**CREATE** **OR** **REPLACE** **PACKAGE** logs

 **AUTHID** **CURRENT\_USER**

*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**

*%author*

*Bill Coulam (bcoulam@dbartisans.com)*

 *A collection of routines responsible for logging messages to some output*

 *device, be it the screen, a file and/or a logging table. Where messages are*

 *directed depends upon the targets specified in the "Default Log Targets"*

 *parameter (see usage and notes for further info).*

*%usage*

 *The most simple and common use of this package is to call logs.dbg() to comment*

 *AND instrument your code, logs.msg() whenever you want to record something*

 *important, and logs.err() when you need to record context of variables and*

 *database state when exceptions are trapped and handled.*

 *<code>*

 *logs.dbg('Attempting to open file '||l\_file\_nm);*

 *...*

 *logs.msg('Daily refresh started at '||dt.get\_systs);*

 *...*

 *logs.err('Request on pipe '||l\_pipe\_nm||' timed out.');*

 *However, this API is rather flexible and provides the parameters and overloaded*

 *routines needed to handle most any typical logging requirement.*

 *If you wish logging to be directed at the filesystem, you must set up a few*

 *parameters in the framework's parameter structures (APP\_PARM, APP\_ENV\_PARM).*

 *The parameters required for a file destination are "Default Log File Directory"*

 *and "Default IO File Name".*

*%note*

 *Logging to the screen and logging table can only be turned on or off, not*

 *redirected somewhere else. Of the three logging targets, only file logging can*

 *be redirected to a specific directory and/or file. To explicitly change the*

 *logging directory, change the value of the "Default Log File Directory"*

 *parameter. To change the logging file name, change the value of the*

 *"Default IO File Name" parameter. Or, you can dynamically change either or both*

 *by using logs.set\_file\_parms(), logs.set\_file\_dir() and/or logs.set\_file\_nm()*

 *for the session.*

 *By default, messages to logs.dbg() are suppressed. If you wish these to*

 *begin appearing in your logging targets, you must change the value of the*

 *"Debug" parameter in APP\_ENV\_PARM. If you are writing unit tests or trying to*

 *replicate a bug using a PL/SQL client or test harness, you can bypass the*

 *table-based debug toggle altogether and override current debug settings by*

 *calling set\_dbg(TRUE).*

 *The name of this package should have been LOG, but LOG is an Oracle keyword,*

 *so I had to use a plural noun, instead of the active verb like I was hoping,*

 *otherwise various PL/SQL programming editors would uppercase the word "log"*

 *every time you tried to call this package, which is opposite to the keyword*

 *case style rule of most shops.*

*%design*

 *PRIMARY USE OF LOGS*

 *The three primary log routines dbg(), msg() and err() are meant to handle all*

 *verbose debugging output, application logging and error recording. But one*

 *can also use warn() and info() which wrap msg(), making informational and*

 *warning messages easier to send.*

 *LOGGING CONTENT TYPE*

 *Application messages can be debug, exception/error, informational and warning*

 *messages (see the CNST package for the message type code constants). I refer*

 *to informational and warning messages as "application logging."*

 *Application Logging*

 *Application logging generally involves recording useful processing status*

 *and context, audit trail data, records handled, before and after control*

 *states, etc. Use logs.msg() with severity of cnst.INFO, or use logs.info()*

 *to do application logging.*

 *Error Logging*

 *Error logging involves recording variable state and parameter context at the*

 *time and point of error. Use logs.msg() with severity of cnst.ERROR, or use*

 *logs.err() to do error logging.*

 *Warnings*

 *There are also warning messages that fall somewhere between application*

 *logging and error handling. They are worrisome conditions that someone should*

 *look at within the next few hours or days to determine if there is something*

 *more sinister going on that warrants deeper attention. Use logs.msg() with*

 *severity of cnst.WARN, or use logs.warn() to send warnings.*

 *Debug Logging*

 *Debugging messages contain detailed, low-level context that only a programmer*

 *would appreciate, so they can quickly see exactly which paths a program took*

 *and what happened at each step along the way. Use logs.dbg() for these. Use*

 *logs.dbg() liberally so that when the inevitable production bug pops up, it is*

 *trivial to turn on debugging (%see logs.dbg below) and immediately see where*

 *things went wrong.*

 *SUGGESTED LOG TARGETS*

 *In development the targets could be set to the screen and table, both readily*

 *useable. In testing, since little will be tested with SQL\*Plus, logging to the*

 *screen will usually be turned off. In production, logs are sent to either the*

 *log table or a file, but not both (too many moving parts to manage/monitor),*

 *and never to the screen. I prefer to table since it is readily available for*

 *query, mining and reporting.*

*%future*

 *Might add the ability to send output to a named pipe, so that a 3GL application*

 *could provide a constant monitor into database messages.*

*<pre>*

*Artisan Date Comments*

*============ ========= ========================================================*

*bcoulam 1997Dec30 Creation*

*bcoulam 2008Feb08 Refactored heavily from the msg package.*

*bcoulam 2008Mar10 Added explicit getters and setters for directory and log*

 *file access.*

*bcoulam 2008May15 Added line number as an optional parameter to most logging*

 *routines.*

*bcoulam 2008May20 Added fine-grained filters to debug mode, so debug logs*

 *only get written for certain packages, session or user.*

*<i>*

 *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ LGPL License \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

 *Copyright (C) 1997-2008 Bill Coulam*

 *This library is free software; you can redistribute it and/or*

 *modify it under the terms of the GNU Lesser General Public*

 *License as published by the Free Software Foundation; either*

 *version 2.1 of the License, or (at your option) any later version.*

 *This library is distributed in the hope that it will be useful,*

 *but WITHOUT ANY WARRANTY; without even the implied warranty of*

 *MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU*

 *Lesser General Public License for more details.*

 *You should have received a copy of the GNU Lesser General Public*

 *License along with this library; if not, write to the Free Software*

 *Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA*

*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/*

**AS**

*--------------------------------------------------------------------------------*

*-- PUBLIC CURSORS*

*--------------------------------------------------------------------------------*

*--------------------------------------------------------------------------------*

*-- PUBLIC TYPES*

*--------------------------------------------------------------------------------*

*--------------------------------------------------------------------------------*

*-- PUBLIC CONSTANTS, VARIABLES, EXCEPTIONS, ETC.*

*--------------------------------------------------------------------------------*

TARGET\_SCREEN **CONSTANT** **VARCHAR2**(10) := 'Screen';

TARGET\_FILE **CONSTANT** **VARCHAR2**(10) := 'File';

TARGET\_TABLE **CONSTANT** **VARCHAR2**(10) := 'Table';

*--TARGET\_PIPE CONSTANT VARCHAR2(10) := 'Pipe';*

DEBUG\_PARM\_NM **CONSTANT** app\_parm.parm\_nm%**TYPE** := 'Debug';

*--------------------------------------------------------------------------------*

*-- PUBLIC FUNCTIONS*

*--------------------------------------------------------------------------------*

*/\*\*-----------------------------------------------------------------------------*

*get\_targets:*

 *Returns the current target(s) receiving output as a delimited string (useful*

 *when debugging the operations of the LOGS routines).*

*------------------------------------------------------------------------------\*/*

**FUNCTION** get\_targets **RETURN** **VARCHAR2**;

*/\*\*-----------------------------------------------------------------------------*

*get\_log\_dir:*

 *Returns the name of the directory being used as the destination for file logs.*

 *If the caller has not explicitly set the directory using set\_log\_dir() or*

 *set\_log\_parms(), this will be the directory specified by the*

 *"Default Log File Directory" parameter.*

*------------------------------------------------------------------------------\*/*

**FUNCTION** get\_log\_dir **RETURN** **VARCHAR2**;

*/\*\*-----------------------------------------------------------------------------*

*get\_log\_nm:*

 *Returns the name of the logging file. If the caller has not designated a*

 *specific file name via set\_log\_nm() or set\_log\_parms(), this will return the*

 *Default IO File Name (see io.get\_default\_filename).*

*------------------------------------------------------------------------------\*/*

**FUNCTION** get\_log\_nm **RETURN** **VARCHAR2**;

*/\*\*-----------------------------------------------------------------------------*

*get\_log\_path:*

 *Returns the full path and name of the logging file. If the caller has not*

 *designated a specific file via set\_log\_nm() or set\_log\_parms(), this will amount*

 *to the default directory path and default file name.*

*------------------------------------------------------------------------------\*/*

**FUNCTION** get\_log\_path **RETURN** **VARCHAR2**;

*--------------------------------------------------------------------------------*

*-- PUBLIC PROCEDURES*

*--------------------------------------------------------------------------------*

*/\*\*-----------------------------------------------------------------------------*

*set\_targets:*

 *Routine meant to temporarily (for this session) override the default log*

 *destinations specified by the parameter "Default Log Targets".*

*%design*

 *Logging can be routed to stdout, the APP\_LOG table, a log file, all three, or*

 *any combination. The default logging destinations are controlled by a record*

 *named "Default Log Targets" in APP\_PARM and APP\_ENV\_PARM. You should create a*

 *"Default Log Targets" record in these tables for each environment. The parm\_val*

 *for "Default Log Targets" should adhere to this scheme:*

 *"Screen=Y|N,Table=Y|N,File=Y|N"*

 *You would only call this routine, logs.set\_targets(), if you need to*

 *temporarily override the defaults set by that parameter.*

 *If set\_targets isn't called AND "Default Log Targets" isn't configured, all*

 *logging will default to the table target (APP\_LOG).*

 *If you set the file toggle to TRUE, the filename will default to what is*

 *specified by the "Default IO File Name" parameter that should have already been*

 *set up for the IO package. If you wish the log file name to be different from*

 *the default you should use logs.set\_log\_nm() to change it. This will remain in*

 *effect for the session.*

 *One call at the top of the driving procedure to set\_targets() is usually*

 *sufficient. If, in the middle of your code you have a special block that*

 *needs to go to a different target than that set for the rest of the session,*

 *you may call logs.to\_table() or logs.to\_file() directly.*

*%usage*

 *<code>*

 *BEGIN*

 *logs.set\_targets(FALSE,TRUE,TRUE);*

 *-- OR optionally use named notation, like so:*

 *logs.set\_targets(*

 *i\_stdout => FALSE*

 *, i\_table => TRUE*

 *);*

 *io.p(... -- ignores targets, goes to screen*

 *logs.dbg(... -- uses targets if debug toggle is turned on*

 *logs.msg(... -- uses targets*

 *END;*

 *<code>*

*%param i\_stdout TRUE means log messages will be routed to the screen (via io.p).*

*%param i\_table TRUE means log messages will be routed to the APP\_LOG table.*

*%param i\_file TRUE means log messages will be routed to a file.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** set\_targets

(

 i\_stdout **IN** **BOOLEAN** **DEFAULT** **FALSE**,

 i\_table **IN** **BOOLEAN** **DEFAULT** **FALSE**,

 i\_file **IN** **BOOLEAN** **DEFAULT** **FALSE**

);

*/\*\*-----------------------------------------------------------------------------*

*set\_log\_parms:*

 *Sets the target directory and/or file name for all logging. This directory and*

 *file name are set system-wide by the "Default Log File Directory" and*

 *"Default IO File Name" parameters seen in APP\_PARM\_VW. If you wish them to be*

 *other than the default, call this routine to change one, or the other, or both*

 *explicitly. If you leave either of the parameters blank, the default will be*

 *used instead.*

*%param i\_file\_dir The name of the directory where you wish log files to be*

 *written if different than the default.*

*%param i\_file\_nm The name of the file if you wish the logging to go to a file*

 *named other than the default.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** set\_log\_parms

(

 i\_file\_dir **IN** **VARCHAR2** **DEFAULT** io.get\_default\_filename,

 i\_file\_nm **IN** **VARCHAR2** **DEFAULT** io.get\_default\_dir

);

*/\*\*-----------------------------------------------------------------------------*

*set\_log\_dir:*

 *Sets the target directory for all logging, overriding the directory indicated by*

 *the system-wide "Default Log File Directory" parameter.*

*%param i\_file\_dir The name of the logging directory (should match the name of*

 *an Oracle directory object).*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** set\_log\_dir(i\_file\_dir **IN** **VARCHAR2**);

*/\*\*-----------------------------------------------------------------------------*

*set\_log\_nm:*

 *Sets the target file name for all logging, overriding the fiel name indicated*

 *by the system-wide "Default IO File Name" parameter.*

*%param i\_file\_nm The name of the logging file.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** set\_log\_nm(i\_file\_nm **IN** **VARCHAR2**);

*/\*\*-----------------------------------------------------------------------------*

*set\_dbg:*

 *Toggles the state of debugging for the current session in which it is called.*

 *This method of turning on debugging is really meant only for development where*

 *unit tests are being conducted through SQL\*Plus scripts. If you need to turn*

 *debugging on in production, use the "Debug" parameter in APP\_ENV\_PARM.*

 *%see logs.dbg() for further info on the dynamic debug toggle.*

 *set\_dbg(BOOLEAN) is meant for SQL\*Plus and PL/SQL-fluent callers.*

 *TRUE turns debugging on*

 *FALSE turns it off*

 *set\_dbg(VARCHAR2) is meant for non-Oracle speakers, like Java and other layers*

 *in the application stack that might need to persist debugging messages.*

 *'all','on','y','yes','true' all turn debugging on*

 *'none','off','n','no','false' all turn debugging off*

 *'session=','unit=','user=' will filter debugging (%see logs.dbg for explanation)*

*%usage*

 *Developer logs into SQL\*Plus or writes an anonymous block. In either*

 *case, the developer calls logs.set\_dbg(TRUE);*

 *Then from the same session or PL/SQL block, the developer runs the desired*

 *PL/SQL routine.*

 *Any calls to logs.dbg in the underlying layers will then be routed to the*

 *target(s) set either through the "Default Log Targets" parameter in*

 *APP\_ENV\_PARM, or through the set\_targets overriding routine. If you do not set*

 *any logging targets for the session through either method, then all debugging*

 *will default to being routed to the APP\_LOG table.*

*%param i\_dbg\_val Meant for non-PL/SQL callers.*

 *{\*} 'all','on','y','yes','true' all turn debugging on*

 *{\*} 'none','off','n','no','false' all turn debugging off*

 *{\*} 'session=','unit=','user=' will filter debugging (%see logs.dbg for explanation)*

*%param i\_state Meant for SQL\*Plus and PL/SQL-fluent callers.*

 *{\*} TRUE turns debugging on*

 *{\*} FALSE turns it off*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** set\_dbg (i\_dbg\_val **IN** **VARCHAR2**);

**PROCEDURE** set\_dbg (i\_state **IN** **BOOLEAN**);

*/\*\*-----------------------------------------------------------------------------*

*msg:*

 *Call this version of msg() when you have a named, pre-built message in APP\_MSG*

 *to call upon. Pass the message code (app\_msg.msg\_cd) and leave the i\_msg*

 *parameter blank; this will be filled in for you as the framework looks up the*

 *message based on the message code. Ensure you let msg() know what type of*

 *message you want logged (%see cnst.DEBUG, cnst.INFO, cnst.WARN and cnst.ERROR).*

 *If are inventing a one-off message on the spot, you may pass*

 *any short string, name or identifier you wish for the message code AND a*

 *message. You might use msgs.DEFAULT\_MSG\_CD for your ad-hoc message code. If the*

 *i\_msg parameter is filled, log will not bother to look up the code in app\_msg.*

 *All three msg() routines are built to be used within exception handlers. If*

 *you just want the message to be logged, leave the i\_reraise parameter blank. If*

 *you want to halt processing, set i\_reraise to TRUE. For example, wrap your*

 *PL/SQL in a sub-block and give it its own exception block. This allows you to*

 *trap errors, log them if you wish and continue with the next statement or*

 *iteration, e.g.*

 *Remember to explicitly pass in ERROR or WARN for the severity. If you don't, your*

 *exceptions will default to INFO messages and won't be picked up by the log-*

 *scanning application. The log-scanning app does not come with the framework. It*

 *must be custom-built per each shop's needs. It is usually implemented as a cron*

 *or scheduled Oracle job that reads through the latest N minutes of APP\_LOG*

 *records, recording the last-scanned timestamp somewhere. It then emails certain*

 *people or groups with high-severity errors so that issues in production can be*

 *proactively detected and sometimes handled before the end users even notice or*

 *report them.*

*%design*

 *I was unable to give i\_sev\_cd a default. By doing so, the 1st and 3rd overloads*

 *of msg() conflicted. They would compile, but would conflict at runtime.*

*%usage*

 *<code>*

 *--- simple use of logs.msg (3rd overload)*

 *logs.msg('Parameter was '||l\_length||' characters long. Must be 500 or less.');*

 *-- better use, inventing message on the spot*

 *logs.msg('Invalid Parameter', cnst.ERROR, 'Parameter must be 500 characters or less');*

 *-- even better use of the canned messages in APP\_MSG*

 *logs.msg('Invalid Parameter', cnst.ERROR);*

 *-- best use of LOGS and parameterized canned messages in APP\_MSG*

 *logs.msg('Invalid Parameter', cnst.ERROR,*

 *msgs.fill\_msg('Invalid Parameter', 'i\_copyright', '500'));*

 *BEGIN*

 *-- do some stuff, probably start a loop*

 *FOR i in ...*

 *BEGIN -- begin sub-block*

 *... do more stuff*

 *EXCEPTION*

 *WHEN lx\_whatever THEN*

 *logs.msg('Invalid Product', cnst.WARN);*

 *WHEN DUP\_VAL\_ON\_INDEX THEN*

 *logs.msg(c.DEFAULT\_MSG\_CD, cnst.ERROR,*

 *'That row is already in '||l\_table, TRUE);*

 *WHEN OTHERS THEN*

 *logs.err(TRUE);*

 *END; -- end sub-block*

 *END; -- end outer block*

 *</code>*

*%future*

 *Add parameterized versions of logs.msg to match the signature of msgs.fill\_msg*

 *so that msgs.fill\_msg does not need to be called explicitly. It could then be*

 *called implicitly by logs.msg when it detects parameters are being fed to it*

 *to replaced placeholders in the canned message.*

*%param i\_msg\_cd Named message, found in APP\_MSG table. You can also invent one*

 *on the spot, like "My Awesome Message", or use the default Ad-Hoc*

 *Message code found in msgs.DEFAULT\_MSG\_CD.*

*%param i\_sev\_cd Message severity. cnst.ERROR, cnst.WARN, cnst.INFO, cnst.DEBUG*

*%param i\_msg Error/Application message. Will be truncated if longer than 32K.*

*%param i\_reraise TRUE or FALSE. Whether you want the exception raised again*

 *(which will halt most programs unless they have a higher-level*

 *exception handler).*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** msg

(

 i\_msg\_cd **IN** app\_log.msg\_cd%**TYPE**,

 i\_sev\_cd **IN** app\_log.sev\_cd%**TYPE**,

 i\_msg **IN** **VARCHAR2** **DEFAULT** **NULL**,

 i\_reraise **IN** **BOOLEAN** **DEFAULT** **FALSE**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*msg:*

 *This version of msg() is primarily used in exception handlers when dealing with*

 *built-in or application-specific error codes. The Core framework provides a way*

 *to associate error IDs with each standard message (%see APP\_MSG). However, it is*

 *much easier, and recommended, to mainly deal with and code to the message code,*

 *not the numeric identifier.*

 *Do not call RAISE\_APPLICATION\_ERROR explicitly. Use this version of logs.msg()*

 *instead. If reraise is TRUE and the ID is positive (from APP\_MSG or invented*

 *ad-hoc), or if the ID is in the -20000 to -20999 range, RAISE\_APPLICATION\_ERROR*

 *will be called for you after logging the message.*

 *Remember to explicitly pass in ERROR or WARN for the severity. If you don't, your*

 *exceptions will default to INFO messages and won't be picked up by the log-*

 *scanning application. The log-scanning app does not come with the framework. It*

 *must be custom-built per each shop's needs. It is usually implemented as a cron*

 *or scheduled Oracle job that reads through the latest N minutes of APP\_LOG*

 *records, recording the last-scanned timestamp somewhere. It then emails certain*

 *people or groups with high-severity errors so that issues in production can be*

 *proactively detected and sometimes handled before the end users even notice or*

 *report them.*

*%usage*

 *<code>*

 *EXCEPTION*

 *WHEN excp.gx\_row\_locked THEN*

 *-- This example uses SQLCODE, but substitutes a contextual message*

 *-- then proceeds with processing*

 *logs.msg(SQLCODE, cnst.INFO, 'Another session has item'||l\_item\_id||' locked.');*

 *WHEN lx\_control\_violation THEN*

 *-- This example uses a programmer-defined error ID, raising an error*

 *-- to the GUI*

 *logs.msg(-20001, cnst.WARN, 'The number of records processed does not equal*

 *the number of records input.', TRUE);*

 *WHEN OTHERS THEN*

 *-- This example uses the err() incarnation of msg which*

 *-- automatically logs an ERROR-level SQLERRM and re-raises the exception.*

 *logs.err;*

 *</code>*

*%param i\_msg\_id Positive or negative error number, either from APP\_MSG or from*

 *list of Oracle error num.*

*%param i\_sev\_cd Message severity. cnst.ERROR, cnst.WARN, cnst.INFO, cnst.DEBUG*

*%param i\_msg Error/Application message. Will be truncated to 32K if longer.*

*%param i\_reraise TRUE or FALSE. Whether you want the exception raised again*

 *(which will halt most programs unless they have a higher-level*

 *exception handler).*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** msg

(

 i\_msg\_id **IN** app\_msg.msg\_id%**TYPE**,

 i\_sev\_cd **IN** **VARCHAR2**,

 i\_msg **IN** **VARCHAR2** **DEFAULT** **NULL**,

 i\_reraise **IN** **BOOLEAN** **DEFAULT** **FALSE**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*msg:*

 *This version of msg is provided to allow very quick and easy additions to the*

 *log targets. Any calls to this version of log will have the severity defaulted*

 *to INFO, the msg code to "Ad-Hoc Msg", and the routine to Unknown if it can't*

 *be determined from the call stack.*

 *This version of msg is especially helpful for quick-and-dirty debugging during*

 *development when lots of errors are causing rollbacks, and you wish to see*

 *what is going on, by taking advantage of the autonomous transactions that the*

 *LOGS routines provide.*

*%param i\_msg Error/Application message. Will be truncated to 32K if longer.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** msg(i\_msg **IN** **VARCHAR2**);

*/\*\*-----------------------------------------------------------------------------*

*err:*

 *This barebones version of err() will automatically fill the msg with SQLERRM*

 *before logging to the targets. It is primarily used in rare (should be outlawed)*

 *WHEN OTHERS sections.*

*%param i\_reraise Defaults to TRUE, which will raise an error after logging the*

 *message. If you wish to prevent the program from halting its*

 *processing, you will need an exception handler, pass FALSE in*

 *this parameter to keep the exception from raising.*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** **err**

(

 i\_reraise **IN** **BOOLEAN** **DEFAULT** **TRUE**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*err:*

 *This version of err is a lazy way of calling logs.msg, as it automatically*

 *assumes a sev\_cd of ERROR and dispenses with standard message codes.*

*%param i\_msg A message about the detected error and its context. Will be sent*

 *to the log targets.*

*%param i\_reraise Defaults to TRUE, which will raise an error after logging the*

 *message. If you wish to prevent the program from halting its*

 *processing, you will need an exception handler, pass FALSE in*

 *this parameter to keep the exception from raising.*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** **err**

(

 i\_msg **IN** **VARCHAR2**,

 i\_reraise **IN** **BOOLEAN** **DEFAULT** **TRUE**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*warn:*

 *warn() is a lazy way of calling logs.msg, as it automatically*

 *assumes a sev\_cd of WARN and dispenses with standard message codes.*

*%param i\_msg The warning you wish to record and its context, which will be sent*

 *to the log targets.*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** warn

(

 i\_msg **IN** **VARCHAR2**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*info:*

 *info() is a lazy way of calling logs.msg, as it automatically assumes a*

 *severity of INFO and dispenses with standard message codes.*

*%param i\_msg The information or notes you wish to record, which will be sent*

 *to the log targets.*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** **info**

(

 i\_msg **IN** **VARCHAR2**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

*/\*\*-----------------------------------------------------------------------------*

*dbg:*

 *Logs debug messages. Enables dynamic "peeking" into the workings and context of*

 *routines without having to attach a debugger, take downtime, recompile code,*

 *etc. Simply pass a detailed, formatted message in the first parameter. The*

 *routine name and line number from which logs.dbg() was called will be found*

 *transparently, unless you choose to pass in the routine and line number*

 *explicitly.*

*%design*

 *Know that the various logs.msg routines are meant for error\_handling and logging*

 *that should always be on. Calls to the logs.dbg routine are transient. They will*

 *only log output when debugging is turned on either by parameter or by override*

 *(see below). If debugging has been switched on, the debug message will be*

 *written to the targets you set by parameter or override (%see set\_targets).*

 *TURNING ON DEBUG MODE BY PARAMETER*

 *In APP\_PARM is a shared parameter named "Debug". Its value in APP\_ENV\_PARM for*

 *a given application and environment follows the syntax:*

 *off|all|session=<session\_id>|unit=<pkg1[,proc1,trigger1,etc...]>|user=<client\_id>*

 *This means there are four "filters" that can be applied to debug logging:*

 *1) all = log all calls to logs.dbg().*

 *2) session = log any calls to logs.dbg() that belong to the given session ID.*

 *3) unit = log any calls to logs.dbg() that come from the given PL/SQL unit(s).*

 *4) user = log any calls to logs.dbg() attributed to the given client identifier.*

 *and of course*

 *5) off = all calls to logs.dbg() will be ignored.*

 *Filters for all, session and user are single-valued. They can't be combined and*

 *they can only have one value. The only filter that is multi-valued is unit.*

 *If you want to show dbg() calls coming out of more than one package, just write*

 *a comma or space-delimited list of package names in the parm\_val column for parameter*

 *"Debug". Here are examples of app\_env\_parm.parm\_val values for parameter "Debug":*

 *off*

 *all*

 *session=18*

 *unit=DRIVER, DAILY\_LOAD\_PKG, GIS\_MAP\_PKG, AIUD\_REF\_TRG*

 *user=doejohn*

 *When done capturing debug messages for your filter, be sure to update parm\_val*

 *back to off.*

 *TURNING ON DEBUG MODE BY OVERRIDE*

 *%see set\_dbg. Just call set\_dbg('on') or set\_dbg(TRUE) to turn debugging on*

 *for your current session. This is usually only used by anonymous PL/SQL blocks*

 *or SQL\*Plus scripts in unit test harnesses.*

 *DEBUG CHECK INTERVAL*

 *This was designed to not impose unecessary overhead in environments with heavy*

 *transaction/record processing. So rather than checking the parameters for an*

 *updated Debug value on every call of logs.dbg(), it only checks every N minutes,*

 *N being another configurable parameter, specified by the value of the*

 *"Debug Toggle Check Interval", which defaults to checking every minute if not*

 *configured.*

 *This means that you cannot turn debug mode on and expect immediate output.*

 *When you discover a session, PL/SQL unit or user that requires a look into the*

 *debug logs of their process, turn on debug mode using the value in APP\_ENV\_PARM*

 *as outlined above, then wait the N minutes before you inform the user they can*

 *try again. At that point, you should be able to monitor the new data in APP\_LOG*

 *or the logging file to see the new debug data.*

 *This polling, table-based design allows you to leave your logs.dbg() calls*

 *peppered throughout your code. There is no need to comment them out or use 10g*

 *conditional compilation syntax to hide them for production. Since we are often*

 *verbose and detailed in debugging/info messages, this is a great way of*

 *documenting the code as well.*

 *DESIGN ALTERNATIVES REJECTED*

 *We rejected the option of checking the parameter table upon every call to*

 *logs.dbg(). We felt this was simply too much overhead for most systems'*

 *performance goals.*

 *We were forced to reject the idea of using global application contexts,*

 *dbms\_pipe or dbms\_alert, as all these mechanisms do not work at all, or well,*

 *in Oracle RAC clusters.*

*%param i\_msg Fully formatted debug message. The format is up to the user of the*

 *framework.*

*%param i\_routine\_nm This will be determined automatically for you. Only pass*

 *this if you want to record a source name different than*

 *what the call stack says. If you do pass this in, it is*

 *usually the package.routine where the message came from.*

 *Could be the name of a trigger, object method, type body, etc.*

*%param i\_line\_num This will be determined automatically for you. Only pass this*

 *in if you want to record a line number for the debug message*

 *that is different from the line on which logs.dbg is called.*

*------------------------------------------------------------------------------\*/*

**PROCEDURE** dbg

(

 i\_msg **IN** app\_log.log\_txt%**TYPE**,

 i\_routine\_nm **IN** app\_log.routine\_nm%**TYPE** **DEFAULT** **NULL**,

 i\_line\_num **IN** app\_log.line\_num%**TYPE** **DEFAULT** **NULL**

);

**END** logs;