, 233
cross-reference (XREF) database table 90
CT access services
  description 169
  list of functions 170
CT archive 116, 169
  archive header 117
  calculate offset 170
  close 170, 184
  create/truncate/open for update 170, 186
  find index 170, 187
  format 117
  header 117
  index 117, 118
  open 170, 195
  open for update 170, 200
  read header 170, 196
  read index 170, 197
  read record 170, 198
  read records 170, 199
  record 117
  write header 170, 201
  write index record 170, 202
  write record 170, 203
  write records 170, 204
CT file
  creation 124
  description 51, 80
  header 119
  record 119
CT files 67, 72, 75
CT statement (in AC file) 93
CT type definition (CTG script) 118
CT_CALC_OFFSET 170

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

CT_CLOSE 170, 184
CT_CREATE 170, 186
CT_FIND_INDEX 170, 187
CT_GET_HDRLEN 189
CT_GET_NAME 190
CT_GET_NCTS 191
CT_GET_NRECS 192
CT_GET_RECLEN 193
CT_GET_TYPE 194
CT_OPEN 170, 195
CT_READ_HDR 170, 196
CT_READ_INDEX 170, 197
CT_READ_REC 170, 198
CT_READ_RECS 170, 199
CT_UPDATE 170, 200
CT_WRITE_HDR 170, 201
CT_WRITE_INDEX 170, 202
CT_WRITE_REC 170, 203
CT_WRITE_RECS 170, 204
CTARC structure 117
CTG files 65, 70, 73
CTG script 80
  FIELD statement 118, 121
  format 117
  HEADER statement 118, 119
  RECORD statement 118
  SKIP statement 118, 123
  TABLE statement 118
CTGEN utility 49, 80, 116, 117, 124, 126
CTLIST utility 80, 126
CTNDX structure 117

D

data transfer
  examples 47
  methods 46
data types 24, 90, 160
  ANALOG 90
  FLOAT (floating-point) 90
  LONGANA (long analog) 90
  MAILBOX 90
  storage in kernel area 26
  storage in user area 26
  value 26
  value range and accuracy 26
database access services
  called by miscellaneous functions 177
  description 159
  list of functions 159
  sample 162
  use 159
database table
  contents 79
  converting 78, 80, 115
  definition 14, 94
  description 49
  design 79
  field definition 97
  format 90
  OBJECT 89
  task-specific 89
  TYPE 89
  XREF 89
dBASE
  compatible database library 112, 113, 127, 128
  compatible database manager 113
debugging tasksprogram' 81
DESC statement (in AC file) 93, 100
designing task-specific database tables 91
DIGITAL 24
dimensions 33

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

directory organization 65, 69, 73
DLL files 66, 71, 74
Domain
  defined 22
  items associated with 22, 40
  shared 39
domains 39
DRW files 67, 72, 75

E

edit procedure 95
EDIT statement (in AC file) 93, 95
EDIT statement in AC file 113
EDIT statement in ac file 113
editing
  level of developer 96
editor program execution from CM 106
element array
  one dimension 33
  three dimension 34
  two-dimensional 33
element array dimensions 33
Element Descriptions 35
elements 21, 161
  ANALOG 24
  DIGITAL 24
  FLOAT 24
  getting information about 176
  LONGANA (long analog) 24
  MAILBOX 24
  MESSAGE 24
  predefined 35
  structure 26
END statement (in AC file) 93, 102
environment
  reading from OS/2 248
environment access services
  description 169
environment variables
  FLAPP 53, 63, 64, 68
  FLINK 53, 63, 64, 68
  look up 248
error numbers 176, 225
  FactoryLink client process status 155
  reference list 153
  returned by CT access functions 154
  returned by kernel services 153
  symbolic representations 152
exception processing 28
EXE files 65, 70, 73
exit the calling process 134, 157, 287
EXT files 67, 71, 74

F

FactoryLink
  API 38, 130
  architecture 21
  components 23
  environment variables 53
  files and directories 53
  how architecture affects new task development 48
  kernel 23, 148
  library 22, 149
  library functions 149
  multi-user considerations 39
  operation 21
  real-time database 23, 160
  recommended optional programs 38
  tasks 35
FactoryLink environment variables
  expansion in path names 54

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

FLDOMAIN 54
FLINK 53
FLNAME 54
FLUSER 54
in path names 54
field
definition 97
heading 98
relational 99
selection 101
translation 121
FIELD statement
in AC file 93, 97
in CTG script 118, 121
files
creating 264
FL.OPT 66, 71, 74
FL_ACCESS_MEM 167, 205
FL_ALLOC_MEM 167, 207
relationship to FL_FREE_MEM 229
FL_CHANGE_READ 160, 208
comparison to FL_CHANGE_WAIT 213
FL_CHANGE_READ_TAG_LIST 165, 210
FL_CHANGE_WAIT 160, 161, 212
comparison to FL_CHANGE_READ 209
similarity to FL_CHANGE_WAIT_TAG_LIST 215
FL_CHANGE_WAIT_TAG_LIST 165, 214
FL_CLEAR_CHNG 160, 216
FL_CLEAR_SYNC 160, 162
FL_CLEAR_WAIT 160, 217
FL_COUNT_MBX 166, 218
FL_CREATE_RTDB 220
FL_DBFMTT 171, 222
FL_DELETE_RTDB 224
FL_ERRNO 152, 176, 225
FL_EXIT_APP 226
FL_FORCED_WRITE 159, 227
FL_FREE_MEM 167, 229
relationship to FL_ALLOC_MEM 207
FL_GET_APP_DIR 169, 230
FL_GET_APP_GLOBALS 231
FL_GET_CMD_LINE 169, 232
FL_GET_COPYRT 176, 233
FL_GET_CTRL_TAG 169, 234
FL_GET_ENV 132, 176, 235
FL_GET_MSG_TAG 236
FL_GET_NPROCS 176, 237
FL_GET_PGM_DIR 169, 238
FL_GET_STAT_TAG 169, 239
FL_GET_TAG_INFO 176, 241
FL_GET_TAG_LIST 165, 243
FL_GET_TICK 177, 245
FL_GET_TITLE 176, 246
FL_GET_VERSION 133, 176, 247
FL_GETVAR 248
FL_GLOBAL_TAG 177, 249
FL_HOLD_SIG 168, 251
FL_ID_TO_NAME 158, 252
FL_INIT 176, 253
FL_INIT_APP 254
FL_LOCK 176, 256
relationship to FL_UNLOCK 311
FL_NAME_TO_ID 157, 257
FL_PATH_ACCESS 56, 258
FL_PATH_ADD 57, 259
FL_PATH_ADD_DIR 54, 57, 260
FL_PATH_ALLOC 57, 261
FL_PATH_CLOSEDIR 57, 263
FL_PATH_CREATE 57, 264
FL_PATH_CWD 57, 265
FL_PATH_DATE 57, 266

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

FL_PATH_FREE 57
FL_PATH_GET_SIZE 267
FL_PATH_GET_TYPE 268
FL_PATH_INFO 57, 269
FL_PATH_MKDIR 57, 270
FL_PATH_NORM 54, 57, 272
FL_PATH_OPENDIR 57, 273
FL_PATH_READDIR 57, 274
FL_PATH_REMOVE 57, 276
FL_PATH_RMDIR 57, 277
FL_PATH_SET_DEVICE 54
FL_PATH_SET_DIR 54, 57, 278
FL_PATH_SET_DRIVE 57, 279
FL_PATH_SET_EXTENSION 57, 280
FL_PATH_SET_FILE 54, 57
FL_PATH_SET_NAME 281
FL_PATH_SET_NODE 54, 57, 282
FL_PATH_SET_PATTERN 54, 57, 283
FL_PATH_SET_VERSION 57
FL_PATH_SIZE 57
FL_PATH_SYS 57, 284
FL_PATH_TIME 57, 286
FL_PATH_TYPE 57
FL_PROC_EXIT 134, 157, 287
FL_PROC_INIT 158
FL_PROC_INIT (obsolete) 288
FL_PROC_INIT_APP 290
FL_QUERY_MBX 166, 292
  prior call to FL_COUNT_MBX 219
  relationship to FL_READ_MBX 297, 299
FL_READ 159, 294
  example 163
FL_READ_APP_MBX 298
FL_READ_MBX 166, 296
  relationship to FL_QUERY_MBX 293
FL_RECV_SIG 168, 300
  relationship to FL_SEND_SIG 302
FL_RESET_APP_MEM 301
FL_SEND_SIG 168, 302
FL_SET_CHNG 160, 304
  relationship to FL_CLEAR_CHNG 216
FL_SET_OWNER_MBX 166, 305
FL_SET_SYNC 160, 161
FL_SET_TAG_LIST 164, 306
  relationship to FL_GET_TAG_LIST 244
FL_SET_TERM_FLAG 157, 307
FL_SET_WAIT 160, 161, 308
FL_SLEEP 175, 309
FL_TEST_TERM_FLAG 133, 157, 310
FL_UNLOCK 176, 311
FL_WAIT 176, 312
FL_WAKEUP 176, 313
FL_WAKEUP_PROC 176, 315
FL_WRITE 316
  159
  comparison to FL_FORCED_WRITE 228
  example 163, 164
FL_WRITE_APP_MBX 320
FL_WRITE_MBX 166, 318
FL_XLATE 171, 322
FL_XLATE_GET_TREE 172, 329
FL_XLATE_INIT 171, 324
FL_XLATE_LOAD 172, 326
FL_XLATE_SET_TREE 172, 332
FLAPP 53, 63, 64, 68
  /CT 126
  ASC 67, 71, 74
  CT 67, 72, 75
  DRW 67, 72, 75
  LOG 67, 72, 75
  NET 67, 72, 75
  PROCS 67, 72, 75

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

RCP 67, 72, 75
RPT 67, 72, 75
SPOOL 67, 72, 75
flib library 149
FLIB.H 117, 149, 152, 153, 161, 165
FLINK 53, 63, 64, 68
/AC 79, 109
/CTGEN 80, 118, 124
/KEY 79, 108
/LIB 149
/MSG 324
AC 65, 70, 73
BIN 65, 70, 73
BLANK 65, 70, 73
CTGEN 65, 70, 73
EDI 66, 70, 73
INC 66, 70, 73
INSTALL 66, 70, 73
KEY 66, 70, 73
LIB 66, 71, 73
LOG 66, 71, 74
MSG 66, 71, 74
OPT 66, 71, 74
PAK 66, 71, 74
RPT 74
FLOAT (floating-point) 24
FLREST utility 49, 80
FLRUN 49
FLSAVE utility 49, 80
forced-write call 27, 52, 159, 227
format version number services
description 171
function 171
FRM files 74
functions, representation of 13

G

G files 67, 72, 75
GETENV (standard C library fns) 248
GLOBAL.CT 249
GROUPS files 67, 72, 75

H

H files 66, 70, 73
hardware requirements 18
header files
FLIB.H 117, 149, 152, 153, 161, 165
platform-specific 149
HEADER statement (in CTG script) 118, 119
HEADING statement (in AC file) 93, 98
help 46

I

INDEX statement (in AC file) 93, 102
information panels 45, 91
informing FactoryLink about new task 109
initialization
application instance 254
initialize
calling process 132, 158, 288, 290
FactoryLink application 254
FactoryLink kernel 176, 253
multiple threads, same invocation 289, 291
installation
creation of medium 82
inter-process communication (IPC) 165, 167
IPC
inter-process communication (IPC) 165

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

K

KENV structure 169, 176, 235
kernel 23, 27, 148, 159
  version number 176, 247
kernel and library services
  CT access 156, 169
  database access 156, 159
  environment access 156, 169
  error numbers 153
  format version number 156, 171
  mailbox 156, 165
  memory management 156, 166
  message translation 156, 171
  miscellaneous 156, 176
  path manipulation 156, 171
  process management 156
  signal 156, 167
  sleep 156, 175
  tag list registration and notification 156, 164
KEY file
  creation 50, 79, 108
  description 49, 79, 108
  missing 109
  sample 108
KEY files 66, 70, 73

L

LIB files 66, 71, 73
library, FactoryLink 22, 50, 149
linking object modules 81
linking the object modules 84, 85
LOCAL files 67, 72, 75
locking the database 162, 176, 256
LONGANA (long analog) 24

M

mailbox
  definition 165
  determining number of messages in 166, 218
  monitor 166
  query 166, 292
  read by application instance 298
  read from 166, 296
  set owner 305
  set owner of 166
  validate 166
  write to 166, 318
  write to (by application instance) 320
mailbox services
  comparison to signal services 165
  description 165
MAKE_FULL_PATH 171, 334
Manual, PAK User
  important terms 14
  use 11
MBXMSG structure 165
memory
  accessing 167, 205
  allocating 167, 207
  freeing 167, 229
memory management services
  description 166
MESSAGE 24
message translation services
  description 171
miscellaneous services
  description 176
  list of functions 176
Multiple User Environment 22
multi-user considerations 39

Key

A = Application Editor   C1 = Core Tasks   C2 = Data Logging   C3 = Data Reporting
C4 = Communications   C5 = Power SPC   F = Fundamentals   R = Reference

N

normalized path name
allocating 261
convert to string 284
normalized path names 55
notational conventions 13

O

OBJECT database table 89, 90, 91
open architecture 17
operator, definition for this manual 14
optional programs, recommended 38

P

PAK
ProgrammersAccessKit' 17
PANEL statement (in AC file) 93, 96
panels 45
control 45, 91
designing 91
information 45, 91
layout 91
Path functions 55, 334
path functions
adding subdirectories 260
build normalized directory path 265
check file access mode 258
concatenating paths 259
create directory 270
create working directory 265
delete directory 277
delete file 276
directory searches 273, 274
ending directory searches 263
file date/time stamp 266
file size 267, 268
file time stamp 286
initializing 269
setting directory paths 278
setting drive names (device names) 279
setting file extensions 280
setting file name 281
setting node name 282
setting wildcard search pattern 283
path manipulation services
description 171
function 171
path name
full 171
Path Name Format 54, 55
path name format 63, 64, 69
path names
building 55
normalized 55
with environment variables 54
platform definition 81
pointer return
to copyright message 176
to name of product 246
to product name 176
predefined elements 35
PRG files 67, 72, 75
PRINTF 325
process management services
description 156
sample 158
use 157
process startup
obtaining task ID 291
program directory (/flink) 70
program directory (FLINK) 65, 73

program directory retrieval 169, 238
program files 53
programmer, definition for this manual 14
ProgrammersAccessKit
  description' 17
  softwarerequirements' 18
  use' 17

R

read calls
  changed value of element 160, 208
  changed value of element (sleep if no change) 160, 212
  changed value of element in tag list 165, 210
  change-read call 159
  change-wait call 159
  CT header 170, 196
  CT index 170, 197
  CT record 170, 198
  CT records 170, 199
  description 27
  message from mailbox 166, 296
  message from mailbox by application instance 298
  value of element 159, 294
real-time database
  description of 23, 50, 160
  element 161
  locking 162, 176, 256
  structure of 24
  structure of elements 26
  supported data types 24
  unlocking 162, 176, 311
  wait to read, write, or access 176
RECORD statement (in CTG script) 118, 119
RELATE statement (in AC file) 93, 99
relational field 99
requirements
  hardware 18
  software 18
return conventions 152
RPT files 67, 72, 75
RTDB
  creating an instance of 220
  deleting an instance of 224
RTMCMD element 156
Run-Time Manager 37, 51, 78, 109, 110, 129, 130, 132, 135
  interaction with other tasks 130
run-time requirements 132

S

SELECT statement (in AC file) 93, 101
selection field definition 101
SEQ statement (in AC file) 93, 101
sequence numbers 101
setting up configuration environment 87
shared domain 39
shutting down the system 155
signal services
  comparison to mailbox services 167
  description 167
signals
  definition of 167
  preventing/allowing delivery of 168, 251
  receiving for calling process 168, 300
  sending to target process 168, 302
SKIP statement (in CTG script) 118, 123
sleep services
  description 175
  function 175

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

software requirements 18
source modules
- compilation example 81
- compiling 81, 84, 85
- creation 50, 81
- creation of 84, 85
- example 135
- platform definition 81, 84, 85

SPOOL 172, 336
spool file or line 172, 336
SPRINTF 223, 325
status of client processes 155
storage of data types
- in kernel area 26
- in user area 26

string
- creation of target 172
- preparation of formatted 171, 222

string, creation of target 342, 344, 345
structures
- CTARC 117
- CTNDX 117
- KENV 169, 235
- MBXMSG 165
- MSG 165
- VAL union 162
- VPTR 166

symbolic representations of error numbers 152
sync bits
- clear 160
- description 161
- set 160

sync flags
- sync bits 160

System Configuration Table 37, 80, 109
system shutdown 155
system-specific path name
- convert to normalized 272

T

TABLE statement (CTG script) 118
tag list
- read change in value 165, 210
- registration 164, 306
- retrieval 165, 243
- wait for change in value 165, 214

tag list registration/notification services
- description 164

tag name
- assignment 29

tag names 30
tag number
- description 162
- retrieval 177

task
- defining the name 93
- defining the title 93
- description 21, 35
- design conventions 131
- design guidelines 77

TASK statement (in AC file) 92
task-specific database tables
- description 89, 90
- design 91

termination flag
- check status 133, 157, 310
- set 157, 307

testing
- configuration environment 80, 109, 110
- database table conversion process (to CTs) 80, 126

TITLES file 80, 109

Key

A = Application Editor	C1 = Core Tasks	C2 = Data Logging	C3 = Data Reporting
C4 = Communications	C5 = Power SPC	F = Fundamentals	R = Reference

translation
- creation of target string 341
- FactoryLink ID to process name 158, 252
- get address of current tree 329, 332
- key to associated message 171, 322
- loading new tree from file 326
- message translation initialization 171, 324
- of directory and file names into path 55, 334
- process name to FactoryLink ID 157, 257

TSPRINTF 172, 341, 342, 344, 345
TXT files 66, 71, 74
TYPE database table 89, 90
TYPE statement (in AC file) 93, 99

U

unlocking the database 162, 176, 311
user domain 39
user, definition for this manual 14
utilities
- CDBLIST 111, 126
- CTGEN 49, 80, 116, 117, 124, 126
- CTLIST 80, 126
- FLREST 49, 80
- FLSAVE 49, 80

V

VAL union structure 162
VALIDATE statement (in AC file) 93, 96

W

wait bits
- clear 160, 217
- description 28, 161
- set 160, 308

wait flags
- wait bits 160

wait to read/write/access database 176
wake up
- mask of FactoryLink client processes 176, 313
- specified FactoryLink process 176, 315

write calls
- CT header 170, 201
- CT index record 170, 202
- CT record 170, 203
- CT records 170, 204
- description 27
- forced-write call 159
- message to mailbox 166, 318
- message to mailbox (by application instance) 320
- value of element 159, 316

writing the tasksprogram' 78, 81

X

XREF database table 89, 90

Key

A = Application Editor C1 = Core Tasks C2 = Data Logging C3 = Data Reporting
C4 = Communications C5 = Power SPC F = Fundamentals R = Reference