CICS TS MQ Interface for All

Thomas Dunlap  Themis, Inc.  tomd@themisinc.com

Please use the ‘Question Tab’ for questions instead of the Chat window.

Link to get the PDF:  www.themisinc.com/webinars
A LOOK AT THE APPLICATION PROGRAM
Each application program which will execute using MQ call statements require a few basic inclusions of generated code. The COPY shown are the one required by all applications using MQ functions. We have suppressed the list of the CMQV structure since it is over 20 pages of COBOL variable constants.

The WS-VARIABLES are the minimum necessary to support bot getting and putting messages from and to queues. The MQ-CONN would normally be the handle acquired by an MQCONN function, but that call along with the MQDISC call are ignored by CICS / MQ interface. It is CICS itself that has performed the MQCONN to the queue manager.

The MQ-HOBJ-I and MQ-HOBJ-O represent the handles, acquired on MQOPEN for a queue, used to specify which queue the other calls relate to. The MQ-COMPCODE and MQ-REASON are the variables which the application must test after every MQ call to determine its success.
This slide contains the message input and output areas used on the MQGET and MQPUT calls. Our messages are rather simple and limited in size. So we have put the data areas in the WORKING-STORAGE SECTION. Many times you will have these as copy books rather than native COBOL variables.

Since the WORKING-STORAGE area is being copied for each task using the program, we recommend that larger messages areas be put into the LINKAGE SECTION. Here we use a size of 200-300 K as a suggestion. However, that size will vary depending upon your CICS environment. When the application is using message areas in the LINKAGE SECTION, it will be responsible to perform a CICS GETMAIN command to acquire the storage for the message area.
Now to the actual executable MQ call statements on this slide. Here we see the COBOL statements necessary to populate the variables for an MQPUT1 call. The MQPUT1 call is a combination of an MQOPEN, MQPUT, and MQCLOSE in a single call. Many times your CICS application will work with a single message within a task, so this call saves you a bit of code, plus interacting with the MQ interface two times. This does mean less overhead to the executing task. However, this savings disappears when you process more than one message with an MQPUT1 call.

Since we are putting a message to a queue, the first three COBOL MOVEs populate the message area for our simple message. It is a “request message”, passed to another CICS task to process and return a result.

The MOVEs to the MQOD-variables setup the queue name for the MQOPEN portion of the call.

The MOVEs to the MQMD-variables setup MQ options which describe the message and functions desired. The MQMT-REQUEST identifies this as a “request message”, plus indicates the use of the MQ “ReplyToQ” function. The MOVEs for MQMD-REPLYTOQ and MQMD-REPLYTOQMG provide the values which will be used by the other MQ application to respond with the “reply message”. The MQFMT-STRING informs MQ that the data portion of the message is strictly character data and may participate in data conversion. The setting of MQMI-NONE is important to ensure that each message obtains a unique message identifier. The setting of MQCI-NONE means that the receiving application will not be retrieving specific messages.

The COMPUTE statement sets the MQPMO-NO-SYNCPPOINT option, which means that the message is not put within Unit-of-Work (UOW) control. If the CICS task were to abend, the message would not be rolled back.

Next is the actual MQPUT1 call itself. It is a “static call” which means it will resolve to an actual module at Linkage Edit time. You cannot use the COBOL DYNAM option with MQ applications since the name on the call is not the actual module name. We will show how to code MQ dynamic calls at the end of this section. The EVALUATE statement show a simple example of testing how the call completed.
On this slide we show an example of the MQOPEN call to access a queue. The COMPUTE statement sets options to specify the queue is input to the application, browsing of messages will be allowed, and the call is to fail if the queue manager is being shutdown.

The two MOVE statements provide the queue name and clear the queue manager name. This is followed by the actual MQOPEN call. The MQ-HOBJ-I is the returned handle which will be used on subsequent MQGET calls to specify which queue is being utilized.

The EVALUATE statement will check how the call completed.
Here we see an example of an MQGET call. The first to MOVEs are necessary to ensure retrieval of the “next message” from a queue. If the application were to exclude setting these variables before an MQGET, then ME would take the value in them and look for a specific message. We will cover this condition at the end of this section.

The MOOVE of 20000 to the variable MQGMO-WAITINTERVAL set the time to wait for a message to arrive in the queue of 20 seconds. The COMPUTE statement sets the MQGMO-WAIT option to activate the “get with wait” function, plus to get the message outside of UOW control. That means the message will not be rolled back if the task abends. It also means that by default, the message when successfully retrieved will be removed from the queue.

We then see the actual MQGET call.

Finally, we see the EVALUATE statement to check completion condition of the call. Please note the check for MQMI-NONE, which is our method to check for existence of at least one message.
On this slide we see an example of an MQCLOSE call. The setting of the MQ-OPTIONS is just to clear the variable to indicate a normal MQCLOSE call. This is followed by the actual MQCLOSE call itself.

As with previous calls, the EVALUATE statement is used to check the status of the call.
This slide contains the variables with the values for each of the actual module names that can be used to code dynamic calls in the application. You will notice that the module names are different than the call module names we have seen earlier. This is why you cannot use the COBOL DYNAM option when compiling a CICS MQ application.

We use the MQPUT1 call to show an example of the minor change necessary to any MQ call to make it a dynamic call.
The MQMD_MSGID will always be a unique, generated value as long as the applications do their part. The application is required to always set the attribute to MQMI_NONE or include the option MQPMO_NEW_MSG_ID for every MQPUT or MQPUT1 call. If they do not provide this code then the MsgId generated on the first call is reused for all subsequent calls. This is due to the fact that the generated MsgId is passed back to the application by MQ in the MQMD-MSGID attribute. This is intended to provide the application with a unique “tracking marker” for each message.

When retrieving messages the MQMD_MSGID attribute should be set to MQMI_NONE before the MQGET call, or the MQGMO_MATCHOPTIONS attribute set with MQMO_NONE. If the application does not provide this code then only the first message is retrieved and the second MQGET call issued will completely pass through the rest of the messages in the queue and return an error. This is due to the fact that with the first MQGET call the MsgId is populated with a valid value so when the second MQGET is issued the queue manager acts like you are searching for a particular message which will not be found.
A LOOK AT TRACE FOR THE APPLICATION PROGRAM
This slide contains the portion of trace entries representing the MQOPEN call to access the specified queue. The bracketed entries on the left represent the total time spent in the CICS MQ interface. Most of this time (processor time) will actually be captured and reported on the MQ region side. This is because control is transferred from the MQ interface to MQ as a secondary address space very quickly. We have always classified the time spent between “ERM ENTRY” and “ERM EXIT” as time when the application has turned control over to the queue manager and entered a wait state.

The bracketed entries on the right represent the total time spent in the queue manager region itself. The “MQTRU ENTRY” trace entry occurs just before giving control to MQ. The “MQTRU EXIT” occurs just after MQ gives control back to the CICS MQ interface. This time is considered to be solely done on behalf of the queue manager region and the CICS application is in a wait state for its completion.

While the times shown on the slide might seem to be relatively low, on our small little mainframe we have executed around 1 million instructions to complete this MQOPEN call.
This slide contains the portion of trace entries representing the MQGET call to retrieve the reply message from the queue. The bracketed entries on the left represent the total time spent in the CICS MQ interface. Most of this time (processor time) will actually be captured and reported on the queue manager region side. This is because control is transferred from the MQ interface to the queue manager as a secondary address space very quickly. We have always classified the time spent between “ERM ENTRY” and “ERM EXIT” as time when the application has turned control over to the queue manager and entered a wait state.

The bracketed entries on the right represent the total time spent in the queue manager itself. The “MQTRU ENTRY” trace entry occurs just before giving control to MQ. The “MQTRU EXIT” occurs just after DB2 gives control back to the CICS MQ interface. This time is considered to be solely done on behalf of the queue manager and the CICS application is in a wait state for its completion.

While the times shown on the slide might seem to be relatively low, on our small little mainframe we have executed around 250,000 instructions to complete this MQGET call.
This slide contains the portion of trace entries representing the MQPUT1 call to put the request message to a queue. The bracketed entries on the left represent the total time spent in the CICS MQ interface. Most of this time (processor time) will actually be captured and reported on the queue manager side. This is because control is transferred from the MQ interface to the queue manager as a secondary address space very quickly. We have always classified the time spent between “ERM ENTRY” and “ERM EXIT” as time when the application has turned control over to the queue manager and entered a wait state.

The bracketed entries on the right represent the total time spent in the queue manager itself. The “MQTRU ENTRY” trace entry occurs just before giving control to the queue manager. The “MQTRU EXIT” occurs just after the QM gives control back to the CICS MQ interface. This time is considered to be solely done on behalf of the QM and the CICS application is in a wait state for its completion.

While the times shown on the slide might seem to be relatively low, on our small little mainframe we have executed around 500,000 instructions to complete MQPUT1 call.
This slide contains the portion of trace entries representing the MQGET call to retrieve a message from the queue. The bracketed entries on the left represent the total time spent in the CICS MQ interface. Most of this time (processor time) will actually be captured and reported on the queue manager side. This is because control is transferred from the MQ interface to queue manager as a secondary address space very quickly. We have always classified the time spent between “ERM ENTRY” and “ERM EXIT” as time when the application has turned control over to the queue manager and entered a wait state.

The bracketed entries on the right represent the total time spent in the queue manager itself. The “MQTRU ENTRY” trace entry occurs just before giving control to the queue manager. The “MQTRU EXIT” occurs just after the QM gives control back to the CICS MQ interface. This time is considered to be solely done on behalf of the QM and the CICS application is in a wait state for its completion.

While the times shown on the slide might seem to be relatively low, on our small little mainframe we have executed around 250,000 instructions to complete this MQGET call.
This slide contains the portion of trace entries representing the SQL SELECT statement to find the average and maximum SALARY. The bracketed entries on the left represent the total time spent in the CICS DB2 interface. Most of this time (processor time) will actually be captured and reported on the DB2 region side. This is because control is transferred from the DB2 interface to DB2 as a secondary address space very quickly. We have always classified the time spent between “ERM ENTRY” and “ERM EXIT” as time when the application has turned control over to DB2 and entered a wait state.

The bracketed entries on the right represent the total time spent in the DB2 region itself. The “MQTRU ENTRY” trace entry occurs just before giving control to the queue manager. The “MQTRU EXIT” occurs just after DB2 give control back to the CICS MQ interface. This time is considered to be solely done on behalf of the queue manager and the CICS application is in a wait state for its completion.

While the times shown on the slide might seem to be relatively low, on our small little mainframe we have executed around 500,000 instructions to complete this MQCLOSE call.
CICS MQ interface as part of the EXEC CICS RETURN command being executed. CICS is the coordinator for committing modified data during execution of the application within the CICS region. There is a lot of activity going on during this process of ending and cleanup of the application. We have carved out the activity associated with the CICS MQ interface only.

The trace entries represented by connector A, represent the total time within the CICS MQ interface itself. The entries represented by the connector B, show the time the interface communicated with Resource Recovery Services (RRS) address space on z/OS. The entries represented by connector C, is the actual time spent in the queue manager itself for “prepare to commit”. The entries represented by connector D, is the actual time spent in the queue manager to perform its portion of the commit.
CEDF SCREENS FOR MQ INTERFACE
On this slide we can see the application map which allows for entry of the information that will produce the request message. This is a simple inventory update application where two CICS transaction, two queues and one VSAM file are involved. It is as close to a real production application we can get during a class workshop.
On this slide we see the MQPUT1 call about to execute. As you can see, there is no clear indication that it is an MQPUT1. We can determine it is the MQPUT1 because ARG 001 is the MQOD, ARG 002 is the MQMD, and ARG003 is the MQPMO. Please remember, ARG000 through ARG 007 represent the “call parameters” in the order which they are specified on the MQPUT1 call. Also, IBM does not show the complete data area represented by each argument, only the first 16 bytes.
On this slide we can see the completion of the MQPUT1 call. Again, this determined by the MQOD, MQMD, and MQPMO being supplied. Also, IBM shows the first 16 bytes of each data area. However, ARG 006 is the completion code and ARG 007 the reason code. Both of these are binary full words. So to determine their value you have to hit the F2 key to switch to Hex. You will then be able to see the MQ returned completion and reason codes.
On this slide we can see the MQOPEN call about to be executed. The only indication of this is that ARG001 is the MQOD. The ARG 000 through ARG 005 represent the MQOPEN call parameters in the sequence which they were coded.
On this slide we can see the completion of the MQOPEN call. Once again, ARG 004 is the completion code and ARG 005 the reason code. To view them you have to hit F2 to switch to Hex. Both are full word binaries, so only look at the first 4 bytes of ARG 004 and ARG 005.
On this slide we can see the MQGET call about to execute. We determined this by the fact that ARG 002 is the MQMD and ARG 003 is the MQGMO.
On this slide we can see the completion of the MQGET call. The completion code is ARG 007 and the reason code ARG 008. To see their values you have to switch to Hex mode.
On this slide we can see the MQCLOSE call about to execute. We can only determine this by the lack of any MQ structure being present in the arguments.
On this slide we can see the completion of the MQCLOSE call. The first full words of ARG003 and ARG 004 are the completion code and reason code, respectively.
Finally we see the application map being populated with the returned values from the replay message.

Our little application took the values entered on the top of the screen and constructed a “Request message”. It then issued an MQPUT1 call to put the message to the “transaction queue”, which cause MQ to trigger the QT00 transaction to execute as a background task. The QT00 application then retrieved the messages from the queue and updated the inventory VSAM file. It then issued an MQPUT1 to put the “Reply message” to the “results queue”. This caused the RT00 transaction (the one working with the map) to wake-up from its MQGET with wait, populate the screen and send the map back to the user.
STATISTICS FOR MQ INTERFACE
This slide contains the first portion of statistics for the CICS MQ Interface. The top portion (through “futile attempts”) contains basic information about the interface including the name of the queue manager, its MQ release, name of the initiation queue, plus the connection status and when the connection was established. We also see the current number of tasks using MQ and the number of calls issued while the connection was not active (futile attempts).

The next portion contains the total number of API calls issued, the number which have completed successfully, plus the name of the API crossing exit.

The next portion contains the number of API calls by each type, from OPEN (MQOPEN) to SET (MQSET) requests.

The lower portion provides the number of internal MQ calls issued, number completed successfully, number of I/Os required, and the number of times a TCB switch was necessary.
This slide shows the rest of the MQ interface statistics. The first 4 lines are statistics related to recovery processing as required during MQ interface start up. Unless the CICS region was not terminated abnormally, there should be no values in these 4 lines.

The next 2 lines show the number of tasks which were Backed out or Committed. This is assuming that not task issued a EXEC CICS SYNCPOINT or SYNCPOINT ROLLBACK command. We then see the Number of tasks perform along with the number of single or 2 phase commits performed.

The last 13 line provide statistics about the number of each of the unusual MQ calls issued. These calls represent new features added in MQ Version 7 and are not widely used.
MQ RESOURCE DEFINITIONS
This slide provides a simple view of how CICS will interface to resource managers like IBM MQ or DB2. The key to working with other resource managers is the Task Related User Exit or TRUE programs. These are provided by CICS to provide the cross region communication with the resource manager in the other address space.

The CICS application will execute the command to access one of the defined resource managers. These commands are the normal API functions expected by that resource manager, which include:

- CALL 'MQxxxx' – for WebSphere MQ requests
- EXEC SQL – for DB2 requests
- EXEC DLI – for IMS requests

CICS maintains a pool of Task Control Blocks or TCBs to process these requests on, thus freeing up the normal CICS TCBs to perform other work. There is an SI parameter, MAXOPENTCBS, which specified the number of TCBs CICS could use. All of the work performed on these TCBs is “threadsafe” to minimize TCB switching and improve performance.

The CKAM transaction is the “alert monitor” transaction for the MQ adapter. The transaction handles unscheduled events which are produced by the queue manager. This includes such events like the queue manager being shutdown or waiting for the queue manager to start up.

The CKTI transaction is the CICS trigger monitor. The trigger monitors in MQ perform MQGET with wait on a define initiation queue (INITQ) associated with the application. In our case for the CICS region that CKTI is running in. When a trigger event occurs, the queue manager puts a message into the INITQ. This will wake up CKTI, which then analyzes the message contents (it is known as a trigger message) and will start the application it indicates.
This slide shows the only resource definition associated with the CICS MQ interface. It serves to define the connection to the desired queue manager. CICS can only be connected to one queue manager at a time. To connect to a different queue manager you would have to stop the interface, change this resource, and start the interface once again.

The option `MQNAME(MQAA)` specifies the name of the queue manager with which CICS will establish a connection, maximum of 4 characters. The queue manager must be on the same LPAR where CICS is running.

The option `INITQNAME(TCICSTA3.INITQ)` specifies the name of the initiation queue the CICS trigger monitor will use. This queue must be defined in the queue manager and specified in the INITQ option for application queues. CICS will automatically start the CKTI transaction, passing it this name to use in its processing.

Unlike the CICS DB2 Interface, there is no option to limit the number of “Open TCBS” the CICS MQ Interface will use. We know that it will may approach the value you specify in MAXOPENTCBS. The limiting factor is the number of currently in use TCBs by the CICS DB2 interface.
Here we see the QLocal definition for the “initiation queue” in support of MQ triggering in a CICS region. At minimum there must one of these for each CICS region you define the MQCONN resource. However, it is possible to have more than one CKTI (the trigger monitor) task running within the same region. This means you may need additional initiation queues to support them.

We have added the SHARE and DEFSOPT(SHARED) options to ensure that multiple concurrent CKTI task could access the same initiation queue if necessary.

The MAXDEPTH and MAXMSGL options have been set to more than is necessary for an initiation queue. Since the trigger monitor is processing trigger messages as fast as they arrive, the depth should never approach anywhere near the 1000 message limit. The trigger message is currently less than 1000 bytes, so 4000 byte size limit is 4 times what is necessary.

You should always specify MSGDLVSQ (message delivery sequence) as FIFO (first in first out). This takes functions like priority message processing out of the way and helps ensure that the trigger monitor can process the trigger messages as soon as they arrive.
This slide contains the queue and process definitions we used in support of our little inventory application. The **THMZ00.TRANSACTION** queue is the one where the "Request message" was put with the input from the initial screen to pass to the QT00 update transaction. This is why you see the "trigger" attributes populated plus the **PROCESS** and **INITQ** options.

The **THMZ00.RESULTS** queue is the one which the initiating transaction, RT00, will issue the MQGET with wait against, plus the queue where the "Reply message" will be put by the QT00 transaction.

The **THMZ00.EXEC.CICS** is the process definition which represents the QT00 transaction to be triggered.
SVC DUMP INFORMATION FOR MQ INTERFACE
This slide contains the “CICS MQ Interface Summary” from an SVC dump. The upper portion simply contains information about the MQ connection. This includes the name of the queue manager name involved and its version. Another key piece of information is the name of the specified initiation queue. You can also see the current status of the connection.

The “TRANSACTION SUMMARY” show all of the tasks in the CICS region which have or are currently accessing MQ resources. Much of the information is of minimal interest, such as TcaAddr, TieAddr, and Uowid. The LotAddr is the control blocks used for the task during execution. The Tcb in MQ indicates if the task currently has a MQ request outstanding.

We have ignored the “TRIGGER MONITOR TRANSACTION SUMMARY” since the CKTI transactions will also show the same information in the “All TRANSACTION SUMMARY”.

The key portion is the “TRANSACTION SUMMARY”
The first data area is the DFHMQINI or MQ initialization control block.

+0 Half-word length of the data area
+2 14-byte identifier of the data area
+18 Name of the initiation queue
+48 Task number of the CKAM monitor task
+60 Name of the MQCONN resource definition

The second data area is the “LIFE OF ADAPTER” data area. There is not much information of use to use in the data area.

The third data area is the “LIFE OF CONNECTION”.

+14 Sub-system name of the queue manager
+18 CICS Applid
+50 Sub-system of the connected QM

Most of the rest of the data is the statistics which are written to SMF.
Here we find the Life Of Task (LOT) control block. The data area is used while the task is executing to manage the application making requests to the queue manager. The data area is no longer documented. However, the last Supplemental Data Areas manual was for CICS TS V4.1 and the control block seems to have not changed with subsequent releases.

The LOT basic format is as follows:

- **+0**  Half-word length of data area
- **+2**  14 character eye-catcher
- **+20** CICS transaction ID
- **+24** CICS task number
- **+28** User ID
- **+48** TCA address
- **+108** Name of the initiation queue if the LOT represents a CKTI transaction
- **+138** Name of the queue accessed by last MQ call if LOT is for a user transaction
CKQC FUNCTIONS
This is the initial screen for the CKQC transaction. The three items at the top of the screen; Connection, CKTI and Task; are drop down menus with more selection choices.
This drop down is shown by putting the cursor on the “Connection topic” and hitting the enter key. You will notice the four choices, and we have entered a “4” to go to the “Display” function.
This slide shows the output of the Display function. It is set of general information about the CICS MQ interface. On the top portion we see the CICS region name, connection status, queue manager name, MQCONN resource name, and the name of the initiation queue.

In the center portion we see the current number of tasks in-flight, total MQ calls issued, and the current number of running CKTI transactions.

The lower portion contains a break down of the MQ calls issued to this point in time. The MQOPEN, MQCLOSE, MQGET, GETWAIT, MQPUT, MQPUT1, MQING, and MQSET and simply the number of each call executed. The GETWAIT is the number of MQGET calls issued with the MQGMO-WAIT option.

The Run OK value represent the number of calls which have completed successfully. The value in Futile represents the number of MQ calls issued while the connection was not active.

Under the “Syncpoint” category, Tasks is the total number of tasks using MQ to this point in time. The Backout and Commit are counts for the tasks which have been rolled back and committed.

Under the “Recovery” category, Indoubt is the number of UOWs which were indoubt at MQ interface start up. The UnResol is the number which were not resolved. The Commit is the number resolved by a Commit and Backout is the number resolved by Backout (rollback).

Under the “Flows” category, Calls is the number of flows to the queue manager on the connection, and SyncComp is the number which have completed synchronously. The values SuspReqd and Msg wait indicate the number flows which had to wait for completion of the MQ call. The value of Switched represents the number of calls where a TCB switch occurred.
From the Connection drop down menu, we now choose option 3 to Modify the connection. Here you see the pop-up menu with the modify options. We have a limited set of choices on what to modify. We can simply reset the statistic discussed earlier, and either enable or disable the API crossing exit.
We then choose option 2 on the Connection drop down menu, Stop the MQ connection. Here you see the pop-up for the Stop function. We either choose to “Quiesce” or “Force” stop the connection. Quiesce will allow all currently active tasks to complete and disallow any new MQ tasks from starting. Force will cause the abnormal termination of any current tasks and disallow any new MQ tasks from starting.
Finally we choose option 1 from the Connection drop down menu, Start the connection. Here you see the pop-up menu for the Start function. We are required to enter two values, 1 is the name of the queue manager to connect with, and 2 is the name of the initiation queue the CKTI transaction is to use. The queue manager name may be different than the one specified by the MQCONN resource definition.
On this slide we have placed the cursor on the CKTI option on the top menu line and hit enter. You are presented with the CKTI drop down menu. You will notice the three possible choices. We have picked option 3 to display the current CKTI information.
Here is the Display screen for the current CKTI transaction(s). We see only one CKTI running in the CICS region at the moment. However, we can start up more CKTI transactions as required. There is not much information on this screen. Simply the CICS task number, status, number of API calls issued, and the name of the initiation queue.

<table>
<thead>
<tr>
<th>Task Num</th>
<th>Task Status</th>
<th>Thread Status</th>
<th>Num of APIs</th>
<th>Last API</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000051</td>
<td>Normal</td>
<td>Msg Wait</td>
<td>9</td>
<td>MQGET</td>
</tr>
</tbody>
</table>

Initiation Queue Name: TCICSA3.INITQ

DF1M00461 I TCICSA3 Top of display.
F1=Help  F7=Backward  F8=Forward  F12=Cancel  Enter=Refresh

Display screen from “Option 3” on CKTI drop down.
Now we choose option 2 from the CKTI drop down menu, Stop the CKTI task. Here you see the pop-up menu where you specify the name of the initiation queue for the CKTI task you want to stop. The assumption is that every CKTI task will be using a different initiation queue.
Finally we chose option 1 on the CKTI drop down menu, Start a CKTI transaction. Here you see the pop-up menu for the Start function. You need only specify the name of the initiation queue that the CKTI transaction will use. You can start more than one CKTI transaction by simply specifying the same or a different initiation queue.
This screen is obtained by putting the cursor on the Task topic at the top of the initial screen and hitting enter. This display shows a list of all of the current tasks which has performed MQ calls. You will see any of your tasks, plus the CKTI tasks currently running in the CICS region. On this display you find the basic information about the task, plus its current status. The Res Sec column indicates if “resource security” is active for the transaction. The API Exit column indicates if the API crossing exit is in effect for the transaction. You will also see the “Last MQ call” issued and the internal Thread ID for the task.