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.demon.co.uk
URL: http://www.hunteng.co.uk

Selecting Memory Models and Libraries

Rev 1.0. R.Williams 04-04-00

This document discusses how to select the best memory model for your application and how to choose between the libraries that are provided as part of the HERON eXpressDSP Framework. The reason for using each particular memory model is discussed.

For the majority of users it is expected that the default small memory model will be the most appropriate memory model with which to build their application. For these users this document should be used as a guide to making the best use of the small memory model.

For the remaining users, their development cycle will usually start with the small memory model and move to one of the large memory models according to the changing requirements of their application. For these users, this document explains when to move to a large memory model, and the issues that must be considered when doing so.

This document considers the choice of memory model and library when using the HERON-API.

It is assumed that the reader is already familiar with using Code Composer Studio, DSP/BIOS and the HERON-API and that they have read the documents that are provided on the HUNT ENGINEERING CD that introduce the tools and how to build a DSP application.

The HERON-API

The HERON-API is the communications library that HUNT ENGINEERING provides to perform the inter-processor and processor to I/O communications. Its purpose is to prevent the user from needing to intimately understand the communications mechanism, by providing an optimised way to use the limited DMA resources of the 'C6000 in a choice of ways.

This is the component of the HERON eXpressDSP Framework that makes it possible to use multiple processors and to interface with I/O modules.

It also serves the purpose of providing a common software interface to the various 'C6000 HERON modules that HUNT ENGINEERING produce or plan to produce. Although the hardware of those modules will be different, the HERON-API interface will stay the same.

The HERON-API is fully integrated and dependent upon DSP/BIOS. This means that you must learn how to set up the DSP/BIOS environment to properly use the HERON_API.

For complete information on how to use the features of the HERON-API refer to the user documentation for the HERON-API.

Memory Models

The 'C6000 C Compiler supports five memory models. A memory model defines how data is accessed and how function calls are performed. Each memory model provides a different approach to code and data access.

A DSP application will typically contain several components, each of which may possibly use a different memory model. In order to control how each component of an application performs data access and code access, the user must consider three types of application component.

The first type of component is a C source file that is included in the project. The memory model that is used by this component is defined by the compiler options that are set as part of the project options, and by use of the compiler keywords `near` and `far`.

The second type of component is an assembly source file. The memory model that is used by this component is set in the assembly itself, by the way that data access and code accesses are written.

The third type of component is a library included by the project. The memory model that is used by that library is defined when that library is built.

In the case of the Run Time Support (RTS) library provided by T.I., one library is provided that will work with any combination of memory model used by the other application components. This is possible because T.I. are able to integrate features into the compiler that control how the RTS library is used. By default, RTS functions are called with the same convention as ordinary user coded functions, and all RTS data is defined as far data.

In the case of the HERON-API, HUNT ENGINEERING provides four libraries, each of which uses a different memory model. By selecting the appropriate library from a choice of several libraries, the memory model used by that library component can be defined.

When developing an application there are two main points to consider that affect the memory model you choose to compile with and the libraries that you link to.

The first is how data is placed and accessed, and the second is how code is placed and accessed.

Data Access

The C compiler creates a default section in which it places all global and static data. This section is called `.bss`. With the creation of this section, the C compiler also generates a pointer to the beginning of this section. This pointer is called the Data Page Pointer (DP).

When you build your application for the default compiler memory model (the small memory model), the compiler will by default place all global and static data in the `.bss` section and will access that data using the `near` access method. The near access method will access this data relative to the Data Page Pointer, and each access will take one assembly instruction. This is the most efficient form of data access.

With the small (default) memory model, there is a requirement that the size of the `.bss` section is less than 32Kbytes. This is because for the small memory model, a near data access uses an offset from the Data Page Pointer, where this offset is limited to 32Kbytes in size.

An alternative method of data access is the `far` access method. The far access method can be used by compiling with one of the large memory model options or by using the `far` keyword when declaring data. By accessing data as far, that data is now no longer placed in the `.bss` section, and there is now no 32Kbyte limit on the size of all that data.

However, unlike near accesses, far accesses are less efficient as each access will take three assembly instructions to perform. The extra instructions will increase the time the access takes to complete, and will also increase the size of the program code.

So there is a trade-off between the use of near data and far data. Near data offers the best performance but a program is limited to having 32Kbytes of near data at most.

Far data does not have this 32Kbyte limit, but offers slower performance than near data.

Code Access

When a function is called, the program must perform a branch from the function that is currently executing to the new function that has been called. This branch can be performed in one of two ways.

The first method is generated by the compiler adding a relative offset to the current program counter in order to perform the branch. This method is the `near` access method, and takes one assembly instruction to perform. It is the most efficient form of function call, and hence offers the best performance.

The second method is generated by the compiler forming an absolute address to which the program counter will branch. This method is the `far` access method. It is not as efficient as a near function call as three assembly instructions are required. The extra instructions will increase the time the function call takes to complete, and will increase the size of the program code.

When using a near function call, there is a requirement that the relative offset for the branch is less than +/-1Mword from the current program counter position. If this requirement is not met, a far function call is required.

When building with the small (default) memory model, all function calls are treated as near. By compiling with one of the large memory model options or by using the `far` keyword when declaring functions, a far call will be used.

As a general rule, far function calls should be used where the total code size is greater than 1Mword (or 4Mbytes), or if code sections are being placed in separate memory segments. Where code is placed in more than one memory segment, this will usually result in an address span between the code in one segment and the code in another exceeding 1Mword. In this case, far function calls must be used.

The Default Memory Model

By default, when you start a new project in Code Composer Studio the small memory model is chosen.

The small memory model offers the best performance, and for most applications is the best choice of memory model. It is expected that for the majority of users the small memory model can, and should be used. Only when the size of program code or data exceeds certain limits will a large memory model be required.

Where the limits for data access or code access have been exceeded by an application, a linker error will result when the application is built. The linker errors that are created when this occurs can be used as a guide to making an appropriate change of memory model. Please refer to the section on 'Linking Issues' for a description of what to do when one of these linker errors occur.

For users of the HERON-API, the library `herons.lib` offers the best performance, and again, for most applications is the best choice of library. It is recommended that users developing a new application start with the library `herons.lib`. If during development there is a significant increase in data size or code size, an alternative HERON-API library may then be required.

The following sections discuss how to get the most out of the small memory model, when to compile using the large memory model, and when to use another HERON-API library.

Making the Most of the Small Memory Model

The C compiler creates a default section (named `.bss`) in which it places all global and static data. Data placed in this default section will be accessed using `near` data accesses. A near access will take one assembly instruction, and is the most efficient form of data access.

Data placed in sections other than the `.bss` section will be accessed using a `far` access. A far access requires three assembly instructions, which introduces more cycles per access, and requires more storage for program code.

Due to the way that near accesses are implemented by the compiler, there is a requirement that the size of the `.bss` section is less than 32Kbytes. This is because for the small memory model, a near data access uses an offset from the Data Page Pointer, where this offset is limited to 32Kbytes in size.

In can be seen from this, that for any program the storage of data can be split into two parts. The most important and frequently accessed variables should be placed in the `.bss` section where they can be accessed as near data, and all remaining data should be placed in other sections to be accessed as far data.

In order to make the most of the small memory model, you should therefore carefully consider where data is placed to make the most of the 32Kbyte limit for near data.

To have data placed in the `.bss` section and be accessed as near, you must simply declare that data globally (or maybe as a static local variable), not using the `far` keyword in the declaration. Having done this, compiling with the small memory model will result in that data being placed in the `.bss` section.

To have data not placed in the `.bss` section there are several alternative approaches that may be used.

By using the `far` keyword in the declaration of global or static data, that data will be placed in a section named `.far`.

By using the `#pragma DATA_SECTION` directive in the source code, that data will be placed in the section named in the directive. The data will then be accessed as far. Note, by using the `DATA_SECTION` pragma you will need to declare that data as `far` in order to ensure it is properly accessed. This solution is therefore little different to that given in the previous paragraph.

By accessing data via a pointer, the data access automatically becomes far (apart from accesses to update the value of the pointer itself). Similarly, by dynamically allocating memory using functions such

as the T.I. provided RTS function `malloc`, that data will be accessed via a pointer and will therefore be accessed as far data.

By compiling with the large memory model option `–ml0`, all `aggregate` data will become accessed as far data, with `scalar` data remaining near data. (Note, aggregate data refers to grouped data elements such as arrays and structs, and scalar data refers to single data elements such as `char`, `int`, and `long`.)

By compiling with the large memory model option `–ml3`, all data (that is, both scalar data and aggregate data) will become accessed as far data.

We can see from the options outlined above that there are several ways in which we can control how data is allocated into separate program sections. We can also see that it is important to try to achieve a placement where the most important variables are stored in the `.bss` section and the remaining variables are stored elsewhere.

In order to make the most of the performance benefits of near data accesses, your application should be built to use the small memory model. When building your application you should compile using the small memory model, and link to the HERON-API library `herons.lib`. Data should be allocated according to the guidelines given below.

1. Declare the most important variables as global data or static data. These would typically be variables that are accessed frequently, or variables that are accessed during a time critical section of code. Do not use the far keyword in the declaration of these variables. Do not use a `DATA_SECTION` pragma to control the placement of these variables.
2. Where you are using large data arrays in your processing, you should either declare a pointer to each 'array' and dynamically allocate storage for that array, or you should declare the array globally, and use the `far` keyword in the declaration. By doing so, that data will be allocated in the section named `.far`. Note, storage can be dynamically allocated by using the T.I. run-time-support library functions such as `malloc`.

When following these guidelines, the `.bss` section will contain the global data that you defined as per point 1, as well as global data used by the HERON-API and global data used by DSP/BIOS. If the sum total of this global data exceeds 32Kbytes a linker error will be issued when building the application.

When this happens you must reduce the amount of data stored in the `.bss` section by changing some of your global data declarations to include the `far` keyword or to have that data dynamically allocated.

Alternatively you can link to the library `heronl0.lib` in place of the library `herons.lib`. The library `heronl0.lib` will not use the `.bss` section. Instead, HERON-API global data will be placed in a far data section named '`heronapi_data`'.

Choosing a Large Memory Model

The library `herons.lib` has been built to access all data as near. As such, all global data used by this version of the HERON-API library is placed in the default section `.bss`. The only exception to this is the global error variable `heronerr` which is always declared as far and is placed in the section '`heronapi_data`'. All function calls are performed as near.

The `herons.lib` library should be used where the total amount of global and static data in your application plus the HERON-API global data is less than 32Kbytes, and where the total size of the program code is less than 1Mword (or 4Mbytes). Please note all code sections should be placed in the same memory segment. In this case, your application should be built using the default small memory model to achieve the best performance.

Where the total amount of application code exceeds 1Mword, or if code is being placed across several memory segments, then the application should be built to use far function calls. In this case you should build with the memory model option `–ml1`, and you should link to the library `heronl1.lib`. This version of the library will access data as near and will perform far function calls.

When you need your application to use all of the `.bss` section you should link to the library `heronl0.lib`. This version of the library places all of the library global data in a far data section named '`heronapi_data`'. By using this library, your application is free to use all of the `.bss` section. In this case you can build with small memory model, but please ensure that you keep the size of the `.bss` section to less than 32Kbytes. This library will perform all function calls as near.

If you want to combine the features of both the `heronl0` and `heronl1` libraries, you can use the library `heronl3.lib`. This version of the library will place all library global data in a far data section named '`heronapi_data`'. The library will perform all function calls as far. In this case you should build your application with the memory model option `–ml1`, keeping the size of the `.bss` section to less than 32Kbytes.

The table below summarises when you would use each library.

Is the size of the `.bss` section <32Kbytes inc. HERON-API data?	Are all code sections to be placed in the same memory segment?	Is the total code size <1Mword?	
Yes	Yes	Yes	Use herons.lib
Yes	No	Yes	Use heronl1.lib
Yes	Yes	No	Use heronl1.lib
No	Yes	Yes	Use heronl0.lib
No	No	Yes	Use heronl3.lib
No	Yes	No	Use heronl3.lib

Linking Issues

When linking your application the linker will generate error messages if the rules for near data accesses or near function calls are broken.

If you are using the small memory model and are accessing data in the `.bss` section, the following error message will be generated when the size of the `.bss` exceeds 32Kbytes.

```
>> relocation value truncated at 0x?? in section .text, file example.obj
```

In this case, you have too much global and static data in your application. This problem can be solved by either dynamically allocating some of the data using functions such as `malloc`, or by declaring some of your data as far, or by using a different HERON-API library that doesn't use the `.bss` section.

If you try to call a function using a near function call, and if that function is too far to be reached with the normal program counter relative (PC-relative) branch instruction, you will see the following linker error message:

```
>> PC-relative displacement overflow. Located in file.obj, section .text,
SPC offset ??
```

In this case, you are building your application to use near function calls and either the amount of code in your application exceeds 1Mwords, or you have placed some code sections in one memory segment, and other code sections in a separate memory segment.

Other than reducing the size of your code, the simplest way to remove this problem is to build your application using the `–ml1` memory model in order to use far function calls.

When re-building your application with this memory model, you will also need to link to a different HERON-API library. If you were previously linking to the library `herons.lib` then you will need to use `heronl1.lib` instead. However if you were linking to the library `heronl0.lib` then you will need to use `heronl3.lib`.