



**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.hunteng.co.uk>



## The “mysl” Server/Loader example.

Rev 3.4 JT 18/02/03 (added troubleshooting tips)

The DSP code in the “mysl” example demonstrates the use of the stdio library supplied with the Server/Loader. The PC code in the “mysl” example shows how the Server/Loader library can be used to create your own customised Server/Loader.

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 2.0 made for HERON-API V2.3

Example revision 3.0 made for CCS V1.2

Example revision 3.1 made for Server/Loader 3.2

Example revision 3.2 made for Server/Loader V3.3

Example revision 3.3 made for Server/Loader 4.0

Example revision 3.4 added troubleshooting tips

## **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 `stdio.c`. (If you happen to have done the "stdtest" example, this is exactly the same C file.) 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 executable is provided, so that you can test that you have a working DSP program, before you start creating your own PC program. The PC C++ code consists of one C++ file called `mysl.cpp` (Microsoft C/C++) or `mybl.cpp` (Borland C/C++).

There are 2 other examples that show how to use the Server/Loader library. They are in "`..\sl_api\batch`" and "`..\sl_api\exe`". Both of these examples show how to use the Server/Loader to boot a system of processors and FPGAs. After booting, a custom host PC program takes over to implement custom I/O between the processors and the host PC. This is useful if e.g. you want to create a Windows host program, or want to transfer custom messages, or want to define your own host protocol.

## **Hardware setup**

The example shows the communication between a HERON module and the host. This means that the HERON module must be connected to the host via a HERON FIFO.

With a HEART based board, such as the HEPC9, such a (duplex) connection must be created using HEART. This can be done with the Server/Loader, by adding some HEART statements in the network file. The "Create new HERON-API project" plug-in will automatically do this for you; this is explained in more detail later.

When using an HEPC8, then a HERON module must be slot 1 of the HEPC8. The module must have its default routing jumpers set to 1, so that it boots from the host.

If you are running the demo on a different hardware configuration, you will need to change 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 provided 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 “mysl” 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 “w32.bat” (uses the standard Server/Loader) and “my32.bat” (uses the provided customised Server/Loader).

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

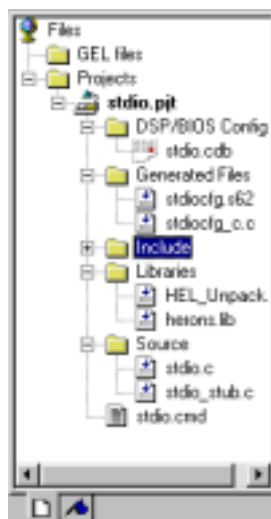
HUNT ENGINEERING provide several Code Composer Plug-in tools that allow you to make your development faster. The first is one that sets up Code Composer ready for your hardware, so you don’t need to configure device drivers etc and can be found from the Start → Programs → HUNT ENGINEERING → AutoConfigure 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 mysl example, simply 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 “mysl” example directory.
- Change “Files of type” to “Project Name (use \*.c, \*.cpp as template).
- Select the “stdio.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\mysl\stdio”. The “c:\examples\c6x” part may differ in your case, depending on where you stored the “mysl” example.
- Create the project by clicking on “Create New Project”.
- At the “HERON Module Select” 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 “stdio\_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 “w32.bat”. This uses the standard Server/Loader. The “w32.bat” batch file has been created by the plug-in. The “my32.bat” batch file will run the same network file (using “stdio.out”), but will use the provided customised Server/Loader executable “win32\mysl.exe”.

This demo is almost the same as the “stdtest” example for the Server/Loader but it uses a Server/Loader that is custom built, allowing you to make additional features.

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

## Heart

For HEART based systems, such as the HEPC9, it is useful to know how the FIFO links have been created – in this example. If you open the network file as created by the “Create new HERON-API project” plug-in, you can see that 2 HEART connections have been defined between the DSP module and the host interface:

```
#-----
# Nodes description
# ND  BD_nb  ND_NAME  ND_Type  CC-id  HERON-ID  filename(s)
#-----
#      c6      0      HERON      ROOT      (0)      00000003  stdio.out
#-----
# The actual HEART programming statements. Used by both Server/Loader and
# HeartConf.
#      from:slot  fifo  to:slot  fifo  timeslots
#-----
#      heart      host1  0      heron  0      1
#      heart      heron  0      host1  0      1
```

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 this simple example this network file is sufficient. But for more complex situations you may have to add or edit the generated network file.

## Manually Setting up the Project

For your information (or if there is a problem with the plug-in) 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). The AutoConfigure plug-in should already have done this.

In Code Composer, select ‘Project → new’ and choose the path for your project. Its name must be “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 demo.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 democfg.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.

## Troubleshooting

Depending on HERON module type and size of the code and data in your project, you may find you encounter an error message as follows:-

```
Error: can't allocate .IDRAM$heap ...
```

The reason is that the sum of (software) segment sizes that are to be placed in the IDRAM memory is larger than the IDRAM memory size. Solution is to reduce in size one or more (software) segments, or to move software segments from IDRAM onto external memory (for example SBSRAM, SDRAM).

Projects created with the Create New HERON-API project plug-in are configured to create a ‘map’ file. In your project directory you can find a \*.map file (where \* is the name of your project) after you built your project. The ‘map’ file will show you where the different software segments have been placed. This allows you to see segment sizes and where they were placed.

A snippet from the \*.map file of this example would show you:-

```
.bss          0      800000a0      00005590      UNINITIALIZED
              800000a0      00004080      stdio.obj (.bss)
              80004120      0000081c      stio62s.lib : bootload.obj (.bss)
```

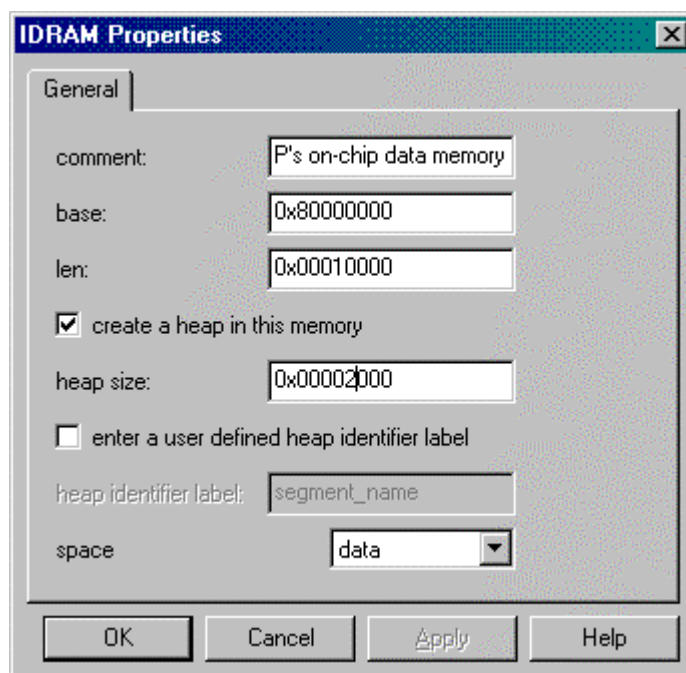
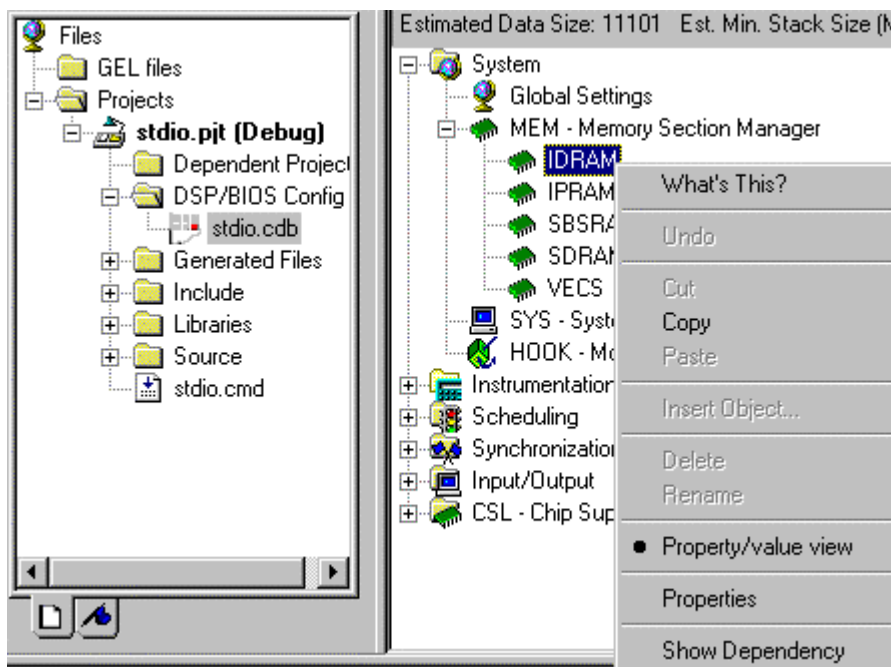
The ‘.bss’ segment is all global data found in all software sections of your code and libraries. You can see immediately that the C example file takes quite a chunk of memory: 0x4080 bytes. When we inspect the C file, stdio.c, we can see that this is mostly ‘static int Data[4096]’. One way of solving the problem is to dynamically allocate the memory, instead of using a static global array:-

```
Int * Data = NULL;
Data = malloc(4096*sizeof(int));
```

Dynamically allocated memory is allocated from the (user) heap. The user heap is located in external memory: SDRAM. However, you don’t always want to put your data in external memory.

But sometimes you want to have your global data in internal memory. In that case, to fix the problem, instead of dynamically allocating global data, try reducing the size of one or more segments. One of the segments you could reduce is size is the system stack or the system heap (note: *system* heap is used by DSP/BIOS, *user* heap by your program; they are different). DSP/BIOS uses the system heap to dynamically allocate DSP/BIOS objects. For projects that use few DSP/BIOS resources it's safe to reduce the size of the system heap from the default 0x4000 bytes to 0x2000 or even 0x1000 bytes.

To reduce the IDRAM heap size, open your CDB file. Go to 'MEM – Memory Section Manager' and open the properties for IDRAM (right-click on the IDRAM entry). There's an entry 'heap size' which defaults to 0x4000. Change this to 0x2000 or 0x1000. Click OK. Now rebuild.



The heap in IDRAM is used by the 'system' (ie DSP/BIOS). (If you would try to un-tick 'create a heap in this memory' and CCS will warn you that try to remove system heap.)

## How to compile the customised Server/Loader 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:

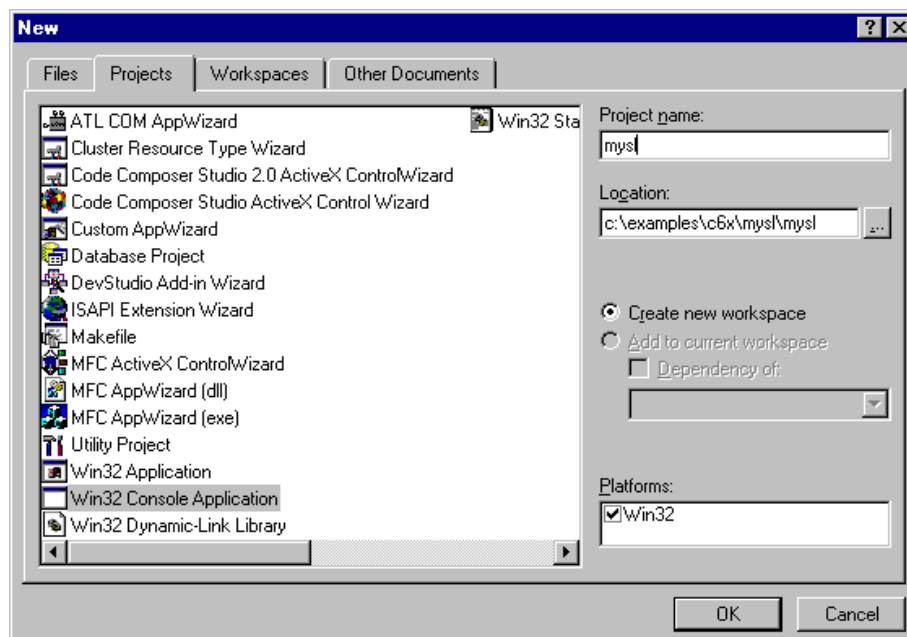
### 1) Create a new workspace.

File → New

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

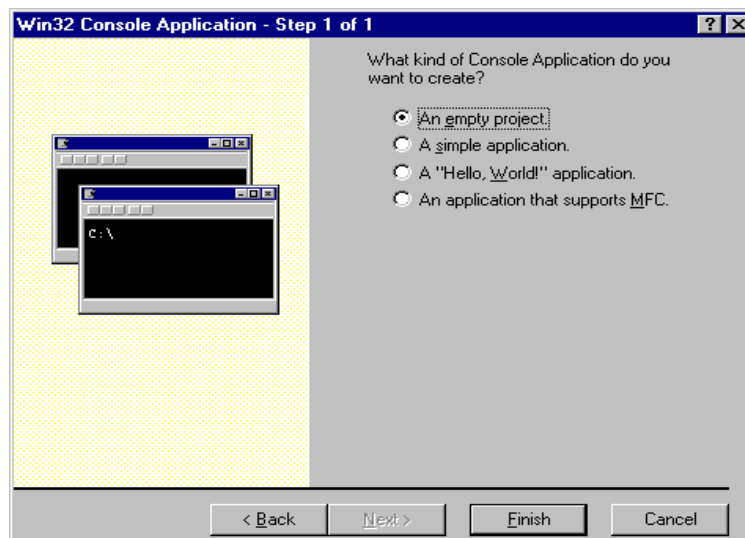
On the right, at “Location”, select your example directory, e.g. “c:\examples\c6x\mysl”.

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



Click “OK”.

Another window appears, select “An empty project”.



Click “Finish”.

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

## 2) Add source files.

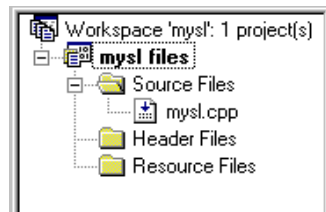
There's only 1 source file to add: "mysl.cpp".

Project → Add To Project → Files, now browse to the "mysl" example directory.

Select the "mysl.cpp" source file. (The "mybl.cpp" file is a similar file for use with Borland C/C++.)

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 "mysl files" (if you chose a project name other than "mysl", you should read your project name in place of "mysl"). Next, open "Source File". The "mysl.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++).

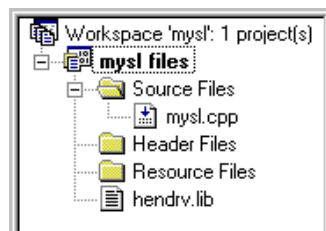
Project → Add To Project → Files, now browse to the API installation directory, e.g. "c:\heapi".

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++).

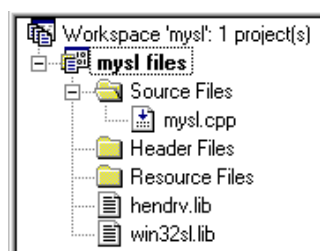
Project → Add To Project → Files, now browse to the API installation directory, e.g. "c:\heapi". (Visual C/C++ may already have put you here.) Now browse to "hes\lib\win32".

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

Select the "win32sl.lib" library file.

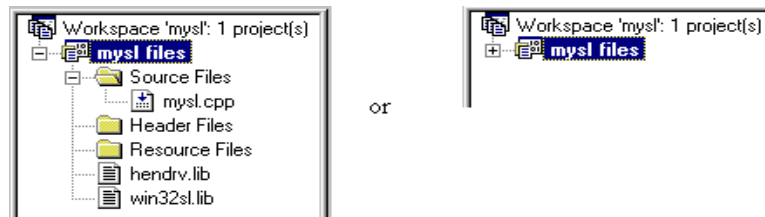
Click "OK".

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 “mysl files” is currently selected, or close the workspace.

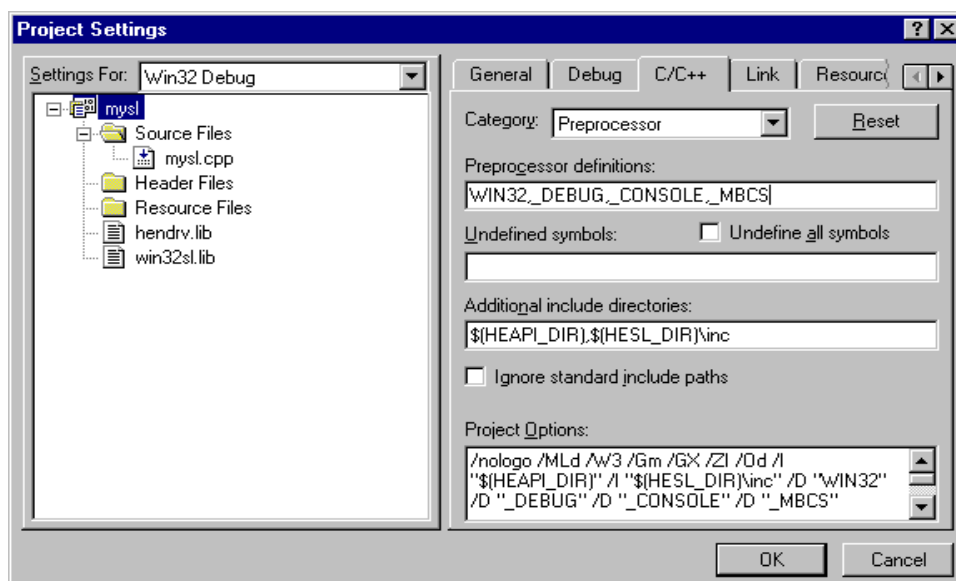


Select “Project → Settings”.

Click the “C/C++” tab.

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

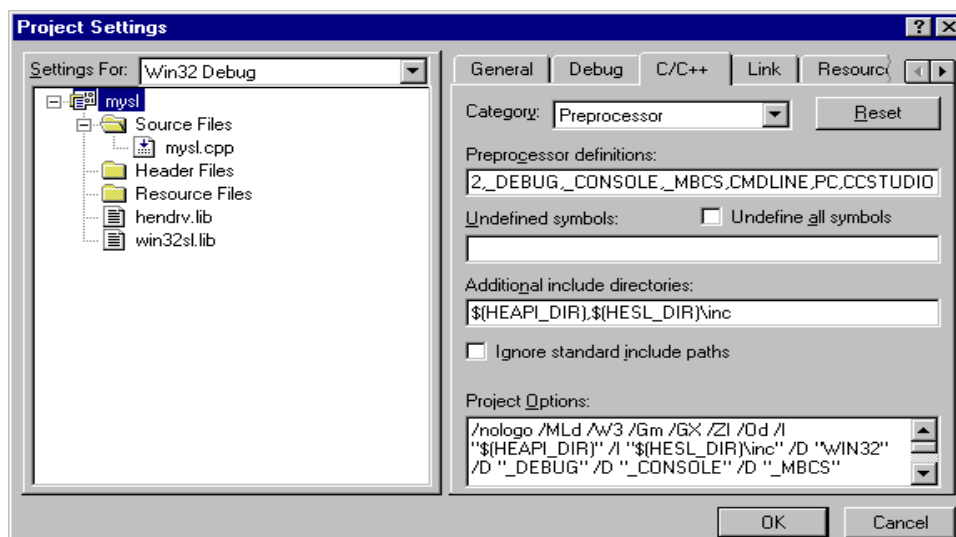
Now add the include directories, “\$(HEAPI\_DIR)” (for the HUNT ENGINEERING API) and “\$(HESL\_DIR)\inc” (for the Server/Loader). Separate the two using a comma (“,”).



No, don’t click “OK” yet, next we need to add preprocessor defines.

## 6) Add preprocessor definitions “CMDLINE”, “PC” and “CCSTUDIO”

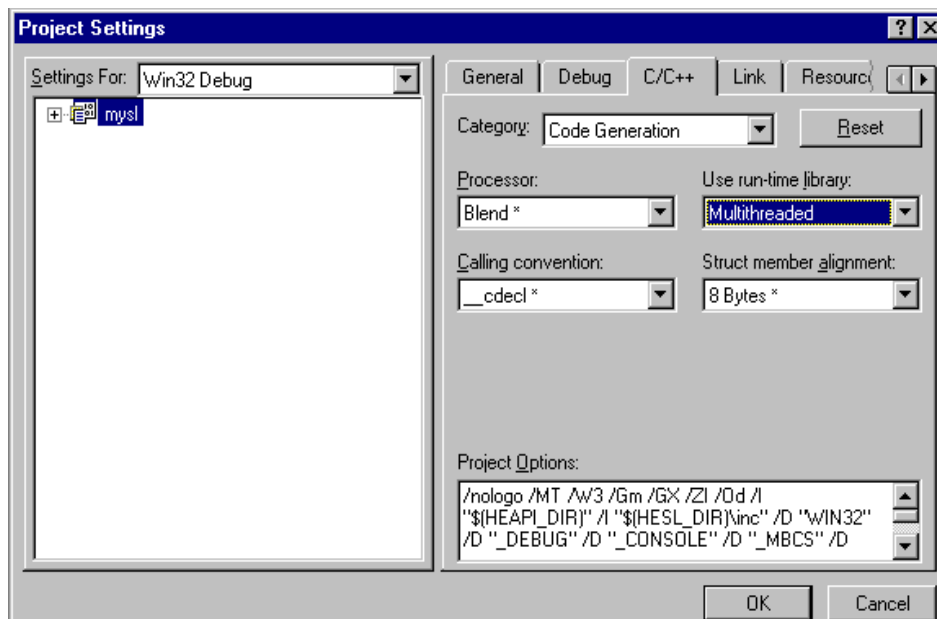
Whilst still having “Project Settings” open at the “C/C++” tab and “Category” set to “Preprocessor”, now add 3 preprocessor definitions: “CMDLINE”, “PC” and “CCSTUDIO”. (Borland C/C++ users must not define “CCSTUDIO”).



The reason for this is that the Server/Loader “network.h” file uses those preprocessor definitions, and that the Server/Loader library was compiled with the preprocessor definitions set as per the above. If you prefer, instead of defining preprocessor definitions as shown, you could also edit the “network.h” file and “fix” the preprocessor definitions (i.e. set “CMDLINE”, “PC” and “CCSTUDIO” to 1 in “network.h”). However, you may break compatibility with any future releases of the Server/Loader, i.e. you would have to re-edit your “network.h” if you upgrade to a new Server/Loader version.

## 7) 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”.

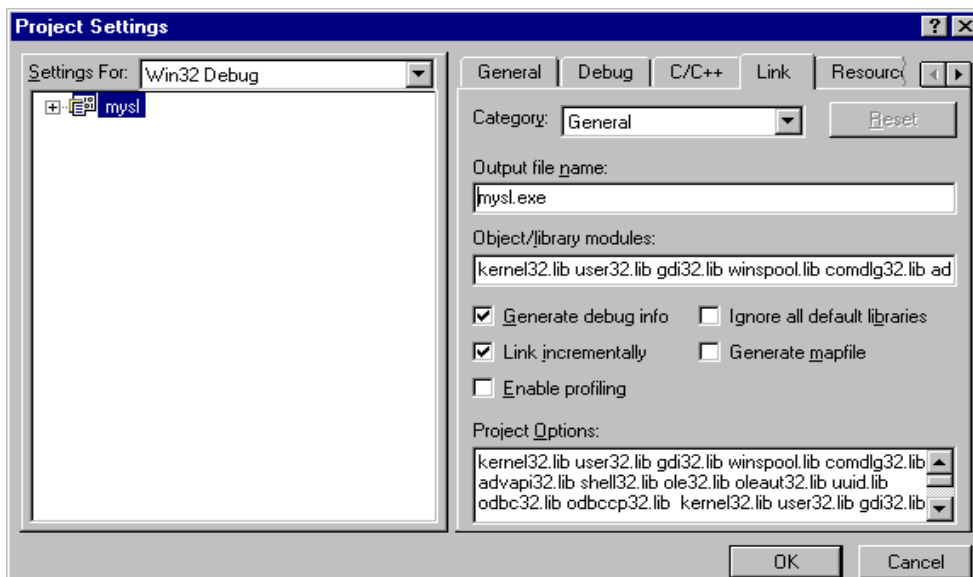


You may want to change the output file path, in that case first go to 8 below. If not, click the “OK” button.

## 8) 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 “my32.bat”, set the output file to “..\win32\mysl.exe”. Upon building the project, the output file would overwrite the existing mysl.exe as delivered as part of the example. In these instructions, we keep the new “mysl.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 “mysl.exe”, i.e. delete the “Debug\” part.



## 9) 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 mysl.exe” or “Project → Build All”. Visual C/C++ may complain about the “common” class being a non dll-interface class. You can ignore such complaints.

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

To run the resulting executable (“mysl.exe” if you named your project “mysl”), it’s perhaps easiest if you create a batch file to run it. Simply copy the existing “my32.bat”. This batch file has 1 line as follows:

```
win32\mysl -I%HESL_DIR%\lib -rls%1 network
```

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

```
mysl\mysl -I%HESL_DIR%\lib -rls%1 network
```

Open a DOS box, and change directory to the your “mysl” directory. Run the newly created batch file. What you see should be identical as when you run “w32.bat” or “my32.bat”.

## How to compile the customised Server/Loader 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

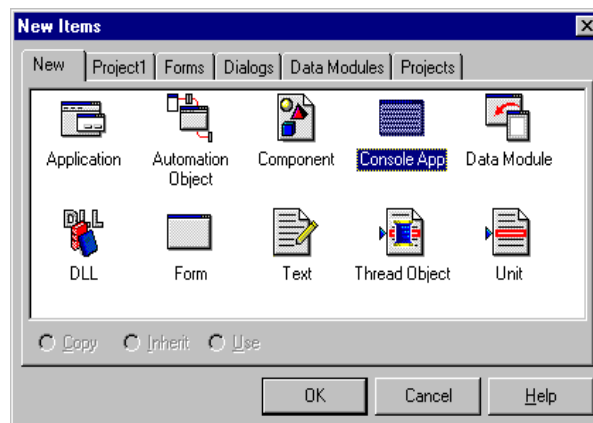
- in a Windows 95/98 DOS Box (Console Window)
- in a Windows NT DOS Box (Console Window)

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”.

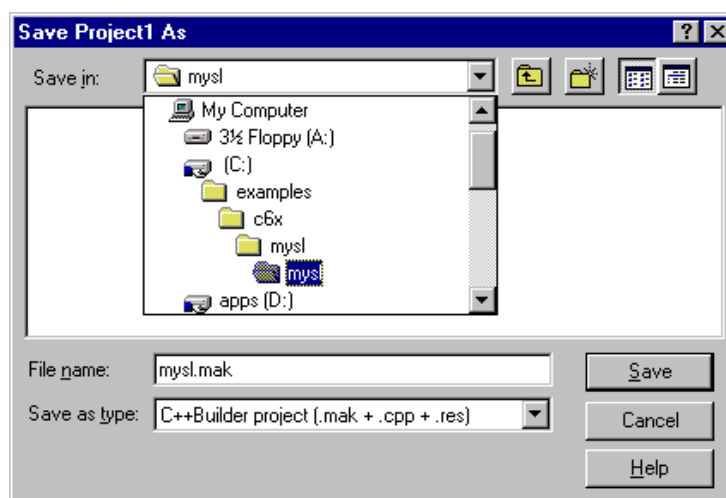


Click “OK”.

### 2) Save the newly created project.

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

The “Save As” window should look similar to:



### 3) Add source files

There's only one source file to add: “mysl.cpp”. This file is located in the main “mysl” example directory, i.e. one directory lower (higher?). The idea is to overwrite the “mysl.cpp” created by Borland with the example “mysl.cpp”. Thus, copy “mysl.cpp” from the main “mysl” example directory over the “mysl.cpp”

file that was created by Borland C/C++ Builder when saving the project.

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

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

```
#include "signal.h"
```

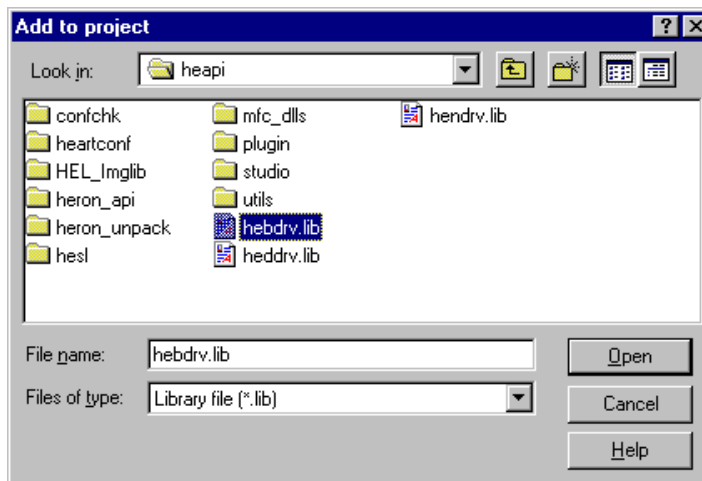
The reason for this is to get Borland C/C++ Builder to add extra source lines at a proper location.

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”.

Click “Open”.

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

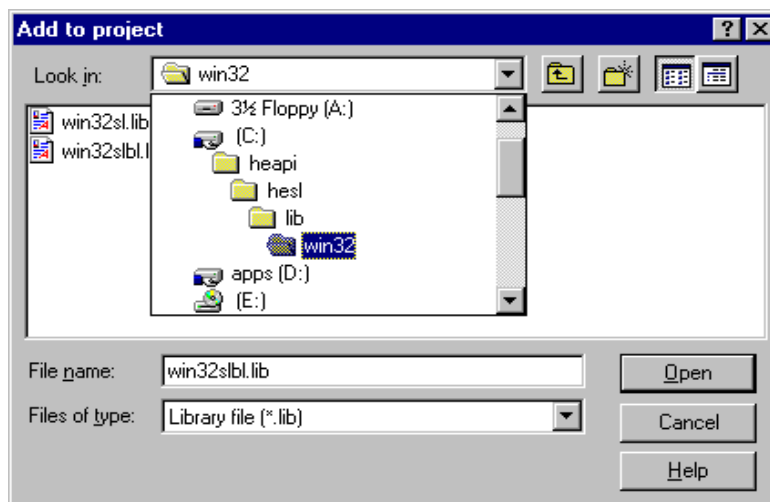
The library file to add is “win32slbl.lib”, located in the “hesl\lib\win32” directory of the API installation.

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 ("mysl.cpp") that import the libraries. But sometimes these lines get inserted at the wrong place. Review your "mysl.cpp" file and make sure the library lines are inserted at a proper point; if not, change the position. For example, what would work nice is the following:

```
...
#include "signal.h"

#include "heapi.h"
#include "network.h"

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

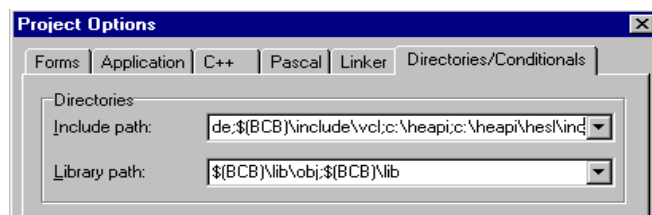
#if (WIN32 && CCSTUDIO)
#include "ccif.h"
#endif
...
```

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

Select Options → Project.

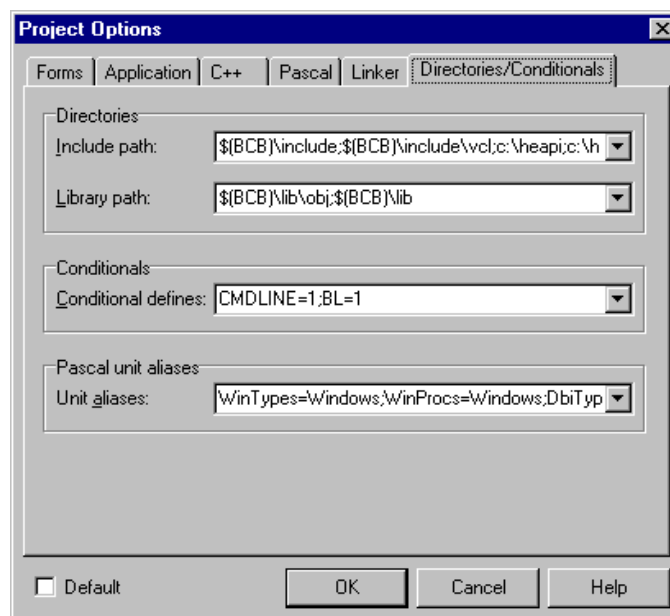
Select the "Directories/Conditionals" tab.

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 definitions "CMDLINE" and "BL".

In the same window, add 2 preprocessor definitions to the "Conditional defines" edit box. They are "CMDLINE=1;BL=1". The first, "CMDLINE=1" is necessary because it is defined in "network.h" and that is the way the Server/Loader library was build for Borland. The second, "BL=1", is necessary, because it selects the Borland specific code in the example code ("mysl.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 (“mysl.exe” if you named your project “mysl”), it’s perhaps easiest if you create a batch file to run it. Simply copy the existing “bl32.bat”. This batch file has 1 line as follows:

```
win32\mybl -I%HESL_DIR%\lib -rls%1 network
```

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

```
mysl\mysl -I%HESL_DIR%\lib -rls%1 network
```

Open a DOS box, and change directory to the your “mysl” directory. Run the newly created batch file. What you see should be identical as when you run “w32.bat” or “bl32.bat”.