ADI	@ADI_ACTIONS

Used to specify the actions and timing associated with ADI_SEND and ADI_REQUEST.

Keyword:

@ADI_ACTIONS

Usage:

Keyword used to specify the actions and timing associated with ADI_SEND and ADI_REQUEST.

Data Fields:

start_type	code to specify when the commands will be executed
success_path	code for what action to take when all commands have been completed - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a test procedure pathname.
failure_path	code for what action to take if communications fail to complete successfully - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a test procedure pathname.

Example Specification:

```
@ADI_ACTIONS
        #start_type     success_path      failure_path
        AT_START        MODE_TERMINATION  99
```

ADI	**@ADI_REQUEST**

Used to support communication with the ADI breadboard development system.

Keyword:

@ADI_REQUEST

Usage:

Used to support communication with the ADI breadboard development system.

Data Fields:

variable	Variable name received from ADI breadboard
ASSET variable	CyFlex variable where value will be placed

Example Specification:

```
@ADI_REQUEST
        #variable                   CyFlex variable where value will be placed
        'css.adi_global_c::timeout'  adi_req
```

Notes:

All of the specifications require that ECM variable names be enclosed in single quotes, designating that they are a literal string.

ADI	**@ADI_SEND**

Used to support communication with the ADI breadboard development system.

Keyword:

@ADI_SEND

Usage:

Used to support communication with the ADI breadboard development system.

Data Fields:

variable	code to specify when the commands will be executed
value	code for what action to take when all commands have been completed - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a test procedure pathname.

Example Specification:

```
@ADI_SEND
        #variable                   value
        'css.adi_global_c::timeout'  2
```

Notes:

All of the specifications require that ECM variable names be enclosed in single quotes, designating that they are a literal string.

Send command to an AK device (ASAP Interface)

Keyword:

@AK_COMMAND

Usage:

Send a command to a device that supports the ASAP3 interface over a serial link (RS-232). The command string will be parsed and converted to a string which the device can interpret. The value **ASAP** should be provided for the *instrument_name*. The ASAP3 strings supported are identified below. The sequence of use of the strings required on initialization is:

- init
- identify
- select

After that, the remaining commands may be used as desired. Monitoring allows measured parameters to be continuously monitored by CyFlex and for their values to be placed in CyFlex variables. It is recommended that a *monclear* command be issued before a new list is specified to avoid transferring data that is not necessary, which would cause performance degradation. Parameter *set*s and *get*s allow fixed parameters to be sent or received. This is supported while monitoring is active or inactive.

The processes **ASAPDrv** and **ASAPMgr** are required to be operating to support the use of this keyword. The commands /specs/cmds/start_ASAP3 and /specs/cmds/slay_ASAP3 start and stop these processes.

Data Fields:

```
@AK_COMMAND
  #start_type stop_code       failure_action
  AT_START    MODE_TERMINATE   99
  #instrument name
  ASAP
  #command key strings
  "command"
```

Data field	Meaning
start_type	code for when to send the command - options are AT_START and AFTER-STABILITY
stop_path	code for action to take when communication has completed successfully - options are MODE_TERMINATE, NONE, RETURN, a mode number, or a test procedure pathname
fail_path	code for execution path to take if communication fails or an error is reported by the device - options are MODE_TERMINATE, NONE, RETURN, a mode number, or a test procedure pathname
instrument_name	the device (should normally be ASAP)

command strings	a list of commands to execute in sequence

Command Strings:

The following command strings are supported:

"timeout asset_var"	defines the CyFlex variable that contains the timeout value
"init"	initialize the interface with the device
"identify ASSET"	identify the host
"select_file desc_file bin file"	identify desc file and binary file, for ETAS VS-100, these are the root names of the DAMOS file and HEX file, respectively
"monclear"	clear the monitor list
"monadd remote_par asset_var"	add monitor parameter, retrieving remote_par and placing the value in asset_var
"monon"	start monitoring mode
"monoff"	stop monitoring mode
"getpar remote_var asset_var"	get parameter value from remote system
"setpar remote_var asset_var"	set parameter value in remote system
"emergency"	emergency detected by CyFlex

Example Specification:

```
@AK_COMMAND
#start_type stop_code failure_action
AT_START MODE_TERMINATE ELSE_MODE
#instrument name
ASAP
#command key strings
```

```
"timeout asap_to"                        #defines the variable that contains the timeout value

"init"                                   #initialize the interface with the device

"identify ASSET"                         #identify the host

"select_file PL056201 PL056201"          #identify desc file and binary file

"monclear"                               #clear the monitor list

"monadd efps_u_w efps_u_w"               #add monitor parameter

 "monon"                                 #start monitoring mode

"monoff"                                 #stop monitoring mode

 "getpar RTMC_PHI_BIMI_CA phi_ca"        #get parameter value from remote system

"setpar RTMC_PHI_BIMI_CB phi_enbl"       #set parameter value in remote system

"emergency"                              #emergency detected by CyFlex
```

ECM Communications	@ASAP3_ACTIONS

Command used to communicate with an ASAP3 application running on TCP/IP connection.

Keyword:

@ASAP3_ACTIONS

Usage:

Data Fields:

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:

```
@ASAP3_ACTIONS
        #start_code     success_path   fail_path
        AT_START        MODE_TERMINATE /specs/gp/quit
```

Support Tasks	**@ASC**

Send command to asynchronous device.

Keyword:

@ASC

Usage:

Send a command to a device connected to a serial port. The command string will be parsed and converted to a string which the device can interpret based on a configuration file. The configuration file must be in /specs directory and must be named device.cfg, where device is the same as that used in the @ASC specification. For this example, the file would be /specs/pager.cfg.

Data Fields:

start_type	code for when to send the command - options are AT_START, AFTER_STABILITY, EXTERNAL_SYNC
stop_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.
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.
device	basename of the device configuration file.
command	command string

Example Specification:

```
@ASC
        #start_type     stop_path   fail_path   device  command
        AT_START        NONE        NONE        pager   "page 1999W 115 Wh0;"
```

Page number 1999 and send the code 115. This requires the proper specification file for paging in /specs/pager.cfg.

Other Examples:

```
@ASC
        #start_type     stop_path       fail_path       device  command
        AT_START        MODE_TERMINATE  NONE            calterm "run ram"
```

Send a command to a Calterm to have the ECM run from RAM. Terminate the mode when the communication is complete.

```
@ASC
        #start_type     stop_path   fail_path           device  command
        AT_START        NONE        NONE            modem   "hang up"
```

Send a command modem device to hang up.

Specify when to send ATA commands and options for completion and failure.

Keyword:

@ATA_ACTIONS

Usage:

This keyword is used to specify the actions and timing associated with all ATA communications for a test mode. The `start_type` specifies when the commands will be executed, the `stop_path` specifies what action will be taken when all commands have been completed, and the `fail_path` specifies what to do if the communication fails to complete successfully.

Data Fields:

start_type	code for when to send the message - options are AT_START, AFTER_STABILITY
stop_path	code for what action to take when the communication is complete - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.
fail_path	code for what action to take if there is a failure to communicate with the ECM - options include NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.

Example Specification:

```
@ATA_ACTIONS
        #start_type stop_path          fail_path
        AT_START    MODE_TERMINATE     /specs/gp/comm_fail
```

Start sending the ATA commands at the start of the mode, terminate the mode when all are complete, and if there is a failure, go to the test mode specified by the `@ELSE_MODE` keyword.

Notes:

This keyword is used only when there are other ATA communication commands such as `@ATA_SEND`, `@ATA_REQUEST`, and `@ATA_COMMAND_MESSAGE`.

Send a data_command to the ECM.

Keyword:

@ATA_COMMAND_MESSAGE

Usage:

The following command codes are supported.

command_code	function
RUN_FROM_DEV	run from Development (RAM)
FAULT_CODE_ERASE	erase fault codes
STOP_BCST	stop broadcasting
START_PB_BCST	Start public broadcasting
RUN_FROM_EEPROM	run from Default (EEPROM)
STOP_ENGINE	stop engine

Data Fields:

command_code	code for the message

Example Specification:

```
@ATA_COMMAND_MESSAGE
        #command_code
        RUN_FROM_EEPROM
```

Issue the command to run from EEPROM and terminate the mode when a reply is received.

Other Examples:

```
@ATA_COMMAND_MESSAGE
        #command_code
        STOP_BCST
```

Issue a command to the ECM to stop broadcasting.

```
@ATA_COMMAND_MESSAGE
        #command_code
        START_PB_BCST
```

Issue a command to the ECM to start broadcasting.

ECM Communications	**@ATA_MODIFY_BITS**

Toggle a single bit in an ECM variable.

Keyword:

@ATA_MODIFY_BITS

Usage:

The bit number of a particular ECM variable can be set to 0 or 1.

Data Fields:

ecm_variable name	The ECM variable name as defined in an E2M file.
bit_number	a number from 0 to 15. Zero is the most significant bit.
value	0 (off) or 1 (on)

Example:

```
@ATA_MODIFY_BITS
                    #ecm_variable          bit_number              value
                    'ECM_VAR'                  2                     1
```

ECM Communications	@ATA_RAMP

Ramp an ECM value.

Keyword:

@ATA_RAMP

Usage:

The value in the ECM will be updated every second until the `end_target` is reached. If the `ramp_rate` is not specified, then the ramp rate is computed from the mode time. The targets and ramp rate may be constants, variables, or expressions. The ECM variable may be a literal string, or a string variable. The literal string must be surrounded by single quotes. The targets and ramp rate may be constants, variable labels, or computed expressions. The computed expressions must be surrounded by double quotes. Communication of values to the ECM is somewhat different than setting the value of a CyFlex variable. The value is transmitted as a string and has no units associated with it. Because of this, constants in the target or ramp_rate fields should be entered without units. Hex values should be preceded by "0x". See @ATA_SEND for more detailed discussion of format requirements for values sent to the ECM.

As with the @ATA_SEND keyword, each communication with the ECM that is a request to change a value in the ECM is automatically followed by a request to read it back. If the read-back does not agree with the value sent, an error message will be generated.

Data Fields:

`ECM_variable`	`the name of the variable in the ECM or string variable`
`start_target`	`the starting value of the ramp`
`end_target`	`the ending value of the ramp`
`ramp_rate`	`the ramp rate in units/second`

Example Specification:

`@ATA_RAMP`

#ECM_variable	start_target	end_target	ramp_rate(optional)
'USC_FUEL'	200	240	1
'USC_ADV'	RampStart	RampEnd	RampRate
myECMvar	"mystart + inc"	"myend + inc"	

Rate at which ECM values is updated.

Keyword:

@ATA_RAMP_INTERVAL

Usage:

Allows the specification of the rate at which ECM values will be updated during the @ATA_RAMP operation.

Data Fields:

interval	Rate in which an ECM value is updated during the @ATA_RAMP operation

Example Specification:

```
@ATA_RAMP_INTERVAL
        #interval
        1[sec]
```

ECM Communications	@ATA_REQUEST

Read a value from the ECM.

Keyword:

@ATA_REQUEST

Usage:

Send a message to the ECM, requesting the value of a particular ECM variable.

Data Fields:

ECM_variable	the name of the variable in the ECM or a CyFlex string variable which contains the ECM variable name
ASSET_variable	the name of the variable where the result will be placed

Example Specification:

```
@ATA_REQUEST
     #ECM_variable      ASSET_variable
        ‘TVO’           tvo
```

Send a message requesting the value of the TVO variable.

Other Examples:

```
@ATA_REQUEST
     #ECM_variable        ASSET_variable
     ‘TVO’                tvo
     ‘TVC’                tvc
     ‘ALPHA’              alpha
```

Read the values of `tvo`, `tvc`, and `alpha` from the ECM. Terminate the mode when all three values have been read.

Update a variable in the ECM.

Keyword:

@ATA_SEND

Usage:

A message is sent to the ECM via the ATA communication link to modify the value of a particular ECM variable.

The ECM_variable may be a literal string, designated with single quotes, or it may be a string variable label which contains the ECM variable name. The string variable name must not be enclosed in quotes.

The value may be a constant, variable label, or expression. Constants may be expressed as a decimal value or as hex, as appropriate for the particular variable. Hex values must be preceded by the characters 0x (0xFF) or enclosed in single quotes (‘FF’).

If the value is expressed as a variable label, then the variable label cannot begin with "0x". The value sent to the ECM for a variable will be a decimal number unless the variable is a string variable. For a string variable, the contents of the variable are sent as is.

If the value is an expression, then the value sent to the ECM will always be a decimal number.

Each message that is sent to the ECM to change the value of an ECM variable is automatically followed by a message to read the value of the variable. If the value read back does not match the value sent, then a single retry is attempted. If the value does not agree a second time, an error message is generated.

Data Fields:

`ECM_variable`	`the name of the variable in the ECM or a string variable containing the name`
`value`	`the new value of the ECM variable or a variable containing the value, or an expression which computes the value`

Example Specification:

```
@ATA_SEND
        #ECM_variable           value
        ‘USC_FUEL’              350
```

Change the value of the variable `USC_FUEL` to 350.

Other Examples:

```
@ATA_SEND
        #ECM_variable           value
        # change the value of USC_FUEL to the value of the CyFlex variable,
myTVO
        ‘USC_FUEL’              myTVO

        # change the value of USC_ADV to the sum of CyFlex variables, myTVC and
        # TVCincrement
        ‘USC_ADV’               "myTVC + TVCincrement"

        # change the value of CYL_CUTM to hex FF
        ‘CYL_CUTM’              ‘FF’

        # change the value of CYL_CUTO to hex EC
        ‘CYL_CUTO’              0xEC

        # change the value of the ECM variable which is contained in the CyFlex string
        # variable, myECMvar to the value contained in the CyFlex variable myECMvalue
        myECMvar        myECMvalue
```

Modify several variables in the ECM.

Support Tasks	**@AUXILIARY_TASK**

Start an auxiliary support task.

Keyword:

@AUXILIARY_TASK

Usage:

This keyword is used to spawn a special type of support task called an "auxiliary" task. This task can perform some special support function for the test scheduler, but must be designed to accept start, and stop messages and must return a reply message which signals success or failure.

Data Fields:

`start_type`	`code for when to send a "start" signal to the task. Options are AT_START, AFTER_STABILITY, EXTERNAL_SYNC, NONE`
`stop_path`	`code for what action to take when the auxiliary task completes its function. Options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.`
`fail_path`	`code for what action to take when the auxiliary task signals that is has failed to accomplish its function. Options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.`
`task_pathname`	`the auxiliary task name`
`command_line`	`arguments for the auxiliary task (enclose the command line in quotes)`

Example Specification:

```
@AUXILIARY_TASK
     #start_type                stop_path              fail_path

     AT_START                   MODE_TERMINATE         /specs/gp/gp_idle

     #task_pathname             command_line

     /asset/bin/engine_start    "/specs/starter.407"
```

Spawn the engine start task with specifications for how to start in file /specs/starter.407. Branch to the gp_idle procedure if the engine fails to start.

Notes:

The command line must be enclosed in double quotes and may contain more than one argument, depending on the particular task.

Other Examples:

```
@AUXILIARY_TASK
      #start_type                stop_path               fail_path

      AT_START                   MODE_TERMINATE          23

      #task_pathname             command_line

      /asset/bin/engine_start    "/specs/starter.407"
```

Spawn the `engine_start` task. Jump to mode 23 if the engine fails to start

```
@AUXILIARY_TASK
#start_type stop_type failure action

AT_START MODE_TERMINATE ELSE_MODE

/asset/bin/vrbl_to_file "/specs/vrbls_tvo READ run_index"
```

#start_type	stop_path	fail_path
AT_START	MODE_TERMINATE	ELSE_MODE
#task_pathname	command_line	
/asset/bin/vrbl_to_file	"/specs/vrbls_tvo Read run_index"	

Spawn the variable to file task and branch to the ELSE mode if there is a failure.

Support Tasks	**@BACKGROUND_TASK**

Execute a command or task in the background.

Keyword:

@BACKGROUND_TASK

Usage:

Any command or task can be executed in the background. No synchronization or error checking is performed.

Data Fields:

start_type	code for when to execute the command - options are AT_START, AFTER_STABILITY
command	the command string (enclose in quotes)

Example Specification:

```
@BACKGROUND_TASK
#start_type            command
AT_START               "/asset/bin/meterlog"
```

Execute the meterlog command.

Notes:

The command string must be enclosed in double quotes and may include several arguments.

Other Examples:

```
@BACKGROUND_TASK
        #start_type         command
        AT_START            "/specs/cmds/my_script"
        AT_START            "rm /data/PC_format/my_log"
        AFTER_STABILITY     "cp /data/PC_format/my_log /dos/a/mylog"
```

Issue the hotkey command and remove the `my_log` file at mode start. After stabilization is complete remove copy the mylog file to a DOS floppy in drive a:

Specify the tolerance of a control variable.

Keyword:

@CONTROL_TOLERANCE

Usage:

This specification sets the tolerance parameter for a control variable. The value is not modified when the mode terminates.

Data Fields:

`variable`	`variable label or macro for speed and torque. Options are SPEED_VAR and TORQUE_VAR.`
`tolerance`	`control tolerance (units required)`

Example Specification:

```
@CONTROL_TOLERANCE
        #variable               tolerance
        SPEED_VAR               50[rpm]
        TORQUE_VAR              20[lb_ft]
        int_man_f               5[deg_f]
```

Creation	**@CREATE_EVENT**

Create events that will be used during a specific gp_test.

Keyword:

@CREATE_EVENT

Usage:

This keyword was created in response to the amount of volume and complexity that has been created in limit_specs.NNN. Sometimes it is advantageous to have created events that exist only during the duration of a specific test.

Data Fields:

`event_name`	`The name of the event to be created`

Example Specification:

```
@CREATE_EVENT
        #event_name
        my_event1
        my_event2
The events my_event1 and my_event2 are created with this keyword. Up to 16
events can be created per test procedure
```

Notes:

The events in this specification must be unique or an error will occur.

The events created from a @CREATE_EVENTS are placed in the Engineering Units Buffer when an "NT" command is issued and exist until the next "nt" occurs. Whenever an "nt" is issued, all of the existing events, which were created by that instance of gp_test, are destroyed. Then when the new files are read, any events specified in those procedures are created. If you slay "gp_test", the created events will remain until a new "nt" is received.

Special Note:

The keyword @CREATE_EVENT is similar to @GLOBAL_EVENTS and @REGISTERED_EVENTS. The @CREATE_EVENT must be placed in the header section of a test procedure file somewhere between the "start_mode" and the first @MODE.

Creation	**@CREATE_EXPRESSION**

Create computed expressions that will be used during a specific gp_test.

Keyword:

@CREATE_EXPRESSION

Usage:

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.

Data Fields:

variable	The variable name of label used
type	The variable can be REAL, INTEGER, LOGICAL or 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 doesn't 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".

Special 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.

Creation	**@CREATE_TRANSITION_EVENTS**

Create associated events to a logical variable that will be used during a specific gp_test.

Keyword:

@CREATE_TRANSITION_EVENTS

Usage:

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 create logical events that exist only during the duration of a specific gp_test.

Data Fields:

logical_variable_label	The logical variable name. Its transition status will spawn true and false events
true_event_name	The name of the event that is spawned when the logical variable goes from FALSE to TRUE
false_event_name	The name of the event that is spawned when the logical variable goes from TRUE to FALSE

Example Specification:

```
@CREATE_TRANSITION_EVENT
     #logical_variable_label     true_event_name     false_event_name
            mystate                my_state_on         my_state_off
```

The variable mystate will spawn an event called mystate_on when the logical goes from an OFF to ON state. The event mystate_off will be spawned when the variable mystate goes from a logical ON to OFF. Up to 16 transition events can be created per test procedure.

Notes:

The logical_variable_label must be created using the @CREATE_VAR keyword. The transition events created exist in the Engineering Units Buffer when the "nt" command is issued and remain there until the next "nt" command is issued. If you slay "gp_test", the created event will remain until a new "nt" is received.

Special Note:

The keyword @CREATE_TRANISITION_EVENT is similar to @GLOBAL_EVENTS and @REGISTERED_EVENTS. The @CREATE_TRANISITION_EVENT must be placed in the header section of a test procedure file somewhere between the "start_mode" and the first @MODE.

Creation	**@CREATE_VAR**

Create variables that will be used during a specific gp_test.

Keyword:

@CREATE_VAR

Usage:

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 variables that exist only during the duration of a specific gp_test.

Data Fields:

variable	The variable name or label used
type	The variable can be REAL, INTEGER, LOGICAL or STRING
units	The type of units to be used with the created variable
initial value	The initial value to be used with the created variable

Example Specification:

```
@CREATE_VAR
     #(up to 16 variables per procedure)
     #label     type     units    initial_value
     mynewx     REAL     psi            -
     count      INT      none           2
     mystate    LOGI     none          OFF
     mysting    STING     -        "up to 80 characters"
```

The variable mynewx is created as a REAL with psi as its units with no initial value. The variable count is created as an INTEGER with units set to none and an initial value of 2. The variable mystate becomes a LOGICAL set to an initial value of OFF. The variable mystring is created as a STRING that can include a message up to 80 characters long.

Notes:

The variables in this specification must be unique or an error will occur. The variables created from a @CREATE_VAR are placed in the Engineering Units Buffer when an "nt" command is issued and exist until the next "nt" occurs. Whenever an "nt" is issued, all of the existing variables, which were created by that instance of gp_test, are destroyed. When the new files are read, any variables specified in those procedures are created. Note that, unlike gen_labels.NNN, the previous value of the variable is not preserved. The '-' symbol means 0 or OFF for the initial value. It does not mean "keep the current value".

Special Note:

The keyword @CREATE_VAR is similar to @GLOBAL_EVENTS and @REGISTERED_EVENTS. The @CREATE_VAR must be placed in the header section of a test procedure file somewhere between the "start_mode" and the first @MODE.

ECM Communications	**@CUTY_ACTIONS**

Command used to communicate with a CUTY application running on TCP/IP connection.

Keyword:

@CUTY_ACTIONS

Usage:

Data Fields:

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:

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

CyberApps	@CYBER

Issue commands to change the operating parameters of a Cyber Application.

Keyword:

@CYBER

Usage:

Issue a command to the Cyber application or to the CyberServer. The command code will determine the action taken.

Data Fields:

command	A command key - see the table below.
name	The system or component name
value	The system or component value

Cyber Commands	Arguments
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 above commands configure CyberTruck to use the 38th Street route and set the truck mass to 75000 pounds.

CyberApps	@CYBER_ACTIONS

Issue action commands to the @CYBER keyword.

Keyword:

@CYBER_ACTIONS

Usage:

This keyword directs when the command will take place during the mode. If the commands fail, then an alternate path may be taken.

Data Fields:

start_code	At what point during the mode should execution of the commands begin
success_path	If all commands are successful, then…
failure_path	If a command fails, then…

start_code_options	
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. (See @STABILITY_SPECS)

success_path and failure_path options	
NULL	The 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 Specification:

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

The above 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.

Specify the dyno controller mode & open_loop position.

Keyword:

@DYNO

Usage:

This specification selects the dyno controller mode to be either OPEN_LOOP or CLOSED_LOOP. If the mode is CLOSED_LOOP, then only the control_mode data field is required and the target values are ignored. If the mode is OPEN_LOOP, then the start_target must be specified. The end_target is optional. If the start and end targets are both entered and are different, then the controller output will be ramped linearly over the mode time_out interval. The targets are always expressed as percent of full scale output.

If CLOSED_LOOP is specified, then the dyno is controlled as specified with the @ENGINE_CONTROL_MODE keyword.

Data Fields:

control_mode	open/closed loop (OPEN_LOOP or CLOSED_LOOP)
start_target	used only if mode is open_loop, units must be % of full scale
end_target	used only if mode is open_loop and ramping is required, units must be % of full scale

Example Specification:

```
@DYNO
#control_mode           start_target            end_target
OPEN_LOOP               0[%]                    50[%]
```

Ramp the dyno excitation from 0 to 50% over the mode interval.

Notes:

The start and end targets may be constants, variable labels, or expressions. Expressions must be enclosed in double quotes. Units are required for all constants.

Other Examples:

```
@DYNO
        #control_mode           start_target            end_target
        OPEN_LOOP               0[%]
```

Turn the dyno excitation off.

```
@DYNO
        #control_mode           start_target            end_target
        CLOSED_LOOP
```

Set the dyno controller to closed loop mode.

Branching	**@ELSE_MODE**

Specify an alternate path to execute next.

Keyword:

@ELSE_MODE

Usage:

This specified path is an alternate to the normal next_mode. It will be executed only if certain conditions are met, such as the failure of the conditional tests. Use the "RETURN" macro to return to a calling procedure.

Data Fields:

`mode_number`	`the mode or test procedure to execute as an alternate next mode`

Example Specification:

```
@ELSE_MODE
        #mode_number/procedure
        91
```

Use mode 91 as the next_mode if conditions dictate.

Notes:

An alternate path is usually specified as an option with the "ELSE_MODE" macro for fail_path data fields. See @AUXILIARY_TASK for an example.

Also see the options for @STABILITY_ACTION. The alternate mode can be taken after stabilization is complete.

Other Examples:

```
@ELSE_MODE
        #mode_number/procedure
        RETURN
```

Return to the calling procedure.

```
@ELSE_MODE
        #mode_number/procedure
        /specs/gp/gp_shutdown
```

Execute the gp_shutdown procedure as an alternate path.

Outputs	@EMAIL

Send an email message or pager message.

Keyword:

@EMAIL

Usage:

An email or page may be generated from a test procedure. This is usually used to inform someone of the progress of a test or that something has gone wrong with the test, such as a premature shutdown. The specification requires a "receiver" and a "message". The message may be a short string such as "test done in tc115", or can be derived from a file. The "receiver" may be any email address such as Joe_Engineer@notesbridge.cummins.com, or JoeAtHome@aol.com. Three paging systems are supported with email type domain names.

1. Cummins 8800 Pager System

xxxx.cummins@assetpager.ctc.cummins.com (where xxxx is the pager number)

The message should be a numerical string that can be displayed by the pager such as "115".

2. Indiana Paging Network

xxxxxxx.indiana@assetpager.ctc.cummins.com

3. Skytel

xxxxxxx.skytel@assetpager.ctc.cummins.com

Data Fields:

start_code	code for when to send the message. Options are AT_START, AT_END, and AFTER_STABILITY
receiver	The intended recipients of the message. This may be a literal string, the label of a string variable, or a computed expression. The string may contain multiple receivers.
message	The message to be sent. This must be either a quoted literal string or a filename.

Example Specification:

There may be up to 4 separate specifications per test mode.

```
@EMAIL
        #start_code   receiver                               message
        AT_START   'Len_Logterman@notesbridge.cummins.com'    "help, I have fallen"
        AT_START   User_email                               /specs/canned_msg
```

Send the message "help, I have fallen" to the Lotus Notes account of Len Logterman. The macro 'NOTES' can be substitued for 'notesbridge.cummins.com'. Send the message contained in the file /specs/canned_msg to the address defined by the string variable User_email.

Notes:

Note that the variable User_email is initialized in the /specs/engine_specs.NNN file and should be maintained by the user of the test system. It should always contain the email addresses of the persons responsible for the test object (engine).

Other Examples:

```
@EMAIL
        #start_code    receiver                            message
        AT_START           '1400.cummins.assetpager.ctc.cummins.com'
"77500"
        AT_START        "User_email     + ' dick '
        /specs/canned_msg
```

Send the page 77500 to pager #1400.

Send the contents of the file /specs/canned_msg to "dick" on the Test System network email system and to whatever email addresses are contained in the string variable User_email.

Engine Control	@ENGINE_CONTROL_MODE

Specify the method of engine control.

Keyword:

@ENGINE_CONTROL_MODE

Usage:

There are 6 possible methods of controlling an engine/dyno combination that are supported by CyFlex. The methods differ from one another by which variable is the feedback for either the throttle or dyno controller.

Options are:

```
1 or DYNO_DYNO_TORQUE_THROT_SPEED

2 or DYNO_NET_TORQUE_THROT_SPEED

3 or DYNO_OTHER_THROT_SPEED

4 or DYNO_SPEED_THROT_GROSS_TORQUE

5 or DYNO_SPEED_THROT_NET_TORQUE

6 or DYNO_SPEED_THROT_OTHER
```

Data Fields:

control_mode

Example Specification:

```
@ENGINE_CONTROL_MODE
        #mode
        4
```

Set the control mode to 4, i.e., have the dyno control speed and have the throttle control gross torque.

Notes:

mode control combination

```
1 dyno controlling dyno_torque, throttle controlling speed
2 dyno controlling net_torque, throttle controlling speed
3 dyno controlling some other variable, throttle controlling speed
4 dyno controlling speed, throttle controlling gross_torque
5 dyno controlling speed, throttle controlling net_torque
6 dyno controlling speed, throttle controlling some other variable
```

For non-motoring dynos, operation of the engine above idle speed with no load, requires that the throttle be used for speed control. This requires that the engine_control_mode be changed to 2.

For high gain speed-control fuel pump governors, mode 2 may also be recommended.

Other Examples:

```
@ENGINE_CONTROL_MODE
        #mode
        2
```

Have the throttle control speed, dyno control torque.

```
@ENGINE_CONTROL_MODE
        #mode
        DYNO_SPEED_THROT_OTHER
```

Have the dyno control speed and the throttle control manifold pressure. The other variable, in this case, manifold pressure is defined in the ctrl_specs.NNN file.

Engine Control	@FEED_FORWARD

Specify the feed-forward characteristics of a controller.

Keyword:

@FEED_FORWARD

Usage:

This specification is used to modify the feed-forward characteristics of a controller, including turning feed-forward on and off, selecting the feed-forward variable and setting the feed-forward gain. The selections remain in effect when the current mode is terminated.

Data Fields:

loop	an index or variable label
active_flag	flag indicating whether feed-forward is ON or OFF
FF_label	the feed-forward variable
gain	the feed-forward gain (constant, variable, or expression)

Example Specification:

```
@FEED_FORWARD
        #loop               active_flag  FF_label   gain
        DYNO_CTRLER         ON           power      0.4
        THROT_CTRLER        OFF
        int_man_t           ON           power      .01
```

Flowbench	@FLOWBENCH

Command designed to control the TIMPELMAN head flow rigs.

Keyword:

@FLOWBENCH

Usage:

Data Fields:

device	Is one of the following: VALVE_OPENER, TURNING_TABLE, FRAME_DRIVE, BLOWER
command	Is one of the following: INIT, SET_REF, MOVE_POSITIVE_LIMIT, MOVE_NEGATIVE_LIMIT, FIRST_STEP, NEXT_STEP, MOVE_TO_STEP, MOVE_TO_ABSOLUTE, MOVE_TO_RELATIVE, STOP_MOVEMENT, STOP_BLOWER, START_BLOWER, RETRY_BLOWER, RESET_FREQUENCY, RESET_PM
return variable	Label of an INTEGER variable that will contain the value of the position manager response to the last command issued. The last command issued will be any command that gets an error return from the position manager on the last command in the list. Any negative response from the position manager will result in the "failure_path" being taken. For either success or failure, the return variable will contain the position manager response for the last command issued.
value(optional)	The value field can be a constant, variable label, or computed expression. It is assumed that the units of the value sent to the position manager are to be [mm].

Example Specification:

```
@FLOWBENCH
        #device        command        return variable      value(optional)
        VALVE_OPENER   SET_REF        retn
        VALVE_OPENER   MOVE_RELATIVE  retn_valve           10[mm]
```

Flowbench	**@FLOWBENCH_ACTIONS**

Command designed to control the TIMPELMAN head flow rigs.

Keyword:

@FLOWBENCH_ACTIONS

Usage:

Data Fields:

start_code	code for when to send the message - options are AT_START, AFTER_STABILITY - default is AT_START
success_path	code for what action to take when the 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 there is a - options include NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname. - default is NONE

Example Specification:

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

Data Acquisition	**@FR_LOG_FILE**

Specify the data file for logging fuel readings.

Keyword:

@FR_LOG_FILE

Usage:

This keyword defines the data file that will be used to log fuel reading data in a columnar format that is compatible with spreadsheets.

Data Fields:

`file_pathname`	`the full pathname of the data file`

Example Specification:

```
@FR_LOG_FILE
        #file_pathname
        /data/fuel_log/fr_fuel_data
```

Log fuel reading data to the file /data/fuel_log/fr_fuel_data if the fr_log_enab flag is ON.

Notes:

The logging of fuel reading data can be turned on and off by setting the state of the variable fr_log_enab. This data logging operation is entirely separate from PAM data files and transfers.

To enable logging, use:

```
set fr_log_enab ON
```

Data Acquisition	@FUEL_READING

Take fuel readings.

Keyword:

@FUEL_READING

Usage:

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.

Data Fields:

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, or 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

Example Specification:

```
@FUEL_READING
     #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.

Notes:

Specifying a non-zero desired_time will change the value of the target_fr_tim variable.

Other Examples:

```
@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.

Data Acquisition	**@FUEL_READING_STATS**

Take fuel samples until specified statistical criteria are met. This keyword is similar to @FUEL_READING except that the number of readings taken may be variable and will depend upon the specified statistical confidence requirements. Also, it is optional to have a single composite datapoint transmitted to PAM which represents the mean value of the set of readings which meet the confidence criteria. In addition, data which is grossly in error may be discarded from the set as outliers.

Keyword:

@FUEL_READING_STATS

Usage:

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.

Data Fields:

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, or a procedure file pathname.
number_readings	the maximum number of fuel readings - if reached, data set is considered complete - a minimum of 3 readings will be taken
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
dp_storage_method	flag to save all readings as datapoints or only the composite average - ALL=save all, ONE=composite only

outlier_significance	The probability of erroneously rejecting a good observation. A value of 0.01 would mean that there is a 1% chance of rejecting a good reading. A low significance level such as 0.01 is recommended. Levels greater than 0.05 should not be common practice.
deviation_min	a minimum standard deviation from the mean to be considered for outlier evaluation

variable	a variable for which confidence criteria will be evaluated
confidence_interval	an error band for the variable
confidence_level	a probability that the maximum error lies within the specified confidence_interval
target	(not used at this time)

Example Specification:

```
@FUEL_READING_STATS
#start_type          stop_path
 AT_START          MODE_TERMINATE
#number_of_readings     interval     extern_sync_event          desired_time
     4[none]           0.00[sec]             -                    2.5[min]
#save_type [ ALL/ONE ]     #outlier_significance     min_deviation
         ALL                        .0[none]           .005[none]
#up to 32 variables may be listed
#variable          confidence_interval         confidence_level
target_value
FR_BSFC              .001[lb/hp-hr]              .95[none]
FR_RPM                   5[rpm]                  .95[none]
speed_setpt
```

Request up to 10 fuel readings after stabilization is complete. Terminate the mode when enough readings have been to meet all confidence criteria. Save all the individual readings as datapoints, unless they have identified as outliers.

Data Acquisition	@FUEL_READING_SYNC

This function provides the ability 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.

Keyword:

@FUEL_READING_SYNC

Usage:

This keyword allows multiple processes to be synchronized with fuel readings when multiple fuel readings have been requested in a mode. 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 overriden by the specified timeout (0 timeout indicates no timer). The timeout for a particular line doesn't 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 also, that the maximum specified delay for this entire process is the value of the variable "FR_wrte_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 ASSET Gazette.6b.97-"Variables, Events, and Processes associated with fuel readings"

Data Fields:

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.

The "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.

Engine Controls	@GET_FF_GAIN

Used to retrieve the current value of the feedforward gain.

Keyword:

@GET_FF_GAIN

Usage:

Data Fields:

loop	The label of a feedback variable if the loop is a user loop or one of the following macros for dyno or throttle control: DYNO_CTRLER, THROT_CTRLER, SECOND_DYNO_CTRLER
gain_label	Label for variable to retrieve information about

Example Specification:

```
@GET_FF_GAIN
        #loop          gain_label
        int_man_t      int_ff_gain
        DYNO_CTRLER    dyno_ff_gain
```

Engine Controls	**@GET_PID_GAINS**

Used to retrieve the current value of the gains and store them in real variables.

Keyword:

@GET_PID_GAINS

Usage:

Data Fields:

`loop`	
`prop_var`	
`integral_var`	
`derivative_var`	

Example Specification:

```
@GET_PID_GAINS
        #loop                     prop_var  integral_var  derivative_var
        DYNO_SPEED_GAINS          dyno_pg   dyno_ig       dyno_dg
        THROT_GROSS_TORQUE_GAINS  thrg_pg   thrg_ig
        int_man_t                 int_pg    int_ig        int_dg
```

Branching	@IF_FALSE

A list of variables or expressions which must be FALSE to execute the mode.

Keyword:

@IF_FALSE

Usage:

This keyword is followed by 1 or more labels of logical variables or expressions which must all be FALSE if the mode is to be executed. If any one of the variables is TRUE then the execution path is determined by the @ELSE_MODE specification, if there is one. If no @ELSE_MODE is specified then execution proceeds to the default_next_mode specified in with the @MODE keyword.

Data Fields:

variable_labels	valid labels of logical variables

Example Specification:

```
@IF_FALSE
        #variable_labels/expressions
        SpeedLT400
```

If SpeedLT400 is FALSE, execute this test mode.

Branching	**@IF_TRUE**

A list of variables or expressions which must be TRUE to execute the mode.

Keyword:

@IF_TRUE

Usage:

This keyword is followed by 1 or more labels of logical variables which must all be TRUE if the mode is to be executed. If any one of the variables is FALSE then the path is determined by the @ELSE_MODE specification, if there is one. If no @ELSE_MODE is specified then execution proceeds to the default_next_mode specified in with the @MODE keyword.

Data Fields:

`variable_labels`	`valid labels of logical variables or expressions`

Example Specification:

```
@IF_TRUE
        #variable_labels/expressions
        SpeedGT800
        tvo350
        "RPM < 1000[rpm]"
```

If both SpeedGT800 and tvo350 are TRUE and the RPM variable is less than 1000 rpm, then execute this test mode.

ECM Communications	@KESET_ACTIONS

Command used to communicate with a KESET application running on TCP/IP connection.

Keyword:

@KESET_ACTIONS

Usage:

Data Fields:

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:

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

ECM Communications	@KESET_GET

Command used to get a value for a specific variable from the ECM.

Keyword:

@KESET_GET

Usage:

Data Fields:

ECM_name	The registered name for a kesettcp instance
ECM_variable	This is a string and may be a constant, variable label, or computed expression - a constant must be expressed as a literal string with single quotes. Note that the ECM_variable label is a string, so that the label should normally be enclosed with single quotes such as 'RUN_LOC'. If the ECM_variable is not quoted, then gp_test will interpret that as the label of a CyFlex string variable which contains the ECM_variable name
ASSET_label	The label of the variable where the result will be placed

Example Specification:

```
@KESET_GET
        #ECM_name   ECM_variable               ASSSET_label
         KTCP      "'injector' + cyl_number"  fixed_label
```

Command used to set a value for a specific variable from the ECM

Keyword:

@KESET_SET

Usage:

Data Fields:

ECM_name	The registered name for a kesettcp instance
ECM_variable	This is a string and may be a constant, variable label, or computed expression - a constant must be expressed as a literal string with single quotes. Note that the ECM_variable label is a string, so that the label should normally be enclosed with single quotes such as 'RUN_LOC'. If the ECM_variable is not quoted, then gp_test will interpret that as the label of a CyFlex string variable which contains the ECM_variable name
value	This may be a constant, variable label, or computed expression

Example Specification:

```
@KESET_SET
        #ECM_name   ECM_variable  value
         KTCP       'Some_label'  100[none]
```

Mode Termination	@LIMIT_SPECS

Set a limit that will terminate the mode.

Keyword:

@LIMIT_SPECS

Usage:

Specifies a list of variables which may have limits set on them. Up to 32 variable specifications may be used per keyword. If the limit is violated for the period specified, then the mode is terminated. If the next_path field is 0 or "-", then the default_next_mode is executed, otherwise control is passed to the mode or test procedure specified for the limit that was violated. The limit value may be expressed as a constant, variable label, or computed expression.

Data Fields:

variable	the real variable on which to set the limit
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
next_path	an optional path to execute if this limit is violated

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.

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.

Other Examples:

```
@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.

Mode Termination	@LIMIT_SPECS_ALL

Specify a number of limits that will terminate the mode only if all are simultaneously violated.

Keyword:

@LIMIT_SPECS_ALL

Usage:

Specifies 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, otherwise control is passed to the mode or test procedure specified for the limit that was violated. The limit value may be expressed as a constant, variable label, or computed expression.

Data Fields:

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
next_path	an optional path to execute if this limit is violated

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.

Branching & Looping	@LOOP_CONTROL

Specify a repetitive looping operation.

Keyword:

@LOOP_CONTROL

Usage:

This specification is used to create loops of execution. It is placed in the last test mode of the loop. There may be multiple loops in a procedure file and they may be nested. The loop counter variable is optional. The loop count is zeroed when a test procedure is started. The num_repeats field may be a constant, variable, or expression.

Data Fields:

num_repeats	the number of times the loop is to be repeated before proceeding to the next_mode specified in @MODE_CONTROL
next_loop_mode	the first test mode of the loop
loop_counter	a variable which can be used to display or log the current loop count

Example Specification:

```
@LOOP_CONTROL
        #num_repeats   next_loop_mode      loop_counter
        10             97                  loop1
```

This is the last mode of a loop beginning at mode 97. The loop will be executed 10 times. The current loop count will be placed in the variable loop1.

Notes:

If a loop_counter variable is specified, it must already exist. It is usually created in the gen_labels.NNN specification file.

Other Examples:

```
@LOOP_CONTROL
        #num_repeats   next_loop_mode   loop_counter
        5              2
```

Loop back to mode 2, 5 times before proceeding to the next mode.

```
@LOOP_CONTROL
        #num_repeats   next_loop_mode      loop_counter
        cyc_cnt        97                  lp3
```

Loop back to mode 97, a number of times determined by the value of the variable cyc_cnt before proceeding to the next mode.

Timing & Branching	**@MODE**

Specify mode number, mode time, next mode and description.

Keyword:

@MODE

Usage:

This keyword is used to declare the maximum time for a test mode and the next mode to execute when this mode is complete. This keyword must be present for every mode. A zero time-out or dash, "-", indicates that the mode has indefinite length and will be terminated by some means other than a simple time out. The timeout field may be either a constant, variable label, or computed expression.

Data Fields:

mode_number	(1-99)
time-out	maximum time for the mode
default_next_mode	the next mode to execute when this mode is complete
description	60 character description of the mode

Example Specification:

```
@MODE
        #mode_number   time_out   default_next_mode
        93             30[sec]    54
        #description
        shut the engine down
```

Spend 30 seconds in this mode and then jump to mode 54.

Notes:

Use RETURN for the default_next_mode to return from a sub-procedure to the calling procedure.

The mode description can be displayed on the monitor screen in any display group. The file /specs/gp/gp_header must contain the definition for which display string will be used for the description. This is usually TEST_DESC. If multiple copies of the test scheduler are operating as might be the case if the computer is controlling two engines, then there will be two header files in use and each must have a different display string specified. The second version would usually use TEST_DESC_2. See chapter 1 of the Test Scheduler manual for a description of the header file.

Other Examples:

```
@MODE
        #mode_number   time_out   default_next_mode
        93             my_time    54
        mode 93
```

Timing & Branching	@NO_RUN_PROCEDURE

Forces a procedure to be read into memory before it is actually executed.

Keyword:

@NO_RUN_PROCEDURE

Usage:

This is used to force a procedure to be read into memory before it is actually executed for cases where the procedure name is a variable or is derived from a vrbl file. In cases such as that, the name of a procedure is not actually known when the test is first started (when 'nt' is issued), and it could take several seconds to read it during runtime. By forcing the read prior to start of the test, this non0realtime issue can be avoided.

Data Fields:

pathname	Name of the test procedure file to be read and loaded into memory without running it in this mode.

Example Specification:

```
@NO_RUN_PROCEDURE
        #pathname
         proc_string
```

Engine Control	**@OTHER_CTRL_VAR**

Specify the "other" control variable target.

Keyword:

@OTHER_CTRL_VAR

Usage:

This specification selects the reference value for the "other" control variable. This may be the setpoint for the throttle loop, depending on the engine control mode. It is meaningful only if the throttle controller is in closed loop mode and the engine control mode is 6. The end_target and the ramp_rate are optional. If the end target specified is different from start target and the ramp rate is not specified, then the ramp rate is computed from the start and end targets and the mode timeout value.

Data Fields:

start_target	the reference value at the start of the mode
end_target	optional reference value at the end of the mode
ramp_rate	optional rate at which to ramp from the start to end target values

Example Specification:

```
@OTHER_CTRL_VAR
        #start_target       end_target  ramp_rate
        500[deg_f]          600[deg_f]
```

Ramp the "other" control variable from 500 to 600 degrees over the mode interval. This example would be appropriate for something like a control of turbine inlet temperature by the throttle, with the dyno controlling engine speed.

Notes:

The data fields may be constants, variable labels, or expressions. Constants must have units. The units must be those of the control variable that has been specified in the ctrl_specs.NNN file. The units of ramp_rate are entered in the same units. The denominator is assumed to be seconds. For example, 10[deg_f] would specify a ramp rate of 10 deg_f/sec.

If the ramp rate is specified such that the end target is reached before the mode terminates, then the ramping stops when the end target is reached.

If the ramp rate is specified such that the end target is not achieved when the mode terminates, the ramping may continues unless the next mode modifies the speed target.

Other Examples:

```
@OTHER_CTRL_VAR
        #start_target           end_target      ramp_rate
        30[in_hg]
```

Set the target pressure to 30 inches of mercury. This might be appropriate if the throttle is being used to control boost pressure.

```
@OTHER_CTRL_VAR
        #start_target           end_target      ramp_rate
        boost_tar               10[psi]         1[psi]
```

Ramp from the value of the "boost_tar" variable to 10 psi pressure at a rate of 1 psi/sec.

Engine Control	**@OUT_CHAN_CONFIG**

Reconfigure a control output channel.

Keyword:

@OUT_CHAN_CONFIG

Usage:

Data Fields:

`type`	
`source`	
`chan/out_label`	
`bias`	
`span/gain`	
`filter`	

Example Specification:

```
@OUT_CHAN_CONFIG
        #type  source       chan/out_label  bias  span/gain  filter
        AO     Dyno_CM      17              0      100        0
        RV     pp_cyl_p_CM  prbs_out        0       2         0
```

Notes:

This is generally used to modify the bias or gain of a controller output signal. (see ctrl_specs.nnn for current configuration)

Data Acquisition	**@PAM_DATAPOINT**

Create a datapoint without forcing a fuel reading.

Keyword:

@PAM_DATAPOINT

Usage:

Take one or more datapoints during this test mode. If the desired_time is 0 or "-", the time specified by the variable target_fr_tim will be used. This is identical for @FUEL_READING except that the actual fuel sample is not taken.

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

Data Fields:

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 support task completes its function. Options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.
number_readings	the number of datapoints to request
interval	the time between requests (when number_readings > 1)
sync_event	an event name for external synchronization
desired_time	the desired sample time

Example Specification:

```
@PAM_DATAPOINT
        #start_type        stop_path
        AFTER_STABILITY    MODE_TERMINATE
        #number_readings   interval   sync_event    desired_time
        1                  0.0[sec]   -             0[sec]
```

Request 1 datapoint after stabilization is complete. Terminate the mode when the data collection is complete.

Notes:

Specifying a non-zero desired_time will change the value of the target_fr_tim variable.

Either @FUEL_READING or @PAM_DATAPOINT may be used in a particular test mode, but not both.

Other Examples:

```
@PAM_DATAPOINT
   #start_type         stop_path
   AFTER_STABILITY     MODE_TERMINATE
   #number_readings    interval   sync_event   desired_time
   3                   fr_int     -            30[s]
```

Take 3 datapoints at an interval determined by the value of the fr_int variable, each 30 seconds long. Terminate the test mode when all 3 datapoints have been completed.

Used to automatically generate a name or several names for a particular fuel reading or set of fuel readings.

Keyword:

@PAM_GROUP_LIST

Usage:

The PAM database allows groups of datapoints to be "named" and then retrieved by the group name. For example, all of the datapoints collected at rated speed might be named "1800rated", or repeat datapoints collected for quality assurance purposes might be named "quality". This can make the extraction of a set of data much simpler.

Nine string variable names have been reserved to stored group names. They are PAM_grp_1 through PAM_grp_9.

The @PAM_GROUP_LIST keyword should be used in the same mode where the fuel reading is taken, since the evaluation of computed expressions will be done at the mode start. There may be up to 9 group names assigned to a datapoint. The specification may be the label of a string variable, a literal string (enclosed with single quotes) or a computed expression (enclosed with double quotes). You may enter the different group names on 1 or more lines.

Data Fields:

Example Specification:

```
@PAM_GROUP_LIST
        #list of string variables for PAM group name(s)
        #(1 or more lines - up to 9 labels per line)
        #the entry may be a label, leteral string, or computed string
        PAM_grp_1   PAM_grp_2   PAM_grp_3   my_group
        'groupx'    "PAM_grp_1 + test_mode"
```

Notes:

The group names in PAM are limited to 25 characters, even though a string variable may be 80 characters long. Only the first 25 characters of the string will be used for the group name.

Outputs	@PARAMETERS

Modify a CyFlex variable.

Keyword:

@PARAMETERS

Usage:

This specification allows any CyFlex variable to be modified during the course of a test. A list of variables may be specified. For logical variables, if the value is a constant it should be expressed as ON or OFF. Real variables must be specified with units.

Data Fields:

start_type	code for when to set the value - options are AT_START, AFTER_STABILITY, AT_END
variable	a CyFlex variable label.
value	the new value of the variable. This may be a constant, or may be obtained from another variable, or from a computed expression.

Example Specification:

```
@PARAMETERS
        #start_type     variable        value
        AT_START                tvo350  ON
```

Set the logical variable, tvo350 to the TRUE state when the test mode starts.

Other Examples:

```
@PARAMETERS
        #start_type     variable             value
        AT_START        tvo350  "if engine_spd > 400[rpm] TRUE else FALSE"
        AT_START        my_var             oil_rifle_p
        AT_START        my_string       ‘this is a string variable’
        AT_START        NOTIFY  "  'oil pressure is at ' + oil_rifle_p "
```

Set the value of variable tvo30 ON if the engine speed is above 400 rpm, otherwise, set if OFF. Set the value of the variable my_var to be the same as the value of oil_rifle_p. Set the value of the string variable, my_string, to‘this is a string variable’. If the oil pressure is 10 psi, the string variable NOTIFY will be set to "oil pressure is at 10.0"

Engine Control	@PID_GAINS

Modify controller gain.

Keyword:

@PID_GAINS

Usage:

Modify the gains for a PID control loop. The gain changes remain in effect after the mode is terminated. The gain values may be specified as a constant, variable, or computed expression. The gain_set field may be either the label of a feedback variable if the loop is a user loop or one of the following macros for dyno or throttle control:

DYNO_SPEED_GAINS

DYNO_DYNO_TORQUE_GAINS

DYNO_NET_TORQUE_GAINS

DYNO_OTHER_GAINS

THROT_GROSS_TORQUE_GAINS

THROT_NET_TORQUE_GAINS

THROT_SPEED_GAINS

THROT_OTHER_GAINS

Data Fields:

gain_set	controller index or variable label
prop_gain	proportional gain
int_gain	integral gain
der_gain	derivative gain

Example Specification:

```
@PID_GAINS
   #gain_set                 prop_gain    int_gain     der_gain
   DYNO_SPEED_GAINS          -.01[none]   -.001[none]  0.0[none]
   THROT_GROSS_TORQUE_GAINS   .01[none]    .001[none]  0.0[none]
   int_manf_t                .001[none]    .001[none]  0.0[none]
```

Set the gains of the dyno controller, throttle controller, and the user loop controlling intake manifold temperature.

Specifies optional actions for starting and exception conditions when using the @PIERBURG_COMMANDS keyword.

Keyword:

@PIERBURG_ACTIONS

Usage:

Specify actions associated with Pierburg commands.

Data Fields:

start_type	code for when to send a start signal to the collector task - options are AT_START, AFTER_STABILITY
stop_path	code for what action to take when the auxiliary task completes its function - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.
fail_path	code for what action to take when the auxiliary task signals that is has failed to accomplish its function - options include NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.

Example Specification:

```
@PIERBURG_ACTIONS
        #start_type    stop_path        fail_path
        AT_START       MODE_TERMINATE   /specs/gp/pier_fail
```

Start sending commands to the Pierburg meter at the start of the test mode. Terminate the mode when the communication is complete. Jump to the pier_fail procedure if there are any faults detected with the communication.

Support Tasks	@PIERBURG_COMMANDS

Issue commands to a Pierburg smokemeter.

Keyword:

@PIERBURG_COMMANDS

Usage:

Send a command to a Pierburg smokemeter through a serial port.

Data Fields:

command	command string
value	an optional data value to be inserted into command string
return_value	optional real variable where a result obtained from the meter will be placed

Example Specification:

```
@PIERBURG_COMMANDS
        #command                        value           return_value
        "get mean value"        -                       pier_smk
```

Request the mean value of the last reading from the smokemeter.

Notes:

Command options are: (X is a variable)

```
get measured values             get measured value              get mean value
get maximum deviation           get error code                  get condition
single measurement              grey test                       white
balancing
black balancing                 stop                            flush
paper feed                      take measuring stroke           toggle
measuring valve
change parameter X to Y
use inlet 1                     use inlet 2                     use inlet 3
means: X                        strokes: X                      idle strokes:
X
flushings: X                    flush time: X                   last flush
time: X
pause time: X                   error limit: X                  ao offset: X
ao range: X                     ao                              black
balancing: off
black balancing: on             white balancing: off
white balancing: on             light: measurement              light: always
after flushing: off             after flushing: on
```

Other Examples:

```
@PIERBURG_COMMANDS
        #command                value           return_value
        take measuring stroke
```

Command the meter to start a measuring cycle.

Support for multiple Pierburg smokemeters

Keyword:

@PIERBURG_CONFIG

Usage:

If there is a need to run more than one Pierburg smoke meter in a test cell, this keyword allows the specification of unique names for:

- the ASC configuration file

- the integer variable where condition codes are written

- the integer variable where error codes are written

- the diagnostic output filename

Data Fields:

`name_stem`	`the ASC configuration file name`
`condition_code`	`the integer variable where condition codes are written`
`error_code`	`the integer variable where error codes are written`
`error_file`	`the diagnostic output filename`

Example Specification:

```
@PIERBURG_CONFIG

#name_stem condition_code error_code error_File

pierburg PierCondCode PierErrCode /data/errors/pier_err_log
```

Notes:

The @PIERBURG_CONFIG specification must be placed just above the @NAME keyword in the header file (usually gp_header or pier_header). If only one Pierburg meter is used, no changes are necessary to the header file. One difference, however, is that the diagnostic output no longer goes to the console. Instead, the default output filename is /data/errors/pier_err_log

Branching	**@PROCEDURE**

Execute a sub-procedure.

Keyword:

@PROCEDURE

Usage:

Test procedures may call other test procedures. A particular test mode may consist of running a complete sub-procedure through this keyword. The sub-procedure must specify a RETURN for the default_next_mode to return to the calling procedure. The Test Scheduler supports up to 31 sub-procedures. Procedures are not re-entrant. That is, a procedure cannot be called again from itself or from within another procedure that is called.

Data Fields:

`file_name`	`the full pathname of a separate test procedure`

Example Specification:

```
@PROCEDURE
        #file_name
        /specs/gp/gp_setup
```

Run the procedure gp_setup.

```
@PROCEDURE
   #file_name
   /specs/gp/gp_start;25
```

Run the procedure gp_start and begin with mode 25.

Notes:

The normal mode timeout is not used for a mode that includes the @PROCEDURE keyword. A test mode which contains this keyword should not contain any other keywords which take some kind of action. They will not be in effect. Only the following keywords can be used in the same mode:

@IF_TRUE

@IF_FALSE

@SWITCH

@ELSE_MODE

The following tree illustrates an illegal sequence of procedure calls (gp_map is called a second time before the first call returns to gp_root):

gp_root-□ gp_map-□ gp_fr-□ gp_map

Outputs	**@RAMP**

Ramp a variable.

Keyword:

@RAMP

Usage:

You may specify up to a total of 8 ramps in any mode. This limit applies to the total of all @RAMP specs and all @RAMP_DYNAMIC specs.

There may only be a total of 16 ramping operations going on at the same time. This will include the sum of all ramps generated with 'NONE' termination codes in previous test modes, plus all ramps running in the current mode.

Data Fields:

variable	Label of the target label
start	Optional start target (constant/variable/expression). If the start target is entered as a dash, then the start target is taken from the current value
end	Optional end target (constant/variable/expression). The end target may be entered as a dash only for the case where the termination option is MODAL (default) if the termination option is NONE, then the end target must be specified
rate	Optional ramp rate (constant/variable/expression) If the rate is not specified or is entered as a dash '-', then it is computed from the start/end and mode time. The reate is implied as [units]/sec
termination	Optional termination mode - defaults to AT_END NONE - the ramping operation will continue until the end target is reached even if the test mode expires AT_END - the ramping operation is terminated when the mode expires. The last value is frozen at the value reached by ramping and is not stepped to the end target

Example Specification:

```
@RAMP
        #variable  start  end  rate  termination
        my_var     x      y    z     AT_END
```

Notes:

The start, end and rate values are determined only once, when the ramping operation is launched.

Outputs	@RAMP_DYNAMIC

Ramp a variable with dynamic evaluation of the end target and ramp rate.

Keyword:

@RAMP_DYNAMIC

Usage:

You may specify up to a total of 8 ramps in any mode. This limit applies to the total of all @RAMP specs and all @RAMP_DYNAMIC specs.

There may only be a total of 16 ramping operations going on at the same time. This will include the sum of all ramps generated with 'NONE' termination codes in previous test modes, plus all ramps running in the current mode.

Data Fields:

variable	Label of the target label
start	Optional start target (constant/variable/expression). If the start target is entered as a dash, then the start target is taken from the current value
end	Optional end target (constant/variable/expression). The end target may be entered as a dash only for the case where the termination option is MODAL (default) if the termination option is NONE, then the end target must be specified
rate	Optional ramp rate (constant/variable/expression) If the rate is not specified or is entered as a dash '-', then it is computed from the start/end and mode time. The reate is implied as [units]/sec
termination	Optional termination mode - defaults to AT_END NONE - the ramping operation will continue until the end target is reached even if the test mode expires AT_END - the ramping operation is terminated when the mode expires. The last value is frozen at the value reached by ramping and is not stepped to the end target

Example Specification:

```
@RAMP_DYNAMIC
        #variable  start  end  rate  termination
        yore_var   ex     why  zee
```

Notes:

This functions identically to the @RAMP capability, except that the end and rate are dynamically re-evaluated every FAS intervals.

Data Acquisition	@RECORDER_GROUP

Initiate a recorder sampling process.

Keyword:

@RECORDER_GROUP

Usage:

This specification allows the user to initiate high speed data collection processes during a test mode. The resources of the recorder collector task are used. An appropriate recorder specification file must exist.

Data Fields:

start_type	code for when to start data collection - options are AT_START, AFTER_STABILITY
stop_path	code for what action to take when the data acquisition is complete - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.
continue_flag	flag for whether data collection should continue after the mode terminates - (YES/NO)
spec_file_pathname	the name of the recorder specification file

Example Specification:

```
@RECORDER_GROUP
        #start_type   stop_path   continue_flag   spec_file_pathname
        AT_START      NONE        NO              /specs/rspecs
```

Collect data according to the specifications in /specs/rspecs. Start the data collection at the start of the mode. Terminate the data collection if it takes longer than the mode interval.

Outputs	**@SET_DISPLAY_STATUS**

Modify the display status of a variable.

Keyword:

@SET_DISPLAY_STATUS

Usage:

This specification allows a test procedure to modify the display status of a CyFlex variable. The display status is an attribute of a variable which determines the color and blinking of a variable when it is displayed on the monitor. The normal (default) status of a variable allows it to be displayed without blinking in the background color specified in the dsply_specs.NNN file. This can be modified to blink and change the background color or both. Up to 16 variables may be specified per test mode.

Data Fields:

start_type	a macro which defines when to modify the status - options are AT_START, AT_END, AFTER_STABILITY
label	the label of a CyFlex variable
code	the new display status

Example Specification:

```
@SET_DISPLAY_STATUS
        #start_type            label           code
        AT_START               counter         BLINK
        AT_START               counter2        RED
        AFTER_STABILITY        stab_var        BLINK_YELLOW
```

Notes:

Valid status options are:

NO	BLINK	BLUE	GREEN	CYAN	RED
MAGENTA	YELLOW	WHITE	BLINK_BLUE	BLINK_GREEN	
BLINK_CYAN	BLINK_RED	BLINK_MAGENTA	BLINK_YELLOW	BLINK_WHITE	

Outputs	@SET_EVENTS

Set an event at mode start.

Keyword:

@SET_EVENTS

Usage:

Each event listed will be set at the time indicated by the start_type field. The event length is zero. The events must already exist.

Data Fields:

start_type	code for when to set the event - options are AT_START, AT_END, AFTER_STABILITY
event_name	the name of an event to set

Example Specification:

```
@SET_EVENTS
        #start_type             event_name
        AT_START                get_it_going
        AT_END                  recdr_start
```

Set the event get_it_going at the start of the test mode and set the event recdr_start at the end of the test mode.

Creation	**@SPAWN_CO_PROCESS**

Spawn a child process.

Keyword:

@SPAWN_CO_PROCESS

Usage:

Spawn a child process that is automatically slayed when the test procedure is terminated. This keyword must be specified before the start mode.

Data Fields:

`command`	`Process to be spawned`
`arguments`	`Arguments for the spawned process`

Example Specification:

```
@SPAWN_CO_PROCESS
        /asset/bin/floger /specs/flogging +w
        /asset/bin/floger /specs/flogon
```

Engine Control	**@SPEED**

Specify the engine speed target.

Keyword:

@SPEED

Usage:

This specification selects the reference value for the engine speed variable. This may be the setpoint for either the throttle or dyno loop, depending on the engine control mode, but is usually the dyno setpoint. It is meaningful only if the controller is in closed loop mode. The end_target and the ramp_rate are optional. If the end target specified is different from start target and the ramp rate is not specified, then the ramp rate is computed from the start and end targets and the mode timeout value.

Data Fields:

start_target	the reference value at the start of the mode
end_target	optional reference value at the end of the mode
ramp_rate	optional rate at which to ramp from the start to end target values

Example Specification:

```
@SPEED
        #start_target   end_target
        ramp_rate
        Rated_rpm       "Rated_rpm * 0.9[none]"
```

Ramp from rated speed to 90% of rated speed over the mode interval.

Notes:

The data fields may be constants, variable labels, or expressions. Constants must have units. The units of ramp_rate are entered as units of speed. The denominator is assumed to be seconds. For example, 10[rpm] would specify a ramp rate of 10 rpm/sec.

If the ramp rate is specified such that the end target is reached before the mode terminates, then the ramping stops when the end target is reached.

If the ramp rate is specified such that the end target is not achieved when the mode terminates, the ramping may continues unless the next mode modifies the speed target.

Other Examples:

```
@SPEED
        #start_target   end_target      ramp_rate
        1200[rpm]
```

Set the speed target to 1200 rpm.

```
        @SPEED
        #start_target   end_target      ramp_rate
        1000[rpm]       1000[rpm]       100[rpm]
```

Ramp from the speed reference of the previous mode to 1000 rpm at 100 rpm/sec.

Stability	**@STABILITY_ACTION**

List action to be taken when stabilization occurs.

Keyword:

@STABILITY_ACTION

Usage:

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
```

Data Fields:

action_code	a code which indicates certain special actions to perform

Example Specification:

```
@STABILITY_ACTION
        #action_code
        MODE_TERMINATE
```

Terminate 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.

The code TERMINATE_TO_ELSE_MODE is used 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.

Other Examples:

```
@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.

Stability	**@STABILITY_SPECS**

List stability criteria.

Keyword:

@STABILITY_SPECS

Usage:

This keyword is used 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. See the chapter on Stability for a more complete explanation of each type of stability criterion.

Data Fields:

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 expression.
tolerance	the tolerance for the criteria.
minimum_reference	for type=K_VARIANCE, the lower threshold for the reference

Example Specification:

```
@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.

Notes:

The reference data field may be either a constant, variable label, or a computed expression.

Other Examples:

```
@STABILITY_SPECS
        #type_code       variable   timeout  rate   reference  tolerance   min_ref
        VARIANCE         fuel_rate  10[sec]  SLO    -          1.0[lb/hr]
        TIME_DELAY       -          20[sec]
```

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

Branching	@SWITCH

Specify an alternate execution path which depends on the value of an integer.

Keyword:

@SWITCH

Usage:

The value of an integer variable or expression is compared to a list of case/path pairs. If the switch value matches one of the cases, then the corresponding path is executed. Otherwise, the current mode is executed. Use the "RETURN" macro to return to a calling procedure.

Data Fields:

switch_value	the label of an integer variable or a computed expression
case	an integer value – if a match of the "switch value" is found in this list, then the corresponding execution path is used
path	a mode number, procedure filename, or RETURN

Example Specification:

```
@SWITCH
        #switch value
        count
        #case           path
        1               91
        2               92
        3               /specs/gp/gp_test3
        4               RETURN
```

If the value of count is 1, then go to mode 91. If the value of count is 2, then go to mode 92. If the value of count is 3, then jump to procedure gp_test3. If the value of count is 4, then return to the calling procedure. If the value of count is any value other than 1, 2, 3, or 4, then execute the current mode.

Notes:

The @SWITCH keyword is used to control the execution path of a test procedure when there are several paths that can be taken, depending on the value of some variable. The variable might be something like an error code read back from a Pierburg meter, or the value of a counter which is keeping track of something.

Other Examples:

```
@SWITCH
   #switch value
   "error_code / 10[none]"
   #case           path
   1               /specs/gp/gp_errA
   2               /specs/gp/gp_errB
   3               /specs/gp/gp_errC
```

Mode Termination	**@TERM_ALL_EVENTS**

Specify a list of events which must all be received to terminate the mode. This keyword would be used instead of @TERMINATION_EVENTS if the conditional AND is appropriate.

Keyword:

@TERM_ALL_EVENTS

Usage:

Events can be used to force the test to terminate the present mode and advance to the next mode. This specification is a list of event names which are to force the mode to terminate.

Data Fields:

`event_name`	`the name of the event`

Example Specification:

```
@TERMINATION_EVENTS
        #event_name
        stop_this
        rec_done
        startup
        alldone
```

Mode Termination	**@TERMINATION_EVENTS**

Specify events which are to terminate the test mode.

Keyword:

@TERMINATION_EVENTS

Usage:

An event can be used to force the test to terminate the present mode and advance to the next mode or to a specific test mode or to a different test procedure. This specification is a list of event names which are to force the mode to terminate. If the next_path field is 0 or - or not specified, then the next mode is executed.

Data Fields:

event_name	the name of the event
next_path	the next mode or test procedure to execute (optional)

Example Specification:

```
@TERMINATION_EVENTS
        #event_name     next_path
        stop_this       0
        rec_done        3
        startup         /specs/gp/gp_start
        alldone         RETURN
        my_event        /specs/gp/gp_event;25
```

If the stop_this event is received, terminate the mode and proceed to the next mode. If the rec_done event is received terminate the mode and branch to mode 3. If the startup event is received, jump to the gp_start procedure. If the alldone event is received, return to the calling procedure. If the my_event is received, the gp_event procedure is started in mode 25.

Branching	@TEST_CYCLE_END

Designate this mode as the last mode of a test cycle.

Keyword:

@TEST_CYCLE_END

Usage:

This is used to count cycles in a test. It is used only when the cycle count must be maintained over a long period of time and when the test must be terminated after the required number of cycles is complete. When the cycle count reaches the maximum_cycles, then control is passed to the test_complete_path. The test_complete_path may be a mode number in the current procedure, a procedure filename, or the RETURN macro to return to a calling procedure.

Data Fields:

maximum_cycles	the maximum number of test cycles
cycle_counter	a variable assigned to counting of the cycles
test_complete_path	the mode or procedure to execute when the test is complete

Example Specification:

```
@TEST_CYCLE_END
   #maximum_cycles   cycle_counter   test_complete_path
   30000             test_cycles     91
```

Run the test until the test_cycles variable reaches 30000, then proceed to mode 91.

Notes:

The value of the cycle counter can be reset to zero or modified by the user with the svar command.

Other Examples:

```
@TEST_CYCLE_END
   #maximum_cycles   cycle_counter   test_complete_path
   100               cyc_cntr        RETURN

@TEST_CYCLE_END
   #maximum_cycles   cycle_counter   test_complete_path
   1000              test_cycles2    /specs/gp/gp_shutdown
```

Engine Control	@THROTTLE

Specify the throttle controller mode & open_loop position.

Keyword:

@THROTTLE

Usage:

This specification selects the throttle controller mode to be either OPEN_LOOP or CLOSED_LOOP. If the mode is CLOSED_LOOP, then only one data field is required and the target values are ignored. If the mode is OPEN_LOOP, then the start_target must be specified. The end_target is optional. If the start and end targets are both entered and are different, then the controller output will be ramped linearly over the mode time_out interval. The targets are always expressed as percent of full scale output.

Data Fields:

control_mode	open/closed loop (OPEN_LOOP or CLOSED_LOOP)
start_target	used only if mode is open_loop, units must be % of full scale
end_target	used only if mode is open_loop and ramping is required, units must be % of full scale
ramp_rate	optional argument to specify the open_loop ramp rate

Example Specification:

```
@THROTTLE
#control_mode   start_target    end_target      ramp_rate
OPEN_LOOP       100[%]
```

Set the throttle to full open position.

Notes:

The start and end targets may be constants, variable labels, or expressions. Expression must be enclosed in double quotes. Units are required for all constants.

Other Examples:

```
@THROTTLE
#control_mode   start_target    end_target          ramp_rate
OPEN_LOOP       min_thr        "(max_thr - min_thr)/ 2.0[none]"
```

Ramp the throttle from the value in the min_thr variable to the value determined by the computed expression.

```
@THROTTLE
#control_mode   start_target    end_target      ramp_rate
CLOSED_LOOP
```

Set the throttle controller to closed loop. The throttle position will be determined by the PID control.

Engine Control	@TORQUE

Specify the torque target.

Keyword:

@TORQUE

Usage:

This specification selects the reference value for the torque variable. This may be the setpoint for either the throttle or dyno loop, depending on the engine control mode, but is usually the throttle setpoint. It is meaningful only if the controller is in closed loop mode. The end_target and the ramp_rate are optional. If the end target is specified is different from start target and the ramp rate is not specified, then the ramp rate is computed from the start and end targets and the mode timeout value.

Data Fields:

start_target	the reference value at the start of the mode
end_target	optional reference value at the end of the mode
ramp_rate	optional rate at which to ramp from the start to end target values

Example Specification:

```
@TORQUE
#start_target   end_target      ramp_rate
1000[lb_ft]
```

Set the torque reference to 1000 lb_ft.

Notes:

The data fields may be constants, variable labels, or expressions. Constants must have units. The units of ramp_rate are entered as units of torque. The denominator is assumed to be seconds.

Other Examples:

```
@TORQUE
#start_target   end_target      ramp_rate
Peak_torq
```

Use the variable Peak_torq for the torque control reference.

```
@TORQUE
   #start_target        end_target      ramp_rate
   " (Max_trq - Min_trq) * .5[none] "
```

Set the torque control reference to 50% of the difference between the Max_trq and Min_trq variables.

ECM Communications	@UNICO_ACTIONS

Command used to communicate with an UNICO controller running on TCP/IP connection.

Keyword:

@UNICO_ACTIONS

Usage:

Data Fields:

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:

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

ECM Communications	@UNICO_GET

Command used to get a value for a specific variable from the ECM.

Keyword:

@UNICO_GET

Usage:

Data Fields:

controller_variable	The name of a UNICO interface control variable. This may be a constant, variable or computed expression which resolves to a valid CyFlex 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
```

Command used to set a value for a specific variable from the ECM.

Keyword:

@UNICO_SET

Usage:

Data Fields:

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

Example Specification:

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

Control	@USER_LOOP

Specify PID control loop mode and target.

Keyword:

@USER_LOOP

Usage:

This specification allows user PID loops to be set to open or closed loop and to have the reference value modified. If the mode is OPEN_LOOP, then the targets must be expressed in percent of full scale output. If the mode is CLOSED_LOOP, then the targets must be appropriate units for the variable. The ramp_rate and end_target are optional. If the ramp_rate is not specified, and the start and end targets are different, then the ramp rate is based on the mode timeout value.

Data Fields:

control_mode	open or closed loop mode, OPEN_LOOP/CLOSED_LOOP
variable	reference variable
start_target	the target at the start of the mode
end_target	optional target at the end of the mode
ramp_rate	optional rate at which to ramp the target

Example Specification:

```
@USER_LOOP
#control_mode   variable        start_target    end_target      ramp_rate
CLOSED_LOOP     port_in_p       15[in_hg]
CLOSED_LOOP     fuel_temp       100[deg_F]      110[deg_F]
OPEN_LOOP       air_inlet_t     100[%]
```

Set the reference for point_in_p to 15 in_hg. Ramp the reference for fuel_temp from 100 to 110F over the mode interval, and set the controller for air_inlet_t to full open.

Notes:

The target may be expressed as a constant, variable label, or computed expression.

Other Examples:

```
@USER_LOOP
#control_mode   variable        start_target    end_target      ramp_rate
CLOSED_LOOP     aa_temp         compute_aa
```

Set the control reference for the loop controlling aa_temp to be the value of the variable, compute_aa. This might be used for a situation where the desired intake manifold temperature is a function of speed and load and is being continuously computed and placed in the variable compute_aa.

Outputs	@VZ_CONFIG

Virtual zero/span for emissions carts.

Keyword:

@VZ_CONFIG

Usage:

Data Fields:

device	NOX, CO2, HC, CO2_RAW
action_code	VZ, VS, HZ, HS, RANGE
value	Constant, label or computed expression

Example Specification:

```
@VZ_CONFIG
#device         action_code     value
NOX             VZ              0[none]
CO2             VS              6[ppm]
```

Data Acquisition	@WRITE_VALUES

Write formatted data to a file.

Keyword:

@WRITE_VALUES

Usage:

This keyword allows the user to write ASCII data into a file and control the data, format, and rate through the test procedure specification. In effect, a flexible data logging operation can be created with the test scheduler. Up to 32 specifications can be included with this keyword.

Data Fields:

start_type	code for when to write the data. Options are AT_START, AT_END, AFTER_STABILITY
file_name	the file where the data will be written
variable	a CyFlex variable which will be written according to the format string
format_string	a C format string used for formatting the write.

Example Specification:

```
@WRITE_VALUES
#start_type     file_name       variable        format_string
AT_START        /data/tq_sp     -               RPM
AT_START        /data/tq_sp     RPM             %11.2f
```

Write a header line and the current value of engine speed into the file /data/tq_sp each time this mode is executed.

Notes:

Each write operation is appended to the end of the file.

Support Tasks	@AKSYNC

Send command to asynchronous device using AK protocol with synchronized communications.

Keyword:

@AKSYNC

Usage:

Send a command to a device connected to a serial port. The command string will be parsed and converted to a string which the device can interpret based on a configuration file. The XXXX configuration file must be in /specs directory and must be named device.cfg, where device is the same as that used in the @ASC specification. For this example, the file would be /specs/pager.cfg.XXXXX

Data Fields:

Instrument	Name of instrument the AK_sync task has attached to.
Command	Instrument specific command, format defined in instrument speicification file. (ie. AVL483.spec)

Example Specification:

```
@AKSYNC
#instrument name
AVL483
#command key strings
# get a reading
"ADIL SS_dil_typ SS_dil - - -"
"ASTF SS_error1 SS_error2 SS_error3 SS_error4 SS_error5 SS_error6"
"AKON avl_483_soot avl_483_corr"
```

Other Examples:

```
@AKSYNC
#instrument name
AVL483
#command key strings
# set dilution parameters
"EDIL SS_dil_typ_TR SS_dil_TR 1.00 10.00 1.00"
```

Specify when to send an AKSYNC command. failure.

Keyword:

@AKSYNC_ACTIONS

Usage:

This keyword is used to specify the actions and timing associated with all AKSYNC communications for a test mode. The `start_type` specifies when the commands will be executed, the `stop_path` specifies what action will be taken when all commands have been completed, and the `fail_path` specifies what to do if the communication fails to complete successfully.

Data Fields:

start_type	code for when to send the message - options are AT_START, AFTER_STABILITY, AT_END
stop_path	code for what action to take when the communication is complete - options are NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.
fail_path	code for what action to take if there is a failure to communicate with the ECM - options include NONE, MODE_TERMINATE, RETURN, a mode number, or a procedure file pathname.

Example Specification:

```
@AKSYNC_ACTIONS
        #start_type stop_path          fail_path
        AT_START    MODE_TERMINATE     /specs/gp/comm_fail
```

Start sending the AKSYNC commands at the start of the mode, terminate the mode when all are complete, and if there is a failure, go to the test mode specified by the `@ELSE_MODE` keyword.

Notes:

This keyword is used only when there are other AKSYNC communication commands such as `@AKSYNC`.

Cause the AKSYNC collector task to be terminated ans restarted.

Keyword:

@AKSYNC_RESTART

Usage:

```
@AKSYNC_RESTART
@AKSYNC
#instrument
 AVL483
```

Notes:

This keyword is used only when there are other AKSYNC communication to specify the specific instance of AK_sync task.