CyFlex® Test Manager User Guide

Version 9

February 14, 2024

Developed by Transportation Laboratories

Version History

Version	Date	Revision Description
1	1/4/2016	Initial publication
2	3/30/2018	Added new instance definition format to *Section 8 Instance Definition File*
3	4/3/2020	Added *Section 2.12 Sequence of Execution* Retrofit to new template
4	8/18/2021	Revised *Section 2.11 Conditional Tests* on page 4 and *Section 2.12 Sequence of Execution* on page 5 to add supporting content for the new `@PATH_OPTIONS` keyword as of CyFlex 6.3.26.
5	12/202021	Removed row describing `AT_NT` kill option from *Table 1* on page 3 As applicable, removed command usage content with and added hypertext linked cross-references to usages on cyflex.com and to cyflex.com reference documentation.
6	6/2/2022	Updated all hypertext linked cross-references to cyflex.com usage help descriptions
7	9/22/2022	Removed duplicate `@PID_GAINS` entry from step 3 of *Section 2.12 Sequence of Execution* on page 5
8	2/20/2023	Removed all references to Cummins Electronic Engine Controller (ECM)
9	2/14/2024	Rebrand to TRP Laboratories

Document Conventions

This document uses the following typographic and syntax conventions.

- Commands, command options, file names or any user-entered input appear in Courier type. Variables appear in Courier italic type.

  Example: Select the `cmdapp-`*`relVersion-buildVersion`*`.zip` file….

- User interface elements, such as field names, button names, menus, menu commands, and items in clickable dropdown lists, appear in Arial bold type.

  Example: **Type**: Click **Select Type** to display drop-down menu options.

- Cross-references are designated in Arial italics.

  Example: Refer to *Figure 1*…

- Click intra-document cross-references and page references to display the stated destination.

  Example: Refer to *Section 1* Test Manager Overview on page 1.

  The clickable cross-references in the preceding example are *1*, *Overview*, and on page 1.

CyFlex Documentation

CyFlex documentation is available at https://cyflex.com/. View **Help & Docs** topics or use the **Search** facility to find topics of interest.

Table of Contents

LIST OF TABLES

1 Test Manager Overview

The Test Manager application manages various aspects of a high level “test” process, including:

- Setting operating points for closed loop controllers
- Starting and stopping data acquisition processes
- Communicating with external devices such as engine ECUs
- Controlling specialized measurement devices or communicating with support applications that can off-load parallel operations such as determining stability of variables
- Performing continuous computations of user supplied expression or testing for limit violations

The Test Manager has the application name `gp_test` which stands for “general purpose test”. The “tests” are designed by a user through a language written in editable text files called “procedure files”. The language is unique to CyFlex and to the `gp_test` application. The language consists of a set of keywords that define various test functions. Each keyword is followed by one or more lines of specification for the testing function.

Refer to:

- *Section 4 Procedure Files* on page 10
- cyflex.com usage help for `gp_test`

2 Test Manager Concepts and Terminology

This section introduces some basic concepts and terminology used in this manual. Detailed descriptions are included in subsequent sections.

2.1 Procedure

A text file that contains the instructions for a test or portion of the test is referred to as a "procedure". When a test is started, the user issues a command to start a particular procedure by referring to the file name. This initial procedure is referred to as the "root procedure". It may reference or jump to other procedures, which would be referred to as sub-procedures.

2.2 Mode

A procedure file contains sections that are grouped as "modes". Each mode describes a particular state of the testing process. A test is conducted by sequencing through various modes.

2.3 Keyword

The characteristics of a mode are specified with "keywords". Keywords are special text descriptions of an action or process to be executed in a mode. The keywords always begin with the @ symbol and are followed by an upper-case text string without any blank spaces.

A keyword will be the only field on a line but will be followed by one or more lines to specify the characteristics of the action (keyword specifications). An example is the keyword which defines the start of a test mode and the end of the previous mode specification.

```
@MODE
```

Note:

With the exception of `@MODE`, all keywords are optional

2.4 Instance

More than one copy of Test Manager may be running at any time in a system, with each one performing different functions. Each copy of the Test Manager is referred to as an "instance" of the manager. Each instance must have a unique name. Each instance is launched with a reference to an "instance definition file" than contains the name and various characteristics that will be unique to the instance.

2.5 Events

Events are signaling mechanisms which are transmitted from one process to another in the system. The Test Manager relies on various events to know when to transition from one mode to another or to determine which mode or procedure should be executed. Refer to *CyFlex Events* for details.

2.6 Mode Phases

Each test mode has timing phases that are described by special keys. These keys are used to specify at which phase an action should occur. In the procedure files the specification of the phase is often referred to as the `start_code`.

Table 1: Timing Phase Keys

Timing Key	Descriptions
`AT_START`	The action should be executed at the beginning of a mode
`AT_END`	The action should be executed at the end of a mode, immediately before transitioning to the next mode or procedure.
`AFTER_STABILITY`	This defines a time while in a particular mode when stabilization has been achieved. There are optional keywords to define what the requirements are for stability. When those requirements are met, then the action is executed.
`AT_NT`	This code is rarely used but can specify to kill a process previously launched if a new test command is received by the `nt` command. .

2.7 Support Services

The Test Manager relies on many other processes to manage a test sequence. These will be referred to as **support** services. Some of the services, such as the PID control process, are standard services that are always present and other services will be launched by the Test Manager specifically to cooperate with the manager to perform the test sequence.

2.8 Test Commands

Various functions of the manager can be invoked by the user by typing commands or having them issued through scripts or other processes. For example, starting a procedure is invoked by the `nt` (new test) command which is followed by the file name of the root procedure.

Refer to *Section 3 Executing Test Manager Commands* on page 7 for details.

2.9 Value as “Constant/Label/Expression”

Many fields in a procedure file specify a “value” to be used by the Test Manager. The “value” might be an integer, a floating-point number, a logical (on/off) value, or a text string. The instructions for a particular keyword may indicate that the value can be expressed as a constant, variable label, or a computed expression. For each of these three forms, the evaluation of the value is always handled as if it were a computed expression. A constant or variable label are just simple forms of a value that can be part of a computed expression.

When the value field supports computed expressions, a constant value must always be expressed with an attached unit’s designation. Examples:

```
Constant     --    100[kpa]
Label        --    x_pressure
Expression   --    “ my_setpoint + 10[in-hg] “
                   “ '/data/logs.' + @year_month_day()  “
```

Note:

Expressions are always enclosed by the double quote symbol.

2.10 Path

Many keywords will have fields that define an execution path for the next test mode. The path may be for a successful completion (SUCCESS_PATH) of an external process by a support service, or a failure (FAILURE_PATH) of that process. There are several ways to specify a "path".

Mode Number	The mode number of a test mode within the same procedure file.
Procedure Filename	The complete path name of a procedure file
Combination of Above	A procedure file name followed by the starting mode in that file
NEXT_MODE	Use the DEFAULT_NEXT_MODE" of the @MODE keyword as the path
MODE_TERMINATE	same as NEXT_MODE
TEST_DONE	Terminate the test and wait for a user command (no path taken)
ELSE_MODE	Use the path defined by the @ELSE_MODE keyword
RETURN	Return to the calling procedure (to be used only in sub-procedures)
NONE	Stay in the current mode

2.11 Conditional Tests

Four keywords determine whether or not the normal actions of a mode are to be executed:

- @IF_TRUE
- @IF_FALSE
- @SWITCH
- @PATH_OPTIONS

@Note:

The @PATH_OPTIONS keyword is available in CyFlex.6.3.26 and later versions.

These keywords are special in that they are always evaluated first when a mode execution begins since they are used to determine whether the other keywords which appear in the mode are actually to be executed. They are evaluated in the sequence shown in the above list. They are evaluated first regardless of their position in the procedure file relative to other keywords for that mode. For example:

```
@IF_TRUE
" test_running && data_good "
```

Where "test_running" and "data_good" are logical variables that are either true or false.

The preceding specification would cause the immediate exit to the DEFAULT_NEXT_MODE path if either "test_running" or "data_good" were FALSE. Some additional control of the execution path can be provided with the @ELSE_MODE keyword:

```
@ELSE_MODE
/specs/gp/not_running
```

If both keywords appeared in the same mode and the test failed, then instead of jumping to the DEFAULT_NEXT_MODE, the Test Manager would jump to the /specs/gp/not_running procedure.

2.12 Sequence of Execution

The Test Manager (gp_test) uses a unique language for specifying actions within a test mode for setting control targets, requesting fuel readings, setting variable values, and so on. The gp_test language is not a scripting language where the sequence of operation is based on the sequence in which the operations are specified. Instead, there is a hard-coded sequence of operation. The following describes the sequence of actions when a test mode is started. This sequence is unrelated to the order in which the corresponding keywords are entered in the gp_test procedure file.

1. Perform conditional tests to see if the mode is to be skipped. The conditional tests are @IF_TRUE, @IF_FALSE, @SWITCH,and @PATH_OPTIONS. The conditional the tests are evaluated in the order stated in *Section 2.11* on page 4. If the evaluations of these tests indicate that a new mode or test procedure should be started, then the mode is terminated and none of the other actions are executed except those which have a start_code of AT_END.
2. Any @PARAMETERS or @SET_DISPLAY_STATUS values will be set.
3. Any control loop operations are executed:

@SPEED, @TORQUE	@DYNO	@THROTTLE
@USER_LOOP	@PID_GAINS	@ENGINE_CONTROL_MODE
@FEED_FORWARD	@GET_PID_GAINS	@GET_FF_GAIN
@CONTROL_TOLERANCE		

4. Send any events specified by @SET_EVENTS.
5. Configure any limits specifications specified by @LIMIT_SPECS.
6. Configure any ramping operations specified by @RAMP or @RAMP_DYNAMIC.
7. Configure fuel reading operations specified by @FUEL_READING, @PAM_DATAPOINT, @FUEL_READING_SYNC, and @FUEL_READING_STATS.
8. Communicate with any applications launched by @AUXILIARY_TASK or @TEST_TABLE.
9. Configure any stability operations specified by @STABILITY_SPECS.
10. Perform communications specified by @ASC.
11. Perform any file writing operations specified by @WRITE_VALUES.
12. Launch background scripts specified by @BACKGROUND and @BACKGROUND_TASKS.
13. Perform AK and DEVCOM communications.
14. Perform CYBER related configuration commands.
15. Send any specified emails specified by @EMAIL.

16. Perform UNICO related communications with the dyno controller.
17. Configure operations specified for multi-filter changers (@MFC_CMD).
18. Configure operations specified by @CHAIN_EVENTS (CyFlex.6.3.0 and later).
19. Configure state machine operations specified by @STATE_MON_ACTIONS and related STATE_MON keywords.

As an example, if you use @AUXILIARY_TASK to read a value from a file, specify the index in the file (often using the count integer variable). In the gp_test procedure file you could do the following:

```
#########################################################
@AUXILIARY_TASK
AT_START                                          MODE_TERMINATE
                          99
/cyflex/bin/vrbl_to_file   “specs/gp/myfile READ count”
@PARAMETERS
AT_START                              count                  100
```

This would cause vrbl_to_file to read index 100 from “myfile”, because @PARAMETERS are executed BEFORE @AUXILIARY_TASK operations. Therefore, it does not matter where the @PARAMETERS keyword is positioned within the mode, that operation happens before the value of the ‘count’ variable is sent to vrbl_to_file.

2.13 Trace File

The Test Manager will produce a log of the sequence of test modes to a disk file for diagnostic purposes. The trace includes a time stamp, mode number, the procedure file name, mode number, and a short description of what caused the mode to terminate. Refer to *Section 8 Instance Definition File* on page 16 for how to specify where to write the file and how to make it active.

3 Executing Test Manager Commands

Use the following commands to communicate with the Test Manager.

`nt`	Start a new test
`adv`	Advance to next mode
`idle`	Issue the `idle_mode` event
`hold`	Hold test in this mode
`release`	Release the previous hold
`gp_stop`	Issue the `stop_test` event
`suspend`	Suspend the test in this mode
`ck`	Check a test procedure file

Refer to the following sub-sections for descriptions of these commands.

3.1 Start New Test

Use the `nt` command to instruct `gp_test` to read and start a new procedure file. If the `+r` option is specified, the test procedure execution is modified without reading the procedure file. The `+r` option may only be used to change the execution path within the currently executing procedure or within the root procedure. Causing the test to jump into a sub-procedure which is not currently executing is disallowed.

Refer to cyflex.com usage help for `nt`.

Refer to the following for supplemental information:

- *Section 4 Procedure Files* on page 10
- *Section 5 Test Modes and Keywords* on page 12
- *Section 6 Timing, Events, and Mode Termination* on page 13

3.2 Advance

Use the `adv` command to advance to the next gp_test mode. Refer to cyflex.com usage help for `adv`.

Note:

Some uses of the `adv` command are advancing the test to bypass the specified time or to get around stability criteria that have been specified but that have not been met. The command is particularly useful during the initial checkout of a new test procedure.

Refer to *Section 8 Instance Definition File* on page 16 for supplemental information.

3.3 Idle

The `idle` command sends the `idle_mode` event to all instances of the Test Manager. Only the instance which has a matching instance name will respond to this command. The default name is `test`.

If the currently executing test procedure has a @REGISTERED_EVENTS specification for this event, then the path specified in the test procedure will be taken. If not, and the current instance of the Test Manager has an @UNIVERSAL_REGISTERED_EVENTS specification in the instance definition file for this event, then the path specified in the file will be taken. This is normally used to cause the test to jump to a test mode which sends the test object to a safe operating condition and stays in that mode indefinitely, but it is optional to specify the action that results. Only one instance of the Test Manager, the one with the matching name, will be able to act on this event.

Refer to the following for supplemental information:

- cyflex.com usage help for idle
- *Section 8 Instance Definition File* on page 16

3.4 Hold

The hold command holds the Test Manager (gp_test) in the current mode. The mode timer continues to run but will not cause the mode to terminate until the hold is released. The command sends the hold_mode message event to all instances of the Test Manager. Only the instance with a matching name will process the event. Refer to cyflex.com usage help for hold.

Note:

The common use of the hold command is to hold the test to maintain the current operating condition. The command is particularly useful during the initial checkout of a new test procedure. Use the suspend command to hold the mode and prevent the timer from running.

Refer to *Section 8 Instance Definition File* on page 16 for supplemental information.

3.5 Release

The release command releases gp_test from a hold of the current mode. The command sends the release_mode message event to all instances of the Test Manager. Only the instance with a matching name will process the event. Refer to cyflex.com usage help for release.

Note:

Issuing the release command is not meaningful unless the hold command has previously been issued. The release command will result in mode advancement only if the conditions specified in the mode for mode advancement have otherwise been satisfied.

Refer to *Section 8 Instance Definition File* on page 16 for supplemental information.

3.6 Stop Test

The gp_stop command sends a message event named stop_test. The message contains the instance name specified in the command. The default name is test.

If the currently executing test procedure has a @REGISTERED_EVENTS specification for this event, then the path specified in the test procedure will be taken. If not, and the current instance of the Test Manager has an @UNIVERSAL_REGISTERED_EVENTS specification in the instance definition file for this event, then the path specified in the file will be taken. This is

normally used to execute the test procedure, `/spec/gp/gp_shutdown`, but it is optional to specify the action that results. Note that only one instance of the Test Manager, the one with the matching name, will be able to act on this event. Refer to cyflex.com usage help for `gp_stop`.

Refer to *Section 8 Instance Definition File* on page 16 for supplemental information.

3.7 Suspend

The `suspend` command suspends the current test mode. Use this command when the test is to be temporarily suspended and the time spent while suspended is not to be counted as part of the mode time. The command sets the `suspend_mode` message event which contains the `instance_name` argument. Refer to cyflex.com usage help for `suspend`.

Refer to *Section 8 Instance Definition File* on page 16 for supplemental information.

3.8 Check

The `ck` command reads a specified test procedure file checks for proper syntax. A new copy of the file is made with the same filename except an X is appended. For example, a new filename would be `/ram/gp_hotshutX`. If there are no errors, this new file contains the same information as the new file, but with cleaner formatting. Refer to cyflex.com usage help for `ck`.

Refer to *Section 4 Procedure Files* on page 10 for supplemental information.

4 Procedure Files

A test procedure is defined by a text file, called a procedure file which contains a description of a set of operating points called test modes. The procedure file is organized into blocks of data which define these test modes.

There may be up to 999 test modes in each procedure file. Mode 0 is not used.

Up to 127 procedure files may be linked together to form a complex test sequence or the test may consist of just one test procedure. Several test procedures may be linked sequentially or one procedure may call another and the sub-procedure may call other sub-procedures. A method is supplied to allow a sub-procedure to return control to the calling procedure when the sub-procedure is complete.

4.1 Example Test Procedure

The following is a simple example of a test procedure file.

```
##----------procedure file global definition section ----------------------##
#start_mode (when this procedure is started, the start_mode will be executed
#first
1
#limit this procedure to be used only by the instance of gp_test named “test”
@INSTANCE
test
#create some variables to be used in this procedure
@CREATE_VAR
      #label      type        units       initial_value display_resolution
      cycle_count INT         none        0[none]
      test_name   STRING      -           'no name'
##-------------------start of mode specifications----------------------##
@MODE
      #mode timeout     default_next_mode
      1     1[sec]      2
      #text description of this mode's purpose
      Set some initial values
@PARAMETERS
      #start_code label                   value
      AT_START    test_name               “ 'example_test' + cycle_count “

@MODE
      #mode timeout     default_next_mode
      2     1[sec]      3
      #text description of this mode's purpose
      Write the current test name to a file
@WRITE_VALUES
      #start_code file_name         value             C-format string
      AT_START    /data/example     test_name         “name=%s\n”

@MODE
      #mode timeout     default_next_mode
      3     0[sec]      4
      #text description of this mode's purpose
      Set the coolant temp, increment counter and wait for loop to reach 150
@USER_LOOP
      #open/closed variable   start_target      [end_target]      [ramp_rate]
```

```
      CLOSED_LOOP  coolant_t  my_temp
@LIMIT_SPECS
      #variable   value       upper/lower rate  period      path
      my_temp     150[f]      U           SLO   0[s]        MODE_TERMINATE
@PARAMETERS
      #start_code label                   value
      AT_START    cycle_count             “cyflex_count + 1[none]”

@MODE
      #mode timeout     default_next_mode
      4     0[sec]      2
      #text description of this mode's purpose
      call the sub-procedure /specs/gp/do_nothing – upon return go to mode 2
@PROCEDURE
      #pathname
      /specs/gp/do_nothing
```

5 Test Modes and Keywords

The test mode specification is a method for describing how the test object is to be operated and what data should be acquired.

There are a small number of items which must be specified for each mode and many optional features. The features could be called actions, operations, or specifications, and each feature is defined in the data file by a keyword. The keyword is a special character string which informs the Test Manager the data to follow pertains to a particular operation until a new keyword is encountered. Each keyword:

1. Begins with the @ character
2. Is followed by a string which must not contain any blanks
3. Is the only information on that line
4. Consists of all upper-case characters
5. Must be spelled correctly to be recognized

The first keyword for each test mode is always `@MODE`. This defines the start of a mode specification and the end of the previous mode specification. The `@MODE` keyword defines the mode number, the maximum time to spend in the mode, which path to execute next, and provides a line to describe the purpose of the test mode. The mode number must be a number from 1 to 999.

The `@MODE` keyword is the only required keyword for a mode specification.

```
@MODE
  #MODE_NUMBER   MAX_TIME        DEFAULT_NEXT_MODE
    1                   10[sec]              2
  #description
  This is mode number one.
```

The test mode `description` line is written to a CyFlex string variable, such as `TEST_DESC` so that it can be displayed to the operator. Each instance of the Test Manager should use different variables for this description. The description variable is specified in the instance definition file. Refer to *Section 8 Instance Definition File* on page 16.

The mode numbers are tags which do not imply any order or sequence. Thus, the modes can be numbered in any order and control may be passed from one mode to another mode in any sequence. It is recommended, however, that the user maintain an orderly numbering and sequencing so the file may be more easily read and interpreted.

Notes:

Any line which begins with the # symbol is treated as a user comment and is ignored by the Test Manager when reading the file. Comment lines such as those shown above are usually carried along to provide reference as to the meaning of the fields on the line below the comment.

The # symbol only designates a comment if it is the first character on the line. It can be in any column but must be the first non-blank character. A line such as the following could cause an error.

```
     1          10[sec]                2   #this is the first mode
```

6 Timing, Events, and Mode Termination

The time duration of the test mode may be precisely specified or it may be determined by the operations taking place. For instance, a test mode may consist of taking a data acquisition process which, when complete, terminates the mode. The exact duration this requires is not known ahead of time, but the specification may be written in such a way that the mode is terminated as soon as the cooperating process is complete.

The internal mechanism which CyFlex uses to control and synchronize processes such as these is called an event. Each event is a signal which passes from one part of the software to another to indicate when something happens. In some cases, data may be included in the event to provide more detail about what occurred. This type of event is called a message event.

With one exception, the mechanism for terminating a test mode is for the Test Manager to receive an event. The Test Manager will listen for various events which are used to terminate a mode. There is a timer event which is used to keep track of the maximum mode time. The manager also attaches to certain system events such as that generated by a user command to advance to the next mode. Other events are automatically created to signal the completion of data acquisition processes, violation of a limit, reaching required stability, and so on.

The exception mentioned above is for a mode with a `MAX_TIME` specified as a negative value. This is used to indicate that the mode is to be executed as fast as possible and then terminate. This is called an “immediate” mode. A zero value for the `MAX_TIME` indicates that there is no timer for the mode and the mode must be terminated either by the completion of a cooperating process or by a user command such as “adv” (advance to next mode).

There are keywords which allow the user to designate that the mode terminates when a specified event or group of events is received and to set certain events at the beginning of any test mode. These are used to synchronize with external processes.

When an event is received by the Test Manager, the manager must first determine if the mode should be terminated. If the mode is to be terminated, the manager will then determine which mode or test procedure is executed next. This is referred to as a “path”. This will depend on which event was received and other factors, such as loop counters.

The following is a list of different circumstances which can cause a test mode to terminate and proceed to the next mode in the sequence or to a user-specified path (mode or procedure).

1. A negative time-out for the mode
2. Completion of the specified time-out for the mode
3. Completion of a data acquisition or support task function with `MODE_TERMINATE` specified for a `success_path` or failure path
4. Completion of a stabilization requirement
5. Completion of an auxiliary procedure
6. Violation of a limit specification
7. Receiving any of a list of events specified with the `@TERMINATION_EVENTS` keyword
8. Receiving all of the list of events specified with the `@TERM_ALL_EVENTS` keyword
9. Receiving an event specified with the `@GLOBAL_EVENTS` keyword
10. Receiving an event specified with the `@REGISTERED_EVENTS` keyword
11. Receiving an event specified in the instance definition file as `@UNIVERSAL_EVENTS`

12. Receiving an event specified in the instance definition file as `@UNIVERSAL_REGISTERED_EVENTS`
13. Receiving an event from the `adv`, `idle`, `release`, or `gp_stop` commands
14. Receiving an event from the command `nt` to start a new test
15. Receiving an event from the command `nt` to alter the path of the current test

7 Multiple Instances

More than one copy of the Test Manager may be used simultaneously in the same test system. Each instance (copy) can be performing a different function and operating on separate procedure files. Normally, there should only be one instance of the Test Manager managing a particular system or test fixture. The separate instances may be totally independent or may have some relationship and be communicating or synchronizing by events. For example, one instance may be managing the primary test object and the other may be managing the HVAC system of the test cell and would be completely independent. In another example, one instance may be managing the test object while another manages the operation of a data sampling system. The test object manager would signal the data sampler when it needed data and the sampler would respond when the data was available.

It is possible to designate a root procedure to be run only by a particular instance by placing the `@INSTANCE` keyword in the procedure file with the name of the instance which will be allowed to run the procedure.

```
@INSTANCE
  test
```

The `@INSTANCE` keyword must appear in the global definition section of a procedure file after the `start_mode` and before the first @MODE keyword.

It is recommended that all root procedures use the `@INSTANCE` keyword.

8 Instance Definition File

Each instance of the Test Manager requires a definition file to define some parameters which apply to that instance of the Test Manager, regardless of what procedure or mode is being executed. The contents of the file contain information that should be unique for each instance, so there should be a separate definition file for each instance. When the Test Manager is started, it is spawned with an argument that is the pathname of the instance definition file:

```
gp_test <instance_definition_file> &
```

The default file name is `/specs/gp/gp_header`, so if no argument is specified, then `/specs/gp/gp_header` will be used.

The instance definition file contains the following specifications:

`@INSTANCE`	Instance name
`@PRIORITY`	Priority of `gp_test`
`@MODE_NUMBER_LABEL`	Variable label to display the current mode number
`@PROCEDURE_NUMBER_LABEL`	Variable labels to display the current test number and Procedure filename
`@MODE_COUNTDOWN`	Variable label to display the remaining time in mode
`@MODE_DESCRIPTION_STRING_ARRAY`	Up to 4 string variable labels to display mode progression
`@STABILITY_DESCRIPTION`	Variable label to display the current state of stability
`@TERM_ALL_EVENTS_LIST`	Labels to display events received and waiting for the `@TERM_ALL_EVENTS` keyword
`@LIMIT_SPECS_ALL_STATUS`	Labels to display limit specs waiting and received for the `@LIMIT_SPECS_ALL` keyword
`@TRACE_FILENAME`	A filename for writing the trace information
`@TRACE_ENABLE`	Label of a variable to enable tracing and the initial state
`@REREAD`	Flag to force a re-read of all spec files when an `nt` command is issued
`@ERROR_STRICT_ENFORCEMENT`	Flag to enforce clean startup of a test (no errors)
`@UNIVERSAL_EVENTS`	See following example
`@UNIVERSAL_REGISTERED_EVENTS`	See following example

```
############  Example gp_header file  ###################################

# This is an `Instance Definition File`, sometimes referred to as a
# gp_test header file.  This file is used to define certain variables and
# features of a particular instance of the `Test Manager` (gp_test).
# Each instance of gp_test that is launched must have a unique name,
# and that is defined below by the @INSTANCE keyword.  @INSTANCE
# is the only keyword that is required in this file and it should be
# at the top of the file.  All other keywords are optional and may be
# entered in any order.  If a keyword is not used, a default value
# will be inserted.
# Variable labels do not need to exist or be created by another
# application. The variables will be created by gp_test if they
# do not exist.
#
# The `Instance Definition File` is specified as an argument when
# launching 'gp_test'.
#
#    gp_test <filename> &
#
# The default filename is '/specs/gp/gp_header', but each instance
# of gp_test must have a unique Instance definition file
# Type 'use gp_test' for additional help on this command.
#########################################################################
# Supply a unique name for this instance - up to 31 characters
@INSTANCE
   test

# Priority for this instance - default is 16
@PRIORITY
   16


# Supply a label for an integer variable that will be used
# to display the current test mode number
# The default label will be test_mode_<instance name>

@MODE_NUMBER_LABEL
   test_mode


# supply labels for integer and string variables which
# will be used to display the procedure number and name
# of the current test procedure.
# The default labels will be test_num_<instance_name>
# and test_name.<instance_name>

@PROCEDURE_NUMBER_LABEL
   test_num   test_name_test


# Supply a label for the 'countdown' variable which will
# be used to display the time remained until a mode times out.
# The default label will be countdown_<instance_name>
```

```
@MODE_COUNTDOWN
   countdown


# Supply up to 4 string variable labels that will be used
# to display the mode descriptions of the current test mode
# and the previous 3 modes # Default labels are:
#   TEST_DESC_<instance_name>
#   PREV_MODE1_<instance_name>
#   PREV_MODE2_<instance_name>
#   PREV_MODE3_<instance_name>

@MODE_DESCRIPTION_STRING_ARRAY
   TEST_DESC   PREV_MODE1   PREV_MODE2   PREV_MODE3


# Supply the label of a string variable that will be used
# to display the current state of the stability specification
# if the @STABILITY_SPECS keyword was used in the current mode
# The default label will be STABILITY_<instance_name>

@STABILITY_DESCRIPTION
   STABILITY1

# Supply the labels of two string variables that will be used
# to display the event names associated with the @TERM_ALL_EVENTS
# keyword if the current mode contains that keyword.
# The first string will contain the event_names of input events
# that haven`t yet been received.  The second label will contain
# the event name of the output event that has been set if all
# the input events have been received
# The default labels will be TrmEV_waiting_<instance_name>
# and TrmEV_done_<instance_name>

@TERM_ALL_EVENTS_LIST
   TrmEV_waiting   TrmEV_done

# Supply the labels of two string variables that will be used
# to display information associated with the @LIMIT_SPECS_ALL
# keyword if the current mode contains that keyword.
# The first string will contain the names of unsatisfied limits
# that haven`t yet been received.  The second label will contain
# the event name of the output event that has been set if all
# the input events have been received
# The default labels will be TrmLIM_waiting_<instance_name>
# and TrmLIM_done_<instance_name>
```

```
@LIMIT_SPECS_ALL_STATUS
   TrmLIM_waiting   TrmLIM_done

# Supply the label for a string variable which will be the
# the file pathname of the trace output for this instance and the
# label of an integer variable that will define the maximum
# number of trace entries for the file.  When the number of
# entries reaches this maximum, the file will be renamed by
# appending the instance name and a new trace file will be
# opened.
# The defaults are `/specs/gp/TRACE_<instance_name>` and 1000

@TRACE_FILENAME
   /specs/gp/TRACE   1000

# Supply the label of a LOGICAL variable that can be used
# to enable or disable the gp_test trace output and the
# initial value of the variable (default is ON).  The value
# should be expressed as ON or OFF.  The default label will
# be `gp_trace_<instance_name>`

@TRACE_ENABLE
   gp_trace   ON


# The @REREAD keyword is itself a flag.  If present in
# this file, it specifies that all procedures must be
# re-read whenever an `nt` command is issued.  If not,
# files specifications can be re-used if they were
# read previously and have not been modified on disk
# since read.  This speeds up the startup of a test if
# any of the sub-procedures are re-used.
# Simply comment out this keyword for normal operation
#@REREAD



# Inserting the  @ERROR_STRICT_ENFORCEMENT keyword
# will mean that a gp_test procedure cannot be started
# if there are errors detected when reading any procedure file.

@ERROR_STRICT_ENFORCEMENT
```

```
# The @UNIVERSAL_EVENTS keyword is used to direct control of
# the execution of gp_test to a test procedure when the specified
# event is received, regardless of which test procedure is currently
# being executed.  However, if the same input event is specified as
# a @GLOBAL_EVENTS in the currently executing procedure, the path
# specified in @GLOBAL_EVENTS will take precedence.
# You may specify up to 128 event/procedure lines in this file.
# Note that there are no default specifications for this keyword,
# although the usual specifications would be to provide a response
# for the `abort_limit` and `emergency` events in the case of the
# gp_test instance which is managing the engine itself.
# The gp_test instances which handle data acquisition systems,
# or auxiliary test cell systems should not have entries which
# manage engine shutdowns directly

@UNIVERSAL_EVENTS
#   event_name            procedure
   emergency              /specs/gp/gp_emergency
   abort_limit            /specs/gp/gp_shutdown


# Registered events are special event messages which contain the
# name of a particular instance of gp_test for which the message
# is intended.  There are a small number of such events and they

# listed below along with the user command which will send these
# events:
#     event name        user command
#     idle_mode         idle
#
# The @UNIVERSAL_REGISTERED_EVENTS keyword is used to direct control of
# the execution of gp_test to a test procedure when the specified
# event is received, regardless of which test procedure is currently
# being executed.  However, if the same input event is specified as
# a @REGISTERED_EVENTS in the currently executing procedure, the path
# specified in @REGISTERED_EVENTS will take precedence.
# You may specifiy up to 128 event/procedure lines in this file.
# Note that there are no default specifications for this keyword.
# The gp_test instances which handle data acquisition systems,
# or auxiliary test cell systems should not have entries which
# manage engine shutdowns directly

@UNIVERSAL_REGISTERED_EVENTS
#   event_name                procedure
   stop_test                  /specs/gp/gp_shutdown
   idle_mode                 /specs/gp/gp_idle
```

9 Event Domains, Universal Events, Global Events and Registered Events

9.1 Event Domains

The Test Manager recognizes three domains of influence for incoming events:

1. Mode domain:

   Events which apply only to the currently active test mode

   These domains are somewhat hierarchical. There can be several instances of `gp_test` running at a test cell, each instance can be handling many procedures, and each procedure can have many modes. At any time, each instance of the Test Manager can only be executing a single mode of a single procedure, but multiple instances can be running simultaneously.

   Methods for modifying the execution path of a test can be associated with each of these domains. Shown below are examples of keywords which can be used to modify the execution path listed by their corresponding event domains.

2. Procedure domain:

   Events which apply to the currently active test procedure regardless of what mode is active

3. Instance domain:

   Events which apply to an instance of the Test Manager regardless of what mode or procedure is active

9.1.1 Mode Domain

```
@TERMINATION_EVENTS
@TERM_ALL_EVENTS
Events associated with @LIMIT_SPECS
Events associated with @STABILITY_ACTION
Events associated with support processes
@FUEL_READING
@FUEL_READING_SYNC
@AUXILIARY_TASK
etc.
```

9.1.2 Procedure Domain

```
@GLOBAL_EVENTS
@REGISTERED_EVENTS
```

Note:

`REGISTERED_EVENTS` apply only to a single instance, but their domain is further limited to the procedure in which they are specified

9.1.3 Instance Domain

@UNIVERSAL_EVENTS
@UNIVERSAL_REGISTERED_EVENTS

Hard-coded "registered" events

STOP_MODE	Issued by the gp_stop command
IDLE_MODE	Issued by the idle command

The @UNIVERSAL_EVENTS and @UNIVERSAL_REGISTERED_EVENTS apply regardless of what mode or procedure is active. They must be specified in the instance definition file.

If the same event is specified for more than one domain, the order of precedence is as follows:

1. Procedure domain
2. Instance domain
3. Mode domain

Thus, if the emergency event were specified in @GLOBAL_EVENTS, in @UNIVERSAL_EVENTS, and also in @TERMINATION_EVENTS, the specification in @GLOBAL_EVENTS would be the one which would determine the execution path, since its precedence is higher.

The practical implications are:

- When an instance of gp_test is started, the universal events are activated. Prior to the issuance of the nt command, gp_test will respond to emergency or abort events with a user-specified procedure.
- Actions for events specified in a gp_test procedure override what is specified as the default for the instance of the gp_test in which it runs.
- If the emergency and abort processes are always the same for all procedures, they only need to be specified in the instance definition file (e.g., /specs/gp_header) and not in each procedure file.

Emergency or shutdown procedures can be re-initiated even when the procedure is still active.

9.2 Universal Events

The @UNIVERSAL_EVENTS specification is placed only in the instance definition file for an instance of the Test Manager. It is a list of event/path pairs that define which test procedure to jump into when the event is received.

```
@UNIVERSAL_EVENTS
# event_name next_mode   test_procedure/pathname
Emergency        -       /specs/gp/gp_emergency
```

9.3 Universal Registered Events

The universal registered events act the same as universal events, except the event must be one of a special kind of event intended for a specific instance of the Test Manager in a system which is running multiple instances. This allows command of a particular procedure to change course when there are multiple copies of the Test Manager running. These events are always generated by an operator command such as `gp_stop` or `idle`. Each of these commands has an optional argument which is the name of the instance which must respond to the command. The default name is `test`. Refer to *Section 3 Executing Test Manager Commands* on page 7 for further definition of test commands which use the instance name.

It is recommended that something similar to following be included at the top of the instance definition file. Caution is required when following this example. The shutdown and emergency procedure names may vary from cell to cell and facility to facility.

```
@UNIVERSAL_EVENTS
# event_name   next_mode     test_procedure/pathname
emergency          -         /specs/gp/gp_emergency
abort_limit        -         /specs/gp/gp_shutdown
@UNIVERSAL_REGISTERED_EVENTS
stop_test                    /specs/gp/gp_shutdown
```

9.4 Global Events

The first section of a test procedure specification file contains the mode number where the test normally begins and some optional keywords unique for the procedure specified in the file. These keywords are `@GLOBAL_EVENTS` and `@REGISTERED_EVENTS`. These keywords may appear only once in the procedure file and must appear before the first `@MODE` keyword. The following describes their use.

These keywords are used to define events to which the Test Manager will respond during any mode of the test procedure, i.e., in the procedure domain. The specification allows the execution to jump to any mode in the test procedure, to another test procedure, or return to a calling procedure. This allows the user to build procedure files to handle exceptional events, such as one providing system shutdown. If the execution jumps to another procedure, then the global and registered events for that new procedure will start applying

Example:

```
@GLOBAL_EVENTS
#event_name    next_mode     test_procedure
emergency          -         /specs/gp/gp_emergency
abort_limit       45         -
```

9.5 Registered Events

This section is used to define special events the Test Manager receives from certain user commands. These events are called registered events because they contain the name by which various copies of the Test Manager are registered. This is required for situations where a single computer may be controlling more than one test fixture and where there is more than one Test Manager running. The message event the user command sends will contain the name of the version of the Test Manager for which the command is intended. These commands usually default to the name `test` which is the name for the standard version of the Test Manager. Commands such as `gp_stop` and `idle` are commands that transmit the instance name through the events `stop_test` and `idle_mode`, respectively. Specifying these event names with the `@REGISTERED_EVENTS` keyword, means the Test Manager will recognize those events in the same way as the `@GLOBAL_EVENTS` except the registered name in the event must match the instance name of this version of the Test Manager.

Example:

```
@REGISTERED_EVENTS
#event_name    next_mode     test_procedure
stop_test          -         /specs/gp/gp_cooldown
idle_mode         93
```

For both `@GLOBAL_EVENTS` and `@REGISTERED` events:

- If neither the `next_mode` nor the `test_procedure` fields are specified, then the mode is just terminated and the normal path is taken.
- If the `next_mode` field is specified, but the `test_procedure` is not, then the test will jump to that mode in the current test procedure.
- If the `test_procedure` field is specified, but the `next_mode` is not, then the test will jump to that sub-procedure. If the sub-procedure exits via a `RETURN`, then the current procedure will begin executing at the `default_next_mode` specified for the mode interrupted. Any actions in the interrupted mode which were specified to be executed `AT_END` will have been executed before the sub-procedure was called so the process immediately jumps to the `default_next_mode`.
- If both `next_mode` and `test_procedure` are specified, it will jump to the `next_mode` upon returning from the sub-procedure.

  Note that when any test mode is interrupted, actions specified with the "`start_code`" `AT_END` are executed before moving to the next procedure or mode.
- If the `test_procedure` file does not exist or cannot be opened, then the `next_mode` will be executed.

10 Looping and Branching

Every test mode has specifications for which mode is to be executed when the current mode is complete. This may be a simple unconditional specification to always proceed to the next mode or it can be a more complicated conditional specification based on loop counters, tests of logical variables, or dependent upon what causes the mode to terminate.

A mode may be executed once or repetitively. If a test consists of performing a single operation periodically, it can be accomplished by a single mode specification of the required duration which has the `default_next_mode` specified with its own mode number.

A set of test modes may be executed once or repetitively. This is called looping. There may be several loops in a test procedure and they may be nested.

The Test Manager provides two mechanisms which allow the user to keep track of the loop execution counters. One mechanism is automatic and produces a rather cryptic displayable string which shows the state of loop counters in the form, `a/b:c/d` where ‘a’ represents the number of executions of the inner-most loop which is specified to execute ‘b’ times and ‘c’ represents the number of executions of the outer-most loop which is specified to execute ‘d’ times.

This format gets extended to however many nested loops have been entered. This display of the loops is tacked onto the normal mode description string that is created by the user as part of the `@MODE` keyword specification.

The second mechanism allows the user to create an integer variable which will contain the current execution count of a loop and can be displayed, logged, or be operated on like any other CyFlex variable. Use the `@LOOP_CONTROL` keyword to specify looping. The keyword must be placed in the terminating mode of the loop. There may be only one loop terminating at a particular mode. An example loop specification is shown below:

```
@LOOP_CONTROL
# number_of_repeatsloop_start_mode   loop_counter_variable
10                        1          loopx
```

The preceding specification means to branch to mode `1` after this mode is complete and do it `10` times before proceeding to the default`_next_mode` specified with the `@MODE` keyword.

A special form of loop counting is supported in a way which is required for long term testing where the procedure may be restarted, the computer might be re-booted or even have a new CyFlex revision installed during the course of the test. The difference between a `test_cycle` and a normal loop is the value of the `cycle_counter` is automatically maintained on disk so it can be restored if the test or computer is re-started (a `go`). The test cycle counter value must be manually reset to zero (“`svar test_cycles 0`”) by the operator if the test is to be restarted. Normal loop counters are automatically reset to zero each time a test is started. The keyword `@TEST_CYCLE_END` is placed in the terminating mode of a loop. Every time that mode is completed, the cycle counter variable gets incremented. An example of the specification is shown below:

```
@TEST_CYCLE_END
#maximum numbercycle_counter  test_complete_path
10000          test_cycles   54
```

A mode may have a set of "conditional" tests which must be met for the mode to be executed. Failure to pass the tests will cause control to pass immediately to the next mode or to a special alternate path called the `else_mode` which can be specified by the @ELSE_MODE keyword. This alternate path can be taken as a result of several special occurrences associated with failures of auxiliary processes, completion of a stability requirement, or failure of the conditional tests. The alternate path specified by @ELSE_MODE may be either a mode number, a test procedure filename, or a RETURN to the calling procedure.

One test procedure may cause another procedure to be executed as if the second procedure were a mode of the calling procedure. This method can be used to chain tests together or to build a structured method for performance mapping. As an example, four procedure files could be built which would:

1. Sweep through a sequence of speeds, calling procedure 2 at each speed
2. Sweep through a sequence of flow settings, calling procedure 3 at each flow setting
3. Sweep through a sequence of temperature settings, calling procedure 4 at each temperature setting
4. Conduct a data acquisition process

Each of the procedures described above would be very short and simple to write but could produce an extensive set of data.

The @PROCEDURE keyword allows this sort of branching. The only specification field required is the pathname of the procedure file.

```
@PROCEDURE
/specs/gp/gp_subtest
```

When a sub-procedure returns to the calling procedure using a RETURN exit path, the next mode executed will be the `default_next_mode` of the same mode where the @PROCEDURE keyword was located.

Note that the `time_out` value is not used in a mode when the @PROCEDURE keyword appears in the mode. The Test Manager will jump into to sub-procedure immediately and when the sub-procedure returns, it jumps immediately to the next mode.

11 Closed Loop (PID) Controls

One of the CyFlex standard services is closed loop control. The control task may or may not be present in a particular configuration of CyFlex. This will depend on the hardware available and the function of the test system. Many test configurations will use a `user_ctrl_task` or `eng_ctrl_task` to operate the article under test and to control some fluid temperatures or pressures. If present, a control task can be thought of as a support service for the Test Manager, since the Test Manager can send messages to change the controller mode, targets, gains, tolerance, and feed-forward characteristics. The PID control tasks will not be spawned by the Test Manager if not present when the Test Manager starts.

The CyFlex PID control tasks have been designed to handle simple single-input, multiple-output loops and to handle the two loops associated with controlling speed and torque of an engine/dyno combination. With the engine/dyno combination, there are usually two parameters which can be manipulated to control speed and torque. These two parameters are the dyno command and the engine fueling, usually referred to as the dyno and throttle.

In most of an engine operating envelope, the speed can be controlled by the dyno and the torque can be controlled by the throttle. However, with an absorbing only dyno such as eddy current dyno (typical in most test cells), at no-load operation, the speed must be controlled by the throttle.

In addition, it is possible for the dyno or throttle to be used to control engine parameters such as peak cylinder pressure, manifold boost, exhaust temperature, etc.

The definition of which controller is controlling which variable is through a parameter called the `engine_control_mode`. The keyword `@ENGINE_CONTROL_MODE` allows this to be modified during a test procedure. The keywords `@DYNO` and `@THROTTLE` permit selection of open or closed loop and positioning in open loop mode. The keywords `@TORQUE` and `@SPEED` permit setting the reference values.

12 Data Acquisition

CyFlex supports several mechanisms for acquiring and logging data. Most of these data acquisition methods can be activated and controlled through a test procedure file.

For example, the fast data logging task `floger` operates independently once it is spawned, but it can be spawned from a test procedure using the @BACKGROUND or @CO_PROCESS specifications and controlled from within the Test Manager by setting events and variables.

Table 2 shows a partial list of data acquisition processes which may be specified as part of a test mode with the options for how they may be started. Any combination of the processes may be used simultaneously. Each type of data acquisition process may be independently specified to start at the beginning of the mode or after the stabilization criteria are satisfied and in some cases at other times during the mode. Most of the data acquisition processes are handled by support services. The support process is sent a configuration message which contains information about how to do the data acquisition so the Test Manager will usually just signal the support process to start or stop and will wait for a reply message, indicating the process is complete.

Table 2: Data Acquisition Processes

Data Acquired	start_type Options	Macro Used in Procedure File
Fuel readings	Once at beginning of mode Once at end of mode Start after stabilization Synchronized with external event	AT_START AT_END AFTER_STABILITY EXTERNAL_SYNC
PAM_datapoint	Once at beginning of mode Once at end of mode Start after stabilization Synchronized with external event	AT_START AT_END AFTER_STABILITY EXTERNAL_SYNC
@WRITE_VALUES	Start at beginning of mode Start at end of mode Start after stabilization	AT_START AT_END AFTER_STABILITY

13 Special Devices

There are many intelligent instruments which can be connected to a test computer and used to collect specialized measurements. Often these devices can be controlled and accessed through a serial port or similar interface. Some of these devices are explicitly supported from the Test Manager and some can be supported through asynchronous device handlers with @AK, @AKSYNC, and @DEVCOM commands if a device configuration file is available. *Table 3* lists keywords that can be used to provide communication and synchronization for special devices of this type.

Table 3: Special Device Communication Keywords

Keywords	Device
@ASC	Modem, pager, barometers, dew point meters, etc.
@DEVCOM, @DEVCOM_ACTIONS @AKSYNC, @AKSYNC_ACTIONS, @AKSYNC_RESTART @AK_COMMAND	Devices supporting the AK protocol