Using the Debugger

by Paul Turley

Debugging programs
Debugging with Aurora is done through a specially compiled version of your program to embed source, line number, and variable name information into the executable. Debug executables are not meant for general distribution and should not be used for any other purpose except finding problems with your code.

Creating a debug executable
Before an application can be debugged it must be built in debug mode.

For single file compiling check the "Debug build" checkbox in the Executable options dialog when creating the executable. The setting will be remembered each time you compile so be sure to uncheck the box when you are finished debugging and build a normal application.

For projects change the project options by selecting the *Project* menu and choosing *Options*, then check the "Debug build" checkbox and save the project. Rebuild you project by selecting the *Build menu* and choosing *Rebuild All*

Starting the debug session
Once a debug executable has been created start the debug session by selecting the *Build menu*, choosing *Debug* and finally choosing *Start*. Alternately you can press F9 or click the debug start/continue button on the toolbar.

The IDE will change to debug view in the output window and attempt to debug the executable. Watch the debug view window for informational messages such as DLL's being loaded, any exceptions, and output from the OutputDebugString API function. If an exception or breakpoint is encountered the debugger will stop the execution of the program being debugged and the IDE will enter debug operations mode.

Stopping the debug session
At any time you can end the program being debugged while it is running by selecting the *Build* menu and choosing *Debug->Stop*. Alternately click the stop button on the toolbar. After the debug session is over the IDE will return to normal mode and you can make any changes to your code, rebuild and debug again if necessary.

Controlling the debugger
The debugger will pause execution of your program any time it encounters a system exception or a breakpoint set with the #break statement. At this time the "Debug Context Display" window will open as well as the source file containing the line where execution was paused. The line will be highlighted for ease of identifying.

If the program was paused due to a #break statement you can continue execution where it left off by Selecting the *Build* menu and choosing *Debug->Continue*, pressing F9 or clicking on the debug start/continue button in the toolbar.

A program that generates a system exception such as an *access violation* cannot be continued and must be restarted. The source file and line number will be shown to allow determining where the fault occurred.

Single stepping
After a programs execution has been paused with a #break statement you can single step one line at a time by selecting the *Build* menu and choosing *Debug->Single Step*, pressing the F10 key or clicking on the single step toolbar button. The debugger will try and find the next source line in the executable and if it exists in your executable the line will be executed. You cannot single step past the end of the main function or into DLL code.

The context display
As mentioned above whenever an error occurs or your program is paused by the #break statement or exception the Debug Context Display window will open. This window shows the call stack, register contents, local variables and a disassembly listing. The disassembly listing shows the previously executed line and the line about to be executed.

The call stack shows the subroutines execution has traversed to get to the current point of execution. For example if SubroutineA calls SubroutineB and a #break statement is located in SubroutineB the call stack might look like:

Aurora1! SubroutineB + 13 File: C:\Aurora programs\Aurora1.src, Line: 43
Aurora1! SubroutineA + 111 File: C:\Aurora programs\Aurora1.src, Line: 27
kernel32! CreateProcessInternalW + 4471

Each line in the call stack provides the information of:
executable name! Subroutine name + offset in bytes File: Source file name Line: line number in file

The first line in the call stack is always the last point of execution before the exception or #break. Source file and line number information will only be available for programs and projects compiled in debug mode. They will not usually be available for DLL's or other system code as indicated above. They are also not available in the linker libraries used by the compiler itself.

The local variable display shows the contents of variabls within the subroutine where the #break statement was encountered.

Outputting data to the debug view
While your program is running under a debugging session you can display information in the debug view of the IDE by using the OutpurDebugStringA API function. OutpurDebugStringA accepts one string parameter which can be used to display the contents of variables while the program is running, or to display state information to aid in following the execution path. To use the OutputDebugString function declare it in your source:

Code:
```
declare import,OutputDebugStringA(string *str);
sub main()
{
....
OutputDebugStringA("Program starting");
OutputDebugStringA("A=" + NumToStr(A));
error = MyFunction(27);
OutputDebugStringA("MyFunction returned: " + NumToStr(error));
...
}
```

Preparing your source for debugging
When building in debug mode the compiler automatically defines the preprocessor symbol DEBUG. You can use this symbol to create conditional compiling based on regular or debug mode. Using conditional compiling ensures that no debugging references are left in your code for a normal build of your executable. Using the example from above:

```
#ifdef DEBUG
OutputDebugStringA("Program starting");
OutputDebugStringA("A=" + NumToStr(A));
#endif
error = MyFunction(27);
#ifdef DEBUG
OutputDebugStringA("MyFunction returned: " + NumToStr(error));
#endif
```