

FINGER ON THE TRIGGER

STEVE CLOSE SHOWS YOU HOW TO USE FILE TRIGGERS TO IMPROVE YOUR BUSINESS PROCESSES

➤ One theme that seems to be common in many iSeries installations today is the requirement to add extra functionality to their ERP system without performing any modifications to the underlying code. This can be because of the desire to keep with their supplier's base software package to make upgrades easier and avoid compromising their maintenance contracts, or simply a fear of diving in and changing applications which are working perfectly.

Fortunately, OS/400 (i5/OS) provides a relatively easy way to add in extra business logic by monitoring database file changes. In this article I am going to demonstrate how to add a new business function using triggers that will send a message to a message queue whenever a customer goes over his credit limit. This is purely an example of the kind of functionality that can be added – it is of course possible to achieve a myriad of improvements – the only limit really being your imagination.

When a trigger is added to a file it is necessary to specify what type of event you wish the trigger to process and also which program you wish to call to handle the trigger. The four event types are:

1. *INSERT – The program will be called whenever a record is added.
2. *DELETE – The program will be called whenever a record is deleted.
3. *UPDATE – The program is called whenever a record is updated.
4. *READ – The program is called whenever a record is read from the file.

There are some very important things to bear in mind when using triggers. Firstly, the call to the trigger program becomes part of the actual I/O processing of the file. This means that your trigger program needs to execute very quickly. If a long-running task is to be performed then it is probably better that your trigger program submits a job to perform the processing.

Secondly, when adding a trigger to a file you will need to gain an exclusive lock on the file. For most of your production files this will probably be impossible during the day. To my mind you have two choices here – you can do this out of hours and manually enter the commands or you can write a CL program to do this and schedule it to run when the files will not be in use. To minimise the disruption to your social life it is probably worthwhile deciding on all the files you may wish to trigger, both now and in the future, and do this exercise all in one go.

Thirdly, make sure that the trigger routine you call is robust. If any user intervention is required, such as answering a message on QSYSOPR message queue because of a program error, the calling job will stop until the message is answered.

Fourthly, once the trigger is active the operating system will put a lock on the program called by the trigger – this means that you will not

be able to recompile it without ending the triggers. This is why I feel the best approach is to write a generic trigger program (such as the one provided below) that handles all triggers – once this program works correctly it will never need changing. This generic program will then call other programs to perform the necessary actions. Because the second-level programs are not the actual trigger programs against the file, they will not be locked by the operating system and you will be able to recompile these at your leisure.

Finally, bear in mind that because the files you are using for triggers can be updated by any job running anywhere on your system, it is important to ensure that any objects needed by the trigger routines are accessible to all these jobs – the easiest way to achieve this is to put all the programs into a library that is in your system library list.

Here is the RPG ILE code for the generic trigger program that you will specify in the ADDPFTRG command later. The purpose of this program is to extract the data provided by the trigger routine and perform whatever processes we wish. For the purposes of this article, I will call this program TRIGGER1.

```
D file          s          10
D library      s          10
D member      s          10
D beforeimg   s          32767
D afterimg    s          32767
D triggerevt  s           1
D triggertme  s           1
D
D              ds
D ccsidx      1          4
D ccsid       1         4b 0
D rrnx       5          8
D rrn        5         8b 0
D orgoffsetx  9         12
D orgoffset   9        12b 0
D orglensex  13         16
D orglen     13        16b 0
D newoffsetx 17         20
D newoffset  17        20b 0
D newlenx   21         24
D newlen    21        24b 0
D
D              sds
D username   254        263
* Parameters passed by OS400 are the buffer containing the before
* and after image, and the length of the data in the buffer
C *entry      plist
```

```

C          parm          inbuffer      65535
C          parm          buflenx       9
0
* Extract file, library and member names from the input string
C          eval          file=%subst(inbuffer:1:10)
C          eval          library=%subst(inbuffer:11:10)
C          eval          member=%subst(inbuffer:21:10)
* Extract trigger event and time (Before or after file update)
C          eval          triggerevt=%subst(inbuffer:31:1)
C          eval          triggertme=%subst(inbuffer:32:1)
* Extract CCSID of the record updated
C          eval          ccscid=%subst(inbuffer:37:4)
* Extract relative record number of the record updated
C          eval          rrnx=%subst(inbuffer:41:4)
* Extract original record offset and length
C          eval          orgoffsetx=%subst(inbuffer:49:4)
C          eval          orglenx=%subst(inbuffer:53:4)
* Extract new record offset and length
C          eval          newoffsetx=%subst(inbuffer:65:4)
C          eval          newlenx=%subst(inbuffer:69:4)
* Extract old record image
C          eval          beforeimg=%subst(inbuffer:
C          orgoffset+1:orglen)
* Extract new record image
C          eval          afterimg=%subst(inbuffer:
C          newoffset+1:newlen)
* Call trigger handling program
C          call          'TRIGGER2'
C          parm          file
C          parm          library
C          parm          member
C          parm          username
C          parm          triggerevt
C          parm          triggertme
C          parm          ccscid
C          parm          rrn
C          parm          beforeimg
C          parm          afterimg
* End
C          return

```

AN EXPLANATION OF THE CODE

There are two parameters passed into the program by the operating system:

1. A data stream that contains all the information about the triggered event that has just occurred.
2. A four digit binary field that contains the length of the above data string.

The main block of code in the C specs splits up the data stream into the constituent parts that we are interested in:

- The name of the file that generated the trigger is in positions 1 to 10.
- The library the file resides in is in 11 to 20.
- The member name is in 21 to 30.
- The type of trigger generated is in position 31. A value of 1 means

this is an insert, 2 means update, 3 means delete and 4 means file read.

- The CCSID of the file is a 4 byte binary field starting in position 37.
- The relative record number of the record processed is a 4 byte binary number starting at 41.
- The next two fields are the offset (the starting position in the string) and length of the before image of the record.
- These are followed by offset and length of the after image.
- The next two statements extract into single fields the actual before-and-after record information.

After this there is a call to another generic trigger program (TRIGGER2) passing all the parameters we have extracted from the string. For the rest of these programs we are not constrained by the fact that the programs will be locked – they can easily be changed whenever new needs arise.

PROGRAM TRIGGER2

We now have to write the code to perform whatever actions we require for each trigger. We can either put all the logic into program TRIGGER2 or else we can use this purely as a driver program to call a separate program for each triggered file and triggered type. This is my preferred approach as all the trigger programs will be much smaller and easier to maintain. This program assumes that your customer file is called CUSTOMER in library TRIGGEREXP (obviously, you will need to replace this with the name of your actual file).

```

D file          s          10
D library       s          10
D member        s          10
D beforeimg     s          32767
D afterimg      s          32767
D triggerevt    s          1
D triggertme    s          1
D username      s          10
D ccscid        s          9b 0
D rrn           s          9b 0
C *entry        plist
C          parm          file
C          parm          library
C          parm          member
C          parm          username
C          parm          triggerevt
C          parm          triggertme
C          parm          ccscid
C          parm          rrn
C          parm          beforeimg
C          parm          afterimg
* Select group that only processes customer file
C          select
C          file          wheneq    'CUSTOMER'
C          library       andeq    'TRIGGEREXP'
* Select group that only processes updates
C          select
C          triggerevt    wheneq    '3'
C          call          'CUSTTRGUPD'
C          parm          beforeimg

```

```
C          parm          afterimg
* end select groups
C          ends1
C          ends1
* end
C          return
```

This program will receive every trigger for every file. In its embryo state as above we are only performing any actions when the file is called CUSTOMER, the library is TRIGGEREXP and it is an update, which is trigger type 3.

The table below details the trigger types and which record images are passed:

| Trigger type | Text | Before image | After image |
|--------------|--------|--------------|-------------|
| 1 | Insert | | Y |
| 2 | Delete | Y | |
| 3 | Update | Y | Y |
| 4 | Read | Y | |

**PROGRAM CUSTTRGUPD –
PROCESS TRIGGERS WHEN A CUSTOMER RECORD IS UPDATED**

```
D b_recording e ds          extname(customer) prefix(b_)
D a_recording e ds          extname(customer) prefix(a_)
D beforeimg  s              32767
D afterimg   s              32767
D command    s              80
D commandlen s              15 5
D message    s              80
* Parameters passed by TRIGGER2
C *entry     plist
C           parm          beforeimg
C           parm          afterimg
* Set before and after data structures to values from original trigger
C          eval          b_recording=beforeimg
C          eval          a_recording=afterimg
* Only want to send message if account balance is over limit now
* but was not over credit limit before
C          if            b_accbal <= b_crdlmt
C                      and a_accbal > a_crdlmt
C          eval          message='Account ' + a_accnum +
C                      ' has exceeded their credit limit'
C          eval          command='SNDMSG MSG('' + %trim(message)
C                      + ''')' + ' tousr(QSYSOPR)'
C          call          'QCMDEXC'
C          parm          command
C          parm          80          commandlen
C* End of if group
C          endif
C* End program
C          return
```

The first two lines of this program define two data structures which will contain the before and after images of the customer record being updated. The fields within the data structure will be prefixed by b_ for the before images and a_ for the after images. We are interested in two fields from the customer file, the credit limit (CRDLMT) and the account balance (ACCBAL) – we will now have before and after versions of these two fields called B_CRDLMT and A_CRDLMT as well as B_ACCBAL and A_ACCBAL. We want to send a message when the account is over the credit limit after the update but they weren't before the update.

Obviously, you can change the code to send an email or a text message rather than send a message to the QSYSOPR message queue. If you wish to set up a file for testing these routines an example of the necessary DDS is:

```
A          R RCUSTOMER
A          ACCNUM          8          COLHDG('Account' 'number')
A          ACCNAME         30         COLHDG('Account' 'name')
A          ACCBAL           9P 2      COLHDG('Account' 'balance')
A          CRDLMT           9P 2      COLHDG('Credit' 'limit')
A          K ACCNUM
```

STARTING THE TRIGGER PROCESS RUNNING

The first thing needed is to add a trigger to the physical file that you wish to monitor – this is done with the ADPPFTRG command.

```
ADPPFTRG          FILE (TRIGGEREXP/CUSTOMER)          TRGTIME (*AFTER)
TRGEVENT (*UPDATE) PGM ( TRIGGEREXP/TRIGGER1) TRG (CustomerUpdateTrigger)
```

You should get a confirmation message saying: 'Trigger added to physical file.'

You will need to add one trigger for each I/O type you wish to process, ie insert, update, delete and read. It is recommended that you give the triggers a meaningful naming structure as in the example above. This makes it much easier if you ever want to permanently stop the trigger which you do with the command RMVPFTRG. For example:

```
RMVPFTRG FILE (TRIGGEREXP/CUSTOMER) TRG (CustomerUpdateTrigger)
```

Once the programs have been compiled and the triggers added to the file, the process will be live. The TRIGGER2 program can be changed whenever you wish to handle new files and different trigger events. ■

If you like to download a fully working version of the code outlined here you can do so from www.uti400.com/triggerdownload.

STEVE CLOSE

Steve Close is the technical director of utilities 400 Limited.
He can be contacted at sclose@uti400.com