developerWorks®

Debug IBM i programs graphically

A step-by-step introduction to IBM i debugger

Jin Jiang Zhang
Li Jiu Yu
Shuang Hong Wang

October 14, 2011

Debugging IBM i programs can be challenging if you are new to IBM i or used to debugging applications with graphical tools. This article shows you how to debug programs graphically using the integrated IBM i debugger.

Overview of IBM i debugger

A debugger tool is a critical component of application development environments. Developers can use the debugger to isolate problems, understand software code paths and even evaluate potential code changes. And a powerful debugger can greatly enhance the efficiency of a software developer.

IBM i developers are familiar with the green screen debuggers that are available on IBM i and its predecessor platforms (AS/400, iSeries, and System i). The green screen debuggers are accessed with the `STRDBG` (Start Debug) and `STRISDB` (Start Interactive Source Debugger) commands. While these debuggers work, they do not provide the productivity and ease-of-use features of modern graphical debuggers. The good news is that there are two graphical debug tools available from IBM to boost your productivity.

Most IBM i developers are only aware of the graphical debugger that's part of the IBM Rational Developer for Power Systems software offering. The graphical debugger that gets overlooked is the graphical IBM i debugger that is included with the IBM i operating system. This debugger is known as the IBM System i5 Debugger in product documentation, but this article will refer to it as the graphical IBM i debugger for convenience. This integrated graphical debugger was first made available on the V5R1 release of OS/400.

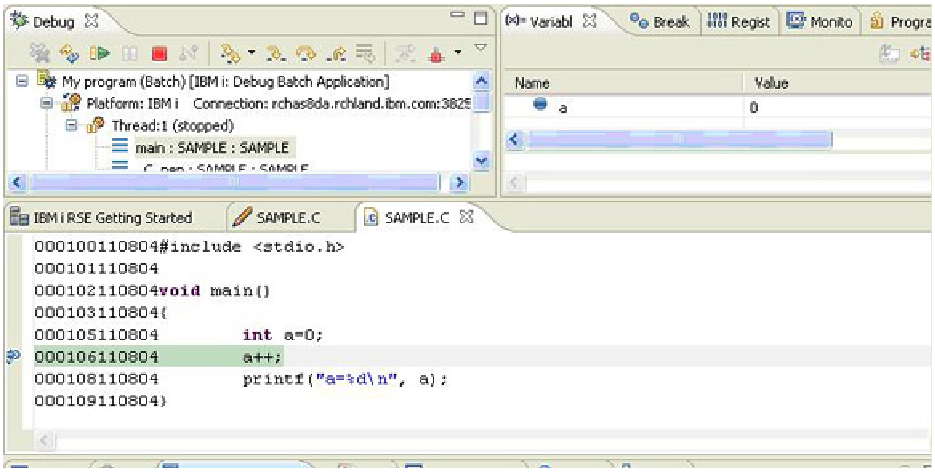
IBM Rational Developer for Power Systems software debugger

The IBM Rational Developer for Power Systems product is an IDE used to develop applications that run on IBM Power Systems. It's a powerful tool which provides RPG and COBOL development tools for IBM i and C/C++ and COBOL development tools for AIX. Developers can use this tool to create, edit, compile, and debug your source code on IBM i. Figure 1 shows the graphical debugger interface supplied by the IBM Rational Developer product. Programmers can also use the tool to create libraries, source physical files, and members on any IBM i server.

The graphical toolset supports the typical steps of creating and debugging application code:

1. Create a new library if you do not want to using the existing ones
2. Create a physical file for containing source code members
3. Create a member for writing source code
4. Compile the source code member with debug information
5. Add the program file, created by the compiler, to service entry point
6. Debug the program (steps are similar to other IDEs)

Figure 1. Rational Developer for Power Systems software



My program (Batch) [IBM i: Debug Batch Application]
Platform: IBM i   Connection: rchas8da.rchland.ibm.com:3825
Thread:1 (stopped)
main : SAMPLE : SAMPLE

Name	Value
a	0

```
000100110804#include <stdio.h>
000101110804
000102110804void main()
000103110804{
000105110804        int a=0;
000106110804        a++;
000108110804        printf("a=%d\n", a);
000109110804}
```

Graphical IBM i debugger

Graphical IBM i debugger can be regarded as the graphical version of traditional the green screen debugger for IBM i. The graphical IBM i debugger is packaged in two JAR files provided by IBM Toolbox for Java. The graphical debugger can run on any platform such as Windows, Linux and Unix platforms that supports Java. The article will discuss this graphical debugger in more detail.

Pros and cons of the two debuggers

As you can see from the content of Table 1, the Rational debugger is the more powerful one between the two debuggers. But it requires much more disk spaces than the graphical IBM i debugger. Graphical IBM i debugger is a lightweight debugger, requiring only about 5M bytes on workstations while providing powerful debug functionalities. Besides, it is very easy to install and you can use it on different platforms in the same way because it is Java based. Most importantly, the graphical IBM i debugger is included with the IBM i operating system, which means there is no additional charge for using the debugger. In contrast, usage of the Rational debugger requires the purchase of the entire Rational IDE product.

Table 1. Comparisons of the debuggers for IBM i

	Rational debugger	Graphical IBM i debugger
Functions	Ability to launch code editor from debug session. Remote debug	Remote debug
Disk space	About 1G bytes	About 5M bytes
User experience	Very good	Good
Cost	Additional license fee required	None, included with IBM i operating system

Installing graphical IBM i debugger

Typically, there are two methods used to install the graphical IBM i debugger. One method is accomplished by installing the IBM System i Navigator client. The second method involves manually copying the required JAR files onto your workstation.

Installing and starting graphical IBM i debugger via IBM System i Navigator

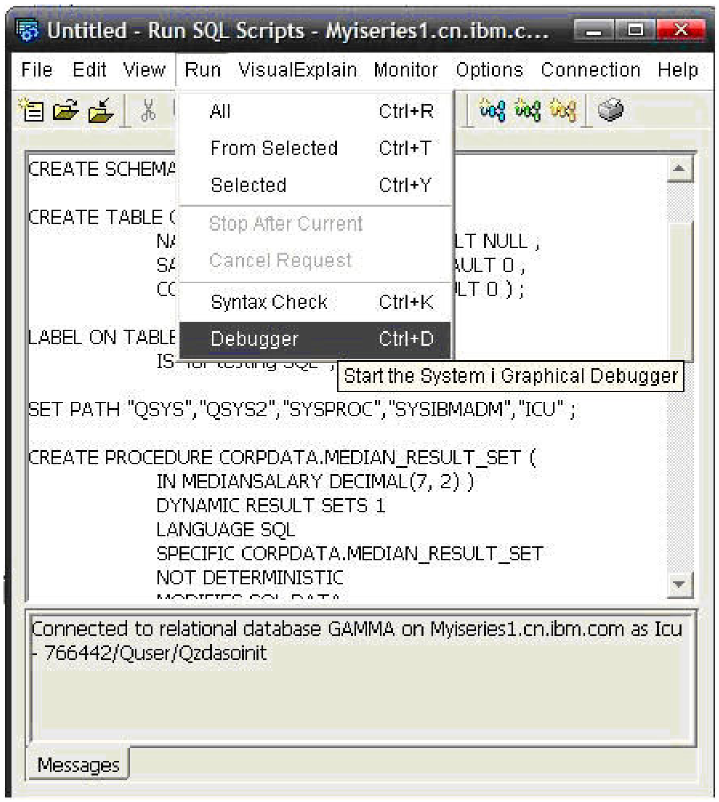
The IBM System i Navigator client provides end users a graphical user interface instead of the traditional green screen for administering and managing IBM i systems. Besides that functionality, it also embeds the graphical IBM i debugger for developers to use. If you already have the IBM System i Navigator client installed on your workstation, then you need not perform any steps to install the graphical IBM i debugger. More information about the IBM System i Navigator offering can be found in the Resource section.

Once the IBM System i Navigator client is installed, starting the debugger is done by performing the following steps:

1. Expand the following items successively in the navigation tree of your IBM System i Navigator client: `My connections -> Your System Name-> Databases`.
2. Right-click on the desired database, and select the `Run SQL Scripts` task.
3. In the `Run SQL Scripts` window, select `the Debugger task from the Run pull-down menu` as shown in Figure 2.

There is no doubt that starting the debugger in this way is more convenient for debugging SQL procedures, functions and triggers. This interface can also be used to start the debugger for programs without SQL.

Figure 2. Starting graphical IBM i debugger from Run SQL Scripts


Untitled - Run SQL Scripts - Myiseries1.cn.ibm.c...
File Edit View Run VisualExplain Monitor Options Connection Help
All Ctrl+R
From Selected Ctrl+T
Selected Ctrl+Y
Stop After Current
Cancel Request
Syntax Check Ctrl+K
Debugger Ctrl+D
Start the System i Graphical Debugger
Messages

```
CREATE SCHEMA
CREATE TABLE (
                NA                 LT NULL ,
                SA                 ULT 0 ,
                CC                 LT 0 ) ;
LABEL ON TABLE
                IS
SET PATH "QSYS","QSYS2","SYSPROC","SYSIBMADM","ICU" ;
CREATE PROCEDURE CORPDATA.MEDIAN_RESULT_SET (
                IN MEDIANSALARY DECIMAL(7, 2) )
                DYNAMIC RESULT SETS 1
                LANGUAGE SQL
                SPECIFIC CORPDATA.MEDIAN_RESULT_SET
                NOT DETERMINISTIC
```

```
Connected to relational database GAMMA on Myiseries1.cn.ibm.com as Icu
- 766442/Quser/Qzdasoinit
```

Installing graphical IBM i debugger via IBM Toolbox for Java

If you do not have the ability to install the IBM System i Navigator client, there is an alternative way to install the graphical IBM i debugger. This method copies specific components out of the IBM Toolbox for Java. The toolbox which consists of a set of JAR files, is not designed specifically for debugging programs on IBM i, but for developing Java programs that utilize IBM i resources.

Specifically, only two JAR files are needed on your workstation and they are provided by the IBM Toolbox for Java. There are two methods to get the JAR files required by the graphical debugger onto your workstation. One is by downloading the JTOpen package onto your workstation. JTOpen is the open source versions of the IBM Toolbox for Java. Both packages contain identical code. The second method is by accessing a system that has the IBM Toolbox for Java installed. The first method is pretty simple and straightforward, so that method is not discussed in detail. Refer to the IBM Toolbox for Java item in the Resource section. The second method is outlined below.

To get the two JAR files from an IBM i system, you need access to a server which has the IBM Toolbox for Java package installed. The IBM Toolbox for Java is not required to be installed on a system in order to debug programs running on it using the graphical IBM i debugger. By default, the toolbox should already be installed on your system. To check if the IBM Toolbox for Java is installed on your system, follow these steps:

1. Logon to your IBM i system
2. On a command line, enter `go licpgm`, which navigates you to the `Work with Licensed Programs` panel.

3. Select option 10, `Display installed licensed programs`. This option produces a list of the currently installed licensed programs.
4. Page down to view the installed programs in the `Display Installed Licensed Programs` panel. If the IBM Toolbox for Java product appears in the list, then you know that the product is installed.

If the toolbox is not installed, follow the steps below to install the IBM Toolbox for Java:

1. On a command line, enter `go licpgm`.
2. Select option 11, `Install licensed programs`.
3. Type 1 in the `Option` column on the entry named `Extended Base Directory Support` to install the toolbox.
4. After the installation, verify that the IBM Toolbox for Java is installed using the previously documented steps.

For detailed information and prerequisites on installing IBM Toolbox for Java, refer to the book IBM Toolbox for Java, which is found in the IBM i Information Center.

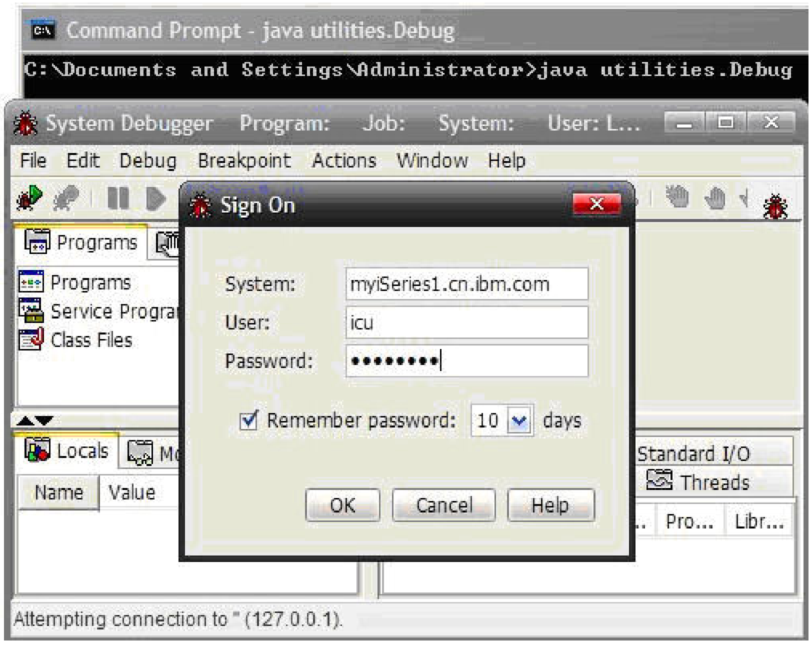
Assuming that the IBM Toolbox for Java is installed, the next step is getting the needed JAR files installed on your workstation. The installation steps documented here are for a Windows workstation. The installation procedure will be similar for other types of workstations.

1. Copy the two JAR files, jt400.jar and tes.jar, from `/QIBM/ProdData/HTTP/Public/jt400/lib/` on IBM i (or from the JTOpen package you just copied to your workstation). These two JAR files are the IBM Toolbox for Java components which provide the graphical debugger functionality.
2. Add the path for the two JAR files to your CLASSPATH environment variable. For example, if you copied the two JAR files to C:\iDebugger\ on your Windows workstation, you should update the CLASSPATH variable by appending the following line: ;C:\iDebugger\jt400.jar;C:\iDebugger\tes.jar

At this point, the graphical IBM i debugger installation is completed. In order to run the graphical debugger, you must have a proper JDK or JRE version installed on your workstation. The minimum JDK and JRE version required depends on the IBM i OS version from which you copy the JAR files. For example, the minimum version required for the IBM i V5R4 release is Java version 1.3.1. The IBM i 7.1 version of the IBM Toolbox for Java requires version 1.4 Java. A detailed listing of the software requirements for the graphical IBM i debugger can be found in the IBM i Information Center. In order to access the graphical debugger's help documents successfully, you need to have the jhall.jar file added to your CLASSPATH. This JAR file can be obtained from the JavaHelp system. For more detailed information about prerequisites and installation of the graphical IBM i debugger, please refer to the IBM i Information Center book titled, *System i5 Debugger*

You can invoke graphical IBM i debugger with the following command from a Windows command prompt to verify your installation: `java utilities.Debug`. Your result should be similar to the graphical debugger output in Figure 3.

Figure 3. Starting IBM i System Debugger


```
C:\Documents and Settings\Administrator>java utilities.Debug
```
System: myiSeries1.cn.ibm.com
User: icu
Password: •••••••••
✓ Remember password: 10 days

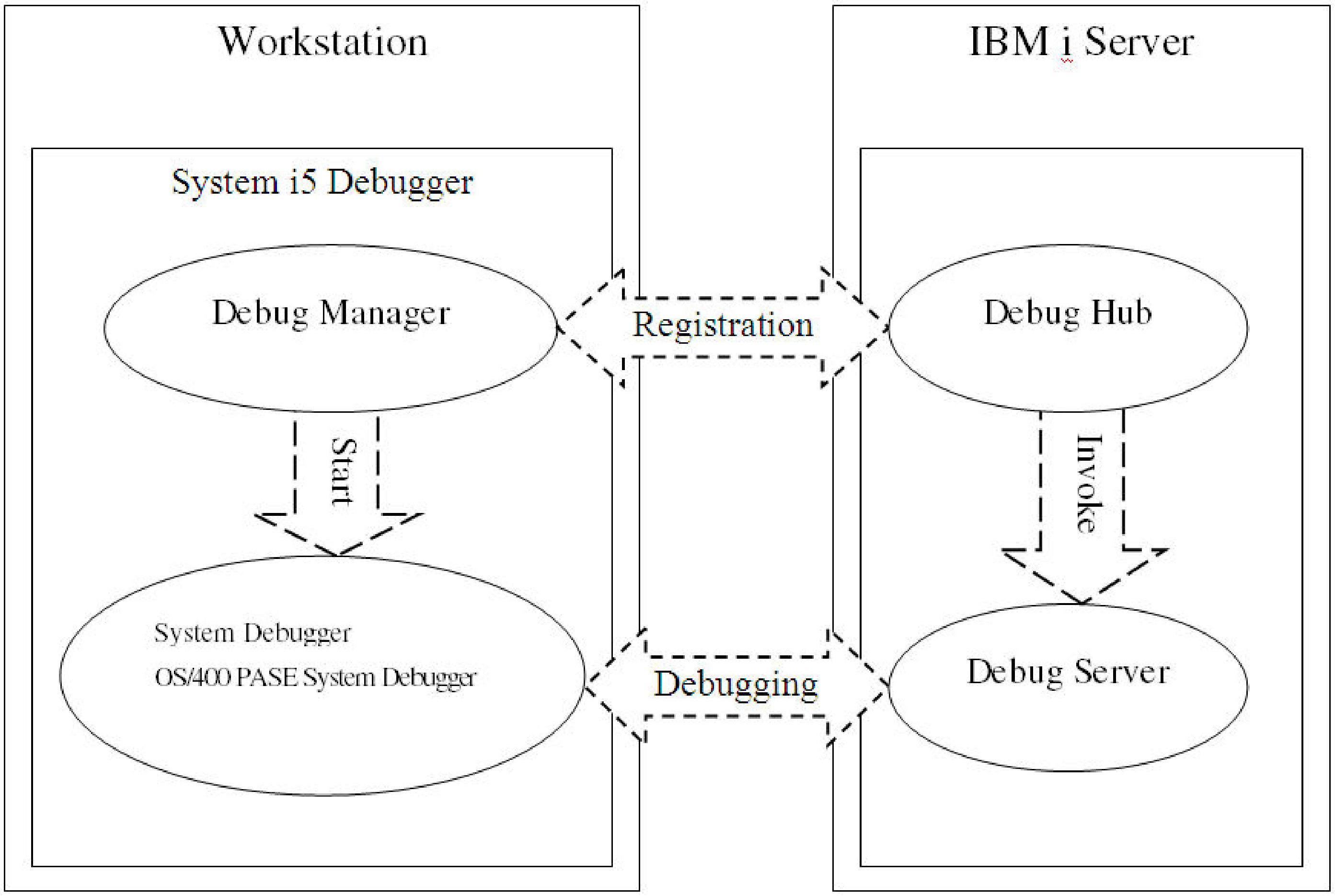
Attempting connection to '' (127.0.0.1).

Command Prompt - java utilities.Debug
System Debugger Program: Job: System: User: L...
File Edit Debug Breakpoint Actions Window Help
Sign On
Programs
Programs
Service Progra
Class Files
Locals
Name
Value
Standard I/O
Threads
Pro...
Libr...
OK
Cancel
Help

Overview of graphical IBM i debugger

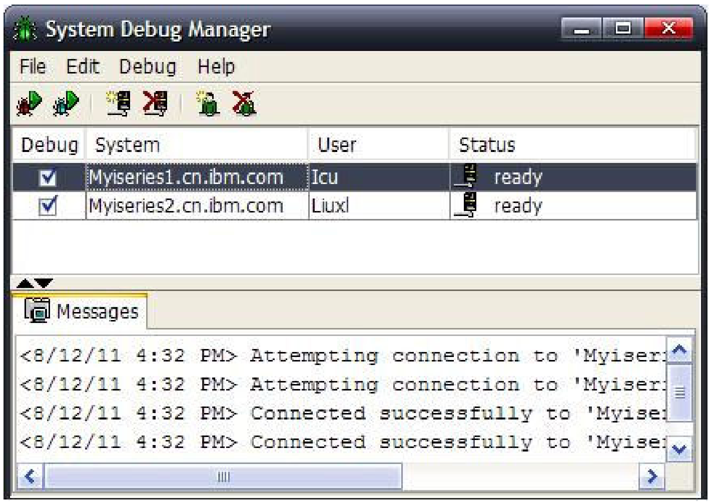
Let's first look into the debugger's architecture. As shown in Figure 4, on the workstation side, graphical IBM i debugger is composed of theree components: Debug Manager, System Debugger and OS/400 PASE System Debugger. On the IBM i side, the graphical debugger relies on the Debug Hub and Debug Server components.

Figure 4. Graphical IBM i debugger architecture



The Debug Manager is responsible for registering clients with the Debug Hub components, so that the graphical debugger will be enabled on IBM i. Besides that, it helps manage your debug settings by adding or removing systems or users. The Debug Manager provides a convenient interface to start the debugger if you have to debug programs on many different systems. Figure 5 displays the user interface of the Debug Manager. The Debug Hub component stored the user information in its registry. By storing that user information, the Debug Hub knows to automatically launch the graphical IBM i debugger when that same user issues the `STRDBG` CL command. Another function performed by the Debug Hub component is to invoke the Debug Server when it receives requests from remote workstations. For instance, when you start a graphical debug session from the Debug Manager or Windows command line, a request will be received by the Debug Hub. The Hub then turns around and creates a corresponding Debug Server instance to handle debugging commands from the debug session. The PASE System Debugger is for programs that run in a PASE environment. This debugger provides similar debugging functions as the graphical IBM i debugger as well as some other functions specific to the PASE environment. For additional information on the PASE environment, reference the book titled, PASE for IBM i, in the IBM i 7.1 Information Center. The graphical IBM i debugger can debug programs written in ILE and non-ILE languages as well as Java.

Figure 5. Debug Manager



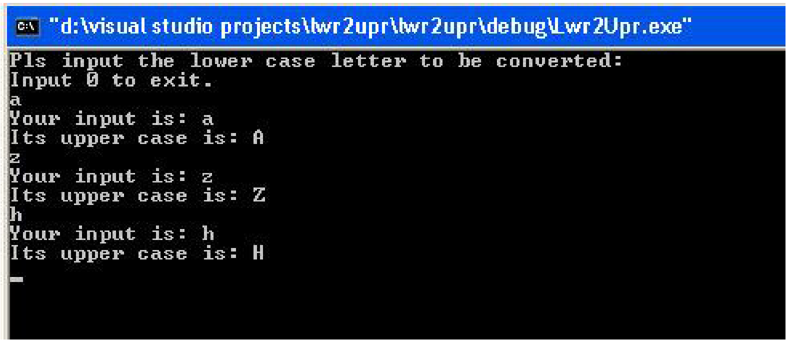
Debug	System	User	Status
✓	Myiseries1.cn.ibm.com	Icu	ready
✓	Myiseries2.cn.ibm.com	Liuxl	ready

```
<8/12/11 4:32 PM> Attempting connection to 'Myiser
<8/12/11 4:32 PM> Attempting connection to 'Myiser
<8/12/11 4:32 PM> Connected successfully to 'Myise
<8/12/11 4:32 PM> Connected successfully to 'Myise
```

Graphical IBM i debugger in action

This section, will show you how to utilize the graphical IBM i debugger by debugging a sample program. The sample C program provides a simple function – it converts a lower case English letter to its corresponding upper case letter. The program runs without any errors in a Windows environment as shown by the successful output in Figure 6.

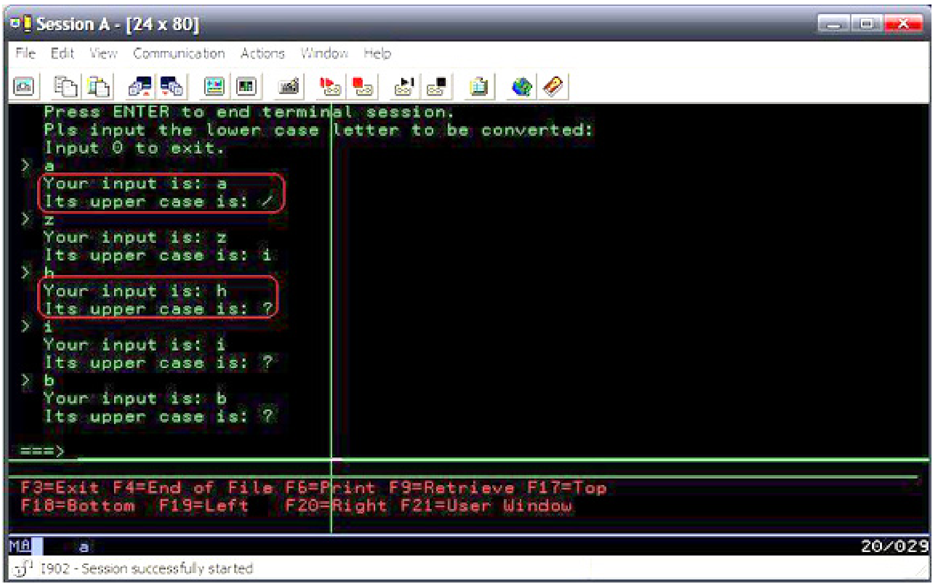
Figure 6. Output from successful execution of sample program


"d:\visual studio projects\lwr2upr\lwr2upr\debug\Lwr2Upr.exe"

```
Pls input the lower case letter to be converted:
Input 0 to exit.
a
Your input is: a
Its upper case is: A
z
Your input is: z
Its upper case is: Z
h
Your input is: h
Its upper case is: H
_
```

Running the same C program logic on IBM i, produces the incorrect output displayed in Figure 7. For example, the C program running on IBM i incorrectly converts the lower-case letter 'a' to '/' and converts the lower-case letter 'z' to 'i'.

Figure 7. Incorrect output from program running on IBM i



```
   Press ENTER to end terminal session.
   Pls input the lower case letter to be converted:
   Input 0 to exit.
 > a
   Your input is: a
   Its upper case is: /
 > z
   Your input is: z
   Its upper case is: i
 > h
   Your input is: h
   Its upper case is: ?
 > i
   Your input is: i
   Its upper case is: ?
 > b
   Your input is: b
   Its upper case is: ?

 ===>

 F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
 F18=Bottom  F19=Left   F20=Right F21=User Window
MA      a                                              20/029
I902 - Session successfully started
```

Let's use the graphical IBM i debugger to find the cause of the incorrect output. The source code of the sample program is included in Listing 1 for your reference.

Listing 1. Source code of the sample program

```
//LWR2UPR.c source code.
//This is the sample code used for debugging, which would fail
//when running on IBM i system.

#include <stdio.h>

//Convert a lower case letter to its corresponding upper case letter.
//Return value 0 indicates successful process, otherwise return 1;
int convert2Captial(char* pChar)
{
 //determine if the user's input is legal
  if((*pChar < 'a') || (*pChar > 'z'))
 {
  printf("Input is error. Pls input a letter between a and z:\n");
  return 1;
 }
 else
   {
       *pChar -= 32;
   return 0;
 }
}

int main(void)
{
   char chInput; //The letter that user inputs
   char* pChar; //Pointer to the letter
 int nRet;

 printf("Pls input the lower case letter to be converted:\n");
 printf("Input 0 to exit.\n");

 //Gets letter from terminal:
   while ((chInput = getchar()) != '0')
   {
    pChar = &chInput;
  if(*pChar != '\n')
  {
   printf("Your input is: %c\n", *pChar);
  }
    else
  {
   continue;
  }

    if (0 == (nRet = convert2Captial(pChar)))
  {
   printf( "Its upper case is: %c\n", *pChar);
  }
    else
  {
   continue;
  }
 }

 printf("Size of a inteter in byte is: %d\n", sizeof(nRet));
 printf("Size of a char in byte is: %d\n", sizeof(chInput));
 printf("Size of a pointer in byte is: %d\n", sizeof(pChar));

 return 0;
}
```

Logon to the graphical IBM i debugger

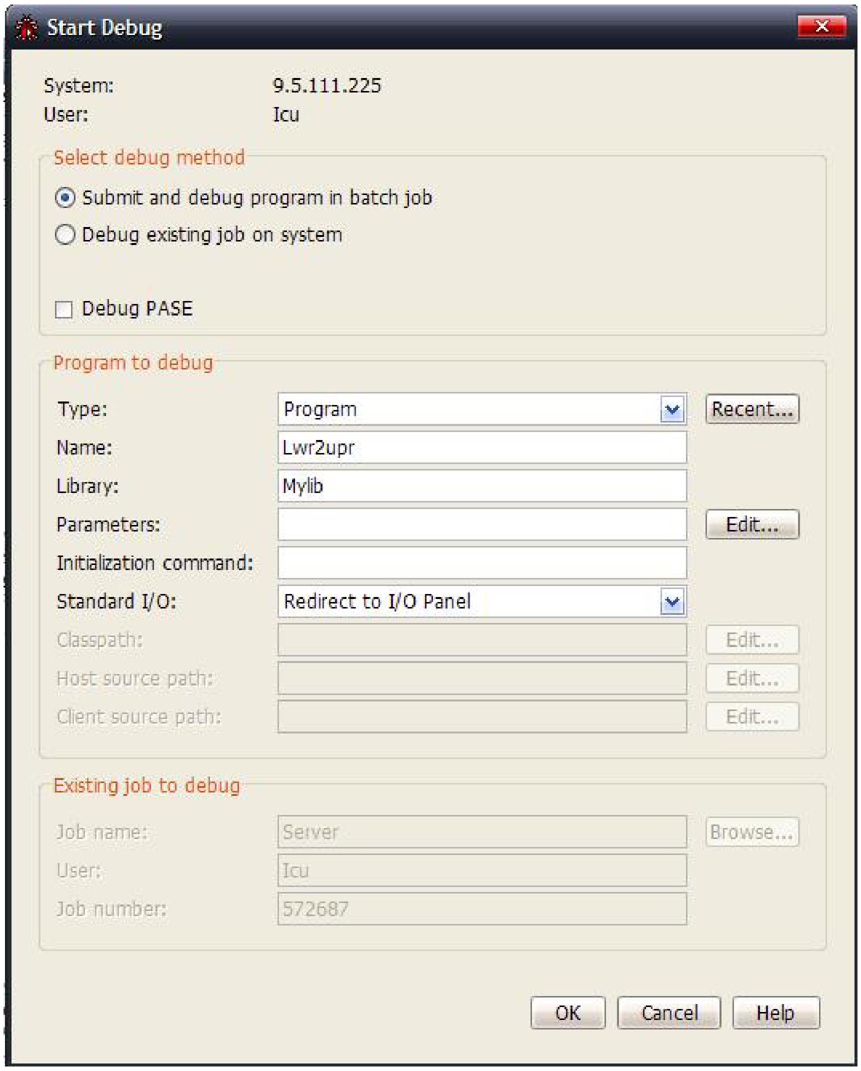
From the Debug Manager interface shown in Figure 3, use your IBM i user profile and password to start a debug session by logging onto the IBM i system containing the program that is being debugged.

Start the program debug session

Once you are logged on, you should automatically be prompted with the `Start Debug` panel displayed in Figure 8. If the `Start Debug` panel does not appear automatically, press the `F4` key or select the `Start` task from the Debug pull-down menu.

As shown in this panel, you can debug a program in two different modes. The first mode, `Submit and debug program in batch job,` is for the debug of a program which is not yet running. The second mode, `Debug existing job on system,` is for debugging a program that already is running on the targeted system. This mode is more suitable for programs that initialize successfully or programs that operate as a server.

Figure 8. Specify the program to debug



Start Debug

System: 9.5.111.225
User: Icu

Select debug method
(●) Submit and debug program in batch job
( ) Debug existing job on system
[ ] Debug PASE

Program to debug

Type:	Program	Recent...
Name:	Lwr2upr	
Library:	Mylib	
Parameters:		Edit...
Initialization command:		
Standard I/O:	Redirect to I/O Panel	
Classpath:		Edit...
Host source path:		Edit...
Client source path:		Edit...

Existing job to debug

Job name:	Server	Browse...
User:	Icu	
Job number:	572687	

This debug example will use the `Submit and debug program in batch job` mode. The next step is supplying the appropriate contents for the `Program to debug` section. The values for this section are very important to debug the intended program. There are three possible values for the Type field: "Program" and "Class File". The "Class File" value is for Java programs, while the "Program" value is for programs that are written in other high-level languages, such as C, RPG and COBOL. A final value, "Service Program" is available if you choose to debug an existing job. A Service Program on IBM i is similar to a shared library on other platforms. A service program cannot be run directly, so that is why the value is only available when using the debug existing job mode. The sample program is composed in C, so the "Program" value is specified for program type.

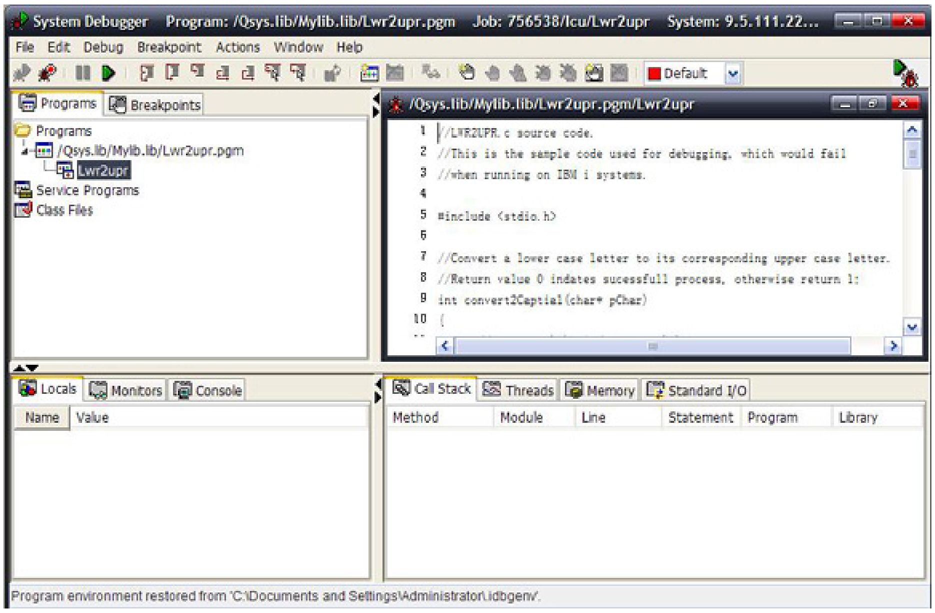
Next, you need to fill in the program name and the library name of the IBM i program object being debugged. There are no additional parameters and initialization commands for the sample program, so leave those fields blank. The final field in the `Program to debug` section is the `Standard I/O` field. The parameter sets where the program's output will be directed. There are two options for this field: "Send to spool file" and "Redirect to I/O panel". If your program's output is large, then a spool file is a good choice for this field. With the other option, you can redirect your program's output to the graphical IBM i debugger's I/O panel, which will be discussed later in this article. If your program is interactive and requires the user to input value, then you should choose the I/O panel option. Furthermore, this option allows you to easily check your program's output on the debugger's I/O panel without logging on to IBM i to view the contents of a spool file. For this example, the "Redirect to I/O Panel" option is used. With all of the input fields completed, you can click the OK button to start debugging the sample code.

If you want to debug an existing job, you have to obtain the job's information on IBM i to fill in the corresponding entries in `Existing job to debug` field. There are two methods to get the job information. The first method is to press the `Browse` button on the bottom half of the `Start Debug` window in Figure 8. Selecting this button will result in a dialog window that presents filter options to help you will find the job that needs to be debugged. The second method is to log on to IBM i, enter command `wrkactjob`, and scroll down to find the job you want to debug.

Reproduce the problem using the debugger

The graphical IBM i debugger's user interface is shown in Figure 9. The interface is divided into four windows. Source code for the debugged program is displayed in the top-right window, called the `Source Window`. If the source code is not displayed in the window, one of the possible reasons may be that the program object does not contain necessary debug information. In this case, please recreate the program with proper debug parameter value. The remaining windows contain different tags in them. For statement simplicity, the article will refer to them by their tag names regardless of their location. For instance, the Programs tag located in the left-top window will be called the `Programs Panel`.

Figure 9. System Debugger's Interface

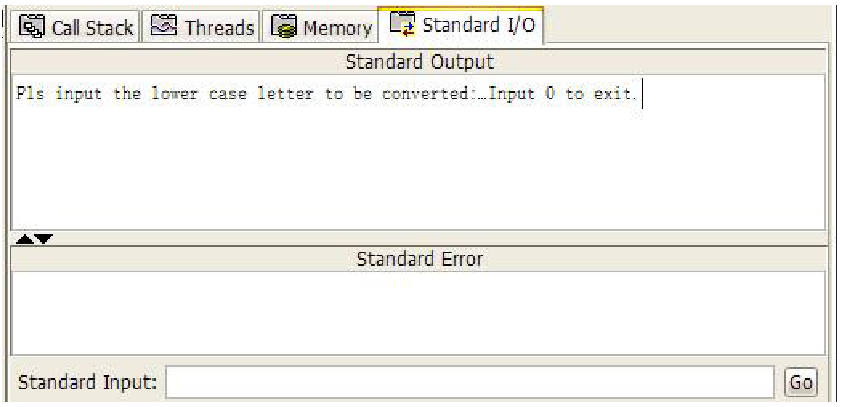


```
//LWR2UPR.c source code.
//This is the sample code used for debugging, which would fail
//when running on IBM i systems.

#include <stdio.h>

//Convert a lower case letter to its corresponding upper case letter.
//Return value 0 indates sucessfull process, otherwise return 1;
int convert2Captial(char* pChar)
{
```

Program environment restored from 'C:\Documents and Settings\Administrator\.idbgenv'.

Since the "Redirect to I/O Panel" option was chosen when starting this program in debugger, all the input and output actions can be performed in the `Standard I/O` panel located in the bottom window on the right. You do not need to interact with the IBM i system with an emulator session. Instead, you can interact with the program from the graphical debugger interface. Just simply click on the `Standard I/O` tab to access the interface panel shown in Figure 10. Then, run the program by selecting the green triangle `Resume` button in the top toolbar or press the `F12` key. After a moment, a message asking the user to input the lower case letter will appear in the `Standard Output` section of the `Standard I/O` window. You can then input the testing letter from the `Standard Input` section and click `Go` button beside the panel.

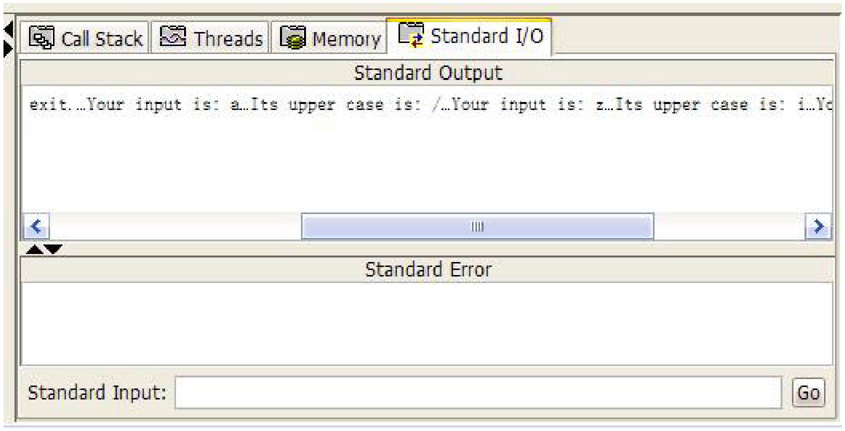
Figure 10. Graphical Debugger Standard I/O panel



```
Pls input the lower case letter to be converted:…Input 0 to exit.
```

Figure 11 contains the outputted result from the program call. The results show that the lower case letter "a" is mistakenly converted to a forward slash "/" and "z" is converted to letter "i", which is the same output the program produces when run on IBM i as shown in Figure 7.

Figure 11. Standard Output Example



```
exit.…Your input is: a…Its upper case is: /…Your input is: z…Its upper case is: i…Yo
```

Review the source code logic

Now that the program error has been reproduced, the next step is to review the source code to determine where the debug breakpoints should be set. The logic of the code is not complicated. First, the sample program accepts an input letter from the terminal as shown in the following code snippet.

```
while ((chInput = getchar()) != '0')
```

Second, if the user has chosen not to exit (the input letter is not "0"), program control is transferred to the function convert2Captial to convert the letter to upper case.:

```
if (0 == (nRet = convert2Captial(pChar)))
```

Third, the function convert2Captial, after checking the input letter is indeed a legal lower case letter, converts the letter to its corresponding upper case letter according to their ASCII code points.

Finally, return from the called function to main, and display the results to terminal.

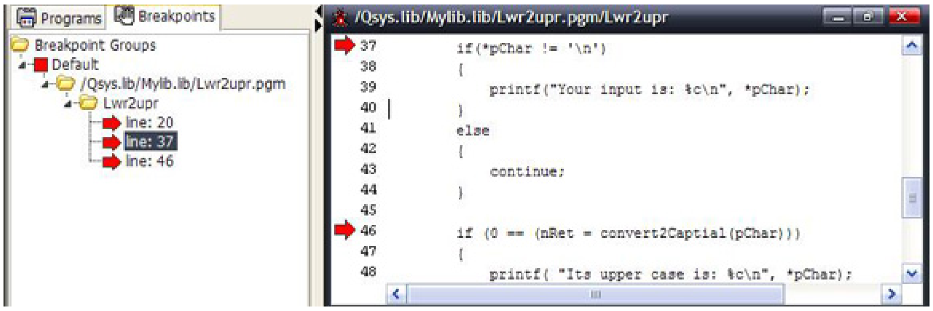
Adding breakpoints

To determine the cause of the incorrect output, breakpoints are added to the following 3 lines of code in the `Source Window` by right-clicking in the corresponding line and choosing the `Add Breakpoint` task or simply double-clicking the left side of the line number.

```
1) return 0;
2) if(*pChar != '\n')
3) if (0 == (nRet = convert2Captial(pChar)))
```

The debugger screen should now look similar to the screen displayed in Figure 12. In the `Source Window`, two red arrows are displayed next to the corresponding lines in the source code to signal that breakpoints are associated with those lines of code. If you want to browse all of the breakpoints, just click the `Breakpoints` tab. From this tab you are provided useful property settings about each breakpoint such as condition of the breakpoint, thread settings if the program is multi-threaded and the style. You can right-click the node in the `Breakpoints` window and select `Edit Breakpoint…` or `Breakpoint Properties…` to display the corresponding setting dialogs. In this example, a condition of `*pChar !='A'` could be added to the breakpoint on line 20 so that the program execution is only stopped when an incorrect result is produced when the input is lower case letter 'a'. This conditional breakpoint capability provides developer a lot of flexibility in the debug process. Besides that, you can also change the breakpoints symbol color by creating different breakpoint groups. Different breakpoint colors may help you manage breakpoints for debugging different types of problems within a complex program as shown in Figure 12.

Figure 12. Breakpoints in groups



```
37        if(*pChar != '\n')
38        {
39            printf("Your input is: %c\n", *pChar);
40        }
41        else
42        {
43            continue;
44        }
45
46        if (0 == (nRet = convert2Captial(pChar)))
47        {
48            printf( "Its upper case is: %c\n", *pChar);
```

It's possible you may not find the bug in the program before having to go on to other activities and close the debugger. What happens to all the breakpoints you set when the debugger session is ended? Fortunately, all of the breakpoints are remembered by the debugger. All you have to do is just to start the graphical IBM i debugger and this will restore all aspects of the last session's debug environment such as the breakpoints, the monitored variables, and the program objects being debugged. All of these settings automatically come back the next time you start graphical IBM i debugger which is a great way to improve the productivity of your daily work.

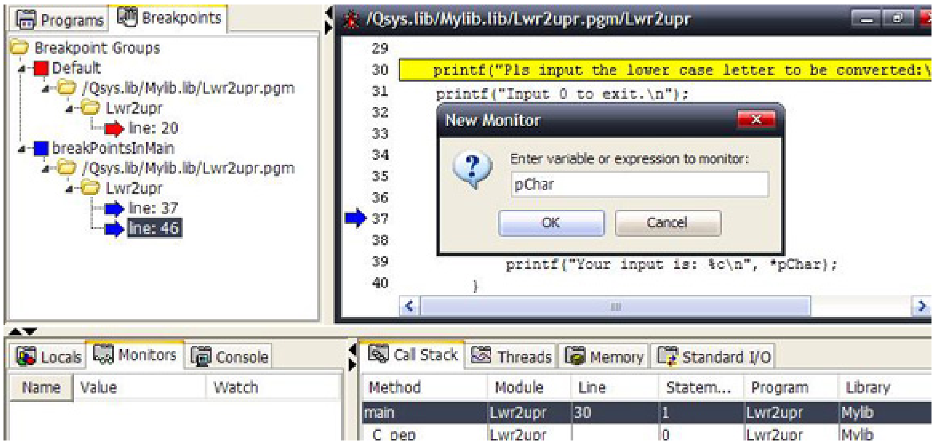
Examining variables

The capability to examine the value of variables is one of the most common functions that any debugger should provide. The graphical IBM i debugger provides four different interfaces to check a variable's value – Source window, Console panel, Locals panel or Monitors panel. Looking at a variable's value in the Source window is simple and straightforward. Either place your cursor over the variable you want to check or highlight the variable in the Source window. Both of these actions will result in the current value of variable being displayed in the message section at the bottom of the debugger window or in a flyover window in the Source window. In Console panel, you can run the `EVAL` command to manually display the value of a variable or expression. The Locals panel is used for monitoring variables within the current function and the variable values are automatically displayed in the Value column of the panel. If you want to monitor the value of specific variable during execution of the program logic, then you have to add those variables to the Monitors panel. In this section, you will learn how to examining variables on the Monitors panel.

In the sample program, the variable pChar needs to be monitored. Add the pChar variable to the monitor queue by right-clicking any space in the Monitors panel or Source Window and selecting the New Monitor task in the popup menu. Specify the variable (pChar) you want to monitor as shown in Figure 13. The New Monitor menu task is disabled when the program is not stopped on a statement in the source code. By default, the stopped position of the debugger is indicated by a yellow background line as demonstrated in Figure 13.

After the variable is monitored, the continued debug of the program shows that the value of the pChar is converted to an undesired character ("a" to "/) when the program execution reaches line 20 shown in Figure 14.

Figure 13. Set Variable to Monitor on Monitors panel



```
    printf("Pls input the lower case letter to be converted:\
    printf("Input 0 to exit.\n");

        printf("Your input is: %c\n", *pChar);
    }
```

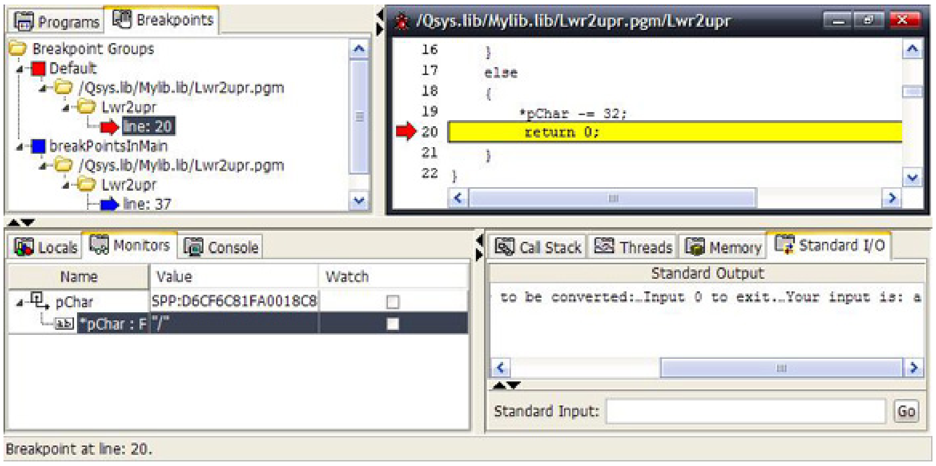
Enter variable or expression to monitor: pChar

Name	Value	Watch

Method	Module	Line	Statem...	Program	Library
main	Lwr2upr	30	1	Lwr2upr	Mylib
C pep	Lwr2upr		0	Lwr2upr	Mylib

On the Monitors panel, there is a column named `Watch` for every monitored variable. If you choose to watch a variable by clicking the `Watch` column of the row, then the program will automatically stop after the statement that caused the value of the watched variable to chang. That is to say, a watched variable will function like a breakpoint when its value changes. Providing that there are breakpoints or watched variables defined, you can simply click the `Resume` button in the toolbar to run the program. This enables execution to continue until the debugger encounters a line where there is a breakpoint set or a watched variable is changed. Besides these options for controlling the program execution, there are several different kinds of step execution options provided on the toolbar.

Figure 14. Monitors Panel before "a" is converted



```
16        }
17        else
18        {
19            *pChar -= 32;
20            return 0;
21        }
22    }
```

Name	Value	Watch
pChar	SPP:D6CF6C81FA0018C8	☐
*pChar : F	"/"	☐

Standard Output

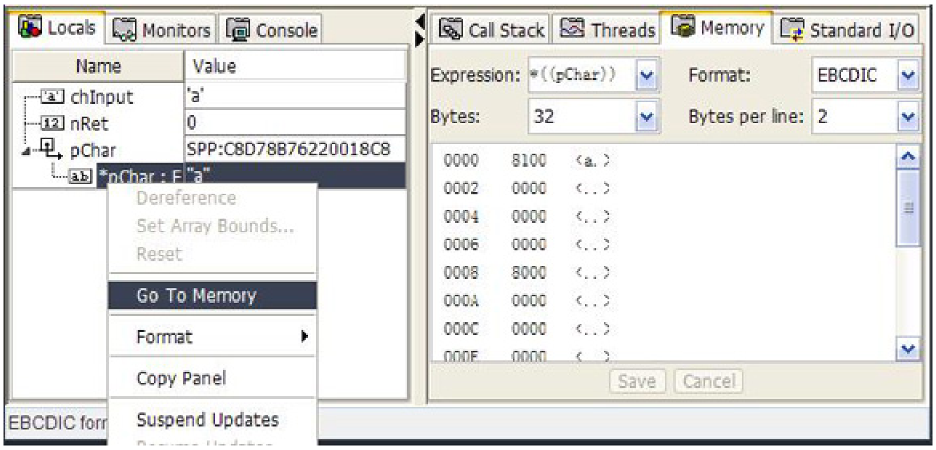
```
to be converted:…Input 0 to exit…Your input is: a
```

Breakpoint at line: 20.

Using the Memory Panel

The `Memory` panel displays the memory of the variables or expressions. To view the memory values associated with a variable or expression, you can just type the variable name or expression into the `Expression` field of the panel. However, the simplest method is to right-click on the desired variable in the `Locals` or `Monitors` panel and select the `Go to Memory` task in the context menu as shown in Figure 15. There are three columns of output in the `Memory` panel. The first column is the memory sequence number in hexadecimal form. The second column contains just the memory's contents in hexadecimal format. The third column, called *character representation area*, displays the characters (if displayable) corresponding to the hexadecimal values in the second column. The characters are enclosed in brackets. You can customize the number of bytes displayed per line and how many bytes of memory are to be displayed by setting the `Bytes per line` and `Bytes` field respectively. Another customization option is the "Size of" setting for the `Bytes` field. This option causes the panel to only display the memory allocated to the specified variable or expression.

Figure 15. Memory Panel before upper-case conversion



Name	Value
chInput	'a'
nRet	0
pChar	SPP:C8D78B76220018C8
*pChar : E	"a"

Expression: *((pChar))
Format: EBCDIC
Bytes: 32
Bytes per line: 2

```
0000    8100    <a. >
0002    0000    <. .>
0004    0000    <. .>
0006    0000    <. .>
0008    8000    <. .>
000A    0000    <. .>
000C    0000    <. .>
000E    0000
```

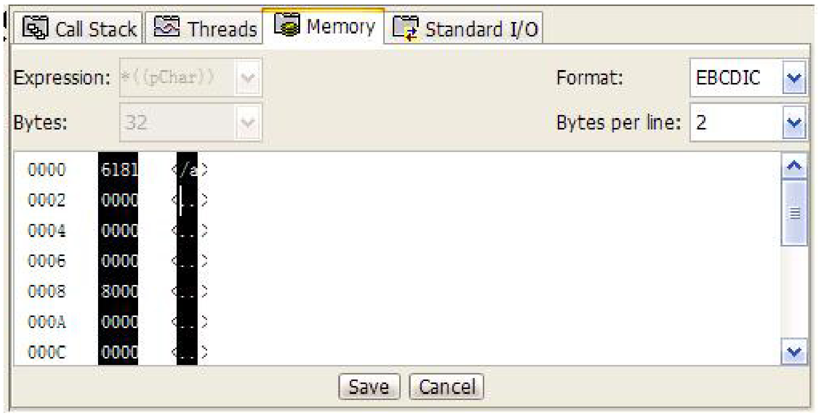
Let's examine the memory for the variable pChar because this variable is associated with the letter being converted. In this example, when lower case letter "a" is input, the corresponding memory value is 0x81 as shown in Figure 15. After the converting, the memory is changed to 0x61 which represents the forward slash character (/) as shown in Figure 16. Why? After checking the ASCII table, the code point for letter "a" code should be 0x61, while that for forward slash "/" should be 0x2F. The memory values for the variable are not consistent with the ASCII code points. You may notice that there is a field naming `Format` in the `Memory` panel. The field is used to control which encoding system is used to display the memory values as character strings. The options are EBCDIC, ASCII, Unicode and UTF32. At this point, the problem is clear. The IBM i operating is an EBCDIC-based operating system, not ASCII-based. The following code expression for converting to the corresponding upper case letter is not correct for EBCDIC code points:

```
*pChar -= 32;
```

After checking the EBCDIC code table, EBCDIC code point for letter 'a' is 0x81, and that for "/" is 0x61, which is consistent with Figure 16. Furthermore, the EBCDIC code point distance from a lower case letter to its corresponding upper case letter is 64, not -32. Thus, the problem within the sample program logic has been identified!

Another point of this panel is that memory contents can be changed at debugging time. To change the content, position the cursor to the place where you want to change the memory value in either the second column or third column. Then, type the new value you want to use and click the `Save` button to change the memory content as shown in Figure 16.

Figure 16. Change the memory's contents



Expression: *((pChar))
Bytes: 32
Format: EBCDIC
Bytes per line: 2

```
0000  6181  </a>
0002  0000  <..>
0004  0000  <..>
0006  0000  <..>
0008  8000  <..>
000A  0000  <..>
000C  0000  <..>
```

Fixing the code

From this point, we know that the problem is caused by a different encoding system: EBCDIC. How to revise the code to make it function correctly on different platforms such as Windows, IBM i or UNIX operating systems? The simplest way is to change the logic in the following line of code in the function convert2Captial:

```
*pChar -= 32;
```

to this updated logic:

```
*pChar -= ('a' – 'A');
```

Besides that problem, there is another problem in the function convert2Captial:

```
if((*pChar < 'a') || (*pChar > 'z'))
```

This statement is used to determine to check if the input letter is legal. It's clear from the EBCDIC code points table that the code points from 'a' to 'z' are not successive. So, the verification code logic needs to be changed to the following:

```
if((*pChar < 'a')
|| ((*pChar > 'i') && (*pChar < 'j'))
|| ((*pChar > 'r') && (*pChar < 's'))
|| (*pChar > 'z'))
```

At this point, the code revisions are complete and the program runs successfully on IBM i.

Other functions of Graphical IBM i Debugger

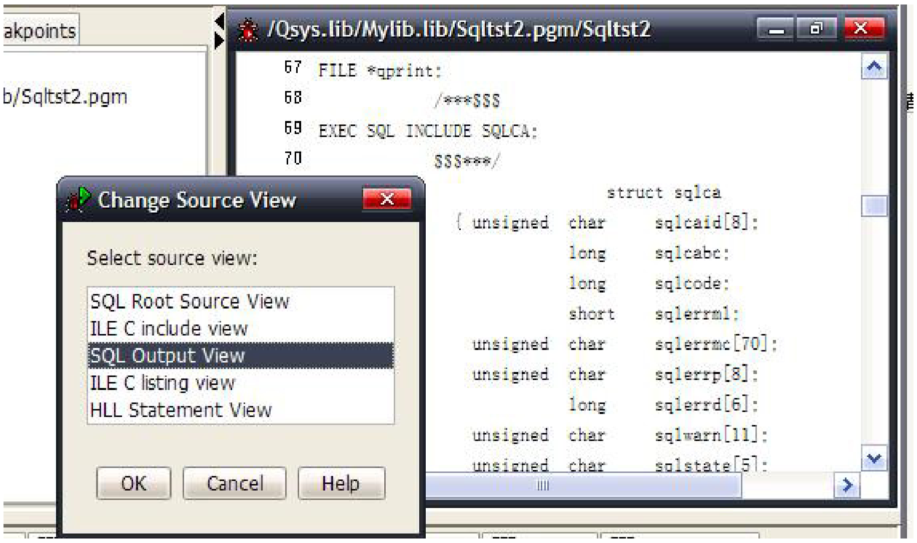
Source view can be changed by right clicking the `Source Window` and select `Change Source View`. The available options in the `Change source view` dialog depend on how you compile the source code. For example, if you create a program with OPTION(*SHOWINC), then `ILE C include view` will be available in the dialog as shown in Figure 17. This view is useful when you want to examine the contents of the included header files. Specifically, when you are developing SQL-embedded programs for IBM i and pre-compile it with parameter DBGVIEW(*SOURCE) and

COMPILEOPT('option(*showinc)'), then available source views for the `Source Window` will be similar as shown in Figure 17. The `SQL Output View` will help to debug SQL procedures, functions and triggers, if they are created or altered with the proper debug option set. For more detailed information on graphically debugging SQL procedures, please refer to the corresponding paper in the Resource section.

As indicated previously, the program that you are debugging can be very complex. The program may consist of many source code files and modules, or it may contain thousands of lines of source code. You many wonder in these cases if you can easily jump to a specific line in large source files or locate the correct source code file? Fortunately, there are several ways to achieve that type of navigation with the graphical IBM i debugger:

- Double-click on the entry in the `Call Stack` panel, which will navigate into the current stop position, as indicated by the *Line* column, in different methods, functions or modules.
- Double-click the breakpoint in the `Breakpoints` panel, which will navigate into the corresponding breakpoints line in `Source Window`.
- Double-click the program object, such as a program, a service program or a Java class file, which will open the corresponding source file in `Source Window`.
- Right-click the entry in the `Monitors` panel, and select `Go To Source`.
- Right-click in the `Source Window` and select `Find…` or `Go To Line…`.

Figure 17. Change source view dialog



```
FILE *qprint;
                /***$$$
EXEC SQL INCLUDE SQLCA;
                $$$***/
                              struct sqlca
                  { unsigned  char      sqlcaid[8];
                              long      sqlcabc;
                              long      sqlcode;
                              short     sqlerrml;
                    unsigned  char      sqlerrmc[70];
                    unsigned  char      sqlerrp[8];
                              long      sqlerrd[6];
                    unsigned  char      sqlwarn[11];
                    unsigned  char      sqlstate[5];
```

The `Threads` panel can be used to debug a multi-threading program. The `Call Stack` panel makes it easy to display the call stack of the current program. The `Console` panel is used to display system messages and to execute IBM i debug commands from this panel. For more detailed information about all of these functions, please refer to the graphical IBM i debugger's documentation.

Summary

In this article, you were introduced to two graphical debuggers for IBM i with a comparison of the debuggers to help developers choose the debugger most suitable for them. You also learned how to install the graphical IBM i debugger using different methods. And most importantly, you learned how to utilize the graphical IBM i debugger for troubleshooting a program.

Resources

Here are some useful resources you can refer to for some detailed information mentioned in the article:

1. IBM i Information Center topic on IBM i Debugger.
2. Download the paper Graphical debugging makes procedural SQL debugging on i5/OS even easier.
3. Read the developerWorks article on IBM Toolbox for Java and JTOpen.
4. Information about introducing, installing and using IBM System i Navigator.
5. IBM Rational Developer for Power Systems Software.