WHEN YOU NEED TO BE SURE

Test Manager Support Tasks and Keywords

Version 5

May 31, 2022

Developed by SGS North America, Inc.

Version History

Version	Date	Revision Description
1	11/1/2017	Initial publication
2	8/23/2018	Format to SGS brand
3	4/2/2020	Retrofit to new template
4	12/1/2021	As applicable, added hypertext linked cross-references to cyflex.com usage help and CyFlex Manuals
5	5/31/2022	Updated all hypertext linked cross-references to cyflex.com usage help descriptions

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* 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 Overview

Several special applications other than `gp_test` can be controlled by the specifications in a procedure file. The keywords which support these applications are not handled any differently in creating the file, but there is a fundamental difference in how the process is handled. An external task is spawned which runs concurrently with the Test Manager to manage the requested process. This support application will be signaled by the Test Manager to perform various operations. The sequence of communication generally consists of the steps described in *Section 2 Handling of Support Applications* on page 2.

2 Handling of Support Applications

The following is the sequence of communication to perform support tasks.

1. The Test Manager spawns the support application with event names that will support the communication with `gp_test`. This is usually a ‘start’ event, a ‘stop’ event, a ‘configuration’ event, and a ‘reply’ event.
2. Upon starting a test mode that uses the support features, `gp_test` will send the ‘stop’ event to clear existing specifications, followed by one or more ‘configuration’ events that supply the details of functions it is to perform in this mode. These details are the lines in the spec file which follow the `@keyword`.
3. If the keyword specifications include a `start_code` and that code is `AT_START`, then the ‘start’ event is sent immediately to signal the support application to begin its operations.
   - If the code is `AFTER_STABILITY`, then the ‘start’ event is not sent until the specifications supplied with `@STABILITY_SPECS` are satisfied.
   - If the code is `AT_END`, then the ‘start’ event is sent just prior to termination of the mode. The `AT_END` option is not used by every support application.

   See `/specs/gp/gp_template` for a list of options for each keyword.
4. Most support applications will send a `reply` event upon finishing its operations. This reply will contain a `SUCCESS` or `FAILURE` code. If a `success_path` and a `failure_path` are supplied for this function, then that will terminate the test mode and `gp_test` will jump to the mode or procedure specified for that termination path. Refer to *Section 5.1 @CUTY_ACTIONS* on page 6 for an example of specifying these paths.
5. If the test mode is terminated for some reason unrelated to the support application, such as a mode timeout then `gp_test` will send the ‘stop’ event to the support application and it will cease its operations. However, the support application does not terminate. It will remain available for the next test mode which requires its services.
6. Support applications may be designed to handle keywords supplied in a specific test mode, in which case an instance will be spawned for each mode where the keyword is used.
   - `vrbl_to_file`, refer to *Section 15.1 Test Tables and vrbl_to_file Applications* on page 24
   - `state_mon`, refer to cyflex.com usage help for `state_mon`

   Other support applications may be designed so that one instance of the support application handles all modes where the keyword is used.
   - `stability`, refer to cyflex.com usage help for `stability`
   - `fr_collect` (fuel reading control), refer to this presentation on cyflex.com
   - `ecm_communication (cuty_coll, asam3_coll, *ramping)`, refer to cyflex.com usage help for the *ECM Communication* category.
   - AK_`sync`, refer to cyflex.com usage help for `AK_sync`

3 Asynchronous Communication

Use the @ASC keyword to send a command or series of commands to intelligent devices which are attached on serial (RS-232) ports. The name field specifies to which device to send the command. A configuration file must exist for each device.

Table 1: @ASC Data Fields

Data Field	Explanation

`stop_path`	The `stop_path` determines what happens when all of the commands are complete. The `MODE_TERMINATE` option specifies the test proceeds to the next mode, otherwise the current mode remains in effect until a timeout or other termination event occurs. Options are:
	`NONE`
	`MODE_TERMINATE`
	`WAIT_FOR_AUX`
	`TIMEOUT`

`fail_path`	The `fail_path` determines what action will be taken if a communications error occurs or a fault code is returned from the device. Options are:
	`NONE`
	`TEST_DONE`
	`RESTART`
	`ELSE_MODE`
	`NEXT_MODE`

Example specification:

@ASC

```
   #strt_type stop_path fail_code name  command   interval event
   AT_START   NONE      NONE    calterm "monitor" 0        -
```

4 DEVCOM Device Communication

DevCom is a Device Communication subsystem of CyFlex, used in testing scenarios to control and communicate with Intelligent Electronic Devices that support a serial communications protocol. Smoke meters are an example.

The DevCom subsystem is a collection of applications, device drivers, and user-configurable specification files, developed to support a wide range of intelligent devices. This is accomplished by allowing the user to customize communication with a particular device by changing the specification file to work with the device's characteristics without having to develop a unique software application for that device.

Refer to the *Device Communication User Guide* for additional information.

4.1 @DEVCOM_ACTIONS

Use the `@DEVCOM_ACTIONS` keyword to specify the actions and timing associated with all the DevCom communications for a particular test mode.

Table 2: @DEVCOM_ACTIONS Data Fields

Data Field	Explanation
`start_code`	Code for when to send the command. Options are `AT_START` or `AFTER_STABILITY`. Default is `AT_START`.
`success_path`	Code for what action to take when communication is complete. Options are `NONE`, `MODE_TERMINATE`, `RETURN`, a mode number, or a procedure file pathname. Default is `NONE`.
`fail_path`	Code for what action to take if communication fails. Options are `NONE`, `MODE_TERMINATE`, `RETURN`, a mode number, or a procedure file pathname. Default is `NONE`.

Example specification:

```
DEVCOM_ACTIONS
       #start_code   success_path    fail_path
       AT_START      MODE_TERMINATE  /specs/gp/quit
```

4.2 @DEVCOM

The `@DEVCOM` keyword specifies, on the first line, a device name (instrument), a configuration file for that instrument, and an optional field for restarting the support application when starting this test mode. This line is followed by up to 20 "commands". Each command is a string consisting of a device command keyword such as the `AOPT` shown in an example below, followed by a number of CyFlex variable names. This command is send by `gp_test` to the support application where, using the configuration file, it is translated into a device specific message. Refer to the *Device Communication User Guide* for information on how to set up a configuration file for a particular instrument/device.

Example specification:

```
#####################################################################
# DEVCOM .. commands are used to communicate with an AK communications
device,
#        usually an AVL smoke meter.
#
# instrument name - Name associated with a task which actually
communicates with the device.
#
#  spec_filename
#
#  RESTART        - anything entered as the 3rd fiels will cause
#                   devcom_coll task to be terminated and restarted
#
#  for example:

@DEVCOM
#instrument_name     spec_filename        (optional RESTART)
   AVL483            /specs/xyz           RESTART

# Here are some examples for AVL 483 smoke meter.
#
# in the /specs/AVL483.spec
#
#"AOPT,%d %d %d %d %d"
#
#
# in gp test script

"AOPT SMBlkPcnt SMWhtVal SMGreyVal SMBlkVal"

#  Integer values retreived from executing the command will be placed
#  in asset variables SMBlkPcnt SMWhtVal SMGreyVal and SMBlkVal.
#
# in the /specs/AVL483.spec
# EDIL %d %f %f %f
#
# in a gp test script

"EDIL SS_dil_typ_TR SS_dil_TR 1.00 10.00 1.00"

#  The values from asset variables SS_dil_typ_TR and SS_dil_TR are
#  passed into the command as well as the litteral values 1.00 10.00
and 1.00.
#
#
```

5 CUTY Communication

Several Test Manager keywords enable control of the ECM through communication with a CUTY system. Four keywords specify the commands which are sent to the ECM and one keyword specifies the timing and responses to completion of those actions. The latter keyword is @CUTY_ACTIONS. It contains three data fields for the `start_type`, `stop_path`, and `fail_path`. Those data fields have the same function as other keywords, so they will not be described in detail here.

5.1 @CUTY_ACTIONS

If the @CUTY_ACTIONS keyword is not used, but one or more of the other CUTY commands are used, then the actions default to AT_START, NONE, and NONE for `start_type`, `stop_path`, and `fail_path`, respectively.

Example specification:

```
@CUTY_ACTIONS
   #start_type   success_path       fail_path
   AT_START      MODE_TERMINATE     /specs/gp/gp_hndl_cuty;23
```

In the preceding example, the communication would begin at the start of the test mode. When all commands have been completed, the mode would be terminated and if there was a failure of communication, execution would be passed to mode 23 in test procedure `/specs/gp/gp_hndl_cuty`.

All CUTY commands specified in a particular test mode are queued in the order they are entered in the procedure file. When communication begins, the commands are sent in that order as rapidly as the ATA driver can process them. A reply is expected for each command to indicate the next command can be sent. When all of the queued commands are sent, the mode will be terminated if the `stop_path` is MODE_TERMINATE. If an error in communication occurs, the `fail_path` option is used.

5.2 CUTY_SET

Use the CUTY_SET keyword to modify the value of a parameter in the ECM. The value field that is transmitted to the ECM is always a string. The actual string to be transmitted may be specified by enclosing it in single quotes. The value may be a constant, variable, or expression. For instance, the FUELOVER variable expects HEX number format, so a value of FF or 0xFF would be a valid field and the value of 255 would not give the same result. Each variable that is sent to the ECM is followed with a request to read the value back to verify that the change actually took place. If the value read back is different than the one transmitted, then one retry attempt is made. If the 2nd retry is unsuccessful in changing the value, then an error is reported.

Example specification:

```
@CUTY_SET
   #ECM_name  ECM_variable                value
   ECM0             'T_AIM_PermitSwitchEnbl'   0[none]
   ECM0             'T_ATM_bs_Enbl'            0x0FFF8197'
```

5.3 @CUTY_RAMP

Use the @CUTY_RAMP keyword to generate ramping operations on ECM variables. A support task will be spawned to manage the commands required to generate the ramps. The targets and ramp rate may be expressed as decimal constants, variable labels, or computed expressions. Note that the units of any variable transmitted to the ECM must be [none]. The constants used in the @CUTY_RAMP specification do not require that the units be appended. The `termination` field is optional. The only option for that field is FREEZE. If FREEZE is specified, then when the mode is terminated, the last value that was produced will be the final output value. Otherwise, the final output value will always be the `end` value of the ramp.

Example specification:

```
@CUTY_RAMP
#ECM_name ECM_variable       start             end_target rate termination
ECM0     'SOI_Override_Val'  soi_override_val    soi_setpt 0.5[none]
```

5.4 @CUTY_GET

The @CUTY_GET keyword retrieves the value of an ECM variable from the ECM and places it in a CyFlex real variable. This can be used to test that a value was really modified or to get data which will be logged as part of a fuel reading, displayed, etc.

Example specification:

```
@CUTY_GET
#ECM_name       ECM_variable                          CyFLex_label
ECM0            'EVT_ti_DieselOntime2_T[0]'           array_value
```

5.5 @CUTY_COMMAND_MESSAGE

Use this keyword to send various commands to the ECM.

Example specification:

```
@CUTY_COMMAND_MESSAGE
   #ECM_Name         command_code
   ECM0              REQ_CHGLOCK
```

The commands of the preceding keywords are queued in the order they are entered in the procedure file. It is possible to use the same keyword more than once to control the sequence of transmission of the commands. The example below illustrates this:

```
@CUTY_SET
   #ECM_name         ECM_variable               value
   ECM0              'T_AIM_PermitSwitchEnbl'   0[none]
   ECM0              'T_ATM_bs_Enbl'            0x0FFF8197'


@CUTY_GET
   #ECM_name         ECM_variable                          CyFLex_label
     ECM0            'EVT_ti_DieselOntime2_T[0]'           array_value

@CUTY_SET
   #ECM_name         ECM_variable            value
   ECM0              'T_ATM_bs_Enbl'         0x0FFF8197'
```

6 ASAM3 Communication

Several Test Manager keywords enable control of the ECM through communication with a CUTY system. Four keywords specify the commands which are sent to the ECM and one keyword specifies the timing and responses to completion of those actions. The latter keyword is @ASAM3_ACTIONS. It contains three data fields for the `start_type`, `stop_path`, and `fail_path`. Those data fields have the same function as other keywords, so they will not be described in detail here.

Refer to *ASAM3 MC Interface Setup* for supplemental information.

6.1 @ASAM3_ACTIONS

If the @ASAM3_ACTIONS keyword is not used, but one or more of the other CUTY commands are used, then the actions default to AT_START, NONE, and NONE for `start_type`, `stop_path`, and `fail_path`, respectively.

Example specification:

```
@ASAM3_ACTIONS
   #start_type   stop_path          fail_path
   AT_START      MODE_TERMINATE     /specs/gp/gp_hndl_asam;23
```

In the preceding example, the communication would begin at the start of the test mode. When all commands have been completed, the mode would be terminated and if there was a failure of communication, execution would be passed to mode 23 in test procedure `/specs/gp/gp_hndl_asam`.

All ASAM3 commands specified in a particular test mode are queued in the order they are entered in the procedure file. When communication begins, the commands are sent in that order as rapidly as the ATA driver can process them. A reply is expected for each command to indicate the next command can be sent. When all of the queued commands are sent the mode will be terminated if the `stop_path` is MODE_TERMINATE. If an error in communication occurs, the `fail_path` option is used.

6.2 @ASAM3_SET

Use the @ASAM_SET keyword is to modify the value of a parameter in the ECM. The value field that is transmitted to the ECM is always a string. Specify the actual string to be transmitted by enclosing it in single quotes. The value may be a constant, variable, or expression. For instance, the FUELOVER variable expects HEX number format, so a value of FF or 0xFF would be a valid field and the value of 255 would not give the same result. Each variable that is sent to the ECM is followed with a request to read the value back to verify that the change actually took place. If the value read back is different than the one transmitted, then one retry attempt is made. If the 2nd retry is unsuccessful in changing the value, then an error is reported.

Example specification:

```
@ASAM3_SET
   #reg_name ECM_name      ECM_variable                value
   asam3_1   ECM0          'T_AIM_PermitSwitchEnbl'    0[none]
   asam3_1   ECM0          'T_ATM_bs_Enbl'          0x0FFF8197'
```

6.3 @ASAM3_RAMP

Use the @ASAM3_RAMP keyword to generate ramping operations on ECM variables. A support task will be spawned to manage the commands required to generate the ramps. The targets and ramp rate may be expressed as decimal constants, variable labels, or computed expressions. Note that the units of any variable transmitted to the ECM must be [none]. The constants used in the @ASAM3_RAMP specification do not require that the units be appended. The `termination` field is optional. The only option for that field is FREEZE. If FREEZE is specified, then when the mode is terminated, the last value that was output will be the final output value. Otherwise, the final output value will always be the `end` value of the ramp.

Example specification:

```
@ASAM3_RAMP
#reg_name ECM_name  ECM_variable start end         rate  termination
asam3_1   ECM0    'SOI_Override'  soi  soi_setpt   0.5   FREEZE
```

6.4 @ASAM3_GET

The @ASAM3_GET keyword retrieves the value of an ECM variable from the ECM and places it in a CyFlex real variable. This can be used to test that a value was really modified or to get data which will be logged as part of a fuel reading, displayed, etc.

Example specification:

```
@ASAM3_GET
#reg_name ECM_name    ECM_variable                 CyFLex_label
asam3_1   ECM0 'EVT_ti_DieselOntime2_T[0]'         array_value
```

6.5 @ASAM3_COMMAND_MESSAGE

Use this keyword is used to send various commands to the ECM.

Example specification:

```
@ASAM3_COMMAND_MESSAGE
   #reg_name        ECM_Name              command_code
   asam3_1          ECM0           REQ_CHGLOCK
```

The commands of the preceding keywords are queued in the order they are entered in the procedure file. It is possible to use the same keyword more than once to control the sequence of transmission of the commands. The example below illustrates this:

```
@ASAM3_SET
   #reg_name ECM_name ECM_variable              value
   asam3_1   ECM0    'T_AIM_PermitSwitchEnbl'   0[none]
   asam3_1   ECM0    'T_ATM_bs_Enbl'            0x0FFF8197'


@ASAM3_GET
   #reg_name ECM_name  ECM_variable                          CyFLex_label
     asam3_1 ECM0      'EVT_ti_DieselOntime2_T[0]'           array_value

@ASAM3_SET
   #reg_name ECM_name        ECM_variable           value
   asam3_1   ECM0            'T_ATM_bs_Enbl'        0x0FFF8197'
```

7 Stability

7.1 @STABILITY_ACTION

Use the @STABILITY_ACTION keyword to specify actions when stabilization occurs.

If the @STABILITY_SPECS keyword is used to specify stabilization criteria, then this keyword may be used to specify what actions are required after the criteria are met. Possible actions are:

- MODE_TERMINATE
- TERMINATE_TO_ELSE_MODE
- WAIT_FOR_STABILITY

Table 3: @STABILITY_ACTION Data Field

Data Field	Explanation
action_code	A code which indicates certain special actions to perform

Example specification:

```
@STABILITY_ACTION
        #action_code
        MODE_TERMINATE
```

The preceding specification terminates the test mode when stabilization is complete.

Notes:

The actions associated with any keyword which uses the AFTER_STABILITY macro for a start_type is assumed to be one of the actions taken when stability is complete.

The MODE_TERMINATE action means that when stabilization is complete, the test mode is immediately terminated. It may be terminated prior to the completion of stability by other mechanisms, such as timeout, limits, etc.

The WAIT_FOR_STABILITY action means that no other mechanism for mode termination may precede the completion of stability. If some other action occurs prior to completion of stability, the request to terminate is suspended until stabilization is complete. The WAIT_FOR_STABILITY action code by itself does not specify that the mode be terminated, only that no other action can cause termination prior to stability.

Use TERMINATE_TO_ELSE_MODE to force the execution of the mode specified with keyword @ELSE_MODE when stability occurs.

The action codes may be used in combination to achieve the desired effect.

Additional example specifications:

```
@STABILITY_ACTION
        #action_code
        TERMINATE_TO_ELSE_MODE
```

Completion of stabilization will cause a branch to the mode specified by the @ELSE_MODE keyword.

```
@STABILITY_ACTION
       #action_code
       WAIT_FOR_STABILITY
```

This mode cannot be terminated until stabilization is complete. Completion of stability will, however, not necessarily cause the termination of the mode.

7.2 @STABILITY_SPECS

Use the @STABILITY_SPECS keyword to specify a list of the stability criteria that are to be evaluated during the test mode. Stability is complete when all of the specified criteria are achieved. Refer to *Section 7.1 @STABILITY_ACTION* on page 10 for a more complete explanation of each type of stability criterion.

Table 4: @STABILITY_SPECS Data Fields

Data Field	Explanation
type_code	The type of criteria. Options are TIME_DELAY, VARIANCE, DEVIATION, CURRENT_DEVIATION, K_VARIANCE, STD_DEVIATION.
variable	The variable label to which the criteria is supplied (except type = TIME_DELAY)
timeout	The time window associated with the criteria (except type = CURRENT_DEVIATION) .
rate	the rate at which the criteria is evaluated
reference	The reference value for the criteria. This may be a constant, variable, or computed expression.
tolerance	The tolerance for the criteria.
minimum_reference	For type=K_VARIANCE, the lower threshold for the reference.

Example specifications:

```
@STABILITY_SPECS
#type_code  variable  timeout  rate reference  tolerance min_ref
DEVIATION    TORQUE    20[sec]        SLO  1200[lb_ft]  10.0 -
```

The engine torque must be within 10 lb.-ft of 1200 for 20 seconds to have stability.

```
@STABILITY_SPECS
#type_code  variable  timeout   rate  reference  tolerance  min_ref
VARIANCE    fuel_rate  10[sec]  SLO     -        .0[lb/hr]
TIME_DELAY     -       20[sec]
```

If after at least 20 seconds the fuel_rate does not wander by more than 1 lb./hr. for 10 seconds, stabilization is achieved.

8 Test Limits

The Test Manager (gp_test) uses two keywords to allow changing the path of a test procedure based on limits set on one or more variables. The functionality is very similar to that supported by the limit application; refer to *Limits Monitoring Applications*. Specifying limits with the gp_test keywords is only used to change the path of the test procedure When a test procedure is loaded and contains either of the keywords, the test_limits support application is launched. As the test procedures are read prior to start of the test, each of the limit specifications is sent to the test_limits application as a configuration message. The limits are not active until the mode in which the limit specification appears is started. Upon termination of the mode, those limit specifications are disabled.

8.1 @LIMIT_SPECS

Use the @LIMIT_SPECS keyword to specify up to XXX limits per test mode. Each limit specification has an optional next_path. This is the path that the procedure will jump to if the limit is violated while this test mode is being executed. The default next_path is MODE_TERMINATION, meaning terminate the mode when the limit is violated. Refer to *Section 3.1, @LIMIT_SPECS* in *Common Test Manager Keywords* for further details.

Example specification:

```
@LIMIT_SPECS
  #label  value      type   interval period_out next_path
  RPM     2400[rpm]  U      MED      10[sec]   /specs/gp/gp_shutdown
       oilrfl_p  60[psi]   U      MED     5[sec] /specs/gp/gp_reset;25
```

Set an upper limit of 2400 RPM on engine speed. Execute the gp_shutdown test procedure if this is exceeded for at least 10 seconds continuously. If oil rifle pressure exceeds 60 psi for 5 seconds, then run the gp_test procedure starting in mode 25.

Note:

> The processing of the limit occurs only during the mode in which it is specified. It is enabled when the mode starts and disabled when the mode terminates.

Violation of a limit will not cause the display to blink.

Additional example specification:

```
@LIMIT_SPECS
  #label      value      type   interval   period    next_path
  coolant_t   60[deg_F]   U       SLO      0[sec]       22
  RPM         400[rpm]    L       SLO      0[sec]    /specs/gp/gp_done
  oil_p "oil_model-5[psi]"  L     SLO      0[sec]    RETURN
```

Branch to mode 22 if the coolant temperature exceeds 260F during this test mode and jump to procedure gp_done if the engine speed drops below 400 rpm.

If the oil_p variable is more than 5 psi below the oil_model variable, return to the calling procedure.

8.2 @LIMIT_SPECS_ALL

Use the `@LIMIT_SPECS_ALL` keyword to specify a list of variables with limits set on them. If the **`all`** of the limits are violated, then the mode is terminated. If the `next_path` field is `0` or `-,` then the `default_next_mode` path (in `@MODE`) is executed. The limit value may be expressed as a constant, variable label, or computed expression.

Table 5: @LIMIT_SPECS_ALL Data Fields

Data Field	Explanation
`exit_path`	The path to execute when/if all the specified limits are simultaneously violated. This may be a mode number, a procedure pathname, `MODE_TERMINATE`, or `RETURN`.
`variable`	A variable on which the limit is set. This may be a real, integer, statistical, property, or composition variable.
`value`	The limit value (constant/variable/expression)
`type`	Upper or lower limit: `U` | `L`
`interval`	The rate at which to check the limit `FAS` | `MED` | `SLO`
`period_out`	the period for which the limit must be violated before the action is taken

Example specification:

```
@LIMIT_SPECS_ALL
        #exit_path
        MODE_TERMINATE
        #label    value          type     interval   period_out
        RPM       2400[rpm]      U        MED        10[sec]
        blow_by   10[in_h2o]     U        SLO        0[s]
```

Set an upper limit of 2400 rpm on engine speed and an upper limit of 10[in_h2o] on blow_by. Terminate the test mode if both are violated.

Notes:

The processing of the limit occurs only during the mode in which it is specified. It is enabled when the mode starts and disabled when the mode terminates.

Violation of a limit will not cause the display to blink.

Two string variables can be specified to give the operator feedback on the state of this specification.

9 Test Compute

Use the @CREATE_EXPRESSION keyword to create computed expressions that will be used during a specific gp_test. This keyword was created in response to the amount of volume and complexity that has been created in gen_labels.*NNN*. Sometimes it is advantageous to have computed expressions that exist only during the duration of a specific test.

Note:

The keyword @CREATE_EXPRESSION must be placed in the header section of a test procedure file somewhere between the start_mode and the first @MODE.

Table 6: @CREATE_EXPRESSION Data Fields

Data Field	Explanation
variable	The variable name of label used
type	The variable can be: REAL|INTEGER|LOGICAL|STRING
units	The type of units to be used with the created variable
event/timer	The event name or timer designation that will evaluate the expression
expression	The computed expression to be used

Example specification:

```
@CREATE_EXPRESSION
#(up to 16 per procedure)
@label  type  units event/time expression
myvar   REAL  rpm     1000    "if RPM>Idle_Speed then 700[rpm] else
Idle_Speed

mydesc   STRING  -       1000      "'test'+count"
```

The variable myvar is created as a REAL with RPM as its units and evaluated once a second. The expression states that if RPM is greater than the value of Idle_Speed then set myvar to a value of 700 rpm otherwise set it to the value of Idle_Speed. The variable mydesc is created as a string variable that includes the value of test added to count.

Notes:

This keyword is the functional equivalent of gen_labels.NNN. However, @CREATE_EXPRESSION does not support a history flag, tolerance, and a display format. The display format defaults to 2 places for REAL variables.

The true/false descriptions of LOGICAL variables default to ON/OFF. The history flag is OFF and the default tolerance is 1.0. The variable in a @CREATE_EXPRESSION specification will be created if it does not already exist. If it does exist, but there is no computed expression associated with it, then the computed expression will be created,

If the variable already exists and has a computed expression, then an error is reported.

The use of the @CREATE_EXPRESSION keyword causes gp_test to spawn the new task named comptest.

10 Fuel Reading Control

The associated keywords take fuel readings and generate PAM datapoints.

10.1 @FUEL_READING

Use the @FUEL_READING keyword to take one or more fuel readings during this test mode. If the desired_time is 0 or -, the time specified by the variable target_fr_tim will be used.

The number_of_readings, interval, and desired_time data fields can all be specified as a constant, variable label, or computed expression.

Table 7: @FUEL_READING Data Fields

Data Field	Explanation
start_type	Code for when to send a start signal to the collector task. Options are: AT_START |AFTER_STABILITY |EXTERNAL_SYNC
stop_path	Code for what action to take when the fuel reading collector task completes its function. Options are: NONE| MODE_TERMINATE| RETURN |a mode number| a procedure file pathname.
number_readings	The number of fuel readings to request
interval	The time between requests (if number_readings > 1)
sync_event	An event name for external synchronization
desired_time	The desired fuel reading sample time; specifying a non-zero desired_time will change the value of the target_fr_tim variable.

Example specifications:

```
 #start_type          stop_path
AFTER_STABILITY       MODE_TERMINATE
#number_readings      interval        sync_event   desired_time
1                     0[s]            -            0[s]
```

Request 1 fuel reading after stabilization is complete. Terminate the mode when the fuel reading is complete.

```
@FUEL_READING
       #start_type          stop_path
       AFTER_STABILITY      MODE_TERMINATE
       #number_readings     interval  sync_event     desired_time
       num_read             5[min         -          90[sec]
```

Take three fuel readings to be determined by the value of the variable num_read at five-minute intervals, each 90 seconds long. Terminate the mode when all three fuel readings have been completed.

10.2 @FUEL_READING_SYNC

Use the @FUEL_READING_SYNC keyword to synchronize several processes that are required to generate a PAM datapoint. The keyword allows the construction of a chain of events that provide the synchronization.

This keyword allows multiple processes to be synchronized with fuel readings. The synchronization is handled externally from gp_test. The specification consists of a list of output events that will be emitted in the sequence that they are listed. Each output event is emitted when all of the input events listed on its line and all preceding lines have been received. This condition is overridden by the specified timeout (0 timeout indicates no timer). The timeout for a particular line does not start until the output event on the previous line has been emitted. All input events are attached at the time a fuel reading is requested, so if an input event of a later specification line is received before those of a preceding line, it is still considered to be satisfied, but the corresponding output event would not be emitted until all those preceding it have been emitted.

Note:

The maximum specified delay for this entire process is the value of the variable FR_write_delay. If that time expires after the issuance fr_ready, the datapoint will be written even if fr_write_ok is not received. For a better understanding of the variables and events associated with fuel readings, refer to Gazette.6b.97-" Variables, Events, and Processes associated with fuel readings"

Table 8: @FUEL_READING_SYNC Data Fields

Data Field	Explanation
timeout	maximum wait time for the specified input events - the output event is issued if this timeout expires before all of the input events are received.
output_event	An event that will be set when all of the specified input events are received or the timeout expires
input_events	Up to 4 input events which must all be received before this sequence in the chain is satisfied.

Example specification:

```
@FUEL_READING_SYNC
#when all the input events have arrived, the output event is emitted
#and we go to the next spec. Keep doing that until the list is
#complete

#event_sync (event sequences required to complete a datapoint)
#max_timeout          output_event      input_event_list (up to 4)
  0[sec]               TS_StrtAcq       fr_ave_strt
  0[sec]              TS_OpCondCmp      HS_AcqInPrg fr_ready HS_AcqCmp
  0[sec]               fr_write_ok      HS_AnlsCmp
```

Notes:

- Fr_write_ok should always be the last output event.
- FR_write_delay is automatically set to 4 minutes when @FUEL_READING_SYNC is used.
- @FUEL_READING_SYNC can only be used in modes where @FUEL_READING or @FUEL_READING_STATS are also used.

11 Write Values

Use @WRITE_VALUES to write text data into a file and control the data, format, and rate through the test script. Essentially, a data logging type of operation may be created through gp_test. The most likely use is to capture the value of a particular variable after the operating conditions have been obtained through the test script.

Table 9: @WRITE_VALUES Data Fields

Data Field	Explanation
start_type	Code for when to send a start signal to the collector task. Options are: AT_START | AFTER_STABILITY | AT_END | NEW_FILE. NEW_FILE means "open file to WRITE, removing previous copy of the file
file_name	The file where the data will be written. The filename can be a computed expression, using the + symbol for string concatenation.
value	The ASSET variable or expression from which the value is to be obtained. A dash - indicates no value/variable.
"format_string"	The C format string to be used for formatting the write. Quotes are required.
units (optional)	Optional definition of output units if the value is a computed expression. Default units will be used if the value is a computed expression and units are not entered unless the expression is enclosed in braces { }.

Example specification:

```
@WRITE_VALUES
 #start_type  file_name          value           "format_string"  units
 AT_START     /data/tq_sp        -                "rpm         "
 AT_START     /data/tq_sp        -                "torque   \n"
 AT_START     /data/tq_sp        NOTIFY           "%s\n"
 AT_START     /data/tq_sp        ctl_spd          "%11.2f "
 AT_START     /data/tq_sp        "ctl_spd/2[none]"  " %10.3f "      rpm
 AT_START     /data/tq_sp        "{ ctl_spd/2[none] }"  " %8.1f "
 AT_START     /data/PC_format/pms_wrt.csv  pms_cart  ",@2.0i  "
 AT_START     /data/PC_format/pms_wrt.csv  count     ",@2.0i  "
```

12 State Monitor

```
#-------------------------------------------------------------------------
# @STATE_MON_ACTIONS  -

# success_path        - exit path for successful return from state_mon
#                       This may be MODE_TERMINATE, NONE, mode number,
#                       or another gp_test procedure

# failure mode        - The exit path for a failure return from state_mon
#                       This may be MODE_TERMINATE, NONE, mode number,
#                       or another gp_test procedure

# read_mode           - must be READ or READ_ONCE

# action_code         - code to indicate the operational method used by
#                       state_mon - one of the following options
#                          VERIFY, MONITOR, IMMEDIATE


@STATE_MON_ACTIONS
   #success_path     failure mode      read_mode         action_code
   MODE_TERMINATE       10            READ_ONCE          VERIFY


#-------------------------------------------------------------------------
# @STATE_MON_SPEC_FILES

# spec_file_pathname - the pathname of the 'state_mon' specifications.
#                      There can be a maximum of sixteen files

# state_index        - the label of the variable that will contain the
#                      index value that should be read from the file.
#                      The label should exist.

@STATE_MON_SPEC_FILES
   # spec_file_pathname                  state_index
     /specs/stbl/state_mon_specs          state_index


#-------------------------------------------------------------------------

# @STATE_MON_EXCEPTIONS

# time_out           - the length of time to allow the variables to reach
#                      the specified states.  Valid entries are a value,
#                      an Cyflex label, or a computed expression.
#                      The label should exist.

# timeout_path       - the path to be taken when a timeout occurs. Valid
#                      entries are mode number, MODE_TERMINATE, NONE, or
#                      another gp_test procedure

# state_change_path  - the path to be taken when state_mon indicates that
#                      a state variable, specified in one of the spec files,
```

```
#                       with an action extension of _S has failed. Valid
#                       entries are mode number, MODE_TERMINATE, NONE, or
#                       another gp_test procedure

# warning_fail_path  - the path to be taken when state_mon indicates that
#                       a state variable, specified in one of the spec files,
#                       with an action extension of _W has failed. Valid
#                       entries are mode number, MODE_TERMINATE, NONE, or
#                       another gp_test procedure

# critical_fail_path - the path to be taken when state_mon indicates that
#                       a state variable, specified in one of the spec files,
#                       with an action extension of _C has failed. Valid
#                       entries are mode number, MODE_TERMINATE, NONE, or
#                       another gp_test procedure

# read_error_path    - the path to be taken when state_mon indicates that
#                       a read error occurred when the specification files
#                       were read. Valid entries are mode number,
#                       MODE_TERMINATE, NONE, or another gp_test procedure

@STATE_MON_EXCEPTIONS
# time_out  timeout   state_change   warning_fail   critical_fail
read_error
#            path         path           path            path          path
   30[sec]    90      MODE_TERMINATE      10            15             20

#-------------------------------------------------------------------------
```

13 Cyber Apps

13.1 @CYBER_ACTIONS

Use this keyword to direct when the command will take place during the mode. If the commands fail, then an alternate path may be taken.

Table 10: @CYBER_ACTIONS Data Fields

Data Field	Explanation
`start_code`	At what point during the mode should execution of the commands begin
`success_path`	If all commands are successful; refer to *Table 11*
`failure_path`	If a command fails; refer to *Table 12*

Table 11: @CYBER_ACTIONS success_path Options

Data Field	Explanation
`AT_START`	At the beginning of the mode
`AT_END`	At the end of the mode
`AT_START_AND_END`	At the beginning and ending of the mode
`AFTER_STABILITY`	After Stability has been achieved. Refer to *Section 7.2 @STABILITY_SPECS* on page 11.)

Table 12: @CYBER_ACTIONS failure_path Options

Data Field	Explanation
`NULL`	`NULL` designates 'does not apply'
`MODE_TERMINATE`	Allow the mode to end and execute the `default_next_mode`.
`RETURN`	Return to the calling `gp_test` procedure.
`90`	Mode to mode `90` of this test
`/specs/gp/gp_Cainit2`	Execute the `gp_test` called `gp_Cainit2`

Example specifications:

```
@CYBER_ACTIONS
#start_code       success_path       failure_path
 AT_START        MODE_TERMINATE           90
```

The preceding command orders the @CYBER keyword to execute its commands at the beginning of the mode. If any commands fail, then move to mode 90 of the test. If all commands are successful, then allow the mode to terminate and execute the default next mode.

13.2 @CYBER

Use @CYBER to issue a command to the Cyber application or to the CyberServer. The command code will determine the action taken.

Table 13: @CYBER Data Fields

Data Field	Explanation
command	A command key; refer to *Table 14*
name	The system or component name
value	The system or component value

Table 14: @CYBER Commands and Arguments

Command	Argument
CA_APPLICATION	<application_name><application_file>
CA_COMPONENT	<component_name><component_file>
CA_PARAMETER	<parameter_name><parameter_value>
CA_LOAD	<cyberapps_name>
CA_RUN	
CA_PAUSE	
CA_STOP	
CA_BEGIN_CONFIG	
CA_END_CONFIG	

Example specification:

```
@CYBER
#command              name              value
CA_PAUSE
CA_COMPONENT         'route'         'Indy38thSt'
CA_PARAMETER         'VehMass'        75000[lbs]
CA_RUN
```

The preceding commands configure `CyberTruck` to use the 38th Street route and set the truck mass to `75000` pounds.

14 Unico Dyno Controller

Use the @UNICO_ACTIONS command to communicate with an UNICO controller running on TCP/IP connection.

Table 15: @UNICO_ACTIONS Data Fields

Data Field	Explanation
start_code	Code for when to send the command. Options are: AT_START | AFTER_STABILITY. Default is AT_START.
success_path	Code to specify action to take when communication is complete. Options are: NONE | MODE_TERMINATE | RETURN | a mode number | a procedure file pathname. Default is NONE.
fail_path	Code to specify action to take if communication fails. Options are: NONE | MODE_TERMINATE | RETURN | a mode number | or a procedure file pathname. Default is NONE.

Example specification:

```
@UNICO_ACTIONS
      #start_code   success_path    fail_path
      AT_START      MODE_TERMINATE /specs/gp/quit
```

14.1 ECM Communications

14.1.1 @UNICO_GET

Use the @UNICO_GET command to obtain a value for a specific variable from the ECM.

Table 16: UNICO_GET Data Fields

Data Field	Explanation
controller_variable	The name of a UNICO interface control variable. This may be a constant, variable, or computed expression which resolves to a valid ASSET label
ASSET_label	The label of the variable where the result will be placed

Example specification:

```
@UNICO_GET
      #controller_variable       ASSSET_label
      "'injector' + cyl_number"  fixed_label
```

14.1.2 @UNICO_SET

Use the @UNICO_SET command to set a value for a specific variable from the ECM.

Table 17: @UNICO_SET Data Fields

Data Field	Explanation
controller_variable	The name of a UNICO interface control variable. This may be a constant, variable, or computed expression which resolves to a valid ASSET label
value	This may be a constant, variable label, or computed expression.

Example specification:

```
@UNICO_GET
       #controller_variable        value
       'Some_label'                100[none]
```

15 Auxiliary Tasks

The Test Manager allows for the design of a general type of support task with no unique purpose, but with a defined communication protocol with the Test Manager. An auxiliary task can be designed to perform a special function within a test mode. It must support start and stop events from the Test Manager and it must `reply` to the Test Manager when its function is complete. The reply may indicate a SUCCESS or FAILURE. When starting the auxiliary task, the Test Manager may provide it with command line arguments. This may be the name of a specification file which the auxiliary task uses.

An example of an auxiliary task is performing an engine start. This procedure can be fairly complicated and involve multiple stages. There may be different requirements for how to perform an engine start from one engine or test to another.

Use the task `engine_start` to perform this function. It may be used in any test mode by specifying the @AUXILIARY_TASK keyword. Refer to cyflex.com usage help for `engine_start` for supplemental information.

Example specification:

```
@AUXILIARY_TASK
    #start_type       stop_path           failure_action
    AT_START          MODE_TERMINATE      ELSE_MODE
    #task_pathname                  "command_line"
    /asset/bin/engine_start         "/specs/starter"
```

- Specify the name of the task in the `task_pathname` field.
- Specify the command line arguments are specified in the `command_line` field. Enclose the command line in double quotes since it could contain multiple arguments.
- Specify a specification file for the `engine_start` utility.

15.1 Test Tables and vrbl_to_file Applications

The most commonly used auxiliary application is the `vrbl_to_file` application. Refer to cyflex.com usage help for `vrbl_to_file` for supplemental information.