WHEN YOU NEED TO BE SURE

CyFlex® Master Scheduler

Version 3

June 20, 2022

Developed by SGS North America, Inc.

Version History

Version	Date	Revision Description
1	1/25/2016	Initial publication
2	4/10/2020	Retrofit to new template
3	6/20/2022	Added hypertext linked cross-reference to cyflex.com usage help for scheduler in *Section 1.1 System Watchdog* on page 1 and updated additional hypertext linked cross-references to cyflex.com usage help descriptions.

Document Conventions

This document uses the following typographic and syntax conventions.

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

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

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

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

- Cross-references are designated in Arial italics.

  Example: Refer to *Figure 1*…

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

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

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

CyFlex Documentation

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

Table of Contents

1 Overview

The `scheduler` task monitors the integrity of processes running in CyFlex.

1.1 System Watchdog

The `scheduler` task manages the system *watchdog*. The watchdog is a digital output channel which is assumed to perform the function of shutting down a test system in a safe manner if the system is not functioning properly. The watchdog channel equates to a heart-beat that must be continually toggling between the `ON` and `OFF` states at a certain rate (once per second). If the change of state does not occur within a certain period of time, a shutdown sequence starts to deactivate the test system. The characteristics of the actions that occur when the watchdog channel stops toggling are controlled by the design of an external hardware system. The hardware system must be designed to perform an appropriate sequence of actions. Some systems may not be equipped with this hardware. The external watchdog system will vary from site-to-site or cell-to-cell since they are not controlled by the software. Refer to cyflex.com usage help for `scheduler` for command syntax and options.

Identify the watchdog channel to the `scheduler` with the `do_specs` command keyword `WATCHDOG` in the specifications of the digital output channels. Refer to cyflex.com usage for `do_specs`.

1.2 Critical and Non-Critical Applications

An application registers itself with the `scheduler` when it starts up and may subsequently modify its registration features. Each application can register as being a “critical” task or a “non-critical” task. If a failure occurs with a task that registered as “critical’, then the `scheduler` task will stop toggling the “watchdog” channel. The external watchdog hardware system, if it exists, will start the system shutdown sequence for which it was designed.

Should the faulty application recover from the failure and signal the `scheduler` accordingly, the `scheduler` will begin toggling the watchdog channel again. This does not guarantee that the external hardware will immediately recover, since some systems have been designed to require a manual reset of the watchdog circuitry by the test cell operator.

1.3 Initial State

Applications may be designed to be initialized with a reconfiguration in progress. Examples are `limit_specs` and `evnt_rsp`. The result is that the `scheduler` will immediately begin counting down the specified timeout for reconfiguration. If the timeout limit is exceeded before the appropriate configuration of the application takes place and the application is critical, then the watchdog will be suspended. For example, if either of these applications is launched in the `go.scp` startup file, but `limit_specs`” is not launched, then the watchdog will fail and the engine cannot be started. Any application can be designed to operate this way as a protection to ensure that it is properly configured at startup. Refer to *Section 2 Registering an Application with the Scheduler* on page 2.

2 Registering an Application with the Scheduler

The following code segment shows the use of the function `ms_initialize()` to register the application with the `scheduler`.

```
//This is stripped down code for an application that only supports the SLO //interval
//
//   my_app 19 SLO +c &
//
#include “asset.h”
#include “errors.h”
#include “asset_pt.h”
#include “sys_attr.h”

main ( int argc, char *argv[]  )
{
   union
   {
      GLOBAL_CONFIG_EVENT global_config;
   } event_in;

                                          // the message structure we will
                                          // send to the scheduler
   PROCESS_DONE_EVENT
      done;
                                          //  this is a list of input and
                                          //  output events
   long
      config_eid,
      timer_eid,
      global_eid,
      done_eid,
      wait_eid;

   short
      status = NO_ERROR;

   LOGICAL
      critical,
      hold_in_config;



                                          // create our own session
   _setsid();
                                          // register with child_adm so that
                                          // slay_stuff will kill this app

   status = join_layer( APPLICATION_LAYER );

   if( status != NO_ERROR )
   {
      log_error( ACTION( ERR_SCRN | STD_OUT ),
                 status,
                 "couldn't join app layer" );

      exit( -1 );
   }

                                          // place our PID in the done message
```

```
   done.task_id = getpid();
                                        /*  set up the done event message */
   done.process_interval = Sys_attr->plist[ 2 ].interval;



                                           // determine if +c argument is there

   critical = ( strcmp( argv[argc-1],"+c" ) == MATCH )?TRUE:FALSE;

                                        // if critical, initialize to reconfig in
                                        // progress
   hold_in_config = critical;

                                           // this function gets the event id of
                                           // and registers
                                           // with the scheduler for those
                                           // timers specifed
   status += ms_initialize(
                     argc - 1,
                     argv + 1,
                     “my_app”,
                     &done_eid,
                     &global_eid,
                     10,  /* reconfig timeout (sec)
                     20,  /* max overruns of timer signal (sec)
                     critical
                   );

                                           // do initialization and attach to
                                           // the process timer event and the
                                           // process config event
   status += init( &timer_eid, &config_eid);

   if( status != NO_ERROR )
   {
      log_error( ACTION( ERR_SCRN ),
                 TASK_INITIALIZATION_FAILURE,
                 "unable to initialize properly" );
      exit( 0 );
   }
                                        /* loop forever waiting on an event
                                        from the event administrator */
   for (EVER)
   {
                                        /*  wait on  event */

      status   = event_wait ( &event_in,
                              sizeof(event_in),
                              &wait_eid );

      event_found = FALSE;

                                        /* We should never get an error from
                                        the event_wait, but if we do we can
                                        go into an infinite loop.  The
                                        following code with the set_timer
                                        delay forces this process to give up
                                        cpu time to other processes.  At least
                                        we will be able to run some other shell
                                        to diagnose the problem. */
      if( status != 0 )
```

```
      {
         log_error( ACTION( ERR_SCRN ),
                    EVENT_WAIT_ERROR,
                    "error from event_wait- status=%d",
                    status );
         sleep( 1 );
         continue;
      }


      if( wait_eid == timer_eid )
      {
                                       // send reconfig state to scheduler
         done.reconfig = cfg_in_progress();
                                       /* set the done event */
         status = event_set( done_eid,
                             &done,
                             sizeof (done) );


                                       /* is the event a specified input
                                       event */
         process( wait_eid );
      }

      else if( wait_eid == config_eid )
      {
        config();

      }
                                /* was the event a configuration? */

      else if( wait_eid == global_eid )
      {
         if( reconfigure( event_in.global_config )
         {
             config_variables();
          }
      }

   }                                   /* end of for loop */

}                                      /* end of function */
```

2.1 Timers

An application may inform the `scheduler` that it is using any or all of the 6 defined process intervals: WARP/FAS/MED/SLO/USR1/USR2. These interval values are defined for the system by command line arguments when the `scheduler` is started:

```
scheduler PRI=21 FAS=20 MED=100 SLO=1000 USR1=2000 USR2=5000 WARP=5 &
```

The `PRI` option specifies the priority at which the `scheduler` with `21`as the recommended value. The various process intervals are optional, but there must be at least one interval specified. The values are in units of milliseconds. Only those intervals specified on the

command line of the `scheduler` may be used by other applications for registration with the scheduler.

When an application informs the `scheduler` that it is using a particular process interval, the `scheduler` expects that the application will signal the `scheduler` that it has completed the processing associated with that interval. It does this each time it receives that timer event. It also informs the `scheduler` whether it is actually processing data or whether it is currently in a “reconfiguration” state. It sends this information to the `scheduler` by setting the DONE message event. This message includes the process ID, timer value, and reconfiguration state.

When registering a particular timer with the `scheduler`, the application must also specify two limits.

1. The maximum time allowed for the reconfiguration state
2. The maximum time allowed between the timer event and the DONE event

If either of these limits is exceeded, then the `scheduler` will respond depending on whether the application is registered as being ‘critical’ or not. If critical, the watchdog output is killed and an error message is generated. If non-critical, the only action is an error message.

3 Launching an Application

Not every application can use the `scheduler` and watchdog. The application must be designed to programmatically support the registration, timer handling, and DONE event response. Assuming that the application is designed properly, there is a general form for launching such applications, although there may be exceptions. Refer to the cyflex.com Usage Help Manual for application details.

```
my_app  <priority> <list of intervals> [+c]  &
```

`+c` indicates that the task is to be registered as “critical”. It must be the last argument.

Example:

```
my_app 16 FAS SLO +c &
```

The list of intervals is determined as a function of the application and possibly which process intervals are included in its specifications.

4 Failures

4.1 Failure Modes

The `scheduler` task handles three modes of failure and will perform either the critical or non-critical actions for all three:

1. Application has died and never sends the DONE event
2. Application is not able to process the timer events fast enough and exceeds the maximum time limit for a particular process interval
3. Application is in the reconfiguration state longer than the specified maximum time

An additional failure mode can occur which will not be apparent to the `scheduler` task that it cannot report. One of the processes handling the DO output function could fail and thus cause the watchdog hardware circuitry to initiate a shutdown process. Possible cause are:

- The `do_logi_xfer` task died.
- The translation of DO specifications failed or was not run.
- The DO driver failed or was not activated properly.
- The DO hardware channel that operates the external watchdog hardware failed.
- The external watchdog hardware failed.

4.2 Diagnosing a Failure

1. The `scheduler` task will generate error messages indicating failures:

```
Error 0 in Task: scheduler       ,NID: 3 PID:  17977   On:13:13:09 01/05/10
File: ms_sig_list.c                    Line: 195
watchdog suspended due to named process<comp_ctrl>[12170] for interval <20>

Error 0 in Task: scheduler       ,NID: 3 PID:  17977   On:13:13:09 01/05/10
File: ms_sig_list.c                    Line: 183
Named process <comp_ctrl>[12170] removed from 20 list
```

2. The `ms_diag` application will report all of the applications that have registered with the `scheduler` and show their current state and a summary of all failures. The example `ms_diag` output below shows that the `do_logi_xfer` and `comp_ctrl` tasks are not responding; they were slayed in this case. The `comp_ctrl` task also was registered as critical and thus caused the suspension of the watchdog output as reported in the error message above.

```
###############################################################################
The following entries may have the several keys appended
 to the line. The following are possible keys
   *0    > the process has overrun its response counter
           and is not responding
   *R    > the process is responding but has been in
           the reconfiguration state too long
   *C    > the process is a critical task

  Enter 'use ms_diag' for more information on Active Flag

Index  Active      Task Name     PID  Process   Overrun      Reconfig
       Flag                           Rate   value/limit   value/limit
```

```
 0      1     ai_transfer     2618   FAS       0/500          0/5001 *C
 1      0    do_logi_xfer     DEAD   FAS     251/250          0/2501 *O
 2      1     ao_transfer     9792   FAS       0/500          0/2501
 3      1       ctrl_task    12159   FAS       0/500          0/2501 *C
 4      0       comp_ctrl     DEAD   FAS     501/500          0/2501 *O*C
 5      0            dwpt    14221   FAS       0/500          0/2501
 6      1       comp_perf    12169   FAS       0/500          0/2501
 7      0        GL_SM415    12175   FAS       0/500          0/2501
 8      0           Limit    26512   FAS       0/500          0/25001*C
 9      0             fac     7681   FAS       0/500          0/25001*C
10      1         RunAver     6659   FAS       0/1000         0/2501

 0      1     ai_transfer     3643   MED       0/100          0/1001 *C
 1      1       ctrl_task    12162   MED       0/100          0/501  *C
 2      0              ng    12171   MED       0/100          0/501  *C
 3      0            hsda    12172   MED       0/100          0/501
 5      0            dwpt    14221   MED       0/100          0/501
 6      1       comp_perf    12169   MED       0/100          0/501
 7      0        GL_SM415    12175   MED       0/100          0/501
 8      0           Limit    26512   MED       0/100          0/5001 *C
 9      0             fac     7681   MED       0/100          0/5001 *C
10      1         RunAver     6659   MED       0/200          0/501

 0      1     ai_transfer     3643   SLO       0/10           0/101  *C
 1      1    do_word_xfer    29247   SLO       0/10           0/26
 2      1       fici_xfer    31297   SLO       0/10           0/151
 3      1       ctrl_task    12162   SLO       0/10           0/51   *C
 4      1        EvntResp    12164   SLO       0/20           0/51   *C
 5      1             fac     7681   SLO       0/10         122/501  *C
 6      0            hsda    12172   SLO       0/10           0/51
 7      1           Limit    26512   SLO       0/10         122/501  *C
10      1       comp_perf    12169   SLO       0/10           0/51
13      0             pms    11975   SLO       0/10           0/501
15      1        cell_mon    12802   SLO       0/10           0/101  *C
16      1           gasfl    15058   SLO       0/10           0/51
17      1         RunAver     6659   SLO       0/20           0/51
18      1           volef     4819   SLO       0/10           0/26
19      1        addwater     2772   SLO       0/10           0/26
```

Failure causes

The following is a list of processes that have overrun their response counter or their configuration counter. It also means that the process is still not responding to the scheduler. If a particular entry contains 'critical', then the watchdog would have been suspended as a result of the overrun.

```
overrun             do_logi_xfer id=   DEAD FAS
critical overrun       comp_ctrl id=   DEAD FAS
```

4.3 Recovering from a Watchdog Failure

After appropriate investigation of the cause and problem correction, use either of the following methods to recover from a watchdog-induced shutdown:

1. Run a `go`.
2. Restart the offending task(s) and enter the `clear_watchdog` command.

Also, depending on the particular configuration of the watchdog hardware circuitry, the watchdog circuit may have to be manually reset. Refer to cyflex.com usage help for the `clear_watchdog` command.