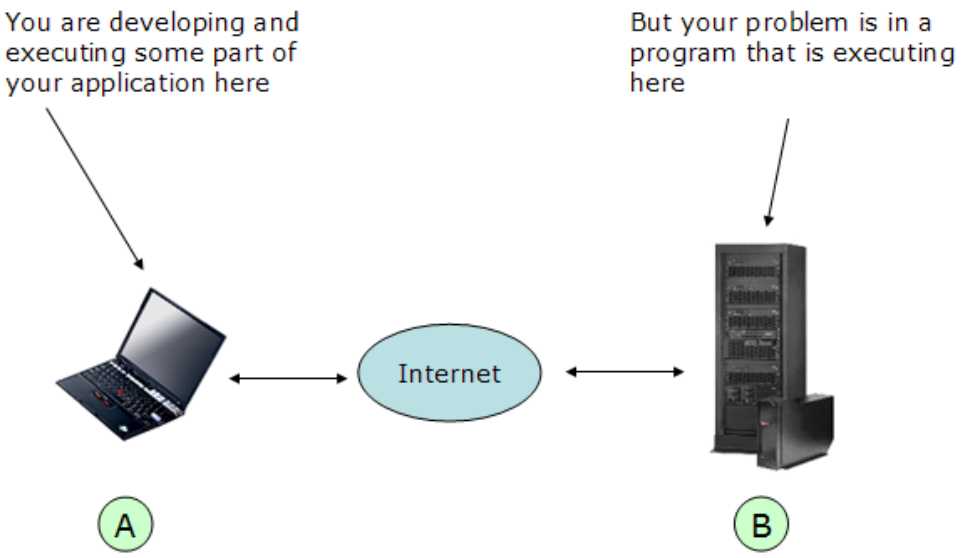
Debugging Applications in Tiered Environments

Developing applications that run on multiple tiers present unique challenges when it comes to locating problems. The challenges are usually related to this situation:



A **might be executing a**	B might be executing a
Web Browser	**WAM (Web Application Module)**
Web Browser	**WEBEVENT function**
Visual LANSA Windows Application	**A function called by CALL_SERVER_FUNCTION**
Visual Basic Application	**Web Service**
LANSA Integrator User Agent	**A function handling the User Agent request**

When dealing with problems like these the first question is always:

Is the program executing on B failing or is it just producing unexpected results?

If it is failing (ie: crashing, freezing or producing operating system or LANSA fatal error messages) then the first thing you need to do is have a look at #Locating Fatal Error Details

If it is just producing unexpected results or executing in an unexpected or apparently inexplicable way then you might look at using #An Application Level Tracing Strategy

Sometimes, in complex error situations you might be asked by your product vendor to send them #System Level Tracing details.

Locating Fatal Error Details

If your application is freezing up or failing on  then do this:

When  is	Action
iSeries Only All Platforms (including iSeries)	**Look for a spool file named QPJOBLOG that was produced by the job that you were running on the iSeries server. Typically this job log contains detailed error information that will help you to isolate the cause of your problem.** **Search the file system (ie: IFS on iSeries) for a file named** X_ERR.LOG**. *Remember to look on the server (ie: B) not on your own PC.* Often this file contains error information that will help you to isolate the cause of your problem.** **If you doing web work also search for a file named** LANSAWEB.LOG**. Sometimes this file contains error information that will help you to isolate the cause of your problem.**

System Level Tracing

If you are asked by your product vendor to produce a system level trace then it will be in a specific area of the LANSA product and they will provide you with detailed instructions on how to do this. Typically system level tracing creates *enormous amounts of trace data*. It also slows your applications down by a large factor, so leaving tracing turned on accidentally can have undesirable side-effects.

An Application Level Tracing Strategy

One way to locate application level problems is to use an *application level tracing strategy* in your RDMLX functions and components.

In some situations producing a trace file mayprove to be a more rapid and effective way to locate a problem than using remotely controlled "code level" program debugging is, especially in production environments.

Following is a generic application level tracing *example* that uses two components named U_TRACEO (the tracing object) and U_TRACEM (the tracing manager).

Here's what you need to know to using them:

Creating reusable components U_TRACEO and U_TRACEM

U_TRACEO and U_TRACEM can be used to trace Components, WAMs and RDMLX functions *on iSeries or Windows server platform* (provided that the LANSA partition is RDMLX enabled). Their source code and creation instructions are included at the end of this document.

Defining U_TRACEM into your programs

To use U_TRACEM you first need to define it in your program:

```
* Declare optional tracing component

Define_Com Class(#U_TRACEM) Name(#Tracer) Reference(*Dynamic)
Define Field(#TraceOn) Type(*char) Length(1) Default(N)
Def_Cond Name(*TraceOn) Cond(#TraceOn = Y)
```

Note that the definition uses Reference(*Dynamic) so that the #Tracer object is only created when it's actually needed. This means there's no real overhead in leaving the definition in your program even when it is promoted to a production environment.

Turning U_TRACEM on (and off)

To use U_TRACEM in your program you need to create an instance of it like this:

```
* Turn tracing on (this code is normally conditioned or commented out)

If_Ref Com(#Tracer) Is(*null)
  Set_Ref Com(#Tracer) To(*Create_as #U_TRACEM)
  #TraceOn := Y
Endif
```

By conditioning or commenting out this code you can turn tracing on or off at will.

Note: Refer to a later section for ideas about how to dynamically condition the execution of trace code.

Using U_TRACEM to produce application level trace information

The *#Tracer.Log* method is used to produce trace details. Here's some examples:

Trace the start of a program …

```
If (*TraceOn)
  #Tracer.Log EVENT(*Function + " started") ISSUER(#Com_Owner)
Endif
```

Trace data values, program flow and loops …

```
Select Fields(#Empno #GiveName #Salary #PostCode) From_File(Pslmst)

       If (*TraceOn)
          #Tracer.Log EVENT("Read employee row") VALUE1(#Empno) VALUE2(#GiveName)
          VALUE3(#Salary.AsString) VALUE4(#PostCode.AsString) ISSUER(#Com_Owner)
       Endif

       * << your logic >>
       * << your logic >>
       * << your logic >>
       * << your logic >>

Endselect
```

Conditional tracing (often hard to do when debugging) …

```
Begin_Loop Using(#Index) To(5000)

   If (*TraceOn)
      If (#Index >= 4995)
         #Tracer.Log EVENT("High end loop " + #Index.AsString + " started")
```

```
            Issuer(#Com_Owner)
         Endif
      Endif

      * << Your logic >>
      * << Your logic >>

   End_Loop
```

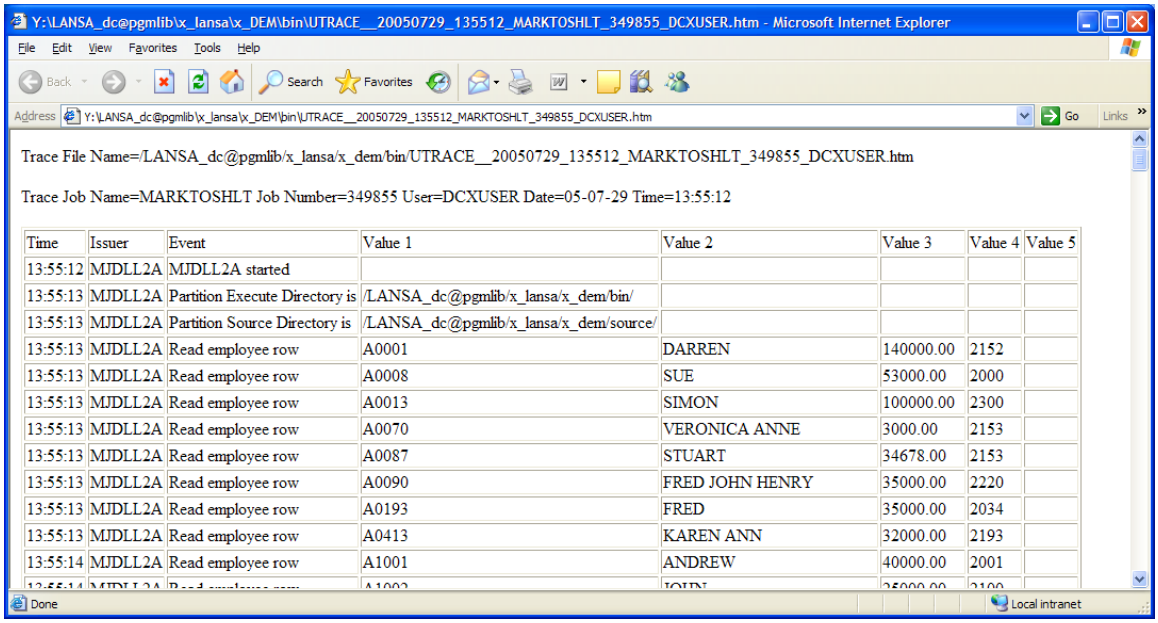
Trace the end of a program …

```
   If (*TraceOn)
         #Tracer.Log EVENT(*Function + " ended") ISSUER(#Com_Owner)
   Endif
```

Locating and Viewing U_TRACEM trace output

U_TRACEM produces a HTML formatted trace output file.

This is an example of a trace file produced on an iSeries server:



U_TRACEM creates trace files named like U_TRACE_xxxxxxxxxxxx.HTM in the current partition's execute directory.

On an iSeries platform the trace file would be created in an IFS directory named something like I:\LANSA_dc@pgmlib\x_lansa\x_DEM\bin\ (where I: is a drive mapped to the iSeries IFS system).

On a Windows server trace files would be created in a folder named something like C:\Program Files\LANSA\X_WIN95\X_LANSA\x_dem\execute.

If you put a short cut on your desktop to I:\LANSA_dc@pgmlib\x_lansa\x_DEM\bin\ (say) then it easy to locate and display any trace files you produce. To view a trace file double click on it and it should display in your web browser.

If a lot of developers are producing trace files then you can make the trace files easier to identify by setting a prefix for the trace file name like this:

```
If_Ref Com(#Tracer) Is(*null)
    Set_Ref Com(#Tracer) To(*Create_as #u_TraceM)
    #TraceOn := Y
    #Tracer.TraceFilePrefix := "MYTRACE"
Endif
```

(You need to set the prefix before you log any trace events).

You can of course modify the U_TRACEO example supplied to produce output in any format anywhere you like.

Using U_TRACEM in called RDMLX functions

When components, WAMs or functions call other components or functions they can all use U_TRACEM simultaneously and a single output trace file will be produced. This is useful when the order that logic is executed is important in identifying a problem.

Leaving using U_TRACEM in your code

When U_TRACEM is activate it maysignificantly slow down your program(s).

However, the overhead of leaving U_TRACEM (deactivated) in your code even when it's promoted to a production environment is quite small. At execution time the only overheads are slightly larger executable objects and the cost of evaluating the *TraceON (ie: #TraceOn = Y) conditions around each block of trace logic.

The counterpoint to the small overhead is that being able to conditionally turn tracing on in a production environment may be a valuable problem diagnostic tool.

One simple way to leave tracing logic in your code is to use a "flag file" to indicate whether tracing should be turned on or off. This small subroutine demonstrates the use of flag file …..

```
Subroutine Name(TraceOn)
   Define Field(#RetCode) Type(*char) Length(2)
   If_Ref Com(#Tracer) Is(*null)
      Use Builtin(OV_FILE_SERVICE) With_Args(CHECK_FILE (*Part_Dir_Execute +
      U_TRACE_ + #Com_Owner.Name + '.dat')) To_Get(#RetCode)
      If (#RetCode = OK)
         Set_Ref Com(#Tracer) To(*Create_as #u_TraceM)
         #TraceOn := Y
       Endif
   Endif
Endroutine
```

This logic looks for a "flag file" named U_TRACE_XXXXXXXXXX.DAT (where XXXXXXXXXX is the name of the program or component) in the current partition's execute directory. If a flag file is found it turns trace mode on. This approach allows you to dynamically turn tracing on or off simply be creating or destroying a flag file named U_TRACE_XXXXXXXXXX.DAT in the partition's execute directory.

The cost of doing the check for the existence of the flag file is relatively small so it would probably be viable to do this in most routines and most situations. However, if you were to call a function 100 times in a loop then the file check overhead may significantly impact the program. In this situation you would be better to pass the required trace state into the function via the exchange list or as a parameter.

Appendices

U_TRACEO – The Application Tracing Object

Create a *reusable component* named U_TRACEO. Enable it for full RDMLX. Delete it's source code and then copy and paste in the following RDMLX code. Compile on our PC and also Check In and compile on your iSeries server.

```
FUNCTION OPTIONS(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_OBJT)

Define #FileNo *Dec 3 0 default(-1) Desc("Trace file number. -1 is not open.")
Def_Cond *Open (#FileNo >= 0)
Def_Cond *NotOpen (#FileNo = -1)

Define #RetCode *char 2 desc("Builtin function return codes")
Def_Cond *Ret_OK (#RetCode = OK)
Def_Cond *Ret_NotOK (#RetCode *ne OK)

DEFINE_COM CLASS(#Std_Obj) NAME(#TraceFilePrefix)
Define_Pty TraceFilePrefix Set(*Auto #TraceFilePrefix)

DEFINE_COM CLASS(#Prim_acol<#Prim_alph>) NAME(#ScanStrings)
DEFINE_COM CLASS(#Prim_acol<#Prim_alph>) NAME(#ReplaceStrings)

* ============================================================
* Log a trace event to the trace file
* ============================================================

Mthroutine Log
Define_map *input #Prim_alph #Event
Define_map *input #Prim_alph #Value1 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value2 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value3 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value4 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value5 Mandatory("<<DFT>>")
Define_map *input #Prim_Objt #Issuer Pass(*By_Reference)
Define_Com #Prim_alph #FileName

Define_com #Prim_acol<#Prim_alph> #Values

* Set indexable references to all 5 input values

Set_ref #values<1> #Value1
Set_ref #values<2> #Value2
Set_ref #values<3> #Value3
Set_ref #values<4> #Value4
Set_ref #values<5> #Value5

* If the trace file is not open then open and start the HTML content

If *NotOpen
#FileName := ( *Part_Dir_execute + #TraceFilePrefix + "UTRACE_" + *YYYYMMDDC + "_" + *TIMEC +
"_" + *JobName + "_" +  *JobNbr + "_" + *User + ".htm")
USE BUILTIN(STM_FILE_OPEN) WITH_ARGS(#FileName "Write Text CodePage=00819") TO_GET(#FileNo
#RetCode)
If *Ret_NotOK
Message ("Trace file " + #FileName + " could not be opened for output")
Return
Endif
#Com_Owner.Write Line("<html><head></head><body>")
#Com_Owner.Write Line("<div>" + #Com_Owner.MakeHTMLString(("Trace File Name=" + #FileName))
```

```
+ "</div><div> </div>")
#Com_Owner.Write Line("<div>" + "Trace Job Name=" +  *jobname + " Job Number=" + *jobnbr + "
User=" + *User + " Date=" + #Com_Owner.FormatDateorTime(*YYMMDDC "-") + " Time=" +
#Com_Owner.FormatDateorTime(*Timec ":") + "</div><div> </div>")
#Com_Owner.Write Line("<table
border='1'><tr><td>Time</td><td>Issuer</td><td>Event</td><td>Value 1</td><td>Value
2</td><td>Value 3</td><td>Value 4</td><td>Value 5</td></tr>")
Endif

* Write the trace data out

#Com_Owner.Write Line("<tr><td>" + #Com_Owner.FormatDateorTime(*TimeC ":") + "</td><td>" +
#Com_Owner.MakeHTMLString(#Issuer.Name) + "</td><td>" +
#Com_Owner.MakeHTMLString(#Event.Trim) + "</td>")

For #value in(#values)
If (#Value = "<<DFT>>")
#Com_Owner.Write Line("<td> </td>")
else
#Com_Owner.Write Line("<td>" + #Com_Owner.MakeHTMLString(#Value.Trim) + "</td>")
Endif
Endfor

#Com_Owner.Write Line("</tr>")

* Clean up

Invoke #Values.RemoveAll

* Finished

Endroutine

* ================================================================
* Make a string into valid HTML data
* ================================================================

MthRoutine FormatDateorTime
Define_map *input #Prim_Alph #In
Define_map *input #Prim_Alph #Sep
Define_map *result #Prim_Alph #Out
#Out := #In.Substring(1 2) + #Sep + #In.Substring(3 2) + #Sep + #In.Substring(5 2)
Endroutine

* ================================================================
* Make a string into valid HTML data
* ================================================================

MthRoutine MakeHTMLString
Define_Map *input #Prim_Alph #In
Define_Map *Result #Prim_Alph #Out
Define_Com #std_num #FoundAt

#Out.Value := #In.Value

Set #FoundAt Value(0)
For Each(#Scan) In(#ScanStrings) Key(#ScanIndex)
DoUntil (#FoundAt <= 0)
#FoundAt := #Out.Value.PositionOf(#Scan.Value (#FoundAt + 1))
If (#FoundAt > 0)
#Out.Value := #Out.Value.ReplaceSubString(#FoundAt.Value #Scan.Value.CurSize
#ReplaceStrings<#ScanIndex>.Value)
Endif
EndUntil
EndFor
```

```
Endroutine

* ================================================================
* Write a line to the trace file
* ================================================================

MthRoutine Write
Define_Map *input #Prim_Alph #Line

If *Open
USE BUILTIN(STM_FILE_WRITE) WITH_ARGS(#FileNo #Line.Trim) TO_GET(#RetCode)
USE BUILTIN(STM_FILE_WRITE_CTL) WITH_ARGS(#FileNo CRLF) TO_GET(#RetCode)
Endif

Endroutine

* ================================================================
* Handle destruction of this object
* ================================================================

EVTROUTINE HANDLING(#Com_Owner.DestroyInstance) OPTIONS(*NOCLEARERRORS
*NOCLEARMESSAGES)

* If trace file open, add termination HTML format and then close

If *Open
#Com_Owner.Write Line("</table></body></html>")
USE BUILTIN(STM_FILE_CLOSE) WITH_ARGS(#FileNo) TO_GET(#RetCode)
Endif

Endroutine

* ================================================================
* Handle creation of this object
* ================================================================

EVTROUTINE HANDLING(#Com_Owner.CreateInstance) OPTIONS(*NOCLEARERRORS
*NOCLEARMESSAGES)

* Set up default scan/replace strings

Begin_Loop from(1) to(3) Using(#Std_Num)
Set_ref #ScanStrings<#Std_Num> (*Create_as #Prim_Alph)
Set_ref #ReplaceStrings<#Std_Num> (*Create_as #Prim_Alph)
End_Loop

Set #ScanStrings<1> Value("&")
Set #ReplaceStrings<1> Value("&amp;")

Set #ScanStrings<2> Value("<")
Set #ReplaceStrings<2> Value("&lt;")

Set #ScanStrings<3> Value(">")
Set #ReplaceStrings<3> Value("&gt;")

Endroutine


END_COM
```

U_TRACEM – The Application Tracing Manager

Create a *reusable component* named U_TRACEM. Enable it for full RDMLX. Delete it's source code and then copy and paste in the following RDMLX code. Compile on our PC and also Check In and compile on your iSeries server.

```
FUNCTION OPTIONS(*DIRECT)
BEGIN_COM ROLE(*EXTENDS #PRIM_OBJT)

* Declare a single shared of the tracing object

Define_com #U_TraceO Scope(*Shared)

* Special prefix that can be added to trace file names to make it easier to identify

Define_Pty TraceFilePrefix Set(Set_FilePrefix)

* Handle prefix property setting

PtyRoutine Set_FilePrefix
Define_Map *input #std_Obj #pty_StdObj
#U_TraceO.TraceFilePrefix := #Pty_StdObj.Trim + "_"
Endroutine

* Delegate the log tracing request into the single shared instance

Mthroutine Log
Define_map *input #Prim_alph #Event
Define_map *input #Prim_alph #Value1 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value2 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value3 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value4 Mandatory("<<DFT>>")
Define_map *input #Prim_alph #Value5 Mandatory("<<DFT>>")
Define_map *input #Prim_Objt #Issuer Pass(*By_Reference)

Invoke #U_TraceO.Log Event(#Event) Issuer(#Issuer) Value1(#Value1) Value2(#Value2)
Value3(#Value3)  Value4(#Value4)  Value5(#Value5)

Endroutine

END_COM
```