

[print](#) | [close](#)

APIs by Example: Locking Parts of a User Space

[System iNetwork Programming Tips Newsletter](#)

[Carsten Flensburg](#)

Carsten Flensburg

Thu, 03/15/2007 (All day)

User spaces are a powerful tool in every System i programmer's toolbox. One interesting way to use them is as a means of communicating between different jobs in the system. Because a pointer can address user spaces, you can use them the same way that you use variables, making passing a large amount of data between jobs easy.

When you program in this manner, you have to be careful that two different jobs aren't changing the user space at the same time. The Lock Space Location (LOCKSL) MI built-in function provides a means of locking sections of a user space in much the same way that you'd lock records in a database file.

In today's APIs by Example, I show you how to combine the use of the Space Location MI built-ins and user space APIs to address the problem of concurrent update conflicts.

Background Information About MI Built-ins

MI built-in functions provide access to the same wealth of system information and functions as the underlying MI instructions. Whereas MI instructions are available only to the MI compiler, the MI built-ins are readily available to all ILE compilers.

Although MI built-ins look like procedure calls, in reality they're direct interfaces to corresponding MI instructions. They make MI instructions directly accessible to all ILE languages. The ILE C and C++ languages also have their own wrappers for some of the MI function interfaces, documented in the ILE C/C++ MI Library Reference. A link to this manual is included at the end of this article.

It is unnecessary to reference any service program or binding directory during compilation to make the built-ins available to the compiler. All the ILE compilers have implicit access to the MI built-ins. In fact, the compiler often replaces an MI built-in reference with an inline call to the actual MI function to avoid the overhead connected to an external call.

Each MI built-in is documented in the Bound Program Access box of the section documenting the corresponding MI instruction. Some MI instructions are unavailable as an MI built-in or a C/C++ MI Library Function. A missing Bound Program Access box is an indication of this situation. For IBM's online MI instruction documentation, please follow the link at the end of this article.

As for the specific name of an MI built-in, which is most often an underscore-prefixed and digit-suffixed version of the MI instruction name, the *C/C++ MI Library Function* manual is a good source of information. Check out the "Notes on Usage" section of the documentation for each MI library function; if one or more MI built-ins are available for the function, the name(s) of the built-in(s) can be found here.

Locking Locations in a User Space

The MI instructions available to control and avoid simultaneous updates to the same location within a user space are listed as part of the Retrieve Pointer to User Space (QUSPTRUS) API documentation. I've included links to this and all other user space API documentation at the end of this article. This example is based on the MI LOCKSL instruction, which is also available as the `_LOCKSL1` and `_LOCKSL2` MI built-ins. I use the `_LOCKSL1` built-in.

The `_LOCKSL1` built-in grants the space location to the issuing process according to the lock request. For this function, if the requested lock cannot be immediately granted, the process enters a synchronous wait for the lock for a period up to the interval specified by the process default timeout value. This interval is defined by a job's default wait time (DFTWAIT) attribute.

If the wait time is exceeded, the MCH5804 exception is sent to the process waiting for the lock to be granted. To avoid the resulting function check and a possible escalation of the error condition, you need to place a monitor group around the call of the `_LOCKSL1` built-in, or put similar evasive actions into play (e.g., error opcode extender, condition handler).

The `_LOCKSL2` built-in version of the LOCKSL MI instruction lets you specify a wait time that overrides the job's default wait time value. Both built-ins offer a cheap (in terms of CPU cycles) way to lock a specific user space, ensuring that competing updates are carried out sequentially. In this example, I demonstrate this facility for the purpose of creating an incrementing counter function, but any other user space-based store and retrieve function could of course also be implemented with this technique.

To release a space location lock, there's a corresponding Unlock Space Location (UNLOCKSL) MI instruction, which also has a couple of MI built-in equivalents. The `_UNLOCKSL1` built-in unlocks a single space location lock, and the `_UNLOCKSL2` built-in can release multiple space location locks in one call. In this example, UNLOCKSL1 is demonstrated.

The `_LOCKSLn` and `_UNLOCKSLn` built-ins support the following lock states:

LSRD_LOCK	Lock-Shared-Read
LSRO_LOCK	Lock-Shared-Read-Only
LSUP_LOCK	Lock-Shared-Update
LEAR_LOCK	Lock-Exclusive-Allow-Read
LENR_LOCK	Lock-Exclusive-No-Read

I have included two versions of the incrementing counter facility: one that demonstrates the core functionality of the involved built-ins and APIs, and another that shows how to wrap the built-in and API calls up in service program procedures, to increase the reusability and ease the implementation of the counter functions.

Here's the outline of the basic counter function program, which is designed to run in two or more parallel jobs:

1. A user space is created to store the current counter value. The user space is created with the Replace parameter set to `*NO`, to ensure that jobs submitted later do not replace the first user space created.
2. A pointer to the user space is retrieved by the QUSPTRUS API.

3. The `_LOCKSL1` built-in takes the user space pointer as its first parameter and a lock state as its second parameter. The lock state is set to `LENR_LOCK` (Lock-Exclusive-No-Read). While this lock is in effect for the space location identified by the pointer to the user space, no other MI lock mechanism is granted a lock to the user space location.
4. While the user space location is locked, the numeric value mapped to the user space is updated as 1 is added to the current value. The resultant value is stored in a program variable, to allow the lock to be released.
5. The `_UNLOCKSL1` built-in releases the lock obtained in step 3, making the user space location available to other processes requesting a lock.
6. The numeric value calculated and saved in step 4 is written to an output file, together with other information identifying the program name, job number, and system timestamp of the event.
7. Steps 3 to 6 are repeated 10,000 times.

The service-program-supported version of the counter function program performs basically the same steps, so irrespective of the version you run, you end up with a test data file containing 20,000 records, if you run the test using the provided test CL programs. The data will look similar to the following sample obtained using the Run Query (`RUNQRY`) command:

Line+.....1.....+.....2.....+.....3.....+.....4.....+.....5.....+.....6.....+.....	Record ID	Program name	Job number	Record timestamp
000001	13.39.49.039000	1	CBX1701	617731	2007-03-12-
000002	13.39.49.062000	2	CBX1701	617730	2007-03-12-
000003	13.39.49.066000	3	CBX1701	617731	2007-03-12-
000004	13.39.49.083000	4	CBX1701	617730	2007-03-12-
000005	13.39.49.083000	5	CBX1701	617731	2007-03-12-
000006	13.39.49.083000	6	CBX1701	617730	2007-03-12-
000007	13.39.49.084000	7	CBX1701	617731	2007-03-12-
000008	13.39.49.084000	8	CBX1701	617730	2007-03-12-
000009	13.39.49.084000	9	CBX1701	617731	2007-03-12-
000010	13.39.49.084000	10	CBX1701	617730	2007-03-12-

You could also use the Display Physical File Member (`DSPPFM`) command to inspect the test data file, but the `RUNQRY` method formats the data a little nicer. There's no guarantee that the records are output in the same sequence as they are numbered; that depends on how the system's database manager jobs are writing the records to disk. But the records will be in sequence for each job number, and there will be no duplicate record IDs.

The two test programs, CBX1701T and CBX1702T, respectively, each submit two identical jobs to job queue QSYSNOMAX in subsystem QSYSWRK. If this is unacceptable on your system, change the SBMJOB command's Job Queue (JOBQ) parameter in both programs to the preferred job queue name. Please note, however, that for the test to work, it has to be a multithreaded job queue. In this context, a multithreaded job queue defines a job queue that lets more jobs having the same job priority be active at the same time.

To verify a job queue's operational attributes, run the command Display Subsystem Description (DSPSBSD) and select option 6, *Job queue entries*. Check out job queue QSYSNOMAX in subsystem QSYSWRK to see the operational attributes of a multithreaded job queue.

Here are the few steps involved in running the basic test version:

1. Add the library containing the compiled objects to your job's library list, if it is not there already:

```
ADDLIB LIB(library name)
```

2. Call the test program CBX1701T from a command line. This program submits two jobs CBX1701T to job queue QSYSNOMAX:

```
CALL PGM(CBX1701T)
```

3. After the two submitted jobs have completed, use the RUNQRY command to verify the content of the test data file CBX1701F:

```
RUNQRY QRYFILE(CBX1701F)
```

To run the second version of the test program, call program CBX1702T in step 2 and run the query against the file CBX1702F in step 3. Please note that this APIs by Example merely serves as a proof of concept with respect to the combination of user spaces and the lock space location MI instructions, as well as a useful (I hope) starting point for your own employment of this facility.

This APIs by Example includes the following sources:

```
CBX170  -- RPGLE  -- Counter ID Functions - service program
CBX170_B -- SRVSR  -- Counter ID Functions - binder source
CBX170_H -- RPGLE  -- Counter ID Functions - prototypes
CBX1701  -- RPGLE  -- Lock Space Location MI Built-in - test 1
CBX1701T -- CLP    -- Lock Space Location MI Built-in - test 1
CBX1702  -- RPGLE  -- Lock Space Location MI Built-in - test 2
CBX1702T -- CLLE   -- Lock Space Location MI Built-in - test 2

CBX1701F -- PF      -- Counter Data File - test 1
CBX1702F -- PF      -- Counter Data File - test 2
```

To create all these objects, follow the compilation instructions in the source headers.

This article demonstrates the following MI built-in functions:

Lock Space Location (LOCKSL|_LOCKSL1) built-in:

<http://publib.boulder.ibm.com/infocenter/iseres/v5r4/topic/rzatk/LOCKSL.htm>

Unlock Space Location (UNLOCKSL|_UNLOCKSL1) built-in:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/rzatk/UNLOCKSL.htm>

All MI instructions and built-in functions are documented here:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/rzatk/mitoc.htm>

ILE C/C++ MI Library Functions:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r3/topic/books/sc092418.pdf>

This article demonstrates the following user space APIs:

Retrieve Pointer to User Space (QUSPTRUS) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qusptrup.htm>

Create User Space (QUSCRTUS) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/quscrtus.htm>

Delete User Space (QUSDLTUS) API:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/qusdltus.htm>

All user space APIs are documented here:

<http://publib.boulder.ibm.com/infocenter/iserics/v5r4/topic/apis/obj5.htm>

You can retrieve the source code for this API example from:

http://www.pentontech.com/IBMContent/Documents/article/54282_182_LockUserSpace.zip.

Source URL: <http://iprodeveloper.com/rpg-programming/apis-example-locking-parts-user-space>