Who calls me? → Me → Who do I call?

Recently, and long overdue, I completed converting a large CW5.5 veterinary application to a multi-DLL CW6.3 app. The CW5.5 app contained 927 procedures in 42 modules. Redeploying these procedures to DLLs and keeping straight "who called who" such that none of the DLLs had circular references looked like a lot to keep straight.

The three programs presented in this paper were created to help verify that I had done that. These programs weren't intended to be an end in themselves but as this exercise unfolded and these tools grew it became apparent that they may be useful to the Clarion community. The newsgroups and Clarion Magazine have been a great resource for me and this paper is my attempt to give something back that someone might find useful.

A Little Background for Perspective

After much trial and error I ultimately settled on nine DLLs. This paper doesn't address that exercise but rather the tools that I wrote to help verify that the procedures were deployed to an appropriate DLL. They enabled me to quickly find where some procedure was placed if testing the new restructured app presented some illegal function that might indicate that a procedure would be better placed elsewhere. Internal development documentation is likely to be enhanced as well.

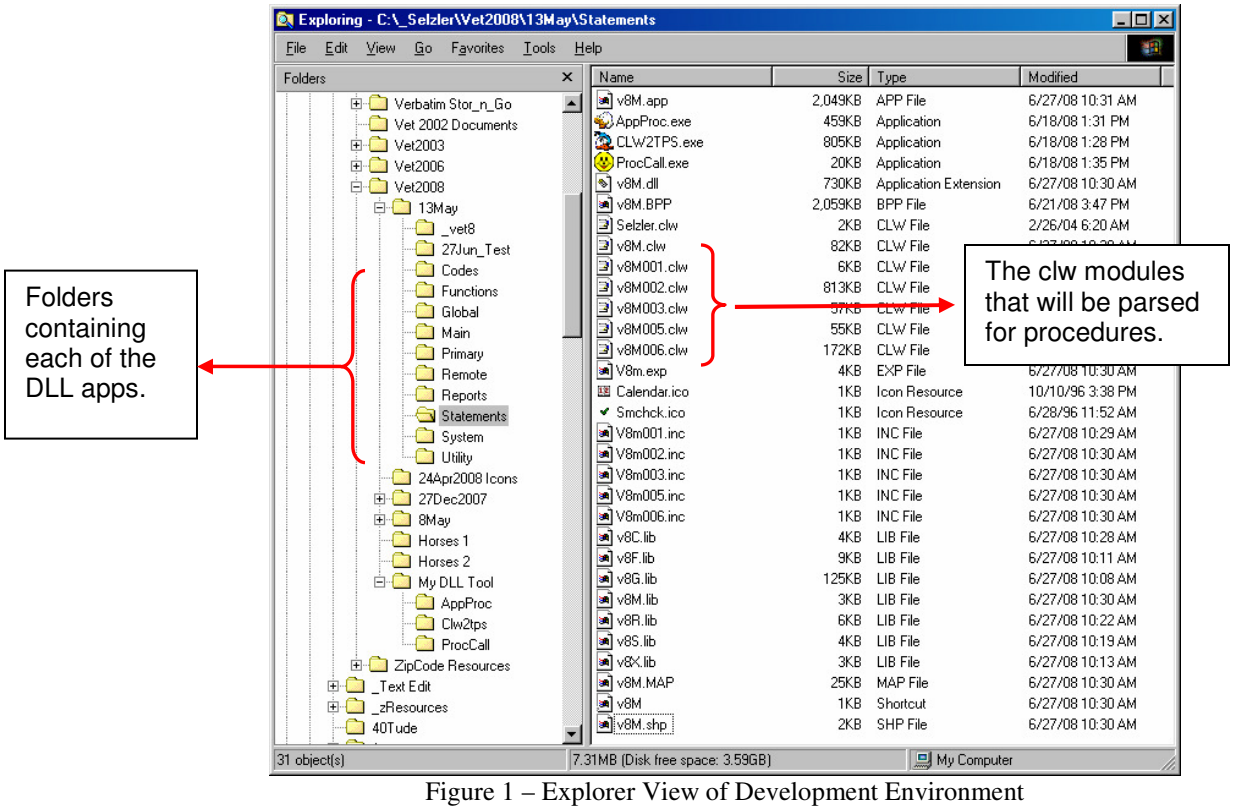


Figure 1 – Explorer View of Development Environment

The app being evaluated will become **Vet8.exe** so each DLL was named **v8** plus a single letter indicative of the DLL's functionality. E.g. Statement procedures are placed in the

v8M.DLL shown above in Fig 1. This isn't important to how one may choose to define the subordinate DLL apps but perhaps will help viewing the screen prints and some of the discussion in this paper. Each **app.DLL** produces a "Default Program" clw file that contains a reference to each module identifying the procedure names in each module.

```
  PROGRAM

  INCLUDE('ABERROR.INC'),ONCE
  INCLUDE('ABFILE.INC'),ONCE
  INCLUDE('ABFUZZY.INC'),ONCE
  INCLUDE('ABUTIL.INC'),ONCE
  INCLUDE('ERRORS.CLW'),ONCE
  INCLUDE('KEYCODES.CLW'),ONCE

  MAP

    MODULE('V8S.DLL')
BrowseProvider              PROCEDURE,DLL              !
BrowseSystemDefaults        PROCEDURE,DLL              !
SelectClient_Short          PROCEDURE,DLL              !
Set_mServiceDate_System PROCEDURE(Long),DLL            ! (Long)
    END
    MODULE('V8X.DLL')
BackupFiles2LocalDisk  PROCEDURE,DLL                   !
    END

!--- Application Global and Exported Procedure Definitions ------------------------------------------
    MODULE('V8M001.CLW')
Main                PROCEDURE   !
    END
    MODULE('V8M002.CLW')
RptStatementAllClients_Alphabetical PROCEDURE(Long)                        !18p -  m:ClientNbr
fStatementOptions      FUNCTION(Byte,*Long,*Long,*Byte,*Long),Byte         !11p -  INI
fConfirmAgeStatementSelections FUNCTION(*Long,*Long,Byte,Long),Byte        !6p - CS(60)
```

Modules with DLL extension are external.

CLW modules are local.

… lots more code evaluated …

As the first program reads this file for procedures and the modules that contained them, it ultimately will encounter **"! Declare"**, below, which indicated that the remainder of that file was no longer of interest and it quit.

```
  ! Declare functions defined in this DLL
v8M:Init            PROCEDURE(<ErrorClass curGlobalErrors>, <INIClass curINIMgr>)
v8M:Kill            PROCEDURE
  ! Declare init functions defined in a different DLL
    MODULE('V8F.DLL')
v8F:Init            PROCEDURE(<ErrorClass curGlobalErrors>, <INIClass curINIMgr>),DLL
v8F:Kill            PROCEDURE,DLL
    END
```

… lots more code but of no interest…

Figure 2 – Default Program Code Snippet

In a single app this was would be in the include files. The programs discussed here focus only on the DLL apps that have already been constructed. I placed each DLL app in its own folder. This shouldn't be necessary but because I was breaking up such a big app it was convenient for me.

Capture All Procedure Names

Each of the "Default Program" **clw** and **inc** files was copied to a working folder: "**_Vet8"** to be read by the first of the three programs and capture all of the primary application procedures into one tps file, Fig 3. The "_" prefix isn't important except to alphabetically segregate the folder names in Explorer.

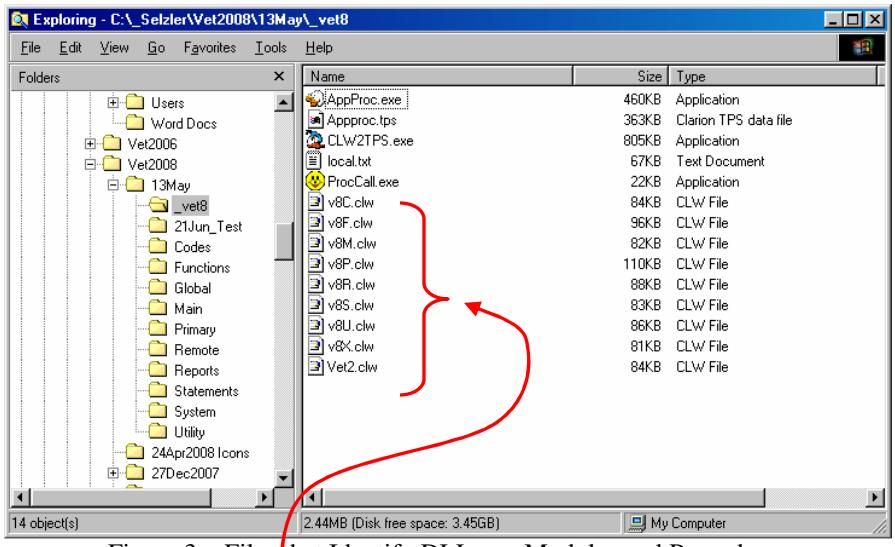


Figure 3 – Files that Identify DLL app Modules and Procedures

When all "Default Program" **clw** and **inc** files are conveniently gathered, run the program: **CLW2TPS**, to read each of those **clw** and **inc** files and accumulate all of the entire app's procedures, Fig 4.

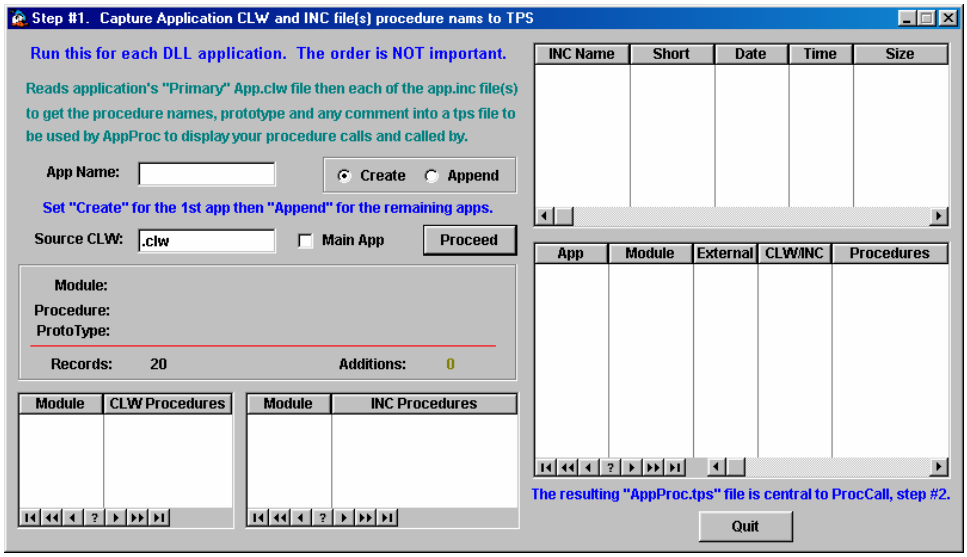


Figure 4 – The CLW2TPS Program

Set "Create" on the 1st **clw** file read and "append" an all others. Order is not important.

CLW2TPS builds the file AppProc.tps identifying each procedure, their source app, member module, name, prototype, whether it was local to this DLL or some external DLL, whether it's a Procedure or a Function and finally any comment. The 3rd program, **AppProc** Fig 6, allows viewing of this data and will be presented in more detail later in this paper.

```
AppProc          FILE,DRIVER('TOPSPEED'),PRE(APR),CREATE,BINDABLE,THREAD
pkAppProcID              KEY(APR:AppProcID),NOCASE,OPT,PRIMARY
kApplication             KEY(APR:Application,APR:ProcedureName),DUP,NOCASE
kModule                  KEY(APR:Module,APR:ProcedureName),DUP,NOCASE
kProcedureName           KEY(APR:ProcedureName),DUP,NOCASE

Record           RECORD,PRE()
AppProcID                LONG
Application              STRING(20)
Module                   STRING(20)
ProcedureName            STRING(100)
FP                       STRING(1)
Prototype                STRING(100)
External                 BYTE
Calls                    LONG
Called                   LONG
RemoteLaptop             BYTE
Config                   LONG
PassMap                  LONG
ModuleData               STRING(1)
GlobalData               STRING(1)
Comment                  STRING(250)
```

The fields: **Calls** through **Comment** were of interest to me and are not critical or even pertinent to this paper.

Figure 5 – AppProc.tps Dictionary

The comments contained information useful to me including my running indication of the number of called procedures that was maintained manually during development. This number was carefully accounted for but not necessarily trustworthy.

APPPROC is the program that presents the results of this capture but at this point it is only beginning to become useful. The evaluated app's procedures have been identified. External procedures are shown in red and the prototypes have a ",DLL" suffix.

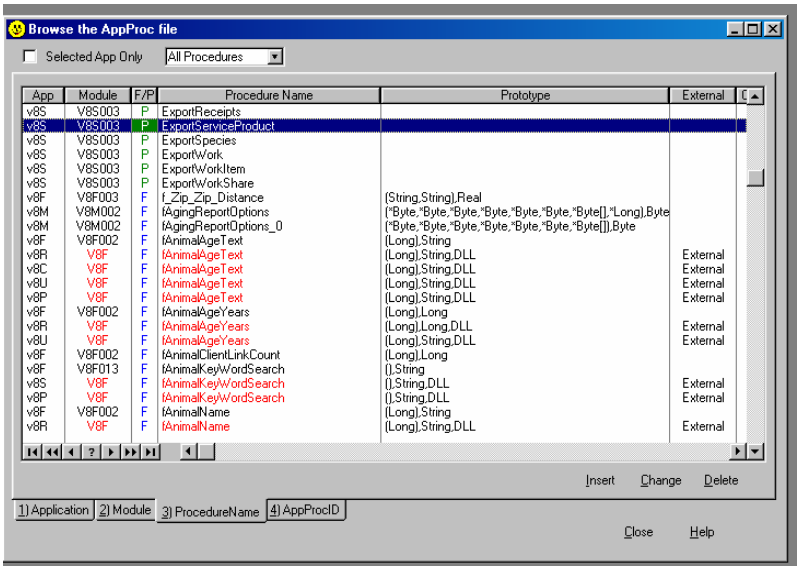


Figure 6 – AppProc Program View of the AppProc.tps Data

Filters allow viewing: a) **All** b) **External** c) **Local** procedures.

At this stage, one only knows where local and external procedures are located, app and module wise. Module names without numeric suffixes are external. This is pretty useful by itself but to know what procedures were called by each procedure and additionally the procedures that call that procedure would be even better. That's where we're headed

Each DLL development folder contains everything necessary to build that DLL, Fig 1. The important files for this exercise are the generated clw files. The "Default Program" un-numbered **clw** file doesn't usually contain any generated procedures unless one has put procedures in that module. I'm guilty of moving procedures from module to module for whatever perverse purpose seemed important at that moment so I am trying to not assume on the practices of other developers.

Rather than bother to exclude processing that file, simply assume that someone might place some procedures there and let these programs check. This minimizes manual interaction to single out or exclude files for the following program,

The Big One! Parse the Code for Procedures They Call.

The **PROCCALL** program allows the user to evaluate either a selected clw file in the folder of interest or all clw files sequentially.

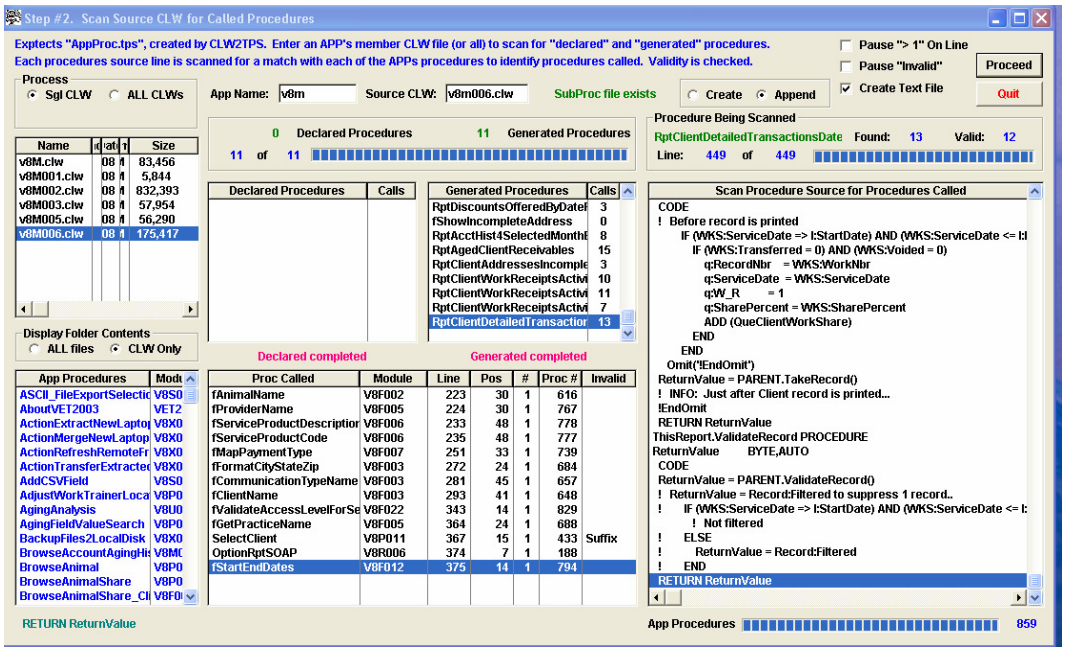


Figure 7 - ProcCall "Who Calls Who" Window.

Who calls me? → Me → Who do I call?

I recommend that one put a copy of the **ProcCall** program in each of the development folders and choose to let it process all clw files. Each clw module file in the folder is scanned line by line for "Declared" and "Generated" procedures. The directory list at the upper left highlights the currently selected module file.

```
  MEMBER('v8M.clw')                          ! This is a MEMBER module

  INCLUDE('ABBROWSE.INC'),ONCE
  INCLUDE('ABPOPUP.INC'),ONCE
  INCLUDE('ABTOOLBA.INC'),ONCE
  INCLUDE('ABWINDOW.INC'),ONCE

              MAP
                INCLUDE('V8M003.INC'),ONCE        !Local module procedure declarations
              END

fInsertNewClientAccountHistoryRecord PROCEDURE  (pClientNbr,pStatement,pInterest) ! Declare
Procedure
r:Return            BYTE
l:LogError          STRING(50)
  CODE
    CLEAR(ACH:AccountHistoryNbr,1)
    SET(ACH:pkAccountHistoryNbr,ACH:pkAccountHistoryNbr)
      IF Access:AcctHist.PREVIOUS() = Level:Benign
        NextLink# = ACH:AccountHistoryNbr
      ELSE
        NextLink# = 0
      END
    ACH:AccountHistoryNbr   = NextLink# + 1
    ACH:ClientNbr           = pClientNbr
    ACH:StatementDate       = pStatement
    ACH:Balance             = fSummAccountBalance (CLI:ClientNbr)
    ACH:Current             = CLI:Current
    ACH:Aged_30Days         = CLI:Accounts30
    ACH:Aged_60Days         = CLI:Accounts60
    ACH:Aged_90Days         = CLI:Accounts90
      IF Access:AcctHist.INSERT() = Level:Benign
          IF fConfigSetting (1) = Enable
            LogTestMessage('AcctHist Record ('& ACH:AccountHistoryNbr &') created.|'   &|
                        '|Client:  '& fClientName (pClientNbr) )
          END
        r:Return = Success
      ELSE
        l:LogError = CLIP(Error()) &', AH# ('& ACH:AccountHistoryNbr &'), C# ('& pClientNbr &')'
        LogVetErrors (l:LogError, 'AcctHist', Insert)
        r:Return   = Failure
      END

                                … lots of code …

ViewClientTrainers PROCEDURE (pClientNbr)          ! Generated from procedure template - Window

l:ClientNbr         LONG                          !
l:ClientName        STRING(40)                    !
Window              WINDOW('Client"s Trainers'),AT(,,217,165),FONT('MS Sans
Serif',8,,FONT:regular),SYSTEM,GRAY,DOUBLE,MDI
                    LIST,AT(8,18,201,120),USE(?List1),VSCROLL,FORMAT('160L(2)|M~Trainer
Name~C(0)@s40@56L(2)|M~Trainer Nbr~C(0)@n-14@'),FROM(QueClientTrainers)
                    BUTTON('Close'),AT(165,145,45,14),USE(?Close)
                  END
ThisWindow          CLASS(WindowManager)

… lots more code …
```

Examples of procedures found within the source

Figure 8 – Snippet of Module Code being Parsed for Procedures

Each source line is parsed specifically for: **"! Declare"** marks the beginning of a series of source lines that ends when either another "**! Declare**" or a "**! Generated from**" occurs which is the beginning of another procedure and brackets procedure source within the module being processed. The check is reversed to process generated procedures.

PROCCALL processes Declared procedures then Generated procedures as two separate sequential passes of each source module. This choice was an arbitrary decision but doing all of one then all of the other made coding a little simpler.

```
LOOP
  NEXT(ApSource)
  IF ErrorCode() = 33 THEN BREAK.
  SourceData = APS:SourceData
    IF SourceData = '' THEN CYCLE.
      CASE DeclaredGenerated
        OF 1
          IF INSTRING('! Declare Procedure',SourceData,1,1) > 0
                ScanSourceLine
            !  FoundProcedure.  Start to fill the source eval queue
          ELSE
            IF INSTRING('! Generated from',SourceData,1,1) > 0
              !  Encountered the other "Type"
                ScanSourceLine
            ELSE
              !  Fill_SourceQue
            END
          END
        OF 2
          IF INSTRING('! Generated from',SourceData,1,1) > 0
```

Figure 9 – Pseudo Code Snippet to Fill the Procedure Evaluation Queue

One couldn't simply search for the word **"Procedure"** because Clarion generates a lot of these as a part of the ABC method structure. I use ABC.

The source of each pertinent procedure, as encountered in the module, is placed in the larger list, at right Fig 7. Then each line is parsed to look for each of the full app's identified procedures, contained in the AppProc.tps file created by the **CLW2TPS** program identified in Fig 4. These procedures are shown at lower left of Fig 7.

```
LOOP J# = 2 TO Records(QueSource)
  GET(QueSource,J#)
  SourceLine = q3:Source
  S# = 1
    LOOP K# = 1 TO Records(QueAppProc)
      GET(QueAppProc,K#)
        IF CLIP(q4:ProcName) = CLIP(CallingProcedure)
          CYCLE
        ELSE
          IF q4:Encountered = 1 THEN Cycle.
              !  Subsequent lines could call the procedure again,
              !  don't care about these.  Save some indicator that this
              !  procedure was already called by the calling procedure.
            ProcedureStart = INSTRING(CLIP(q4:ProcName),SourceLine,1,S#)
              IF ProcedureStart => 1
                DO CheckNotInterested
              END
          END
    END
END
```

Figure 10 – Pseudo Code Snippet to Scan Each Source Line for a Procedure

Each source line in the list at right, Fig 7, is evaluated for all of the procedures present. In my case that was 927 procedure names that were looked for.

Validity Checks.

Just parsing each line of the source for the existence of a string that matches the name of one of the main apps many procedures isn't sufficient. The following considerations are checked for validity.

- ✓ Any procedure may call another many times but only one occurrence of any called procedure is of interest. Subsequent usage is skipped.
- ✓ A "use variable" could have the same name as a procedure. I do that occasionally as it facilitates internal documentation and code maintainability. So when **PROCCALL** gets a procedure hit it checks the character immediately preceding to see if it is a "**?**". It notes the hit in the list at lower center but mark it "Use Var", declares this an invalid call and skips it.
- ✓ The procedure name could be part of a string constant. **PROCCALL** checks the character immediately preceding to see if it is a " **'** ", notes the hit, marks it "String", declares this an invalid call and skips it.
- ✓ The procedure name could be after a "**!**" indicating it was part of a comment. **PROCCALL** checks for the first occurrence of a "**!**" to see if that occurs at an earlier line position than the procedure name hit. If so it notes the hit marks it "Comment", declares it an invalid call and skips the rest of the line.

It gets even trickier. I noticed some habits that I have that could lead to false hits. A procedure name could be contained within another procedure name. It could have some special prefix or suffix. It could be an argument for a CLIP(ProcName), START (ProcName) or something else.

- ✓ Consider:
  - **_ProcedureName**
  - (**ProcedureName** ← this is likely a legitimate hit.
  - Something**ProcedureName**
  - **ProcedureName**Something

A SubProc.tps record is written for each called procedure identified.

SubProc	**FILE,DRIVER('TOPSPEED'),PRE(SPR),CREATE,BINDABLE,THREAD**	
pkSubProcID	KEY(SPR:SubProcID),NOCASE,OPT,PRIMARY	
fkProcedureCalled	KEY(SPR:ProcedureCalled),DUP,NOCASE	
fkCallingProcedure	KEY(SPR:CallingProcedure),DUP,NOCASE	
Record	RECORD,PRE()	
SubProcID		LONG
ProcedureCalled		STRING(100)
CalledProcedureModule		STRING(20)
CountProceduresCallingMe		LONG
CallingProcedure		STRING(100)
CallingProcedureModule		STRING(20)
CountProceduresCalledByMe		LONG

Figure 11 – SubProc.tps Dictionary

Each "declared" procedure processed is identified in the list immediately at the right of the App's module list. Then, serially, each "generated" procedure is processed. These procedures are shown in the list at the right of the "declared" list as they are processed. Any identified procedures parsed from the source lines are shown in the list below the declared and generated procedure lists, bottom center of Fig 7.

Results Summary Upon Completion

When done a Results summary is presented.

Mostly this is a signal of completion The **AppProc** program allows the user to view this data whenever they wish.

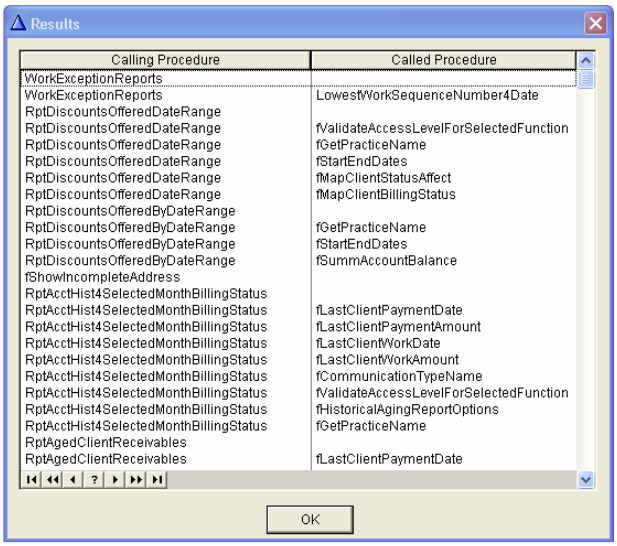
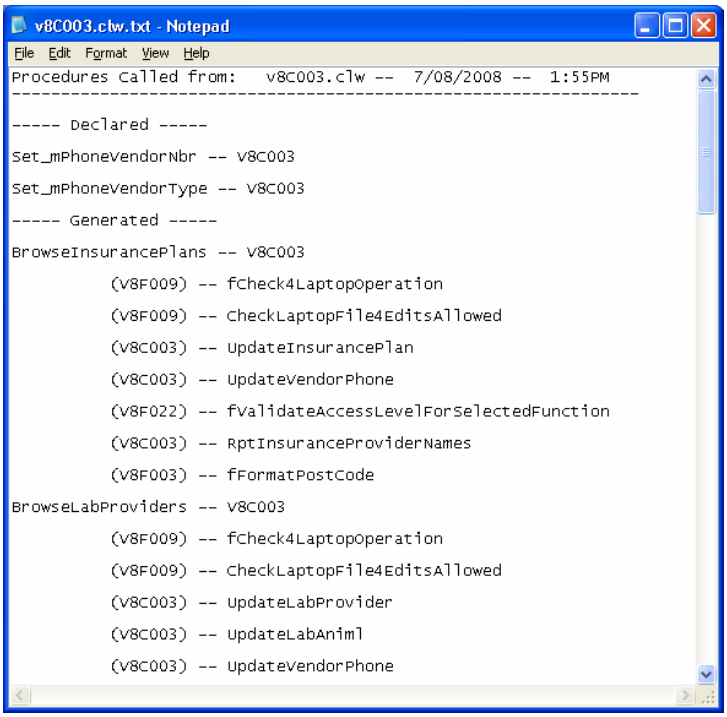


Figure 12 – AppProc Completion Summary

Optional Text file



An optional ASCII text file may be produced for what use that may provide. See the "Create Text File" check box at the upper right of fig 7.

If activated, these are automatically named after the module. E.g. Module **v8m002.clw** would have a text file named **v8m002.clw.txt** which can be viewed by any text editor.

Called procedures, if any, are shown indented following the calling procedure.

Figure 13 – Optional Text File Summary

DLL Hierarchy Considerations

As each DLL app is compiled and export files are created, successive DLL apps will be dependent on their awareness of defined exported procedures. Running **PROCCALL** isn't sensitive to this order. However if you run it in each of the respective development folders, the resulting SubProc.tps file will need to be copied to the folder to be processed next so that that folders procedures can be appended to that growing file.

Use APPPROC to Review it All.

AppProc app, first introduced in Fig 6 is now more useful because of all the SubProc data accumulated by ProcCall, Fig 14.

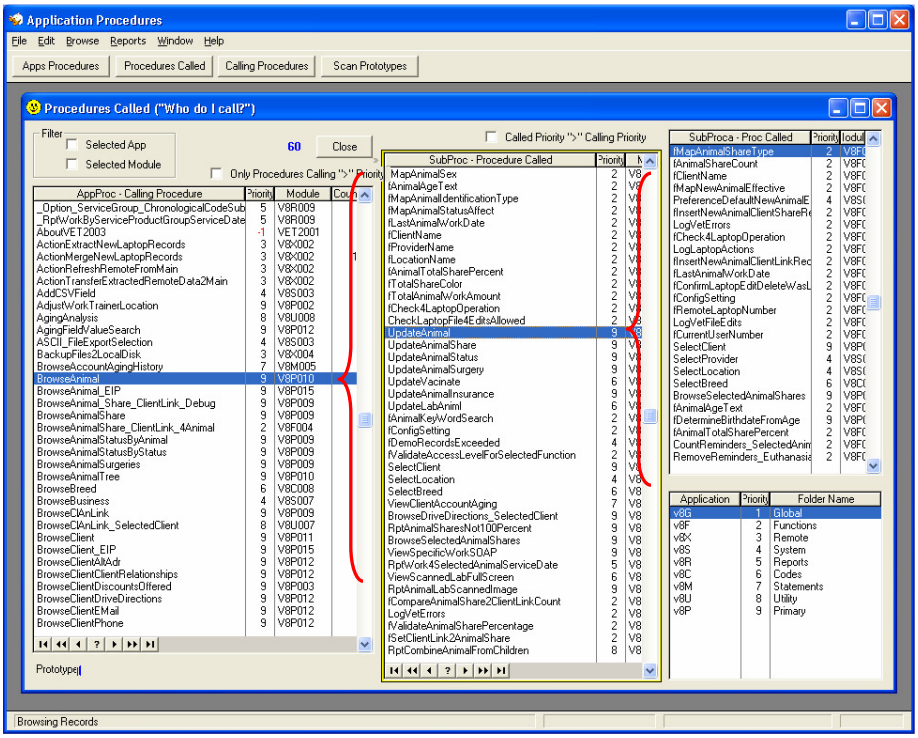


Figure 14 – AppProc "Who do I Call."

Each procedure, left, presents all of the procedures that it calls, center, and they in turn show (if you are interested) the procedures that they call, right. Unlike the procedure tree view in the IDE, the repetition stops at that level. Each of these three lists present their source module and their DLL priority. The lower the priority number the more fundamental the DLL placement.

Procedures are intended to call external procedures equal or lower priority levels. If a procedure calls another with a higher priority number it is presented in red and should be considered to be moved to a DLL with a higher priority number.

The list at the lower right simply reminds of your DLL hierarchy.

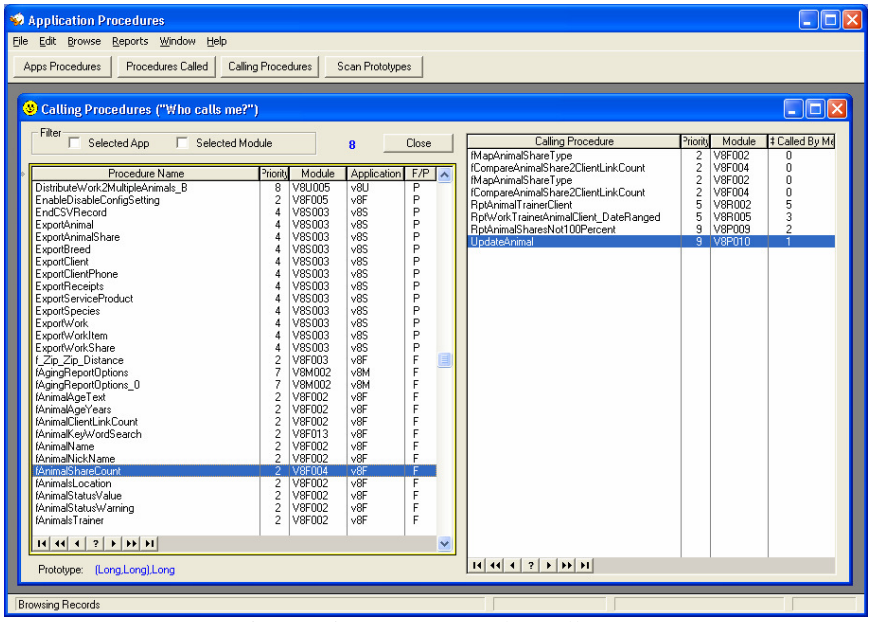


Figure 15 – AppProc "Who Calls Me."

Who calls me presents the SubProc.tps data in reverse. I think it is of less value but interesting never the less.

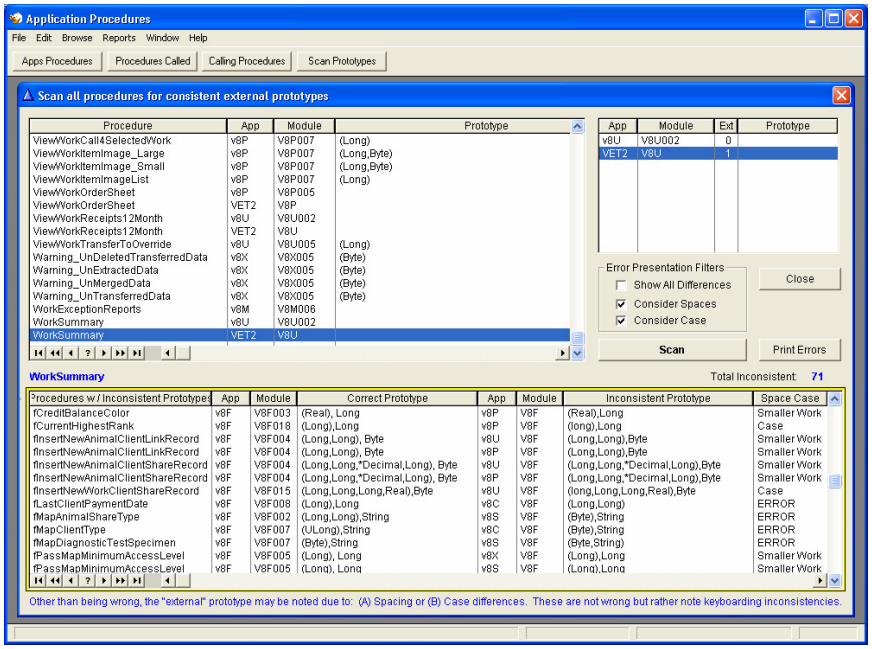


Figure 16 – AppProc Scan Prototypes for Inconsistencies

As one can quickly see from the large number of member procedures, it becomes tedious to visually check all of them for prototype data entry errors. So I created a scan, Fig 16, to find them and then got a little carried away to suit some of my coding conventions

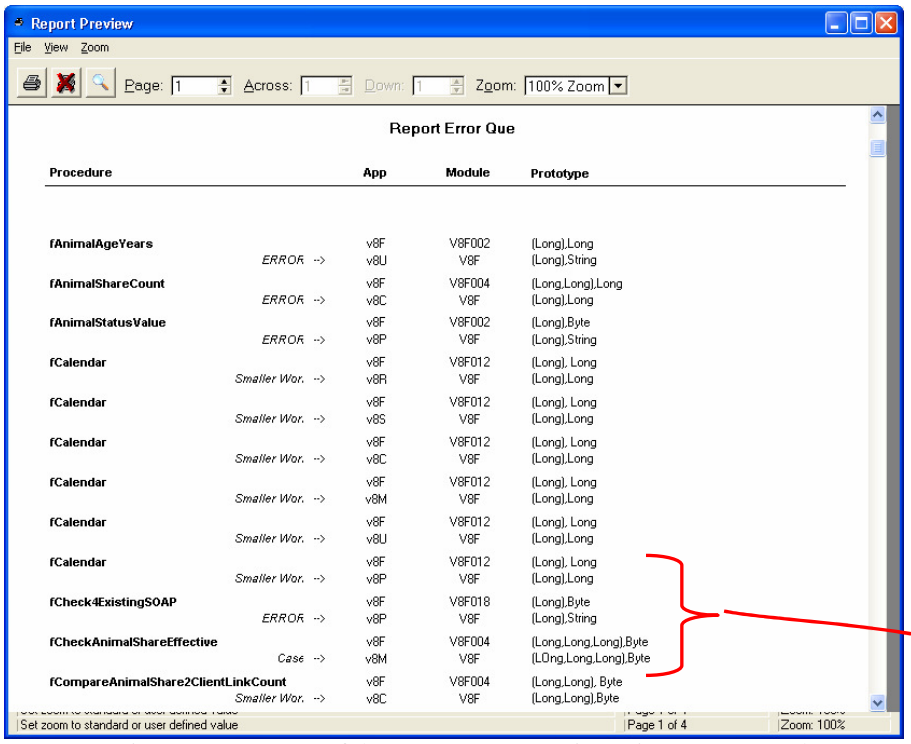


Figure 17 – Report of the Prototype Inconsistencies Encountered

The following enlarged snippet from the report, Fig 17, illustrates a little clearer, the comparisons being checked for inconsistencies. Besides looking for prototype content errors in the external lib modules, I like to retain some typographical conventions. The two that I check for here are font "case" and inconsistent placement of "spaces".

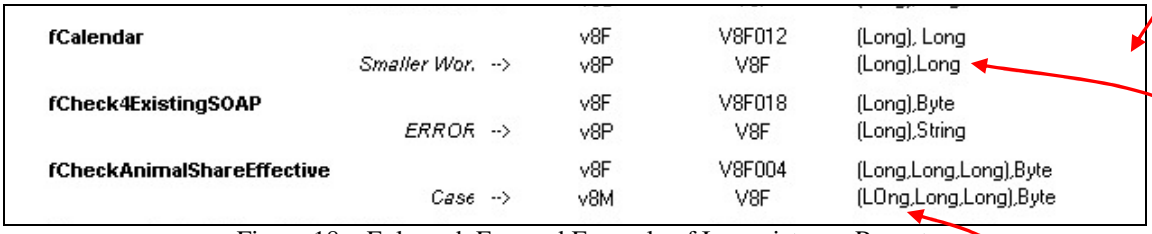


Figure 18 – Enlarged, Focused Example of Inconsistency Report

Different font "case" Inconsistent "spacing"

These are likely not errors (although spaces could be) but rather fussiness on my part. Prototype typing errors, however, can illicit error messages that are cryptic at best.

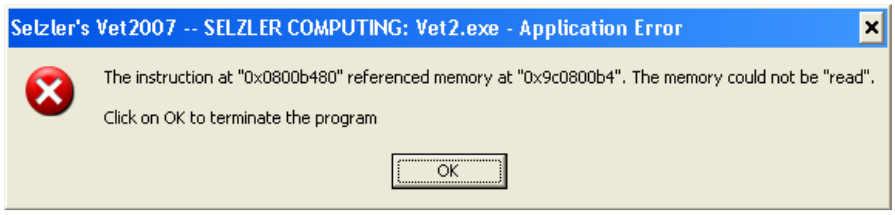


Figure 19 – A Cryptic Error from the Vet8 app being analyzed.

I was presented with the one in Fig 19 that stymied me for awhile until I decided to check some prototype definitions.

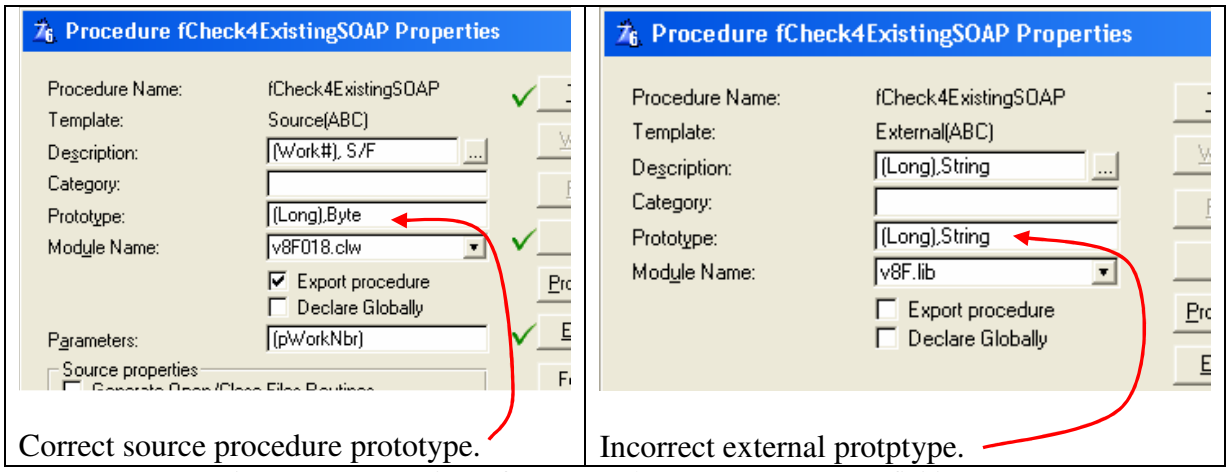


Figure 20 – Comparison of Correct and Incorrect Prototype Definitions.

After creating this check I found a few more errors, which I may or may not have otherwise found or worse a client may call with a message that would have taken a long time for me to find. You all know how clear user descriptions of what they were doing just before an error is presented: "I didn't do any thing wrong!"

Some Code References

On 8 May 2008, I posted: "Curious about many DLL's" to the comp.lang.clarion newsgroup and Maartin responded with a link http://www.softprohk.nl/ReadProc.zip to a program that he had developed. His program is a very useful presentation and the work presented in here is an evolutionary outgrowth of his work. Thank you Maartin.

Note (Dave), not to be part of this paper: Is this legitimate to refer to in this paper? Perhaps I should offer a copy of all of this to him. I wish to give credit to where credit is due.

Summary

These programs are prototypes written using CW 6.3EE that I wrote for myself but I suspect that other Clarion developers might find them or pieces of them useful as their app's are broken into DLLs. The first two are hand coded. The third was a template app.

Another use may facilitate internal development documentation. I usually write some booklets to serve as maps to my apps. User documentation may be served as well but the detail is probably more than the typical user would ever care to look at.

See the companion paper that uses the DllTutor example provided by Soft Velocity.

About Me

I grew up in a wide spot in the road in North Dakota, got a degree in aeronautical engineering from the University of Washington and had a real small part putting Neal Armstrong and Buzz Aldrin on the moon. I've built control software for a chemical laser in assembler and was project engineer responsible for the I/O portion of an operating system for an AN/UYK (PDP-11) for a Marine Corps battlefield computer while at TRW. I consulted to Raytheon on the Patriot missile.

In 1982 I started my own company, Selzler Computing, to write software for equine veterinarians. That first program was written on and for a dual floppy TRS 80. Remember that one? I've built a physician Electronic Medical Record package in CW2.003 and have been using Clarion since version 1 for Windows. My family and I live in Longmont, Colorado.