# Table of Contents

1. About the author ....................................................... 1
2. Abbreviations .......................................................... 2
3. Terms ......................................................................... 3

**MIGRATION GUIDE**

4. Introduction .................................................................. 4
5. Prerequisites ............................................................. 5

6. Creating your iLLD demo codebase ................................. 6
   6.1. Merge demo, drivers and template .............................. 7
   6.2. Integrate simulated IO ........................................... 7

7. Establishing a baseline .................................................. 11

8. Migration ....................................................................... 12
   8.1. Installing the baseline ........................................... 18
   8.2. Migrating the baseline option sets ......................... 23
      8.2.1. Compiler options .......................................... 23
      8.2.2. Assembler options ........................................ 27
      8.2.3. Linker options ............................................. 27
   8.3. Runtime testing .................................................... 29

9. Conclusion ..................................................................... 32

**APPENDIXES** ............................................................. 33

Appendix A: Bibliography .................................................. 34
Appendix B: Document history .......................................... 35
1. About the author

Henk-Piet Glas (Principal Technical Specialist, Embedded Software), received his Beng in Information Technology from NHL college, Leeuwarden. With over 20 years of experience he has worked in both the embedded and data warehousing industry. His activities include dedicated FAE support, selected FAE trainings and webinars, application note development, and technical contributions to fora.

Henk-Piet enjoys the creativity of independent coursework. He’s experimented with screenwriting, video editing and creative documentary. For a while he aspired theatre making and during a brief stint in 2001, he toured the UK and Ireland with a seriously funny theatre company called Ridiculusmus. Recent creative work includes a handful of columns and several short stories.
2. Abbreviations

API
Application Programming Interface

BIFACES
Build and Integration Framework for Automotive Controller Embedded Software.

BSP
Board Support Package

DAS
Device Access Server

IDE
Integrated Development Environment

IFX
Infineon

iLLD
Infineon Low Level Drivers

IO
Input/Output

SFR
Special Function Register

UDE
Universal Debug Engine
3. Terms

AURIX™
The next generation of Infineon’s TriCore™ 32-bit microcontroller architecture, featuring a multicore implementation. AURIX™ combines easy-to-use functional safety support, a strong increase in performance and a future-proven security solution in a highly scalable product family.

BIFACES
An application development environment that provides a unified platform for AURIX software development and upcoming microcontrollers. Target users include application engineers and customers who use the application examples or demo software drivers for prototyping within the automotive domain.

Device Access Server
An abstraction layer between any third party debugger and the target system, developed by Infineon. It includes several debug utilities that can be very useful when dealing with third party debugger connection issues.

Free TriCore Entry Toolchain
Restricted version of the TriCore Development Platform, supporting a restricted selection of AURIX derivatives. Following a single year duration its license automatically expires.

HighTec
Since its establishment in 1982, HighTec has been a privately owned company and the world’s largest commercial open source compiler vendor.

Infineon
Leading innovator in the international semiconductor industry. Infineon designs, develops, manufactures and markets a broad range of semiconductors and complete system solutions for selected industries.

Infineon Low Level Drivers
Developed by Infineon, its aim is to provide access and configuration functions for the integrated peripherals of Infineon Microcontrollers. Together with SFR header files they are a fundamental part of the infrastructure for tests and applications which are developed by several IFX teams.

MyICP
An online information portal hosted by Infineon that provides access to additional content, services, and customised information. Users must register to make use of this service.

PLS
One of the world’s leading manufacturers of development tools for 16-bit and 32-bit microcontroller families.

Universal Debug Engine
Debugging solution by PLS featuring debug support for a wide range of 16-bit and 32-bit microcontrollers including AURIX.
4. Introduction

BIFACES was developed by Infineon with the intent of allowing application engineers and automotive users to develop application notes and demonstration software.

BIFACES iLLD demo projects consist of non-managed makefiles. That is, a set of generated makefiles, but without control via Eclipse’s compiler, assembler and linkers options. The aim of this application note therefore is to transform these into managed HighTec projects, thereby regaining control.

In the first stage the iLLD demo codebase is imported as a project derived from makefile. We will test its codebase to make sure it actually works. When it does, it can then subsequently be used as a baseline. We then create a regular HighTec project and start migrating the baseline, effectively using it as a repository for code, generated content, linker script and tool settings.

The migration completes with a rudimentary comparison of the map file and runtime results to those collected when we tested the baseline. This application note takes about 1 hour to complete. This can be reduced to less than 10 minutes as you become more practised.
5. Prerequisites

For this application note you will need to meet the below requirements.

<table>
<thead>
<tr>
<th>Component</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>free_tricore_entry_tool_chain.zip</td>
<td>v4.9.1.0-Infineon-2.0</td>
</tr>
<tr>
<td>BIFACES_V1_0_2_Win32.zip</td>
<td>v1.0.2</td>
</tr>
<tr>
<td>BaseProjects_AURIX1G_V1_0_1_10_0.zip</td>
<td>v1.0.1.10.0</td>
</tr>
<tr>
<td>iLLD_1_0_1_10_0_TC2xx_Drivers_And_Demos_Release.zip</td>
<td>v1.0.1.10.0</td>
</tr>
</tbody>
</table>

For runtime testing we used the TC277 TFT Application Kit (fitted with D-step). The Free Entry TriCore Toolchain includes UDE Starterkit 4.10. We will use its Eclipse plugin and therefore mostly refer to it as the UDE perspective, or simply UDE. We connect to the onboard wiggler by means of a USB cable.

Using the onboard wiggler is an imposed restriction of the UDE starterkit. External wigglers only work with the professional version of the Universal Debug Engine.

The professional tools do not include a debugger. That voids the debugging chapters in that you must use your own professional debugger instead.

This document replaces the previous version based on the Infineon Software Framework. Meanwhile this framework has been replaced with BIFACES. Since some of its migration steps no longer apply, we renewed this document accordingly.
6. Creating your iLLD demo codebase

As a starting point for this application note you will need to decide which iLLD demo you want to migrate. Since these demos come without makefiles and linker scripts, they will first have to be wrapped into a matching base framework template before you can build them. Also, since the base framework project contains an iLLD subset, generally you will have to replace it with its entire set of drivers. And in addition you must add simulated IO support to allow UDE to redirect printf statements to its integrated terminal. The end result is what we will refer to as the iLLD demo codebase.

6.1. Merge demo, drivers and template

For this application note we used the iLLD SCU Clock Demo. We will therefore start with the following three software components:

1. AURIX iLLD ScuClockDemo
2. AURIX iLLD Base Framework template for TC277 D-step
3. AURIX iLLD drivers for TC277 D-step

And wrap them into one. These three can be obtained from the packages listed in chapter Prerequisites. The foundation of any iLLD demo is the base framework template of your chosen derivative.

![Figure 1. The base framework template for TC277 D-step](image-url)
Its subfolder 0_Src contains the functional part of the template (a skeleton main for each core) and a subset of the iLLD drivers, contained in folders AppSw and BaseSw respectively.

![Image: The default base framework sample codebase](image1.png)

**Figure 2. The default base framework sample codebase**

Both of these folders may be purged, because we will replace them with our intended iLLD demo and the full set of iLLD drivers.

![Image: Purge the default base framework sample codebase](image2.png)

**Figure 3. Purge the default base framework sample codebase**
Following the purge, drill down into the iLLD driver archive and pull its entire codebase.

![Figure 4. Copy the full iLLD codebase](image)

Subsequently paste the pulled content into your iLLD demo.

![Figure 5. Install iLLD codebase](image)

You also need to replace the functional part by pulling the ScuClockDemo codebase from the iLLD sample archive.

![Figure 6. Pulling the SCU clock demo codebase](image)
And paste it side-along the iLLD driver codebase.

![Image](image1.png)

**Figure 7. Install SCU clock demo codebase**

Your initial base framework template is now the foundation of the iLLD demo of your choice.

![Image](image2.png)

**Figure 8. Finished integration of template, iLLD demo and iLLD drivers**
The only thing left is to chose a more generic name.

![Image of a file explorer window with a folder named 'Demo' and other files]

**Figure 9. Renaming the base framework project**

### 6.2. Integrate simulated IO

For debugging we will use UDE’s integrated simulated IO terminal. This allows C library functions like `printf` and `scanf` to interact with the user without having to extend your program with a native IO channel. In order for this to work, low level functions `read` and `write` must be overloaded to make use of the PLS API. The Free Entry TriCore Toolchain includes source code for this by means of `simio_pls_tc.c` and `simio_pls.h`, which can be found in the BSP folder. Simply add them to your project as demonstrated in **Figure 10**.

![Image of a file explorer window with source files located in 'BSP' folder]

**Figure 10. Integrate PLS simulated IO support**
7. Establishing a baseline

We can now create a baseline. That is, a pre-build of the untarnished iLLD demo that will be used as a reference once the actual migration is complete. Start your HighTec IDE and proceed to import your iLLD demo codebase using the following steps:

1. Select File → New → Project...
2. Select C/C++ → Makefile project with Existing Code and press Next
3. Select Browse choose your demo and press OK

Since the iLLD demo codebase makes use of the BIFACES framework, we need to add two project environment variables making sure that it will build. Start by adding BINUTILS_PATH using these steps:

1. Right-click the Demo project and select Properties
2. Navigate to C/C++ Build → Environment
3. Click Add… and create BINUTILS_PATH

Also see Figure 11. Note that you must substitute its path with that of your own BIFACES installation.

![Figure 11. Creating BINUTILS_PATH project environment variable](image)

Note that project environment variables have a transient nature. They only exist for the duration of your build. Proceed to create a PATH variable that is derived from BINUTILS_PATH.

![Figure 12. Derive PATH from BINUTILS_PATH environment variable](image)
During a build this variable will also be temporarily pushed into your native OS environment, however since it already exists, it will be appended to the native one. Figure 13 shows the two variables we just created. Note that PATH has meanwhile been replaced with the one from the OS. Variable ${BINUTILS_PATH} has been appended to its rear end. Since typically PATH is relative long, this is not visible in the snapshot.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINUTILS_PATH</td>
<td>C:\FX\BifacesWin32\bin</td>
<td>USER.Config</td>
</tr>
<tr>
<td>CWD</td>
<td>C:\Users\glas\work\EthDemo\</td>
<td>BUILD SYSTEM</td>
</tr>
<tr>
<td>PATH</td>
<td>C:\Program Files (x86)\Java\jre1.8.0_161\bin\client...</td>
<td>USER.Config</td>
</tr>
<tr>
<td>PWD</td>
<td>C:\Users\glas\work\EthDemo\</td>
<td>BUILD SYSTEM</td>
</tr>
</tbody>
</table>

Figure 13. Project variable overview

Use project variables over Windows environment variables

Avoid adding the aforementioned project environment variables to your Windows environment. They are only required whilst establishing the baseline. More importantly, some utilities installed by BIFACES are also installed by HighTec. These have been known to conflict if they occur on the same PATH. Regular HighTec projects will then break during builds.
Following the import you must make one last change which involves retouching the `B_GNUC_TRICORE_PATH` makefile variable to point to the installation of the Free Entry TriCore Toolchain. Using the HighTec Project Explorer, navigate to Demo → 1_ToolEnv → 0_Build → 1_Config → Config_Tricore_Gnuc and double-click Config_Gnuc.mk. Then adjust it. You see this depicted in the next snapshot, which also shows the relative location within your baseline project.

![Figure 14. Retouching B_GNUC_TRICORE_PATH makefile variable](image)

When this is done you can proceed to build your project as depicted in Figure 15.

![Figure 15. Building the baseline](image)

Your ELF program image can now be flashed onto the target. For this you must create a UDE configuration. The way this works is by means of the following steps:

1. Goto the HighTec Project Explorer
2. Right-click your Demo project
3. select Debug As → Debug Configurations...
4. Click Universal Debug Engine icon
5. Click New launch configuration button
To your right there now is a **Main** panel. In here you must enter the relative location of your ELF application. Figure 16 shows how.

![Debug Configurations](image)

**Figure 16. Adding ELF application to UDE build configuration**

Switch to the **UDE Startup** panel and create a target configuration file using the following steps:

1. Click the **Create Configuration** button
2. Tick the **Use a default target configuration** radiobutton
3. Drill down to **Application Kit with TC277 D-Step (Multicore Configuration)**
4. Press **Finish** choose location and press **Save**

**Figure 17** shows that a default configuration AppKit_TC277D.cfg has been created.

![Debug Configurations](image)

**Figure 17. Default AppKit_TC277D.cfg configuration**
Press the **Debug** button. The UDE perspective will be opened, and it will attempt to connect to your target. Upon success you are presented with its flash utility dialog. Press **Program All** and wait for it to complete. Then press **Verify All** to make sure things were done right. Then press **Exit**. Figure 18 shows the resulting UDE perspective.

![Figure 18. UDE perspective after flashing the baseline](image)

Select **Views ➔ Simulated I/O**. This adds the **Simulated I/O** view to the UDE perspective.

![Figure 19. Adding Simulated I/O view to UDE perspective](image)
Set the following application breakpoint.

![Image of application breakpoint](image)

**Figure 20. Application breakpoint**

Run the application using **Debug → Start Program Execution**. **Figure 21** lists the expected output of the **Simulated I/O** view.

![Image of simulated I/O output](image)

**Figure 21. Simulated I/O output when hitting breakpoint**

We have now confirmed the baseline application. We’ll proceed to use it as both a reference and a repository throughout the next chapter.
8. Migration

Everything is now in place to start with the actual migration. First thing we'll need to be doing is to create a managed HighTec project and install the baseline codebase. We also need to configure some initial project settings such as the iLLD header files search list. We subsequently migrate the compiler, assembler and linker makefile rules and then lastly build and verify the migrated project. We'll do this by means of a rudimentary comparison of its mapfile against the baseline mapfile. If that looks promising we test the runtime results as before.

8.1. Installing the baseline

First create a managed HighTec Project.

1. Select File → New → HighTec Project
2. For Project name enter Managed and press Next
3. Select Application Kit TC277 D-Step from the hardware selection tree
4. Tick the Create empty project checkbox and press Finish

You now have an empty HighTec project that needs to be populated and configured. Its default configuration name is called iRAM. Switch to configuration Default and rename it to debug.

1. Select Project → Properties
2. Navigate to C/C++ Build → Settings
3. Press Manage Configurations... button in top-right corner
4. Click the Default configuration and press Set Active
5. Click Rename... and enter debug

Figure 22 shows the resulting dialog. Press OK to exit.
Your active project configuration is updated as depicted in Figure 23.

Figure 23. Confirming renamed configuration

Navigate to Demo → 0_Src and copy AppSw and BaseSw. Navigate to Managed → src and paste them.

Figure 24. Copy baseline codebase

Figure 25. Paste baseline into iLLD demo

Figure 26. iLLD baseline installed
Navigate to Demo → 1_ToolEnv → 0_Build → 1_Config and copy the Lcf_Gunc_Tricore_tc.lsl linker script. Paste it into the top node of Managed.

![Image of HighTec Project Explorer showing Copy option for Lcf_Gunc_Tricore_tc.lsl]

**Figure 27. Copy Lcf_Gunc_Tricore_tc.lsl linker script**

Navigate to Demo → 1_ToolEnv → 0_Build → 9_Make and copy the Tricore_IncludePathList.opt iLLD header files search list.

![Image of HighTec Project Explorer showing Copy option for Tricore_IncludePathList.opt]

**Figure 29. Copy iLLD header files search list**
Paste it into the top node of Managed.

![HighTec Project Explorer](image)

**Figure 30. iLLD header files search list pasted into iLLD demo**

**The implications of skipping the baseline**

The iLLD header files search list is a generated file from BIFACES. If it is missing, it most likely means that you skipped the baseline chapter.
Rename the aforementioned filenames to something more pleasing. Figure 31 shows how.

![HighTec Project Explorer](image1)

**Figure 31. Renamed linker script and iLLD header files search list**

Open Managed.i and replace 0_Src with ../src. Then add it to the compiler Miscellaneous options menu. Note the @ prefix which instructs the compiler to use it as an option file. Also note the ${ProjDirPath} to align with the base folder.

![HighTec Project Explorer](image2)

**Figure 32. Retouching the iLLD header files search list**

![HighTec Project Explorer](image3)

**Figure 33. Add search list as a compiler option file**

The iLLD header files search list consists of paths that are relative to the working folder of your current configuration. It (demo) is created during your first build.
Lastly add the Managed.x linker script to the General options of your linker. Also note the 
${ProjDirPath}$ to align it to the base project.

![Figure 34. Retouching the iLLD header files search list](image)

### 8.2. Migrating the baseline option sets

Migrating baseline makefile option sets is a step-by-step process of translating individual baseline command line options to their equivalent HighTec IDE project settings. For this we return to module Config_Gnuc.mk which contains the makefile variables as depicted in Figure 35. The next three chapters will migrate them one by one.

![Figure 35. Baseline makefile rules](image)

#### 8.2.1. Compiler options

B_GNUC_TRICORE_CC_OPTIONS contains the compiler option set. Start by mapping -mtc161. This option defines the architecture instruction set that must be used. Since the core instruction set version is derived from the CPU project settings there is no need to migrate it because it is already in place. To review the current CPU type goto Project \ Properties and select C/C++ Build \ Settings \ Global Options.
Figure 36. Architecture (-mtc161) derived from CPU type

Option -g enables compiler debug information. It is mapped as follows:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Debugging**
3. From the **Debug Level** dropdown box select **Default (-g)**

Figure 37. Enable compiler debug information (-g)

Option -O2 is an optimisation group that optimises for speed. These are the steps to enable it:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Optimization**
3. From the **Optimization Level** dropdown box select **Speed**

Figure 38. Optimise for speed (-O2)

The meaning of -fno-common is that uninitialised data is stored in a .bss section. Each optimisation group sets it by default. In other words, no need to do anything.
Next in line is `-fstrict-volatile-bitfields`, an option that assures forced read/write accesses to individual bitfields. This option doesn't have a dedicated IDE setting and must be added as a miscellaneous option instead.

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Miscellaneous**
3. Add `-fstrict-volatile-bitfields`

![Figure 39. Adding options without dedicated IDE settings](image)

Options `-fdata-sections` and `-ffunction-settings` can be mapped in one go. These suffix a variable name and/or function name to the default section name which greatly benefits garbage collection during linking. They can be set using the following steps:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Code Generation**
3. Tick **Generate a section for each data object (-fdata-sections)** checkbox
4. Tick **Generate a section for each function (-ffunction-settings)** checkbox

![Figure 40. Suffix default section names](image)
Next one up is -Wall, which is mapped as follows:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Warnings**
3. Untick all checkboxes
4. Tick **All Warnings (-Wall)** checkbox

![Figure 41. Check for most warnings (-Wall)](image)

Option -std=c99 defines the 1999 ISO coding standard that should be checked against. It is our last compiler option, and it is mapped as follows:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Compiler → Language Dependent**
3. From the **C standard (-std)** dropdown box select **C 99**

![Figure 42. Coding for C 99 (-std=c99)](image)
8.2.2. Assembler options

BIFACES uses the steering program (tricore-gcc) to execute the compiler, assembler and linker rules. The steering program itself is in charge of sifting through their associated option sets and deciding which flags apply to itself, which ones need to be translated, redirected or sometimes even dropped. Knowing this, provides some insight as to why Figure 35 creates a carbon copy of B_GNUC_TRICORE_CC_OPTIONS when declaring the assembler option set B_GNUC_TRICORE_ASM_OPTIONS; it anticipates translation. The trick is to find out what effective assembler option set it is translated to. Figure 43 shows how you can do this from the command line. The commands it lists were run from an ssh session connected to CentOS.

![Figure 43. Extracting the assembler option set](image)

What this tells us is that most options have actually been dropped. Only -mtc161 is passed and option -g is translated to -gdwarf2. Neither of these needs to be migrated because these are already the defaults. In the same way as shown in Figure 36 the assembler architecture is derived from the chosen project CPU type.

The takeaway of the previous is that almost always you can stick to the default assembler project options.

8.2.3. Linker options

The linker command line options are assigned to variable B_GNUC_TRICORE_LD_OPTIONS starting with -mtc161. For the same reason as mentioned for compiler command line option -mtc161 there is no need to map it. We can therefore quickly move on to -Wl,--gc-sections. This option actually consists of two parts. The first (-Wl) tells the steering program that the next option must be redirected to the linker. This is sometimes necessary if the steering program doesn’t know how to handle a specific option.
The second part (\texttt{--gc-sections}) instructs the linker to remove unreferenced sections, the very reason why we generated such fine-grained sections in Figure 40. Any default project has this option already enabled, but these are the steps in case you wish to make sure:

1. Select \textbf{Project \rightarrow Properties}
2. Select \textbf{C/C++ Build \rightarrow Settings \rightarrow TriCore C Linker \rightarrow Optimization}
3. Tick \textit{Remove unreferenced sections (\texttt{-Wl,--gc-sections})} checkbox

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure44.png}
\caption{Remove unreferenced sections (\texttt{--gc-sections})}
\end{figure}

Next, option \texttt{-nostartfiles} is migrated using the following steps:

1. Select \textbf{Project \rightarrow Properties}
2. Select \textbf{C/C++ Build \rightarrow Settings \rightarrow TriCore C Linker \rightarrow General}
3. Tick \textit{Skip standard system startup files when linking (\texttt{-nostartfiles})} checkbox

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure45.png}
\caption{Dropping default startup code and constructor initialisation}
\end{figure}

\textbf{BIFACES, C++, and the constructor attribute}

The previous drops both the default startup code and the constructor/destructor initialisation sections. At the point of writing this document BIFACES doesn’t support C++ demos, so that makes perfect sense. However note that regular C functions declared with the constructor attribute will therefore also cease to work.
The last option is `-Wl,-n` so essentially the `-n` linker command line option. This option switches off the alignment of the program headers within an ELF archive. As a result they become physically smaller without effecting the program image itself. As with the the garbage collection command line option it is already the default for any project. To review it, use the following steps:

1. Select **Project → Properties**
2. Select **C/C++ Build → Settings → TriCore C Linker → Miscellaneous**

![figure](image)

*Figure 46. Creating physically smaller ELF files (-Wl,-n)*

We have now completed the migration of all project options. Proceed building the project as before.

### 8.3. Runtime testing

Following the build, a folder `demo` has been created in the root of project `Managed`. Amongst others it contains the project map file and the application ELF file. A successful build alone is no guarantee that the application will work as designed. Let’s do the following quick test:

1. Goto the **HighTec Project Explorer**
2. Double-click **Demo → 2_Out → Gnuc → 0_Src → EthDemo_tc.map**
3. Search for **Memory Configuration** block
4. Remove all rows with **Used** column set to `0x00000000`
5. Double-click **Managed → pfsh0 → Managed.map**
6. Search for **Memory Configuration** block
7. Remove all rows with **Used** column set to `0x00000000`
8. Rearrange both map files to horizontal side-by-side view
Figure 47 lists the edited mapfiles.

![Mapfiles Comparison](image)

**Figure 47. Rudimentary comparison of retouched mapfiles**

Note that in the framework demo project memory demo consumes 0x801F73B8 bytes of memory. The consumption for the migrated project is identical. While this is not a binary comparison it is promising nonetheless.

Let’s proceed to take a look at the runtime results.

1. Go to the HighTec Project Explorer
2. RMB Managed
3. Select **Debug As → Debug Configurations...**
4. Click **Universal Debug Engine** icon
5. Click **New launch configuration** button

The **Universal Debug Engine** icon will spawn a child that inherits its name from our project. Proceed as follows:

1. Click **UDE Startup** tab
2. Click **Create Configuration** button
3. Tick **Use a default target configuration** radiobutton
4. Drill down to **Application Kit with TC277T D-Step (Multicore Configuration)**
5. Press **Finish** choose location and press **Save**
6. Press **Debug**

The UDE perspective will now be launched and the debugger tries to connect to the target. On success you may proceed to flash the application as before.
Next place a breakpoint and run the target by means of **Debug → Start Program Execution**. This will produce identical output as before.

![Simulated I/O results](image)

**Figure 48. Runtime simulated IO results**

There you have it. Our original baseline has been successfully migrated to a HighTec IDE equivalent.
9. Conclusion

In this application note we showed how to migrate BIFACES demonstration software to managed HighTec projects. We showed how to do this with the least amount of effort using a non-managed baseline project for comparison and repository. While this application note by no means reflects all possible paths that might unfold during a routine migration, we hope nonetheless that it will serve as a stronghold when such circumstances occur.
Appendixes
Appendix A: Bibliography

Appendix B: Document history

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes to the previous version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>6-2017</td>
<td>starter template</td>
</tr>
<tr>
<td>2.0</td>
<td>7-2017</td>
<td>initial draft</td>
</tr>
<tr>
<td>2.2</td>
<td>8-2017</td>
<td>feedback rounds #1 and #2</td>
</tr>
<tr>
<td>2.3</td>
<td>9-2017</td>
<td>featuring bi-target document compilation.</td>
</tr>
<tr>
<td>3.0</td>
<td>5-2019</td>
<td>update to most recent BIFACES</td>
</tr>
</tbody>
</table>