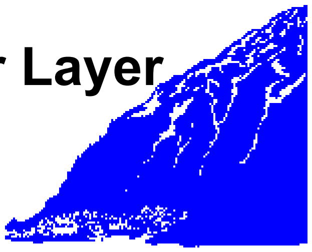
GUI ScreenIO Client/Server Layer Job Timeout Facility



The GUI ScreenIO Server daemon supports idle job termination; the ability of the server to terminate jobs after a configurable period of user inactivity. This document explains how this is works, and the consequences of using this feature.

Purpose.

The purpose of this facility is to prevent a remote user from starting a task, and walking away thereby tying up a server "seat". It also can compensate for unreliable network connections where the connection drops during periods of inactivity. (It is difficult to distinguish inactivity from a network interruption in a transaction processing environment where users may remain inactive for varying periods of time).

Basic Synopsis.

Each time the remote client sends data to the application via the TCP/IP connection, a timer is reset to zero. If that timer ever reaches your designated interval, the Server instructs the application to shut down, and three minutes later, if it is still running, it is forcibly shut down.

Is this wise?

We would be remiss if we did not point out that terminating a program at other than its designed exit point is fought with peril and could lead to serious data corruption.

However, network connections can often be failure prone by their vary nature, and there is no point in continuing to run a job which requires user interaction when that user is unable to communicate with the job, either because that job is waiting for end-user input from an absent user, or the connection is down, or (we realize this is *HIGHLY* unlikely) your program goes into a never ending loop. Sooner or later the job will have to be terminated in an abnormal way.

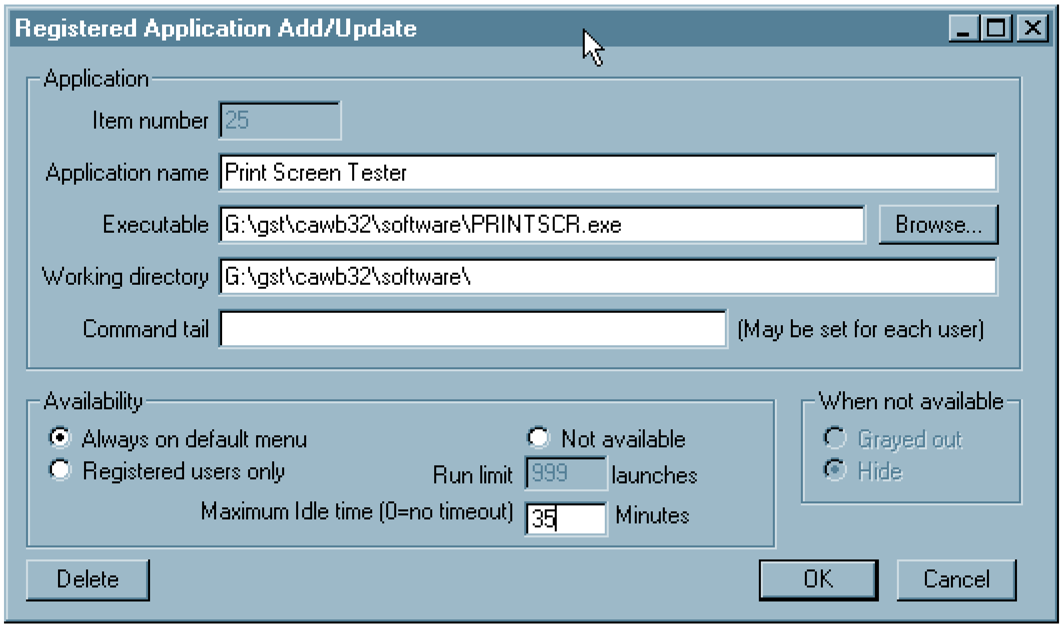
We have designed the Job Timeout mechanism with the intent of giving your job every opportunity to gracefully and properly shut down when termination is required.

Job Timeout is capable of brutal and unforgiving job termination, and it WILL terminate your jobs at the designated interval, *but not without providing **several opportunities** for a graceful shutdown*.

Job Timeout is OFF and unused by default. You have to specifically turn it on.

Setup and Use.

To setup Job Timeout, you use the Server Configuration Utility, and put a *number of minutes* in the timeout field on the Registered Application definition page as shown in the image.


Registered Application Add/Update

Application

Item number	25
Application name	Print Screen Tester
Executable	G:\gst\cawb32\software\PRINTSCR.exe
Working directory	G:\gst\cawb32\software\
Command tail	(May be set for each user)

Availability

[✓ Always on default menu] Registered users only Not available

Run limit 999 launches

Maximum Idle time (0=no timeout) 35 Minutes

When not available

Grayed out [✓ Hide]

An application with a 35 minute timeout set.

Zero means there is no timeout enforced. Any positive number will cause jobs to be terminated after that amount of time **if** there is no interaction with the remote client.

Job Timeouts can be set on an application by application basis. It applies to all jobs run for a given application regardless of which user runs the job. Some jobs may not require timeouts, while others may.

One minute is the smallest increment of time tracked.

One should set timeout intervals with as much leniency as can be afforded, with due regard to the user's work flow, and the demands of the normal work place, such as interruptions caused by phone calls, conversations, refilling the coffee cup, or "parking" the previously consumed cups of coffee.

Since Job Timeout applies to the entire application, you should set it for the longest period, not the shortest. If you have places in your application which require shorter timeouts, such as when the application is holding a lock on a key resource, we recommend you use Panel Timeouts within the application itself to release the lock rather than relying on a system wide utility that cannot, by its nature know anything about the specific requirements of any give screen.

How it works.

When the user selects an application from their menu, the request is sent to the Server. The Server then spawns *an instance* of that application for that user. We refer to an individual instance of an application as a *job*.

If the application database says this job should run with timeouts enabled, the server starts a separate timer for each instance of this application.

The application starts running (as if you had typed its complete path specification and file name at the console, or clicked an icon), and upon the first call to GUI ScreenIO, the job's GS32.DLL will establish a bidirectional connection with the Server. (Remember the Server and the application are running in the same machine. This connection is from one program to another all within the same machine). This is called the *Status Socket*.

GS32.DLL asks the server to “hand off” the TCP/IP connection from the Client so that it can begin sending panels from the application to the remote user. This connection is called the *Client Socket*.

The Server hands off this client socket to the application's GS32.DLL, and from then on till the application ends, it is no longer in direct communication with the Client. Instead, the application's GS32.DLL handles all communication between the application and the client.

Each time the application sends and receives data to or from the remote Client via the Client Socket, its GS32.DLL passes *status information* back to the server via the Status Socket. This information consists only of counts of bytes sent and received, and the name of the panel being displayed. This information is displayed on the Server Control Panel.

Status data is sent each time the user at the remote Client interacts with the application, either because they pressed a function key, clicked a pushbutton, selected a menu item, or triggered a hot-field event. The Server's timer for this job is reset to zero each time GS32.DLL sends status information to the Server.

All of this is handled automatically by the application's copy of GS32.DLL talking with the Server, with no need of special coding by the programmer.

Status information is not transferred across the TCP/IP connection to or from the client, it is strictly within the computer upon which the Server runs.

Job Timeout Sequence of Events.

If the timer reaches the designate number of minutes several things happen to cause a shutdown of the application. These are listed below in the order they occur:

1. The Server sends a Timeout-Stop command to the Application. Actually, the Server sends this to the GS32.DLL of that application via the Status Socket.

2. The GS32.DLL sends the application a Close-And-Stop event. Every panel has a Close-And-Stop event in its copybook. (value 8010).

   At this point the application should shut down if the programmer followed our recommended way of coding. **THIS IS YOUR FIRST CHANCE** at a graceful exit.

   This is the same event the application would receive if the user clicked the **X** in the window title bar, or if Windows itself were to be shut down. Every application is expected to handle this operation from anywhere at anytime. (However, we are aware that not all programmers do so, and offer an additional method to deal with this as described below).

3. If the application responds to the 8010 by showing another panel, the GS32.DLL will respond to that panel with another 8010 Close-And-Stop event. This subsequent panel (and/or any subsequent panels) **will NOT be sent to the client**.

   This is for several reasons: The user is not interacting with the Client at the remote computer, and would not see these panels, OR the connection to the remote Client may be down. In most cases of a timeout, the Client is not expecting data from the server, it is waiting for user input.

4. If the application responds to the 8010 with the SAME panel, GS32.DLL counts the number of times that panel is redisplayed in response to successive 8010s.

   Upon the third one, GS32.DLL is forced to assume that the programmer did not code any logic to handle Close-And-Stop events. These are your **SECOND AND THIRD CHANCES** for a graceful shutdown.

   Remember: Your application will be given 3 successive 8010 events for the same panel before further steps are taken.

   On the other hand, the application will be given an unlimited number of 8010s if it responds by displaying a different panel, as might be the case if your application was "Backing Out" of some task. This means that you really only have to handle 8010's LOCALLY for each panel, returning control to the prior panel, which in turn returns control to whatever panel invoked it, etc., until your application reaches your first screen and then exits normally. Many Legacy ScreenIO applications were programmed this way, and its not an unreasonable approach.
   This logic is not particularly bright and would be fooled by a two panel loop, where two panels were displayed alternately. If so, other means will be brought to bear in shutting down your wayward application. See below.

5. If the third Close-And-Stop event to the same panel is answered with yet another display of that same panel, GS32.DLL will send a ***Mandatory Stop*** event. This event DOES NOT APPEAR in any panel copybook. Its value is a Negative 8010.
   The negative 8010 is handled by GS.COB. GS.COB is the tiny interface program we supply which is your application's gateway to GS32.DLL. GS.COB, as supplied by us, knows how to handle Mandatory Stop events. Its pretty simple minded: it does a COBOL STOP RUN. This is an alternative method to handling timeouts, which does not require that code to handle application termination be provided for every panel in your system. It allows you to handle this centrally in GS.COB, but it only works when running under the Client/Server layer.

   STOP RUN may not be the right thing to do. You should maybe modify this to properly close any files, shutdown and databases, cancel any transactions which are in-flight, and release any resources (locks) held by your application. We can't help you with this because we don't know what your application is doing or how it does it. You can do anything you need to do in response to a Mandatory Stop, just don't call GS32.DLL again trying to display any panel. (Other than a DO-CLOSE for your Main panel.) See the comments in the source code of GS.COB.

THIS IS YOUR FOURTH (and last) CHANCE to effect a graceful shutdown. You must finish this work in less than a minute.

6. If your application ignores the Mandatory Stop event (negative 8010) and calls GS again, GS32.DLL will close its sockets to both the remote Client, and the Server. This signals each that the application is terminating. The Server, upon seeing that the status socket closed, moves the Application from its “Active” queue, to its Inactive Queue (affectionately known as the Dead Pool). It no longer counts against the licensed number of seats at this point.

7. GS32.DLL will then use a Microsoft *ExitProcess* API call. This ExitProcess is essentially what windows tasks do when they STOP RUN, EXIT PROGRAM or GOBACK to the calling program, (usually Windows itself).

   ExitProcess closes all handles (files, screens, etc) opened by the process, signals all threads of your application to exit, all DLLs to unregister themselves and release any resources, and it signals the task to shutdown, and sets the termination status to 111. It is slightly more brutal than a programmer designed exit, but still allows your compiler's runtime to exit gracefully, even if your code can't.

   Note: most COBOL compiler's documentation specifically states that upon exiting a main program, the runtime will close any files that are open and release all resources. This may or may not apply to file systems associated with third party databases. Many of them can/will close out properly as well. (Properly does not imply correctly in this instance.)

8. At this point your program is shut down. Unless it isn't. See below.

All of the above happens very quickly, usually within milliseconds. Remember that GS32 will not send anything to the remote Client if it gets a Timeout-Stop command. So no time is spent sending data across the network, (which may have failed anyway).

On the Client Side

After termination of the application at the server, the remote Client will continue to display the last screen shown. It will just sit there until the user returns from the two martini lunch (or whatever), and clicks a button or something. Then the Client will advise the user that the job has timed out (or the connection was broken) and the session must be re-started, at which point the Client exits back to Windows. The user is not left wondering what happened. The Client provides explicit information.

If GS32 Does Not Get Control.

Clearly the above requires that your program is responsive to, and waiting for, data sent by the Client. It must have just called GS, (or must eventually call GS) and be waiting for control to return from GUI ScreenIO. There is a high probability that your application will be in this state, because transaction processing systems spend the vast majority of their processing time waiting for human input.

If your program was in a computational loop, or running a time consuming complex task, or waiting for a lock to clear so that it could read data from a file, or in any other way **NOT** calling GS, the application's GS32.DLL will never receive control, so it will never be able to process Timeout-Stop commands and pass Close-And-Stop events to your program.

For this reason, even if you are running a task that takes quite a while, its a good idea to periodically call GS to display some progress indication to the end user, and, most of all, be prepared to handle Close-And-Stop events. This is just good programming practice, and a user-friendly approach anyway. This is an excellent use for progress bar controls.

If you don't call GS periodically, AND you choose to use the Timeout feature, you run the risk never being aware of requests to terminate gracefully.

Meanwhile Back at the Server...

Remember this all started with the Server sending a Timeout-Stop command to your application's GS32.DLL.

The Server keeps track of these, and will send two of these Timeout-Stop commands at one minute intervals, unless of course your application terminates properly. (See! Even more chances for your application to wake up and die right!)

One minute after the second of these Timeout-Stop commands the server will use an even more brutal hammer to kill your job.

This is another Microsoft Windows API called ***TerminateProcess.*** Its equivalent to you selecting the task in the windows Task Manager, and terminating it. Its final, immediate, and there is nothing at all graceful about it.

So if GS32 is not getting control (your program is in a tight loop or not calling GS), it still gets killed about three minutes after the Timeout period expired.

TerminateProcess does not give your job a chance to close files, release locks or anything else. It just kills the program and releases any resources it may have held, and sets the exit status to 222.

You can see its far better for the programmer to properly handle Close-And-Stop and Mandatory-Stop events than letting windows kill your programs.

Un-killable Tasks.

There are occasionally tasks in the windows environment that just refuse to die, and the Server can not kill these. Typically these can't be killed by Task Manager either. Reboot is often required for these. Should the Server encounter one of these tasks that TerminateProcess can't even kill, it simply abandons it, and it does not count against your licensed number of seats.

Testing Timeouts.

At the Server control panel, you can double click any running task and a popup will appear from which you can choose to ask the task to terminate (send it a Timeout-Stop), or immediately terminate it (use the windows TerminateProcess API).

If you have programmed your application in the recommended way (always honoring Close-And-Stop) OR you have a properly coded GS.COB (which honors Mandatory-Stops), your task should terminate immediately when you click “Request the Task to Terminate”. We strongly urge you to test this before using Timeouts. We strongly urge you to test it in that part of your application MOST likely to lose data or corrupt your database and evaluate the effects

Pay special attention to GS.COB. It can cover a lot of programming sins, by doing a proper shutdown even if you don't want to code proper handling of Close-And-Stop in every part of your application. It may be easier to release records, back out in-flight transactions, and close files in GS.COB than it would be to handle this in dozens of places in your application.

Alternatives.

Applications developed for network deployment (as well as single user deployment) may be better served by coding that takes into account that humans get interrupted, distracted, and suffer flaky communication systems.

You can use simple steps such as coding Panel Timeouts, which return an event to your program if a user sits on a screen too long tying up critical resources). Using panel timeouts is simple, its just one more condition to test in your panel display loop:

```
PERFORM WITH TEST AFTER
   UNTIL panel-CLOSE-AND-STOP OR panel-TIMEOUT
    CALL 'GS' USING panel-1 panel-2 panel-3 panel-4
    ...
    ...
  END-PERFORM
  GOBACK.
```

This way, your application will “step backward” till it finally encounters a the Main Panel, at which time it will exit the application. So the user that walks away from the application will have it exit automatically after a suitable period of time.

This allows you to avoid record contention, allow for batch processing, and allows you to perform file backups even when forgetful users fail to close out applications.

If record contention and batch processing are not issues, the most likely other reason for using any kind of timeout scheme is to allow use of a Server with a number of seats smaller than the number of users, on the assumption that not all users will be connected at once.

While this might provide some economy of scale, it may be a false saving in the end, especially if it requires additional programming and runs the risk of data corruption.

We have found that, given a reliable network, even a reliable Internet connection, the GUI ScreenIO Client/Server Layer is more than capable of keeping a application running over the network for weeks on end. Timeouts may never be needed if your programming practices never lock records or files for more than the millisecond it takes actually update the records.

There may not be a need for timeouts at all.

But if you do need timeouts, the built in Job Timeout mechanism of the Client/Server Layer will likely perform the task efficiently while still affording your application all the opportunity it needs to safeguard your data's integrity.