Presentation for the
2013 REXX Language Association Symposium
Raleigh, NC
May 5-8, 2013

Embedding ISPF Assets Within REXX Code

By Frank Clarke

High-level overview: Because a REXX program can actually read its own source, it's possible to build sufficient 'smarts' into it to enable unpacking of subsidiary material that has been packaged with the actual code.

For decades, MVS and its successors have been doing read-aheads: by default, five (5) buffers are filled by the first SIO operation; another SIO doesn't happen until all the records in the first buffer have been processed and SIOs are done buffer-at-a-time from then on. Typical half-track blocking on modern DASD is upwards of 20K, so unless a REXX program occupies substantially more than 100K, it will be entirely in main storage within microseconds of being invoked.

If the REXX program also includes embedded subsidiary material (panels, skeletons, help text, tables, &c.) all of that has also been loaded into buffers. Unpacking that material to VIO datasets happens at main-storage speed rather than DASD speed. No I/O beyond the initial load of the program.

All that is needed is a suitable logical framework for identifying the subsidiary material, unpacking it, and making it available via in-storage libraries.

In the mid-90s I was working at GTE Data Services in Tampa writing REXX/ISPF applications for their Configuration Management department. It occurred to me then that it made more sense to issue messages via the program code itself rather than package them into an ISPMLIB member and fetch them via a message-ID – for the simple reason that such messages are almost always specific to a particular application program. The likelihood any given message might be used by more than one program is approximately zero.

Having concluded that messages ought to be internalized to their issuing routine, it was an easy leap to conclude the same for panels and skeletons: the likelihood any single panel or skeleton will be used in more than one program is also approximately zero. There is simply no benefit to storing panels externally to the programs which will utilize them. They are not going to be shared in any but the most unusual circumstances.

As soon as the decision is made to internalize such elements, problems we never knew we had – evaporate: we impose naming conventions on panels and skeletons (as well as many other data forms) in order that we may identify this element in its library as being related to that other element in some other library. If the elements are permanently welded together, there's no need to name <u>any</u> of them in any sort of rigid convention; they can have any names we want to give them; we can even name them (horrors!) to indicate their function.

I developed a rudimentary protocol for identifying embedded ISPF assets, along with the code to extract them to appropriate temporary libraries and to make those libraries active.

In 1998 while working Y2K contracts, I brushed up against one Alastair Gray at Philip Morris in Zurich, Switzerland. Either I discovered him doing embedding of ISPF assets or he discovered me doing it. We struck up a professional relationship, traded notes, and converged our methods such that we wound up using the same code for this task, or code that was so similar as to be fraternal twins. The version I use is called “DEIMBED”.

We start by composing the element text as a comment at the back of the REXX program:

```
/*---- the embedded assets form a single giant comment following
                        the last executable line.         ----------------------*/

/*
)))                 PLIB INSTALL   (an ISPF panel)
)ATTR
  % TYPE(TEXT)    INTENS(HIGH) SKIP(ON)
  + TYPE(TEXT)    INTENS(LOW)  SKIP(ON)
  _ TYPE(INPUT)   INTENS(HIGH) CAPS(ON) JUST(LEFT) PAD('_')
  @ TYPE(OUTPUT) INTENS(HIGH) CAPS(ON) JUST(LEFT)
  $ TYPE(INPUT)   INTENS(HIGH) CAPS(ON) JUST(LEFT)
)BODY WINDOW(68,5)
+
+ ISPTLIB DSN%==>$tlibds                                      +
+      ...CMDS%==>$z    +
+
)INIT
  .ZVARS    = '(CMDTBL)'
  .HELP     = INSTALH
  .CURSOR   = TLIBDS
)PROC
  VER  (&TLIBDS,DSNAME)
  VER  (&CMDTBL,NAME)
  VPUT (CMDTBL,TLIBDS) PROFILE
)END
)))                 PLIB INSTALH      (another ISPF panel)
)ATTR
  % TYPE(TEXT)    INTENS(HIGH)   SKIP(ON)
  + TYPE(TEXT)    INTENS(LOW)    SKIP(ON)
  _ TYPE(INPUT)   INTENS(HIGH)
  ! TYPE(OUTPUT) INTENS(HIGH)   SKIP(ON)
  @ TYPE(OUTPUT) INTENS(LOW)    SKIP(ON)
)BODY EXPAND(||)
%TUTORIAL |-| COMPILE -- Install Shortcut |-| TUTORIAL %Next Selection
===>_ZCMD

+
    Enter a Library datasetname and membername to identify your personal
    command table.  A shortcut will be generated at that location.

   If you do not have a personal command table, leave this information blank
    and the installation step will be skipped.

   It is%HIGHLY RECOMMENDED+that you have a personal command table which can
    be activated as by (e.g.) ADDCMDS.
)PROC
)END
*/
```

Note the “)))”

DEIMBED scans the source code from “sourceline()” (the last line) until it finds the opening comment, collecting data as it goes. Each time it detects a “)))” it pauses to write the collected data to an appropriate library:

```
do while sourceline(currln) <> "/*"
  text = sourceline(currln)
  if Left(text,3) = ")))" then do  /* package the queue          */
    parse var text ")))" ddn mbr .   /* PLIB PANL001  maybe     */
   if Pos(ddn,ddnlist) = 0 then do  /* doesn't exist          */
     ddnlist = ddnlist ddn       /* keep track              */
      $ddn = ddn || Random(999)
      $ddn.ddn = $ddn
      address TSO "ALLOC FI("$ddn")" fb80po.0
      "LMINIT DATAID(DAID) DDNAME("$ddn")"
      daid.ddn = daid
      end
    daid = daid.ddn
    "LMOPEN DATAID("daid") OPTION(OUTPUT)"
    do queued()
      parse pull line
     "LMPUT DATAID("daid") MODE(INVAR) DATALOC(LINE) DATALEN(80)"
    end
    "LMMADD DATAID("daid") MEMBER("mbr")"
    "LMCLOSE DATAID("daid")"
    end                           /* package the queue          */
   else push text                  /* onto the top of the stack  */
   currln = currln - 1             /* previous line              */
 end                              /* while                      */
```

The **red** text establishes a (VIO) library on the first occurrence of a given “ddn” token. The **green** text spills the collected data onto the library with the specified membername (“mbr”).

This code, the heart of DEIMBED, must be exercised by a call to DEIMBED early in the program's execution.

Having unpacked the embedded assets and spooled them to libraries, the libraries themselves must be made active:

```
/*
  DEIMBED and LIBDEF local material.
.  ----------------------------------------------------------------- */
   address ISPEXEC

  call DEIMBED

  dd = ""
   do Words(ddnlist)                    /* each LIBDEF DD             */
     parse value ddnlist dd  with  dd ddnlist
     $ddn   = $ddn.dd                  /* PLIB322 <- PLIB            */
     "LIBDEF  ISP"dd "LIBRARY  ID("$ddn") STACK"
   end
   ddnlist = ddnlist dd
```

At this point, anything on any of the generated libraries is available for use by ISPF services or any other process.

When the program finishes, good programming practice dictates that we clean up after ourselves:

```
/*
   Scrub the environment by removing the LIBDEFs for local ISPF
   material.
.  ----------------------------------------------------------------- */
   address ISPEXEC

  dd = ""
   do Words(ddnlist)                    /* each LIBDEF DD             */
     parse value ddnlist dd  with  dd ddnlist
     $ddn   = $ddn.dd                  /* PLIB322 <- PLIB            */
     "LIBDEF  ISP"dd
     address TSO "FREE  FI("$ddn")"
   end
   ddnlist = ddnlist dd
```

What benefits ought we expect to see from handling external program elements in this fashion?

1. A developer will typically invoke ISPF in TEST-mode when doing development. In TEST-mode, ISPF caches nothing, and does all service requests via I/O. Changes made to (e.g.) a panel are thus reflected immediately. You always see the last-saved version. The penalty for this is that all service requests involve I/O, even if they aren't part of the current development effort.

   If TEST-mode is not used, changes to that panel will not be seen because the panel is served out of the cache – which is relatively static. ISPF, however, caches based on DSN+member and a freshly-extracted panel is placed into a different dataset; this newly-extracted member is recognized as a different member than that in the cache and is served when requested. Development can thus be done without TEST-mode and without having to restart ISPF: a huge savings in I/O.

2. During program development or maintenance, if it's necessary to refer to a panel, skeleton, or other asset, generally, these will be found within the current member.

3. When distributing or installing an application, only the enclosing REXX code need be distributed or installed.

4. Naming standards no longer have a benefit and thus no longer have a *raison d'être*. Internally stored elements may have any name and can now be functionally-named. That is: their names may now be truly descriptive.

   Because the libraries are LIBDEF'd into place when needed and LIBDEF'd out of existence when no longer needed, there is no danger of creating a naming conflict, one of the hazards prevented by naming standards.

When used in production, the act of loading the program source also loads all the associated elements such that they are immediately available for packaging to VIO datasets. The I/O savings in development are now replicated for users in production.

In fact, I have been unable to detect any costs related to embedded program elements. There seem to be nothing but benefits.

For examples of how to embed software elements within REXX code, see my REXX code-examples page: http://home.roadrunner.com/~mvsrexx/REXX/. There you will find DEIMBED along with several routines which depend upon it. For sheer utility, I recommend TBLOOK.