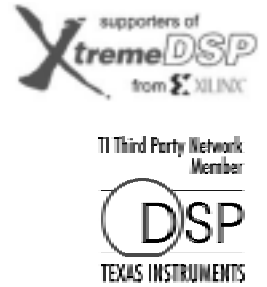




**HUNT ENGINEERING**  
Chestnut Court, Burton Row,  
Brent Knoll, Somerset, TA9 4BP, UK  
Tel: (+44) (0)1278 760188,  
Fax: (+44) (0)1278 760199,  
Email: sales@hunteng.co.uk  
<http://www.hunteng.co.uk>  
<http://www.hunt-dsp.com>



## The “sl\_api (threads)” Server/Loader example.

Rev 1.0 JT 15/11/03 (new for SL 4.08)

This example will work with HEART boards, such as the HEPC9, with 1 or more DSP modules. The “sl\_api\threads” program explains how a user program using the Server/Loader Library could be used to boot and serve a network of DSP processors, while using a separate thread to communicate with the same (or another) node. That is, the Server/Loader Library (thread) is serving ‘stdio’ requests from a DSP processor, while a thread also accesses the board using the host API.

There are 2 general approaches:

1. Use the Server/Loader to reset, load and serve the network. Then start a second program (a user program), which opens the API and communicates with one of the programs loaded onto the DSPs by the Server/Loader. This user program must NOT reset the board! This method is used in the sample in sub-directory "parallel". You can run it using the "host.bat" batch-file.
2. Use the Server/Loader Library to reset, load and serve the network of DSPs, then use the API to communicate with the loaded program in a separate thread. In this case, both Server/Loader and API are used within one executable file. This method is used in the sample in sub-directory "threads". You can run it using the "host.bat" batch-file.

The DSP code in the “sl\_api” example uses HERON-API to control the Config line and the Digital Outputs. It then uses HERON-API to send a message to the host program over the HERON FIFO.

The use of HERON-API means that the example is easily changed to use any HERON C6000 module. HERON-API uses DSP/BIOS internally so must be built using Code Composer Studio.

This document describes how to make the project and build the DSP application.

### History

Example revision 1.0 made for Server/Loader V4.08

## **Example software**

The example that we supply consists of two parts. There's C code that makes up a DSP example program. This DSP code consists of just one C file called "test.c". It needs to be built using Code Composer Studio and uses the HERON-API software that has been installed on your PC when you did the "install drivers and tools" from your CD.

The other part is C++ code that makes up a PC host example program. A PC host example executable is provided, so that you can test that you have a working DSP program, before you start creating your own PC host example program. The PC C++ code consists of one C++ file called "host.cpp".

There is another example that shows how to combine Server/Loader and host API. This is located in directory "..\..\sl\_api\parallel". Both the "sl\_api\parallel" and "sl\_api\threads" examples show how to use the Server/Loader to reset, load and serve a system of processors and FPGAs. After booting, a custom host PC program or thread communicates with a node in the system as well, to implement custom I/O between one or more processors and the host PC. In the case of the "parallel" example, the standard Server/Loader is used to reset and load the system. A separate PC host program is then started that communicates with (DSPs on) the system. This PC host program is what makes up the host PC example. In the case of this example ("threads"), the Server/Loader is integrated in the host PC program; you thus only need to start 1 program, which then uses the Server/Loader Library to boot the system. It then continues, using the API for communications with (DSPs on) the system.

## **Hardware setup**

This example will only work with HEART boards, such as the HEPC9, with one or more HERON processor modules, one of them in slot 1 (not necessarily so, but the network file assumes a module in slot 1; you can run the example with a module in a different slot, in that case change the network file). The example shows the communication between one HERON module and the host. One connection between host and module is used for Server 'stdio' communications, and another connection is used for user-program host-module communications.

With a HEART based board, such as the HEPC9, those (duplex) connections must be created using HEART. This can be done with the Server/Loader, by adding some HEART statements in the network file. A template "network" file is included with this example, but when your hardware is different from the assumed configuration in the network file please edit the network file.

If you are running the demo on a different hardware configuration, you will need to change the #defines in the DSP source code to reflect the connections that you have, and also the network file that describes the connections to the Server/Loader.

## **DSP/BIOS**

DSP/BIOS is the multi-threading environment provided as part of the Code Composer development Environment. It also provides services for configuring processor features such as hardware interrupts and timers.

As DSP/BIOS is included in Code Composer Studio, along with the Compile tools for the C6000, all users of HERON hardware will be able to use it.

This example is configured and built using Code Composer and DSP/BIOS.

## **HERON API**

HERON\_API is the hardware independence layer that we provide to access HERON FIFOs and other features of the HERON modules. It allows the DMA engines of the processor to be used when transferring to and from the FIFOs without knowledge of the FIFO hardware, or the DMA engines.

## **Starting**

We assume that a user of this example has previously installed Code Composer and followed the confidence checks. They should also be familiar with using Code Composer.

## **Configuring the example**

You need to create both the DSP example program and the PC host example program. To help you, we have included a working PC host example program, so that you can test your DSP program. The PC host example program is located in the “win32” sub-directory of the “sl\_api\threads” example directory. It is advised that you first create the DSP program. You can then check if the DSP program you created works by using “my32.bat” (uses the provided host PC program).

## **How to create the DSP program**

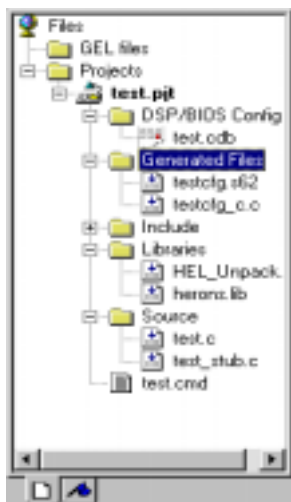
HUNT ENGINEERING provide several Code Composer Plug-in tools that allow you to make your development faster. The first is one that sets up Code Composer ready for your hardware, so you don't need to configure device drivers etc and can be found from the Start → Programs → HUNT ENGINEERING → AutoConfigure CCS. We assume that this is already set up, but the plug-in also copies cdb files etc into the correct locations.

When you start with the “threads” example, copy the source files from the CD into a new directory.

To create a new CCS project, use the “Create new HERON-API project” plug-in. How to use this plug-in is explained in detail in the C6x “Starting Development” example. This is available via the HUNT ENGINEERING CD: Getting Started → to start using C6000 modules and tools look here → Starting your development. This will open the “starting\_development.pdf” document. Read this first – if you have not already done so.

To create the example CCS project:

- Start Code Composer Studio
- Choose Tools→HUNT ENGINEERING→Create new Heron-API project.
- Use the “Browse” button to browse to your “sl\_api\threads” example directory.
- Change “Files of type” to “Project Name (use \*.c, \*.cpp as template).
- Select the “test.c” file.
- Click the “Open” button.
- Verify that in the “Project Name” edit box you have the correct path and name, e.g. something like “c:\examples\c6x\sl\_api\threads\test”. The “c:\examples\c6x” part may differ in your case, depending on where you stored the “sl\_api\threads” example.
- Create the project by clicking on “Create New Project”.
- At the “HERON Module Select” window select the HERON module you create the project for.
- The plug-in will ask whether the project should be created for the Server/Loader. Answer “Yes”.



The plug-in creates and includes a file called “test\_stub.c”. This file ‘links’ the Server/Loader DSP library with the HERON-API library. (This allows the Server/Loader DSP library to be independent of the HERON-API library). The CDB files as delivered with the HERON-API installation have a task ‘TSK0’ with entry point ‘\_maintask’. Therefore, the project is ready to be compiled. You can now build the demo by choosing Project → Rebuild all. There should be no errors.

Finally, note that files that have the same name as the project are automatically added to the project, e.g. if the project is called “test.prj”, then if a file “test.c” exists in the same directory, the plug-in will automatically add it to the project. If no such C file exists, the plug-in will generate a template C file with the name “test.c”.

## Loading and Running the Example

The DSP JTAG chain must be released before you can successfully run the demo. If you are still running Code Composer Studio the processors will be halted. Do a Debug → Run Free (on all processors). If you have already exited Code Composer Studio, but have not done a “Run Free”, then run the API JTAG reset function. This can be done as follows: Start → Programs → HUNT ENGINEERING → API JTAG reset.

The example can be run with the “ex.bat” batch file. The plug-in will also have created a “w32.bat” batch file. However, you cannot use the “w32.bat” batch file with this example. Here’s what the “ex.bat” batch file looks like:

```
win32\ex1 %1
```

The batch file can accept extra arguments implemented by “ex1.exe”. Currently that is the verbose option only (“-v”).

```

Segment of 0x0200 bytes, loading to address 03000000 (.hwi_vec).
Segment of 0x0d20 bytes, loading to address 03000200 (.rtdx_text).
Segment of 0x3b40 bytes, loading to address 03000f20 (.bios).
Segment of 0x4000 bytes, loading to address 03004a60 (.text).
Segment of 0x4000 bytes, loading to address 03008a60 (.text).
Segment of 0x1a40 bytes, loading to address 0300ca60 (.text).
Segment of 0x0360 bytes, loading to address 0300e4a0 (.sysinit).
Segment of 0x4000 bytes, loading to address 0300e800 (heronapi_code).
Segment of 0x0620 bytes, loading to address 03012800 (heronapi_code).
[Server/Loader returned board info: hep9a 1 0]
Trying to serve 1 boards.
Server thread 0 (hep9a 1, fifo 1 in, 1 out <-> node heron)
Write word. This should make HERON LEDs flash 10 times.
Entering Server mode.

Server/Loader running.
Read back the message.
Leaving server mode.
Serving 1 nodes completed.
Message received was 'abcdef', as expected.
End.

D:\server_loader_examples\c6x\s1_api\threads>
D:\server_loader_examples\c6x\s1_api\threads>

```

In the above picture, “ex.bat” was run using the verbose option (“ex -v”). Communications between the Server/Loader (thread) and DSP module will have run in parallel with the communications between the host API thread and the DSP module. For this to work, two connections between host and module were necessary. In the network file you will find: -

```

# Create one duplex HEART connection for use with our API program:
heart      host1  0      heron   0      1 NOSERVE
heart      heron  0      host1   0      1 NOSERVE
# and another duplex HEART connection for use with Server/Loader:
heart      host1  1      heron   1      1
heart      heron  1      host1   1      1

```

By default the Server/Loader will examine all host – board connections. If there's a duplex connection between a module and a host, it will serve that module. If there are multiple duplex connections between a module and a host, the Server/Loader will choose one connection to serve. The NOSERVE keyword is to tell the Server/Loader not to use that connection to serve a module. We can now be sure that our host API program can safely use this module to host connection.

The Server/Loader will automatically serve modules that have a duplex connection created between a DSP module and the host interface. If there's no connection, or the connection is simplex (one-way), then the Server/Loader is unable to execute “stdio” requests from that DSP module.

The “Create new HERON-API project” plug-in will always create a network file (if there's not a network file present already) with 1 DSP HERON module defined and connected to the host. In the case of very simple situations this network file is sufficient. But for more complex situations you will have to add or edit the generated network file. The network file for this example, for example, was hand-edited to suit using 2 host – module fifo connections.

The next step is to create your host PC example program. This is explained after the next section.

### **Manually Setting up the Project**

For your information (or if there is some problem) here is how to set up the project yourself:-

Make sure that you have copied all of the .cdb files from the directory %HEAPI\_DIR%\heron\_api\cmd into the directory C6000\bios\include under the directory where your Code Composer Studio installation is (usually c:\ti).

In Code Composer, select ‘Project → new’ and choose the path for your project. The name must be test for this demo.

Select ‘File → New → DSP/BIOS Config’ and choose the correct .cdb file for your hardware. This will have a name that uses your HERON module number and possibly an option that is available for that module.

In the DSP/BIOS config tool, right click on Global properties, and check that the CLKOUT property is set to the frequency of your processor module. This is used by DSP/BIOS to calculate the correct settings for the timer period.

This .cdb file has some items set up which are for HERON-API. DO NOT CHANGE THESE!

For this example you need to set up a Task that is called TSK0. Under its properties set its function to be “\_maintask”.

Use ‘File → Save’ to save the cdb file to the project directory as test.cdb.

Saving the .cdb file will generate a .cmd file, but that file will not place the sections heronapi\_code and heronapi\_data. For this reason there is a .cmd file supplied by us, in the directory %HEAPI\_DIR%\heron\_api\cmd that will be called by your heron module number and have \_slbios.cmd at the end, i.e. heronx\_slbios.cmd. You need to copy this to your project directory. This cmd file also adds the stio62s.lib to the project. It is done here as it must be in the project before the rts6201.lib which also defines the standard I/O functions.

Now add the source file to the project and the .cdb, and also the heronx\_slbios.cmd. Edit the .cmd file that you have inserted and change the .cmd file that it includes to replace the \*\*\*\*\* by the name of your .cdb file. I.e. change \*\*\*\*\*cfg.cmd to be testcfg.cmd. Because Code Composer Studio does not support the use of environmental variables in the library path you also need to change the line that has %HESL\_DIR% to have the actual path name of where you installed the Server/Loader.

Add the HERON\_API library “herons.lib” from the directory %HEAPI\_DIR%\heron\_api\lib to the project.

Go to Project Options and add %HEAPI\_DIR%\heron\_api\inc and %HESL\_DIR%\inc to the include

path.

Select `-o3` optimisation from the compiler optimisation menu.

The default `.cdb` file will actually place all code into external memory, and switch on the program cache. This is a good general purpose setting, but might need to be changed for your actual application.

Next, you need to add the file “`stub.c`” to the project (Project→Add Files to Project, select “`stub.c`” that resides in the example directory that you created). Edit this file and make sure that the proper heron file is included (e.g. if you have a `HERON1` module, ensure that “`stub.c`” has “`#include <heron1.h>`”, if you have a `HERON4` module, ensure that “`stub.c`” has “`#include <heron4.h>`”, and so on).

You can now build the demo by choosing Project → re-build all. There should be no errors.

## How to compile the host example for Win32 (with Microsoft Visual C/C++)

In order to rebuild this example it is necessary to use Microsoft Visual C++ V6 running under Windows 95/98 or NT/W2K. The resulting program can only be run

- in a Windows 95/98 DOS Box (Console Window)
- in a Windows NT DOS Box (Console Window)

The following instructions assume that you are familiar with using the Microsoft Developer Studio and how to manipulate the workspace:

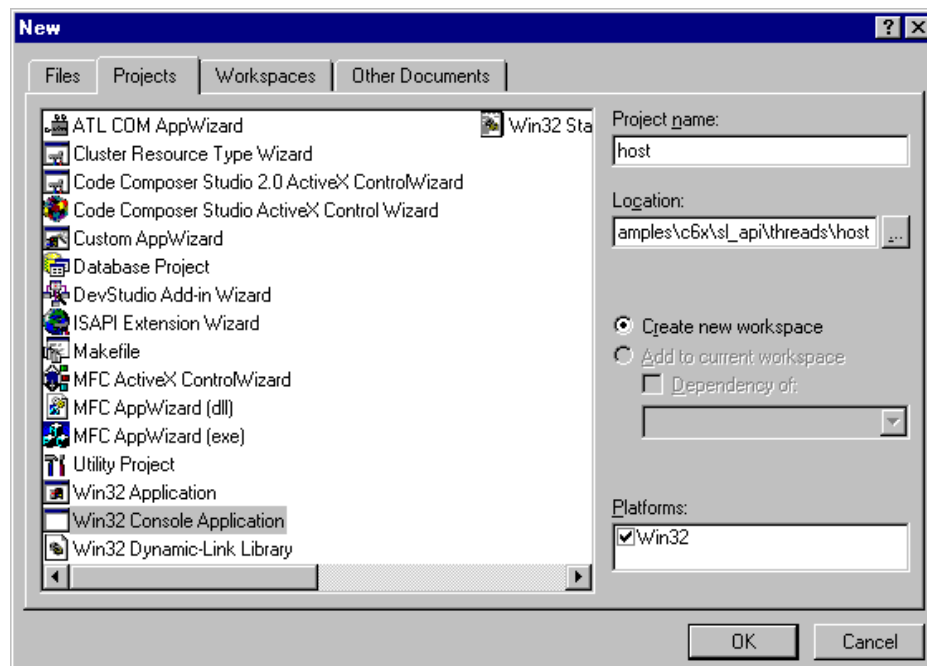
### 1) Create a new workspace.

File → New

In the left-hand pane, select “Win32 Console Application”.

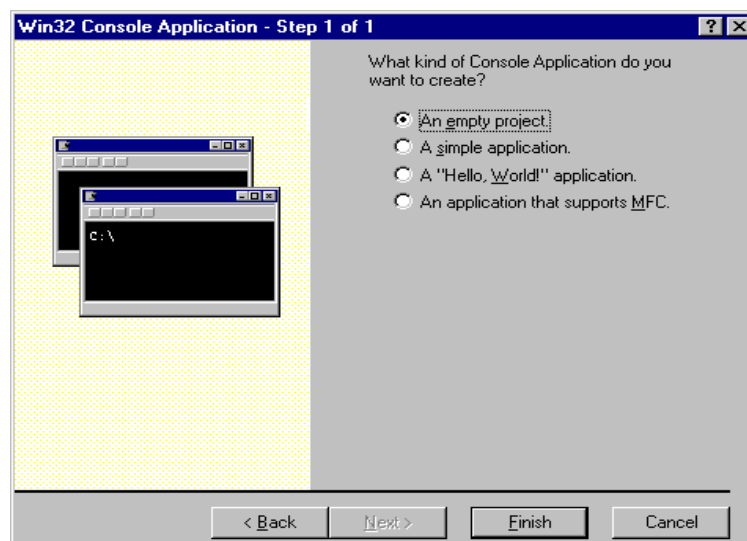
On the right, at “Location”, select your example directory, e.g. “c:\examples\c6x\sl\_api\threads”.

On the right, at “Project name”, choose a name, e.g. “hosr”.



Click “OK”.

Another window appears, select “An empty project”.



Click “Finish”.

A new window appears, click “OK” to confirm.

## 2) Add source files.

There's only 1 source file to add: "threads.cpp".

Project → Add To Project → Files, now browse to the "sl\_api\threads" example directory.

Select the "threads.cpp" source file.

Click "OK".

You can verify that the source file has been added, by looking in the workspace window on the left. Open the workspace by clicking on the '+' before "host files" (if you chose a project name other than "host", you should read your project name in place of "host"). Next, open "Source File". The "threads.cpp" file should be listed there.



## 3) Add the HUNT ENGINEERING API library.

The HUNT ENGINEERING API library is called "hendrv.lib" ("hebdrv.lib" for Borland C/C++).

Project → Add To Project → Files, now browse to the API installation directory, e.g. "c:\heapi".

Change the field "Files of type" to "Library Files (\*.lib)".

Select the "hendrv.lib" library file.

Click "OK".

You can verify that the source file has been added, by looking in the workspace window on the left. In an opened workspace, you can see the library file at the bottom.



## 4) Add the Server/Loader library.

The Server/Loader library is called "win32sl.lib" ("win32slbl.lib" for Borland C/C++).

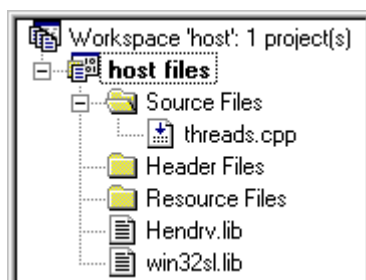
Project → Add To Project → Files, now browse to the API installation directory, e.g. "c:\heapi". (Visual C/C++ may already have put you here.) Now browse to "hesl\lib\win32".

Change the field "Files of type" to "Library Files (\*.lib)".

Select the "win32sl.lib" library file.

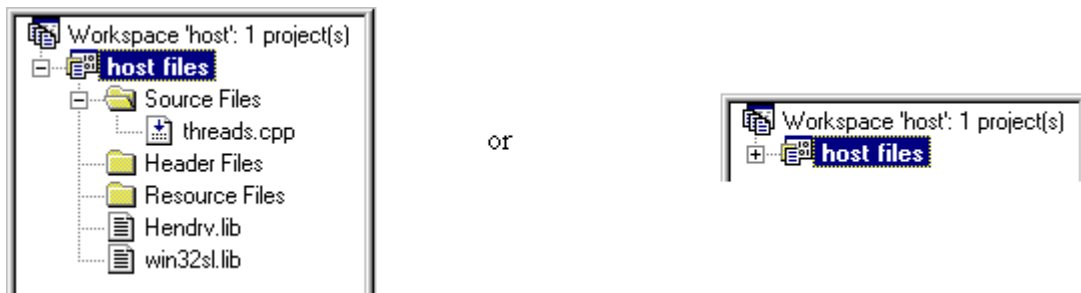
Click "OK".

You can verify that the source file has been added, by looking in the workspace window on the left. In an opened workspace, you can see the Server/Loader library file at the bottom.



**5) Change the project settings to include API and Server/Loader include files.**

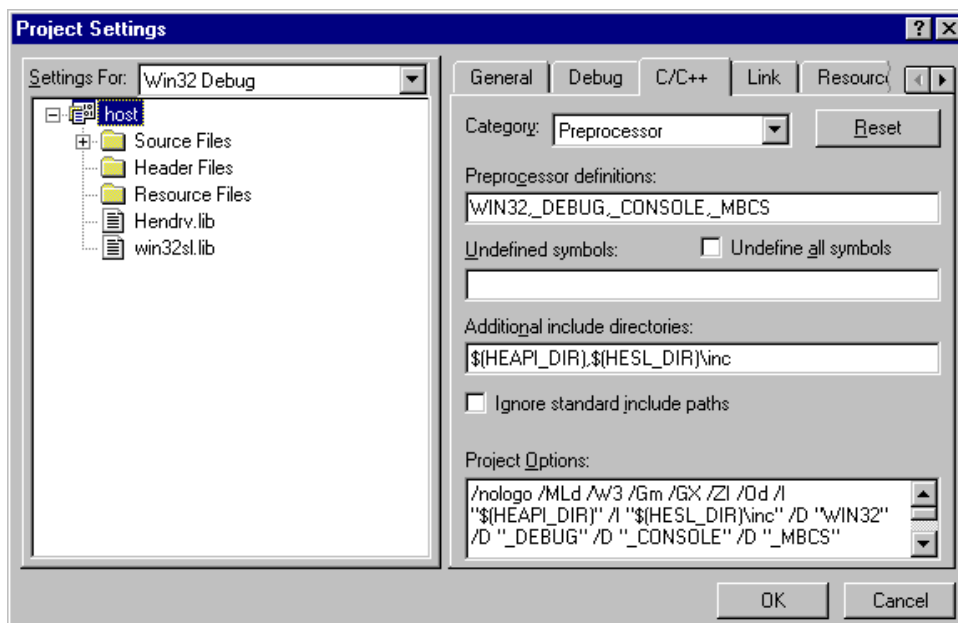
In the left-hand project workspace, be sure that “host files” is currently selected, or close the workspace.



Select “Project → Settings”. Click the “C/C++” tab.

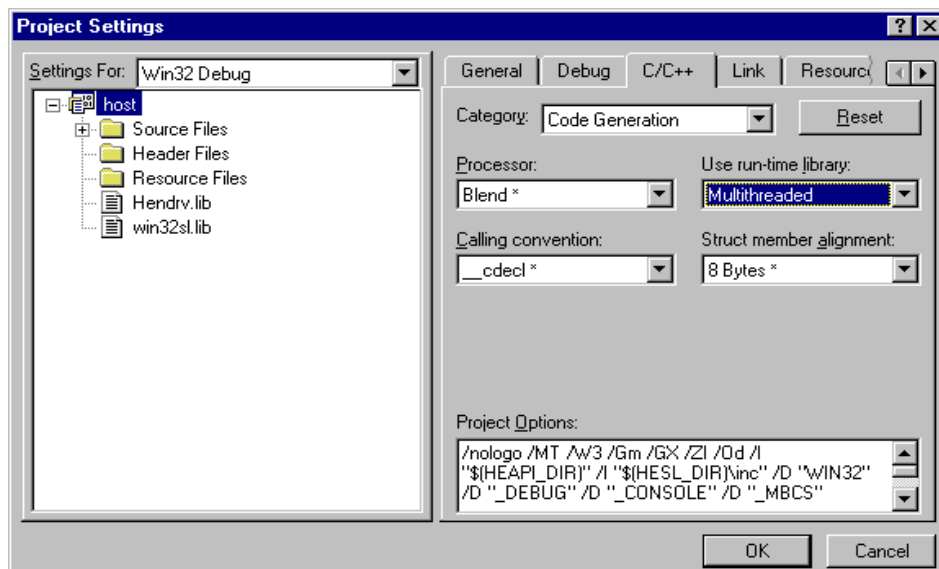
In “Category” at the top of the window, select “Preprocessor”.

Now add the include directories, “\$(HEAPI\_DIR)” (for the HUNT ENGINEERING API) and “\$(HESL\_DIR)\inc” (for the Server/Loader). Separate the two using a comma (“,”).



**6) Make sure that the project is generated with multi-threading support.**

Continuing with the “Project Settings” window, now change category to “Code Generation”. Set the field “Use run-time library” to “Multithreaded”.

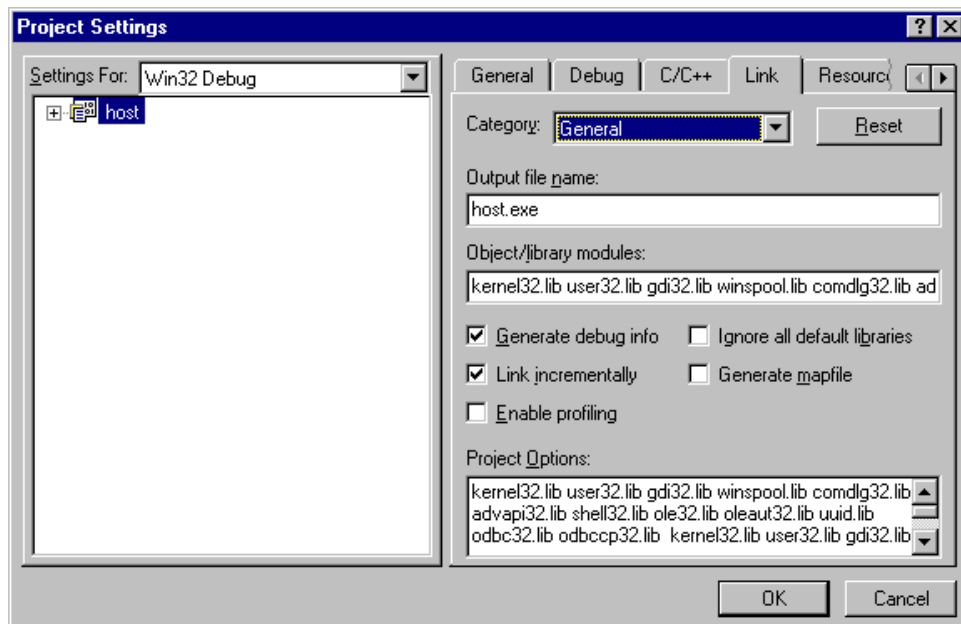


You may want to change the output file path, first go to 7 below. If not, click the “OK” button.

## 7) Optionally, set the output file

It may be useful to change the path of the output file. If you want to use the existing batch file “host.bat”, set the output file to “..\win32\host.exe”. Upon building the project, the output file would overwrite the existing host.exe as delivered as part of the example. In these instructions, we keep the new “host.exe” in the separate sub-directory, but do change the path in deleting the default “debug\” part of it.

Assuming that the “Project Settings” window is still open, select the “Link” tab. Make sure that “Category” field is set to “General”. Now change the output file field to “host.exe”, i.e. delete the “Debug\” part.



## 8) Build the project.

Close the “Project Settings” window by clicking “OK”, if the window was still open.

You can now build the project, e.g. “Project → Build host.exe” or “Project → Build All”.

## 9) Create a new batch file to run your new executable.

To run the resulting executable (“host.exe” if you named your project “host”), it’s perhaps easiest if you create a batch file to run it. Simply copy the existing “host.bat”. This batch file looks like:

```
win32\host %1
```

Change the part that identifies the executable (win32\host) to the newly created executable. If you named your project “host”, then you would change the above to something like:

```
host\host %1
```

Open a DOS box, and change directory to your “threads” directory. Run the newly created batch file. What you see should be identical as when you run “host.bat”.

## How to compile the host program for Win32 (with Borland C/C++)

In order to rebuild this example it is necessary to use Borland C/C++ running under Windows 95/98 or NT/W2K. The resulting program can only be run

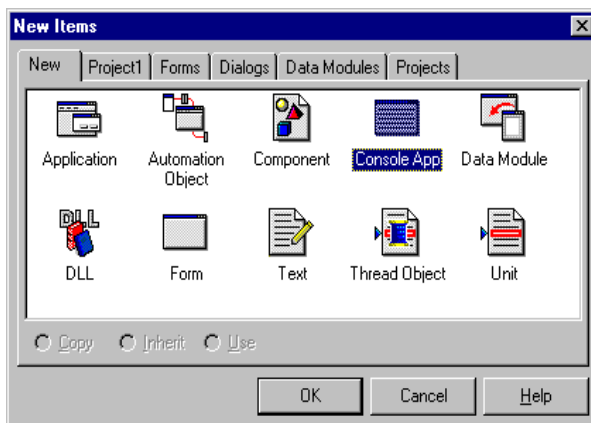
- in a Windows 95/98 DOS Box (Console Window)
- in a Windows NT DOS Box (Console Window)

The following instructions assume that you are familiar with using the Borland compiler and how to manipulate the workspace. The Borland compiler we use is quite old, and it may be that with your version the details of how to do something may differ; but the steps themselves should still be valid.

### **1) Create a new workspace.**

File → New.

Select “Console App”.

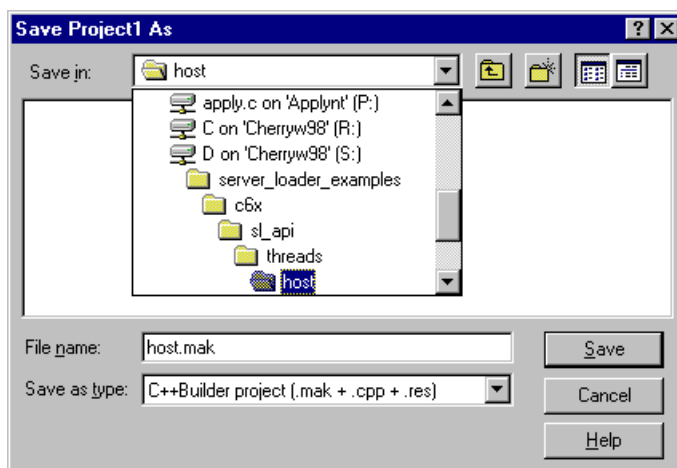


Click “OK”.

### **2) Save the newly created project.**

Create a sub-directory in the “sl\_api\threads” example directory, for example “host”. Then save the newly created project (File → Save Project As) in this newly created sub-directory, using the name “host”. This will create a make file “host.mak” using a C++ file “host.cpp”.

The “Save As” window should look similar to:



### **3) Add source files**

There's only one source file to add: “threads.cpp”. This file is located in the main “sl\_api\threads” example directory, i.e. one directory lower (higher?). The idea is to overwrite the “host.cpp” created by Borland with our “threads.cpp”. Thus, copy “threads.cpp” from the main “sl\_api\threads” example directory, delete the “host.cpp” file that was created by Borland C/C++ Builder when saving the project, and rename “threads.cpp” in the “host” directory to “host.cpp”.

#### 4) Add the HUNT ENGINEERING API library.

First, go to the window that has the “host.cpp” file opened. Put the cursor on the line that says:

```
#include "heapi.h"
```

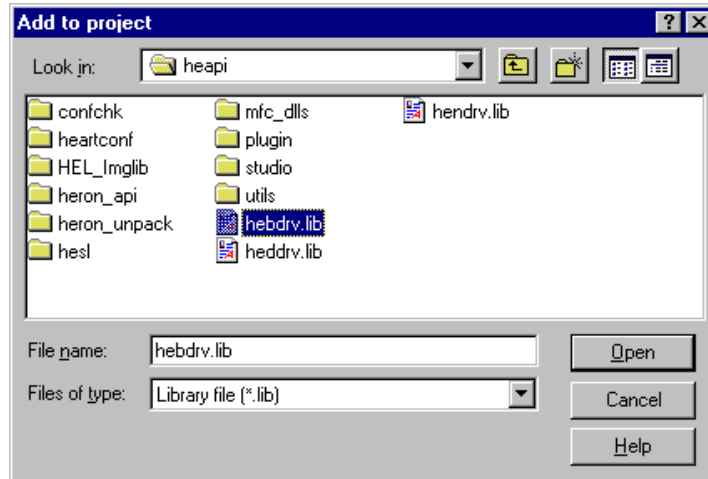
The reason for this is to get Borland C/C++ Builder to add extra source lines at a proper location.

Next, add the HUNT ENGINEERING API library file. The file is called “hebdrv.lib”, and is located in the “\lib\win32” directory of the API installation.

Do a Project → Add to Project.

Browse to the API installation directory.

Change “Files of Type” to “Library files (\*.lib)”.



Select “hebdrv.lib”. Then click “Open”.

#### 5) Add the Server/Loader library.

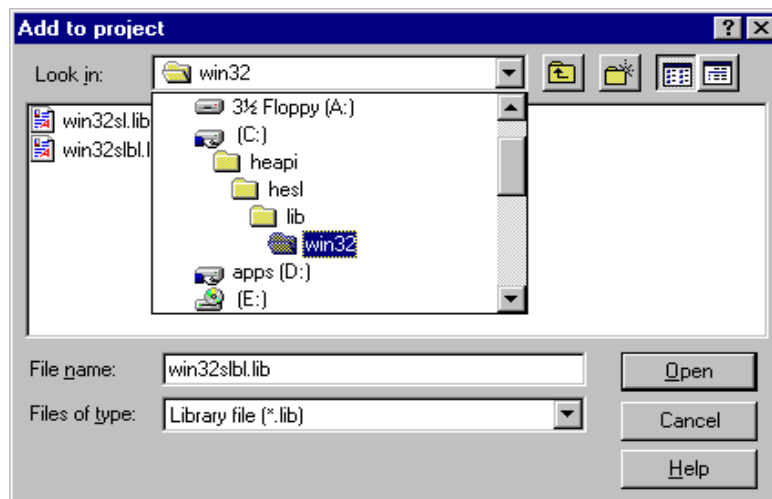
The library file to add is “win32slbl.lib”, located in the “hesl\lib\win32” directory of the API installation.

Project → Add to Project

Browse to the API installation directory, then to “hesl\lib\win32”.

Change “Files of Type” to “Library files (\*.lib)”.

Select “win32slbl.lib”.



Click “Open”.

#### 6) Make sure the libraries are imported correctly.

Borland adds some lines to the source file (“host.cpp”) that import the libraries. But sometimes these lines get inserted at the wrong place. Review your “host.cpp” file and make sure the library lines are inserted at a proper point; if not, change the position. For example, what would work nicely is the

following:

```
...
#include "heapi.h"

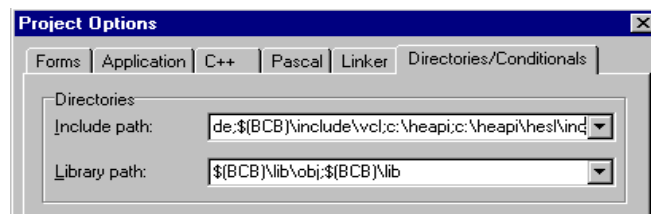
//-----
USELIB( "\\heapi\\hebdrv.lib" );
USELIB( "\\heapi\\hesl\\lib\\win32\\win32slbl.lib" );
//-----
...
```

## 7) Add API and Server/Loader include directories.

Select Options → Project.

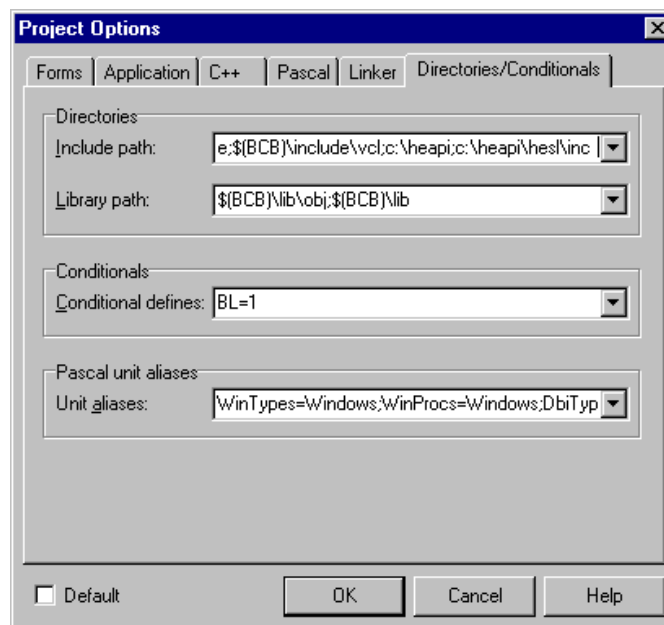
Select the “Directories/Conditionals” tab.

In the “Include path” edit box, add the HUNT ENGINEERING API include directory and the Server/ Loader include directory. In my case, the Borland compiler didn’t allow environmental variables. So I had to add the absolute path. With an API installation in “c:\heapi”, you would thus add “;c:\heapi;c:\heapi\hesl\inc” to the “Include path” edit box. If your Borland compiler version allows environmental variables, you may want to add “;\$(HEAPI\_DIR);\$(HESL\_DIR)\inc” instead.



## 8) Add preprocessor definition “BL”.

In the same window, add 1 preprocessor definition to the “Conditional defines” edit box. The definition to add is “BL=1”. This is necessary, because it selects the Borland specific code in the example code (in “threads.cpp”).



## 9) Build the project.

Project → Build All, or Project → Make.

## 10) Create a new batch file to run your new executable.

To run the resulting executable (“host.exe” if you named your project “host”), it’s perhaps easiest if you create a batch file to run it. Simply copy the existing “hostbl.bat”. This batch file has 1 line as follows:

```
win32\hostbl %1
```

Change the part that identifies the executable (win32\hostsl) to the newly created executable. If you named your project “host”, then you would change the above to something like:

```
host\host %1
```

Open a DOS box, and change directory to the your “sl\_api\threads” directory. Run the newly created batch file. What you see should be identical as when you run the original “host.bat”.