C++ Programming Libraries

Model 793.00 Software
Proprietary data

This manual and the software it describes are both copyrighted, with all rights reserved. Under the copyright laws, neither this manual nor the software may be copied, in whole or part, without written consent of MTS Systems Corporation, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as are made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaded to another person. Under the law, copying includes translating into another language or format. This software may be used in any computer, but extra copies cannot be made for that purpose.

Copyright information

© 2003 MTS Systems Corporation. All rights reserved.

Trademark information

MTS, the MTS logo, and TestWare are registered trademarks of the MTS Systems Corporation.

FlexTest, Station Builder, Station Manager, TestStar, and TestWare are trademarks of the MTS Systems Corporation.


Publication Information

<table>
<thead>
<tr>
<th>Software Version No.</th>
<th>Document No.</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4</td>
<td>100-003-354</td>
<td>02/10/99</td>
</tr>
<tr>
<td>3.0</td>
<td>100-037-834</td>
<td>06/12/00</td>
</tr>
<tr>
<td>3.1</td>
<td>(none)</td>
<td>10/05/00</td>
</tr>
<tr>
<td>3.2</td>
<td>(none)</td>
<td>07/03/01</td>
</tr>
<tr>
<td>3.3</td>
<td>(none)</td>
<td>11/04/02</td>
</tr>
<tr>
<td>3.4</td>
<td>(none)</td>
<td>09/15/03</td>
</tr>
</tbody>
</table>
# Table of Contents

Chapter 1 Introduction ........................................................................................................... 1
  1.1 Scope ......................................................................................................................... 1
  1.2 Unsupported Features ............................................................................................... 1
  1.3 Related Documentation ........................................................................................... 1
  1.4 Technical Support .................................................................................................... 2
      1.4.1 External Customers ......................................................................................... 2
      1.4.2 Internal Customers ....................................................................................... 2

Chapter 2 Overview ............................................................................................................. 3
  2.1 System Overview ....................................................................................................... 3
  2.2 Conventions and Dependent Libraries ...................................................................... 4
      2.2.1 Error Handling ............................................................................................... 5
      2.2.2 Engineering Units .......................................................................................... 5
      2.2.3 MTS 793 Callback classes ............................................................................. 6
      2.2.4 MTS 793 Model classes ............................................................................... 6
      2.2.5 MTS 793 Text class and Natural Language support ....................................... 7
      2.2.6 Real-time event classes .................................................................................. 8
      2.2.7 Allegris Workshop class library ...................................................................... 8
      2.2.8 Allegris Run-Time Type Information (RTTI) ..................................................... 9

Chapter 3 Systems, Stations, and Applications .................................................................... 11
  3.1 The System (RtSystem) ............................................................................................ 11
      3.1.1 Connection to the system .............................................................................. 11
      3.1.2 Station List ...................................................................................................... 12
      3.1.3 Application Thread Watchdog ......................................................................... 12
      3.1.4 Other system-wide settings ............................................................................ 13
  3.2 Connection to the station (RtStation) ....................................................................... 14
      3.2.1 Application Name ........................................................................................... 14
      3.2.2 Application Types ............................................................................................ 15
      3.2.3 Application tolerance to named resource changes ......................................... 16
      3.2.4 Disconnecting from a station ......................................................................... 16
      3.2.5 COM Interface Support .................................................................................. 17
  3.3 Station Overview (RtStation) .................................................................................... 17
      3.3.1 Station Load States ........................................................................................ 17
      3.3.2 Named vs. Unnamed Objects ......................................................................... 18
      3.3.3 Querying for named public objects ............................................................... 19
      3.3.4 Explicit Support for other components .......................................................... 20
      3.3.5 Unloading a Station ....................................................................................... 21
      3.3.6 Scheduler overrun detection ........................................................................... 21
      3.3.7 Other attributes ............................................................................................... 22
  3.4 Multi-controller FlexTest SE Workstations ............................................................... 22
      3.4.1 Controller/Station Selection Dialog (MTSBoxSelect) ..................................... 22
      3.4.2 Lower-level Multi-controller Model Classes (MTSBoxInfo & MTSBox) .......... 24

Chapter 4 Signals ............................................................................................................... 27
  4.1 Signal Attributes ....................................................................................................... 27
      4.1.1 Querying the value of a signal ........................................................................ 27
      4.1.2 Setting the value of a signal ........................................................................... 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.3 Notification of Value Change (RtIntegerSig only)</td>
<td>29</td>
</tr>
<tr>
<td>4.1.4 Signal Dimension and Range Information</td>
<td>29</td>
</tr>
<tr>
<td>4.1.5 Signal type and source</td>
<td>30</td>
</tr>
<tr>
<td>4.1.6 Alternate data streams</td>
<td>31</td>
</tr>
<tr>
<td>4.1.7 Creating Private Signals</td>
<td>32</td>
</tr>
<tr>
<td>4.2 Predefined Time Signals</td>
<td>33</td>
</tr>
<tr>
<td>4.3 Polling signal values</td>
<td>34</td>
</tr>
<tr>
<td>4.3.1 Using the RtSignalList class</td>
<td>34</td>
</tr>
<tr>
<td>4.3.2 Process-wide signal polling</td>
<td>35</td>
</tr>
<tr>
<td><strong>Chapter 5</strong> Actions</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Predefined and station-defined actions</td>
<td>37</td>
</tr>
<tr>
<td>5.2 Application defined actions</td>
<td>38</td>
</tr>
<tr>
<td>5.2.1 Generating callbacks to the application</td>
<td>40</td>
</tr>
<tr>
<td>5.2.2 “Request Test State Change” commands</td>
<td>41</td>
</tr>
<tr>
<td>5.2.3 Custom generator stop commands</td>
<td>41</td>
</tr>
<tr>
<td>5.2.4 Delay commands</td>
<td>41</td>
</tr>
<tr>
<td>5.3 Action types and RtActionInfo</td>
<td>42</td>
</tr>
<tr>
<td><strong>Chapter 6</strong> Control Channels</td>
<td>43</td>
</tr>
<tr>
<td>6.1 RtChannel class</td>
<td>43</td>
</tr>
<tr>
<td>6.2 RtCtrlMode class</td>
<td>45</td>
</tr>
<tr>
<td>6.3 Mode Switching</td>
<td>48</td>
</tr>
<tr>
<td>6.3.1 Static Mode switch</td>
<td>48</td>
</tr>
<tr>
<td>6.3.2 Asynchronous mode switch</td>
<td>49</td>
</tr>
<tr>
<td>6.4 Span and Set Point controls</td>
<td>49</td>
</tr>
<tr>
<td>6.4.1 Changing span and set point explicitly</td>
<td>50</td>
</tr>
<tr>
<td>6.4.2 CE-Compliant Velocity Limiting</td>
<td>51</td>
</tr>
<tr>
<td>6.4.3 Locking channel span and set point controls</td>
<td>52</td>
</tr>
<tr>
<td>6.4.4 Master Span</td>
<td>52</td>
</tr>
<tr>
<td>6.5 Dual-Compensation modes</td>
<td>53</td>
</tr>
<tr>
<td>6.6 Driving external controllers</td>
<td>54</td>
</tr>
<tr>
<td>6.6.1 Program with Feedback channels</td>
<td>54</td>
</tr>
<tr>
<td>6.6.2 Program Only channels</td>
<td>55</td>
</tr>
<tr>
<td><strong>Chapter 7</strong> Segment Generators and Profiles</td>
<td>57</td>
</tr>
<tr>
<td>7.1 Overview</td>
<td>57</td>
</tr>
<tr>
<td>7.2 Locking a segment generator</td>
<td>58</td>
</tr>
<tr>
<td>7.3 Queuing waveforms to be played out</td>
<td>58</td>
</tr>
<tr>
<td>7.4 Program control (stop/start, hold/resume)</td>
<td>60</td>
</tr>
<tr>
<td>7.4.1 Program state machine</td>
<td>60</td>
</tr>
<tr>
<td>7.4.2 Waiting for Program State changes to occur</td>
<td>61</td>
</tr>
<tr>
<td>7.4.3 Transition states</td>
<td>61</td>
</tr>
<tr>
<td>7.4.4 Changing transition times</td>
<td>62</td>
</tr>
<tr>
<td>7.4.5 When segment generators automatically stop</td>
<td>63</td>
</tr>
<tr>
<td>7.4.6 Monitoring station’s composite program state</td>
<td>64</td>
</tr>
<tr>
<td>7.4.7 Breakpoint Holds</td>
<td>64</td>
</tr>
<tr>
<td>7.5 Maintaining counters and signals</td>
<td>65</td>
</tr>
<tr>
<td>7.6 Changing set point through control flags</td>
<td>66</td>
</tr>
<tr>
<td>7.7 Enabling/disabling compensators</td>
<td>66</td>
</tr>
<tr>
<td>7.8 Synchronizing multiple channels</td>
<td>67</td>
</tr>
<tr>
<td>7.8.1 Detecting profile starvation</td>
<td>68</td>
</tr>
<tr>
<td>7.8.2 Staying lock-step synchronized</td>
<td>68</td>
</tr>
<tr>
<td>7.8.3 Resynchronizing</td>
<td>69</td>
</tr>
<tr>
<td>7.8.4 Synchronizing parameter changes</td>
<td>69</td>
</tr>
<tr>
<td>7.8.5 Common Scenarios</td>
<td>70</td>
</tr>
</tbody>
</table>
# Chapter 8 Data Acquisition

8.1 Data Acquisition Overview ................................................................................. 97

8.2 Sequences (RtSequence) ............................................................................... 98

8.3 Specifying when data is collected (RtTrigger) ...................................................... 99

8.3.1 Trigger at constant time interval (RtTimeTrigger) ........................................ 99

8.3.2 Trigger on change in level (RtDeltaTrigger) ................................................. 102

8.3.3 Trigger on crossing a level (RtLevelTrigger) ............................................... 103

8.3.4 Trigger on peak/valley using level (RtReversalTrigger) ............................ 104

8.3.5 Trigger on peak/valley using delay (RtReversalTrigger) ......................... 106

8.3.6 Running max/min acquisition (RtReversalTrigger) ................................ 107

8.3.7 Trigger on End of Segment (RtEndLevelTrigger) .................................. 109

8.4 Buffers (RtBuffer) .......................................................................................... 109

8.4.1 Creating buffers (any buffer type) ............................................................. 109

8.4.2 Accessing buffer data (any buffer type) .................................................... 110

8.4.3 Requesting notification of data (any buffer type) ...................................... 112

8.4.4 Linear Buffers (RtLinearBuffer) ................................................................. 114

8.4.5 Linear Contiguous Buffers (RtLinearCBuffer) ........................................ 115

8.4.6 Circular Buffers (RtCircularBuffer) ............................................................ 116

8.5 Enable/disable data acquisition with a gate ....................................................... 117

8.5.1 Enable/disable acquisition from an RtAction ............................................ 117

8.5.2 Cycle-based data acquisition (RtIntRngDt) ............................................... 119

8.6 High-speed data acquisition ........................................................................... 121

8.7 Time-history data acquisition ......................................................................... 122

8.7.1 Configuring the RtTimeHistAcq .............................................................. 123

8.7.2 Creating and queuing buffers .................................................................... 124

8.7.3 Processing the data ...................................................................................... 125

8.7.4 Starting and stopping the acquisition ....................................................... 125

8.7.5 Detecting errors .......................................................................................... 126

8.7.6 Synchronizing with command generation .................................................. 126

# Chapter 9 Detectors

9.1 Single Floating Point Limit Detector (RtFloatLdt) .............................................. 129

9.1.1 Detector Modes ......................................................................................... 130

9.1.2 Attributes of each limit ............................................................................ 131

9.2 Group Floating Point Detector (RtFloatLdt) .................................................... 133

9.3 Integer Limit Detector (RtIntegerLdt) ............................................................ 134
# Table of Examples

Example 1 — MyDemo’s use of the MTSBoxSelect class ................................................................. 23  
Example 2 — Creation of a typical RtAction generating a callback .................................................. 40  
Example 3 — Creation of an action containing a delay. ................................................................. 42  
Example 4 — Simple RtSegment Example .................................................................................. 59  
Example 5 — Timed data acquisition .................................................................................... 100  
Example 6 — Configuring an RtDeltaTrigger ............................................................................. 102  
Example 7 — Configuring an RtLevelTrigger ............................................................................ 103  
Example 8 — Configuring an RtReversalTrigger (RDT_PEAK_VALLEY_LEVEL) ...................... 105  
Example 9 — Configuring an RtReversalTrigger (RDT_PEAK_VALLEY_TIME) ......................... 107  
Example 10 — Configuring an RtReversalTrigger (RDT_MIN_MAX) ......................................... 108  
Example 11 — Creating a data acquisition buffer ........................................................................ 110  
Example 12 — Accessing buffer data ..................................................................................... 111  
Example 13 — Accessing buffer data (via pointer) ...................................................................... 111  
Example 14 — Buffer notification callback ................................................................................ 113  
Example 15 — Enabling data acquisition from an RtAction ......................................................... 118  
Example 16 — Setting up cycle-based data acquisition .............................................................. 120  
Example 17 — High-speed data acquisition .............................................................................. 121  
Example 18 — Creating and configuring an RtTimeHistAcq ....................................................... 123  
Example 19 — Adding RtTimeHistBuffers to an RtTimeHistAcq .............................................. 124  
Example 20 — Creating and configuring an RtTimeHistAcq synchronized with command generation. ... 126  
Example 21 — Floating-point limit detector ............................................................................... 129  
Example 22 — Creating a peak/valley change detector .............................................................. 139  
Example 23 — Reacting to a station interlock event .................................................................... 149  
Example 24 — Determining where to display errors when handling requested test state changes .......... 159  
Example 25 — Determining if exclusive control requires manual controls to be locked................... 161  
Example 26 — Getting the list of digital input signals (one way). ................................................. 165  
Example 27 — Getting the list of digital input signals (another way) ........................................... 166  
Example 28 — Locking a digital output for exclusive use .......................................................... 166
Table of Figures

Figure 1 — System Overview ...............................................................................................................3
Figure 2 — Overview of Control Channels........................................................................................43
Figure 3 — RtCtrlMode functional overview..................................................................................46
Figure 4 — Segment Generator functional overview ......................................................................57
Figure 5 — Segment Generator State Machine ...............................................................................60
Figure 6 — RtSegment waveshapes ................................................................................................77
Figure 7 — RtCyclic2 waveshapes ....................................................................................................79
Figure 8 — Typical RtCyclic2 waveform. .......................................................................................80
Figure 9 — RtCyclic and RtSweep waveshapes..............................................................................81
Figure 10 — Example of Peak/Valley data acquisition ..................................................................97
Figure 11 — Trigger points for RtTimeTrigger ...........................................................................100
Figure 12 — Trigger points for RtDeltaTrigger ...........................................................................102
Figure 13 — Trigger points for RtLevelTrigger ..........................................................................103
Figure 14 — Sampled points for RtReversalTrigger (RDT_PEAK_VALLEY_LEVEL) ......................104
Figure 15 — Sampled points for RtReversalTrigger (RDT_PEAK_VALLEY_TIME) .......................106
Figure 16 — Data Acquisition buffer ..........................................................................................110
Figure 17 — Sampled points for Cycle-based timed data acquisition .........................................119
Figure 18 — Peak/Valley limit detector (PEAK_VALLEY_CHANGE mode) ......................................138
Figure 19 — Peak/Valley limit detector (UNDER_PEAK mode) ....................................................138

Table of Tables

Table 1 — System unit assignments..................................................................................................5
Table 2 — Saturation values for different hardware ....................................................................47
Table 3 — Mapping RtSegGen 'ProgramState' to RtStation 'ProgramStates' ...............................64
Table 4 — Mapping RtSegGen 'ProgramState' to RtSyncGrp 'ProgramStates' .............................67
Table 5 — Supported ratios of filter rate to system rate ...............................................................71
Table 6 — Normalized Dimensions ............................................................................................73
Table 7 — Segment Generator control flags ..............................................................................74
Table 8 — Buffer callback reason codes ....................................................................................112
Table 9 — Scope of attributes on a Group Floating Point Detector ..........................................133
Table 10 — Scope of attributes on an A-B Compare Limit Detector ...........................................142
Table 11 — Values of predefined HSM integer signals ...............................................................152
Table 12 — Test states, and corresponding RSC light patterns and signal values ......................157
Chapter 1
Introduction

1.1 Scope

This document provides an overview to writing C++ programs to run on top of MTS 793.00 System Software Version 3.4 to control any of the FlexTest or TestStar products supported by this software. This document summarizes the supported programming interface for writing typical applications. The applications will typically be written by MTS Software Engineers, but may be done by some customers.

1.2 Unsupported Features

The classes and header files described in this document contain additional methods that are not described here. In general, capabilities not described in this document are not supported. They may be:

1. Methods only used by the Station Manager to load and maintain the station.
2. Obsolete methods, which are not functional (they may even crash).
3. Capabilities partially implemented, which may not be functional (they may crash).

In general, only the classes and methods described in this document are supported for Version 3.4.

1.3 Related Documentation

This document assumes familiarity with the following information.

- Operational familiarity with MTS 793 Software concepts and user interface — See the Model 793.00 System Software Manual, and other use documentation.
- C++ programming language, and the Visual C++ development environment — see the Microsoft Visual C++ Documentation.
- This manual provides an overview to the 793.00 Real-Time API. Detailed descriptions of the calls are in RT.HLP.
- Some familiarity with 793 model classes — see online help in GENUTIL.HLP
- Some familiarity with 793 error handling classes — see online help in GENUTIL.HLP
- Some familiarity with the 793 engineering units subsystem — see online help in GENUTIL.HLP
- Some familiarity with the 793 callback mechanism — see online help in GENUTIL.HLP
• Familiarity with the Allegris Workshop (formerly C++/Views) is not required, but may be useful in some applications — see online help in VIEWSHLP.HLP.

1.4 Technical Support

1.4.1 External Customers

MTS Systems will provide some level of technical support for this product to external customers. Usually, the external customer is modifying a C++ program provided by an MTS application group. In this case, that application group should handle the technical support for the customer.

When no application group exists, the 793 platform group will manage the technical support. At this time, there are no training resources available to support customers programming to this API.

1.4.2 Internal Customers

The 793 platform group provides technical support and some training for application groups within MTS Systems. This includes:

• Informal training
• Application design review and consulting
• Access to our Tracker database for reporting problems and enhancement requests
• Technical knowledge base (under construction)

These are all available on the MTS Intranet at http://groups.mts.com/cca/.
Chapter 2
Overview

2.1 System Overview

Figure 1 shows an overview of the structure of an MTS 793 system. Refer to this diagram in the following discussion.

A **System** consists of a collection of hardware resources and processors. The system is initialized when **sysload** is run. This does any required initialization, and creates **Hardware Resource** objects within the system, corresponding to the physical hardware contained in the box (e.g., Analog Inputs, Digital Outputs, etc.). The HWI file is used to tell the system what it contains. Once **sysload** has finished this activity, the system is ready to have stations loaded into it.

A **station** is a collection of these hardware resources that will be used to perform some test, along with definitions of how these objects will interrelate. The station defines control channels, signals (input and output), actions, and detectors. These logical entities are created when the station is loaded, and are hooked up to the hardware resources allocated to the station.
A station configuration is the specification of what a station will contain, and how it will be configured. It is a disk file, created and maintained by the Station Builder application. The Station Manager application is used to load a station configuration into the machine, thus creating a functioning station.

Once a station is loaded, the Station Manager that loaded it is the operator’s “window” into that station. Through the Station Manager, the operator can tune the controllers, set operator limits, calibrate feedbacks and otherwise adjust the station. These settings can be saved as parameter sets within the station configuration. The operator can also manually exercise the control channels on the station, and view the condition of the station using a variety of displays, meters, and a scope.

To load a second station, a second instance of Station Manager is invoked. Each copy of Station Manager is the window into that particular station.

For performing a test on a station, the operator will run some test application. This may be Multi-Purpose TestWare (MPT), Basic TestWare, or some more specific application. These applications do not load stations. They connect up to an existing station, and manipulate it to perform the test. The application will manipulate objects already built into the station, as well as creating additional objects. These new objects may include:

- waveforms on the control channels,
- data acquisition to collect data from input signals,
- detectors for monitoring integer and floating point signals,
- actions.

This test application runs in parallel with the Station Manager. While the test is in progress, the operator can continue to watch (and to some extent control) the station through the Station Manager.

This document is concerned with the C++ programming interface that test applications will use. The details of this interface are described in the on-line help file RT.HLP. Both applications and the Station Manager use this interface. Therefore, it contains more than typical applications will use. This document guides the programmer through these classes, showing him/her what portions are appropriate to be used by typical test applications.

### 2.2 Conventions and Dependent Libraries

This real-time API does not stand on its own. It is layered on a variety of other packages. In general, these are documented in their own help files. This section discusses how these packages are used by applications in relation to this API.

**Note:**

The information in this section is not necessary to get a general understanding of the structure of this API. A first reading of this document could skip to the next chapter.

---

1 Some existing custom applications may create additional control channels and controllers. This is not encouraged, and the issues and mechanisms to do it are not described in this document. Future versions will provide better tools for creating niche-specific controllers.
2.2.1 Error Handling

This programming interface propagates errors using the MTS 793 error-handling package. That package principally consists of the MStatus class. An MStatus is a container for status information. Most methods in the RT interface pass in a pointer to an MStatus as the last parameter. If the client passed in a NULL, and an error occurs, the error will be considered fatal, and the application will terminate.

If an MStatus pointer is passed into the method, the method will set the appropriate status indication into the object.

Many methods also return a boolean value, indicating success (TRUE) or failure (FALSE) of the operation. This indication is a duplication of the information in the MStatus object. It is just an easy place for the client to check for failure.

For more information on MStatus and the rest of the error-handling package, see the online reference information in GENUTIL.HLP.

Note:

The MTS 793 error-handling package also contains a mechanism to signal errors via exceptions. The implementation was written before exception handling was built into C++, and therefore has some serious restrictions. Internally, MTS 793 software does not use the catching of exceptions to handle errors. While this mechanism may work, it is not encouraged, since there may be undesirable side effects in the RT API.

2.2.2 Engineering Units

The MTS 793 Engineering Units Subsystem is used to manage engineering unit data and conventions within the system. This subsystem contains a “database” of information about what Dimensions exist (e.g. Length, Time, Force, Strain), and what units can be used to represent values in these dimensions (e.g. in, mm, cm, sec, lbf, mm/mm). This information is contained in the file SYSUNIT.UDF.

Internally, the MTS 793 system operates entirely within a standard set of units called system units. For each dimension, a particular unit was selected, and that is the unit which all the API methods expect the values to be in. The system unit assignments are shown in Table 1. The file SYSDEF.UDF specifies this set of bindings.

Within MTS 793 software, engineering unit conversions generally happen at the user interface. The system and the application carry the data in system units until it is to be presented to the operator, or placed into a public file. At that time, it is converted into the units the operator or application specifies.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Assigned unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>mm</td>
</tr>
<tr>
<td>Force</td>
<td>kN</td>
</tr>
<tr>
<td>Time</td>
<td>Sec</td>
</tr>
<tr>
<td>Frequency</td>
<td>Hz</td>
</tr>
<tr>
<td>Temperature</td>
<td>°C</td>
</tr>
<tr>
<td>Angle</td>
<td>degrees</td>
</tr>
<tr>
<td>Volts</td>
<td>V</td>
</tr>
<tr>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>Ratio</td>
<td>unity</td>
</tr>
<tr>
<td>SegmentCount</td>
<td>Segments</td>
</tr>
<tr>
<td>Others</td>
<td>Combinations of the above.</td>
</tr>
</tbody>
</table>

Table 1 — System unit assignments

Note:

SYSUNIT.UDF and SYSDEF.UDF are distributed with the system, and should not be changed by anyone. Serious problems can
result if these files are changed inappropriately. This is particularly true of SYSDEF.UDF. Just changing the assignments in there will NOT appropriately change what the system does internally!

For more information on the engineering units subsystem, see the online help in GENUTIL.HLP.

### 2.2.3 MTS 793 Callback classes

The MTS 793 callback mechanism provides a type-safe way to specify, store, and dispatch callbacks. It is used throughout the MTS 793 system. It allows a client to specify an object/method to be passed to some other package, and for that package to invoke the callback later (for example, when something completes). The recipient of this callback can store it, and later invoke it, without knowing the class of the client.

This mechanism is based on C++ templates, and is done without losing the type-safe nature of C++ programming.

For more information on this mechanism, see the *Callbacks Overview* within GENUTIL.HLP.

### 2.2.4 MTS 793 Model classes

The MTS 793 model classes are a formalized mechanism for accessing information about the parameters and state of objects. On most of the objects in the RT API, a parameter may be queried directly (via `set` and `get` methods), or a pointer to a *Parameter* subclass can be returned.

Through this *Parameter* object, an application can:

- **Determine the valid values for a parameter** — For numbers, this includes dimension, range, display range, display units and resolution. For enumerations, this includes the list of possible values.

- **Set and query the value** — if the value is rejected, an error is returned.

- **Register callbacks for when the object changes**\(^2\) — if something about the object changes, this callback will be invoked, including some indication of what on the object changed. This may include the value of the object, or some other context information like dimension, range, and resolution. For enumerations, this includes the list of possible choices. This callback will also be invoked if the object is destroyed.

The principal model classes of interest here are:

- **Range** — This class describes a range of possible values for a *Number*. This includes the dimension, min/max values, resolution, etc.

- **Number** — This class models a floating-point number. It contains or references a *Range* to describe the dimension and range for the number. It also contains a *Range* to describe the display range and units, which are used by some controls when viewing this object.

---

\(^2\) Version 2.3 introduced a new syntax for notification callbacks on model objects. This new protocol allows one callback to do what required two callbacks in the past. The old syntax is still supported, however applications are encouraged to use the new syntax.
SubRange — This class describes a range that is constrained by another range. SubRange is a subclass of Range. It also contains a reference to the Range object that constrains it. If its constraining range changes, the subrange may have to change.

GenList — This class is used for most container applications in the interface. It is a cross between an ordered collection and a dictionary. Objects can be inserted and referenced by both index and key name. The key names must be unique on the list. An object may be referenced in the list more than once, but the key name by which it is inserted must occur only once.

OneOfList — This class models an enumeration. It contains a reference to a GenList that specifies the values this OneOfList can take on. The selection can be made or queried by name, index, or object pointer.

TwoState — Is a special OneOfList that can only take on two values.

ManyOfList — This class models an ordered sub-list. It references a GenList that serves as the full list of possible selections.

FilteredList — This class maintains a sub-list, based on a full list and a selection criterion. The criterion is implemented as a callback method that is called to determine if each item in the full list should be in the sub-list.

ParmGroup — This class models a composite object that contains other objects. Most of the classes within the RT interface are subclasses of ParmGroup.

RefParmGroup — This is similar to ParmGroup, except it does not own the objects within it. This class is useful for building up logical groupings of objects that are owned elsewhere.

An overview of the MTS 793 model mechanism in general, and these classes in particular, is available in GENUTIL.HLP.

Note:

These model classes are generally not multi-threaded. For the most part, they should only be manipulated within the main thread of the application. When operating in other threads, the application should use the direct methods available on the RT classes.

When the application needs to access or manipulate Parameter classes from other threads, it must handle synchronization itself. For example, the MPT real-time thread extract values out of model objects, but it locks the model before hand. This keeps the user interface from changing any of the values while the real-time thread is accessing them.

2.2.5 MTS 793 Text class and Natural Language support

Virtually all text strings within the RT API are handled with the Text class. The Text class is in many ways like strings classes from other sources. It manages arbitrarily long strings, and provides the normal assignment, concatenation and comparison operators.
However, **Text** can hold a natural language label, in addition to the normal string value. The **value** is the portion that all comparisons use. It usually is independent of the current natural language. The **label** is usually extracted from a natural language resource file. The label is not used in comparisons. The label is used to display the text to the user. If there is no label, the value is displayed as the natural language string.

Many of the objects in the MTS 397 programming library have names. For most of these objects, they have both an internal name and a display name. A client application should follow the convention of displaying the natural language label to the user.

The method `getNatLangStr()` will return a reference to the natural language label (if it exists), or the value (if the label does not).

### 2.2.6 Real-time event classes

The **Callback** mechanism described in Section 2.2.3 is a programming mechanism that operates within one thread. It simply allows storing a callback for later invocation.

The **RTEvent** class provides an asynchronous multi-thread, multi-process event notification mechanism. An **RTEvent** is created in the context of a particular thread. It has one or more callbacks associated with it. When it is triggered, those callbacks will be delivered in the thread in which the **RTEvent** was created. This delivery is done through the operating-system message queue for the thread, so it is serialized with respect to other event-based activity going on in the thread.

This package allows threads within an application to send events to each other. This mechanism is also used extensively within the real-time API to deliver event notifications from the machine. Much of this is done through **RtActions** (which use **RTEvents** internally).

This mechanism is based on C++ templates, and is done without losing the type-safe nature of C++ programming.

This package also provides **RtTimerEvent**. This is a subclass of **RTEvent**, but delays delivery of the callback for a specified interval.

**Note**

For the **RTEvents** to work in a thread, the application must create an **RtNotifier** object in that thread. This is usually done in the main routine of the application and in the thread procedure for the thread. See Chapter 15 starting on page 169 for more information.

### 2.2.7 Allegris Workshop class library

The **Allegris Workshop** (formerly C++/Views) contains a class library that many MTS 793 applications use for their user interface. The MTS 793 real-time API for C++ uses data classes from that library.

In general, applications need not be concerned with this library. While all RT classes are derived from Allegris classes, most methods that an application will use are supplied by one of the MTS 793 subsystems.

The MTS 793 development kit is shipped to external customers with enough of the Allegris include files to allow applications to be written to this real-time API. If the customer wants to use any of the other Allegris capabilities, the product must be purchased separately.
When the Allegris Workshop is used to develop the user interface portion of the application, the Allegris DLL supplied with MTS 793 must be used (mtsvw001i.lib and mtsvw001.dll).

For more information on the Allegris classes, see its online documentation in VIEWSHLP.HLP.

### 2.2.8 Allegris Run-Time Type Information (RTTI)

The MTS 793 architecture (as well as the Allegris library) was designed before C++ implementations contained the ANSI Run-Time Type Information (RTTI) feature. Allegris implements its own RTTI in the VClass class. This is used within MTS 793 to be able to dynamically determine the class of an object. Some useful things this allows applications to do include:

- Ask an object for its class descriptor (VClass pointer).
  
  See `VObject::getIsA()` in Allegris

- Ask if an object is of a particular class (or subclass)
  
  See `VObject::isA()` in Allegris

- Intelligently down-cast a pointer, throwing a fatal error if the cast is not appropriate.
  
  See `CAST(), CAST_CONST()` in GENUTIL.HLP

- This mechanism is used within RtStation to get lists of all objects of a particular class.
  
  See `RtStation::getAllObjects()` and `RtStation::findRtObject()`.

Applications are free to use the standard C++ RTTI facilities.
Chapter 3
Systems, Stations, and Applications

3.1 The System (RtSystem)

The RtSystem class models the test machine controlled by the MS 793 system software. It serves as the application’s window into the system (machine). A typical application will principally use the RtSystem object to find the station of interest. The system object provides the following capabilities:

- Connection to the system
- List of stations
- Application thread watchdog
- Other system-wide settings

3.1.1 Connection to the system

Explicitly connecting to a system is not necessary. An attempt to connect to a station will implicitly connect to the last system that was loaded with SYSLOAD.

Note

FlexTest SE systems support connecting multiple, independent controllers to a single workstation. In that situation, the application must select what system to connect to. The protocol for handling those systems is discussed in 3.4 on page 22.

Connecting to a particular system is usually done using the method:

```cpp
RtSystem *rtSystem = RtSystem::find( nodeName, applicationName, retStatus)
```

nodeName — This determines which system is to be connected to. If this is blank, null, or not specified, it will try to connect as follows:

1. If a simulated machine is running on the local processor, it will be used.
2. Otherwise, the node specified in the registry is used. That node address is placed there by the Installation Program.

It is typical for applications to accept a “-system” option on their command line, and pass the user-specified node address as this parameter.
**applicationName** — This is stored, but not otherwise used. Eventually it may be passed to a client wanting to know what applications are attached to the system.

A program can only be connected to one system. Making this call multiple times with a different **nodeName** will return NULL.

Destroying the **RtSystem** object will disconnect the application from the system, thus disconnecting from all the stations.

The method **loadAndFind** is similar to **find** but will automatically do a **sysload** if the system is not loaded. This is only useful for applications that do not need to have a station loaded.

---

### Related RtSystem Methods:

```c
static RtSystem * find(const Text *pipeName, const Text *appName, MStatus *status = 0) ;
static RtSystem *loadAndFind(const Text *name, const Text *hwiName, const Text *appName, MStatus *status = 0);
static RtSystem & getRtSystem(); // Returns the currently connected RtSystem.
static const Text & getSystemName(); // Returns the nodeName of the connected system object.
static const Text & getDefaultSystemName(); // Returns the default nodeName. This is the last node downloaded by SYSLOAD.
```

---

### 3.1.2 Station List

Once an **RtSystem** pointer is available, the program can query for the stations. The method:

```cpp
const GenList & getStationNameList(retStatus)
```

returns a **GenList** of station names. This list can be presented to the operator to choose a station to connect to. This list only contains names. The item pointers in the list are all zero.

The method **getStationStateByName(stationName)** allows a program to determine the load-state of a station without actually connecting to it. This is used to defer connecting to the station until it is all the way loaded, or if it is unloading (See Section 3.3.1 on page 17).

---

### Related RtSystem Methods:

```cpp
const GenList * getStationNameList(MStatus *status) const
StationLoadStates getStationStateByName ( const Text & stnName , MStatus * status ) const
```

---

### 3.1.3 Application Thread Watchdog

The **RtSystem** object provides a mechanism to protect the machine if an application locks up or experiences a deadlock. This mechanism is optional, and is **disabled** by default. The application can turn on this watchdog for one of its threads. This is typically used to watch the thread that manages the user interface for the application. Thus, if the application becomes unresponsive to the user, this watchdog will fire.

If the application thread watchdog is enabled, and that thread quits processing windows messages for the specified amount of time, then “Program Interlock” will be asserted on the station. This will have the effect of shutting down any test that is running.
To enable or disable the watchdog, the method `enableMainThreadWatchdog(true)` must be called within the thread to be watched. The method `setMainThreadWatchdogTimeout()` can be used to adjust the timeout period. The default is 15 seconds.

**Note**
Currently, the actual watchdog timeout is the value set by `setMainThreadWatchdogTimeout()` times the number of applications connected to the station. This may change in a future release.

The method `setTemporarilyMainThreadWatchdogTimeout()` will change the timeout value for one interval. Once that interval is done, the timeout will return to the previous value.

Sometimes, an application thread may perform some long operation that it knows will keep the thread out of a windows message loop for a long time (e.g., reading or writing a large file). When the user performs an “Open” or “Save” operation, he/she expects the application to be unresponsive until that operation is complete. The application can use `enableMainThreadWatchdog(false)` to temporarily disable the watchdog, and enable it again when the operation finishes. The class `AutoEnableMainThreadWatchdog` provides a convenient way to do this. Constructing an `AutoEnableMainThreadWatchdog` instance on the stack will disable the watchdog. It will automatically be re-enabled when that instance goes out of scope.

### Related RtSystem Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setMainThreadWatchdogTimeout(unsigned long timeout, MStatus *status = 0)</td>
<td>Sets the main thread watch dog timeout.</td>
</tr>
<tr>
<td>void setTemporarilyMainThreadWatchdogTimeout(unsigned long timeout, MStatus *status = 0)</td>
<td>Temporarily sets the main thread watch dog timeout.</td>
</tr>
<tr>
<td>boolean enableMainThreadWatchdog(boolean state,MStatus *status = 0)</td>
<td>Enables or disables the main thread watchdog.</td>
</tr>
</tbody>
</table>

### 3.1.4 Other system-wide settings

The `RtSystem` object provides the application access to the following system-wide information. The application cannot change any of these attributes. It can only read it. This information is originally specified either in the HWI file, or in how the system is loaded.

- **getSystemClockRate()**: returns the primary clock rate in the machine (in Hz). This is the clock rate usually used for control, and normal data acquisition. See Section 8.2 on page 98 for more information about system rates.

- **getHighClockRate()**: returns the high clock rate (in Hz). This clock is used for RPC data acquisition.

- **simulation()**: Returns TRUE, if the machine connected to is a simulated machine. A simulated machine is a software process running on the workstation that behaves somewhat like a real system. It is useful for developing tests off-line, doing demonstrations, and training.

- **getVelocityLimiterOption()**: The system has an option to be CE-compliant. One of the features of CE-compliant systems is that they must provide a manual operator control that will not move the actuator faster than 10 mm/sec. If this feature is included, this method will return TRUE. See Section 6.4.2 on page 51 for more information about CE-compliant velocity limiting.
3.2 Connection to the station (RtStation)

The RtStation class models a single station in the machine. It serves as the application’s window into that station. As such, it contains two types of attributes:

1. Characteristics of this application’s connection to the station, and
2. Characteristics of the station itself.

The first topic is discussed in this section; the second is discussed in Section 3.2.5 on page 17.

A program connects to a station with the method:

\[
\text{RtStation *rtStation = RtStation::find(stationName, appName, appType, retStatus);} \\
\text{RtStation *rtStation = RtStation::find(stationName, appName, appTolerance, appType, retStatus);} \\
\]

Where:

- stationName — Identifies the station.
- appName — Is a label for the application.
- appType — Determines what type of access the application will have on the station.
- AppTolerance — Determines whether the application can tolerate dynamic changes to the station resource lists.

3.2.1 Application Name

A station contains a list of applications that are connected to the station. Each application is referenced by an applicationName. These names must be unique on the station. If a second application connects to a station with the same name as one already connected, its application name will be modified by appending a number to it. An application can query what name it was connected as via getAppName().

Although not required, the application name will usually correspond to what is placed in the title bar of the main window of the application. Thus, it is normal for an application to query the station for the name it was actually connected as, and update its title bar appropriately.

An application can change its name with setApplication() or setAppName(). This will go through the same name mangling discussed above. Changing the name of one application will never cause another application’s name to change.

The method getAppListParm() will return the list of all applications connected to the station.

The method getID() returns an integer id for the application that is unique for all applications connected to the system. This is not needed by typical applications, but is available.

Related RtStation Methods:

- static RtStation * find ( const Text * stnName , const Text * appName , AppTypes appType = monitorApp , MStatus * = 0 )
- static RtStation * find ( const Text * stnName , const Text * appName , boolean appTolerance, AppTypes appType = monitorApp , MStatus * = 0 )
### 3.2.2 Application Types

When an application connects to a station, it uses the **AppType** to specify what it intends to do on the station. This system has a relatively unprotected interface. It is designed to allow custom applications maximum flexibility and accessibility. Therefore, it is difficult to limit what an application can do. Thus, the access type of an application is more an agreement, rather than an enforced limitation.

The defined application types are:

- **managerApp** — There is at least one manager for a station (typically there is only one, the **Station Manager**). The manager(s) must remain active for the life of the station. A manager application cannot reassign this privilege to any other application. A manager for a station loads, and (explicitly) unloads the station. If that manager terminates, the station will initiate its unload sequence (See Section 3.3.5 on page 21).

- **controlApp** — Control applications are not critical to the existence of the station, but they are an integral part of the test being performed. They will typically drive some or all of the actuators and digital outputs, and determine the behavior of the program state machine (i.e., cause state changes in response to RUN/HOLD/STOP buttons). If a control application terminates abnormally, the station interlock will be tripped, but the station will remain functional.

- **monitorApp** — Monitor applications are interested in watching what is going on, but typically do not control command channels, or the program state machine. They may create data acquisition and detection objects, connect to events, etc. When a monitor application terminates abnormally, the test may continue to run (controlled by some controlling application).

- **summaryApp** — Summary applications are the same as monitor applications, except that the station may be unloaded while the application is still connected. This is intended for system status applications, either locally, or over a network.

It is important to the operator that applications conform to the intent of their declared application type. The system will not fail if this is not done, but the operator may become confused.

The application type can also be changed at any time by calling **setApplication()**.

The method **getAppType(appName)** can be used to determine the application type for any application connected to the station.

#### Related RtStation Methods:

- static RtStation * find ( const Text * stnName , const Text * appName , AppTypes appType = monitorApp , MStatus * = 0 )
- static RtStation * find ( const Text * stnName , const Text * appName , boolean appTolerance, AppTypes appType = monitorApp , MStatus * = 0 )
- void setApplication ( const Text * appName , AppTypes appType , MStatus * status = 0 )
3.2.3 Application tolerance to named resource changes

Most applications are written to assume that the lists of named station resources (channels, signals, actions, etc) remain constant for the life of the application. These applications read these lists when they start, and are not “tolerant” of these lists changing.

Historically, once a station is loaded, the lists of named resources have remained constant. Version 3.0 introduced the “Configuration” access level at which some named resources can be created and destroyed. Entering “Configuration” access level puts the station into “Configuring” loading state (See Section 3.3.1 on page 17). The system will make sure that only “tolerant” applications are connected to the station whenever the station is in configuration mode.

Most test applications do not specify this attribute, which uses the default of FALSE.

The logic involved in writing an application that is tolerant of station resource list changes is not described in this document. It is more complicated than it may seem, because the application needs to be involved in forcing the lists returned by `getAllObjects` to update.

---

### Related RtStation Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtStation * find</td>
<td>(const Text * stnName, const Text * appName, boolean appTolerance,</td>
</tr>
<tr>
<td></td>
<td>AppTypes appType = monitorApp, MStatus * status = 0)</td>
</tr>
<tr>
<td>void setApplication</td>
<td>(const Text * appName, AppTypes appType, boolean appTolerance, MStatus * status = NIL)</td>
</tr>
<tr>
<td>void setApplicationTolerance</td>
<td>(boolean appTolerance, MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean getApplicationTolerance</td>
<td>(MStatus * status = 0)</td>
</tr>
<tr>
<td>const GenList * getIntolerantAppList</td>
<td>(MStatus * status = 0)</td>
</tr>
<tr>
<td>AppTypes getAppType( const Text * appName, MStatus *status = 0 ) const</td>
<td></td>
</tr>
</tbody>
</table>

3.2.4 Disconnecting from a station

An application can explicitly disconnect from a station by destroying the `RtStation` object it acquired when connecting to it. This will do an orderly disconnection from the station.

An application is implicitly disconnected from all connected stations when it terminates. This is true for normal exit, and any type of application crash (including power failure on the PC). These two types of termination have largely the same effect.

- Based on the application type, the station may initiate an unload (`managerApps`), or assert station interlock (abnormal disconnection of `controlApps`).
- All the objects on the station that are owned by this application will be destroyed (See Section 3.3.2 on page 18 for more information about object ownership).
- For explicit disconnection, all references to the station’s resources within this application will be destroyed (See Section 3.3.3 on page 19). When disconnecting implicitly, the application memory has gone away, so this step is not necessary.
3.2.5 COM Interface Support

Starting in V3.0, applications can communicate directly with the Station Manager through a set of Component Object Model (COM) interfaces. That API is described in the separate document *Integrating Applications with Station Manager*. However, the RtStation class has a few methods that support that interface.

**Note**

At this time, this capability is only intended for internal use by MTS application developers. It is not required to write a test application.

If an application is going to use COM to communicate with Station Manager, it will get an interface for the Station Manager application by calling `getStMgrInterface()`.

To use some parts of the COM API, the application must register its interface with the station via `storeAppInterface()`. This allows the Station Manager to initiate communication.

### Related RtStation Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getStmgrInterface</code></td>
<td><code>const IID &amp; iid, void **ppv, MStatus * s = 0</code></td>
</tr>
<tr>
<td><code>storeAppInterface</code></td>
<td><code>const IID &amp; iid, IUnknown * pUnk, MStatus * s = 0</code></td>
</tr>
<tr>
<td><code>clearAppInterface</code></td>
<td><code>MStatus * s = 0</code></td>
</tr>
<tr>
<td><code>getAppInterface</code></td>
<td><code>const Text * appName, const IID &amp; iid, void **ppv, MStatus *s=0</code></td>
</tr>
</tbody>
</table>

3.3 Station Overview (RtStation)

Stations contain the following major attributes:

- Station load state
- General lists of objects in the station.
- Station log support
- Station interlock support
- Program interlock support
- Time Signal manipulation
- Power status control
- Station program state support
- Stop-action notification support
- Scheduler overrun detection

These will be described separately in the following sections.

3.3.1 Station Load States

The life cycle of a station is encapsulated in the state machine `StationLoadStates`. The possible states are:
**STN_UNKNOWN** — Requesting the state of a station that does not exist will return this value.

**STN_CREATED, STN>Loading** — Loading the station is in progress. It is not yet functional. Application connections will not yet be accepted (An error will be returned).

**STN_READY** — Loading is complete. The station is functional. Connections from additional applications will be accepted.

**STN_CONFIGURING** — The station is in “Configuration” access level. Only applications tolerant of station resource list changes will be allowed to connect to the station.

**STN_UNLOADING** — Unloading the station has been initiated. Further application connections will not be accepted (an error will be returned). The station interlock will be set, and will not be reset-able. The station will complete unloading when all non-summary applications have disconnected themselves.

The Station Manager is responsible to get the station up to the **STN_READY** state. Once there, other applications can connect to it and perform tests. Applications cannot connect to a station until it is in the **STN_READY**. Applications may wait for the station to become ready by polling the station’s state with `getStationStateByName()`.

A station cannot be unloaded until all non-summary applications have disconnected themselves. The Station Manager will not allow the operator to try to unload the station until this condition is met.

If the Station Manager (or other manager application) crashes, the station unload process will be initiated (See Section 3.3.5 on page 21).

<table>
<thead>
<tr>
<th>Related RtStation Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>StationLoadStates loadState ( MStatus * status = 0 ) const</td>
</tr>
<tr>
<td>boolean uponStnLoadChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean disconUponStnLoadChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean okToUnload ( MStatus * status = 0 ) const</td>
</tr>
<tr>
<td>(Used by Station Manager)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtSystem Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>StationLoadStates getStationStateByName ( const Text &amp; stnName , MStatus * status )</td>
</tr>
</tbody>
</table>

### 3.3.2 Named vs. Unnamed Objects

A station contains a large collection of objects. These include control channels, signals, hardware resources, interlock control, etc. Some of these are created by the station; others are allocated from the system when the station is created.

All objects on a station have a name field, and an owner. When an object is created, it is given a name (which may be blank) and an owner. These two attributes determine how the object is managed.

**Named, and Owner is the Station** — Named objects that have the station specified as their owner become public objects on the station. They are visible to all applications connected to the station. They typically stay around until the station is unloaded. Typically, named objects are created by the Station Manager and exist for the life of the station. Some examples of named objects include channels, segment generators and signals.
Starting in Version 3.0, Station Manager can add, delete or change the name of certain public objects while the station is in “Configuring” state. Tolerant applications must respond to these changes.

Unnamed, and Owner is the Station — Unnamed objects that have the station specified as their owner actually become owned by the application that creates them. They stay around until they are explicitly deleted, the application disconnects from the station, or the application terminates (by any means). Unnamed objects cannot be accessed by other applications, since they do not have a name to look up. All applications will typically create some unnamed objects. Examples include data acquisition processes, and limit detectors.

Owner is another object — Many objects are owned by some other object. For example, RtBuffer objects are owned by the RtAcq on which they are created. RtCtrlMode objects are owned by the RtChannel. The lifetime of these objects is controlled by their owner.

Applications do not typically create public objects on the station, but it is possible for them to do so. If they do, those objects will outlive the application. On a subsequent run, they will need to check if the object already exists from a previous or concurrent run of the application.

Object names are represented by the Text class. They have both an internal name and a display name. Station Builder enforces the restriction that within a particular class, the display name of an object will not be the same as the internal or display name of another object. This is intended to minimize potential confusion. For example, if a channel’s internal name is “Channel 1”, its display name may be “Channel 1”, but no other channel’s internal or display name may be “Channel 1”. Also, object names are generally case-insensitive.

In most parts of the system, the internal name is used to identify an object, and the display name is only used for display. However, applications may search for objects by any mechanism they choose.

Note:

Objects of many classes may be named, or unnamed. Some classes do not support named objects. For example, RtProfile objects (commands to the segment generator), logically come from one application, so they do not support names.

3.3.3 Querying for named public objects

The application can query for public objects (those owned by the station) either individually, or in lists. Within a station, an object is uniquely identified by its name and class. For example, an RtFloatSignal can have the same name as an RtChannel, but two RtFloatSignals cannot have the same name.

The method RtStation::findRtObject() is used to find an object of a specified name and class.

The method RtStation::getAllObjects() is used to find all the named objects of a particular class (and any subclass). This method returns a pointer to a GenList containing the objects. Ownership of this list is NOT passed to the caller. The RT interface maintains ownership of the list, and it remains valid for the life of the RtStation. A subsequent call requesting the list for the same class will return a reference to the same GenList. The application must NOT delete or modify this list!
On typical systems, all named public objects are created when the station is created, so the contents of these object lists will not change for the life of the station. The **forceUpdate** parameter on the **getAllObjects** can be set to **TRUE** to force the list to be rebuilt, but this is normally not necessary.

**Restriction:**

Version 3.4 contains a restriction where **findRtObject()** can only be called for non-abstract classes. If it is called on an abstract class, it will crash with a nil-pointer error. The abstract classes are:

- **RtAnalogHWR**, **RtActionInfo**, **RtBuffer**, **RtCtrlr**,
- **RtTrigger**. This restriction may be relaxed in a future version.

Many classes also have a static method for finding an instance, given a name. For example:

```c
static RtFloatSig *RtFloatSig::find(RtStation &station, const Text *name, MStatus*status)
```

These methods perform the same operation as **RtStation**:findRtObject().

An object is generally on more than one list. For example, if the application queries for the list of **RtConditioner** objects, and the list of **Rt497acCond** objects, all the 497 ac conditioner objects will be on both lists. That is, both lists will contain a reference to the same **Rt497acCond** instances. The conditioner list will also contain references to other types of conditioners as well.

Applications use these methods to access most of the components within the station.

<table>
<thead>
<tr>
<th>Related RtStation Methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtObject * findRtObject ( const VClass * daClass, const Text &amp; rName , MStatus * status )</td>
</tr>
<tr>
<td>GenList * getAllObjects ( const VClass * daClass, int forceUpdate = FALSE, MStatus *status = 0)</td>
</tr>
</tbody>
</table>

### 3.3.4 Explicit Support for other components

While most **RtStation** class components are generically accessed through the above find/get methods, a few capabilities are accessed using explicit methods in **RtStation**. The reasons for this vary from one component to another. These capabilities will be discussed in more detail in other chapters.

- **Time Signals** (See Section 4.1.6, page 31)
- **Actions** (See Section 5.2, 38)
- **Segment Generator Composite Program State** (See Section 7.4.5, page 63)
- **Notification callbacks for stop-actions** (See Section 7.4.5, page 63)
- **Station Log** (See Chapter 10, page 142)
- **Program and Station Interlocks** (See Chapter 11, page 149)
- **Power control** (See Chapter 11, page 149)
- **Station Test State** (See Chapter 12, page 157)
• Remote Station Controller access (See Chapter 13, page 161)

3.3.5 Unloading a Station

The unloading of a station is initiated either by a manager application explicitly destroying the `RtStation`, or by a manager application ending or terminating abnormally.

When a station unloads, it goes through the following sequence:

1. Assert station interlock. This interlock will remain asserted for the duration of the station unload procedure.

2. Switch to `STN_UNLOADING` state. This will generate a “station load changed” event. Any application can be notified of this event by connecting an action using `uponStnLoadChg()`. Once in this state, additional application connections will not be accepted.

3. Wait for all applications to disconnect themselves from the station. This will wait indefinitely for manager, control, and monitor application. It will only wait 30 seconds for summary applications.

   Non-summary applications are responsible for disconnecting themselves from the station. This allows them to extract any pertinent data or state information before terminating. Depending on the application, this may require operator intervention.

4. Disconnect remaining summary applications. This is done by breaking that application’s link with the system. To the application, this will look as if the system went down. To the system, this will appear as if the application went away.

   Properly written summary applications will disconnect themselves within the time-out period, thus avoiding being disconnected from all the stations.

5. Dispose of all the objects owned by the station.

6. Return reserved objects to the system for subsequent use by another station.

7. Disposes of itself, removing itself from the list of loaded stations.

Depending on the load on the system, this process may take many seconds (even if it doesn’t have to wait for any applications). During this time, the station is unavailable to be connected to, but any attempt to re-load another station to use the same resources will fail (since they have not been de-allocated yet).

3.3.6 Scheduler overrun detection

If the embedded processor in the machine becomes too busy, it may have to discard some of the periodic interrupts that perform the data acquisition and control. In many applications, occasionally missing an interrupt does not affect the test. The machine will allow the test to continue with occasional overruns. However, it will generate an interlock in either of these conditions:

1. The machine misses three interrupts in a row, or

2. The machine misses a certain percentage of the interrupts in one second. For most processors this limit is 5%. On some slower processors, the limit is set higher.
For some applications, missing any interrupts is unacceptable. **RtStation** contains these methods for finding out what the overrun count is, and being notified when an overrun occurs. The method `getOverrunCount()` returns the current count. The overrun count is a *system* count. It starts at zero when the system is loaded, and is not subsequently reset.

After `uponSchedOverrun()` is called, the first time that an overrun occurs the action will be performed. The action will only be fired again after `reEnableSchedOverrunAction()` has been called. This keeps the system from flooding the application with actions.

### Related RtStation Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned long getOverrunCount ( void ) const</code></td>
<td>Returns the current overrun count.</td>
</tr>
<tr>
<td><code>void uponSchedOverrun ( RtAction * anAction , MStatus * status = NIL )</code></td>
<td>Upon receipt of an overrun, performs the action.</td>
</tr>
<tr>
<td><code>void disconUponSchedOverrun ( RtAction * anAction , MStatus * status = NIL )</code></td>
<td>Disconnects the action when an overrun occurs.</td>
</tr>
<tr>
<td><code>void reEnableSchedOverrunAction ( RtAction * anAction , MStatus * status = NIL )</code></td>
<td>Re-enables the action after an overrun.</td>
</tr>
</tbody>
</table>

### 3.3.7 Other attributes

Methods exist for applications to query for the name of the station, the path to the CFG file, and the name of the currently loaded parameter set. Since **RtStation** is a subclass of **Parameter**, its name can be queried by any number of methods in that class. The station name and path are constant for the life of the station. The parameter set name may change, but there is no mechanism to be notified of the change.

### Related RtStation Methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Text getConfigPath(MStatus * = 0) const</code></td>
<td>Gets the path to the configuration file.</td>
</tr>
<tr>
<td><code>Text getParamSetName(MStatus * = 0) const</code></td>
<td>Gets the name of the currently loaded parameter set.</td>
</tr>
</tbody>
</table>

### 3.4 Multi-controller FlexTest SE Workstations

Most 793-based systems only support being connected to one controller. However, 793 System Software does support a single workstation controlling multiple automated FlexTest SE controllers at the same time. In this scenario, each registered controller has its own subdirectory that contains its own HWI and configuration files. In addition, it has its own definitions in the Windows registry.

So in this environment when an application starts, instead of connecting to the **RtSystem** and then deciding what available station to connect to (as described in Section 3.1.2 on page 12), the application must first choose which system (i.e., controller) to connect to, and then choose the station on that system.

### 3.4.1 Controller/Station Selection Dialog (MTSBoxSelect)

For most applications, selecting a controller and station can be done with the **MTSBoxSelect** class. Generally, the **MTSBoxSelect** class displays a dialog allowing the operator to select the system and station to connect to.

Example 1 is a code segment from the **MyDemo** application. It demonstrates the typical use of the **MTSBoxSelect** class for applications that need to connect to an existing station. In this case, it is an MFC application.

```c++
// Includes and definitions required for MTSBoxSelect
#include "MTSBoxInfo.h"
```
```cpp
#include "MTSBoxSelect.h"

RtStation *CMydemoApp::FindAndLoadStation(CWnd* pParent, const char *appName, MStatus *sp)
{
    // if there are registered controllers, this will choose the controller/station
    MTSBoxInfo boxInfo;
    if (boxInfo.getBoxesInstalled()->getCount() > 0) {
        Text boxName; // if command line specifies a controller name, put it here.
        Text stnName; // if command line specifies a station name, put it here.
        MTSBox *pBox = MTSBoxSelect::dialog( AfxGetApp()->m_hInstance, pParent->m_hWnd, boxInfo, boxName, stnName, sp);
        if (!pBox)
            return 0; // error or cancelled, return the status.
        RtSystem::setBoxName((Text&)pBox->parmName());
        return RtStation::find(&stnName, &Text(appName), controlApp, sp);
    }
    // if there are no registered controllers, this will choose the station.
    else
        return FindSingleControllerStation(pParent, appName, sp);
}
```

Example 1 — MyDemo's use of the MTSBoxSelect class

Notes:

1. If there aren’t any registered boxes, handle querying for the station like any other product. (See the MyDemo code for more detail.)

2. If the application is designed to allow the controller name or station name to be specified on the command line, then these specified arguments should be passed into this dialog in these variables.

3. The method RtSystem::setBoxName(name) establishes a specific controller as the controller that this application will reference. When the machine interface goes to the registry to determine what HWI file to use, what default IP address to use, or where to find the configuration files, the information relative to that controller will be returned. Once this call is made, everything about RtSystem discussed in Section 3.1.1 on page 11 will operate as if that were the only controller on the system. Thus, there is no reason to explicitly call RtSystem::find(), since calling RtStation::find() on the next line will implicitly call it.

The MTSBoxSelect::dialog() may not actually display the dialog. If there is only one controller connected to the workstation, or if a controller is explicitly specified, then the operator doesn’t need to make a choice. In situations like this, the dialog() call will return immediately with a valid station name. If there is no valid selection, it will display the dialog, so the user can determine why none of the controllers are appropriate to connect to.

There is a second form of MTSBoxSelect::dialog() that is designed to be used by Allegris classes. The only difference is in how the parent window is specified.
Note

The classes and methods described in this section are still evolving, and will probably change in a future release. However, to support interfacing with a FlexTest SE controller, applications need to include code similar to what is in Example 1.

### Related RtSystem Methods:

```cpp
static void setBoxName(Text &boxName, MStatus *status=0)
```

### Related MTSBoxSelect Methods:

```cpp
static MTSBox *dialog(  HINSTANCE  hInstance,
                        HWND  parent,
                        MTSBoxInfo  &info,
                        Text  &boxName,
                        Text  &stnName,
                        MStatus  *sp=0);

static MTSBox *dialog(  VWindow  *parent,
                        MTSBoxInfo  &info,
                        Text  &boxName,
                        Text  &stnName,
                        MStatus  *sp=0);
```

### 3.4.2 Lower-level Multi-controller Model Classes

**MTSBoxInfo & MTSBox**

The `MBoxSelectDlg` class described in the previous section makes use of the `MTSBoxInfo` and `MTSBox` classes. Most applications do not need to use these classes directly. However, if `MBoxSelectDlg` is not adequate for your application, this information is available.

Creating an `MTSBoxInfo` will create a list of `MTSBox` objects reflecting the controllers that are registered on the workstation. This list is available by calling `MTSBoxInfo::getBoxesInstalled()`.

At this point the `MTSBox` objects in the list reflect the information in the registry, but the attributes that are stored in the controller itself remain in their “undefined” state. Calling `MTSBox::updateFrBox()` will update the attributes to reflect the current state of the controller. None of these attributes are automatically updated. If you want them refreshed, you need to explicitly request the update.

Calling `MTSBoxInfo::update()` is a convenient way to update all the controller information. It will update the list of controllers from the registry (possibly adding or deleting some), and then will call `MTSBox::updateFrBox()` on each registered controller.

Each controller has the following attributes:

- **Name** — This is the box name, and is always available through `Parameter::parmName()`.
- **IP Address** — This comes from the registry, so it is always available.
- **Boot Mode** — (Available after calling `updateFrBox()`) The available boot modes are:
Box State —  (Available after calling \texttt{updateFrBox}) The available box states are:
\begin{itemize}
  \item \texttt{kBoxStateUndefined} — Update hasn’t been done yet
  \item \texttt{kNoBox} — Controller doesn’t answer
  \item \texttt{kIpExists} — (unused)
  \item \texttt{kNotLoaded} — SYSLOAD hasn’t been run yet
  \item \texttt{kLoaded} — SYSLOAD completed successfully
  \item \texttt{kLoadedWithError} — SYSLOAD failed
  \item \texttt{kVersionError} — Onboard software version doesn’t match the software on the workstation.
\end{itemize}

Box Type —  In this version, this will always be \texttt{kIndependent}.

Station Name List —  (Available after calling \texttt{updateFrBox}) This is a list of the names of the stations on the controller.

The method \texttt{MTSBox::okToUse(option)} can be used to determine if a controller is in a state that is appropriate for the application to use. The option gives an indication of what the application will use the controller for.

\begin{itemize}
  \item \texttt{kBoxAny} — Any registered controller (whether it is connected or not) will be okay to use. This is used by applications like Station Builder.
  \item \texttt{kBoxSelectExist} — Any registered controller that is connected and answers up will be okay to use. Tools like SYSUTIL2 use this option.
  \item \texttt{kBoxSelect} — Any registered, connected controller that is in Automation mode but doesn’t have a station loaded will be okay to use. This is used by application like Station Manager or System Loader.
  \item \texttt{kStationSelect} — Any registered, connected controller that is in Automation mode and has a station running on it will be okay to use. This is used by applications that want to connect to an existing station like Basic TestWare or MPT.
\end{itemize}

The method \texttt{MTSBoxInfo::setBoxName(name)} is used by applications like Station Builder that are not going to connect to the controller. Like \texttt{RtSystem::setBoxName()} it sets up the registry access functions to work relative to the specified controller. Only use \texttt{MTSBoxInfo::setBoxName()} if you are NOT going to otherwise use \texttt{RtSystem} or \texttt{RtStation} objects.

\begin{tabular}{|l|}
\hline
\textbf{Related MTSBoxInfo Methods:} \\
\hline
\texttt{MTSBoxInfo()} \hspace{1cm} \texttt{virtual ~MTSBoxInfo()} \\
\texttt{void update()} \\
\texttt{GenList * getBoxesInstalled()} \\
\texttt{int getLoadedStations()} \hspace{1cm} // returns the total number of stations across all the controllers. \\
\texttt{bool setBoxName(const char * pName = 0)} \\
\hline
\end{tabular}
**Related MTSBox Methods:**

- MTSBox()
- virtual ~MTSBox()
- void updateFrBox()

- Text * getAddress()
- BootMode getBootMode()
- BoxState getBoxState()
- BoxType getBoxType()
- GenList * getStationNameList()

- bool okToUse(BoxSelect option)
Chapter 4
Signals

Signals are a fundamental abstraction used throughout a station. A Signal is a named source or sink for a real-time stream of data. It is analogous to a test point on an analog system. Signals are typically used when collecting or viewing real-time data.

Some data streams that are available as signals include:
- Analog input channels (high-level, and conditioned)
- Analog output channels (high-level and valve drivers)
- Controller test-points (e.g., error, command, optimized command)
- Segment generator counters
- Time signals
- General purpose Digital Inputs
- General purpose Digital Outputs

The RtSignal class is the abstract base class for all signals. Most of the interface is defined in this class.

The RtFloatSig class is used for signals whose value is represented by a 32-bit floating-point number. Most signals in the system fall into this category.

The RtIntegerSig class is used for signals whose value is represented by a 32-bit signed integer number. This principally includes digital I/O, and segment counters.

4.1 Signal Attributes

4.1.1 Querying the value of a signal

A signal always has a current value. This value is (in general) always changing. This current value can be accessed by the application in a variety of ways:

**Query current value** — The application can call the `value()` method, or get a Number object through `getValueParm()`. These will get the current value from the machine. The value will be converted to a double-precision floating-point number.

**Query voltage value** — (RtFloatSig only) For floating point signals, the value can be queried in the Volts dimension. The system assumes that full scale is +/- 10 volts. The methods `volts()` and `getVoltsParm()` are used to get this information.

**Query long integer value** — (RtIntegerSig only) The method `RtIntegerSig::longValue()` can be used to get the current value as a long integer.

**RtSignalList** — This object will query a list of signals for their values at a single instant in time. It will return an array of values (one for each signal). This may be used with RtFloatSig or RtIntegerSig objects (but not both at the same time). See Section 4.3.1 on page 34 for more information on using the RtSignalList class.
Polled query — The station contains a centralized mechanism for periodically polling the values of signals. See Section 4.3.2 on page 35 for more information about this mechanism.

Data Acquisition — Data acquisition objects collect data from signals, and buffer the data to the application. (See Chapter 8 starting on page 97 for more information about data acquisition)

Detectors — Limit detectors monitor signals, and react to changes in their value. There are different limit detector classes for both Floating point and integer signals. (See Chapter 9 starting on page 129 for more information about detectors.)

<table>
<thead>
<tr>
<th>Related RtSignal methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual Number * getValueParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual double value ( MStatus * = NIL )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtFloatSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number * getVoltsParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>double volts ( MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtIntegerSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>long longValue ( MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

4.1.2 Setting the value of a signal

Most signals are tied to some real-time data stream that provides a continuously changing value for the signal. However, a few signals are themselves the originator of the values. The most common case for this is digital output.

The current value can be modified either through the Number parameters, or via explicit methods. Again, only RtFloatSig supports using Volts as the dimension.

RtIntegerSig also supports the toggle() and pulse() methods, which are designed for Boolean signals like digital outputs.

<table>
<thead>
<tr>
<th>Related RtSignal methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual Number * getValueParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual void value ( double value, MStatus * = NIL )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtFloatSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number * getVoltsParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>void volts ( double value, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtIntegerSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void longValue ( long value, MStatus * status = 0 )</td>
</tr>
<tr>
<td>void toggle ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>void pulse ( double interval, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
4.1.3 Notification of Value Change (RtIntegerSig only)

*RtIntegerSig* has the option to notify the application when the value of the signal changes. This is normally disabled. When disabled, the application only knows if the value changes by polling for the value (see Section 4.1.1 on pages 27).

This feature is enabled by calling *start()* on the signal. Once the signal is started, any change in the signal’s value will be propagated into the *Number* object returned by *getValueParm()*(). The application can connect a callback to this *Number* to be notified of the change.

In addition, the application can use *connect()* to connect an *RtAction* to be notified when the value changes.

**Note:**

Calling *start()* on an *RtFloatSig* will have no effect.

| Related RtTask methods (base class of RtSignal): |
|-----------------|-------------------|
| void start ( MStatus * status = 0 ) |
| void stop ( MStatus * status = 0 ) |

| Related RtSignal methods (base class of RtIntegerSig): |
|-----------------|-------------------|
| virtual Number * getValueParm ( MStatus * status = 0 ) |

| Related RtIntegerSig methods: |
|-----------------|-------------------|
| void connect ( RtAction * anAction , MStatus * status = 0 ) |
| void disconnect ( RtAction * anAction , MStatus * status = 0 ) |

4.1.4 Signal Dimension and Range Information

A variety of attribute information about the dimension, range, and resolution for a signal is available:

**Dimension and Display Unit** — These are usually accessed through the range objects discussed in this section. The “display unit” represents the selected unit for display in the calibration for this signal. As everywhere else, the values retrieved are always in system units.

**Note:**

In V3.2, the methods *dimension()* and *unit()* return a *Text* objects that do not contain any natural language label. To get the appropriate label, use these strings to create a *DimId* or *UnitId*, and query that object for its name.

**Full Scale** —

The term “full scale” refers to the calibrated range of the signal. This is sometimes referred to as +/- 100%. This is available through the method *getFSParm()*(). This returns a *SubRange*. It also contains the dimension, display unit, resolutions, etc. For some signals (e.g. command signals), this information can change dynamically. If necessary, applications should hook callbacks to this *SubRange* and react to these changes. The methods *lowerFullScale()* and *upperFullScale()* also return this information.
Saturation —
This represents the maximum and minimum values that the application can expect to see on the signal. This is often more than +/- 100%, because we calibrate our signals to have some percentage of over-range capability. Depending on the conditioner, this may be up to 130% of the full-scale value.

The saturation range is accessible through `getValueParm()`–`getRange()`. It is also accessible through the `attribute()` method. The “hard” and “soft” limits are always the same value.

Extent —
On signals that have a sensor assigned to them, the extent is the union of the full-scales of all of the ranges defined in the sensor. The extent will not change when a range-change is performed through Station Manager or another application. This attribute is available through `getExtentParm()`.

On signals that do not have a sensor file assigned (or do not support a sensor file), the extent will contain the same information as the full-scale.

Extent Saturation —
This is represents an approximation of the union of the saturation min/max of each range. It is actually 130% of the extent. This is accessible through `getExtentParm()`–`getRange()`.

Applications should not change any of this dimension and range information. These types of changes are done indirectly, through the object that controls the signal.

<table>
<thead>
<tr>
<th>Related RtSignal methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual Number * getValueParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual const SubRange * getFSParm ( MStatus * = NIL )</td>
</tr>
<tr>
<td>virtual const SubRange * getExtentParm ( MStatus * = NIL )</td>
</tr>
<tr>
<td>virtual Text dimension ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>virtual Text unit ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>virtual double resolution ( MStatus * = NIL )</td>
</tr>
<tr>
<td>virtual double range ( LimitType newLimit , LevelType newLevel , MStatus * = NIL )</td>
</tr>
<tr>
<td>virtual void attribute ( Text * dim , Text * unit , double * res , double * sLower , double * sUpper , double * hLower , double * hUpper , MStatus * = NIL )</td>
</tr>
</tbody>
</table>

4.1.5 Signal type and source

Signals are not themselves a data stream; they are a “tap” on a data stream generated or consumed by some other object. This section describes that relationship.

Note:
Most applications do not need to be concerned with this relationship, and a first reading can skip this section.

Signals come in two types: SOURCE and SINK. This type is available through the method `type()`.

SOURCE —
A source signal taps into an object that is producing a data stream. For example, an Analog Input Hardware resource produces a stream of data from an A/D converter. The signal representing this stream is a source signal.
A sink signal taps into an object that is consuming a data stream. For example, a readout is an analog output hardware resource represents a D/A. This object is connected to something else that produces the data stream. The readout is consuming the data.

A signal has two pointers that may reference the objects that are the source and sink of the data stream. Either or both of these pointers may be NULL. In general, for SOURCE signals, the `source()` method will return a pointer to the object that controls this signal, and `sink()` will return null. For SINK signals, the `sink()` will return a pointer to the object that controls this signal, and `source()` will return NULL.

The following table shows various signals in a system, and how these attributes are set:

<table>
<thead>
<tr>
<th>Usage</th>
<th>type()</th>
<th>source()</th>
<th>sink()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Input signal</td>
<td>SOURCE</td>
<td>RtAnalogInputHWR</td>
<td>0</td>
</tr>
<tr>
<td>Command signal</td>
<td>SOURCE</td>
<td>0 *</td>
<td>0</td>
</tr>
<tr>
<td>Channel's Output signal</td>
<td>SINK</td>
<td>0 *</td>
<td>RtAnalogOutputHWR</td>
</tr>
<tr>
<td>Calculated Channel’s output</td>
<td>SINK</td>
<td>0 *</td>
<td>0</td>
</tr>
<tr>
<td>Readout signal</td>
<td>SINK</td>
<td>0</td>
<td>RtAnalogOutputHWR</td>
</tr>
<tr>
<td>Digital Input signal</td>
<td>SOURCE</td>
<td>RtDigitalInputHWR</td>
<td>0</td>
</tr>
<tr>
<td>Digital Output Signal</td>
<td>SINK</td>
<td>0</td>
<td>RtDigitalOutputHWR</td>
</tr>
<tr>
<td>Calculated Input Signal</td>
<td>SOURCE</td>
<td>RtCalculation</td>
<td>0</td>
</tr>
<tr>
<td>Calculated Output Signal</td>
<td>SOURCE</td>
<td>RtCalculation</td>
<td>RtAnalogOutputHWR or NULL</td>
</tr>
</tbody>
</table>

* Currently, signals that are sourced from control channels or modes still return NULL. This may be changed in a subsequent version.

**Related RtSignal methods:**

- `virtual RtObject * source ( MStatus * status = NIL )`
- `virtual RtObject * sink ( MStatus * status = NIL )`
- `virtual CppSignal type ( MStatus * status = NIL )`

### 4.1.6 Alternate data streams

Up to this point, we have described a signal as a named tap on a data stream. However, some signals have more than one type of data stream associated with them. These alternate data streams always originate from the same source, but may be sampled or filtered differently. There are three types of streams, identified by the enumeration `DataStreamTypes`:

**NORMAL_DATA** — This stream type is sampled at the system rate. It is available through the facilities described in Section 4.1.1 on page 27.

**HIGH_SPEED_DATA** — This stream type is only available from 493.21B and 493.25 conditioners. The data on this stream is available at some sub-multiple of 49152 Hz. The data acquisition classes described in Chapter 8 are used to collect high-speed data. However, a single data acquisition object cannot collect normal data, and high-speed data at the same time. See Section 8.6 on page 121 for how to collect data from this stream type.

**TIME_HISTORY_DATA** — This stream type is only available from 493.21B and 493.25 conditioners, or from a 498.64 ADDA board. The data on this stream is only available through the Time History data acquisition classes described in Section 0 starting on page 122. This data is digitally filtered to avoid aliasing.
An **RtFloatSig** always supports the normal data stream. If the signal represents data from the applicable hardware, and if the feature is enabled, then it may also support the two other stream types. The method `supportsDataStream()` can be used to determine if a specified signal supports a particular data stream.

### Related **RtFloatSig** methods:

```cpp
boolean supportsDataStream ( enum DataStreamTypes dataStream , MStatus * status = 0 )
```

## 4.1.7 Creating Private Signals

Most applications simply make use of the signals that are built into the station. On rare occasion, there is value in an application creating a private, unnamed signal. This signal will not have a source or sink. It will just hold a single value. The application will be responsible for setting this value into the signal.

This signal can be used wherever a signal can be used, including data acquisition and limit detectors. Another use of a private signal is for gating data acquisition objects, as described in Section 8.5.1 on page 117.

To create the signal (floating point or integer), use the appropriate `make()` method, as shown in the following summary. You may destroy the signal when you no longer need it. It will be automatically destroyed when the application goes away.

### Related **RtFloatSig** methods:

```cpp
static RtFloatSig * make ( RtStation & stn ,
  const Text * name , // Specify NULL
  bool hidden , // Specify false
  CppSignal newType , // Specify SOURCE
  RtObject * source , // Specify NULL
  long srcIndx , // Specify 0
  RtObject * sink , // Specify NULL
  long sinkIndx , // Specify 0
  const Text * newDim ,
  const Text * newUnit ,
  double resolution ,
  double lower ,
  double upper ,
  MStatus * status = NIL )
```

### Related **RtIntegerSig** methods:

```cpp
static RtIntegerSig * make ( RtStation & stn ,
  const Text * name , // Specify NULL
  bool hidden , // Specify false
  CppSignal newType , // Specify SOURCE
  RtObject * source , // Specify NULL
  long srcIndx , // Specify 0
  RtObject * sink , // Specify NULL
  long sinkIndx , // Specify 0
  const Text * newDim ,
  const Text * newUnit ,
```
4.2 Predefined Time Signals

A station has four predefined signals monitoring time:

**Time** — This floating-point signal starts at zero when the station is loaded, and continually increases. Its dimension is “Time”, so the signal value is in seconds. Its default display units follow the default Unit Assignment Set.

**Hourly Rollover Time** — This floating-point signal is based on the same counter as “Time” and is of the same characteristics, except it automatically resets to zero after one hour.

**Running Time** — This floating-point signal only increments when the station test state is not stopped or hold. See Chapter 12 starting on page 157 for a description of the station test state.

**Hourly Rollover Running Time** — This floating-point signal is based on the same counter as “Running Time” and is of the same characteristics, except it automatically resets to zero after one hour.

Methods exist on RtStation to set the timer to a particular value or to zero it. Applications typically zero this timer at the beginning of the test.

The reason for having two signals is that since the **Time** signal is a float, its resolution will degrade as the value becomes large. While the initial resolution is at the system rate, once this value reaches two days the resolution will only be 10 milliseconds, and at 6 months, the resolution will only be 1 second. The **Hourly Rollover Time** is available to provide this added resolution. By collecting both signals and post-processing the information, a high-resolution time stamp can be obtained.

The same discussion is applicable to **Running Time** and **Hourly Rollover Running Time**.

**Note:**

The system contains a time-of-day clock that is used by the message log, but the RT interface does not provide access to it. The current time-of-day is available through the ANSI C runtime library, or the Windows-NT `GetLocalTime()` function.

### Related RtStation methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtFloatSig * xout ( MStatus * status = 0 )</td>
<td>(returns <strong>Time</strong> signal)</td>
</tr>
<tr>
<td>RtFloatSig * out ( MStatus * status = 0 )</td>
<td>(returns <strong>Hourly rollover time</strong> signal)</td>
</tr>
<tr>
<td>void resetTimer ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>void setTimer ( double aValue, MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>RtFloatSig * runningTimeSignal ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>RtFloatSig * rolloverRunningTimeSignal ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>void resetRunningTime ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>void setRunningTime ( double aValue, MStatus * status = 0 )</td>
<td></td>
</tr>
</tbody>
</table>
4.3 Polling signal values

4.3.1 Using the RtSignalList class

The RtSignalList class is used to take an immediate snapshot of the value on several signals. In a way, this is another form of data acquisition. However, it is documented here because it is a separate mechanism and just returns a single snapshot of data.

To use an RtSignalList:

1. Call make() to create the object.
2. Call add() or addList() to add signals to the signal list.
3. Whenever you want a snapshot of the data, call the appropriate getData().

A signal list can handle either RtFloatSig or RtIntegerSig signals, but not both types in the same object (it will return an error if you try to do so).

If a signal is added multiple times, a reference count is incremented and only one entry is actually maintained in the list. The signal must be specified in a remove() or removeList() the same number of times before it is actually removed from the list.

The method getPollingList() returns a const reference to the list. The caller must not directly edit this GenList.

Since the signals added to a signal list can include private, unnamed signals (see Section 4.1.7 on page 32), the GenList returned by getPollingList() uses synthetically generated names for the names of the signals in the list. The signals themselves still have their own names, but if you call GenList::getItemNameAt() on the list returned by getPollingList(), you will get a garbage name.

The getData() methods fill in a client-specified array with a snapshot of values (one for each signal in the list). The client is responsible to call the appropriate method, based on whether the RtSignalList was populated with RtFloatSig or RtIntegerSig objects.

The list may be modified at any time with subsequent add and remove calls. The clear() method will remove everything from the list.

The methods start() and stop() should not be used directly by applications.

<table>
<thead>
<tr>
<th>Related RtSignalList methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtSignalList * make ( RtStation &amp; stn , MStatus * status )</td>
</tr>
<tr>
<td>void getData ( long * data , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void getData ( float * data , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void addList ( const GenList * aList , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void add ( RtSignal * aSig , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void remove ( RtFloatSig * aSig , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void removeList ( const GenList * aList , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void clear ( MStatus * sp = 0 )</td>
</tr>
<tr>
<td>const GenList * getPollingList ( MStatus * sp = 0 ) const</td>
</tr>
</tbody>
</table>
4.3.2 Process-wide signal polling

Some applications (like Station Manager) display polled signal data in several independent views. For these applications, **RtStation** implements a built-in signal polling mechanism to merge the polling requests of these separate views. This mechanism will poll (at 1.5 second intervals) the values of all the signals requested by any of the views, and update a “polled value parameter” (a **Number** object) for each of the signals. Each view can then hook the **Number** object up to an appropriate control.

This is done as follows:

1. Call **RtStation::getPollingSigListParm()** to get a pointer to the station’s **RtSignalList** object.

2. Add the signals of interest by calling **RtSignalList::add()** or **addList()**.

This will cause the station to begin updating the **Number** objects returned by **RtSignal::getPolledValueParm()** and **RtFloatSig::getPolledVoltsParm()**. If the application asks these objects for the value, they return the last polled value (rather than asking the machine for the current value).

When the particular view is no longer interested in polling the signals, it should make the corresponding **remove()** calls.

**Note**

Because **RtSignalList** can only handle one data type at a time, the application must choose to only put **RtFloatSig** or **RtIntegerSig** objects into the station’s polling list.

**Note**

Since this mechanism is dealing with **Number** objects, it should only make these calls from the GUI thread.

<table>
<thead>
<tr>
<th>Related RtStation methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void getPollingSigListParm ( MStatus * sp = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtSignalList methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addList ( const GenList * aList , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void add ( RtFloatSig * aSig , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void remove ( RtFloatSig * aSig , MStatus * sp = 0 )</td>
</tr>
<tr>
<td>void removeList ( GenList * aList , MStatus * sp = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtSignal methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual Number * getPolledValueParm ( MStatus * status = NIL )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtFloatSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual Number * getPolledVoltsParm ( MStatus * status = NIL )</td>
</tr>
</tbody>
</table>
Chapter 5
Actions

An action is a defined list of things to do within the machine. The class RtAction models this concept. A few predefined actions always exist within a station. In addition, through the Station Manager’s action editor, the user can create additional named actions that are available for all applications. Finally, applications can create their own (typically unnamed) actions for use within the application.

Actions can be triggered from many places in the system. In general, any place where the application needs to know about the occurrence of some event, an action can be connected to perform that notification. These include (but are not limited to):

- Limit detectors (floating point and integer),
- Command generation waveform completion,
- Power and program state changes,
- Interlocks,
- Other actions
- Etc.

In addition, an action can be explicitly triggered with the triggerAction() method.

5.1 Predefined and station-defined actions

A station contains a few pre-defined, named actions. The predefined actions are:

Indicate — This action does nothing itself. Many events that can trigger actions (e.g. limit detectors) will log the event if an action is performed, so hooking up this action will cause the event to be logged.

Station Power Off — Turns power off on all HSMs on the station.

Interlock — Asserts the station interlock. This will stop the test and turn power off. See Section 11.2 on page 150 for more information about Station Interlock.

Program Interlock — Asserts the program interlock. This generally stops the test, but leaves power on. See Section 11.1 on page 149 for more information about Program Interlock.

Program Stop — This is equivalent to the operator pressing the “Stop” button on the Station Manager main window. This event will be handled by the controlling application, so it will not be as responsive as Program Interlock.

Program Hold — This is equivalent to the operator pressing the “Hold” button on the Station Manager main window. This event will be handled by the controlling application.

In addition, the station contains all the actions created through the Station Manager’s action editor. The application can get a list of all pre-defined and station-defined actions by calling:

```
GenList *actionList = RtStation::getAllObjects(RtActionCls, FALSE, &status);
```
The only things an application should do with these named actions are to connect them to events (e.g. limit detectors, command generation completion, etc.) or explicitly trigger them with `triggerAction()`.

**Related `RtAction` methods:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static RtAction * find ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</code></td>
<td>Find a named action by name and station</td>
</tr>
<tr>
<td><code>void triggerAction ( MStatus * status = 0 )</code></td>
<td>Trigger the named action</td>
</tr>
</tbody>
</table>

### 5.2 Application defined actions

Applications can create their own *unnamed* actions.

**Note:**

An application may create named actions, but the designer must remember that the station takes ownership of these actions, and they are not deleted until the station is unloaded. Deleting the `RtAction` will only remove the application’s reference—the action will remain in the station. Currently, the Station Manager will not “see” named actions that another application creates.

An `RtAction` is a container that holds a set of `RtCommand` objects. An `RtCommand` object holds the information necessary to perform some operation within the machine. Many of the classes within the RT interface know how to make `RtCommand` objects that perform operations on objects of that class. For example, `RtIntegerSig::getPulseCommand()` will create a new `RtCommand` that will pulse the signal when it is triggered.

Ownership of the new `RtCommand` is maintained by the station. An `RtCommand` can be used multiple times within the same or different `RtCommand` objects. If an `RtCommand` is deleted, it is removed from any actions in which it is placed. As with other unnamed objects, they will be cleaned up if the application terminates.

**Note:**

The naming of these methods is inconsistent with the normal use of the “get” verb. Normally, a “get” method returns a reference to an object without returning ownership of it. For these methods returning `RtCommand` objects, the ownership of the new object is also returned.

Currently, the operations an action can perform are:

1. Trigger another action.
2. Generate a callback into the application (see Section 5.2.1 on page 40).
3. Change power states (off, low, high)
4. Assert the program interlock
5. Assert the station interlock
6. Request a test state change (see Section 5.2.2 on page 41).
7. Pulse, toggle, set or increment the value on an integer signal (usually a digital output)
8. Log a message.
9. Initiate a stop action on the segment generators (see Section 5.2.2 on page 41).
10. Initiate a dynamic mode-switch on a channel (see Section 6.3.2 on page 49).
11. Delay a specified amount of time before processing subsequent commands in an action (see Section 5.2.4 on page 41).
In addition to explicitly using **RtCommand** objects, a variety of alternate `make()` and `add()` methods are available for creating actions and adding operations to them. In general, these are short cuts for creating commonly used actions. Except for generating callbacks (discussed in the next section), all the operations can be added explicitly using **RtCommand** objects.

The method **attachAction()** specifies another **RtAction** that will trigger this action. The following statement specifies that if **actionB** is triggered, it will trigger **actionA**.

```cpp
actionA->attachAction(actionB);
```

The more natural way to do this is to use **RtCommand** objects directly using `getTriggerActionCommand()`.

### Related **RtAction** methods:
- `static RtAction * make ( RtStation & stn, const Text * name, MStatus * status = NIL )`
- `static RtAction * makeHydOffAct ( RtStation & stn, const Text * name, MStatus * status = NIL )`
- `static RtAction * makeInterlockAct ( RtStation & stn, const Text * name, MStatus * status = NIL )`
- `static RtAction * makeProgramInterlockAct ( RtStation & stn, const Text * name, MStatus * status = NIL )`
- `static RtAction * makeTestStateAct ( RtStation & stn, const Text * name, ProgramState aState, MStatus * status = NIL )`
- `void add ( RtCommand * cmd, MStatus * status = 0 )`
- `void remove ( RtCommand * cmd, MStatus * status = 0 )`
- `void attachAction ( RtAction * trigger, MStatus * status = 0 )`
- `void detachAction ( RtAction * trigger, MStatus * status = 0 )`
- `RtCommand * getTriggerActionCommand ( MStatus * status = 0 )`

### Related **RtHsm** methods:
- `RtCommand * getHsmOffCommand(MStatus * = NIL);`
- `RtCommand * getHsmLowCommand(MStatus * = NIL);`
- `RtCommand * getHsmHighCommand(MStatus * = NIL);`

### Related **RtFloatSig** methods:
- `RtCommand * getSetValueCommand ( double value, MStatus * status = NIL )`
- `RtCommand * getStepCommand ( double interval, MStatus * status = NIL )`
- `RtCommand * changeSetValueCommand ( RtCommand * cmd, double value, MStatus * status = NIL )`
- `RtCommand * changeStepCommand ( RtCommand * cmd, double interval, MStatus * status = NIL )`

### Related **RtInterlock** methods:
- `RtCommand * getInterlockCommand(MStatus * = NIL);`

### Related **RtIntegerSig** methods:
- `RtCommand * getPulseCommand(double interval, MStatus * = NIL);`
- `RtCommand * getToggleCommand(MStatus * = NIL);`
- `RtCommand * getSetValueCommand(long value, MStatus * = NIL);`
- `RtCommand * getStepCommand ( long interval, MStatus * status = NIL )`
- `RtCommand * changePulseCommand ( RtCommand * cmd, double interval, MStatus * status = NIL )`
RtCommand * changeSetValueCommand ( RtCommand * cmd , long value , MStatus * status = NIL )
RtCommand * changeStepCommand ( RtCommand * cmd , long interval , MStatus * status = NIL )

**Related RtLog methods:**

virtual RtCommand *getPutCommand( const Text &source, Severities severity, const Text &message, 
MStatus * = NIL);
virtual RtCommand *getPutCommand( const Text &source, Severities severity, const MStatus &statMsg, 
MStatus * = NIL);

**Related RtChannel methods:**

RtCommand * getModeSwitchActionCommand( int modeID, Mstatus *status = 0 )

**Related RtStation methods:**

RtCommand * getDelayCommand( double delay, MStatus *status = 0 )
RtCommand * getStopActionCommand(unsigned long id, MStatus * = 0);
RtCommand * getTestStateActionCommand ( ProgramState aState , MStatus * status = 0 );

### 5.2.1 Generating callbacks to the application

As mentioned above, **RtAction** objects can generate callbacks into the application. However, these are not added via **RtCommand** objects. Rather, explicit **make()** or **add()** methods must be used. Callback operations can be mixed with other commands on the same action.

Often, when **RtAction** objects are used to notify the application of an event, it is the only operation that the **RtAction** performs. An example of this is shown in Example 2.

```cpp
RtAction *act = RtAction::make(*rtStn, 0, BOOL_CB0(this, MyClass, myMethod));

// callback to be invoked when this action is triggered.
boolean MyClass::myMethod()
{
    ...        // return value is unused.
    return TRUE;
}
```

**Example 2 — Creation of a typical RtAction generating a callback**

The return value is not used. An application may also use callbacks that take a single integer parameter. No useful information is passed in this parameter.

**Related RtAction methods:**

void add ( RTEvent * evnt , MStatus * status = 0 )
void add ( const Callback0Ptr < boolean > & cb , MStatus * status = 0 )
void add ( const Callback1Ptr < int , int > & cb , MStatus * status = 0 )

static RtAction * make ( RtStation & stn , const Text * name , RTEvent * evnt , MStatus * status = 0 )
static RtAction * make ( RtStation & stn , const Text * name , const Callback0Ptr < boolean > & cb , 
MStatus * status = 0 )
static RtAction * make ( RtStation & stn , const Text * name , const Callback1Ptr < int , int > & cb , 
MStatus * status = 0 )

40 MTS Systems Proprietary September 15, 2003
5.2.2 “Request Test State Change” commands

These commands have the same effect as pressing the Stop, Hold or Run buttons on the Station Manager main window or on the RSC. These are requests sent to the current controlling application, which may respond to them, or ignore them. Because they are sent to the application, the response time for these commands is poorer than the stop or interlock commands. See Chapter 12 on page 157 for more information about the station test state, and controlling applications.

Note:

Through this feature, it is possible for an application to design an action such that the “Run” button will be automatically pressed in response to some real-time event. This generates a significant safety concern, when the operator assumes that once he/she has pressed the “Stop” button, that the program will not automatically resume.

**Related RtStation methods:**

RtCommand * getTestStateActionCommand ( ProgramState aState , MStatus * status = 0 );

**Related RtAction methods:**

static RtAction * makeTestStateAct ( RtStation & stn , const Text * name , ProgramState aState , MStatus * status = 0 )

5.2.3 Custom generator stop commands

The RtCommand for initiating a stop action on the segment generators is available for Station Manager, but is not complete enough for applications to use. The principal problem deals with cleanup during abnormal termination of the application. This may be corrected in a subsequent version.

**Related RtStation methods:**

RtCommand * getStopActionCommand ( unsigned long id , MStatus * status = NIL )

unsigned long allocateStopActionID ( )

void deallocateStopActionID ( unsigned long id )

**Related RtSegGen methods:**

void setStopTask ( long index , long type , long modeID , double target , double time , MStatus * status =0)

void doStopTask ( long index , MStatus * status = 0 )

5.2.4 Delay commands

Actions can contain delay commands, which cause the commands that follow to be delayed by a specified amount of time. How this is done is most easily seen in an example. The code in Example 3 creates an action that generates callback myMethod, delays 5 seconds, and then generates callback myDelayedMethod.
Calling `RtStation::getDelayCommand()` will create a delay command object. That can be added to an action. Multiple delay commands can be put into the same action.

If an action is active (i.e., has been fired but is still in a delay), firing it again will have no effect. Also, if an action is edited or deleted while it is active, its activity will be immediately canceled (it will not execute the remaining commands).

The method `RtCommand::isSynchronous()` returns TRUE for command objects that have the effect of delaying the subsequent commands in an action.

```
// *** create action that generates a single callback...
RtAction *act = RtAction::make(*rtStn, 0, BOOL_CB0(this, MyClass, myMethod));

// *** add a delay command to it...
RtCommand *delayCmd = myStation->getDelayCommand( 5.0 );
act->add( delayCmd );

// *** add another callback to it
act->add(BOOL_CB0(this, MyClass, myDelayedMethod));
```

**Example 3 — Creation of an action containing a delay.**

Another mechanism exists for setting up `RtAction` objects. This involves using the `RtActionInfo` and its subclasses. This is the mechanism used by the Station Manager Event Editor, and closely models the operations the user can define through that interface. This mechanism is layered on top of the `RtCommand` mechanism and supports archiving of the specifications. It is not documented here, because most applications will not be using it (although they could).

```
// *** create action that generates a single callback...
RtAction *act = RtAction::make(*rtStn, 0, BOOL_CB0(this, MyClass, myMethod));

// *** add a delay command to it...
RtCommand *delayCmd = myStation->getDelayCommand( 5.0 );
act->add( delayCmd );

// *** add another callback to it
act->add(BOOL_CB0(this, MyClass, myDelayedMethod));
```

**5.3 Action types and RtActionInfo**

Another mechanism exists for setting up `RtAction` objects. This involves using the `RtActionInfo` and its subclasses. This is the mechanism used by the Station Manager Event Editor, and closely models the operations the user can define through that interface. This mechanism is layered on top of the `RtCommand` mechanism and supports archiving of the specifications. It is not documented here, because most applications will not be using it (although they could).

```
// *** create action that generates a single callback...
RtAction *act = RtAction::make(*rtStn, 0, BOOL_CB0(this, MyClass, myMethod));

// *** add a delay command to it...
RtCommand *delayCmd = myStation->getDelayCommand( 5.0 );
act->add( delayCmd );

// *** add another callback to it
act->add(BOOL_CB0(this, MyClass, myDelayedMethod));
```

**Example 3 — Creation of an action containing a delay.**
Chapter 6
Control Channels

A control channel is an output channel used to control an actuator, motor, or some other physical property. This control channel may include an internal closed-loop control, or that may output a high-level command to a controller outside the system. These two cases are handled very similarly, as discussed in Section 6.6 on page 54.

The structure of a control channel is described in the following sections.

6.1 RtChannel class

The RtChannel class models a control channel. It is a rather tight coupling of three major classes: RtChannel, RtSegGen, and RtCtrlMode. This is summarized in Figure 2.

![Figure 2 — Overview of Control Channels](image)

The rounded boxes are signals maintained by the RtChannel. A control channel contains the following major attributes:

**Control channel type** — The control channel type is one of the following:

- InternalController (internal controller on a valve driver)
- ProgramAndControl (internal controller on a D/A)
- ProgramWithFeedback
- CommandPlusError
- ProgramOnly

**List of control modes** — A control mode specifies what signal or signals are being controlled to, and the algorithm and tuning parameters used to control it. This list of available control modes is defined in the Station Builder, and loaded with the station. A lot of the attributes normally associated with a control channel are actually embedded in each control mode.
Active control mode — Only one control mode is active on a control channel at once. This indicates which one is active. The active mode may be changed explicitly with methods for that purpose, or asynchronously via profiles queued to the segment generator.

Default control mode — This is the control mode made active when the station is initially loaded. In Version 3.4, this is always the first one.

Segment Generator — Each control channel contains one Segment Generator. This object is used to generate all the waveforms on the channel. This is done by creating segment generator profiles, and queuing them onto the segment generator. The segment generator can dynamically switch control modes. See Chapter 7 on page 57 for more information on using a channel’s segment generator.

Output Signal — This is the output signal of the control channel. It is sometimes called the "Drive Signal". It typically goes to the valve driver controlled by this control channel, but this may be an analog output. In future versions, it may go to a calculated output.

Other internal signals — These are the Command, Compensated Command, Error, Active Feedback, and Active Stabilization signals. These signals are maintained by the control channel, and they always reflect information on the active mode. Because of this, they may change dimension when a mode-switch occurs.

HSM reference — The power for an actuator may be supplied via a Hydraulic Service Manifold (HSM). When this is the case, the control channel contains a reference to the HSM Hardware Resource for that HSM. See Section 11.3 on page 150 for more information on HSMs.

Actuator type — This is either NormalActuator, or HydrostaticActuator. The only difference is in how auto-tuning works.

Rate — A control channel runs at a particular update rate. Normal channels run at the system rate, but low-rate channels run lower than that.

Program State — This reflects the program state of the segment generator contained in the Channel.

Saturation state — This is the saturation state of the active control mode. If the active feedback on the active control mode is out-of-range, ability of the control channel to control becomes compromised. See Section 6.2 on page 45 for more information about a control mode’s saturation state.

The “out-of-range but not saturated” condition allows running, but may prohibit certain operations. The method getSaturationState() returns this state. The application can hook up to the OneOfList returned by getFbkStatusParm() to be informed of when this state changes. This callback path will not notify on every change. It will notify on the first change, and limit the number of subsequent ones to avoid overloading the machine.

The SIG_INVALID state occurs when the value on the signal is meaningless. This can occur in calculated signals when certain errors occur in the calculation. It also can occur if the manual offset becomes too large.
Most of the operations on control channels are obvious from the available methods (listed here). A separate discussion of mode switching is in Section 0

**Related RtChannel methods:**

- `static RtChannel * find ( RtStation & stn , const Text &name , MStatus * status = 0 )`
- `ControlChannelTypes getChannelType ( MStatus * status = 0 )`
- `static const GenList * getChannelTypeList ( )`
- `OneOfList * getActuatorType ( )`
- `RtHsm * getHsm ( MStatus * status = 0 )`
- `RtSegGen * segGen ( MStatus * status = 0 )`
- `RtReadWriteSg * segGen ( SegGenMode writeMode , MStatus * status = 0 )`
- `RtFloatSig * driveSignal ( MStatus * status = 0 )`
- `RtFloatSig * command ( MStatus * status = 0 )`
- `RtFloatSig * compensatedCommand ( MStatus * status = 0 )`
- `RtFloatSig * error ( MStatus * status = 0 )`
- `RtFloatSig * activeFeedback ( MStatus * status = 0 )`
- `RtFloatSig * activeStabilization ( MStatus * status = 0 )`
- `SignalStates getSaturationState ( MStatus * status = 0 )`
- `OneOfList * getFbkStatusParm ( MStatus * status = 0 )`
- `TwoState * getPrgStateParm ( MStatus * status = 0 )`
- `double getRTExecutionRate ( MStatus * status = 0 )`

### 6.2 RtCtrlMode class

A control mode specifies what signal or signals are being controlled to, and the algorithm and tuning parameters used to control it. It is more than just a “feedback selector”. A control mode is structured as shown in Figure 3. This is a general diagram, and as such is over-simplified.
Here are the principal attributes of a control mode. The other ones are described in subsequent subsections.

**Name** — Each control mode has a name, which is unique to the control modes on this control channel. Unlike other `RtObject` subclasses, control modes are not looked up on the station.

**Index** — Sometimes a control mode is identified by index. This is its index into the control channel’s mode list.

**Base controller** — This is the main control loop. It is usually a PIDF controller. If the control channel is driving an external controller, this will be null (i.e. no internal base controller). Depending on the type of controller, there will be one or more feedbacks connected to this base controller.

**Cascaded controller** — (future) Cascaded control is not supported in Version 3.4.

**Active feedback** — The active feedback is the signal that the control mode will attempt to track. When this control mode is active, the dimension of the segment generator command will be the same as the dimension of this feedback. In the simple case, this is the primary feedback fed into the controller. However, on dual-compensation modes, this is the feedback signal that the compensator uses.

**Active Stabilization feedback** — This references the stabilization feedback in the base controller (if any). This will be null if there is no base controller, or if it does not have a stabilization feedback.
List of Compensators — Compensators (sometimes called Optimizers) are algorithms that improve the tracking performance of a control mode by altering the command signal. Through the Station Builder, the operator specifies which of the supported compensators will be available for a control channel. All the modes in the channel get the same compensators. In the Station Manager, the compensators for each mode are individually tuned. The application is responsible to turn individual compensators on and off (through methods on the segment generator).

Internal signals — A control mode exports the internal signals “Absolute Error” and “Command”. These are defined for each mode, but only the ones for the active mode are updated. These can be accessed either via methods on `RtCtrlMode`, or by finding them in the station’s signal lists.

Saturation state — If a control mode is out-of-range, its ability to control the channel becomes compromised. There are four possible states: `SIG_IN_RANGE`, `SIG_OUT_RANGE`, `SIG_SATURATED` and `SIG_INVALID`. The specific values where these states change are dependent on the system hardware.

For the most part, typical applications are only interested in a control mode’s active feedback, and list of compensators. If an application wants to get at the tuning parameters for the controller, it can determine which subclass of `RtCtrlr` the “Base controller” is pointing to, down-cast to that class, and read the parameters. Currently, applications are discouraged from modifying tuning parameters of the controllers.

**Table 2 — Saturation values for different hardware.**

<table>
<thead>
<tr>
<th></th>
<th>497.xx</th>
<th>493.xx</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SIG_IN_RANGE</code></td>
<td>abs(value) &lt; 100%</td>
<td>abs(value) &lt; 100%</td>
</tr>
<tr>
<td><code>SIG_OUT_RANGE</code></td>
<td>100% &lt; abs(value) &lt; 120%</td>
<td>100% &lt; abs(value) &lt; 105%</td>
</tr>
<tr>
<td><code>SIG_SATURATED</code></td>
<td>120% &lt; abs(value)</td>
<td>105% &lt; abs(value)</td>
</tr>
</tbody>
</table>

A control mode that is “out-of-range but not saturated” condition allows running, but may prohibit certain operations. The method `getSaturationState()` returns this state, and is available whether or not the control mode is the active mode.

CE Velocity Limited — This attribute is TRUE if CE Velocity Limiting is enabled in the system, and the dimension of this control mode is “Length”. See Section 6.4.2 on page 51 for more details.

**Related `RtCtrlMode` methods:**

```c
static RtCtrlMode * find ( RtChannel & chan , const Text & name , MStatus * status = 0 )
```

- `GenList * getOptimizerList ( MStatus * status = 0 )`
- `RtFloatSig * activeFbk ( MStatus * status = 0 )`
- `RtFloatSig * activeStbFbk ( MStatus * status = 0 )`
- `RtFloatSig * ctrlrErrFbk ( MStatus * status = 0 )`
- `RtCtrlr * basicCtrlr ( MStatus * status = 0 )`
- `RtCtrlr * cascadedCtrlr ( MStatus * status = 0 )`
- `RtCtrlr * ctrlr ( MStatus * status = 0 )`
- `RtFloatSig * modeErr ( MStatus * status = 0 )`
- `RtFloatSig * absoluteErr ( MStatus * status = 0 )`
- `RtFloatSig * command ( MStatus * status = 0 )`
- `SignalStates getSaturationState ( MStatus * status = 0 )`
6.3 Mode Switching

A mode switch is implemented by turning off the current control mode, and turning on a new control mode, initializing it in such a way as to keep the “output” signal value constant (i.e., the valve signal). A mode switch can be initiated in several ways:

**Static mode switch** — by calling one of several `RtChannel::modeSwitch()` methods (see Section 0 on page 48)

**Queued mode switch** — by playing out a queued `RtProfile` to the segment generator (see Section 7.3 on page 58)

**Custom stop action** — the “ramp-to” and “stop at level” actions are implemented using use a custom stop command available in the segment generator. (see Section 5.2.3 on page 41)

**Asynchronous mode switch action** — by triggering a mode-switch `RtCommand` via an `RtAction` (See Section 6.3.2 on page 49)

An attempt to do a mode-switch can fail if one or more of the feedbacks used by the new mode are saturated, and the hydraulic power is on to the channel. In this case, the mode switch is not performed, and the “Program Interlock” is asserted.

Several methods are available to query for the current mode, and to be informed when the current mode changes.

### Related RtChannel methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>OneOfList * getModeParm ( MStatus * status = 0 )</code></td>
<td>Returns a <code>OneOfList</code> that reflects the current control mode.</td>
</tr>
<tr>
<td><code>GenList * modeList ( MStatus * status = 0 )</code></td>
<td>Update the <code>GenList</code> that reflects the current control mode.</td>
</tr>
<tr>
<td><code>GenList * modeList ( boolean forceUpdate , MStatus * status = 0 )</code></td>
<td>Update the <code>GenList</code> that reflects the current control mode.</td>
</tr>
<tr>
<td><code>RtCtrlMode * getModeNamed ( const Text &amp; modeName , MStatus * status = 0 )</code></td>
<td>Get the named control mode.</td>
</tr>
<tr>
<td><code>RtCtrlMode * activeMode ( MStatus * status = 0 )</code></td>
<td>Get the active control mode.</td>
</tr>
<tr>
<td><code>unsigned int activeModeId ( MStatus * status = 0 )</code></td>
<td>Get the active mode ID.</td>
</tr>
<tr>
<td><code>void uponModeSwitch ( RtAction * anAction , MStatus * status = 0 )</code></td>
<td>Upon mode switch is completed.</td>
</tr>
<tr>
<td><code>void disconUponModeSwitch ( RtAction * anAction , MStatus * status = 0 )</code></td>
<td>Disconnect upon mode switch.</td>
</tr>
</tbody>
</table>

### 6.3.1 Static Mode switch

In the calls to `RtChannel::modeSwitch()`, the mode to switch to can be specified by name, or by index. The optional “force” flag is no longer used.

A static mode-switch can only be performed by an application that has the read-write segment generator allocated.

**Note:**

The method `RtChannel::getModeParm()` returns a `OneOfList` that reflects the current control mode. This is updated when the control mode changes. However, this object cannot be used to cause a mode switch. Invoking `selectItemNamed` on the `OneOfList` will change the
state in the local object, but will not propagate the mode switch request into the machine!

<table>
<thead>
<tr>
<th>Related RtChannel methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void modeSwitch ( const Text &amp; newMode, MStatus * status = 0 )</td>
</tr>
<tr>
<td>void modeSwitch ( unsigned int newModeId, MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean modeSwitch ( const Text &amp; newMode, boolean force, MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean modeSwitch ( unsigned int newModeId, boolean force, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

### 6.3.2 Asynchronous mode switch

An asynchronous mode switch is one that occurs at some unknown time in the middle of playing out a waveform (see Section 7.3 on page 58 for how to play out waveforms through the segment generator). An asynchronous mode switch does not affect the segment generator queue. The current waveform continues to play out unaffected.

The method `RtChannel::getModeSwitchActionCommand()` returns an `RtCommand` that will do an asynchronous mode switch when the `RtAction` it is put into is triggered.

An asynchronous mode switch can only be performed:

1. Between control modes that are of the same dimension (e.g., Force or Length), or
2. When the current `RtProfile` is using a “normalized dimension”. (See Section 6.2 on page 45 for a discussion of normalized dimensions in `RtProfile` objects)

A mode switch command that tries to violate these rules will be rejected, and the Program Interlock will be asserted.

This feature is used to asynchronously change the tuning of the control loop while playing out a predetermined waveform. Typically, the application queues a series of `RtProfile` objects that represent the entire waveform. The initial control mode is specified on the first `RtProfile`, and `NEUTRAL_MODE_INDEX` is specified for the control mode on the subsequent ones. During the play-out, some other events can trigger asynchronous mode switches between two different control modes that have different tuning parameters. Since the dimensions of the control modes are the same (or normalized), the levels specified in the profile objects are applicable for either. Since `NEUTRAL_MODE_INDEX` is specified on the subsequent profiles, starting the next profile will not cause a mode-switch.

<table>
<thead>
<tr>
<th>Related RtChannel methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtCommand * getModeSwitchActionCommand( int modeID, Mstatus *status = 0 )</td>
</tr>
</tbody>
</table>

### 6.4 Span and Set Point controls.

Each control mode contains its own span and set point. However, only the span and set point for the active control mode are actually affecting the system. In addition, if the channel is in the Master Span Group, then each mode is tied into the master span control.
Note:

Starting in Version 3.2, each channel only has one channel span control that is shared by all the control modes on that channel. This span is accessible through the span-related methods on each control mode. Changing the span through any of the control modes on a channel affects the span on all of the modes of the channel. A future version will update the programming interface to match this logical design.

The span and set point controls each contain two different values:

**Current value** — This is the value actually being used to calculate the segment generator output.

**Target value** — This is the last value that the user or application has put into the control.

The span and set point controls generally need to be changed in small increments. This is done with a ramp generator on each of these controls. When an application sets a new value, the target is set to that new value. The current value starts to ramp towards that value. In short, the target is the value that the current value tracks.

The “channel span” can be set independently on each control mode. However, Station Manager always sets the same span value into each control mode.

The channel set points can be modified in the following ways:

1. Direct methods on the `RtCtrlMode`. This is discussed in the next section
2. Mode switch command (discussed in Section 0 on page 47)
3. Control flags on profiles queued to the segment generator (discussed in Section 7.6 on page 66)

### 6.4.1 Changing span and set point explicitly

The methods for explicitly changing the span and set point are listed below. Setting a value on one of these controls causes the control to ramp (at a predefined or specified) rate to the target value. Jamming a value on span or set point causes it to step to that value (which probably causes a step in the command output).

The call to `setSetptToValue(value, time, status)` allows the program to specify exactly how long the ramp should take.

The application can also query for the current values. This returned value is the user’s value (the last one set). The current value may still be ramping to that value (i.e., not yet reached the target). The method `done` returns TRUE when the span, set point and master span are all at their target values. This can be used to determine if the set point or span ramp has completed yet.

The application can set or query the values through `Number` objects.

In addition, methods exist to set and query the time used ramp the span and set point. The default value for these is defined in Station Manager. These can be changed by individual applications. Their values will be reset when the segment generator on the channel is unlocked (see Section 7.2 on page 58).

<table>
<thead>
<tr>
<th>Related RtCtrlMode methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void void setSetptToValue ( double aValue, MStatus * status = 0 )</td>
</tr>
<tr>
<td>void void setSetptToValue ( double aValue, double aTime, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
### 6.4.2 CE-Compliant Velocity Limiting

This option addresses CE requirements that a system will not move more than 10 mm/sec during manual operation. It also requires the system to stop when the operator stops moving it. This only applies to specimen setup or insertion—it does not apply to running an automated test.

The CE-Compliant Velocity Limiting Option is enabled with a line in the HWI file. If it is enabled, then it is the responsibility of the software to ensure that at the appropriate times, the actuator behaves as prescribed.

We interpret this requirement to only apply to manual command when using a control mode with “Length” as its base dimension. Station Manager implements this capability using the methods described in this section. Typically, applications do not need to be concerned with this. However, if an application is going to provide its own user interface for doing manual specimen insertion, then it should ensure that this requirement is satisfied.

The method `RtSystem::getVelocityLimiterOption()` tells you if the option is enabled in the HWI file.

The method `RtCtrlMode::getVelocityLimiterOption()` tells you if the option is enabled in the HWI file and the base dimension of this control mode is “Length”.

If you are adjusting setpoint based on an operator’s input for doing specimen setup or insertion, then you should use the `RtCtrlMode::adjustSetptValueCE()` method to do so.

The `adjustSetptValueCE()` method will ramp the setpoint by a specified delta in a specified time. However, it will not ramp for longer than the specified maxTime. If CE compliance is enabled, the velocity will be limited to 10 mm/sec, and it may not make it in the specified time. The reason for the maxTime is to keep the actuator from moving after the user lets go of the control. The max time should be set relatively short (a fraction of a second). The specified ramp time should be set to a rate similar to how often the GUI control is polled.

**Related RtSystem Methods:**

- `boolean getVelocityLimiterOption()`
6.4.3 Locking channel span and set point controls

Channel set point and span controls are always available to applications. However, if an application wants to manage these controls, it probably does not want the operator to be changing these controls at the same time. The application can tell the Station Manager to disable its channel set point and span controls through methods on the `RtReadWriteSG`. It is in that class, because the application must lock the segment generator before it can disable these controls.

Any application can query for this state through `RtChannel` or `RtCtrlMode`.

6.4.4 Master Span

A station contains one master span. Some or all of the station’s control channels are connected to that master span. This is managed by the Station Manager. Master span is modeled with the `RtSpanGrp` class. The application can get hold of the station’s master span group by calling:

```c
RtSpanGrp *masterSpan = CAST(RtSpanGrp, rtStn->findRtObject(RtSpanGrpCls, "Master Span"))
```

This master span is owned by the station. The application may change the master span value, but must not otherwise reconfigure it.

The application can be informed of changes in either the target value or the list of channels by either hooking callbacks up to the corresponding Parameter objects, or by connecting up `RtActions`.

**Note:**

It is not useful for applications to create additional `RtSpanGrp` objects. This is because a segment generator can only belong to one span group, and the operator may use the Station Manager to add any...
channel to the master span group at any time. This would remove it from the application’s span group without warning.

<table>
<thead>
<tr>
<th>Related RtSpanGrp methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number * getTargetParm (MStatus * status = 0)</td>
</tr>
<tr>
<td>GenList * getChanListParm (MStatus * status = 0)</td>
</tr>
<tr>
<td>void rampToTarget (double aValue, MStatus * status = 0)</td>
</tr>
<tr>
<td>void jamToTarget (double aValue, MStatus * status = 0)</td>
</tr>
<tr>
<td>double getTarget (MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean rampDone (MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean uponChanListChg (RtAction * act, MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean disconUponChanListChg (RtAction * act, MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean uponTargetParmChg (RtAction * act, MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean disconUponTargetParmChg (RtAction * act, MStatus * status = 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other RtSpanGrp methods (not used by typical applications):</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtSpanGrp * make (RtStation &amp; stn, const Text * name, MStatus * status = 0)</td>
</tr>
<tr>
<td>static RtSpanGrp * find (RtStation &amp; stn, const Text * name, MStatus * status = 0)</td>
</tr>
<tr>
<td>virtual ~RtSpanGrp ( )</td>
</tr>
<tr>
<td>void addToGroup (GenList &amp; spanList, MStatus * = 0)</td>
</tr>
<tr>
<td>void addItem (RtObject &amp; anObj, MStatus * = 0)</td>
</tr>
<tr>
<td>void removeFromGroup (RtObject &amp; anObj, MStatus * = 0)</td>
</tr>
<tr>
<td>void removeAll (MStatus * status = 0)</td>
</tr>
<tr>
<td>void setRampRate (double aValue, MStatus * status = 0)</td>
</tr>
<tr>
<td>virtual boolean setParmName (const Text * newName, MStatus * status = 0)</td>
</tr>
</tbody>
</table>

### 6.5 Dual-Compensation modes

Dual-Compensation modes are different from normal modes in two respects:

1. The compensators work on a different feedback than the base controller. This allows, for example, the base PIDF controller to be in stroke control, while the compensator controls load values.

2. It shares a base controller with another mode. The dual-compensation mode does not have a base controller of its own. Rather, it uses one from another mode (called its *master mode*). This means that tuning the base controller only needs to be done once, not twice.

The method `getMasterMode()` can be used to distinguish dual-compensation modes from normal modes. This method will return the mode on which it is based, or zero. In addition, the `RtChannel` methods `masterModeList` and `derivedModeList` return list of the two types of modes.

A dual-compensation mode may be either AC-coupled, or DC-coupled. This is specified in Station Builder. The method `acCoupled()` can be called to determine which type the mode is. Normal control modes are never AC-coupled.

In DC-coupled modes, the compensators control both the DC offset, and the AC amplitude of the signal. In this case, the channel set point is inserted before the compensators.
AC-coupled modes, the compensators only control the AC component of the signal. The DC component is controlled by the base controller. In this case, the channel set point is wired after the compensators. For example, Acceleration-over-Stroke applications will be configured as AC-coupled. The accelerometer does not provide a reliable signal for controlling DC. The "Command" and "Compensated Command" signals will be in the "Acceleration" dimension, but the set point control will be in "Length" dimension.

Dual-compensation modes have two additional tuning parameters: Conversion Gain, and Integrator Gain (DC-coupled only). See the user documentation for descriptions of these parameters.

### Related RtChannel methods:
- `GenList * masterModeList ( MStatus * status = 0 )`
- `GenList * derivedModeList ( MStatus * status = 0 )`

### Related RtCtrlMode methods:
- `RtCtrlMode * getMasterMode ( MStatus * status = 0 )`
- `boolean acCoupled ( MStatus * status = 0 )`
- `Number * getCompXDimGainParm ( MStatus * status = 0 )`
- `Number * getCompTrackingGainParm ( MStatus * status = 0 )`
- `void setCompensatorsGain ( double aValue , MStatus * status = 0 )`
- `double getCompensatorsGain ( MStatus * status = 0 )`
- `void setCompTrackingGain ( double aValue , MStatus * status = 0 )`
- `double getCompTrackingGain ( MStatus * status = 0 )`

### 6.6 Driving external controllers

Control channels can be configured without an internal controller. Station Builder allows four different channel types:

**Program and Control** — This is the normal control channel containing an internal controller.

**Program with Feedback** — This type of channel directly outputs the signal coming from the segment generator. However, it also has one or more feedback signals associated with it. These feedbacks determine the calibration for the output. This channel type is described in more detail in Section 6.6.1 on page 54.

**Command Plus Error** — This is the same as Program with Feedback, except the output is driven with the segment generator signal plus the difference between the command and the feedback. The description in Section 6.6.1 on page 54 also applies to these channels.

**Program Only** — This type of channel does not have any feedback signals associated with it. Dual-compensation modes cannot be created on Program Only channels. This channel type is described in more detail in Section 6.6.2 on page 55.

### 6.6.1 Program with Feedback channels

In a Program with Feedback control channel, the segment generator output is scaled and sent directly to the channel’s output signal. The channel may still be configured with one or more control modes. Often, it is configured with just one control mode, which is set up to reflect the control mode that the external
controller is set for. In this way, the channel will be commanded in engineering units that reflect what the specimen will actually see.

For example, if the external controller were a 458 running in stroke control, the station would be configured with one stroke control mode. This would define one feedback signal for “Stroke”. When the proper sensor calibration is assigned to that feedback, the function generator will command that actuator using a length (e.g., inches).

Sometimes, it is desirable to define more than one control mode on this type of channel.

Sometimes, the external controller will support more than one control mode (e.g., “Stroke” and “Load”), and the operator would be able to manually switch between them. In this case, the `RtChannel` should be configured with both control modes (e.g., “Stroke” and “Load”). Now the application and the operator must together ensure that the `RtChannel` has internally switched to the control mode that matches the mode manually selected on the external controller. Failure to do this will cause the system to generate a command in one dimension (e.g. a force), which the external controller will interpret as another dimension (e.g., a displacement).

The channel will perform the internal mode switch in such a way that the value sent to the analog output hardware resource does not change.

Dual-compensation control modes can also be defined for these types of channels.

### 6.6.2 Program Only channels

In **Program Only** channels, the segment generator output is sent directly to the output signal. There is no feedback signal associated with the channels. Station Builder does not provide a mechanism to create control modes, and dual-compensation modes are not supported.

However, the RT API is structured to always have a control mode. Applications go through the current control mode to determine the scaling for the segment generator command. So, **Program Only** channels are automatically configured with one “dummy” control mode with the following characteristics:

- **Internal Name** — “(none)"
- **Display Name** — “(Program only)” (translated appropriately)
- **ActiveFbk** — Points to the `RtFloatSig` that is the **output signal** for the channel. That is not really a feedback signal, but the calibration of this output is placed by Station Manager on that signal. Most applications go through this attribute to determine the full-scales for the current control mode.
Chapter 7
Segment Generators and Profiles

7.1 Overview

Each control channel contains one Segment Generator. This object is responsible for generating the waveforms (profiles) to be played out on that channel. This object is a rather general device, intended to cover a wide variety of requirements. The major tasks which the segment generator performs includes:

1. Locking a segment generator (only one source can control it at once).
2. Queuing profiles to be played out.
3. Program control (start/stop, hold/resume).
4. Maintaining counters and signals.
5. Controlling span and set point.
6. Selecting and enabling compensators
7. Synchronizing multiple channels.
8. Enabling/disabling the up-sample filter.

While this list may look daunting, not all applications need to be concerned with all these areas. This list is in the approximate order of common use. Many applications need not require anything past the first four items.

Figure 4 shows a functional overview of the contents of a segment generator. For the most part, these individual components are not directly exported as separate objects to the application. Rather, they are accessed indirectly via methods on the segment generator. Nonetheless, it is helpful for the programmer to understand the general flow within it. Refer back to this diagram in the discussions in the sections that follow.
7.2 Locking a segment generator

In general, multiple applications have access to a station at once. There is no systemic mechanism built into the station to arbitrate resource conflicts between applications on the same station. It is up to the individual classes to implement any such arbitration.

Driving a control channel with a segment generator is something that can only be done by one application at a time. Thus, an application must lock a channel’s segment generator before doing much with it. This locking mechanism is implemented with a pair of classes: RtSegGen and RtReadWriteSg.

The RtSegGen class provides a subset of the segment generator capabilities, for which locking of the segment generator is not necessary. The application can call RtChannel::segGen() to get a pointer to this object. This subset includes querying for virtually all information, but does not support methods that affect the output of the segment generator. The pointer returned by this method is owned by the station and must NOT be destroyed by the caller.

The RtReadWriteSg class is used by an application that needs to control the segment generator. The call:

```
RtReadWriteSg *sg = myChannel->segGen(WRITE, &retStatus);
```

will lock the segment generator (if possible), and return a RtReadWriteSg pointer. The application then has exclusive write access to the segment generator through that pointer.

The segment generator is unlocked by destroying that RtReadWriteSg object:

```
delete sg;
```

Note

In the method summaries for RtSegGen and RtReadWriteSg in this chapter, each method is followed by a keyword “(Read)”, “(Write)”, or “(Both)” to indicate which of these two classes the method is available in.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean  inUse ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>TwoState *getInUseParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>RtSegGen *segGen ( )</td>
</tr>
<tr>
<td>boolean hasRwSeggen( MStatus * status = 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtChannel methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtSegGen * segGen(MStatus * status = 0 )</td>
</tr>
<tr>
<td>RtReadWriteSg *segGen(WRITE, MStatus *status = 0)</td>
</tr>
</tbody>
</table>

7.3 Queuing waveforms to be played out

A Profile is a specification of some waveform to play out a segment generator. Profiles are created against a particular segment generator. They are parameterized with the appropriate values, and queued to that segment generator. The profiles will then be played out in sequence, without any delay between them. This playout is controlled by the start/stop and hold/resume commands, as well as by multi-channel synchronization.
The class **RtProfile** is the base class for profiles. Different **RtProfile** subclasses generate different types of waveforms. These different profile types are described in Section 7.10 starting on page 72.

### Example 4 — Simple RtSegment Example

Example 4 shows a simple case of generating a ramp. This code:

1. Locks the read-write segment generator for the control channel,
2. Finds the control mode named “Load”,
3. Creates an **RtSegment** profile, to ramp to 100 kN in 10 seconds,
4. Queues the ramp to the segment generator,
5. Starts the segment generator.

Issues like error checking, and determining when the ramp is done have been left out of this example.

A variety of methods are available to manipulate this queue. The method **resetSegGen()** will flush the whole queue. The method **abortTask()** aborts the current profile, and possibly flushes the rest of the queue. If the profile is defined with an END_SOFT flag (see Table 7 on page 74), then the profile will perform the end-taper before aborting. The method can specify an action to be fired when the abort is complete.

The **size()** method returns the number of profiles currently in the queue. See Section 7.10 for more information on defining profiles.

### Related RtSegGen/RtReadWriteSg methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean RtProfile::queue( MStatus *status = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>void resetSegGen ( MStatus * status = 0 )</td>
<td>(Write)</td>
</tr>
<tr>
<td>void abortTask(boolean flushQ, RtAction *anAction, MStatus * = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>long size ( MStatus * status = 0 )</td>
<td>(Both)</td>
</tr>
</tbody>
</table>
7.4 Program control (stop/start, hold/resume)

7.4.1 Program state machine

The play-out of profiles is controlled by the state machine shown in Figure 5. The small solid boxes are the states. The large dashed boxes are state groups to make the transitions easier to understand. A transition line coming from a dashed box is equivalent to it coming from all of the enclosed states. There are three principal states STOP, HOLD, and RUN. The others are transition states. They are discussed in more detail in Section 7.4.3.

These states are defined by the enumeration ProgramState, and accessible via the method RtSegGen::getProgramState(). In addition, a (simplified) composite of this state for all channels on the station is available through RtStation (See Section 7.4.5, page 63).

The application controls the program state by two “switches”: stop/start and hold/resume. Methods by those names are available for changing the switches. Also, calling getRunStopParm will return a TwoState for controlling the stop/start switch (however, this TwoState will not be updated if a different application changes the state).

These two switches combine into four possible events into the segment generator state machine:
- <stop-hold>
- <stop-resume>
- <start-hold>
- <start-resume>

These are visible in Figure 5 on the state transition lines. Notice that both start and resume must be selected for the segment generator to actually start.

![Figure 5 — Segment Generator State Machine.](image_url)
The other event shown in the state machine is <done>. This is signaled when the transition action is complete. Transition actions are discussed in more detail in Section 7.4.3.

The done() method returns the internal state of several different controls. It is useful to determine when transition states or profile begin/end activity is complete. The TransitionControlMask can examine the following internal controls:

- CONTROL_SPAN: Multiplier used for program control and profile tapers.
- CONTROL_SETPT: Offset used for program control and profile tapers.
- USER_SPAN: Channel span (all modes).
- USER_SETPT: Channel set point (all modes).

These flags can be or-ed together to indicate which controls to check. If any of these controls are not at their target (i.e. still moving), then done() returns FALSE. The second form of done() is equivalent to the first form with all flags set.

### Related RtSegGen/RtReadWriteSg methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start (MStatus * status = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>void stop (MStatus * status = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>void hold (MStatus * status = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>void resume (MStatus * status = 0)</td>
<td>(Write)</td>
</tr>
<tr>
<td>boolean running (MStatus * status = 0)</td>
<td>(Both)</td>
</tr>
<tr>
<td>TwoState * getRunStopParm (MStatus * status = 0)</td>
<td>(Both)</td>
</tr>
<tr>
<td>ProgramState getProgramState (MStatus * status = 0)</td>
<td>(Both)</td>
</tr>
<tr>
<td>boolean done(TransitionControlMask mask, MStatus * = 0)</td>
<td>(Both)</td>
</tr>
<tr>
<td>boolean done (MStatus * status = 0)</td>
<td>(Both)</td>
</tr>
</tbody>
</table>

7.4.2 Waiting for Program State changes to occur

Calling one of these program control methods (e.g. start) does not cause the state change "immediately". It sets a flag, and the state change occurs on the next tick of the system clock. For channels run at the system rate, this is immediate enough. However, on low rate channels, or on the simulator it is possible for another method from the application to be sent to the machine before the state change actually occurs.

For example, queue a profile that will switch control modes, call start, and then ask for the current control mode. Depending on how fast the last query goes down, it may read the old mode or the new mode.

The way to avoid this race condition is to make sure the state change occurred before doing the final query. This can be done with the done() method. The done() method can check for completion of several different things, but it always checks the flag that determines that some pending state change has not been processed yet. This code fragment will accomplish this:

```c
seggen->start();
while (!seggen->done(TransitionControlMask(0)))
    ;
// now we can assume the start has been processed.
```

7.4.3 Transition states

Figure 5 shows the states STARTING, STOPPING, HOLDING and RESUMING. These are transition states. The segment generator can be programmed to perform certain actions when going through these
states. For some actions, the transition takes no time (e.g. hold at level), while other transition actions may take several seconds (e.g. soft stop at mean).

Two settings within the segment generator affect transition behavior:

1. Hold/resume behavior, and
2. Stop/start behavior.

These variables are set through control flags specified either on RtProfile objects queued to the segment generator, or via the setControlFlags() member function. When the profile starts executing, the behaviors specified for that profile are set into the segment generator. The behaviors remain in effect for subsequent profiles, until a profile specifies a different behavior. The possible settings are shown in Table 7, starting on page 74.

Note:

The initial hold and stop behaviors are not defined! When a segment generator is locked, these settings may be in any position. An application should explicitly set the behavior it wants on the first profile it queues!

There is no way to query for the current values of these settings.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setControlFlags (long, MStatus * status = 0)</td>
</tr>
<tr>
<td>boolean RtProfile::queue( MStatus *status = 0)</td>
</tr>
</tbody>
</table>

### 7.4.4 Changing transition times

The default time intervals for the transition states, as well as the beginning and ending taper times for profiles can be changed in Station Manager. Applications may query and change these times through the methods summarized here.

Changing these times must be done through the RtReadWriteSg. When the RtReadWriteSg is unlocked, these times return to the times specified in Station Manager.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setStartTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setStopTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setStartRampPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setStopRampPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setHoldTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setResumeTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setHoldRampPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setResumeRampPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setBeginTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>void setEndTaperPeriod(double aTime, MStatus * = 0);</td>
</tr>
<tr>
<td>double getStartTaperPeriod(MStatus * = 0);</td>
</tr>
<tr>
<td>double getStopTaperPeriod(MStatus * = 0);</td>
</tr>
<tr>
<td>double getStartRampPeriod(MStatus * = 0);</td>
</tr>
<tr>
<td>double getStopRampPeriod(MStatus * = 0);</td>
</tr>
</tbody>
</table>
7.4.5 When segment generators automatically stop

Generally, once the application calls `start()`, the segment generator will remain in a non-stopped state, until the application calls `stop()`, or deletes the `RtReadWriteSg`. However, there are a few conditions where the segment generator will switch to STOP state without the application explicitly telling it to. It is important that the application catches these situations and reacts appropriately. Otherwise, it could wait for completion of a profile that will never come.

**Station interlock is triggered** — This will stop all the segment generators. See 11.1 on page 149 for how to detect a station interlock.

**Program interlock is triggered** — This will stop all the segment generators. See Section 11.2 on page 150 for information on how to detect a program interlock.

**Hydraulic Power Loss (HSM Off)** — When an HSM is turned off, it will stop the segment generators on all channels connected to the HSM. See Section 11.3 on page 150 for how to detect changes in an HSM state.

**A mode switch fails** — This usually happens because the feedback is saturated, span is to small, or the required command would be out-of-range. This will trigger program interlock.

**A “stop-at-level” or “ramp-to” action** — This will stop all the segment generators, and may switch control modes. The method `RtStation::uponStopAction()` can be used to detect when this happens.

**Starvation when playing out RtSeries** — This will stop the segment generator seeing the starvation. See Section 7.8.1 on page 68 for more information about profile starvation. Currently, there is no specific notification back to the application that this has happened.
The application can use the station’s composite program state to detect when the generators stop (See Section 7.4.6 on page 64). However, it will only show when all the channels have been stopped. It will also issue the notification when the application initiates the stop.

<table>
<thead>
<tr>
<th>Related RtStation methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void uponStopAction ( RtAction * anAction , MStatus * status = 0 )</td>
</tr>
<tr>
<td>void disconUponStopAction ( RtAction * anAction , MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

7.4.6 Monitoring station’s composite program state

RtStation contains methods that can be used to get a general picture of what all the segment generators on the station are doing. The method RtStation::getState returns an enumeration of type ProgramStates. This is a simplified version of ProgramState, which each segment generator maintains. The method getState maps each segment generator’s state into the simplified state, then returns the maximum state of all the segment generators, per the following rules:

- PRG_STOP is only returned if all the segment generators are STOPPED.
- PRG_RUN is returned if any of them are in RUN, STARTING, STOPPING, HOLDING, or RESUMING.
- PRG_HOLD is returned if any of them are in HOLDING or HOLD, but none are in RUN, STARTING, STOPPING, HOLDING, or RESUMING.

The application can also be notified when this composite program state changes by setting a callback on the OneOfList returned by getStateParm(), or by connecting an action with uponStateChg().

Note

The uponStateChg() action is called for a variety of station state changes, not just this program state.

<table>
<thead>
<tr>
<th>ProgramState</th>
<th>ProgramStates</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>STOP</td>
<td>PRG_STOP</td>
</tr>
<tr>
<td>HOLD</td>
<td>PRG_HOLD</td>
</tr>
<tr>
<td>STARTING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>STOPPING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>HOLDING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>RESUMING</td>
<td>PRG_RUN</td>
</tr>
</tbody>
</table>

Table 3 — Mapping RtSegGen 'ProgramState' to RtStation 'ProgramStates'

7.4.7 Breakpoint Holds

The “breakpoint hold” feature allows the application to mark certain places in the profiles where a hold will be automatically asserted. This feature can be used to implement several different behaviors (like “hold at end-of-segment” and “single-stepping”).

Related RtStation methods:

<table>
<thead>
<tr>
<th>ProgramStates</th>
<th>getState ( MStatus * status = 0 ) const</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneOfList *</td>
<td>getStateParm ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean</td>
<td>uponStateChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean</td>
<td>disconUponStateChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>void</td>
<td>stop ( MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
This feature is principally implemented with the method `RtReadWriteSg::setBreakpointHold(boolean)`, and the `RtProfile` control flags `END_OF_TASK_HOLD` and `END_OF_SEGMENT_HOLD`.

Passing a `TRUE` to `setBreakpointHold()` will assert “breakpoint hold pending”, which will cause the generator to begin watching for the following control flags:

- The `END_OF_TASK_HOLD` control flag instructs the generator to hold when it reaches the end of the profile, if “breakpoint hold pending” is asserted at the time.

- The `END_OF_SEGMENT_HOLD` control flag instructs the generator to hold when it reaches the end of each segment, if “breakpoint hold pending” is asserted at the time. An end of segment is generally defined as when the segment count is incremented.

In either case, the hold that is initiated will perform the current hold behavior, such as “hold-at-level”, “taper-to-zero”, etc. (See Section 7.4.3 on page 61 for more information on hold behaviors.)

When “breakpoint hold pending” is `FALSE`, these control flags are ignored. So, implementing “hold at end-of-segment” can be accomplished by setting `END_OF_SEGMENT_HOLD` on every profile, and leaving “breakpoint hold pending” disabled. When the application wants to hold at the end of the next segment, it simply calls `setBreakpointHold(TRUE)`.

The method `setBreakpointAction()` can be called to set an action to fire when the breakpoint hold actually occurs. Calling this method with a null action pointer disables a previously set action. There is no action by default.

All profile types support the `END_OF_TASK_HOLD`. For some profile types (e.g.: random, series, etc.), the `END_OF_SEGMENT_HOLD` is meaningless and is ignored.

### Related RtSegGen/RtReadWriteSg methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void setBreakpointHold(boolean hold, MStatus *status = 0)</code></td>
<td>(Write)</td>
</tr>
<tr>
<td><code>boolean getBreakpointHold (MStatus *status = 0)</code></td>
<td>(Both)</td>
</tr>
<tr>
<td><code>void setBreakpointAction(RtAction *anAction, MStatus *status = 0)</code></td>
<td>(Write)</td>
</tr>
</tbody>
</table>

### 7.5 Maintaining counters and signals

The segment generator maintains the following counters and signals:

- **commandOutput** — This signal reflects the un-optimized output of the segment generator to the controller. This signal is unique in that its dimension will dynamically change as the segment generator mode-switches. Applications that interact with this signal need to be aware of this. They need to hook up a setup-changed callback to the `Number` associated with this signal, and interpret subsequent data in the new dimension.

- **ext** — This refers to the channel’s external command signal, as defined in the Station Builder. The segment generator does not directly use this signal. The application may use this signal when queuing an external command profile.

- **Segment counter** — The segment generator maintains a count of segments completed. This is available as both an integer signal, and as a floating-point signal. In addition, the application can zero this counter by setting the value to zero through the integer signal. The control flag `NO_INCREMENT_COUNT` can be used to suppress the updating of this counter.
Trace — The percentage of the current segment that is complete, described as a ratio (zero to one). This is useful for determining how much of a long segment has been done. The specific interpretation of this attribute depends on the type of profile currently being played out.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtFloatSig * commandOutput (MStatus * status = 0) (Both)</td>
</tr>
<tr>
<td>RtFloatSig * ext (MStatus * status = 0) (Both)</td>
</tr>
<tr>
<td>RtIntegerSig * count1 (MStatus * status = 0) (Both)</td>
</tr>
<tr>
<td>RtFloatSig * floatCycleCount (MStatus * status = 0) (Both)</td>
</tr>
<tr>
<td>double getTrace (MStatus * status = 0) (Both)</td>
</tr>
</tbody>
</table>

7.6 Changing set point through control flags

As discussed in Section 6.3.2 on page 49, every control mode on a channel contains a span and set point control.

The control flags SETPOINT_SET and SETPOINT_CLEAR can be used to transfer an offset into or out of the set point (See Table 7, starting on page 74).

SETPOINT_SET causes any offset in the segment generator to be transferred into the set point. The output of the segment generator is set to zero. The result is that the value seen by the controller remains unchanged.

SETPOINT_CLEAR clears the set point, and sets the segment generator output such that the value seen by the controller remains unchanged.

These flags can be specified to setControlFlags(), causing the operation to be immediately performed. They can also be specified on an RtProfile, where the operation will be performed when the profile begins (after the mode-switch, but before starting the waveform).

These operations affect the set point target value and the current value, as well as the segment generator output.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void setControlFlags (long, MStatus * status = 0) (Write)</td>
</tr>
<tr>
<td>boolean RtProfile::queue (MStatus *status = 0) (Write)</td>
</tr>
</tbody>
</table>

7.7 Enabling/disabling compensators

Command compensators for the control channel reside in each control mode. The application turns them on and off within the segment generator. Therefore, only the application that has locked the segment generator can turn the compensators on and off.

A command compensator is turned on by selecting it with the enableOptimizer() method. This operation affects all control modes on the channel. For example, if Null Pacing is enabled while in stroke control, when switching to load control, that mode’s Null Pacing will be turned on.
Turning command compensators on and off while the generator is running may cause undesirable effects (e.g. big glitches). They are intended to be turned on or off while the command signal is stationary.

Command optimization is turned off when the segment generator is unlocked.

<table>
<thead>
<tr>
<th>Related RtSegGen/RtReadWriteSg methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void enableOptimizer (ControllerType optType, MStatus * status = 0) (Write)</td>
</tr>
<tr>
<td>void disableOptimizer (MStatus * status = 0) (Write)</td>
</tr>
</tbody>
</table>

### 7.8 Synchronizing multiple channels

Read-write segment generators are synchronized through the `RtSyncGroup` object. The application makes an `RtSyncGrp` object and adds the appropriate `RtReadWriteSg` objects to it. While an `RtSyncGrp` can be made named, this is not usually useful. Applications virtually always create them unnamed.

An `RtReadWriteSg` can only be in one `RtSyncGrp` at a time. An attempt to add it to a second group will return an error.

When an `RtReadWriteSg` is destroyed, it will be removed from any `RtSyncGrp` which it is in.

Segment generators that are synchronized start, stop, hold and resume together. Once `RtReadWriteSg::start()` is called on all the segment generators in the sync-group, they will all start. Once `stop()` is called on any segment generator in the group, they will all stop. The same is true for `resume()` and `hold()`.

The `RtSyncGrp` object has `start()`, `stop()`, `hold()`, and `resume()` methods that simply invoke that method on all the objects contained in the group.

The method `getState()` returns an enumeration of type `ProgramStates`. This is a simplified version of `ProgramState`, which each segment generator maintains. The mapping is shown in Table 4. The method `getState` maps each segment generator’s state into the simplified state, then returns the maximum state of all the segment generators, per the following rules:

- PRG_STOP is only returned if all the segment generators are STOPPED.
- PRG_RUN is returned if any of them are in RUN, STARTING, STOPPING, HOLDING, or RESUMING.
- PRG_HOLD is returned if any of them are in HOLDING or HOLD, but none are in RUN, STARTING, STOPPING, HOLDING, or RESUMING.

**Note:**

`RtSyncGrp` supports synchronizing other types of objects besides segment generators. Currently, the only other object that supports synchronization is the time-history data acquisition.

<table>
<thead>
<tr>
<th>ProgramState</th>
<th>ProgramStates</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>STOP</td>
<td>PRG_STOP</td>
</tr>
<tr>
<td>HOLD</td>
<td>PRG_HOLD</td>
</tr>
<tr>
<td>STARTING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>STOPPING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>HOLDING</td>
<td>PRG_RUN</td>
</tr>
<tr>
<td>RESUMING</td>
<td>PRG_RUN</td>
</tr>
</tbody>
</table>

Table 4 — Mapping RtSegGen ‘ProgramState’ to RtSyncGrp ‘ProgramStates’

Related `RtSyncGroup` methods:

| static RtSyncGrp * make ( RtStation & stn, const Text * name, MStatus * status = 0) |
7.8.1 Detecting profile starvation

A segment generator is starved for data when it runs out of something to play out, and is expecting additional profiles. This can happen when spooling end-levels from a profile file, if the application cannot keep ahead of the segment generator in spooling the data.

When playing out the last profile in a series, it is normal to have nothing following it. The last profile in a series should have the `TIME_SYNC_LAST` control flag set (See Table 7 starting on page 74). This flag tells the segment generator not to expect a subsequent profile, and not to assert a starvation condition.

Currently, when the segment generator is starved for data, it will keep playing out the last level (i.e., dwell at that level) until the next profile is queued. There is no notification to the application when this happens.

The `RtSeries` profile type is special in that if starvation occurs when it completes, it will cause a STOP on that segment generator (causing the whole group to stop).

7.8.2 Staying lock-step synchronized

Segment generators in the same `RtSyncGrp` always start, stop, hold, and resume together. Once they start, they will generally stay synchronized.

If one of the channels becomes starved for data, it needs to wait for the application to queue another `RtProfile` before it can continue. The `TIME_SYNC` attribute determines how channels within an `RtSyncGrp` behave if one or more channels enter this “starvation hold” state.

If `TIME_SYNC_ENABLE` is asserted on all the channels in the group, then if any one becomes starved, all the channels will hold. If `TIME_SYNC_DISABLE` is specified on all channels, then a starvation on one channel will have no effect on any other channel.

Sometimes, you want the channels to remain synchronized, but one channel has less data than the others. You don’t want the other channels to hold when the first channel reaches the logical end of its data. You can get this effect by putting `TIME_SYNC_LAST` on the last profile queued to the shorter channel. When playing that profile, the generator will respond to starvation holds from the other channels, but will not assert a starvation hold when it reaches the end of the profile.

This attribute is controlled by the profile `TIME_SYNC_xxxxx` control flags. (See Table 7 starting on page 74).
Note:

The initial TIME_SYNC setting is not defined! When a segment generator is locked, this setting may be in either position. An application should explicitly set the behavior it wants on the first profile it queues!

7.8.3 Resynchronizing

When repeating a series of profiles on multiple channels, it is common for the length of the total profile on one channel to be different than on another channel. This can be an explicit decision, or a result of floating point run-off. In any case, subsequent passes through the series of profiles may not all start at the same time. This will have the effect of a "synchronization drift" over many repeats.

The PROFILE_RESYNC control flag can be used to correct this situation. When a profile with this flag set begins executing, it will delay until every other segment generator in the group reaches a profile with this flag set. Then they will all begin at the same time. This behavior is independent of the TIME_SYNC_ENABLE attribute. PROFILE_RESYNC can be used to start the channels together, even if you want to allow drift due to starvation once it is started.

Thus, it is common to assert PROFILE_RESYNC in the first profile of a series.

The PROFILE_RESYNC flag will cause about a one sample period delay in the profile play-out of the last channel to assert it (about 1 millisecond). Of course, all channels will start together at that time.

7.8.4 Synchronizing parameter changes

Some RtProfile types explicitly support changing their parameters while they are being played out, and having these changes take effect immediately (e.g., RtCyclic and RtSweep). In this case, the application may need the modification of the parameter to happen at the same time on every channel.

For example, if setFrequency() is invoked on the profile for one of the channels, it will not take effect until some call that sets (or adjusts) parameters is made on each of the others (usually the matching setFrequency()). At that time, all the parameters will begin adjusting. This allows the application to make simultaneous changes on all the channels.

The method adjustAllData() is usually provided that to change all the parameters that can be dynamically modified on that profile type.

Occasionally, this synchronization is undesirable. For example, if each channel has its own amplitude, it would be good to change the level on one channel without having to re-send the amplitude for each of the other channels. This is done by invoking setSyncAdjust(FALSE) on the profile. While this flag is FALSE, parameter adjustments will begin to take effect immediately. This flag may need to be changed back and forth as the application adjusts both synchronized and unsynchronized parameters.

This is type-specific behavior, and is not supported by every profile type. Refer to the discussion of the particular RtProfile subclass for additional details.
7.8.5 Common Scenarios

To illustrate how these different flags interact with each other, here are some common situations.

7.8.5.1 Playing out MPT block-arbitrary profile files

When playing out block-arbitrary profile files, each channel has a different sequence of profiles (rows), and the total sequence in one pass may be a different length on each channel. The synchronization flags are used as follows:

- **PROFILE_RESYNC** is set on the first profile of each pass on each channel, so that each pass starts synchronized.
- **TIME_SYNC_ENABLE** is asserted, so the channels stay lock-step synchronized.
- **TIME_SYNC_LAST** is set on the last profile, so that a starvation condition is not asserted after the last profile.
- **HOLD_AT_LEVEL** and **STOP_AT_LEVEL** are asserted.

The **TIME_SYNC_LAST** flag allows the other channels to continue finishing the current pass when the shortest one finishes. The next pass on the shorter channel will not start because of the **PROFILE_RESYNC** on the first profile of the next pass.

In this scenario, the subsequent pass can be queued, before the current pass finishes, and will start with minimal delay upon completion of the current pass.

The **TIME_SYNC_ENABLE**, **HOLD_AT_LEVEL**, and **STOP_AT_LEVEL** flags need only be set once, since the defaults are to use whatever is set last.

7.8.5.2 Playing out Time Histories relative to current command

When playing out time-history files, each channel has its own series of constant-interval data-points. Typically, each channel has the same number of points, so they start and end together. In addition, for this example, we want to play out the time-history file relative to the current command location. The synchronization flags are used as follows:

- **START_RELATIVE** is set in the first profile on each channel. After any potential mode switch, this channel will read the current command output and add it to all the specified values in the RtSeries.
- **RELATIVE_NOCHANGE** is set on each subsequent profile. This causes them to use the same relative offset that the previous profile used.
- **FILTER_INIT** is set on the first profile of each pass on each channel to initialize the up-sample filter (see Section 7.9 on page 71 for more details on the up-sample filters).
- **PROFILE_RESYNC** is set on the first profile of each pass on each channel, so that each pass starts synchronized.
- **HOLD_AT_MEAN | HOLD_SOFT** is set. This causes hold to taper to mean (i.e. set point).
STOP_AT_MEAN | STOP_SOFT is set. This causes stop to taper to mean (i.e. set point).

TIME_SYNC_DISABLE is asserted. If one channel becomes starved for data, the RtSeries profile type will assert a STOP, which will taper to zero. When this happens, we want the other channels to taper normally. We do not want them to hold, waiting for the starved channel.

TIME_SYNC_LAST is set on the last profile of each pass, so that a starvation condition is not asserted after the last profile (which would cause a STOP).

In this scenario, the subsequent pass can be queued before the current pass finishes, and will start with minimal delay upon completion of the current pass.

The TIME_SYNC_DISABLE, HOLD_xxx, and STOP_xxx flags need only be set once, since the defaults are to use whatever is set last.

7.9 Enabling/disabling the up-sample filter

The up-sample filter is used in RPC operations to smooth the output waveform, without generating frequency distortion. Turning the up-sample filter on and off is an immediate operation, and cannot be placed in the profile queue. Normally, this filter is turned on or off while the channel is not moving. Doing so while playing out a profile may cause a bump.

The method initUpSampleFilter() must be called first, to specify the rate of the levels going into the segment generator. This rate should be specified as the inverse of the “interval” passed to the RtSeries profile (see Section 7.10.10 on page 90). The relationship between this specified rate and the system clock rate determines the coefficients of the filter.

The valid range for this rate is:

$$\frac{\text{systemRate}}{10} \leq \text{rate} \leq \text{systemRate}$$

Table 5 shows the actual ratios between the filter rate and the system rate that are supported. As long as the rate is within the above range, it will be rounded to one of these ratios.

<table>
<thead>
<tr>
<th>Filter Rate</th>
<th>System Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>0.125</td>
<td>0.25</td>
</tr>
<tr>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>0.25</td>
<td>0.5</td>
</tr>
<tr>
<td>0.3</td>
<td>0.6</td>
</tr>
<tr>
<td>0.375</td>
<td>0.75</td>
</tr>
<tr>
<td>0.4</td>
<td>0.8</td>
</tr>
<tr>
<td>0.5</td>
<td>1.0</td>
</tr>
<tr>
<td>0.5</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 5 — Supported ratios of filter rate to system rate.

The up-sample filter can be affected by the execution of a RtProfile with one of the following control flags set:

FILTER_INIT — If this flag is set, all the internal states of the filter are initialized with the current output of the filter.

FILTER_ZERO — If this flag is set, all the internal states of the filter are initialized with zero.

These flags have no effect if the filter is not enabled.

Related RtSegGen/RtReadWriteSg methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void initUpSampleFilter ( double dataRate , MStatus * status = 0 )</td>
<td>(Write)</td>
</tr>
<tr>
<td>void enableUpSampleFilter ( MStatus * status = 0 )</td>
<td>(Write)</td>
</tr>
<tr>
<td>void disableUpSampleFilter ( MStatus * status = 0 )</td>
<td>(Write)</td>
</tr>
</tbody>
</table>
7.10 **Types of Profiles**

A *Profile* is a specification of some waveform to play out a segment generator. Profiles are created against a particular segment generator. They are parameterized with the appropriate values, and queued to that segment generator. The profiles will then be played out in sequence, without any delay between them. This play-out is controlled by the start/stop and hold/resume commands, as well as by multi-channel synchronization.

The profile queue is a FIFO list of references to profiles. A profile can be queued several times, so the queue can contain more than one reference to the same profile.

If the parameters of a profile are modified while it is in the queue (before it starts playing out), the new parameters will be used when it does start to play out. Some types of profiles explicitly support changing their parameters while they are executing, and having the updated parameters immediately used. This is a profile-specific behavior.

If a profile is deleted while it is in the queue, it will be removed. If it is actively playing out, it will be aborted.

Profiles are owned by the `RtReadWriteSg`, and will be deleted when the `RtReadWriteSg` is deleted.

### 7.10.1 RtProfile class

The `RtProfile` class is the base class for all profiles. All `RtProfile` classes have the following attributes:

- **Segment generator reference** — Profiles are created against a particular segment generator. They can only be queued to the owning segment generator. The owning segment generator is specified at creation time, and cannot be modified later. This binding allows proper interpretation of the control mode, and dimension of the end-level.

- **Control mode** — Currently, all profiles specify a single control mode. The control mode is specified using the mode’s index.

  This argument can be specified as `NEUTRAL_MODE_INDEX`, which instructs the profile not to do a mode-switch, and to interpret the end-levels in whatever is the current control mode when the profile starts to play out. See Section 6.3.2 on page 49 for a discussion of where this feature can be used.

- **Control flags** — A variety of control flags are available to affect the behavior of the segment generator while playing out this profile. These flags are detailed in Table 7, starting on page 74. These flags serve a variety of purposes, which are discussed in the appropriate sections.

- **Done action** — This action is signaled when the profile completes execution, or when it is explicitly “skipped”.

- **Start action** — This action is signaled when the profile begins execution. If the profile has `PROFILE_RESYNC` set, then this action will not be fired until all the segment generators in the sync group reach a profile with `PROFILE_RESYNC` set.
Normalized Dimension — Normally, the level data in a profile is assumed to be in the system units for the dimension of the control mode that will be used to play out the levels. The method setNormalizedDim() can be used to specify that the level data will be in a “normalized” dimension instead. In this case, the level data will be converted from the normalized dimension to the control mode’s dimension. This will scale the data so full-scale of the normalized dimension will map to full-scale of the control mode. This feature is principally used when doing asynchronous mode-switches, as described in Section 6.3.2 on page 49.

qCount — This indicates how many references to the profile are currently in the segment generator’s queue. Once queued, a profile remains in the queue until it is finished playing out or is flushed from the queue. This number also reflects every time that the profile is referenced in an RtBlock that is (directly or indirectly) in the queue (see Section 7.10.11 on page 92).

While the RtProfile class serves as the base class for all the “interesting” types of profiles, it can also be directly instantiated. A direct RtProfile instance can be queued like any other profile. When executed, it will perform the mode switch (if necessary), and perform the actions directed in the control flags. It will immediately complete, and signal the RtAction.

<table>
<thead>
<tr>
<th>Normalized Dimension</th>
<th>Full Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unitless</td>
<td>1.0</td>
</tr>
<tr>
<td>Ratio</td>
<td>1.0</td>
</tr>
<tr>
<td>Percent</td>
<td>100.0</td>
</tr>
<tr>
<td>Volts</td>
<td>10.0</td>
</tr>
</tbody>
</table>

Table 6 — Normalized Dimensions

Related RtProfile methods:

```
static RtProfile * make ( RtReadWriteSg & aSg ,
                        unsigned int modeID ,
                        unsigned int ctrlFlags ,
                        RtAction * anAction ,
                        MStatus * status = 0 )

virtual ~RtProfile()
virtual void  queue ( MStatus * status = 0 )
virtual void  play ( MStatus * status = 0 )
virtual void  setAllData ( unsigned int modeID ,
                          unsigned int ctrlFlags ,
                          RtAction * anAction ,
                          MStatus * status = 0 )
virtual void  setAllData ( unsigned int modeID ,
                          unsigned int ctrlFlags ,
                          RtAction * endAction ,
                          RtAction * startAction ,
                          MStatus * status = 0 )
virtual void  setCtrlMode ( unsigned int modeID ,
                            MStatus * status = 0 )
virtual unsigned int getCtrlMode ( MStatus * status = 0 )
virtual void  setCtrlFlags ( unsigned int ctrlFlags ,
                            MStatus * status = 0 )
virtual unsigned int getCtrlFlags ( MStatus * status = 0 )
virtual void  setAction ( RtAction * action ,
                         MStatus * status = 0 )
virtual void  setStartAction ( RtAction * action ,
                            MStatus * status = 0 )
virtual void  setAction ( RtAction & action ,
                        MStatus * status = 0 )
virtual void  setStartAction ( RtAction & action ,
                         MStatus * status = 0 )
virtual RtAction *getAction ( MStatus * status = 0 )
virtual RtAction *getStartAction ( MStatus * status = 0 )
virtual long  qCount ( MStatus * status = 0 )
void  setNormalizedDim ( DimId * dimId ,
                        MStatus * status = 0 )
```
Table 7 — Segment Generator control flags

<table>
<thead>
<tr>
<th>Group</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Profile Resynchronization</strong></td>
<td>PROFILE_RESYNC</td>
<td>This profile will wait until all channels in the sync group load a profile with PROFILE_RESYNC specified. An application may use this flag to re-synchronize repeated profiles.</td>
</tr>
<tr>
<td>(Section 7.8.3, page 69)</td>
<td></td>
<td>Flags in this group apply to the current profile only.</td>
</tr>
<tr>
<td><strong>Relative End Levels</strong></td>
<td>START_RELATIVE</td>
<td>The current command, after any mode switch and set point transfer operations, is used as an offset for this profile.</td>
</tr>
<tr>
<td>Only one value of this group can be specified.</td>
<td></td>
<td>Flags in this group apply to the current profile only.</td>
</tr>
<tr>
<td></td>
<td>RELATIVE_NOCHANGE</td>
<td>This profile will use the same relative offset as the previous profile. If this profile is a different control mode than the previous profile, this flag has no effect.</td>
</tr>
<tr>
<td><strong>Profile Begin/End</strong></td>
<td>BEGIN_SOFT END_SOFT</td>
<td>These flags will cause the profile to taper its beginning and/or its end. What this looks like depends on the profile type. Generally, the amplitude of the waveform is tapered. If the profile is aborted with abortTask, the ending taper will be performed.</td>
</tr>
<tr>
<td>Flags in this group apply to the current profile only.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set point Transfer Action</strong></td>
<td>SETPOINT_NOCHANGE</td>
<td>(default) The set point and previous command values are not changed.</td>
</tr>
<tr>
<td>(Section 7.6, page 66)</td>
<td></td>
<td>Only one value of this group can be specified.</td>
</tr>
<tr>
<td></td>
<td>SETPOINT_SET</td>
<td>The set point is loaded with the generator output and the command is zeroed. For setControlFlags(), this occurs immediately. On an RtProfile, this action occurs after the mode-switch.</td>
</tr>
<tr>
<td></td>
<td>SETPOINT_CLEAR</td>
<td>The set point is cleared and command is loaded with the previous generator output / span. For setControlFlags(), this occurs immediately. On an RtProfile, this action occurs after the mode-switch.</td>
</tr>
<tr>
<td><strong>Up-sample Filter Action</strong></td>
<td>FILTER_NO_CHANGE</td>
<td>(default) The up-sample filter is not changed.</td>
</tr>
<tr>
<td>(Section 7.9, page 71)</td>
<td></td>
<td>Only one value of this group can be specified.</td>
</tr>
<tr>
<td></td>
<td>FILTER_INIT</td>
<td>The internal states of the up-sample filter will be loaded with the current value.</td>
</tr>
<tr>
<td></td>
<td>FILTER_ZERO</td>
<td>The internal states of the up-sample filter will be loaded with the all zeros.</td>
</tr>
<tr>
<td><strong>Segment Counter Control</strong></td>
<td>NO_INCREMENT_COUNT</td>
<td>This flag will cause the current profile to NOT update the channel counters when it completes segments. It has no effect when specified in a call to setControlFlags().</td>
</tr>
<tr>
<td>(Section 7.5, page 65)</td>
<td></td>
<td>This flag only applies to the current profile.</td>
</tr>
<tr>
<td><strong>Breakpoint Hold Control</strong></td>
<td>END_OF_TASK_HOLD</td>
<td>This flag will cause a hold to be initiated if “breakpoint hold pending” is asserted when the current profile completes.</td>
</tr>
<tr>
<td>(Section 7.4.7, page 64)</td>
<td></td>
<td>This flag only applies to the current profile.</td>
</tr>
<tr>
<td></td>
<td>END_OF_SEGMENT_HOLD</td>
<td>This flag will cause a hold to be initiated if “breakpoint hold pending” is asserted whenever the current profile completes a segment.</td>
</tr>
</tbody>
</table>
### Table 7 — Segment Generator control flags (continued)

<table>
<thead>
<tr>
<th>Group</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time Synchronization</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Section 7.8, page 67)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only one value of this group can</td>
<td></td>
<td></td>
</tr>
<tr>
<td>be specified. The value specified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>applies to all profiles until</td>
<td></td>
<td></td>
</tr>
<tr>
<td>changed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME_SYNC_NOCHANGE</td>
<td>(default) Time</td>
<td>Time synchronization is not changed.</td>
</tr>
<tr>
<td></td>
<td>synchronization is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>not changed.</td>
<td></td>
</tr>
<tr>
<td>TIME_SYNC_ENABLE</td>
<td></td>
<td>The generator will respond to and assert data starvation holds.</td>
</tr>
<tr>
<td>TIME_SYNC_DISABLE</td>
<td></td>
<td>The generator will not respond to nor assert data starvation holds.</td>
</tr>
<tr>
<td>TIME_SYNC_LAST</td>
<td></td>
<td>The generator will respond to but will not assert data starvation holds.</td>
</tr>
<tr>
<td><strong>Hold Behavior</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Section 7.4.3, page 61)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The specified behavior applies to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all profiles until changed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOLD_NOCHANGE</td>
<td>(default) The hold</td>
<td>The hold behavior is not changed.</td>
</tr>
<tr>
<td></td>
<td>behavior is not</td>
<td></td>
</tr>
<tr>
<td></td>
<td>changed.</td>
<td></td>
</tr>
<tr>
<td>HOLD_AT_LEVEL</td>
<td></td>
<td>This flag sets the generator to hold at the current command level.</td>
</tr>
<tr>
<td>HOLD_AT_MEAN</td>
<td></td>
<td>This flag sets the generator to hold at the current mean. What the mean is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and how the command is transitioned is determined by the type of profile.</td>
</tr>
<tr>
<td>HOLD_AT_ZERO</td>
<td></td>
<td>This flag sets the generator to hold at zero. How the command is transitioned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to zero is determined by the profile type.</td>
</tr>
<tr>
<td>HOLD_SOFT</td>
<td></td>
<td>This flag, when added to another Hold flag, sets the generator to soften the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hold and resume transitions. This is usually done by tapering amplitude to/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>from zero, while continuing to play out the waveform.</td>
</tr>
<tr>
<td><strong>Stop Behavior</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Section 7.4.3, page 61)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The specified behavior applies to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>all profiles until changed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOP_NO_CHANGE</td>
<td>(default) The stop</td>
<td>The stop behavior is not changed.</td>
</tr>
<tr>
<td></td>
<td>behavior is not</td>
<td></td>
</tr>
<tr>
<td></td>
<td>changed.</td>
<td></td>
</tr>
<tr>
<td>STOP_AT_LEVEL</td>
<td></td>
<td>This flag sets the generator to stop at the current command level.</td>
</tr>
<tr>
<td>STOP_AT_MEAN</td>
<td></td>
<td>This flag sets the generator to stop at the current mean. What the mean is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and how the command is transitioned is determined by the type of profile.</td>
</tr>
<tr>
<td>STOP_AT_ZERO</td>
<td></td>
<td>This flag sets the generator to stop at zero. How the command is transitioned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to zero is determined by the command type.</td>
</tr>
<tr>
<td>STOP_SOFT</td>
<td></td>
<td>When added to another Stop flag, this sets the generator to soften the stop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and restart transitions. This is usually done by tapering amplitude to/from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>zero, while continuing to play out the waveform.</td>
</tr>
<tr>
<td>START_SOFT</td>
<td></td>
<td>The STOP_SOFT flag softens the restart of a profile in the middle, but not</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a start at the beginning. START_SOFT can be added to soften the start in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>this case.</td>
</tr>
</tbody>
</table>
7.10.2 RtStep

An RtStep will immediately step to a specified end level. This step will take up to one clock tick. This subclass has the following profile-specific attributes:

**endLevel** — The level to step to.

Methods exist to change individual attributes, or change them all at once.

### Related RtStep methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtStep * make( RtReadWriteSg &amp; aSg, unsigned int modeID, long ctrlFlags, double endLevel, RtAction * endAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>static RtStep * make( RtReadWriteSg &amp; aSg, unsigned int modeID, long ctrlFlags, double endLevel, RtAction * endAction, RtAction * startAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtStep ( )</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned int ctrlFlags, double endLevel, RtAction * endAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned int ctrlFlags, double endLevel, RtAction * endAction, RtAction * startAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setEndLevel ( double level, MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>virtual double getEndLevel ( MStatus * status = 0 )</td>
<td></td>
</tr>
</tbody>
</table>

7.10.3 RtStay

An RtStay will cause the segment generator to switch to the specified control mode, and stay at the current level for the specified time. This subclass has the following profile-specific attributes:

**interval** — The length of time to delay before the profile completes.

Methods exist to change individual attributes, or change them all at once.

The **HOLD_AT_MEAN** and **STOP_AT_MEAN** during the initial segment are treated as **HOLD_AT_LEVEL** and **STOP_AT_LEVEL** respectively.

**BEGIN_SOFT** and **END_SOFT** are ignored.
The method `RtSegGen::getTrace` returns percent completion of the segment (between zero and one).

<table>
<thead>
<tr>
<th>Related RtStay methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtStay * make(</td>
</tr>
<tr>
<td>RtReadWriteSg &amp; aSg,</td>
</tr>
<tr>
<td>unsigned int modeID,</td>
</tr>
<tr>
<td>long ctrlFlags,</td>
</tr>
<tr>
<td>double interval,</td>
</tr>
<tr>
<td>RtAction * endAction,</td>
</tr>
<tr>
<td>MStatus * status = 0);</td>
</tr>
<tr>
<td>static RtStay * make(</td>
</tr>
<tr>
<td>RtReadWriteSg &amp; aSg,</td>
</tr>
<tr>
<td>unsigned int modeID,</td>
</tr>
<tr>
<td>long ctrlFlags,</td>
</tr>
<tr>
<td>double interval,</td>
</tr>
<tr>
<td>RtAction * endAction,</td>
</tr>
<tr>
<td>RtAction * startAction,</td>
</tr>
<tr>
<td>MStatus * status = 0);</td>
</tr>
<tr>
<td>virtual ~RtStay ( )</td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID,</td>
</tr>
<tr>
<td>unsigned int ctrlFlags,</td>
</tr>
<tr>
<td>double interval,</td>
</tr>
<tr>
<td>RtAction * endAction,</td>
</tr>
<tr>
<td>MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID,</td>
</tr>
<tr>
<td>unsigned int ctrlFlags,</td>
</tr>
<tr>
<td>double interval,</td>
</tr>
<tr>
<td>RtAction * endAction,</td>
</tr>
<tr>
<td>RtAction * startAction,</td>
</tr>
<tr>
<td>MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual void setInterval ( double anInterval , MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual double getInterval ( MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

**7.10.4 RtSegment**

An `RtSegment` will switch to the specified control mode, and generate a single segment with the specified end level, interval, and wave shape. This subclass has the following profile-specific attributes:

- **shape** — `SQUARE`, `RAMP`, or `SINE` as shown in Figure 7.

![SQUARE, RAMP, SINE waveshapes](image)

- **timing type** — This determines how the timing of the segment is specified. It can be specified as `FREQ` (2 segments per second), `PERIOD` (interval for one segment), or `RATE` (engineering units per second).

- **timing** — The frequency, time, or rate value.

- **endLevel** — The level to ramp to.

Methods exist to change individual attributes, or change them all at once. Once the `RtSegment` begins the segment, changes in the attributes will not affect the current segment.
The **HOLD_AT_MEAN** and **STOP_AT_MEAN** during the initial segment are treated as **HOLD_AT_LEVEL** and **STOP_AT_LEVEL** respectively.

**BEGIN_SOFT** and **END_SOFT** are ignored.

The method **RtSegGen::getTrace** returns percent completion of the segment (between zero and one).

### Related RtSegment methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtSegment *make( RtReadWriteSg &amp; aSg, unsigned int modeID, long ctrlFlags, WaveShape aShape, TimingType timingType, double timing, double endLevel, RtAction * endAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>static RtSegment *make( RtReadWriteSg &amp; aSg, unsigned int modeID, long ctrlFlags, WaveShape aShape, TimingType timingType, double timing, double endLevel, RtAction * endAction, RtAction * startAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtSegment ()</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned intctrlFlags, enum WaveShape shape, enum TimingType timingType, double timing, double endLevel, RtAction * endAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned int ctrlFlags, enum WaveShape shape, enum TimingType timingType, double timing, double endLevel, RtAction * endAction, RtAction * startAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setShape ( WaveShape shape, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual WaveShape getShape ( MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setTimingType ( TimingType aType, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual TimingType getTimingType ( MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setTimingValue ( double timing, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual double getTimingValue ( MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setEndLevel ( double level, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual double getEndLevel ( MStatus * status = 0 );</td>
<td></td>
</tr>
</tbody>
</table>
7.10.5 RtCyclic2

An RtCyclic2 will switch to the specified control mode, and generate a series of segments between two end levels. This subclass has the following profile-specific attributes:

**shape** — One of the six shapes shown in Figure 8. These shapes are discussed in more detail below.

**timing type** — This determines how the timing of the segments are specified. It can be specified as **FREQ** (2 segments per second), **PERIOD** (interval for one segment), or **RATE** (engineering units per second).

**Initial timing** — The frequency, time, or rate value for the first segment. This will go from wherever the output is to the first specified level (not used for TRUE_SINE, TRUE_RAMP or TRUE_SQUARE).

**timing** — The frequency, time, or rate value for the subsequent segments. These will go between the end levels.

**endLevel 1** — The first end level.

**endLevel 2** — The second end level.

**segment count** — The total number of segments to generate. If this is -1 the profile will never stop on its own. Any other negative value is an error.

Methods exist to change individual attributes, or change them all at once. Once the RtCyclic2 begins executing, changes in the attributes will not affect the output.

The waveshapes SQUARE, RAMP, and SINE are generated by going from the current level to endlevel 1 using **initial timing**, then going back and forth “segment count - 1” times using **timing**.

The waveshapes TRUE_SQUARE, TRUE_RAMP, and TRUE_SINE do not use **initial timing**. They immediately step to the midpoint between levels 1 and 2, and start a periodic waveform of the appropriate shape with the two levels as the peak and valley. The specified number of half-cycles will be output, and command will end at the mean.

Phased profiles are implemented by incorporating an appropriate delay into the **initial timing**.

Figure 9 shows the waveform generated by the following profile:

```
RtCyclic2::make( sg, mode, 0, RAMP, PERIOD, 5.0, 2.0, 4.0, 2.0, 8 , action, status) ;
```
RtCyclic2 implements the BEGIN_SOFT and END_SOFT control flags to “soften” the beginning and end of the waveform. BEGIN_SOFT cause the profile to first go to the mean, then taper the waveform. END_SOFT will cause the waveform to begin tapering before the cycle-count is completed, such that it the amplitude is zero when the count is complete. This action is done in the generator, independent of the span and set point.

The HOLD_AT_MEAN and STOP_AT_MEAN during the initial segment are treated as HOLD_AT_LEVEL and STOP_AT_LEVEL respectively. In the subsequent segments, they will ramp (or taper) to the mid-point of the two levels.

The method RtSegGen::getTrace returns percent completion of the current segment (between zero and one).

Related RtCyclic2 methods:

```cpp
static RtCyclic2 *make( RtReadWriteSg & aSg,
unsigned int  modeID,
long   ctrlFlags,
WaveShape  aShape,
TimingType  timingType,
double   initTiming,
double   timing,
double   initEndL,
double   endLevel,
long   segCount,
RtAction * endAction ,
MStatus  * status = 0);
```

```cpp
static RtCyclic2 *make( RtReadWriteSg & aSg,
unsigned int  modeID,
long   ctrlFlags,
WaveShape  aShape,
TimingType  timingType,
double   initTiming,
double   timing,
double   initEndL,
double   endLevel,
long   segCount,
RtAction * endAction ,
MStatus  * status = 0);
```
7.10.6 RtCyclic

An RtCyclic will switch to the specified control mode, and generate a series of segments between two end levels. This subclass has the following profile-specific attributes:

**shape** — One of the three shapes shown in Figure 10.

**amplitude** — The peak amplitude for the waveform. No mean is specified. RtCyclic will use the current command value as the mean.

---

**Figure 10 — RtCyclic and RtSweep waveshapes**

---
frequency — The frequency to be used for all segments.

phase — The profile begins and ends at this phase within the waveform. For example, 0° phase generates a sine, and 90° phase generates a cosine. If phase is not zero, the waveform will step to the initial value. Therefore, it is typical to use BEGIN_SOFT when specifying a non-zero phase.

cycle count — The total number of cycles (2 segments) to generate. If this is -1, the profile will never stop on its own. Any other negative value is an error.

Methods exist to change individual attributes, or change them all at once. The RtCyclic class explicitly supports changing most parameters while it is active. The changes will begin to take effect immediately. Changes in frequency, amplitude, and phase take effect slowly, so the changes do not cause large velocities (i.e. generate steps). The program can change most of the parameters while the profile is active, either with explicit calls, or by getting a Number object, and changing the value through it.

Changes in waveshape, control mode, and most of the control flags will not affect an active RtCyclic.

See Section 7.8.4 on page 69 for a discussion of dynamically changing parameters.

Profile begin/end flags and hold/stop behavior flags operate as expected. Since the current value is used as mean, the relative flags have no effect on the output.

The method RtSegGen::getTrace returns percent completion of the cycle, defined as two segments (value is between zero and one).

### Related RtCyclic methods:

<table>
<thead>
<tr>
<th>Related RtCyclic methods:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtCyclic * make(</td>
<td>RtReadWriteSg &amp; aSg,</td>
</tr>
<tr>
<td>unsigned int modeID,</td>
<td>ctrlFlags,</td>
</tr>
<tr>
<td>unsigned int ctrlFlags,</td>
<td>aShape,</td>
</tr>
<tr>
<td>WaveShape aShape,</td>
<td>amplitude,</td>
</tr>
<tr>
<td>double frequency,</td>
<td>startingPhase,</td>
</tr>
<tr>
<td>double cycleCount,</td>
<td>endAction,</td>
</tr>
<tr>
<td>long status * status = 0)</td>
<td></td>
</tr>
<tr>
<td>static RtCyclic * make(</td>
<td>RtReadWriteSg &amp; aSg,</td>
</tr>
<tr>
<td>unsigned int modeID,</td>
<td>ctrlFlags,</td>
</tr>
<tr>
<td>unsigned int ctrlFlags,</td>
<td>aShape,</td>
</tr>
<tr>
<td>WaveShape aShape,</td>
<td>amplitude,</td>
</tr>
<tr>
<td>double frequency,</td>
<td>startingPhase,</td>
</tr>
<tr>
<td>double cycleCount,</td>
<td>endAction,</td>
</tr>
<tr>
<td>long startAction,</td>
<td>status = 0)</td>
</tr>
<tr>
<td>virtual void ~RtCyclic ( )</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData( unsigned int modeID,</td>
<td></td>
</tr>
<tr>
<td>unsigned int ctrlFlags,</td>
<td></td>
</tr>
<tr>
<td>WaveShape shape,</td>
<td></td>
</tr>
<tr>
<td>double ampl,</td>
<td></td>
</tr>
<tr>
<td>double freq,</td>
<td></td>
</tr>
<tr>
<td>double phase,</td>
<td></td>
</tr>
</tbody>
</table>
7.10.7 RtSweep

An **RtSweep** profile is very similar to an **RtCyclic** profile, except it generates a periodic waveform where the frequency continually changes (typically a sine sweep). This subclass has the following profile-specific attributes:

- **shape** — One of the three shapes shown in Figure 10 on page 81.

- **amplitude** — The peak amplitude for the waveform. No mean is specified. The profile will use the current command value as the mean.

- **frequency** — The waveform will begin at this frequency. This can be manually manipulated through `setFrequency(0, newValue)` and `getFrequency(0)`. Changing this value while in **Sweep_Dwell** mode will cause the generator to ramp the current frequency to this new value. Changing this value while the profile is sweeping has no effect.
current frequency— (read-only) This is the current frequency being generated. If enableFreqParmUpdate() has been called, the Number object returned by getFreqParm() will be periodically updated with the current frequency while it is sweeping. This value is also available through getFrequency(3).

phase — The profile begins and ends at this phase within the waveform. For example, 0° phase generates a sine, and 90° phase generates a cosine. If phase is not zero, the waveform will step to the initial value. Therefore, it is typical to use BEGIN_SOFT when specifying a non-zero phase.

frequencyLowLimit — This is the lower limit for frequency, where the sweep will dwell or reverse direction when sweeping down. This can be manually manipulated through setFrequency(1, newValue) and getFrequency(1).

frequencyUpLimit — This is the upper limit for frequency, where the sweep will dwell or reverse direction when sweeping up. This can be manually manipulated through setFrequency(2, newValue) and getFrequency(2).

sweepType — LINEAR_SWEEP, or LOG_SWEEP

sweepRate — This is the rate at which the frequency changes (Hz/Sec or Decades/Sec).

sweepCount — The total number of sweep reversals. Thus, a value of one will just sweep to the first limit (up or down). A value of 2 will sweep to the first limit, turn around, and sweep to the other one. The profile is then complete. A value of -1 will keep reversing indefinitely and the profile never completes.

sweepBehavior — SWEEP_SINGLE or SWEEP_CONTINUOUS. This determines if the profile dwells when it reaches a frequency lower or upper limit. If SWEEP_SINGLE is selected, the profile will dwell at each frequency limit. SWEEP_CONTINUOUS will cause the sweep to automatically reverse when a frequency limit is reached.

sweepMode — SWEEP_DOWN, SWEEP_DWELL, or SWEEP_UP. This determines the initial sweep direction.

The RtSweep class explicitly supports changing most parameters while it is active. The changes will begin to take effect immediately. Changes in the floating-point parameters take effect slowly, so the changes do not cause large velocities (i.e., generate steps). The program can change most of the parameters while the profile is active, either with explicit calls, or by getting a Number or OneOfList object, and changing the value through it.

The sweepCount and sweepBehavior interact as shown in this table:

<table>
<thead>
<tr>
<th>SweepCount</th>
<th>sweepBehavior</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>SWEEP_SINGLE</td>
<td>Single Sweep up/down and dwell at each frequency limit. (The profile never ends).</td>
</tr>
<tr>
<td>-1</td>
<td>SWEEP_CONTINUOUS</td>
<td>Sweep up/down continuously without dwelling. (The profile never ends).</td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>SWEEP_SINGLE</td>
<td>Single Sweep up/down and dwell at each frequency limit. Profile ends when sweep count is reached.</td>
</tr>
<tr>
<td>&gt;= 0</td>
<td>SWEEP_CONTINUOUS</td>
<td>Sweep up/down continuously without dwelling. Profile ends when sweep count is reached.</td>
</tr>
</tbody>
</table>

Changes in waveshape, control mode, and most of the control flags will not affect an active RtSweep.
See Section 7.8.4 on page 69 for a discussion of dynamically changing parameters.

The **SweepMode** enumeration contains the values SWEEP_DOWN=-1, SWEEP_DWELL=0, and SWEEP_UP=1. The **OneOfList** returned by **getSweepMode()** also has these selections, but the indexes of the selections are zero, one, and two respectively. Unlike most other enumerations, in this case the enumeration and the index do not correspond.

The **OneOfList** returned by **getSweepMode()** is automatically updated with the current sweep mode. However, if the lower and upper frequencies are close together, the sweep mode may be changing quickly. To avoid locking the system with notification events, when the sweep mode changes quickly, the **OneOfList** may not be updated on every change; however, the application will see the last change.

Profile begin/end flags and hold/stop behavior flags operate as expected. Since the current value is used as mean, the relative flags have no effect on the output.

The method **RtSegGen::getTrace** returns percent completion of the cycle, defined as two segments (value is between zero and one).

<table>
<thead>
<tr>
<th>Related RtSweep methods:</th>
<th>RtReadWriteSg &amp;aSg,</th>
<th>RtReadWriteSg &amp;aSg,</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtSweep * make(</td>
<td>modeID,</td>
<td>modeID,</td>
</tr>
<tr>
<td>unsigned int</td>
<td>ctrlFlags,</td>
<td>ctrlFlags,</td>
</tr>
<tr>
<td>unsigned int</td>
<td>aShape,</td>
<td>aShape,</td>
</tr>
<tr>
<td>WaveShape</td>
<td>frequency,</td>
<td>frequency,</td>
</tr>
<tr>
<td>double</td>
<td>phase,</td>
<td>phase,</td>
</tr>
<tr>
<td>double</td>
<td>freqLoLimit,</td>
<td>freqLoLimit,</td>
</tr>
<tr>
<td>double</td>
<td>freqUpLimit,</td>
<td>freqUpLimit,</td>
</tr>
<tr>
<td>SweepType</td>
<td>sweepType,</td>
<td>sweepType,</td>
</tr>
<tr>
<td>double</td>
<td>sweepRate,</td>
<td>sweepRate,</td>
</tr>
<tr>
<td>long</td>
<td>sweepCount,</td>
<td>sweepCount,</td>
</tr>
<tr>
<td>SweepBehavior</td>
<td>swpBehavior,</td>
<td>swpBehavior,</td>
</tr>
<tr>
<td>SweepMode</td>
<td>initSwpMode,</td>
<td>initSwpMode,</td>
</tr>
<tr>
<td>RtAction *</td>
<td>endAction,</td>
<td>endAction,</td>
</tr>
<tr>
<td>MStatus *</td>
<td>status = 0);</td>
<td>status = 0);</td>
</tr>
<tr>
<td>virtual ~RtSweep()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData</td>
<td>modeID,</td>
<td>modeID,</td>
</tr>
<tr>
<td>unsigned int</td>
<td>ctrlFlags,</td>
<td>ctrlFlags,</td>
</tr>
</tbody>
</table>

// Linear or Log
// Single or Continuous
// Down, dwell, up
// Linear or Log
// Single or Continuous
// Down, dwell, up
virtual void setAllData ( unsigned int modeID ,
unsigned int ctrlFlags ,
WaveShape shape ,
double ampl ,
double startFreq ,
double phase ,
double freqLoLimit ,
double freqUpLimit ,
SweepType swpType ,
double swpRate ,
long swpCount ,
SweepBehavior swpBehavior,
SweepMode swpMode ,
RtAction * endAction ,
MStatus * status = 0 )

virtual void adjustAllData ( WaveShape shape ,
double ampl ,
double startFreq ,
double phase ,
double freqLoLimit ,
double freqUpLimit ,
SweepType swpType ,
double swpRate ,
long swpCount ,
SweepBehavior swpBehavior,
SweepMode swpMode ,
MStatus * status = 0 )

virtual void setShape ( WaveShape shape , MStatus * status = 0 )
virtual WaveShape getShape ( MStatus * status = 0 )

virtual void setFrequency ( unsigned int whichFreq , double freqValue , MStatus * status = 0 )
virtual double getFrequency ( unsigned int whichFreq , MStatus * status = 0 )
virtual void setAmplitude ( double amplValue , MStatus * status = 0 )
virtual double getAmplitude ( MStatus * status = 0 )
virtual void setPhase ( double phaseValue , MStatus * status = 0 )
virtual double getPhase ( MStatus * status = 0 )
virtual void setSweepMode ( SweepMode swpMode , MStatus * status = 0 )
virtual SweepMode getSweepMode ( MStatus * status = 0 )
virtual void setSweepType ( SweepType swpType , MStatus * status = 0 )
virtual SweepType getSweepType ( MStatus * status = 0 )
virtual void setSweepRate ( SweepType swpType , double swpRate , MStatus * status = 0 )
virtual double getSweepRate ( SweepType swpType , MStatus * status = 0 )
The **RtRandom** profile plays out a random waveform with specified frequency characteristics. This is implemented as a white-noise generator followed by a filter with the specified frequency and shape characteristics.

This subclass has the following profile-specific attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>shape</strong></td>
<td>This determines the shape of the frequency-domain envelope. The supported shapes are: <strong>ONE_OVER_F2</strong>, <strong>ONE_OVER_F</strong>, <strong>FLAT</strong>, <strong>F</strong>, and <strong>F_SQUARE</strong>. These shape names describe the shape of the frequency domain envelope between the lower and upper cut-off frequencies.</td>
</tr>
<tr>
<td><strong>lower frequency</strong></td>
<td>The low frequency contained in the waveform.</td>
</tr>
<tr>
<td><strong>upper frequency</strong></td>
<td>The highest frequency component in the waveform.</td>
</tr>
<tr>
<td><strong>amplitude</strong></td>
<td>The RMS amplitude for the waveform. No mean is specified. <strong>RtRandom</strong> will use the current command value as the mean.</td>
</tr>
<tr>
<td><strong>seed</strong></td>
<td>The seed for the random number generator. This can be specified to generate a repeatable waveform. For example, by specifying the same seed on several channels, the exact same waveform will be played out on each channel. If the seed is not specified, then a random seed will be used.</td>
</tr>
<tr>
<td><strong>duration</strong></td>
<td>The length of time for the profile to remain active before ending on its own. If a -1 is specified, the profile will run indefinitely.</td>
</tr>
</tbody>
</table>

The **RtRandom** class explicitly supports changing most parameters while it is active. The changes will begin to take effect immediately. Changes in amplitude take effect slowly, so the changes do not cause large velocities (i.e. generate steps). Changes to the shape or the frequencies will immediately change the
filter coefficients. The program can change most of the parameters while the profile is active, either with explicit calls, or by getting a **Number** or **OneOfList** object, and changing the value through it.

Changes in waveshape, control mode, and most of the control flags will not affect an active **RtSweep**.

See Section 7.8.4 on page 69 for a discussion of dynamically changing parameters.

Profile begin/end flags and hold/stop behavior flags operate as expected. Since the current value is used as mean, the relative flags have no effect on the output.

If **duration** is specified, the method **RtSegGen::getTrace** returns percent completion of the profile. Otherwise, it is unused.

---

**Related RtRandom methods:**

```
static RtRandom * make( RtReadWriteSg & aSg,  
unsigned int  modeID,  
unsigned int  ctrlFlags,  
WaveShape aShape,  
double  ampl,  
double  lowFreq,  
double  uppFreq,  
unsigned long * seed,  
double  duration,  
RtAction * endAction,  
MStatus * status = 0);

static RtRandom * make( RtReadWriteSg & aSg,  
unsigned int  modeID,  
unsigned int  ctrlFlags,  
WaveShape aShape,  
double  ampl,  
double  lowFreq,  
double  uppFreq,  
unsigned long * seed,  
double  duration,  
RtAction * endAction,  
RtAction * startAction,  
MStatus * status = 0);
```

```
virtual ~RtExternal()  
virtual void  setAllData( unsigned int  modeID,  
unsigned int  ctrlFlags,  
WaveShape shape,  
double  ampl,  
double  lowFreq,  
double  uppFreq,  
unsigned long * seed,  
double  duration,  
RtAction * endAction,  
MStatus * status = 0 )
```

```
virtual void  setAllData( unsigned int  modeID,  
unsigned int  ctrlFlags,  
WaveShape shape,  
double  ampl,  
double  lowFreq,  
double  uppFreq,  
unsigned long * seed,  
```
7.10.9 RtExternal

An **RtExternal** profile uses the value on a specified signal as the segment generation command source. The only processing the segment generator will do is to pass the signal through the span and set point controls.

This subclass has the following profile-specific attributes:

- **period** — The length of time for the profile to remain active before ending on its own. If a `-1` is specified, the profile will run indefinitely.

- **Signal** — The signal to use as the command source.

Methods exist to change individual attributes, or change them all at once. Once this profile starts executing, changes in the parameters do not affect the output.

Any floating-point signal in the station with the same dimension as the control mode can be used as a command source. Typically the “External Signal” defined in Station Builder would be used. This selection is available through the `RtSegGen::ext()` method.

**Note:**

While any signal matching the dimension can be used, some care needs to be taken to avoid specifying signals that would cause a feedback path through the command generator. For example, using the control mode’s feedback signal as a command source would result in open-loop control.
Tapered begin/end is typically used on **RtExternal** profiles to avoid causing a step when activating it. Hold/stop behavior control flags operate as expected. Relative has no effect.

If `period` is specified, the method **RtSegGen::getTrace** returns percent completion of the profile. Otherwise, it is unused.

### Related RtExternal methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtExternal * make( RtReadWriteSg &amp; aSg, unsigned int modeID, unsigned int ctrlFlags, double period, RtFloatSig &amp; signal, RtAction * endAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>static RtExternal * make( RtReadWriteSg &amp; aSg, unsigned int modeID, unsigned int ctrlFlags, double period, RtFloatSig &amp; signal, RtAction * endAction, RtAction * startAction, MStatus * status = 0);</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtExternal ( )</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned int ctrlFlags, double period, RtFloatSig &amp; signal, RtAction * endAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setAllData ( unsigned int modeID, unsigned int ctrlFlags, double period, RtFloatSig &amp; signal, RtAction * endAction, RtAction * startAction, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setPlayoutPeriod ( double aPeriod, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual double getPlayoutPeriod ( MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual void setSignal ( RtFloatSig &amp; aSig, MStatus * status = 0 );</td>
<td></td>
</tr>
<tr>
<td>virtual RtFloatSig * getSignal ( MStatus * status = 0 );</td>
<td></td>
</tr>
</tbody>
</table>

### 7.10.10 RtSeries

An **RtSeries** profile plays out a series of levels, with a constant interval between them. This is typically used for time-history data.

This subclass has the following profile-specific attributes:

**interval** — The time interval between successive points in the array.
array of data levels — The data levels to be played out. These data levels are maintained in the object, not in the application's array. When the application specifies a new array of data levels, it replaces the old data.

phase — This specifies where in the array to start. It is an index into the array. The complete array will be played out. If it starts in the middle, it will end in the middle. However, this attribute is treated like a phase, in that it is ramped in. That is, it will start at the beginning, and play out the first points faster, so as to reach the correct phase. The design of this attribute is likely to change in a subsequent version.

nCount — The number of times to play out the array.

There is no waveshape specified. The up-sample filter is normally used to smooth the output of an RtSeries profile.

The method RtSegGen::getTrace returns percent completion of the profile.

<table>
<thead>
<tr>
<th>Related RtSeries methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtSeries * make( RtReadWriteSg &amp;aSg, unsigned int modeID, unsigned int ctrlFlags, double interval, long dataSize, float * dataPtr, double phase, long nCount, RtAction * endAction, MStatus * status = 0);</td>
</tr>
<tr>
<td>static RtSeries * make( RtReadWriteSg &amp;aSg, unsigned int modeID, unsigned int ctrlFlags, double interval, long dataSize, float * dataPtr, double phase, long nCount, RtAction * endAction, MStatus * status = 0);</td>
</tr>
<tr>
<td>virtual ~RtSeries ()</td>
</tr>
<tr>
<td>virtual void setData ( unsigned int ctrlFlags, double interval, long dataSize, float * dataPtr, double phase, long nCount, RtAction * endAction, MStatus * status = 0 );</td>
</tr>
<tr>
<td>virtual void setData ( unsigned int ctrlFlags, double interval, long dataSize, float * dataPtr, double phase, long nCount, RtAction * endAction, MStatus * status = 0 );</td>
</tr>
</tbody>
</table>
7.10.11 RtBlock

An **RtBlock** is a container of other **RtProfile** objects. The application creates other **RtProfile** objects and adds a reference to them to the **RtBlock**. Additionally, a repeat count can be specified for the **RtBlock**. When an **RtBlock** is played out, it will play each referenced **RtProfile** object in order, repeating the whole sequence the specified number of times.

Since the **RtBlock** class is itself a subclass of **RtProfile**, instances can be nested arbitrarily deep.

**RtBlocks** do not own the profiles added to them; a profile can be inserted into a block multiple times or put into different blocks. (However, attempting to create a circular reference is an error).

Also, since **RtBlock** is a subclass of **RtProfile**, it is owned by the specified segment generator, and all the profiles added to it must be owned by the same segment generator.

There are four different forms of **RtBlock::add()** that can be used to add profiles to the block. They all add profiles to the end of the list. The block does not take ownership of any of the collection objects used to specify the profiles.

The method **remove()** remove all references to the specified profile from the **RtBlock**.

You cannot add or remove profiles from a block while it is queued.

The method **passesDone()** can be used while an **RtBlock** is being played out to determine how far it has progressed.

### Related RtBlock methods:

- static RtBlock * make ( RtReadWriteSg & aSg , unsigned int modeID , long ctrlFlags , RtAction * endAction , MStatus * status = 0 )
- static RtBlock * make ( RtReadWriteSg & aSg , unsigned int modeID , long ctrlFlags , RtAction * startAction , RtAction * endAction , MStatus * status = 0 )
- virtual void add ( RtProfile * aProf , MStatus * status = 0 )
- virtual void add ( GenList * aList , MStatus * status = 0 )
- virtual void add ( VOrdCollect * aList , MStatus * status = 0 )
- virtual void add ( int numProfiles , RtProfile * * profArray , MStatus * status = 0 )
- virtual void remove ( RtProfile * aProf , MStatus * status = 0 )
- virtual void removeAll ( MStatus * status = 0 )
- virtual void setBlockPasses ( long passes , MStatus * = 0 )
- virtual unsigned long passesDone ( MStatus * status = 0 )

7.10.12 RtCyclic2Buffer

An **RtCyclic2Buffer** can be thought of as an array of **RtCyclic2** objects. It is an optimization to improve throughput when spooling MPT block-arbitrary and phase profiles. The **RtCyclic2Buffer** holds the...
information corresponding to an array of \texttt{RtCyclic2} objects in local memory (in the process), and sends it down to the machine when it is queued. Once the \texttt{RtCyclic2Buffer} has been queued, it may be cleared out, and filled with another set of information. This will not affect the previous information queued in the machine. However, an \texttt{RtCyclic2Buffer} cannot be in the queue more than once. Once it has been queued, it must finish playing out (or be aborted), before it can be queued again.

In addition to the array of \texttt{RtCyclic2} information, this object can have a pass count, indicating how many times to repeat the whole array, and an action to trigger when all the passes are done.

The \texttt{RtCyclic2Buffer} class is not a subclass of \texttt{RtProfile}, but it behaves very much like one. However, since it is not a subclass, it cannot be put into \texttt{RtBlock} or \texttt{RtMsgBlaster} objects, and it also does not support the profile optimizations described in Section 7.11 on page 94.

The typical method to spool a large number of profile file rows is to use two \texttt{RtCyclic2Buffer} objects as follows:

1. Fill buffer A with N rows, and queue it with a “buffer A done” action.
2. Fill buffer B with N rows, and queue it with a “buffer B done” action.
3. When buffer A done action fires, clear, refill, and queue buffer A again.
4. When buffer B done action fires, clear, refill, and queue buffer B again.
5. Continue steps 3 and 4 until all the data has been played out.

### Related RtCyclic2Buffer methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>static \texttt{RtCyclic2Buffer * make}</td>
<td>\texttt{(RtReadWriteSg &amp;aSg, RtAction *endAction, MStatus *status = 0)}</td>
</tr>
<tr>
<td>static \texttt{RtCyclic2Buffer * make}</td>
<td>\texttt{(RtReadWriteSg &amp;aSg, RtAction *endAction, RtAction *startAction, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{add()}</td>
<td>\texttt{unsigned int modeID, unsigned int ctrlFlags, WaveShape shape, TimingType timingType, double initTiming, double timing, double initEndL, double endLevel, long nCount, RtAction *endAction, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{add()}</td>
<td>\texttt{unsigned int modeID, unsigned int ctrlFlags, WaveShape shape, TimingType timingType, double initTiming, double timing, double initEndL, double endLevel, long nCount, RtAction *endAction, RtAction *startAction, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{clear() }</td>
<td>\texttt{}</td>
</tr>
<tr>
<td>const \texttt{Cyclic2Parms * getParameters}</td>
<td>\texttt{(unsigned int index, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{setMode()}</td>
<td>\texttt{unsigned int index, unsigned int modeID, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{setCtrlFlags()}</td>
<td>\texttt{unsigned int index, unsigned int ctrlFlags, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{setShape()}</td>
<td>\texttt{unsigned int index, enum WaveShape shape, MStatus *status = 0)}</td>
</tr>
<tr>
<td>void \texttt{setTiming()}</td>
<td>\texttt{unsigned int index, enum TimingType timingType, double initTiming, double timing, MStatus *status = 0)}</td>
</tr>
</tbody>
</table>
7.11 Optimizing Profile Updating

RtSeries and RtCyclic2Buf objects can be used to spool a lot of data through the segment generator very quickly. The process is to fill one buffer and queue it, then fill a second and queue it. When the first one has completed, fill it again and queue it again. For these two object types, the Rt class is explicitly designed to transfer the data efficiently into the machine.

However, there are some applications where the segments to be played out require a variety of different profile types. You can use RtBlock objects to collect these different profile object together so you can queue them efficiently, however you still need to individually fill the profiles before queuing them the second time.

There are two somewhat related mechanisms to address this problem:

- Postponing updates to RtProfile attributes
- Bundling RtProfile updates together with the RtMsgBlaster class

7.11.1 Postponing updates to RtProfile attributes

When parameter updating for a profile is “postponed”, most changes to the attributes of the profile are stored locally at the RT layer, and are not immediately pushed into the machine. They will then be pushed down efficiently when

- The profile is un-postponed
- updateParams() is called
- The profile is queued (which actually calls updateParams()).

This mechanism provides value when used with RtBlock objects. Postponing an RtBlock will cause all profiles it references to become postponed. So the typical sequence is to:

1. Postpone the RtBlock.
2. Change the attributes of all the profiles (directly or indirectly) referenced by the block.
3. Queue the **RtBlock**. This will call **updateParams()** on the block, which will bundle the parameters of all the modified profiles that the block references together and efficiently push them into the machine. Then it will queue the block.

4. Unpostpone the block, or leave it postponed for a subsequent update.

The **updateParams()** mechanism only copies the attributes of an **RtProfile** if they have been modified since the profile was postponed or the last time **updateParams()** was called.

The postpone state is a counter. To clear the postpone state of an object, **postpone(false)** must be called for each call to **postpone(true)** that was made. This allows having a profile referenced multiple times within a tree of **RtBlock** objects. Adding and removing **RtProfile** objects from an **RtBlock** that is postponed updates the postpone state of the **RtProfile** appropriately.

Remember, properly using this “Postpone” feature does not change what is played out. It only makes the transfer of the updated information more efficient.

**Note**

The postpone feature is supported by different **RtProfile** subclasses to varying degrees. Specifically, **RtExternal** and **RtSeries** do not support this feature.

**Related RtProfile methods:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual</td>
<td>postpone ( boolean aState, MStatus * status = 0 )</td>
<td>postpone() to postpone and efficiently update all the profiles in the list. This updateParams() bundles all the required updated parameters across all the channels into one call into the machine.</td>
</tr>
<tr>
<td>bool</td>
<td>isPostponed()</td>
<td>isPostponed()</td>
</tr>
<tr>
<td>virtual boolean</td>
<td>updateParams ( MStatus * status = NIL )</td>
<td>updateParams()</td>
</tr>
</tbody>
</table>

### 7.11.2 RtMsgBlaster class

The **RtMsgBlaster** class allows you to use the **RtProfile** postpone mechanism in more situations than those described in the previous section.

An **RtMsgBlaster** object maintains an ordered list of **RtProfile** objects that may be from the same or different segment generators. The application creates an **RtMsgBlaster** object and adds the **RtProfile** objects to it.

Specifically, you can:

- Use **postpone()** and **updateParams()** to postpone and efficiently update all the profiles in the list. This updateParams() bundles all the required updated parameters across all the channels into one call into the machine.

- Use **queue()** to queue all the profiles in the list. This is done with a single call into the machine. However, unlike **RtBlock**, each profile is individually put into its segment generator’s queue. (If the RtMsgBlaster is postponed, it will first call **updateParams()** on itself.)

**Related RtMsgBlaster methods:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>RtMsgBlaster * make ( RtStation &amp; stn, MStatus * status = NIL )</td>
<td>make() to create an <strong>RtMsgBlaster</strong> object with the specified station and status.</td>
</tr>
<tr>
<td>static</td>
<td>RtMsgBlaster * make ( RtStation &amp; stn, RtProfile * profObj, MStatus * status = NIL )</td>
<td>make() to create an <strong>RtMsgBlaster</strong> object with the specified station, profile object, and status.</td>
</tr>
</tbody>
</table>
static RtMsgBlaster * make ( RtStation & stn , GenList * profileList , MStatus * status = NIL )

void add ( RtProfile * profObj , MStatus * status = NIL )
void add ( GenList * profileList , MStatus * = NIL )
void add ( VOrdCollect * aList , MStatus * status )
void add ( int numProfiles , RtProfile * * profArray , MStatus * status = NIL )

void remove ( RtProfile * obj , MStatus * = NIL )
void remove ( GenList * profileList , MStatus * = NIL )
void removeAll ( MStatus * status = NIL )
boolean updateParams ( MStatus * status = NIL )

void queue ( MStatus * status = NIL )
void postpone ( boolean aState , MStatus * status = NIL )
bool isPostponed ( )
int count ( )
Chapter 8
Data Acquisition

8.1 Data Acquisition Overview

Data acquisition consists of collecting data from one or more signals over a period of time, and presenting it to the application. The application may do with the data as it wishes.

Data may be collected on a constant-interval basis. This is called timed data acquisition. Data may also be collected based on peak/valleys, level crossing, etc.

The process of data acquisition involves configuring up to three types of components:

- One acquisition object (\texttt{RtAcq} class),
- Zero or more buffering objects (\texttt{RtBuffer} subclass), and
- One or more triggering objects (\texttt{RtTrigger} subclass).

Figure 11 shows an example of the relationship between these three types of classes. It shows an acquisition that collects simultaneous data on three signals, based on finding peaks and valleys on the “Axial Load” signal.

The acquisition object is the main entity involved in collecting data. Through it, the application specifies how much data to collect.

One or more triggering objects are needed to specify when data is to be collected. For example, peak/valley data acquisition is accomplished by connecting a reversal trigger up to the acquisition. It is possible to hook more than one trigger up to a single acquisition object. Data will be sampled whenever any trigger fires.

\textbf{Note:}

The triggers discussed here are \textit{sampling triggers}. These triggers determine when individual sample points are collected. This is unrelated to triggering the \textit{start} of data acquisition.
The buffering objects specify what data to collect, and how to buffer it to the application. Currently, there are three different classes of buffering objects:

1. Linear buffers — collects a specified amount of data and quits.
2. Linear contiguous buffers — spools an indefinite amount of data to the application.
3. Circular buffers — collects data indefinitely, keeping just the last buffer full of data.

A buffering object can collect data from either \texttt{RtFloatSig} or \texttt{RtIntegerSig} objects.

An application may connect an \texttt{RtIntegerSig} as a gate for the \texttt{RtAcq} to enable/disable data acquisition based on the value of the signal. Acquisition is enabled when the gate signal’s value is non-zero. A typical application of this is cycle-based acquisition using an \texttt{RtIntRngDt} object to enable acquisition during specific cycles of the command. See Section 8.5.2 on page 119.

**High-speed data acquisition** is an optional mechanism that can collect data faster than the system rate. It uses a high-rate stream of data that is only available on signals connected to certain types of conditioners. This mechanism uses the same \texttt{RtAcq} and \texttt{RtBuffer} classes, but uses a different trigger object. This is described in Section 8.6 on page 121.

**Time-history data acquisition** is yet another optional mechanism for collecting data. It collects data from yet another stream that is filtered to avoid aliasing, and is synchronized with command generation in special ways. This is described in Section 0 on page 122.

### 8.2 Sequences (RtSequence)

Internally, the machine is a sampled system. Most objects in the system that process real-time data, or watch for events do so by periodically sampling their inputs. This applies to data acquisition, limit detection, digital inputs, etc. Servo-control and function generation are also sampled activities. The basic mechanism for these objects to implement their sampling activity is to hook them into a sequence.

A sequence can be thought of as a clock running at a particular rate. From the application’s perspective, the system provides the following sequences:

**HIGH_RATE_SEQUENCE** (or “HI_RATE_SEQ”) — runs at the system clock rate. The rate of this sequence is specified in the HWI file with the SYSTEM RATE key.

**Note:**

The HWI file can contain another clock rate called HI RATE. This rate is used for RPC data acquisition, and is not related to the HIGH_RATE_SEQUENCE.

**MED_RATE_SEQUENCE** (or “MED_RATE_SEQ”) — The rate of this sequence is specified in the HWI file with the MEDIUM SYSTEM RATE key. The rate defaults to 256.0 Hz if the key is not present.

**LOW_RATE_SEQUENCE** (or “LOW_RATE_SEQ”) — This is specified in the HWI file with the LOW SYSTEM RATE key. The rate defaults to 25.6 Hz if the key is not present.

Applications cannot create sequences. Applications only reference sequences to specify what base sample rate will be used for a data acquisition or limit detector object. Sequences can be referenced by a text name, or by an enumeration. In the above list, both the enumeration label and the text name are shown.
Normally, all data acquisition and limit detection is done at the HIGH_RATE_SEQUENCE. The data on virtually all of the signals is sampled at this rate. If a lower sequence is used for data acquisition, there may be jitter in the resulting data.

The method `rate()` returns the rate of the sequence in Hz.

<table>
<thead>
<tr>
<th>Related RtSequence methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>double rate( MStatus * sp = 0 )</td>
</tr>
<tr>
<td>static RtSequence * find( RtStation &amp; stn, const Text * name, MStatus * status = 0 )</td>
</tr>
<tr>
<td>static RtSequence * find( RtStation &amp; stn, SequenceType seqType, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtSystem methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtSequence * findSequence(const Text &amp;name, MStatus *status = 0)</td>
</tr>
</tbody>
</table>

### 8.3 Specifying when data is collected (RtTrigger)

A sampling trigger is an object that watches for some significant event, and tells the `RtAcq` when to sample and/or store the data. There are currently triggers for time, level, delta level, peak/valley level, peak/valley delay and min/max. Multiple triggers may be connected to a single `RtAcq` object. Data will be sampled when any of the triggers fires. These triggers are described in the following sections.

**Note:**

Combining an `RtReversalTrigger` with any other trigger, including another `RtReversalTrigger`, may cause data to be stored out of order. See Section 8.3.4 on page 104 or Section 8.3.5 on page 106 for details.

Occasionally, it is useful to use an `RtTrigger` object in the absence of an `RtAcq`. The most common example of this is running max/min acquisition (discussed in Section 8.3.6 on page 107). This is generally useful when you are only interested in the last data points.

Sampling triggers that are created on an `RtAcq` are owned by that acquisition object, and will be deleted when the `RtAcq` is deleted. If the `RtReversalTrigger` is created without an acquisition object (as discussed in Section 8.3.6 on page 107), then the application is responsible to delete the trigger object.

`RtTrigger` objects used by in the absence of an `RtAcq` object need to be started and stopped explicitly by the application. The start/stop state of `RtTrigger` objects that are created on an `RtAcq`, is managed by the `RtAcq` object. The application typically should not call start or stop on these trigger objects owned by an `RtAcq`.

### 8.3.1 Trigger at constant time interval (RtTimeTrigger)

An `RtTimeTrigger` is used to repetitively trigger the `RtAcq` at a specified time interval.
The **RtTimeTrigger** is the class used by Basic TestWare, MPT and other applications to implement “Timed Data Acquisition”.

Example 5 shows typical code for making and configuring **RtTimeTrigger** and **RtAcq** objects for timed data acquisition.

This example performs the following functions:

1. Find the high-rate **RtSequence**
2. Find the “Axial Load” **RtFloatSig**.
3. Create an **RtAcq**, connecting it to the sequence.
4. Create an **RtTimeTrigger** for the **RtAcq**.
5. Configure the **RtTimeTrigger** for the desired time interval.
6. Get the actual sample period. The specified period is rounded to the nearest integer multiple of the **RtSequence** period.
7. (Optionally) Specify the number of samples to collect, and connect a callback to be invoked when done. The callback must take zero parameters, and return **boolean**. The return value is not meaningful.
8. Set up the buffers (not shown). This is discussed in Section 8.4 on page 109.
9. Start the **RtAcq**.

```cpp
RtSequence *acqSeq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE);
RtFloatSig *floatSig = RtFloatSig::find(rtStn, &Text("Axial Load"));

// make and configure the Acq.
rtAcq = RtAcq::make(*rtStn, acqSeq);

// make and configure reversal detector
RtTimeTrigger *rtTime = RtTimeTrigger::make(*rtAcq);
rtTime->setPeriod(tPeriod);

rtAcq->totalPoints(tPoints);
rtAcq->uponDone(CALLBACK0(this, MyClass, localDone, int));

// buffers must now be set up ...

// now start the acq
rtAcq->start();
```

**Figure 12 — Trigger points for RtTimeTrigger**

**Example 5 — Timed data acquisition**
Calling **reset**() or **clear**() on the **RtTimeTrigger** will start the counting for the current interval over again.

### Related RtAcq methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtAcq * make(RtStation &amp; stn, RtSequence *seq, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>static RtAcq * make(RtStation &amp; stn, RtSequence *seq, enum DataStreamTypes dataStreamType, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtAcq()</td>
<td></td>
</tr>
<tr>
<td>void start(MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>void stop(MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>void reset(MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>void hold(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void resume(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void totalTime(double timeInSec, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>double totalTime(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>double time(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void totalPoints(long totalPoints, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>long totalPoints(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>long count(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void uponDone(Callback0Ptr &lt; int &amp; cb, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void uponDoneDiscon(Callback0Ptr &lt; int &amp; cb, MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>

### Related RtTimeTrigger methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtTimeTrigger * make(RtAcq &amp;acq, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtTimeTrigger()</td>
<td></td>
</tr>
<tr>
<td>void setPeriod(double samplePeriod, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>double samplePeriod(MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>void setRate(double sampleRate, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>double sampleRate(MStatus * = 0)</td>
<td></td>
</tr>
</tbody>
</table>

### Related RtTrigger methods (base class of RtTimeTrigger):

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void hold(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void resume(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void reset(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void connectSignal(RtSignal * aSignal, MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>

### Related RtTask methods (base class of RtTrigger):

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void stop(MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>
## 8.3.2 Trigger on change in level (RtDeltaTrigger)

An **RtDeltaTrigger** detects when a specified signal has changed by a certain amount (plus or minus). When a value outside the band is detected, the **RtAcq** will be triggered to sample data, and the trigger will start looking for a change from the new current level.

The **RtDeltaTrigger** is the class used by Basic TestWare, MPT and other applications to implement “Level Crossing Data Acquisition”.

This type of trigger is useful when a signal sometimes changes quickly, and other times changes very little. It will tend to store more data when the signal is changing, and less data when it is not.

Figure 13 shows at what point in the waveform the **RtDeltaTrigger** will sample the data.

Example 7 shows typical code for making and configuring an **RtDeltaTrigger**.

```c
// make and configure reversal detector
RtFloatSig *floatSig = RtFloatSig::find(*rtStn, &Text("Axial Load"));

RtDeltaTrigger *delta = RtDeltaTrigger::make(*rtAcq, floatSig);
delta->setValue( 1.0F );    // 1.0 kN delta
```

### Example 7 — Configuring an RtDeltaTrigger

Calling **reset()** will reset the current internal state back to its initial condition. The current signal value will become the initial level. The **clear()** will set the delta value back to its default (FLT_MAX).

### Related RtDeltaTrigger methods:

```c
static RtDeltaTrigger * make ( RtAcq &rtAcq, RtFloatSig *sig, MStatus * status = 0 )
virtual ~RtDeltaTrigger ( )
double  value ( MStatus * status = 0 )
void    setValue ( double val , MStatus * status = 0 )
```

### Related RtTrigger methods (base class of RtDeltaTrigger):

```c
void    hold ( MStatus * status = 0 )
void    resume ( MStatus * status = 0 )
void    reset( MStatus * status = 0 )
void    clear ( MStatus * status = 0 )
void    connectSignal ( RtSignal * aSignal , MStatus * status = 0 )
```
8.3.3 Trigger on crossing a level (RtLevelTrigger)

An **RtLevelTrigger** detects when a specified signal crosses a certain level (either direction). When the signal crosses the specified level, the **RtAcq** will be triggered to sample data, and trigger will start looking for the signal to cross back the other way over the level.

This class can be used to implement a “Zero-crossing data acquisition”. This is not used for what is usually called “Level Crossing data acquisition (See Section 8.3.2 on page 102 for that type of acquisition).

If the signal hovers around the specified level, noise could make it cross the level many times; therefore a sensitivity band is specified to avoid multiple triggers in this situation. This sensitivity band is typically a small number, larger than the expected noise level.

Figure 14 shows at what point in the waveform the **RtLevelTrigger** will sample the data.

Example 8 shows typical code for making and configuring an **RtLevelTrigger**.

```cpp
// make and configure level detector
RtFloatSig *floatSig = RtFloatSig::find(*rtStn, &Text("Axial Load"));
RtLevelTrigger *level = RtLevelTrigger::make(*rtAcq, floatSig);
level->value ( 10.0f );  // 10.0 kN level.
level->setNoiseLevelSensitivityBand( 1.0f ); // 1.0 kN noise band.
```

**Example 8 — Configuring an RtLevelTrigger**

Calling **reset()** will reset the current internal state back to its initial condition. The **clear()** method will clear the level and sensitivity back to their default values (both zero).

**Note:**

```
Related RtLevelTrigger methods:
static RtLevelTrigger * make ( RtAcq &rtAcq, RtFloatSig *sig, MStatus * status = 0 )
virtual ~RtLevelTrigger ( )
void setup ( double val , double band , MStatus * status = 0 )
double value ( MStatus * status = 0 )
void setValue ( double val , MStatus * status = 0 )
double noiseLevelSensitivityBand ( MStatus * status = 0 )
void setNoiseLevelSensitivityBand ( double band , MStatus * status = 0 )
```
8.3.4 Trigger on peak/valley using level (RtReversalTrigger)

An **RtReversalTrigger** detects local maximums and minimums. It can be configured for several different algorithms.

The “peak-valley using a level band” method detects peaks and valleys based on the signal reversing direction by more than a specified noise-band.

This is the detector used by Basic TestWare, MPT and other applications to implement “Peak/Valley data acquisition”.

In this mode, when the **RtReversalTrigger** is first started, it determines whether to look for a peak or valley first based on a “delta level” algorithm. No data will be collected until the signal moves by at least the noise-band. Consequently, the initial starting point will never be reported as a peak or valley.

Figure 15 shows at what point in the waveform the peak-valley-band algorithm will sample and store the data.

An **RtReversalTrigger** operates differently than other trigger objects in that there is some delay between when the sample is taken and when it is stored in the buffer. An **RtReversalTrigger** identifies each reversal of the signal as a potential peak or valley. The **RtAcq** temporarily saves the potential peak or valley until it is confirmed as an actual peak or valley, at which time it is stored in the buffer. If an **RtReversalTrigger** is combined with any other trigger object, the delay between sampling of the data and the storing of it in the buffer may cause data to be stored out of order. An application requiring an **RtReversalTrigger** combined with another trigger would normally arrange to acquire a time or cycle count signal so that the correct data ordering can be determined.
Note:

A gate signal connected to the RtAcq may cause a peak or valley to be missed. To be stored in the buffer a sample must be identified as a potential peak or valley and be confirmed as an actual peak or valley during the time data acquisition is enabled.

Example 9 shows typical code for making and configuring an RtReversalTrigger for peak-valley-level operation.

```c
// make and configure reversal detector
RtFloatSig *floatSig = RtFloatSig::find(*rtStn, &Text("Axial Load");
RtReversalTrigger *rdt = RtReversalTrigger::make(*rtAcq, floatSig);
rdt->setMode(RDT_PEAK_VALLEY_LEVEL);
rdt->setNoiseLevelSensitivityBand( 1.0f ); // 1.0 kN noise band.
```

Example 9 — Configuring an RtReversalTrigger (RDT_PEAK_VALLEY_LEVEL)

The maximum() and minimum() methods return the last peak and valley detected. These are useful when the trigger is used without an RtAcq (See section 8.3.6 on page 107).

Calling reset will reset the current internal state, returning to the condition of looking for the first peak or valley. The clear() method will return the mode, delay and noise-band parameters back to their default values (i.e., RDT_MIN_MAX, 0, and 0).

Related RtReversalTrigger methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtReversalTrigger* make( RtAcq &amp; acq, RtFloatSig *sig, MStatus * status = 0 )</td>
<td>Make and configure a RtReversalTrigger</td>
</tr>
<tr>
<td>virtual ~RtReversalTrigger ( )</td>
<td>Destructor</td>
</tr>
<tr>
<td>void setup ( RtReversalMode mode, double band, double delay, MStatus * status = 0 )</td>
<td>Set up the reverse trigger mode, delay and noise-band sensitivity</td>
</tr>
<tr>
<td>RtReversalMode mode ( MStatus * status = 0 )</td>
<td>Get the reverse trigger mode</td>
</tr>
<tr>
<td>void setMode ( RtReversalMode mode, MStatus * status = 0 )</td>
<td>Set the reverse trigger mode</td>
</tr>
<tr>
<td>double maximum ( MStatus * status = 0 )</td>
<td>Get the last detected peak</td>
</tr>
<tr>
<td>double minimum ( MStatus * status = 0 )</td>
<td>Get the last detected valley</td>
</tr>
<tr>
<td>double noiseLevelSensitivityBand ( MStatus * status = 0 )</td>
<td>Get the noise-level sensitivity band</td>
</tr>
<tr>
<td>void setNoiseLevelSensitivityBand ( double band, MStatus * status = 0 )</td>
<td>Set the noise-level sensitivity band</td>
</tr>
</tbody>
</table>

Related RtTrigger methods (base class of RtReversalTrigger):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear ( MStatus * status = 0 )</td>
<td>Clear the trigger state</td>
</tr>
<tr>
<td>void hold ( MStatus * status = 0 )</td>
<td>Hold the trigger configuration</td>
</tr>
<tr>
<td>void resume ( MStatus * status = 0 )</td>
<td>Resume the trigger operation</td>
</tr>
<tr>
<td>void reset( MStatus * status = 0 )</td>
<td>Reset the trigger state</td>
</tr>
<tr>
<td>void connectSignal ( RtSignal * aSignal, MStatus * status = 0 )</td>
<td>Connect a signal to the trigger</td>
</tr>
</tbody>
</table>

Related RtTask methods (base class of RtTrigger):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start ( MStatus * status = 0 )</td>
<td>Start the trigger task</td>
</tr>
<tr>
<td>void stop ( MStatus * status = 0 )</td>
<td>Stop the trigger task</td>
</tr>
</tbody>
</table>
8.3.5 Trigger on peak/valley using delay (RtReversalTrigger)

An RtReversalTrigger detects local maximums and minimums. It can be configured for several different algorithms.

The “peak-valley using a delay” method detects peaks and valleys based on the signal reversing direction for a particular length of time.

Figure 16 shows at what point in the waveform the peak-valley-time algorithm will sample the data. This is the same signal as shown for peak-valley-level algorithm in Figure 15. This demonstrates that the two algorithms may detect a different set of peaks. This also demonstrates that if the delay is too large compared to real data on the signal, that some reversals are not reported, even if the signal has significant slope. In addition, since the next peak is not looked for until the previous valley has been accepted or rejected, some points that are not local maximums or minimums may be reported as peaks or valleys.

**Note:**

In this mode, when the RtReversalTrigger is first started, it determines whether to look for a peak or valley based on the relative values of the first two sample points that are different. In the presence of noise, this may cause the second point to be reported as a peak/valley.

An RtReversalTrigger operates differently than other trigger objects in that there is some delay between when the sample is taken and when it is stored in the buffer. An RtReversalTrigger identifies each reversal of the signal as a potential peak or valley. The RtAcq temporarily saves the potential peak or valley until it is confirmed as an actual peak or valley, at which time it is stored in the buffer. If an RtReversalTrigger is combined with any other trigger object, the delay between sampling of the data and the storing of it in the buffer may cause data to be stored out of order. An application requiring an RtReversalTrigger combined with another trigger would normally arrange to acquire a time or cycle count signal so that the correct data ordering can be determined.

**Note:**

A gate signal connected to the RtAcq may cause a peak or valley to be missed. To be stored in the buffer a sample must be identified as a potential peak or valley and be confirmed as an actual peak or valley during the time data acquisition is enabled.
Example 10 shows typical code for making and configuring an **RtReversalTrigger** for peak-valley-time operation.

```cpp
// make and configure reversal detector
RtFloatSig *floatSig = RtFloatSig::find(*rtStn, &Text("Axial Load"));
RtReversalTrigger rdt = RtReversalTrigger::make(*rtAcq, floatSig);
rdt->setMode(RDT_PEAK_VALLEY_TIME);
rdt->setTimeSensitivityDelayInterval( 0.1f ); // 0.1 sec delay.
```

**Example 10 — Configuring an RtReversalTrigger (RDT_PEAK_VALLEY_TIME)**

The **maximum()** and **minimum()** methods return the last peak and valley detected. These are useful when the trigger is used without an **RtAcq** (See section 8.3.6 on page 107).

Calling **reset()** will reset the current internal state, returning to the condition of looking for the first peak or valley. The **clear()** method will return the mode, delay and noise-band parameters back to their default values (i.e., RDT_MIN_MAX, 0, and 0).

### Related RtReversalTrigger methods:
- **static** RtReversalTrigger * make ( RtAcq &acq, RtFloatSig *sig, MStatus * status = 0 )
- **virtual** ~RtReversalTrigger ( )
- **void** setupTrigger ( RtReversalMode mode , double band , double delay , MStatus * status = 0 )
- **RtReversalMode** mode ( MStatus * status = 0 )
- **void** setMode ( RtReversalMode mode , MStatus * status = 0 )
- **double** timeSensitivityDelayInterval ( MStatus * status = 0 )
- **void** setTimeSensitivityDelayInterval ( double delay , MStatus * status = 0 )
- **double** maximum ( MStatus * status = 0 )
- **double** minimum ( MStatus * status = 0 )

### Related RtTrigger methods (base class of RtReversalTrigger):
- **void** clear ( MStatus * status = 0 )
- **void** hold ( MStatus * status = 0 )
- **void** resume ( MStatus * status = 0 )
- **void** reset( MStatus * status = 0 )
- **void** connectSignal ( RtSignal * aSignal , MStatus * status = 0 )

### Related RtTask methods (base class of RtTrigger):
- **void** start ( MStatus * status = 0 )
- **void** stop ( MStatus * status = 0 )

### 8.3.6 Running max/min acquisition (RtReversalTrigger)

A **RtReversalTrigger** detects local maximums and minimums. It can be configured for several different algorithms.

The “running max/min” method remembers the absolute maximum and minimum value seen on the signal in the time it is active.

---

September 15, 2003  MTS Systems Proprietary  107
Example 11 shows typical code for making and configuring an **RtReversalTrigger** for running max/min operation.

```cpp
// make and configure reversal detector
RtFloatSig *floatSig = RtFloatSig::find(*rtStn, &Text("Axial Load"));
RtReversalTrigger *rdt = RtReversalTrigger::make(*rtStn, 0, acqSeq, floatSig);
rdt->setMode(RDT_MIN_MAX);
rtRdt->start();
```

**Example 11 — Configuring an RtReversalTrigger (RDT_MIN_MAX)**

The application calls **minimum()** and **maximum()** to query for the absolute minimum and maximum value that the **RtReversalTrigger** has seen. Calling **reset()** will reset the current max/min values to the current value on the signal. The **clear()** method will return the mode, delay and noise-band parameters back to their default values (i.e., RDT_MIN_MAX, 0, and 0).

It is possible to hook this sampling trigger to an **RtAcq**. When it triggers the **RtAcq** depends on the mode that is specified:

- **RDT_MIN_MAX** — Triggers every time a new minimum or maximum value is seen.
- **RDT_MIN_ONLY** — Triggers every time a new minimum value is seen.
- **RDT_MAX_ONLY** — Triggers every time a new maximum value is seen.

All these modes always update both the minimum and maximum values. They only differ on when they trigger any **RtAcq** connected to the **RtReversalTrigger**.

<table>
<thead>
<tr>
<th>Related RtReversalTrigger methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtReversalTrigger * make (RtStation &amp; stn, RtSequence *seq, RtFloatSig *sig, MStatus * status = 0)</td>
</tr>
<tr>
<td>static RtReversalTrigger * make (RtAcq &amp; acq, RtFloatSig *sig, MStatus * status = 0)</td>
</tr>
<tr>
<td>virtual ~RtReversalTrigger()</td>
</tr>
<tr>
<td>void setupTrigger (RtReversalMode mode, double band, double delay, MStatus * status = 0)</td>
</tr>
<tr>
<td>RtReversalMode mode (MStatus * status = 0)</td>
</tr>
<tr>
<td>void setMode (RtReversalMode mode, MStatus * status = 0)</td>
</tr>
<tr>
<td>double maximum (MStatus * status = 0)</td>
</tr>
<tr>
<td>double minimum (MStatus * status = 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtTrigger methods (base class of RtReversalTrigger):</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear (MStatus * status = 0)</td>
</tr>
<tr>
<td>void hold (MStatus * status = 0)</td>
</tr>
<tr>
<td>void resume (MStatus * status = 0)</td>
</tr>
<tr>
<td>void reset(MStatus * status = 0)</td>
</tr>
<tr>
<td>void connectSignal (RtSignal * aSignal, MStatus * status = 0)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtTask methods (base class of RtTrigger):</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sequence (RtSequence * seq, MStatus * status = 0)</td>
</tr>
<tr>
<td>void connectToSequence (RtSequence &amp; seq, MStatus * status = 0)</td>
</tr>
</tbody>
</table>
**8.3.7 Trigger on End of Segment (RtEndLevelTrigger)**

An RtEndLevelTrigger detects whenever a specified segment generator reaches the end of a segment. When the segment generator completes a segment, the RtAcq will be triggered to take one sample of data.

The RtEndLevelTrigger defines the end of a segment differently than the segment counters that are maintained by the segment generator. The NO_INCREMENT_COUNT control flag has no impact on what RtEndLevelTrigger triggers on. This class will only trigger on segment completions that are part of RtStay, RtSegment, RtCyclic2, RtCyclic2Buffer, RtCyclic, and RtSweepProfile classes. Further, when using the TRUE_SINE, TRUE_RAMP, and TRUE_SQUARE waveshapes, the RtEndLevelTrigger will trigger at 90° and 270° (the peak and valley).

See Chapter 7 starting on page 57 for more information about the counters on segment generators.

### Related RtEndLevelTrigger methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtEndLevelTrigger * make ( RtAcq &amp; acq, RtSegGen &amp; segGen, MStatus * status = 0 )</td>
<td>Makes an instance of RtEndLevelTrigger</td>
</tr>
<tr>
<td>virtual ~ RtEndLevelTrigger()</td>
<td>Destroys an instance of RtEndLevelTrigger</td>
</tr>
</tbody>
</table>

### Related RtTrigger methods (base class of RtReversalTrigger):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void clear ( MStatus * status = 0 )</td>
<td>Clears the trigger condition</td>
</tr>
<tr>
<td>void hold ( MStatus * status = 0 )</td>
<td>Holds the trigger condition</td>
</tr>
<tr>
<td>void resume ( MStatus * status = 0 )</td>
<td>Resumes the trigger condition</td>
</tr>
<tr>
<td>void reset( MStatus * status = 0)</td>
<td>Resets the trigger condition</td>
</tr>
</tbody>
</table>

### Related RtTask methods (base class of RtTrigger):

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start ( MStatus * status = 0 )</td>
<td>Starts the trigger condition</td>
</tr>
<tr>
<td>void stop ( MStatus * status = 0 )</td>
<td>Stops the trigger condition</td>
</tr>
</tbody>
</table>

---

**8.4 Buffers (RtBuffer)**

Buffers are used by the data acquisition process to store the data, transport it, and present it to the application. There are three buffering mechanisms RtCircularBuffer, RtLinearBuffer, and RtLinearCBuffer. There are many similarities between these buffer classes.

### 8.4.1 Creating buffers (any buffer type)

Buffers are created and connected to an RtAcq with code similar to the following.

This example creates a linear buffer for the acquisition object to collect data from the “Time” and “Axial Load” signals.

```cpp
RtFloatSig *timeSig = RtFloatSig::find(rtStn, &Text("Time"));
RtFloatSig *loadSig = RtFloatSig::find(rtStn, &Text("Axial Load"));
GenList *sigList = new GenList();
```
Example 12 — Creating a data acquisition buffer

The signal list can contain RtFloatSig or RtIntegerSig objects. If both types are attached to the same buffer, extra care must be used when accessing the buffer data.

Buffers are created against an RtAcq object, and are owned by that object. They can be explicitly deleted, but they will be automatically deleted when the RtAcq is deleted.

8.4.2 Accessing buffer data (any buffer type)

An RtBuffer collects data within the machine, copies it up and presents the data to the application in an array. The array may be treated as a 2-dimensional array where the rows of the array contain a value for each signal acquired by the buffer, all sampled at the same instant. Figure 17 shows a typical buffer as a 2-dimensional array and some of the ways that an application may access the data.

Buffers can handle floating point or integer signals. The resulting buffer array will contain floating point, integer data, or a combination of the two. There are two sets of methods for accessing the array, and it is the client’s responsibility to use the appropriate set. Failure to do so will return garbage.

An application will usually use the value() (or iValue) method or the [] operator to access data within the buffer. These functions take care of any offset or unwrap calculations that may have to be done. The count() method returns the number of valid scans of data currently in the buffer. This may not be all the data collected, since some data may still be in temporary buffers within the machine. This pipeline of data is forced into the user-visible array when the RtAcq is done or stopped, or by calling the getData() method. The clear() method is used to discard the current visible data in the buffer. This makes room in the buffer for any data still in the pipeline.

Example 13 shows the typical method for accessing data stored in the buffer.

```
int numScans = rtBuf->count() ;
int numSigs = rtBuf->scanSize() ;

// Write the data to cout in tab separated columns
```
**Example 13 — Accessing buffer data**

An application may choose to access buffer data within the array using a pointer. The `buffer()` (or `iBuffer`) method returns a pointer to the base of array. This pointer will remain constant for the life of the buffer. To use the data pointer an application must take care to perform the offset and unwrap calculations that may be required. The `first()` method returns the index for the first row of the valid data in the buffer and the `scanSize()` method returns the number of signal values in each row.

Example 14 uses calculated pointer values to give the same result as Example 13.

```cpp
int numScans = rtBuf->count() ;
int rowOffset = rtBuf->first() ;
int numSigs = rtBuf->scanSize() ;
int bufSize = rtBuf->size() ;

float *dataArray = rtBuf->buffer() ;

// Write the data to cout in tab separated columns
for( int scan = 0 ; scan < numScans ; scan++ )
{
    // Calculate pointer to the right row of data.
    float *valuePtr = dataArray + ((( rowOffset + scan) % bufSize) * numSigs) ;
    for( int sig = 0 ; sig < numSigs ; sig++ )
    {
        // Access the data using and incrementing the pointer
        cout << *valuePtr++ << '\t' ;
    }
    cout << endl ;
}

// Discard the data that has been written out.
rtBuf->clear() ;
```

**Related RtBuffer methods:**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>float *</td>
<td>buffer(MStatus * status = 0)</td>
</tr>
<tr>
<td>float *</td>
<td>dataPtr(long scanNum, long chan = 0)</td>
</tr>
<tr>
<td>float *</td>
<td>operator[](long scanNum)</td>
</tr>
<tr>
<td>double</td>
<td>value(long scanNum, long chanNum = 0)</td>
</tr>
<tr>
<td>long *</td>
<td>iBuffer(MStatus * status = 0)</td>
</tr>
<tr>
<td>long *</td>
<td>iDataPtr(long scanNum, long chan = 0)</td>
</tr>
<tr>
<td>long *</td>
<td>iValue(long scanNum, long chanNum = 0)</td>
</tr>
</tbody>
</table>
8.4.3 Requesting notification of data (any buffer type)

The application can request to be informed when a certain amount of data has been collected. Applications are commonly interested in when a buffer-full of data is available, but other increments are also possible. The `uponNotify()` method establishes a callback to notify the application of significant events on the buffer. Only one `uponNotify` callback can be applied to a buffer at once. Invoking this method again will cancel the previous assignment.

The signature of this callback includes two integer parameters. The parameters provide the callback routine with the number of valid scans of data in the buffer and the reason why the callback routine is being called. Table 8 describes each of the possible reason codes that an application may see.

<table>
<thead>
<tr>
<th>Reason Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtBuffer::BUFF_INDEX</td>
<td>The buffer object has acquired the number of additional points specified in the <code>setNotifyIndex()</code> method. The buffer has not yet filled and there may be additional data available. This reason code is converted to <code>RtBuffer::BUFF_FULL</code> if the buffer fills.</td>
</tr>
<tr>
<td>RtBuffer::BUFF_FLUSH</td>
<td>Indicates that the buffer now contains the last available data in the “pipeline” at the time the <code>getData()</code> method was called. Additional data may become available. This code will not be sent if there was not data in the pipeline. This code is converted to <code>RtBuffer::BUFF_FULL</code> if the available data fills the buffer.</td>
</tr>
<tr>
<td>RtBuffer::BUFF_STOP</td>
<td>Indicates the last available data in the “pipeline” and no additional data will be collected. This is triggered by a call to the buffers <code>stop()</code> method.</td>
</tr>
<tr>
<td>RtBuffer::BUFF_FULL</td>
<td>The visible data array has filled. The application must call <code>clear()</code> to allow additional data and notifications to be delivered.</td>
</tr>
<tr>
<td>RtBuffer::BUFF_OVERRUN</td>
<td>The data “pipeline” has overrun. The data acquisition sample rate was too high or the application has not cleared the buffer soon enough. The buffer object has been stopped and no additional data will be available.</td>
</tr>
<tr>
<td>RtBuffer::BUFF_RESET</td>
<td>This is triggered by calling <code>reset()</code> on the buffer. It indicates that resetting of the data pipeline has completed. The buffer object will restart when the <code>RtAcq</code> object is next started.</td>
</tr>
<tr>
<td>RtBuffer::ACQ_DONE</td>
<td>Indicates that the <code>RtAcq</code> object has completed the total points or total time set by the application. The acquisition object has stopped and all available data is in the buffer.</td>
</tr>
</tbody>
</table>

Example 15 Shows a typical buffer notify callback routine.

```cpp
boolean MyBuffer::bufferNotifyCB(int reason, int numPoints) {
    switch (reason) {
    case RtBuffer::BUFF_FULL:
    case RtBuffer::BUFF_INDEX:
    case RtBuffer::BUFF_FLUSH:
        writeBuffer(numPoints);
        rtBuf_->clear();
    ```
An application may call the `lock()` method to prevent updates to the buffer contents while it processes the buffer data. This would normally be used if the callback routine signals another thread to process the data. The buffer is effectively locked while the callback routine is active. If all of the processing of buffer data is done within the callback routine, it is not necessary to call `lock()`.

The `unlock()` method must be called after `lock()` to re-enable buffer updates. If there is a buffer notification that can be delivered, the callback routine will be called before `unlock()` returns.

The `getData()` method performs an `unLock()` to enable buffer updates, flushes the data pipeline, and then locks the buffer. Any notification callbacks in the pipeline as well as the BUFF_FULL notification may be delivered before the `getData()` method returns. An application must call `unLock()` after `getData()` if additional callbacks or data are expected to be delivered.

The `clear()` method discards the currently visible data in the buffer. If the buffer is locked, `clear()` will also unlock it.

The buffer notification callback mechanism prevents recursive callbacks from occurring when `unlock()`, `getData()` or `clear()` are called from within the callback routine.

The `RtBuffer` (and therefore also the `RtAcq` that owns it) cannot be deleted from within the notification routine. This will generate a fatal error. Also, if multiple threads are used to access the buffer object, care must be taken to ensure that all threads are done accessing the buffer before it is deleted.

### Related RtBuffer methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setNotifyIndex</code></td>
<td>Sets the notification index</td>
</tr>
<tr>
<td><code>notifyIndex</code></td>
<td>Sends a notification with an index</td>
</tr>
<tr>
<td><code>uponNotify</code></td>
<td>Sends a notification with a callback</td>
</tr>
<tr>
<td><code>uponNotifyDiscon</code></td>
<td>Sends a notification to disconnect a callback</td>
</tr>
<tr>
<td><code>lock()</code></td>
<td>Locks the buffer</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>Clears the buffer</td>
</tr>
<tr>
<td><code>getData()</code></td>
<td>Gets data from the buffer</td>
</tr>
</tbody>
</table>

---

**Example 15 — Buffer notification callback**

```c++
break ;
case RtBuffer::ACQ_DONE:
case RtBuffer::BUFF_STOP:
    writeBuffer(numPoints) ;
    rtBuf_->clear() ;
    closeDataFile() ;
    break ;
case RtBuffer::BUFF_OVERRUN:
    writeBuffer(numPoints) ;
    rtBuf_->clear() ;
    closeDataFile() ;
    MStatus status;
    STS_SET(&status, MStatus(1) << "Data acquisition buffer overrun.");
    error(status);
    break ;
case RtBuffer::BUFF_RESET:
    break ;
}
return (TRUE);
```
8.4.4 Linear Buffers (RtLinearBuffer)

A linear buffer is a simple array that just collects the specified amount of data. Once the specified amount of data has been collected the buffer stops and any additional data is discarded.

The data for an RtLinearBuffer is held in the machine until the application calls getData() or a notification callback is delivered. A typical application for this type of buffer would be high rate one shot data acquisition.

Note

Because of a bug in V3.2, Linear Buffers do not deliver their BUFF_DONE events unless a notifyIndex has been set up. Therefore, you should call setNotifyIndex specifying the size of the buffer (or less)

<table>
<thead>
<tr>
<th>Related RtLinearBuffer methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtLinearBuffer * make ( RtAcq &amp; acq , long bufferSize, RtSignal &amp; sig, MStatus * status = 0 )</td>
</tr>
<tr>
<td>static RtLinearBuffer * make ( RtAcq &amp; acq , long bufferSize, GenList &amp; sigList, MStatus * status = 0 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Related RtBuffer methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>float * buffer ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>float * dataPtr( long scanNum, long chan = 0 )</td>
</tr>
<tr>
<td>float * operator[] (long scanNum)</td>
</tr>
<tr>
<td>double value( long scanNum, long chanNum = 0 )</td>
</tr>
<tr>
<td>long * iBuffer ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>long * iDataPtr( long scanNum, long chan = 0 )</td>
</tr>
<tr>
<td>long iValue( long scanNum, long chanNum = 0 )</td>
</tr>
<tr>
<td>long size ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>long scanSize( )</td>
</tr>
<tr>
<td>long count()</td>
</tr>
<tr>
<td>long first()</td>
</tr>
<tr>
<td>long index()</td>
</tr>
<tr>
<td>void setNotifyIndex(long index, Mstatus * status = 0)</td>
</tr>
<tr>
<td>long notifyIndex(Mstatus * status = 0)</td>
</tr>
<tr>
<td>void uponNotify ( Callback2Ptr &lt; boolean, int &gt; &amp; cb , MStatus * status = 0 )</td>
</tr>
<tr>
<td>void uponNotifyDiscon ( Callback2Ptr &lt; boolean, int &gt; &amp; cb , MStatus * status = 0 )</td>
</tr>
<tr>
<td>void lock()</td>
</tr>
<tr>
<td>void unLock()</td>
</tr>
<tr>
<td>void clear( Mstatus * status = 0 )</td>
</tr>
<tr>
<td>void getData( Mstatus * status = 0 )</td>
</tr>
</tbody>
</table>
8.4.5 Linear Contiguous Buffers (RtLinearCBuffer)

A linear contiguous buffer collects a continuous stream of data into an array. You can think of it as an RtLinearBuffer with a “type-ahead buffer” behind it. When the array becomes full, additional data is stored in an internal pipeline, until the application indicates it is done with the data in the array by calling clear(). At that time, the data currently in the buffer is removed, and the next data in the pipeline is transferred into the buffer’s array.

This type of buffering is useful for collecting a continuous stream of data of long or unknown duration. As long as the application can generally keep up with processing the data, the buffer will take care of short-term latencies.

The internal pipeline will hold up to 8192 samples for each signal collected by the buffer. For example, if the acquisition is collecting timed data at 100 Hz, this is 81 seconds of internal buffering. If the application gets too far behind, an overrun condition will occur, and data will be thrown away. If this happens the reason code for the callback routine will be RtBuffer::BUFF_OVERRUN, the count() method will indicate the number of valid points in the buffer and the point in the data where the overrun occurred. To clear this condition the RtBuffer and RtAcq reset() methods must be called and acquisition restarted by calling the RtAcq start() method.

Note

Linear Contiguous Buffers only push the data into the user’s buffer on these conditions: (1) getData is called, (2) the user-specified notifyIndex is satisfied, (3) the acquisition is stopped, or (4) the internal pipeline becomes over half full. When streaming lots of data through the buffer, it is better to keep the pipeline not as full. This can be done by specifying a notifyIndex equal to the size of the buffer. The application will continue to get the BUFF_FULL messages, but they will come as soon as the data is available, rather than waiting until the internal pipeline is half full.

A linear contiguous buffer can also be “locked” with the lock() method. When the buffer is locked, no additional data will be placed into the array. Any additional data acquired will pile up in the internal pipeline (just as if the buffer were full). When the buffer is unlocked with unLock(), data in the pipeline will begin transferring into the buffer’s array again.

---

Related RtLinearCBuffer methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtLinearCBuffer * make ( RtAcq &amp; acq, long bufferSize, RtSignal &amp;sig, MStatus * status = 0 )</td>
<td>Create a linear contiguous buffer</td>
</tr>
<tr>
<td>static RtLinearCBuffer * make ( RtAcq &amp; acq, long bufferSize, GenList &amp; sigList, MStatus * status = 0 )</td>
<td>Create a linear contiguous buffer with a list</td>
</tr>
<tr>
<td>virtual ~RtLinearCBuffer ( )</td>
<td>Destructor</td>
</tr>
</tbody>
</table>

Related RtBuffer methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float * buffer ( MStatus * status = 0 )</td>
<td>Buffer data</td>
</tr>
<tr>
<td>float * dataPtr( long scanNum, long chan = 0 )</td>
<td>Access data pointer</td>
</tr>
<tr>
<td>float * operator[]( long scanNum)</td>
<td>Access data pointer as an array</td>
</tr>
<tr>
<td>double value( long scanNum, long chanNum = 0 )</td>
<td>Access value as a double</td>
</tr>
<tr>
<td>long * iBuffer ( MStatus * status = 0 )</td>
<td>Access internal buffer</td>
</tr>
<tr>
<td>long * iDataPtr( long scanNum, long chan = 0 )</td>
<td>Access internal buffer pointer</td>
</tr>
<tr>
<td>long iValue( long scanNum, long chanNum = 0 )</td>
<td>Access internal value</td>
</tr>
<tr>
<td>long size ( MStatus * status = 0 )</td>
<td>Access size</td>
</tr>
<tr>
<td>long scanSize( )</td>
<td>Access scan size</td>
</tr>
</tbody>
</table>
8.4.6 Circular Buffers (RtCircularBuffer)

A circular buffer continually collects data into a fixed-length buffer. Once the buffer is full, collection continues, discarding the oldest data. Circular buffers are typically used to store the most recent information after some significant event. The acquisition is just allowed to run, until the event is detected. The acquisition is held, and buffer data is accessed.

The data for a circular buffer is held in the machine until the application calls `getData()` or a notification callback is delivered. The circular buffer is filled, starting at the beginning of the array. When the array is full, it starts filling at the beginning again. When data is requested by the application with the `getData()` method or a notification callback is delivered, the most recent data available in the machine is copied to the visible data array for the buffer. The `count()` method returns number of points in the array. The `value()` or `iValue` method and the `[]` operator perform the offset and unwrap calculation for accessing the data in the array.

Invoking `clear()` on a circular buffer will empty the visible data array. It will start it filling again with newer data held in the machine.

**Related RtCircularBuffer methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtCircularBuffer * make ( RtAcq &amp; acq , long bufferSize, RtSignal &amp; sig, MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>static RtCircularBuffer * make ( RtAcq &amp; acq , long bufferSize, GenList &amp; sigList, MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtCircularBuffer ( )</td>
<td></td>
</tr>
</tbody>
</table>

**Related RtBuffer methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float * buffer ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>float * dataPtr( long scanNum, long chan = 0 )</td>
<td></td>
</tr>
<tr>
<td>float * operator[] (long scanNum)</td>
<td></td>
</tr>
<tr>
<td>double value( long scanNum, long chanNum = 0 )</td>
<td></td>
</tr>
<tr>
<td>long * iBuffer ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>long * iDataPtr( long scanNum, long chan = 0 )</td>
<td></td>
</tr>
<tr>
<td>long iValue( long scanNum, long chanNum = 0 )</td>
<td></td>
</tr>
<tr>
<td>long size ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>long scanSize( )</td>
<td></td>
</tr>
<tr>
<td>long count()</td>
<td></td>
</tr>
<tr>
<td>long first()</td>
<td></td>
</tr>
<tr>
<td>long index()</td>
<td></td>
</tr>
</tbody>
</table>
8.5 Enable/disable data acquisition with a gate

The RtAcq object supports using an RtIntegerSig object to enable and disable the collection of data. This allows the application to cause data acquisition to be started or stopped directly from some event in the machine.

The application uses RtAcq::connectGate() to connect the RtIntegerSig object to the acquisition object as a gate. The data acquisition will be enabled whenever the signal is non-zero.

This can be conveniently used to enable or disable data acquisition based on a digital input, digital output. More complicated uses of this feature are described in the following subsections.

**Related RtAcq methods:**

| void       | connectGate(RtIntegerSig *gateSig, MStatus * =0) |

8.5.1 Enable/disable acquisition from an RtAction

It is possible to use an RtAction to start or stop data acquisition. The action can be connected to any event of interest (e.g., completion of an RtProfile, a limit detector, etc.). This will react to the event within a few ticks of the system rate (usually one or two). This is much faster than having the event handled by the application.

Example 16 shows how to set up an action to enable data acquisition when an action is fired. This example:

1. Creates an unnamed RtIntegerSig object to serve as the gate
2. Creates an RtAction containing a command to set the gate’s value to one
3. Creates an RtAcq, RtTimeTrigger, and RtLinearBuffer to collect the data
4. Connects the gate signal to the RtAcq by calling RtAcq::connectGate()
5. Starts the RtAcq.

When the RtAction is subsequently fired, it will set the value of the RtIntegerSig to one, which will cause the RtAcq to start saving data.
// create an unnamed RtIntegerSig to serve as the gate
Text dimensionName = "";
Text unitName = "";
RtIntegerSig * myGate = RtIntegerSig::make ( *rtStn,
    0,   // name
    false, // hidden
    SOURCE // CppSignal
    0,   // source
    0,   // srcIndex
    0,   // sink
    0,   // sinkIdx
    &dimensionName, // newDim
    &unitName, // newUnit
    1,   // resolution
    0,   // lower
    1);   // upper
myGate->value(0.0);    // the gate is initially closed

// create the action, its commands
RtAction *myAction = RtAction::make(*rtStn, 0);
RtCommand *enableCmd= myGate->getSetValueCommand( 1.0); // set the signal val
myAction->add(enableCmd);

// create the acq and any trigger
RtSequence *acqSeq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE) ;
RtAcq *rtAcq  = RtAcq::make(*rtStn, acqSeq) ;
RtTimeTrigger *rtTime = RtTimeTrigger::make(*rtAcq);
rtTime->setPeriod(0.01);

// buffers must now be set up...
RtFloatSig *timeSig = RtFloatSig::find(*rtStn, &Text("Time"));
GenList *sigList = new GenList() ;
sigList->addItem( *timeSig ) ;
RtLinearBuffer *buf = RtLinearBuffer::make(*rtAcq, 1000, *sigList);

// Connect the acquisition gate
rtAcq->connectGate( myGate ) ;

// now start the range detector and the acq
rtAcq->start();

When the action is subsequently triggered, the value of the RtIntegerSig will be set to non-zero, which will enable the data acquisition.

Example 16 — Enabling data acquisition from an RtAction

<table>
<thead>
<tr>
<th>Related RtAcq methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void connectGate(RtIntegerSig *gateSig, MStatus * =0)</td>
</tr>
</tbody>
</table>
8.5.2 Cycle-based data acquisition (RtIntRngDt)

An **RtIntegerRangeDt** monitors an input **RtIntegerSig**, comparing it to a set of upper and lower limit values. The output **RtIntegerSig** of the detector is set to one whenever the input signal is within the range of an upper and lower limit pair.

This class is used by MPT and other applications to implement “Cycle-based data acquisition”. To do this the input of the detector is connected to an **RtSegGen** count signal (Section 7.4.7 on page 65) and the output of the detector is connected to an **RtAcq** gate signal.

Figure 18 shows the samples that would be acquired with an **RtIntegerRangeDt** set to enable timed acquisition for segments 3 and 4.

The detection ranges for the **RtIntegerRangeDt** are set using the **setRange()** method. An **RtIntegerRangeDt** can operate relative to an offset value. The **offset(newOffset)** method is used to set the offset from the application. The offset can also be set to the current value of the signal with the **latchOffset()** method. The **offset()** method returns what the current offset value is.

Example 17 shows typical code for making and configuring an **RtIntegerRangeDt** as the gate for an **RtAcq** object.

The example performs the following functions:

1. Get the Segment count signal.
2. Find the high-rate **RtSequence**.
3. Find the “Axial Load” **RtFloatSig**
4. Create an **RtIntegerRangeDt**, connecting it to the sequence. The detector should be created before the **RtAcq** so that it executes prior to the **RtAcq**.
5. Set the detection ranges for the **RtIntegerRangeDt**.
6. Create an **RtAcq**, connecting it to the sequence
7. Connect the output **RtIntegerSig** of the **RtIntegerRangeDt** to the **RtAcq** gate.
8. Create and setup an **RtTimeTrigger**.
9. Create and setup the buffers (not shown)
10. Start the **RtIntegerRangeDt**. The range detector should normally be started before the **RtAcq** is started.
11. Start the **RtAcq**. Starting the **RtAcq** correctly sequences and starts the trigger and buffer objects.

![Figure 18 — Sampled points for Cycle-based timed data acquisition](image-url)
```cpp
int lowerLimits[2] = { 3, 9 };
int upperLimits[2] = { 4, 10 };

// Get the segment count signal
RtIntegerSig *segmentCount = rtSegGen->count1();
RtSequence *acqSeq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE);
RtFloatSig *floatSig = RtFloatSig::find(rtStn, &Text("Axial Load"));

// Make the range detector before making the RtAcq
RtIntegerRangeDt rangeDetect =
RtIntegerRangeDt::make(*rtStn, NIL, acqSeq, segmentCount);

// Set the ranges to enable acquisition during cycles 2 and 5
rangeDetect->setRanges(2, lowerLimits, upperLimits);
rtAcq = RtAcq::make(*rtStn, acqSeq);

// Connect the rangeDetector as the acquisition gate
rtAcq->connectGate( rangeDetect->out() );

// make and configure time trigger
RtTimeTrigger *rtTime = RtTimeTrigger::make(rtAcq);
rtTime->setPeriod(tPeriod);
tActPeriod = rtTime->samplePeriod();

// buffers must now be set up ...
....

// now start the range detector and the acq
rangeDetect->start();
rtAcq->start();
```

Example 17 — Setting up cycle-based data acquisition

Related RtIntegerRangeDt methods

```cpp
class RtIntegerRangeDt {
public:
  static RtIntegerRangeDt * make(RtStation &stn, const Text *name, RtSequence *seq, RtIntegerSig *sig, MStatus * = 0);
  static RtIntegerRangeDt * find(RtStation &stn, const Text *name, MStatus * = 0);
  void connectSignal(RtIntegerSig *aSignal, MStatus *status = 0);
  void setRanges(int numRanges, const int *lowerLimits, const int *upperLimits, MStatus * = 0);
  void latchOffset(MStatus* status=0);
  void offset(long newOffset, MStatus* status=0);
  long offset(MStatus* status=0);
  void start(MStatus * = 0);
  void stop(MStatus * = 0);
  RtIntegerSig * out(MStatus * = 0);
};
```

Related RtAcq methods:

```cpp
class RtAcq {
public:
  void connectGate(RtIntegerSig *gateSig, MStatus * =0);
  void connectGate(RtIntegerSig *gateSig, MStatus * =0);
};
```
## 8.6 High-speed data acquisition

High-speed data acquisition is an optional feature that acquires data faster than the system rate. This is only available on 493.21B and 493.25 conditioners, and requires a separate license to be installed.

This mechanism collects data from a separate (higher-speed) data stream coming from these conditioners. This higher data stream is available through the `RtFloatSig` class. See Section 4.1.6 on page 31 for a discussion of alternate data stream types on `RtFloatSig` objects.

For high-speed data acquisition, the application uses the same `RtAcq` and `RtBuffer` objects that are used. However, certain extra rules must be followed.

- The system rate must be 4096 or 6144 Hz.
- The `HIGH_RATE_SEQUENCE` must be specified to the `RtAcq`.
- The `RtAcq` must be told that it will be collecting high-speed data when it is constructed by specifying the stream type `HIGH_SPEED_DATA`.
- `RtHsTimeTrigger` is used, rather than the normal `RtTimeTrigger`.
- Only specific sample-rates are supported. This is based on what the system rate is as shown in the following table. If a different rate is specified, it will be rounded to one of these supported rates.

<table>
<thead>
<tr>
<th>System Rate (Hz)</th>
<th>Available acquisition Rates (Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>49152, 24576, 16384, 12288, 8192, 4096</td>
</tr>
<tr>
<td>6144</td>
<td>49152, 24576, 12288, 6144</td>
</tr>
</tbody>
</table>

- The `RtFloatSig` objects passed to the `RtBuffer` object must have the `HIGH_SPEED_DATA` stream available. The application can use the method `RtFloatSig::supportsDataStream()` to determine if it supports a particular data stream.

Example 18 shows typical code for collecting high-speed data acquisition

```cpp
// make and configure the Acq.
RtSequence *acqSeq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE);
rtAcq = RtAcq::make(*rtStn, HIGH_SPEED_DATA, acqSeq);

// make and configure the trigger
RtHsTimeTrigger *rtTime = RtHsTimeTrigger::make(rtAcq);
rtTime->setPeriod(tPeriod);
rtAcq->totalPoints(tPoints);
rtAcq->uponDone(CALLBACK0(this, MyClass, localDone, int));

// buffers must now be set up ...
RtFloatSig *dispSig = RtFloatSig::find(rtStn, &Text("Axial Displacement"));
RtFloatSig *loadSig = RtFloatSig::find(rtStn, &Text("Axial Load"));
GenList *sigList = new GenList();
sigList->addItem(*dispSig);
sigList->addItem(*loadSig);
RtLinearBuffer *buf = RtLinearBuffer::make(*rtAcq, myBufferSize, *sigList);

// now start the acq
rtAcq->start();
```

**Example 18 — High-speed data acquisition**
Any of the buffering mechanisms described in Section 8.4 starting on page 109 can be used with high-speed data acquisition. However, the signals must all be \texttt{RtFloatSig} objects that support the high-speed stream. You cannot use \texttt{RtIntegerSig} objects when doing high-speed data acquisition.

Because of the large amount of data that can be collected, this high-speed data acquisition is usually only used for collecting short bursts of data. It is possible to overload the bandwidth of the processors or communication links with too much data.

\begin{verbatim}
<table>
<thead>
<tr>
<th>Related RtHsTimeTrigger methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtHsTimeTrigger * make ( RtAcq &amp; acq , MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual ~RtHsTimeTrigger ( )</td>
</tr>
<tr>
<td>void setPeriod ( double samplePeriod , MStatus * = 0 )</td>
</tr>
<tr>
<td>void setRate ( double sampleRate , MStatus * = 0 )</td>
</tr>
<tr>
<td>double sampleRate ( MStatus * status = 0 )</td>
</tr>
<tr>
<td>double samplePeriod ( MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>Related RtAcq methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtAcq * make ( RtStation &amp; stn , RtSequence *seq , enum DataStreamTypes dataStreamType, MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual ~RtAcq ( )</td>
</tr>
<tr>
<td>enum DataStreamTypes typeOfDataStream ( MStatus * = 0 )</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
<table>
<thead>
<tr>
<th>Related RtFloatSig methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean supportsDataStream ( enum DataStreamTypes dataStream , MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
\end{verbatim}

\section*{8.7 Time-history data acquisition}

Acquisition of time-history data has special requirements that the normal data acquisition objects do not handle. This subsystem has been created to handle these requirements. Time-history data acquisition is only available on channels coming from 493.21B and 493.25 conditioners, or 498.64 ADDA boards.

\textbf{Note:}

Using this subsystem requires optional hardware, and an optional software license.

Properly sampled time history data must have zero skew (time delay between channels for the same scan of data) and zero jitter (i.e. a constant interval between samples) in the data being acquired. The data must be digitally filtered and down-sampled in such a way as to avoid aliasing. The data must be spooled to the application on a continuous basis (i.e. no lost points) for storage on disk and for other uses. Data acquisition may be tied to command generation when the system is also playing out a drive file or it may operate independently when the user just wants to sample data.

This capability is implemented with two classes: \texttt{RtTimeHistAcq} and \texttt{RtTimeHistBuffer}. The \texttt{RtTimeHistAcq} object lets the user specify the sampling rate, the channels to acquire, and the total points to acquire. This object also provides the run/stop, hold/resume and filtering controls.

The \texttt{RtTimeHistBuffer} objects are the containers in which the \texttt{RtTimeHistAcq} object will store the acquired data for transport to the application. Data for all channels will be stored in a single buffer, sorted
by channel. The application will queue two or more **RtTimeHistBuffers** to the **RtTimeHistAcq** object. When a **RtTimeHistBuffer** is full, the application will be notified. It is responsible to process the data, and then re-queue the **RtTimeHistBuffer** to the **RtTimeHistAcq**. The application must keep the **RtTimeHistAcq** supplied with buffers, to avoid loosing data.

Data stored in the buffers is floating point data, scaled in system units.

### 8.7.1 Configuring the **RtTimeHistAcq**

The time history acquisition is configured with code like the following:

```cpp
RtTimeHistAcq *acq = RtTimeHistAcq::make(myStation, 204.8f, &status)
acq->addSignal( RtFloatSig::find(rtStn, &Text("Axial Load")));
acq->addSignal( RtFloatSig::find(rtStn, &Text("Torsional Load")));
acq->setActive();
acq->totalPoints(10000);
```

**Example 19 — Creating and configuring an **RtTimeHistAcq**

The acquisition object is created, the signals to be collected are added in the desired order, and the object is “activated”. Once activated, the sampling rate and channel list cannot be modified.

Time history acquisition uses a different stream of data from the hardware, so it can only collect data from signals connected to a hardware resource that supports the alternate data stream. The signals must be **RtFloatSig** objects support the **TIME_HISTORY_DATA** stream type. The application can call **RtFloatSig::supportsDataStream()** to see if that signal supports the time-history data stream. See Section 4.1.6 on page 31 for a discussion of the different stream types available from **RtFloatSig** objects.

The acquisition rate (specified in the **make()** method) is limited to system’s “high clock rate” divided by 4, 8, 10, 16, 20, 32 or 40. (The system’s “high clock rate” is defined in the HWI file with the “HI RATE” keyword.) A down-sampling filter exists for only these rates. Some of these highest rates may not work if there is insufficient processor power.

The **totalPoints()** method allows the total number of points per channel to be specified. The default is to collect “zero points”.

The **setActive()** method initializes the acquisition, including starting up the down-sampling filters. These filters take some time to charge. If data acquisition is started to soon after the filters are enabled, it will hold off actually starting until the filters are charged. This charge-up time is dependent on the down-sampling ratio, but is typically hundreds of milliseconds.

These filters take up processing overhead in the data acquisition processor. In some situations, it may be desirable to turn them off until the application is ready to start the data acquisition. For example, in MPT, a procedure may contain several data acquisition processes that are all created when the procedure is locked. However, only the one currently active needs to have the filters enabled. The application may use **disableFilters()** to stop the filters until just before they are to be used. When turned back on with **enableFilters()** the filters will need the time to charge up again. The subsequent start will be held off until the filters are fully charged.

**Related RtTimeHistAcq methods:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtTimeHistAcq * make ( RtStation &amp; stn , double rate , MStatus * sp )</td>
<td></td>
</tr>
<tr>
<td>boolean addSignal ( RtFloatSig * s , MStatus * sp = 0 )</td>
<td></td>
</tr>
<tr>
<td>boolean addSignal ( GenList * l , MStatus * sp = 0 )</td>
<td></td>
</tr>
</tbody>
</table>
8.7.2 Creating and queuing buffers

Once the `RtTimeHistAcq` has been activated, `RtTimeHistBuffer` objects can be created and queued (creating them sooner will return an error). The buffer inherits the channel assignments from the `RtTimeHistAcq`.

Usually, at least two buffers are created, as shown in this example:

```c
// ***** Create BufferA *****
RtTimeHistBuffer *bufA = RtTimeHistBuffer::make(*acq, 1024, &status)
RtAction *doneA = RtAction::make(myStation, 0, BOOL_CB0(this, MyClass, bufADone));
bufA->attach(doneA);
bufA->queue();

// ***** Create BufferB *****
RtTimeHistBuffer *bufB = RtTimeHistBuffer::make(*acq, 1024, &status)
RtAction *doneB = RtAction::make(myStation, 0, BOOL_CB0(this, MyClass, bufBDone));
bufB->attach(doneB);
bufB->queue();
```

Example 20 — Adding RtTimeHistBuffers to an RtTimeHistAcq

The amount of data a buffer will collect is usually the buffer size specified in the `make` method. However, the application can reduce the number of points a buffer collects with the `queuedCount` method. This must not be greater than the original size specified in the `make()` method.

Related `RtTimeHistBuffer` methods:

```c
static RtTimeHistBuffer * make ( RtTimeHistAcq & rAcq , int size , MStatus * status )
int   channels ( MStatus * sp = 0 )
int   size ( MStatus * sp = 0 )
float * data ( int channel , MStatus * sp = 0 )
boolean queue ( MStatus * sp )
boolean queuedCount ( int c , MStatus * sp = 0 )
int   queuedCount ( MStatus * sp = 0 )
int   storedCount ( MStatus * sp = 0 )
boolean attach ( RtAction & act , MStatus * sp = 0 )
RtAction * detach ( MStatus * sp = 0 )
```
8.7.3 Processing the data

The callbacks invoked by the completion actions must process the data (usually copying it to a disk file), and re-queue the buffers. The data(index) method returns a pointer to a floating-point array for the specified channel. This pointer remains constant for the life of the buffer.

The storedCount() method returns the number of points that were actually placed into the buffer. This may be less than the queuedCount, if the data acquisition was ended prematurely (see the next section).

Time-history data acquisition can be ended prematurely by invoking stop() and then flush(). The stop() method will set the storedCount to the amount of data actually collected. The flush() method will dequeue each buffer, and invoke the completion action. This allows the completion methods to process the partially filled buffer.

### Related RtTimeHistBuffer methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>float * data (int channel, MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>boolean queue (MStatus * sp)</td>
<td></td>
</tr>
<tr>
<td>boolean queuedCount (int c, MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>int queuedCount (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>int storedCount (MStatus * sp = 0)</td>
<td></td>
</tr>
</tbody>
</table>

8.7.4 Starting and stopping the acquisition

The RtTimeHistAcq class implements start, stop, hold and resume methods, but they behave a little different than the analogous methods on other types of objects. Specifically:

- **start()** zeros the “acquired points” counter and starts or resumes data acquisition.
- **resume()** Starts or resumes data acquisition without zeroing the “acquired points” counter.
- **hold(), stop()** These methods behave the same. They stop collecting data. The filters continue to operate.

The “acquired points” counter is also zeroed by the reset, flush, and setMode methods.

### Related RtTimeHistAcq methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>void stop (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>void hold (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>void resume (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>boolean flush (MStatus * sp = 0)</td>
<td></td>
</tr>
<tr>
<td>void reset (MStatus * sp = 0)</td>
<td></td>
</tr>
</tbody>
</table>
8.7.5 Detecting errors

If the application does not keep the \texttt{RtTimeHistAcq} supplied with buffers, it will eventually not have a place to put the data. When this happens, a \texttt{STARVATION} error will be signaled, and the acquisition will be suspended until a buffer becomes available.

A second error that can occur is an internal overrun. This will cause an \texttt{OVERRUN} error to be signaled, and acquisition to shutdown, until a \texttt{reset()} is called.

The application can be informed when an error occurs by connecting a callback with the \texttt{uponError()} method. The application can also query for the error condition with the \texttt{errorStatus()} method. This will return 0, \texttt{STARVATION}, or \texttt{OVERRUN}. When an error occurs, the callback will be fired and the error status will be set to the appropriate value, overwriting any previous status value. This error status will not be cleared until \texttt{reset()} is called. However, it may be overwritten by a subsequent starvation error.

<table>
<thead>
<tr>
<th>Related RtTimeHistAcq methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void \texttt{uponError ( Callback0Ptr &lt; int &gt; &amp; cb , MStatus * sp = 0 )}</td>
</tr>
<tr>
<td>void \texttt{uponErrorDiscon ( MStatus * sp = 0 )}</td>
</tr>
<tr>
<td>long \texttt{errorStatus ( MStatus * sp = 0 )}</td>
</tr>
<tr>
<td>void \texttt{reset ( MStatus * sp = 0 )}</td>
</tr>
</tbody>
</table>

8.7.6 Synchronizing with command generation

Time history data acquisition can be synchronized with command generation by adding it to the same \texttt{RtSyncGroup}, and informing it of this fact with the \texttt{setMode} method. To run synchronized with command generation, Example 19 would be modified as shown in Example 21.

\begin{verbatim}
RtSyncGroup *grp = RtSyncGroup::make(&stn, 0, &status);
RtReadWriteSg *sg = myChannel->segGen(WRITE, &retStatus);
grp->addItem(&sg);
RtTimeHistAcq *acq = RtTimeHistAcq::make(myStation, 204.8, &status)
acq->addSignal( RtFloatSig::find(rtStn, &Text("Axial Load")));
acq->addSignal( RtFloatSig::find(rtStn, &Text("Torsional Load")));
acq->setActive();
acq->totalPoints(10000);
acq->setMode(0);
grp->addItem(acq);
\end{verbatim}

Example 21 — Creating and configuring an RtTimeHistAcq synchronized with command generation.

The \texttt{setMode} must be invoked after the \texttt{RtTimeHistAcq} is activated, but before it is started. The buffers are set up as before. The values for the mode are:

\begin{itemize}
  \item 0 — Slaved to the command sync group
  \item 1 — Independent of the sync group (the default).
\end{itemize}

The \texttt{RtTimeHistAcq} is a normal member of the \texttt{RtSyncGroup}. All members of the \texttt{RtSyncGroup} must be started before it will allow any member to actually start. The \texttt{RtTimeHistAcq} may hold of starting of the group to allow the filters to finish priming.
**Related RtTimeHistAcq methods:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtTimeHistAcq * make ( RtStation &amp; stn , double rate , MStatus * sp )</td>
<td>Create a new RtTimeHistAcq object</td>
</tr>
<tr>
<td>boolean addSignal ( RtFloatSig * s , MStatus * sp = 0 )</td>
<td>Add a float signal to the acquisition</td>
</tr>
<tr>
<td>boolean addSignal ( GenList * l , MStatus * sp = 0 )</td>
<td>Add a generic list of signals to the acquisition</td>
</tr>
<tr>
<td>GenList * signalList ( )</td>
<td>Get the list of signals</td>
</tr>
<tr>
<td>boolean setActive ( MStatus * sp = 0 )</td>
<td>Set the active state of the acquisition</td>
</tr>
<tr>
<td>void enableFilters ( MStatus * sp = 0 )</td>
<td>Enable filters on the acquisition</td>
</tr>
<tr>
<td>void disableFilters ( MStatus * sp = 0 )</td>
<td>Disable filters on the acquisition</td>
</tr>
<tr>
<td>boolean totalPoints ( long tp , MStatus * sp = 0 )</td>
<td>Get the total number of points</td>
</tr>
<tr>
<td>long totalPoints ( MStatus * sp = 0 )</td>
<td>Get the total number of points</td>
</tr>
<tr>
<td>boolean setMode ( long newMode , MStatus * sp = 0 )</td>
<td>Set the acquisition mode</td>
</tr>
<tr>
<td>long mode ( MStatus * sp = 0 )</td>
<td>Get the current acquisition mode</td>
</tr>
</tbody>
</table>
Chapter 9
Detectors

Detectors are objects that watch for some value transition on one or more signals. In response to this event, the detector may trigger an **RtAction**, and may log a message into the station message log. The detector classes currently available are:

- **RtFloatLdt** — Limit detector for floating point signals. It supports independent operation of an upper and lower limit, with independent actions for each. It can be used to watch a single signal, or a group of signals.

- **RtIntegerLdt** — Limit detector for integer signals. It supports independent operation of an upper and lower limit, with independent actions for each.

- **RtBooleanLdt** — Limit detector for integer signals that represent Boolean values. These support just one action, based on the value being zero or non-zero. The triggering modes are designed for monitoring digital input signals.

- **RtPeakValleyLdt** — Senses a change in the peaks and valleys of a floating-point signal.

- **RtFloatABLdt** — A-B compare limit detector. This detector compares two signals, A and B, and trips when the signals differ by some specified amount. The detector can be configured to compare one or more sets of signals.

### 9.1 Single Floating Point Limit Detector (RtFloatLdt)

The **RtFloatLdt** class implements a limit detector that watches for changes in **RtFloatSig** signals. An **RtFloatLdt** object actually contains two limit detectors referred to as “MIN” and “MAX” (or “lower” and “upper”). These two limit detectors are functionally independent, and can be configured, started and stopped independently.

Example 22 sets up an **RFloatLdt** to watch for the “Axial Load” signal going outside a specified range.

```cpp
RtSequence *seq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE);
RtFloatSig *floatSig = RtFloatSig::find(rtStn, &Text("Axial Load"));
RtFloatLdt *ldt = RtFloatLdt::make(*rtStn, 0, seq, floatSig);
ldt->lower(-0.5); // kN
ldt->upper(10.0); // kN

// connect up the actions.
RtAction *trippedAction = RtAction::make(rtStn, 0, BOOL_CB0(this, MyClass, limitTrippedCB));
ldt->connect(MIN, trippedAction);
ldt->connect(MAX, trippedAction);

// now start the detector.
ldt->start();
```

**Example 22 — Floating-point limit detector**
This example performs the following functions:

1. Find the high rate sequence. This determines how often the limit detector process runs. Limit detectors can run at lower rates if you are willing to not see transient noise spikes (See Section 8.2 on page 98 for more information about sequences).

2. Create an unnamed RtFloatLdt, connecting it to the sequence, and the signal.

3. Configure the RtFloatLdt to trip if the signal goes outside the bounds.

4. Hook an action up to the two limits.

5. Start the RtFloatLdt.

Applications will typically create unnamed limit detectors, and destroy them when they are done.

It is possible to create the limit detector without specifying a signal to watch, and connect a signal later by calling connectSignal(). If there is no signal connected to the limit detector, many of the other functions will return an error.

### Related RtFloatLdt methods:

```
static RtFloatLdt * make ( RtStation & stn , const Text * name, RtSequence *seq, RtFloatSig *signal, 
                         MStatus * status = 0 )
static RtFloatLdt * make ( RtStation & stn , const Text * name , MStatus * status = 0 )
static RtFloatLdt * find ( RtStation & stn , const Text * name , MStatus * status = 0 )
virtual ~RtFloatLdt ( )
void  connectSignal ( RtFloatSig * aSignal , MStatus * status = 0 )
void  start ( MStatus * status = 0 )
void  lower ( double aValue , MStatus * = 0 )
void  upper ( double aValue , MStatus * = 0 )
void  connect ( RangeType minOrMax , RtAction * anAction , MStatus * status = 0 )
void  disconnect ( RangeType minOrMax , RtAction * anAction , MStatus * status =0)
```

### Related RtTask methods (base class of RtFloatLdt):

```
void  sequence ( RtSequence * seq , MStatus * = 0 )
void  connectToSequence ( RtSequence & seq , MStatus * status = 0 )
virtual double  rate ( MStatus * status = 0 )
virtual void  start ( MStatus * status = 0 )
virtual void  stop ( MStatus * status = 0 )
virtual void  reset ( MStatus * status = 0 )
```

### 9.1.1 Detector Modes

While the names imply that these two limit detectors define a double-sided limit detector, they can be configured other ways. For example, Station Manager uses one RtFloatLdt watching the “Absolute Error” signal to implement the “Inner” and “Outer” error detectors. In this mode both limits are triggering on “greater than”, with each having its own limit value and action.

This flexibility is implemented with the setMode() method. The RtFloatLdt supports the following DetectorModes:
MIN_MAX — (default) This is the traditional mode where the two limits provide an lower and upper bound. If the signal travels outside these limits for the necessary amount of time, the appropriate action will be taken.

In this mode, if the detector is given a name (i.e. made a station-wide object), then this detector mode will log a message to the message log when it triggers. This logging behavior is only intended to be used by the Station Manager.

TRIGGER — This is the same as MIN_MAX, except that no message is ever logged. This is not really intended to be used in floating-point limit detectors, and may be retired in a future version.

LOWER_WINDOW — In this mode, both limits will trip on “less than”. The signal would normally start higher than both limits. As the signal value becomes smaller, it would first trip one, than the other limit. Each limit would fire its own action.

UPPER_WINDOW — This operates analogous to LOWER_WINDOW, except both limits trip on “greater than”.

Most applications will simply create an unnamed object, and use MIN_MAX. If an application only needs a single limit value, it can simply not configure or start the other side.

Related RtFloatLdt methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DetectorMode mode</td>
<td>MStatus * status = 0</td>
</tr>
<tr>
<td>void setMode (DetectorMode aMode, MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>

### 9.1.2 Attributes of each limit

Each of the two limits contains the following attributes. These can be set independently.

**Limit value** — This value can be set with a variety of methods. There are explicit lower() and upper() methods. A Number parameter is also available through getUpperParm() and getLowerParm(). Also, the method band() will set both limits to plus and minus a specified value.

**Limit action** — This is the RtAction to fire when the limit fires. Applications will usually use the connect() method to set the action.

The methods getUpAction() and getLoAction() return a OneOfList which can be used to connect a named action to the limit. By default, the list of actions available to the OneOfList is all the named actions on the station. However, setActionList(whichList) and setDefaultActionList() can be called to change to a list containing a subset of all the actions. (whichList is a single value of the enumeration EventTypeMask.) These alternate lists of actions are maintained within RtStation using methods getActions(whichList) and getActionManyList(whichList). Each application has its own definitions of these lists (changes in these lists by Station Manager do not propagate into each application). Changing them within an application is possible, but not recommended.
Note

The methods `getUpAction()` and `getLoAction()` can be used to query the action assignments of the named limit detectors created by Station Manager; however, these `OneOfList` objects are not automatically updated (connecting a notification callback will not work). You need to explicitly request the value by calling something like `getSelectedItem()`.

**ActionMode** — This determines whether the detector will automatically reset when the signal value crosses back over the limit value. The options are defined by the enumeration `TriggerModes` as:

- **CONTINUOUS** The detector will automatically reset once the signal value crosses back over the limit value.
- **ONCE** The application must manually call `reset()` to reset the detector. The method `reset()` resets both sides of the limit detector.

**Delay (persistence)** — This is the amount of time (in seconds) the signal must remain above (or below) the limit value before the limit will fire. The default is zero. This is useful for suppressing multiple trigger-reset cycles, or for avoiding false trips in the presence of noise.

**Start/stop** — Each limit can be independently started and stopped, or they can be started and stopped together.

Note:

Extreme care must be taken when using the **CONTINUOUS** option. If the signal value is hovering around the limit value, noise on the signal may make it trip and reset many times very quickly. This can generate a flood of messages within the system, and bog down the links and the computer.

The `hold/resume` manipulates a state similar to `start/stop`, except it operates on both limits in the `RtFloatLdt` object. The default is resume-state. If `hold()` is called, it must be followed by a `resume()` for the detector to be re-enabled.

### Related RtFloatLdt methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Number * getUpperParm (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>Number * getLowerParm (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>OneOfList * getUpActionParm (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>OneOfList * getLoActionParm (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual boolean setActionList (int whichList, MStatus * stat = NIL)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual boolean setDefaultActionList (MStatus * stat)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual void start (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual void stop (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual void start (RangeType minOrMax, MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>virtual void stop (RangeType minOrMax, MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>double lower (MStatus * status = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>void lower (double aValue, MStatus * = 0)</code></td>
<td></td>
</tr>
<tr>
<td><code>double upper (MStatus * status = 0)</code></td>
<td></td>
</tr>
</tbody>
</table>
### 9.2 Group Floating Point Detector (RtFloatLdt)

The **RtFloatLdt** class can also be used to watch more than one signal at a time. When used in this way, you can think of it as an array of individual detectors that share a single **Detector Mode, Lower Action, Upper Action, and Action Mode**.

In this case, the application specifies a list of **RtFloatLdt** objects. Each signal will have a **Lower Limit Value, Upper Limit Value, Delay, and Start/Stop state**.

The list of signals can be specified on the call to **make()**, or at a later time by calling the **connectList()** method.

The **RtFloatLdt** class contains versions of many of the methods listed in Section 9.1.2 that include a `sigIndex`. These allow the attributes for individual signals to be manipulated. The methods that do not specify a signal index generally apply to the first signal. The exception is the **start** and **stop** methods. The forms of **start** and **stop** that do not specify an index apply to all the signals.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Applies to all signals</th>
<th>Applies to each signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detector Mode</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Lower Action Mode</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Lower Action</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Lower Limit Value</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Lower Delay</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Lower Start/Stop State</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Upper Action Mode</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Upper Action</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Upper Limit Value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Upper Delay</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Upper Start/Stop State</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Table 9 — Scope of attributes on a Group Floating Point Detector**

---

```cpp
void upper ( double aValue , MStatus * = 0 )
double band ( MStatus * status = 0 )
void band ( double aValue , MStatus * = 0 )
double delay ( RangeType minOrMax , MStatus * status = 0 )
void delay ( RangeType minOrMax , double aValue , MStatus * = 0 )
TriggerModes actionMode ( RangeType minOrMax , MStatus * status = 0 )
void actionMode ( RangeType minOrMax , TriggerModes aMode , MStatus * status =0 )
DetectorMode mode ( MStatus * = 0 )
void setMode ( DetectorMode aMode , MStatus * = 0 )
void connect ( RangeType minOrMax , RtAction * anAction , MStatus * status = 0 )
void disconnect ( RangeType minOrMax , RtAction * anAction , MStatus * status =0 )
RtIntegerSig * out ( MStatus * status = 0 )
virtual void resume ( MStatus * status = 0 )
virtual void hold ( MStatus * status = 0 )
```
The **Lower Action Mode** and **Upper Action Mode** parameters apply to all the limits. However, when they are set to CONTINUOUS, each limit resets independently when it goes back into range.

The **Report Option** attribute determines how the **RtFloatLdt** object handles multiple limits tripping at the same time. If the **Report Option** is set to **TRUE**, then only the first lower limit and the first upper limit that trip in a single clock tick will fire the action and/or log a message. The other limits will become tripped; they just won’t fire the action. This can help reduce the message traffic when multiple limits trip at once.

The following summary of methods is not a complete list—it is the list of methods in addition to those listed in Section 9.1.2.

### Related RtFloatLdt methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtFloatLdt * make ( RtStation &amp; stn , const Text * name , RtSequence * seq , GenList * sigList , MStatus * status = 0)</td>
<td>Creates a new RtFloatLdt object with the specified parameters.</td>
</tr>
<tr>
<td>void connectSignal ( RtFloatSig * aSignal , MStatus * status = 0 )</td>
<td>Connects the signal to the RtFloatLdt object.</td>
</tr>
<tr>
<td>GenList *getSigList(MStatus *status = 0)</td>
<td>Returns the list of signals associated with the RtFloatLdt object.</td>
</tr>
<tr>
<td>Number *getUpperParm(int sigIndex, MStatus * status = 0 )</td>
<td>Returns the upper limit for the specified signal index.</td>
</tr>
<tr>
<td>Number *getLowerParm(int sigIndex, MStatus * status = 0 )</td>
<td>Returns the lower limit for the specified signal index.</td>
</tr>
<tr>
<td>Number *getPersistenceParm(int sigIndex, MStatus *status = 0)</td>
<td>Returns the persistence parameter for the specified signal index.</td>
</tr>
<tr>
<td>virtual void start (int sigIndex, MStatus * status = 0 )</td>
<td>Starts monitoring the specified signal.</td>
</tr>
<tr>
<td>virtual void stop (int sigIndex, MStatus * status = 0 )</td>
<td>Stops monitoring the specified signal.</td>
</tr>
<tr>
<td>virtual void start (int sigIndex, RangeType minOrMax, MStatus * status = 0)</td>
<td>Starts monitoring the specified signal in trigger mode.</td>
</tr>
<tr>
<td>virtual void stop (int sigIndex, RangeType minOrMax, MStatus * status = 0)</td>
<td>Stops monitoring the specified signal in trigger mode.</td>
</tr>
<tr>
<td>double lower (int sigIndex, MStatus * status = 0 )</td>
<td>Returns the lower limit value for the specified signal index.</td>
</tr>
<tr>
<td>void lower (int sigIndex, double aValue , MStatus * = 0 )</td>
<td>Sets the lower limit value for the specified signal index.</td>
</tr>
<tr>
<td>double upper (int sigIndex, MStatus * status = 0 )</td>
<td>Returns the upper limit value for the specified signal index.</td>
</tr>
<tr>
<td>void upper (int sigIndex, double aValue , MStatus * = 0 )</td>
<td>Sets the upper limit value for the specified signal index.</td>
</tr>
<tr>
<td>double band (int sigIndex, MStatus * status = 0 )</td>
<td>Returns the band value for the specified signal index.</td>
</tr>
<tr>
<td>void band (int sigIndex, double aValue , MStatus * = 0 )</td>
<td>Sets the band value for the specified signal index.</td>
</tr>
<tr>
<td>double delay (int sigIndex, RangeType minOrMax, MStatus * status = 0)</td>
<td>Returns the delay value for the specified signal index.</td>
</tr>
<tr>
<td>void delay (int sigIndex, RangeType minOrMax, double aValue , MStatus * = 0)</td>
<td>Sets the delay value for the specified signal index.</td>
</tr>
<tr>
<td>RtIntegerSig *out (int sigIndex, MStatus * status = 0 )</td>
<td>Returns the output signal for the specified signal index.</td>
</tr>
<tr>
<td>void reportOption(boolean option, MStatus *status);</td>
<td>Sets the report option for the RtFloatLdt object.</td>
</tr>
<tr>
<td>boolean reportOption(MStatus *status);</td>
<td>Checks the report option for the RtFloatLdt object.</td>
</tr>
</tbody>
</table>

### 9.3 Integer Limit Detector (RtIntegerLdt)

Integer limit detectors are directly analogous to floating point limit detectors, except they operate on integer signals (rather than floating point signals). The **RtIntegerLdt** class implements a single integer limit detector—it does not support the group detector mechanism that the floating point detector class supports.

The **RtIntegerLdt** class supports the same states and methods as the **RtFloatLdt** as described in Section 9.1 on page 129 except that the data type is integer, and the signal class is **RtIntegerSig**.

In addition, the TRIGGER mode operates differently. In trigger mode, the upper and lower limits, and the delay attribute are not used. Rather, the detector will fire the upper action when the value of the signal goes
from zero to non-zero, and fire the lower action when the value goes from non-zero to zero. If the
*actionMode* is CONTINUOUS, it will trigger more than once.

**Note**

The TRIGGER mode is the old mechanism for detecting events on
digital input signals. In V3.0, the *RtBooleanLdt* class was introduced
that is a superset of this capability. TRIGGER mode may be removed
from the integer limit detector in a subsequent release.

**Related RtIntegerLdt methods:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtIntegerLdt * make (RtStation &amp;stn, const Text *name, DetectorMode aMode)</td>
<td>RtSequence *seq, RtIntegerSig *signal, MStatus *status = 0)</td>
</tr>
<tr>
<td>static RtIntegerLdt * make (RtStation &amp;stn, const Text *name, DetectorMode aMode, MStatus *status = 0)</td>
<td></td>
</tr>
<tr>
<td>static RtIntegerLdt * find (RtStation &amp; stn, const Text * name, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual ~RtIntegerLdt ()</td>
<td></td>
</tr>
<tr>
<td>void connectSignal (RtIntegerSig * aSignal, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>Number *getUpperParm (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>Number *getLowerParm (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>OneOfList *getUpActionParm (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>OneOfList *getLoActionParm (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual boolean setActionList (int forError, MStatus * stat = NIL)</td>
<td></td>
</tr>
<tr>
<td>virtual boolean setDefaultActionList (MStatus * stat)</td>
<td></td>
</tr>
<tr>
<td>virtual void start (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void stop (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void start (RangeType minOrMax, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void stop (RangeType minOrMax, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>long lower (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void lower (long aValue, MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>long upper (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void upper (long aValue, MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>long band (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void band (long aValue, MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>double delay (RangeType minOrMax, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void delay (RangeType minOrMax, double aValue, MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>TriggerModes actionMode(RangeType minOrMax, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void actionMode(RangeType minOrMax, TriggerModes aMode, MStatus * status =0)</td>
<td></td>
</tr>
<tr>
<td>DetectorMode mode (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void setMode (DetectorMode aMode, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void connect (RangeType minOrMax, RtAction * anAction, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void disconnect (RangeType minOrMax, RtAction * anAction, MStatus * status =0)</td>
<td></td>
</tr>
<tr>
<td>RtIntegerSig *out (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void resume(MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void hold(MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>

**Related RtTask methods (base class of RtIntegerLdt):**

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sequence (RtSequence * seq, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>void connectToSequence (RtSequence &amp; seq, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual double rate (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void start (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void stop (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void reset (MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>
9.4 Boolean Limit Detector (RtBooleanLdt)

The Boolean limit detector is designed to watch integer signals that only have Boolean values, like digital inputs. Many of the attributes are analogous to the RtFloatLdt, except that RtBooleanLdt supports just a single action, and the set of modes are more appropriate for Boolean signals.

The detector modes are defined by the enumeration BoolTriggerMode. These modes fall into two categories:

**Edge-sensitive** — These detector modes trigger based on seeing a change in the signal. For example, if the signal is already non-zero when BOOL_LOW_HIGH detector is started, it will not trigger until the signal goes to zero, and back to non-zero. The edge-sensitive modes are:

- BOOL_HIGH_LOW
- BOOL_LOW_HIGH
- BOOL_EITHER

**Level-sensitive modes** — These detector modes trigger based on seeing a particular level in the signal. For example, if the signal is already non-zero when BOOL_CHAN_HIGH detector is started, it will trigger immediately. The level-sensitive modes are:

- BOOL_CHAN_LOW
- BOOL_CHAN_HIGH

How the detector behaves once it has tripped is controlled by the actionMode attribute. The action mode is either ONCE or CONTINUOUS. This behavior is a function of both action mode and the detector mode as follows:

**ONCE and Edge-sensitive** — Action fires whenever an edge is detected. Output Signal stays high after the first edge is detected.

**ONCE and Level-sensitive** — Action fires once when signal reaches level. Output signal stays high thereafter.

**CONTINUOUS and Edge-sensitive** — Action fires whenever an edge is detected. Output signal goes high for one tick following the edge detection. It then goes back to low.

**CONTINUOUS and Level-sensitive** — Action fires whenever signal goes from outside the level to desired level. Output signal is high when signal is in level and low when signal is outside level.

ONCE mode is the default and it preserves the behavior before 3.4A.

**Note**

The lower, upper and delay attributes (and their associated methods) are not used, and may be removed in a subsequent version.

<table>
<thead>
<tr>
<th>Related RtBooleanLdt methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtBooleanLdt * make ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</td>
</tr>
<tr>
<td>static RtBooleanLdt * make ( RtStation &amp; stn , const Text * name , RtSequence * seq , RtIntegerSig * sig , MStatus * status = 0 )</td>
</tr>
<tr>
<td>static RtBooleanLdt * find ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</td>
</tr>
<tr>
<td>virtual ~RtBooleanLdt ( )</td>
</tr>
<tr>
<td>void connectSignal ( RtIntegerSig * aSignal , MStatus * status = 0 )</td>
</tr>
</tbody>
</table>
OneOfList * getActionParm ( MStatus * status = 0 )
OneOfList * getModeParm ( MStatus * status = 0 )
virtual boolean setActionList ( int forError , MStatus * stat = NIL )
virtual boolean setDefaultActionList ( MStatus * stat )

virtual void start ( MStatus * status = 0 )
virtual void stop ( MStatus * status = 0 )
BoolTriggerMode mode ( MStatus * status = 0 )
void setMode ( BoolTriggerMode aMode , MStatus * status = 0 )
TriggerModes actionMode ( MStatus * status = 0 )
void actionMode ( TriggerModes aMode , MStatus * status = 0 )
void connect ( RtAction * anAction , MStatus * status = 0 )
void disconnect ( RtAction * anAction , MStatus * status = 0 )
RtIntegerSig * out ( MStatus * status = 0 )
virtual void resume ( MStatus * status = 0 )
virtual void hold ( MStatus * status = 0 )

Related RtTask methods (base class of RtIntegerLdt):
void sequence ( RtSequence * seq , MStatus * status = 0 )
void connectToSequence ( RtSequence & seq , MStatus * status = 0 )
virtual double rate ( MStatus * status = 0 )
virtual void start ( MStatus * status = 0 )
virtual void stop ( MStatus * status = 0 )
virtual void reset ( MStatus * status = 0 )

9.5 Peak/Valley Level Detector (RtPeakValleyLdt)

The peak/valley level detector lets an application monitor an input signal for changes in peaks and valleys.

Normally, the process begins by detecting the amplitudes of the first peak and valley. These values become the reference levels for the tolerance range. When a peak or valley exceeds the tolerance range, the process triggers an action.

The tolerance range check can be either single-ended (under-peak detection), or double-ended (change detection). Figure 19 and Figure 20 show the different effects of this mode. Both the peak and valley monitoring must be in the same mode.
This detector can be configured in a variety of ways:

- A specified number of peaks and valleys can be ignored before starting the tolerance checking. This is useful to allow the signal to ‘settle in’ to a steady state before beginning to monitor it.

- The reference values can be specified, rather than letting the process calculate them.

- If an \texttt{RtChannel} object is connected, this detector will suspend checking during profile begin/end tapers, and during run, hold and stop transition states.

- Also, if an \texttt{RtChannel} object is connected, this detector can trigger an event if it does not detect a peak or valley within a certain number of segments on the channel’s segment generator.

Example 23 shows a typical sequence for creating an \texttt{RtPeakValleyLdt}.
This example performs the following functions:

1. Find the high rate sequence. This determines how often the detector process samples the input signal. It is important for Peak/Valley limit detectors to run in the high-rate sequence, so they can accurately determine the peak values (See Section 8.2 on page 98 for more information about sequences).

2. Find the `RtFloatSig` to be monitored.

3. Create an unnamed `RtPeakValleyLdt`, connecting it to the sequence, and the signal.

4. Configure the `RtPeakValleyLdt` with the mode, tolerance, delay count, and to trip if the signal goes outside the bounds.

5. Create and connect an action to be fired.

6. Start the `RtPeakValleyLdt`.

Applications will typically create unnamed limit detectors, and destroy them when they are done.

The peak/valley level detector will trip once on the first peak or valley out of tolerance. Invoking the reset() method will reset both the peak and valley detectors. They will then recalculate the reference (if configured to do so), but will not repeat the delay. To repeat the delay, setDelay() or setupDetector() must be called.

### Example 23 — Creating a peak/valley change detector.

```c
#define TOLERANCE 1.0  // kN
#define DELAY_COUNT 5  // peaks + valleys
#define NOISE_SENSITIVITY 0.05 // kN

RtSequence *seq = RtSequence::find(*rtStn, HIGH_RATE_SEQUENCE);
RtFloatSig *floatSig = RtFloatSig::find(rtStn, &Text("Axial Load"));

// make and initialize new PVLdt object
pvLdt = RtPeakValleyLdt::make(*rtStn, 0, seq, floatSig);

pvLdt->setupDetector(PEAK_VALLEY_CHANGE, FALSE, TOLERANCE, DELAY_COUNT);
pvLdt->setNoiseLevelSensitivityBand(NOISE_SENSITIVITY);

RtAction *pvLdtAction = RtAction::make(*rtStn, 0,
                                      BOOL_CB0(this, trippedCB, limitTripped));
pvLdt->connect(pvLdtAction);
pvLdt->start();
```

---

**Related RtPeakValleyLdt methods:**

- static RtPeakValleyLdt * make ( RtStation & stn , const Text * name ,
  RtChannel * chan , RtSequence * seq , RtFloatSig * sig , MStatus * status = 0 )
- static RtPeakValleyLdt * make(RtStation &stn,const Text *name,
  RtSequence *seq, RtFloatSig *sig, MStatus * = 0)
- static RtPeakValleyLdt * make ( RtStation & stn , const Text * name , MStatus * status = 0 )
- static RtPeakValleyLdt * find ( RtStation & stn , const Text * name , MStatus * status = 0 )
- virtual ~RtPeakValleyLdt ()
- void connectSignal ( RtFloatSig * aSignal , MStatus * status = 0 )
- void start ( MStatus * status = 0 )
- void setupDetector ( PVDetectorMode aMode , boolean useRef , double tolerance , long
cycDelay , MStatus * status = 0 )
### 9.5.1 Attributes of a peak/valley limit detector

The `RtPeakValleyLdt` contains the following attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mode</strong></td>
<td>PEAK_VALLEY_CHANGE, or UNDER_PEAK. This determines if the tolerance band is single-sided, or double-sided (see Figure 19 and Figure 20).</td>
</tr>
<tr>
<td><strong>tolerance</strong></td>
<td>The amount a peak or valley can be different than the current reference without tripping the detector.</td>
</tr>
<tr>
<td><strong>noise level sensitivity</strong></td>
<td>The amount the signal must reverse for a local maximum to be considered a peak, or a local minimum to be considered a valley.</td>
</tr>
<tr>
<td><strong>action</strong></td>
<td>The <code>RtAction</code> to fire when a peak or valley is out of the current tolerance band (shaded gray in Figure 19 and Figure 20).</td>
</tr>
<tr>
<td><strong>delay</strong></td>
<td>The number of cycles (peak/valley pairs) to skip at the beginning before calculating a reference. This is set by the <code>setupDetector()</code> and <code>setDelay()</code> method.</td>
</tr>
</tbody>
</table>

**Note:**

Calling `setupDetector()` or `setDelay()` will restart the delay count. If the delay has already expired, it will cause the delay to occur again, but will not cause the reference to be recalculated afterwards. We may change this behavior in a subsequent release.

| **channel**             | Connecting this detector to an `RtChannel` allows it to monitor the segment generator on that channel for transition tapers. The application will usually not want under-peak detection to occur during these tapers. |
| **segment timeout**     | If this number of segments is seen on the `RtChannel` without detecting a peak or valley, the timeout action will be triggered.                  |
| **timeout action**      | `RtAction` to be triggered when a peak or valley is not detected in a certain number of segments.                                           |
current reference — Normally, the process calculates the reference values after the delay cycles, from the last peak and valley cycles it sees. The application can override this behavior by specifying **TRUE** as the second parameter to **setupDetector()**, setting reference values via **setReference()**. In this case, the delay will still be done before checking the peak/valleys against the tolerance band, but a new reference will not be calculated.

start state — The detector must be explicitly started. It may be stopped and restarted at will.

current max/min — The last peak and valley which have been detected are always available through the **maximum()** and **minimum()** methods.

The **hold/resume** manipulates a state similar to **start/stop**. The default is resume-state. If **hold()** is called, it must be followed by a **resume()** for the detector to be re-enabled.

**Related RtPeakValleyLdt methods:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void connectSignal ( RtFloatSig * aSignal , MStatus * status = 0 )</td>
<td>Connect a signal</td>
</tr>
<tr>
<td>virtual void start ( MStatus * status = 0 )</td>
<td>Start the detector</td>
</tr>
<tr>
<td>virtual void stop ( MStatus * status = 0 )</td>
<td>Stop the detector</td>
</tr>
<tr>
<td>void setupDetector ( PVDetectorMode aMode , boolean useRef , double tolerance , long cycDelay , MStatus * status = 0 )</td>
<td>Set up the detector mode with reference and delay</td>
</tr>
<tr>
<td>void setupDetector ( PVDetectorMode aMode , boolean useRef , double tolerance , long cycDelay , long cycTimeout , MStatus * status = 0 )</td>
<td>Set up the detector mode with reference, delay, and timeout</td>
</tr>
<tr>
<td>void setupDetector ( PVDetectorMode aMode , boolean useRef , double tolerance , MStatus * status )</td>
<td>Set up the detector mode with reference and delay based on timeout</td>
</tr>
<tr>
<td>void connect ( RtAction * anAction , MStatus * status = 0 )</td>
<td>Connect an action</td>
</tr>
<tr>
<td>void disconnect ( RtAction * anAction , MStatus * status = 0 )</td>
<td>Disconnect an action</td>
</tr>
<tr>
<td>void connectTimeout ( RtAction * anAction , MStatus * status = 0 )</td>
<td>Connect an action with timeout</td>
</tr>
<tr>
<td>void disconnectTimeout ( RtAction * anAction , MStatus * status = 0 )</td>
<td>Disconnect an action with timeout</td>
</tr>
<tr>
<td>double noiseLevelSensitivityBand ( MStatus * status = 0 )</td>
<td>Set the noise level sensitivity band</td>
</tr>
<tr>
<td>void setNoiseLevelSensitivityBand ( double band , MStatus * status = 0 )</td>
<td>Set the noise level sensitivity band with a specific band</td>
</tr>
<tr>
<td>double maximum ( MStatus * status = 0 )</td>
<td>Get the maximum value</td>
</tr>
<tr>
<td>double minimum ( MStatus * status = 0 )</td>
<td>Get the minimum value</td>
</tr>
<tr>
<td>void setReference ( double peakRef , double valleyRef , MStatus * status = 0 )</td>
<td>Set the reference values with peaks and valleys</td>
</tr>
<tr>
<td>double reference ( PVDetectorRange rangeType , MStatus * status = 0 )</td>
<td>Get the reference value with a specific range type</td>
</tr>
<tr>
<td>void setDelay ( long aDelay , MStatus * status = 0 )</td>
<td>Set the delay with a specific delay type</td>
</tr>
<tr>
<td>long delay ( PVDelayType aType , MStatus * status = 0 )</td>
<td>Get the delay with a specific delay type</td>
</tr>
<tr>
<td>void setTimeout ( long cycTimeout , MStatus * status = 0 )</td>
<td>Set the timeout with a specific delay type</td>
</tr>
<tr>
<td>long timeout ( MStatus * status = 0 )</td>
<td>Get the timeout with a specific delay type</td>
</tr>
<tr>
<td>RtIntegerSig * out ( MStatus * status = 0 )</td>
<td>Get the output signal</td>
</tr>
<tr>
<td>virtual void resume ( MStatus * status = 0 )</td>
<td>Resume the detector</td>
</tr>
<tr>
<td>virtual void hold ( MStatus * status = 0 )</td>
<td>Hold the detector</td>
</tr>
</tbody>
</table>
9.6 A-B Compare Limit Detector (RtFloatABLdt)

An A-B Compare Limit Detector is used to make sure two signals remain the same (within some tolerance band). It is principally used as a redundancy check with dual-bridge load cells. However, it can be used for other applications.

The RtFloatABLdt class operates much like the RtFloatLdt class in group mode. It has a set of limit detectors that share a few attributes.

When the RtFloatABLdt object is created, the client specifies two signals for each limit. This is done by specifying two lists to the make() method, or by specifying the lists with the connectList() method.

Each detector contains the following attributes. See Table 10 for which attributes apply to all the detectors, and which are specific to each attribute.

- **Limit value** — This is the amount that the two signals must differ before the limit is tripped. This value can be set and queried using the limit() methods, or via the Number parameter returned by getLimitParm(). Each A-B signal pair has its own limit value.

- **Limit action** — This is the RtAction to fire when the limit trips. Applications usually use the connect() method to set the action. This single action applies to all the limits.

  The method getActionParm() returns a OneOfList which can be used to connect a named action to the limit. By default, the list of actions available to the OneOfList is all the named actions on the station. However, setActionList(whichList) and setDefaultActionList() can be called to change to a list containing a subset of all the actions. (whichList is a single value of the enumeration EventTypeMask.) These alternate lists of actions are maintained within RtStation using methods getActions(whichList) and getActionManyList(whichList). Each application has its own definitions of these lists (changes in these lists by Station Manager do not propagate into each application). Changing them within an application is possible, but not recommended.

  **Note**

  The methods getUpAction() and getLoAction() can be used to query the action assignments of the named limit detectors created by Station Manager; however, these OneOfList objects are not automatically updated (connecting a notification callback will not work). You need to explicitly request the value by calling something like getSelectedItem().

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Applies to all signals</th>
<th>Applies to each signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Mode</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Limit Value</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Delay</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Start/Stop State</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Table 10 — Scope of attributes on an A-B Compare Limit Detector

- **ActionMode** — This determines whether each limit will automatically reset when the difference of the A and B signal values crosses back over the limit value. This single
The setting applies to all the detectors in the `RtFloatABLdt` object. The options are defined by the enumeration `TriggerModes` as:

**CONTINUOUS** The each limit will automatically reset once the difference between the A and B signal values crosses back over the limit value.

**ONCE** The application must manually call `reset()` to reset the limits.

**Delay (persistence)** — This is the amount of time (in seconds) the difference in the signals must remain outside the limit value before the limit will fire. The default is zero. This is useful for suppressing multiple trigger-reset cycles, or for avoiding false trips in the presence of noise. This can be set and queried through the `delay()` methods, or through the `Number` object returned by `getPersistenceParm()`.

**Start/stop** — Each limit can be independently started and stopped, or they can be started and stopped together.

**Note:**

Extreme care must be taken when using the **CONTINUOUS** option. If the signal value difference is hovering around the limit value, noise on the signals may make it trip and reset very quickly. This can generate a flood of messages within the system, bogging down the links and the computer.

The **Report Option** attribute determines how the `RtFloatABLdt` object handles multiple limits tripping at the same time. If the **Report Option** is set to **TRUE**, then only the first limit that trips in a single clock tick will fire the action and/or log a message. The other limits will become tripped; they just won’t fire the action. This can help reduce the message traffic when multiple limits trip at once.

The **hold/resume** manipulates a state similar to **start/stop**, except it operates on the while `RtFloatABLdt` object. The default is resume-state. If `hold()` is called, it must be followed by a `resume()` for the detector to be re-enabled.

---

### Related RtFloatABLdt methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static RtFloatABLdt * make ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</code></td>
<td>static constructor to create a new <code>RtFloatABLdt</code> instance</td>
<td></td>
</tr>
<tr>
<td><code>static RtFloatABLdt * make ( RtStation &amp; stn , const Text * name , RtSequence * seq, GenList * sigListA, GenList * sigListB, MStatus * status = 0 )</code></td>
<td>static constructor to create a new <code>RtFloatABLdt</code> instance with sequence and signal lists</td>
<td></td>
</tr>
<tr>
<td><code>static RtFloatABLdt * find ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</code></td>
<td>find a <code>RtFloatABLdt</code> instance by name</td>
<td></td>
</tr>
<tr>
<td><code>virtual ~RtFloatABLdt ( )</code></td>
<td>destructor</td>
<td></td>
</tr>
<tr>
<td><code>boolean  setActionList ( int forError , MStatus * stat = NIL )</code></td>
<td>set action list for error</td>
<td></td>
</tr>
<tr>
<td><code>boolean  setDefaultActionList ( MStatus * stat )</code></td>
<td>set default action list</td>
<td></td>
</tr>
<tr>
<td><code>void  connectList ( GenList * aList , GenList * bList , MStatus * status )</code></td>
<td>connect signal lists <code>aList</code> and <code>bList</code></td>
<td></td>
</tr>
<tr>
<td><code>Number *  getLimitParm ( int sigIndex , MStatus * status = 0 )</code></td>
<td>get limit parameter for signal <code>sigIndex</code></td>
<td></td>
</tr>
<tr>
<td><code>Number *  getPersistenceParm ( int sigIndex , MStatus * status = 0 )</code></td>
<td>get persistence parameter for signal <code>sigIndex</code></td>
<td></td>
</tr>
<tr>
<td><code>OneOfList *  getActionParm ( MStatus * status = 0 )</code></td>
<td>get action parameters</td>
<td></td>
</tr>
<tr>
<td><code>void  start ( MStatus * status = 0 )</code></td>
<td>start detector</td>
<td></td>
</tr>
<tr>
<td><code>void  start ( int sigIndex , MStatus * status = 0 )</code></td>
<td>start detector with signal <code>sigIndex</code></td>
<td></td>
</tr>
<tr>
<td><code>void  stop ( MStatus * status = 0 )</code></td>
<td>stop detector</td>
<td></td>
</tr>
<tr>
<td><code>void  stop ( int sigIndex , MStatus * status = 0 )</code></td>
<td>stop detector with signal <code>sigIndex</code></td>
<td></td>
</tr>
<tr>
<td><code>double  limit ( int sigIndex , MStatus * status = 0 )</code></td>
<td>get limit for signal <code>sigIndex</code></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>void limit (int sigIndex, double aValue, MStatus * = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double delay (int sigIndex, MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void delay (int sigIndex, double aValue, MStatus * = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void reportOption (boolean option, MStatus * status)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean reportOption (MStatus * status)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TriggerModes actionMode (MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void actionMode (TriggerModes aMode, MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void connect (RtAction * anAction, MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void disconnect (RtAction * anAction, MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RtIntegerSig * out (int index, MStatus * = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GenList * getASigList (MStatus * = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GenList * getBSigList (MStatus * = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void resume (MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>void hold (MStatus * status = 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Related RtTask methods (base class of RtFloatABLdt):

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void sequence (RtSequence * seq, MStatus * = 0)</td>
<td></td>
</tr>
<tr>
<td>void connectToSequence (RtSequence &amp; seq, MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual double rate (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void start (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void stop (MStatus * status = 0)</td>
<td></td>
</tr>
<tr>
<td>virtual void reset (MStatus * status = 0)</td>
<td></td>
</tr>
</tbody>
</table>

Related RtStation methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const GenList * getActions(int forEvent, MStatus *stat = NIL);</td>
<td></td>
</tr>
<tr>
<td>ManyOfList * getActionManyList(int forEvent, MStatus *stat = NIL);</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 10
Station Log

Each station contains a station log. The log is a publishing house for messages. Multiple applications may send messages to the log. Applications interested in a copy of the messages may create an **RtLogStream** to tap into the station log’s stream of messages. Messages coming into the station log are sent to all log streams currently connected to it. Currently, there is no limitation on the number of messages in the pipeline, or the number of log streams connected to the log.

The Station Manager connects up a log stream that allows it to log the events to the station log file, and allows its operator to view this information. Other applications may do this same thing to independently write the stream of messages to some file.

Starting in V3.0, Basic TestWare and MPT use a higher-level API that allows Station Manager to manage the test application’s log file. That API is not part of the RT interface, so it is not described here.

The RT interface to the station log is most convenient when the application needs to process the messages itself. The higher-level API is most convenient when the application wants to generate a log file that is handled in the normal way.

10.1 **Station Log Messages**

A message placed into the log has the following attributes:

**Source** — This is a text string indicating which application or portion of the system generated the event. This is only used by applications to filter messages to be stored or viewed. For example, within MPT and StMgr, the operator can specify whether to store all messages, or just those from that application.

There are no restrictions on the contents or length of the source string, except it should not contain multiple lines. In practice, keeping it under 40 characters makes viewing the log easier. To aid in application-level filtering, this string should start with the application name, and optionally include a more detailed component name. Future versions may define additional conventions to aid in filtering messages.

**Severity** — This is an enumeration that further classifies the message. The values are:

- **kDiagnostic**: Information not normally logged, but may be useful for diagnosing problems.
- **kInformation**: Normal information.
- **kWarning**: Operation completed, but with possible side effects.
- **kError**: Operation failed.
- **kFatal**: Operation failed, and application terminated.

The station log does not interpret the severity of a message. It is passed straight through. This is principally used to further characterize the event, and is available for the application to filter the messages by.
Message —

This is the text of the message. It may be arbitrarily long, although it will be truncated at 5000 characters. It may contain multiple lines, separated by line terminators (“\n”). Ideally, the first line should contain the text of the message, with subsequent lines providing additional information. There is no predetermined line length, or message length, but practical limits of viewing and saving the message will ultimately come into play.

Messages sent to the log should already be in the local natural language.

Time stamp —

When a message is put into the RtLog it is immediately given a time stamp. This reflects when the message was logged, not necessarily when the event that generated it happened.

Note:

The message log uses the time-of-day embedded in the system. This is loaded from the workstation with the local time when SYSLOAD initializes the system. It is not subsequently changed. Therefore, if the time-of-day on the workstation is changed (manually or via daylight savings time shift), the system must be reloaded to update its time-of-day.

10.2 RtLog

This class models the station log. The station log is automatically created when the station is created. The easiest way to get a handle to it is to ask the station via RtStation::rtLog(). The only operations a typical application will perform on the log are to put messages into it, and to connect log streams to it. The latter is done with the RtLogStream class.

Related RtStation methods:

```
virtual RtLog * rtLog (MStatus & status = 0)
```

Related RtLog methods:

```
virtual void put ( const Text & source , Severities severity , const Text message )
virtual void put ( const Text & source , Severities severity , const MStatus & status )
static RtLog * find ( RtStation & stn , const Text * name , MStatus * status = 0 )
```

10.3 RtLogStream

An RtLogStream is a tool for reading from an RtLog a stream of messages. RtLogStream objects are created against an RtLog via:

```
RtLogStream * stream = RtLogStream::make(
    *station->getLog(),
    BOOL_CB0(this, MyClass, myLogCallback),
    retStatus);
```

Every time a message comes into the RtLog, it is time-stamped, and sent immediately to each connected RtLogStream. Each log stream contains a queue of unprocessed messages. When the queue is empty, and a message is added to it, the callback specified in the make will be invoked in the application. It is the application’s responsibility to empty the queue, by repeatedly calling get or getFormatted until the stream
is empty. Once the callback has been made once, it will not be made again until the queue becomes empty. Both `get` and `getFormatted` have an optional parameter indicating if there are any more elements in the queue.

The `put` methods are just convenience routines that call directly through to the corresponding `RtLog::put` methods.

When an `RtLogStream` is no longer needed, it should be destroyed, using a normal C++ destruction mechanism.

```
Related RtLogStream methods:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtLogStream * make ( RtLog &amp; rtLog , const Callback0Ptr &lt; boolean &gt; &amp; cb , MStatus * status = 0 )</td>
<td>Creates an <code>RtLogStream</code>.</td>
</tr>
<tr>
<td>virtual ~RtLogStream()</td>
<td>Destructs the <code>RtLogStream</code>.</td>
</tr>
<tr>
<td>virtual long count ( )</td>
<td>Returns the number of entries in the queue.</td>
</tr>
<tr>
<td>virtual RtLog * getLog ( )</td>
<td>Returns a pointer to the log object.</td>
</tr>
<tr>
<td>virtual boolean get ( Text * source = 0 , RtLog :: Severities * severity = 0 , Text * timeStamp = 0 , Text * message = 0 , long numEntries = 0 )</td>
<td>Gets the next entry from the queue.</td>
</tr>
<tr>
<td>virtual boolean getFormatted ( Text * fullMessage, boolean *more )</td>
<td>Gets the next entry from the queue and formats it.</td>
</tr>
<tr>
<td>virtual void put ( const Text &amp; source , RtLog :: Severities severity , const Text message )</td>
<td>Puts an entry into the queue.</td>
</tr>
<tr>
<td>virtual void put ( const Text &amp; source , RtLog :: Severities severity , const MStatus &amp; status )</td>
<td>Puts an entry into the queue.</td>
</tr>
</tbody>
</table>
```
Chapter 11
Interlocks and Power

11.1 Station Interlock

A station has one “station interlock chain” assigned to it when it is loaded. This safety-related interlock will turn off all power on the station in case of some problem.

This object is modeled with the \texttt{RtInterlock} class. The application can get a pointer to the station interlock object by calling \texttt{RtStation::rtInterlock()}. The interlock has the following attributes of interest to an application:

- **State** — The state is either \texttt{interlockReset}, or \texttt{interlockedSet}. When the interlock is tripped, the power will be off. Once interlocked, it stays interlocked until explicitly reset.

- **Override** — If override is asserted, then most sources of the interlock will be ignored, until override is cleared. Some sources (E-stop and internal watchdogs) will fire the interlock, even if override is set. The override state reflects the operator's selection on the Station Manager control panel. Applications may query for the override state, but they should not change it.

The application can force an interlock with the \texttt{interlock()} method. Most applications will not need to reset the interlock. If it does provide such an operator button, it can call the \texttt{resetInterlock()} method. Alternatively, these operations can be done via \texttt{getStateParm()->selectItemAt()}. The \texttt{Override} state can be queried or manipulated through the \texttt{TwoState} returned by \texttt{getOverrideParm()}.

Applications will typically connect an action to the station interlock to be informed when it trips. This can be done as in Example 24.

```
RtAction *act = RtAction::make(*rtStn, 0, BOOL_CB0(this, MyClass, itlkChanged));
rtStn->rtInterlock()->uponInterlock(act);

// callback to be invoked when the interlock state changes.
boolean MyClass::itlkChanged()
{
    if (rtStn->rtInterlock()->state() != interlockReset) {
        // interlock tripped... do something.
    }
}
```

Example 24 — Reacting to a station interlock event

When the station interlock trips, it will automatically stop all the segment generators on the station. Each will perform the appropriate stopping behavior for the profile it is running. The application usually needs to know when the interlock trips, so it knows that the segment generator is now stopped (See Section 7.4.5 on page 63 for more information on segment generators automatically stopping).

The method \texttt{interlockSignal()} returns a pointer to an integer signal that reflects the state of the interlock.
If the application wants to keep the test from running while the program interlock is asserted, it needs to check the state before starting or resuming the test.

**Related RtStation methods:**

- `virtual RtInterlock * rtInterlock ( MStatus * myStatus = 0 )`

**Related RtInterlock methods:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>static RtInterlock * find ( RtStation &amp; stn , const Text * name , MStatus * status = 0 )</code></td>
<td>Find the program interlock by name.</td>
</tr>
<tr>
<td><code>void interlock ( MStatus * status = 0 )</code></td>
<td>Interlock the program.</td>
</tr>
<tr>
<td><code>void resetInterlock ( MStatus * status = 0 )</code></td>
<td>Reset the program interlock.</td>
</tr>
<tr>
<td><code>InterlockStates state ( MStatus * status = 0 )</code></td>
<td>Get the state of the program interlock.</td>
</tr>
<tr>
<td><code>boolean getOverride ( MStatus * status = 0 )</code></td>
<td>Get the override status of the program interlock.</td>
</tr>
<tr>
<td><code>void uponInterlock ( RtAction * action , MStatus * = 0 )</code></td>
<td>Upon the program interlock trip.</td>
</tr>
<tr>
<td><code>void discUponInterlock ( RtAction * action , MStatus * = 0 )</code></td>
<td>Disc upon the program interlock trip.</td>
</tr>
<tr>
<td><code>RtIntegerSig * interlockSignal ( MStatus * status = 0 )</code></td>
<td>Get the interlock signal.</td>
</tr>
<tr>
<td><code>virtual TwoState * getStateParm ( MStatus * status = 0 )</code></td>
<td>Get the state parameter.</td>
</tr>
<tr>
<td><code>virtual TwoState * getOverrideParm ( MStatus * status = 0 )</code></td>
<td>Get the override parameter.</td>
</tr>
</tbody>
</table>

## 11.2 Program Interlock

A station also has a “program interlock” assigned to it. The program interlock is used to keep applications from starting a test until certain conditions are satisfied, and to stop the test when these conditions are violated. When the program interlock trips, it does not shut off power, it just stops the test.

When the program interlock trips, it will automatically stop all the segment generators on the station. Each segment generator will perform the appropriate stopping behavior for the profile it is running. The application usually needs to know when the interlock trips, so it knows that the segment generator is now stopped (See Section 7.4.5 on page 63 for more information on segment generators automatically stopping).

If the application wants to keep the test from running while the program interlock is asserted, it needs to check the state before starting or resuming the test.

Currently, the only external thing tied into the program interlock is the crosshead/grips interlock for TestStar IIs systems. The program interlock is also asserted when a queued mode switch on a channel fails.

The program interlock is also implemented as an **RtInterlock** object. The application can get a pointer to it by calling **RtStation::rtProgramInterlock()**. The program interlock is independent of the station interlock, except that when the station interlock is reset, the program interlock is automatically reset.

For more information on **RtInterlock**, see Section 11.1 on page 149.

## 11.3 Power Control (RtHsm)

**Note**

793 systems historically have been used to drive servo-hydraulic actuators. Therefore, some of the naming conventions in the RT API use “hydraulic” terms. However, 793 systems are also used on systems
that use different technologies (e.g. electrical-mechanical motors). The classes \texttt{RtHsm} and \texttt{RtHps} are used for these applications as well.

A hydraulic service manifold is a “power switch” for hydraulic power. It resides between the hydraulic pump and the actuator/valve assemblies, and allows power to be turned off to an individual actuator, or a set of actuators. The HSM is what reacts to an interlock being asserted, and shuts the hydraulic power off.

The class \texttt{RtHsm} implements the application interface to an HSM on the station. There is one \texttt{RtHsm} object for each physical HSM associated with the station. Actuators may be connected directly to a pump. In this case, the hydraulic pump is presented to the application as an \texttt{RtHsm} as well.

The significant attributes of an HSM are:

**State** — An HSM state is either \texttt{Off}, \texttt{Pilot}, \texttt{Low}, or \texttt{High} (or in a transition state between these states). The full enumeration includes:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerOff</td>
<td></td>
</tr>
<tr>
<td>PowerPilot</td>
<td></td>
</tr>
<tr>
<td>PowerLow</td>
<td></td>
</tr>
<tr>
<td>PowerHigh</td>
<td></td>
</tr>
<tr>
<td>PowerOffToLow</td>
<td></td>
</tr>
<tr>
<td>PowerLowToOff</td>
<td></td>
</tr>
<tr>
<td>PowerLowToHigh</td>
<td></td>
</tr>
<tr>
<td>PowerHighToLow</td>
<td></td>
</tr>
</tbody>
</table>

Not all HSMs support all these states. Some are only off/high. While the “Pilot” states are in the enumeration, they are not used by the system.

**NumberOfStates** — This attribute will always return 2 or 3. It indicates the number of non-transition states that the HSM supports. Currently HSMs are either Off/High or Off/Low/High.

**IsConnectedToMain** — This attribute can be TRUE or FALSE. If TRUE, the HSM cannot be turned on until the Main Power Supply (HPS) is on.

**TypeOfHardware** — This is either \texttt{PowerHPS} or \texttt{PowerHSM}.

An application can find an HSM in one of two main ways:

1. Ask an \texttt{RtChannel} which HSM controls it via \texttt{RtChannel::getHsm()}. If the channel is not assigned to an HSM this will return zero.

2. Get a list of the \texttt{RtHsm} objects on the station by first getting the list of \texttt{RtHsmHWR} objects, then asking each for its \texttt{RtHsm}.

```cpp
GenList *hsmHWRList = myStation->getAllObjects( RtHsmHWRCls, FALSE, &status);
GenList *hsmList = new GenList();
GEN_DO(*hsmHWRList, RtHsmHWR, hwr);
hsmList->addItem(hwr->getHsm());
GEN_END;
```

Most applications are only interested in monitoring the state of the HSMs of interest. This can be queried through:

- The \texttt{getHsmState()} method,
- The \texttt{OneOfList} returned by \texttt{getHsmStateParm()}, or

---

1 This roundabout method is necessary because \texttt{RtHsm} objects are owned by the \texttt{RtHsmHWR} objects, so they do not appear on the station’s list.
The predefined integer signals “XXXX On” and “XXXX High” (where XXXX is the name of the HSM). See Table 11 for what the values of these signals will be. Find these signals in a list of RtIntegerSig objects returned by RtStation::getAllObjects().

In addition, the application can connect an action to be performed when the state changes.

**Note**

Applications will not necessarily see every state transition. When the state change callback is delivered to the application process space, the callback turns around and queries the machine for the new state. In the meanwhile, the state may have changed again. When the second callback occurs, the application will see no change. Applications must tolerate this race condition.

An application may change the power state via setHsmState() or via the OneOfList returned by getHsmStateParm(). When doing so, the application should only specify one of the non-transition states (e.g., PowerOff, PowerLow, or PowerHigh). This will initiate the state change, and the state machine will go through the other states as appropriate.

If the requested new state cannot be accommodated the call will return an error. Applications can avoid this error by calling RtStation::isSafeToChangePowerState() to determine if the proposed change is allowed. This will return a mask indicating the reasons that power cannot be applied. The reason could be one or more of the following:

- IS_SAFE
- STATION_INTERLOCKED
- CHANNEL_RUNNING
- CHANNEL_SATURATED
- POWER_DISABLED
- ACTUATOR_IN_MOTION
- SIGNAL_INVALID
- LOW_BEFORE_HIGH
- MAIN_POWER_MUST_BE_ON

Applications have some flexibility to override this safety condition by passing a PowerSafetyMask to setHsmState(). However, currently, the only safety reason that can be overridden is CHANNEL_SATURATED.

### Table 11 — Values of predefined HSM integer signals

<table>
<thead>
<tr>
<th>State</th>
<th>“XXXX On”</th>
<th>“XXXX High”</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerOff</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PowerPilot</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerLow</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerHigh</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PowerLowToOff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerOffToPilot</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerOffToLow</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerLowToHigh</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>PowerHighToLow</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Related RtHsm methods:**

<table>
<thead>
<tr>
<th>RtHsm method</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneOfList *</td>
<td>getHSMStateParm ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>void</td>
<td>setHsmState ( PowerStates newState , MStatus * status = NIL )</td>
</tr>
<tr>
<td>void</td>
<td>setHsmState ( PowerStates newState , PowerSafetyMask overrideMask , MStatus * status = NIL )</td>
</tr>
<tr>
<td>PowerStates</td>
<td>getHsmState ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>int</td>
<td>numStates ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>void</td>
<td>connect ( RtAction * act , MStatus * status = NIL )</td>
</tr>
</tbody>
</table>
### 11.4 Main Power (RtHps)

**Note**

793 systems historically have been used to drive servo-hydraulic actuators. Therefore, some of the naming conventions in the RT API use “hydraulic” terms. However, 793 systems are also used on systems that use different technologies (e.g., electrical-mechanical motors). The classes `RtHsm` and `RtHps` are used for these applications as well.

A hydraulic power supply (HPS) provides hydraulic power to the actuators via the HSMs. While all hydraulic systems are connected to a power supply somehow, the software may or may not have direct access to it. As the 793 platform expands, the number of permutations and combinations of these devices continues to grow.

Sometimes the system uses a “house pump” over which it has no control. Sometimes the HPS is controlled indirectly through a first-on-last-off protocol. Sometimes, the HPS is directly controlling the actuators (i.e., no HSMs exist). In this last case, the HPS is modeled to the software as an HSM.

Most applications do not care about state of the main power supply; they are only interested in if their channels have power applied, which is controlled by the HSM.

The class `RtHps` implements the application interface to the HPS on the system. An application can find the HPS by calling `RtStation::getMainPower()`. In many configurations, this will return NULL.

If a system contains an HPS, it will have only one. When this is supported on multiple stations, each station will have its own `RtHPS` instance, but it will all reflect the state of the same physical HPS.

An HPS contains the following attributes:

**State**

- An HPS state is either **Off**, **Low**, or **High** (or in a transition state between these states). The full enumeration includes:
  - `PowerOff`
  - `PowerPilot`
  - `PowerLow`
  - `PowerHigh`
  - `PowerLowToOff`
  - `PowerOffToPilot`
  - `PowerOffToLow`
  - `PowerLowToHigh`
  - `PowerHighToLow`
Not all HPS units support all these states. Some are only off/high. While the “Pilot” states are in the enumeration, they are not used by the system.

**NumberOfStates** — This attribute will always return 2 or 3. It indicates the number of non-transition states that the HPS supports. Currently all HPS units are Off/Low/High.

**Visible** — The HPS will appear on the Station Manager control panel, and be controllable from the RSC. This is set in the HWI file.

**FirstOn** — The HPS will automatically come on when the first HSM is turned on. This is set in the HWI file.

**LastOff** — The HPS will automatically turn off when the last HSM is turned off. This is set in the HWI file.

Most applications are only interested in monitoring the state of the HPS. This can either be done with `getHpsState()`, or via the OneOfList returned by `getHPSStateParm()`. In addition, the application can connect an action to be performed when the state changes.

**Note**

Applications will not necessarily see every state transition. When the state change callback is delivered to the application process space, the callback turns around and queries the machine for the new state. In the meanwhile, the state may have changed again. When the second callback occurs, the application will see no change. Applications must tolerate this race condition.

An application may change the power state via `setHpsState()`, or via the OneOfList returned by `getHPSStateParm()`. When doing so, the application should only specify one of the non-transition states (e.g., PowerOff, PowerLow, or PowerHigh). This will initiate the state change, and the state machine will go through the other states as appropriate. If the requested new state cannot be accommodated the call will return an error.

### Related RtHps methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneOfList *</td>
<td><code>getHPSStateParm</code> ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>void</td>
<td><code>setHpsState</code> ( PowerStates newState , MStatus * status = NIL )</td>
</tr>
<tr>
<td>PowerStates</td>
<td><code>getHpsState</code> ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>numStates</td>
<td><code>connect</code> ( RtAction * act , MStatus * status = NIL )</td>
</tr>
<tr>
<td>void</td>
<td><code>disconnect</code> ( RtAction * act , MStatus * status = NIL )</td>
</tr>
<tr>
<td>boolean</td>
<td><code>getVisible</code> ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>boolean</td>
<td><code>getFirstOn</code> ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>boolean</td>
<td><code>getLastOff</code> ( MStatus * status = NIL )</td>
</tr>
</tbody>
</table>

### Related RtStation methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RtHps *</td>
<td><code>getMainPower</code> ( MStatus * status = NIL )</td>
</tr>
</tbody>
</table>
11.5 **Aggregate Station Power States**

The station provides methods to get the *minimum* and *maximum* power state of all the HSMs on the station. In some cases, this allows the application to get the information it needs without individually polling all the HSMs. The minimum and maximum power states do not include transition states.

For example, if an application needs all the HSMs in HIGH before running, it can make sure the minimum state is HIGH. If it needs all the HSMs to be OFF, it can make sure the maximum state is OFF or PILOT.

The main power state (if any) is not included in these aggregate states.

<table>
<thead>
<tr>
<th>Related RtStation methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneOfList * getMaxHydStateParm ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>OneOfList * getMinHydStateParm ( MStatus * status = NIL )</td>
</tr>
<tr>
<td>boolean uponStnHydChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean disconUponStnHydChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>PowerStates getHydState ( RangeType minmax , MStatus * status = NIL ) const</td>
</tr>
</tbody>
</table>
Chapter 12
Additional Station Attributes

12.1 Station Test State and Controlling Applications

A station maintains a “Test State”, which is intended to reflect the Run/Hold/Stop state that the operator considers the test to be in. This test state has been historically maintained by the individual application, and displayed to the operator in an application-specific control panel.

The purpose of a station-wide test state is to:

1. Reflect this test-state on non-application-specific displays. This includes the Remote Station Controller (RSC) and the Station Manager control panel.

2. Route non-application-specific run/hold/stop button presses to the controlling application.

3. Allow monitor and summary applications to watch and react to state changes by the controlling application.

The current test state of a station is accessed and controlled through several interfaces:

- The methods `getTestState()` and `setTestState()`, as described below

- The `OneOfList` returned by `getTestStateParm()` can be used to monitor this state, but cannot be used to set it.

- The predefined integer signals named “Run/Stop” and “Hold” also reflect the current test state, as shown in Table 12. Find these signals in a list of `RtIntegerSig` objects returned by `RtStation::getAllObjects()`.

The enumeration `ProgramState` is used to reflect these states. Table 12 shows these states, and the corresponding light pattern on the Remote Station Controller (RSC).

All applications can query for the current test state, and be notified when the test state changes. This allows monitor and summary applications to react to changes. However, only applications of type `controlApp` or `managerApp` (see Section 3.2.2 on page 15) can control the test state, and even then, only one at a time. The current controlling application is the one that can change the test state, and receives run/hold/stop button presses from station-wide displays like the RSC.

<table>
<thead>
<tr>
<th>ProgramState</th>
<th>RSC Light</th>
<th>Run/Stop Signal</th>
<th>Hold Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUN</td>
<td>Run light on</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>STOP</td>
<td>Stop light on</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HOLD</td>
<td>Hold light on</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>STARTING</td>
<td>Run light blinking</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>STOPPING</td>
<td>Stop light blinking</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>HOLDING</td>
<td>Hold light blinking</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>RESUMING</td>
<td>Run light blinking</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 12 — Test states, and corresponding RSC light patterns and signal values.
The current controlling application can only be changed while the test state is STOP. While this is so, any application that attempts to change the test state will become the controlling application.

For example, when an application wants to start running a test, it can call `setTestState(STARTING)`. This will cause it to become the controlling application and put the test state into STARTING (which will blink the run light on the RSC). After this, any other application that tries to set the test state will get back an error. When this controlling application subsequently calls `setTestState(STOP)`, it will continue to be the controlling application, but this ownership can then be changed by another application calling `setTestState`.

**Note:**

The methods `requestTestControl()` and `releaseTestControl()` can also be used to affect which is the controlling application, but they are only intended to be used by Station Manager, and are not needed by most applications.

It is possible to find out which application is the controlling application through the `OneOfList` returned by `getCtrlAppParm()`. The Station Manager control panel and the RSC main window display to the operator what application is the controlling application. This indicates which application will run if he/she presses the RUN button.

The application can be notified when the test state or controlling application changes by setting a callback on the objects returned by `getTestStateParm()` or `getCtrlAppParm()`, or by connecting an action with `uponStateChg()`.

**Note**

The `uponStateChg()` action is called for a variety of station state changes, not just these states.

As mentioned previously, the Station Manager and RSC run/hold/stop buttons are delivered to the current controlling application. This is done through the “Requested Test State” attribute. The easiest way for applications to field these events is to connect an `RtAction` with `uponReqStateChg()`. This will cause the action to fire whenever this application is the current controlling application, and one of the buttons is pressed. The resulting callback needs to call `getReqTestState()` to determine which button is pressed.

Other applications may also request a state change by calling `setStateChgReq()`. This will route the request to the controlling application. The caller has no way of knowing if the request was accepted, except to wait for a possible test state change notification to arrive.

When the controlling application calls `getReqTestState()`, it receives two pieces of information:

1. Return value of STARTING, HOLDING, or STOPPING. This reflects the most recent request that came from the RSC or another application. If multiple requests came in before the controlling application makes this call, it will only see the most recent one.

2. The source argument gets filled in with the name of the source of the request. This will be an application name, or the name of the RSC resource. If the application cannot perform the requested state change, it should use this argument to determine where to display the error message. This parameter is optional, and can be set to NULL if the caller is not interested in the source of the request.

Example 25 shows how to use the source argument to determine where to display any error an application encounters while processing a requested test state change. The `PMessageBox` class is part of the RSC API.
See Chapter 13 starting on page 163 for more information about this API.

```cpp
boolean TestController::requestTestStateChanged()
{
    MStatus status;
    boolean success = FALSE;

    Text whereFrom;
    ProgramState newState = stn_->getReqTestState(&whereFrom);

    testStateButtonPressed(newState, &status);
    if (!status.isOk())
        displayStatus(status, whereFrom);
}

boolean TestController::displayStatus(const MStatus &status, const Text &whereFrom)
{
    // if no pod or message was not from the pod, display on the screen.
    if (!stn_->getRSC() || whereFrom != stn_->getRSC()->parmName())
        status.display();

    // if no message box is being displayed, create it.
    else if (!pMessageBox_)
        pMessageBox_ = new PMessageBox(*stn_, // station
            _T(""), // RSC name (default)
            status, // message
            _T("Basic TestWare"), // title
            BOOL_CB1(this, TestController, messageBoxDismissed, UINT));

    // else update the text in the existing pod.
    else
        pMessageBox_->putStatus(status);

    return TRUE;
}

// clear the pointer if the window is dismissed.
boolean TestController::messageBoxDismissed(UINT buttonIndex)
{
    pMessageBox_ = 0;
    return TRUE;  // this tells the message box to delete itself.
}
```

Example 25 — Determining where to display errors when handling requested test state changes

Note:

The application can only connect one action to
uponReqStateChg(). Future versions may relax this restriction.

<table>
<thead>
<tr>
<th>Related RtStation methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneOfList * getTestStateParm( MStatus * = 0 );</td>
</tr>
<tr>
<td>ProgramState getTestState(MStatus * = 0) const;</td>
</tr>
<tr>
<td>boolean setTestState(ProgramState aState, MStatus * = 0);</td>
</tr>
<tr>
<td>boolean uponStateChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>boolean disconUponStateChg ( RtAction * act , MStatus * status = 0 )</td>
</tr>
<tr>
<td>OneOfList * getReqStateParm( MStatus * = 0 );</td>
</tr>
<tr>
<td>boolean uponReqStateChg(RtAction *act,MStatus * =0);</td>
</tr>
<tr>
<td>boolean disconUponReqStateChg(RtAction *act,MStatus * =0);</td>
</tr>
<tr>
<td>ProgramState getReqTestState(Text *source = 0, MStatus * = 0);</td>
</tr>
</tbody>
</table>
void stateChgReq(ProgramState requestedState, MStatus * = 0);

OneOfList * getCtrlAppParm( MStatus * = 0 );
const GenList * getCtrlAppListParm( MStatus * = 0 );
boolean uponAppListChg(RtAction *act,MStatus * =0)
boolean disconUponAppListChg(RtAction *act,MStatus * =0)

12.2 Semaphores (RtSemaphore)

RtSemaphore objects are named station objects that are used to coordinate between applications. They are used in a couple of different places to arbitrate the use of resources on the station. However, they can be created and used for other purposes as well.

An RtSemaphore is created by calling RtStation::getSemaphore(name). If a semaphore of that name already exists, a reference to it is returned. If it does not exist, then one will be created and returned.

The method RtStation::getSemaphoreList() returns a list of all the RtSemaphore objects that exist on the station. This list will be automatically updated if a semaphore is created.

Since RtSemaphores are named station objects, they are not destroyed until the station is unloaded.

RtSemaphores have the following attributes:

Name — This is a unique name on the station. Currently, only the internal name is used. The display label is not maintained.

Owner — A semaphore can only have one owner at a time. A request to take() ownership of a semaphore that is already owned by another application will return FALSE. The current owner releases ownership by calling the give() method. If the application that owns a semaphore terminates, the ownership is automatically released.

Value — This string value can be written by the owner, and read by anyone. It is assigned by specifying a value on the take(value) method, or by explicitly calling setSemValue(value). Again, only the internal string is used (not the label).

Several methods are available to query for these attributes (see below).

Related RtStation methods:
const GenList * getSemaphoreList ( MStatus * status = NIL )
RtSemaphore * getSemaphore ( const Text & semName , MStatus * status = NIL )

Related RtSemaphore methods:
boolean take ( const Text & newValue , MStatus * status = NIL )
void give ( MStatus * status = NIL )
Text semaphoreValue ( MStatus * status = NIL )
Text semaphoreOwner ( MStatus * status = NIL )
boolean setSemValue ( const Text & newValue , MStatus * status = NIL )

boolean isSemEqual ( const Text & value , MStatus * status = NIL )
boolean amOwner ( MStatus * status = NIL )
void uponSemaphoreChanged ( RtAction * act , MStatus * status = NIL )
void disconUponSemChanged ( RtAction * act , MStatus * status = NIL )

12.3 Exclusive Control

Note

At this time, Exclusive Control is only exported to the user interface on Automated FlexTest SE systems. However, it is in the API in all systems, and may migrate to into the user interface for other systems in the future.

Exclusive Control is implemented in the FlexTest SE automated mode to allow the operator to dynamically arbitrate which interface (computer or front panel) the operator can use to affect the system. The intention is to allow the operator who “has his fingers close to the actuator” to keep another person from walking up to different interface and making changes that may cause the actuator to move. As such, it is designed to lock out controls that may affect the actuator while a test is not running.

Most controls within a test application are only active while a test is running. These do not need to follow any Exclusive Control protocol. However, sometimes an application does provide live, manual-operation controls what should be locked out when Exclusive Control is not allocated to the screen that the application is using. An example of this is the Home button in Basic TestWare.

For the purposes of exclusive control, there are two different types of application scenarios:

1. Applications that run on the same interface as Station Manager, and
2. Applications that run on a separate computer from Station Manager

Exclusive Control is implemented as an RtSemaphore. See Section 12.2 on page 160 for more explanation about the RtSemaphore class. The application that provides a way to enable/disable exclusive control on an interface locks and unlocks the semaphore. The value of the semaphore is set to a name for the interface.

12.3.1 Applications on Station Manager’s Screen

Most applications fall into this category. Applications that embed themselves into Station Manager definitely fall into this category. This class of application should keep manual operation controls locked unless Station Manager has exclusive control.

The sample routine in Example 27 returns ‘true’ if the state of exclusive control is such that the manual controls need to be locked.

```c
bool areControlsActive(RtStation *stn)
{
    RtSemaphore* sem = stn->getSemaphore("Exclusive Control Semaphore");
    Text owner = sem->semaphoreOwner();
    bool stMgrOwnsIt = owner == "Station Manager";
    bool someoneOwnsIt = (owner != "");
    bool someoneElseOwnsIt = (!stMgrOwnsIt && someoneOwnsIt);

    TextParm* req = CAST(TextParm,
                            sem->getPropertyListParm()->getItemNamed("Exclusive Control Required");
    bool exclRequired = (*req->getTxt() == "Yes");

    return (exclRequired && stMgrOwnsIt) || (!exclRequired && !someoneElseOwnsIt);
}
```

Example 27 — Determining if exclusive control requires manual controls to be locked.
controls should be enabled. In addition, the application should connect an \texttt{RtAction} to the \texttt{RtSemaphore} to be notified when the semaphore changes, so it can lock or unlock the controls appropriately.

\textbf{Note:}

In the current version, the value of the “exclRequired” variable will always be “Yes” on Automated FlexTest SE systems, and “No” on all other systems. In the future, this may be a user-selectable attribute.

\textbf{Note:}

The method \texttt{getPropertyListParm()} is not currently documented or supported. It should only be used in places where you are explicitly instructed to do so.

\subsection*{12.3.2 Applications on Their Own Screen}

The logic for these types of applications is more complicated, and not yet publicly supported.
Chapter 13
Remote Station Controller (RSC)

The Remote Station Controller (RSC or Pod for short) is a dedicated control panel for operating a station. It contains dedicated pushbuttons for Power control and test state (run/hold/stop). It also contains an E-Stop button, a manual control knob, and a display with programmable function keys.

This interface is used to load specimens, and run some pre-defined tests. It is not used to configure systems or tests.

An RSC is assigned to a particular station interlock in the HWI file. When loading a station with Station Manager, the operator may determine whether or not to use the RSC. It is then dedicated to that station for the life of the station.

Most of the interaction with the RSC is handled by the system and Station Manager. An application may interact with the RSC in several ways:

1. By managing the Station Test State, the application can react to the RSC Run/Hold/Stop buttons, and have its test state reflected on the lights (see Chapter 12 on page 157).

2. The application may create pages (windows) on the RSC display. This allows display of information, and handling the function keys while that page has focus. This is done through the RSCAPI DLL, which is (will be) documented elsewhere.

3. When a controlling application receives a Run request from the operator via the RUN pushbutton on the RSC, and it cannot perform the Run for some reason, it is common to display the error message on the RSC. Example 25 on page 159 shows typical code for accomplishing this.

The class \texttt{RtRsc} provides access to the Remote Station Controller. If a station has an RSC allocated to it, the method \texttt{RtStation::getRSC()} will return a non-null pointer to an \texttt{RtRsc}. The only thing an application can do with this pointer is to pass it to the RSCAPI when creating display pages, or query for its name..

\begin{tabular}{|l|}
\hline
\textbf{Related RtStation methods:} \\
\hline
\texttt{virtual RtRsc * getRSC(MStatus * = 0);} \\
\hline
\end{tabular}
Chapter 14
Hardware Resources Revisited

Hardware Resources are objects created when the system is loaded, which reflect the physical hardware contained in the box. When a station is loaded, the hardware resources referenced by the station configuration are “checked out” of the system, and assigned to the station. They remain unavailable for other stations, until the owning station is unloaded. At that time, they are returned to the system pool.

All of this activity is managed by SYSLOAD and the Station Manager, and is transparent to typical test applications. Applications will occasionally need to reference the hardware resource objects to get information.

Currently, there are seven hardware resource classes:
- RtAnalogInputHWR,
- RtAnalogOutputHWR,
- RtDigitalInputHWR,
- RtDigitalOutputHWR,
- RtHsmHWR,
- RtInterlock, and
- RtRsc.

The RtInterlock and RtRsc objects are unique, in that they are assigned to a station at load time, not in the Station Builder. The use of the RtInterlock objects are described in Section 11.1 on page 149, and the RtRsc is described in Chapter 13 on page 161.

The rest are discussed somewhat in this chapter.

14.1 List of signals for a kind of resource

When input and output resources are allocated to a station, signal objects are created with the logical names assigned by the Station Builder. Signals (floating point or integer) are the way most manipulation of those resources is done. However, sometimes the application needs a list of all the signals associated with a particular type of resource (e.g., the RtIntegerSig objects for all the digital inputs).

One way of doing this is shown in Example 28.

```c
const GenList *rtInputList = rtStation_->getAllObjects(RtDigitalInputHWRCls);
GenList *sigList = new GenList();
GEN_DO(*rtInputList, RtDigitalInputHWR, input);
  sigList->addItem(input->getIntegerSignal());
GEN_END;
```

Example 28 — Getting the list of digital input signals (one way).

This code gets the list of all the RtDigitalInputHWR objects. Then, from each of these it queries for the signal associated with the resource, and adds it to a new list. The other types of hardware resources can be handled the same.

An alternate way of doing this is shown in Example 29.
This code scans the list of all `RtIntegerSig` objects, selecting only the ones that have a source of class (or subclass) `RtDigitalInputHWR`. For some types of resources, it would be appropriate to check the `sink()` method instead. See Section 4.1.5 on page 30 for more information about `source()` and `sink()`.

Remember that the name of the HWR objects is the hardware name from the HWI file. The name of the signal is the station logical name specified in the Station Builder.

### 14.2 Locking a digital output for exclusive use

Digital outputs can be assigned in Station Manager to output certain integer signals, or they can be left unassigned, so applications can use them.

When an application wants to use a digital output, it must reserve it by gaining ownership of the `RtSemaphore` associated with that digital output, as shown in Example 30.

```
RtDigitalHWR *digHwr = CAST(RtDigitalHWR, rtOutput_->sink());

boolean lockStat = digHwr->getSemaphore()->take(""};
if (!lockStat)
// could not gain ownership of the digital output.
```

For example, when MPT locks the procedure, it will `take()` the semaphore for each digital output that the test procedure will use. Station Manager will not allow the operator to change the digital outputs that are locked, or assign them to specific signals. This gives the MPT procedure exclusive control over those digital outputs.

The application can also find the semaphore by getting the semaphore list from the station, and looking up the semaphore by the name “XXXXRtDigitalOutputHWRSemaphore” (where XXXX is the name of the digital output hardware resource.

**Note**

This locking mechanism does not lock out `RtActions` that set digital outputs. Care must be used when using digital output commands in `RtActions` so as not to get unpredictable results.
14.3 Other hardware resource information

While signals are the objects that applications will usually manipulate, the hardware resources behind them do much of the actual internal work. Occasionally, applications need to reach into the hardware resource to extract configuration, calibration, or tuning information. Applications may do this, but they should not change any of the settings. Unpredictable results may occur, since the Station Manager (which manages the information) will be unaware of the change, and may change it back without warning.

The structure within a hardware resource may be documented in the future...

14.4 Calculation Parameters

Calculated Signals are station resources that are created in Station Builder and parameterized in Station Manager. The calculations behind these signals are edited in Station Manager, and cannot be viewed or modified in other applications.

However, these calculations can make use of Calculation Parameters to calculate their output values. Calculation Parameters can be modified by applications.

Calculation Parameters are named station objects, so you can find them with RtCalcParm::find(), or with RtStationGetAllObjects() (See Section 3.3.3 on page 19).

In Station Manager, when a calculation parameter is set up, an Access Level is specified. The operator must be at that access level to modify the parameter. However, access level is only a Station Manager concept, so applications are not bound by this specification.

In Station Manager, the user is not allowed to change a calculation parameter that is used in an active control mode's feedback when hydraulic power is being applied. This is a safety policy.

Within a separate application, there are two ways to modify the value of a calculation parameter:

1. Via RtCalcParm::getValueParm()->setValue(xxx) — This path is used by Station Manager. It enforces the safety policy mentioned above.

2. Via RtCalcParm::value(xxx) — This path bypasses the safety checks. You are free to change any calculation parameter at any time. Just be aware that the change will be a step change in the value (there is no smoothing ramp). Therefore, if you are changing something used in the active control loop, you may need to do it in small increments.

Any change that an application makes to a calculation parameter will be reflected in Station Manager’s user interface. It will also mark the parameter set as being modified.

Related RtCalcParm methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static RtCalcParm * find ( RtStation &amp; stn , const Text &amp; name , MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>void attribute ( Text * dim , Text * unit , double * res , double * sLower , double * sUpper , double * hLower , double * hUpper , MStatus * status = NIL )</td>
<td></td>
</tr>
<tr>
<td>Number * getValueParm ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>SubRange * getFSParm ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>double value ( MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>void value ( double val , MStatus * status = 0 )</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>Function</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>boolean</td>
<td><code>getFullScale</code></td>
</tr>
<tr>
<td>void</td>
<td><code>uponFullScaleChange</code></td>
</tr>
<tr>
<td>void</td>
<td><code>disconUponFullScaleChange</code></td>
</tr>
</tbody>
</table>
Chapter 15
Typical Application Sequence

“Typical” applications and “Example” applications are two different things. Examples (like the code fragments shown in this document) are intended to demonstrate some aspect of the structure without being bogged down by unrelated things that are not germane to the discussion at hand. “Typical” applications need to worry about much more. They must:

Handle errors — Error checking is non-existent in our code fragments.

Query the operator for information — our samples all contained hard-coded numbers and resource names.

React to events — Real-time applications need to react to events from both the operator (i.e., STOP button), and the machine. Thus, they tend to do everything within user-interface or real-time callbacks.

May or may not be multi-threaded — Most applications written to this API are not multi-threaded. They handle all events within the main thread. This still gives the effect of concurrency, as long as each callback isn’t blocked for any significant period.

Adding multiple threads to an application adds complexity. This may be necessary to improve responsiveness to a time-critical event, or to spawn some long calculation to the background so as not to block subsequent user interaction. Remember, each thread that will field events from the machine must have an RtNotifier created within it (see Section 2.2.6 on page 8 for more information about handling events).

With these considerations in mind, here is the sequence of the real-time portion of a “typical” test application. How this sequence relates to the user-interface setup depends on many other factors.

1. Connect to a station (Chapter 3)
   1.1. Create an RtNotifier object.
   1.2. Connect to the System.
   1.3. Get the list of loaded stations.
   1.4. Connect to a station of interest.

2. Define command generation (Chapter 7).
   2.1. Query for the list of RtChannel objects.
   2.2. Lock the segment generator for each channel of interest.
   2.3. If necessary, create an RtSyncGrp and add the channels to it.
   2.4. Create and queue RtProfile objects.
   2.5. Create and connect RtAction objects to watch for anything that may automatically shut down the command generation (Hydraulic Interlock, Program Interlock, Stop Actions, or HSM off). See Section 7.4.5 on page 63 for more information).

3. Define data acquisition (Chapter 8).
   3.1. Query for the list of RtFloatSig and RtIntegerSig objects.
   3.2. Create data acquisition objects to monitor and sample the signals of interest.
       3.2.1. Create and configure RtAcq, RtTrigger, and RtBuffer subclasses.
       3.2.2. Create and connect RtAction objects to the events of interest.
4. Define limit detectors (Chapter 9)
   4.1. Query for the list of RtFloatSig objects.
   4.2. Create RtFloatLdt and RtPeakValleyLdt objects to monitor the signals of interest, connecting RtActions.

5. Define Digital Input detectors (Chapter 9).
   5.1. Query for the list of RtIntegerSig objects.
   5.2. Create RtBooleanLdt objects to monitor the signals of interest, connecting RtActions.

6. When ready, start the defined components.
   6.1. Verify that interlocks and power states are in a condition where you will allow the test to run.
   6.2. Start up data acquisition and detectors.
   6.3. Start up command generation.

7. React to the callbacks generated by the user-interface, and the RtActions.

8. At the end of the test:
   8.1. Stop the defined components.
   8.2. Destroy the created objects.
   8.3. Unlock the segment generator.
   8.4. Disconnect from the station.

15.1 Demonstration MFC Application (mydemo)

The application mydemo is distributed with the Real-time Application Programming Interface support as an example of a typical application. It is intended to show how to use some of the features of the interface, without being cluttered with too much unrelated code. The README.TXT file goes into more detail on what the application does, how it was created, and other important aspects of the implementation.

**Note**

Pay special attention to the changes in the wizard-generated file STDAFX.H that are discussed in the README.TXT.

15.2 Example code for Allegris Applications

There is not yet a demonstration program for Allegris applications. It would be useful to have a version of mydemo that used the Allegris classes, but this does not yet exist.

The best examples are the existing supported applications (i.e., Basic TestWare, and MPT).
Chapter 16
Build Procedures and Debugging

16.1 Visual C++

Applications built to run on MTS 793 V3.2 must be built using Visual C++ V6.0 with Service Pack 3 or later. No special options are required, or restrictions imposed on the installation of Visual C++. It can be configured as you wish. Most engineers use the “typical” installation.

Once the installation is complete, the path `x:\ftiim\ntbin` should be added to the library search. This is done by:

1. Open Tools — Options... — Directories
2. Select “Show directories for: Library files”
3. Add `z:\ftiim\ntbin` to the list (where `z:` is the appropriate drive letter)

16.2 Building a Visual C++ project

MTS 793 applications are typically constructed as Windows applications. They can be Console Applications, but they must have a message loop in the main thread to handle events from the machine.

MTS 793 applications must NOT be built for UNICODE.

The following options are required for Visual C++ projects which use the Real-time API.

- **C/C++ — C++ Language** — Enable Run-Time type Information.
- **C/C++ — Code Generation** — use run-time library:
  - Multi-threaded DLL
- **C/C++ — Code Generation** — Calling convention:
  - _cdecl (the default)
- **C/C++ — Code Generation** — Struct member alignment:
  - 8 bytes (the default)
- **C/C++ — Preprocessor** — additional include directories:
  - `\ftiim\ntinclude` (if possible, use relative path-names)
  - `\ftiim\cppviews\ntinclude`
- **C/C++ — Preprocessor** — Preprocessor Definitions:
  - `NDEBUG,_WINDOWS,WIN32_LEAN_AND_MEAN`, // from Microsoft
  - `CV_WIN32,USE_DLL,CV_CTL3D` // from Allegris
  - `__NT__,CPPVIEWS` // from MTS
All of these macros should be specified, even if Allegris is not being used for the user interface. Some of them may not be technically required, but it isn’t obvious which ones are needed for what features. This list may be reduced in subsequent versions.

- Link — General — Object/library modules:
  - `rt.lib`, `rtnote.lib`, `genutil.lib`, `mts32olv.lib`, `mtsvw001i.lib`

- Project Workspace — These source files need to be added directly to the project:
  - `clsimp.cpp`, `rt_imp.cpp`, and `genimp.cpp`

  They are needed to satisfy Allegris run-time type-checking globals of the form xxxxxxxCls. These files are located in the `src\common` or `src\commonnt` directories.

The resulting executable can be put anywhere, but it is normal to put it into the `ntbin` directory.

### 16.3 Debugging

Applications can be debugged using the Microsoft Developer Studio in the normal way. This section describes some techniques that can be used during development. They should not be used while running tests.

#### 16.3.1 Disabling the watchdog

Most applications are debugged in simulation mode. However, there are times when it is useful or necessary to debug using real hardware. The communication path between applications and a real machine is protected with a watchdog timer to properly clean up if the application malfunctions. This watchdog timer is only a few seconds long, and will usually fire if the application hits a breakpoint. The application will see this as a “communication loss” error.

This can be avoided by adding the `/NOWD` flag to the SYSLOAD command. This will disable the watchdog, but will also disallow clearing the station interlock. This is how it was designed as a safety feature.

**Note**

This feature is only available in a debug build of SYSLOAD.

#### 16.3.2 Removing station manager’s “always-on-top”

While loading a station, Station Manager sets the “always-on-top” attribute. This is done so the progress indicator on the status bar continually shows the current activity. While using the debugger, it is convenient to turn this off (so the debugger can occlude Station Manager). This can be done by putting the “/D” option on the Station Manager command line.
16.3.3 “Fatal Error” processing

The error handling mechanism described in Section 2.2.1 on page 5 will report errors that are unhandled or errors with “fatal” severity to the operator via a message box. When the operator dismisses the dialog the application is terminated.

The fatal error message is displayed in a new thread created just for that purpose. The thread that experienced the error will be blocked in a modal message loop that only lets WM_PAINT messages through. Any other threads that see a fatal error will end up in the same place. You can determine which one experienced the initial error by looking at the iAmFirst variable in ErrorManager::fatalExit() within the context of each thread.