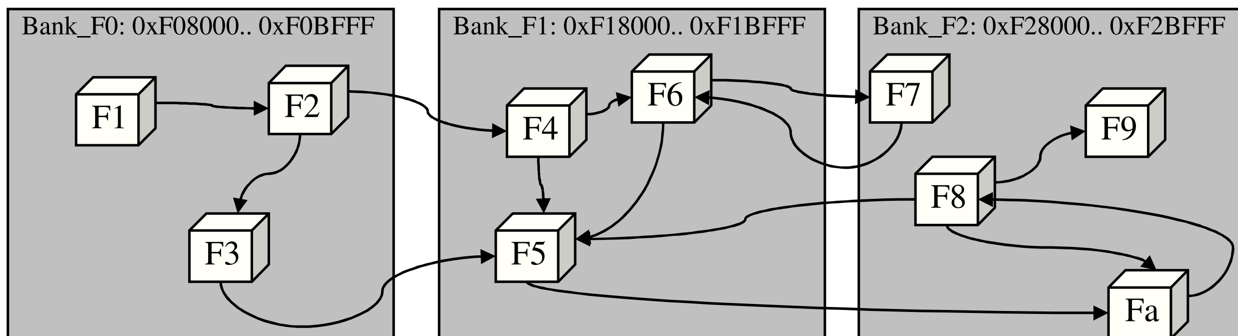
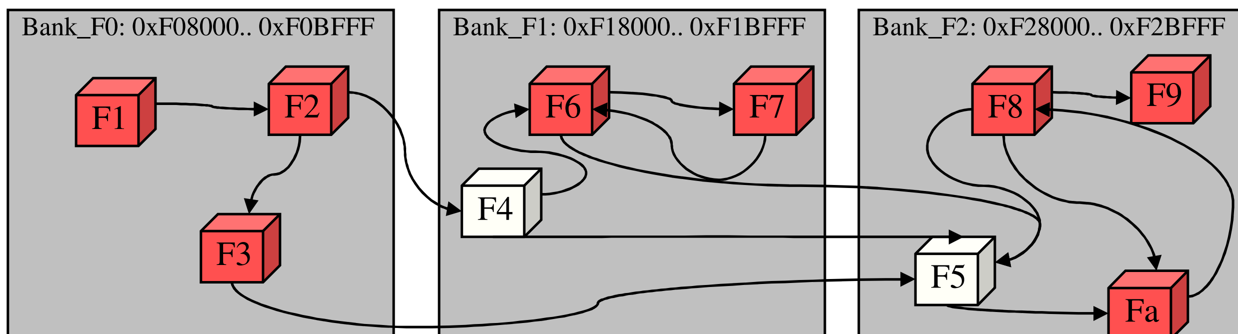
Using the MemoryBanker in S12(X) projects

The CodeWarrior™ HC12 compiler and linker support automatic bank distribution (also called MemoryBanker in this document) for code, data and constant sections. When this optimization is activated, the linker tries to distribute the objects (function, variables, constants) across the different memory areas in the most optimal way. For functions, the linker analyzes the application call tree to determine optimal distribution across the different banks.
For example, suppose this is your application call tree:



All functions are called using FAR calling convention (instruction CALL).
After analyzing the application call tree, the linker distributes the function, as shown in the next diagram, between the different banks because it is more efficient.



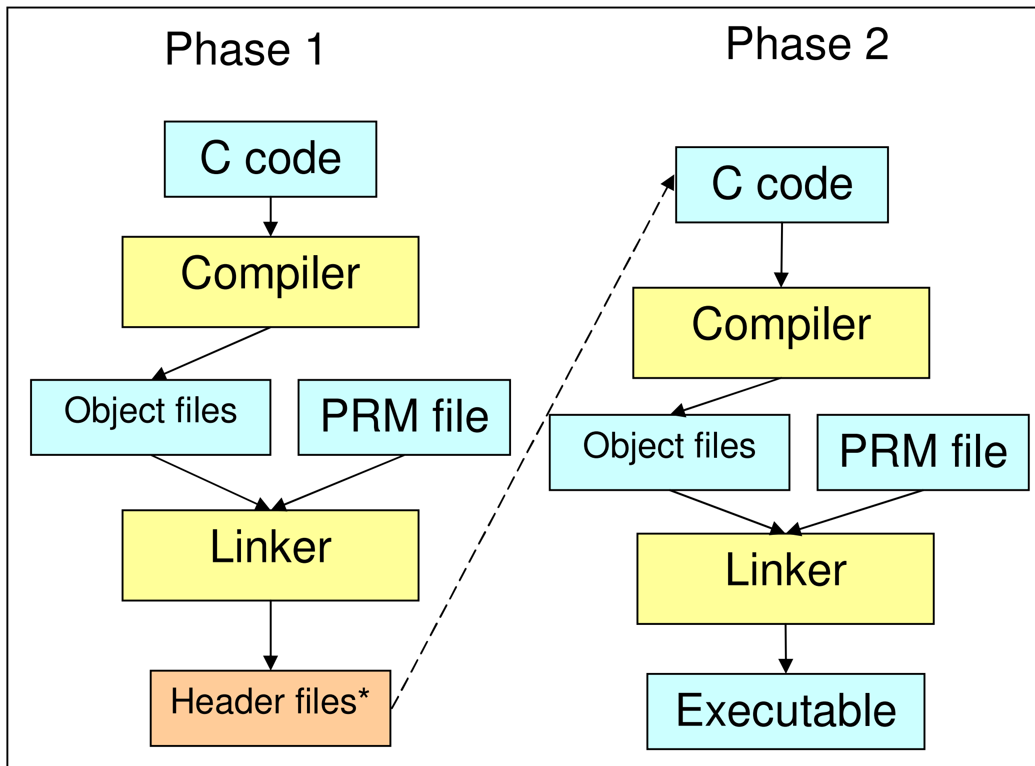
All functions in red can be called using the near calling convention (JSR). This will save code size as well as execution time.

For variables and constants, the linker decides on optimal bank distribution according to the object size and number of time the object is referenced in the application.

How MemoryBanker Works

A two path compilation/linking cycle is needed to achieve optimization.

- Phase 1. During Phase 1 the compiler does the initial compilation and the linker writes the optimal distribution information into new header files.
- Phase 2. During Phase 2, the compiler recompiles sources with the new include files. The linker creates optimal distributed application. The final application is then generated by the linker.



NOTE

Two additional files (except header files) can be generated as the output from the Phase 1: the option file containing specific compiler options for Phase 2 and text file containing the names of the libraries to be used in Phase 2. Advanced users can use these options since they are available for the command line projects only. (See From the Command Line)

When to use the MemoryBanker

In general the MemoryBanker optimization is applicable only for projects where the objects (data or code) do not fit into 16-bit address space. The MemoryBanker can not optimize projects based on small memory model.

Inefficiency of Large Memory Model

Large memory model projects use FAR calling convention and FAR data access that requires page switching by default. It could often lead to inefficient code. e.g., Even the functions located at the

same memory page use FAR call instruction (CALL) to jump from one to another instead of shorter and faster JSR instruction.

Relationship to Memory Models (small, banked, large)

The best candidate for MemoryBanker optimization are the projects that use paged memory to store data/code/constants – typically applications based on custom memory model [1].
MemoryBanker code optimization is also applicable in the applications where data (constants or non-constants) are not banked, but code is spread into multiple code memory pages – banked memory model. Therefore, the MemoryBanker feature can be enabled for a new project only if "Custom memory model" or "Banked memory model" is selected in the Project Wizard.

Devices MemoryBanker Supports

Currently, the MemoryBanker applies to the S12X derivatives that includes MMC units with paging capability to support a global 8 Mbytes memory address space. S12X/S12XE family derivatives that take advantage of MemoryBanker are:

- S12XA
- S12XB
- S12XD
- S12XE
- S12XF
- S12XHZ
- S12XS

Guidance on Which Kinds of Applications Benefit the Most from MemoryBanker

MemoryBanker optimization efficiency depends on several factors, e.g., number of objects, relations between objects, number of calls/data accesses of the objects. In the table below, we list general examples of applications and their benefit from the MemoryBanker :

+	-
Applications that use large memory model in general	Applications that do not use paged memory - small memory model
Applications that often access data objects (variables/constants) or call the objects (functions) placed into several memory pages	Applications accessing paged data objects or contain functions calling other functions sporadically
Complex applications that include many relatively small functions spread into multiple memory pages	Applications that includes relatively big-sized functions with function size close to page size

[1]Custom memory model is in fact adjusted "Large memory model" . In contrast to the large memory model, it enables the MemoryBanker optimization and other memory specific options in order to optimize the application.

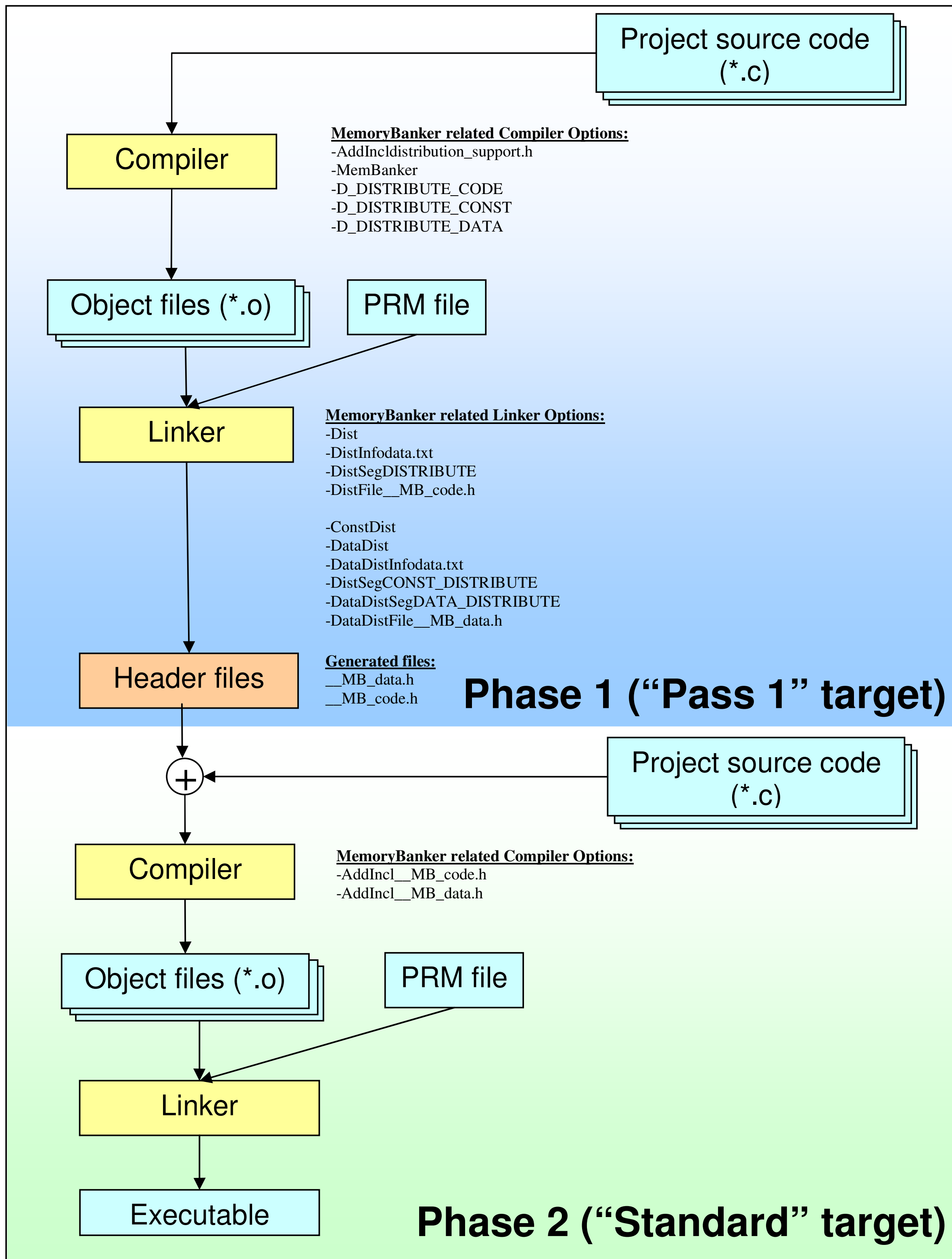
Process - How Can I Activate MemoryBanker for my Application?

This document describes how to create a new project or adjust an existing project to use MemoryBanker. Be aware that build time for the application will increase by a factor of two, since you will go twice through the compile + link process.

The document describes two different ways to activate the MemoryBanker: from the IDE, and from the command line.

From Wizard (IDE)

The Project Wizard can create a "Ready-to-go" project that enables the MemoryBanker feature based on selected options. The instructions below will show you how to create a new project with MemoryBanker. The project that is then created can be used as a template in cases where there is a need to adjust an existing IDE project that uses MemoryBanker. The picture below shows the project phases, component relations and build tools options added into the project by the Project Wizard.

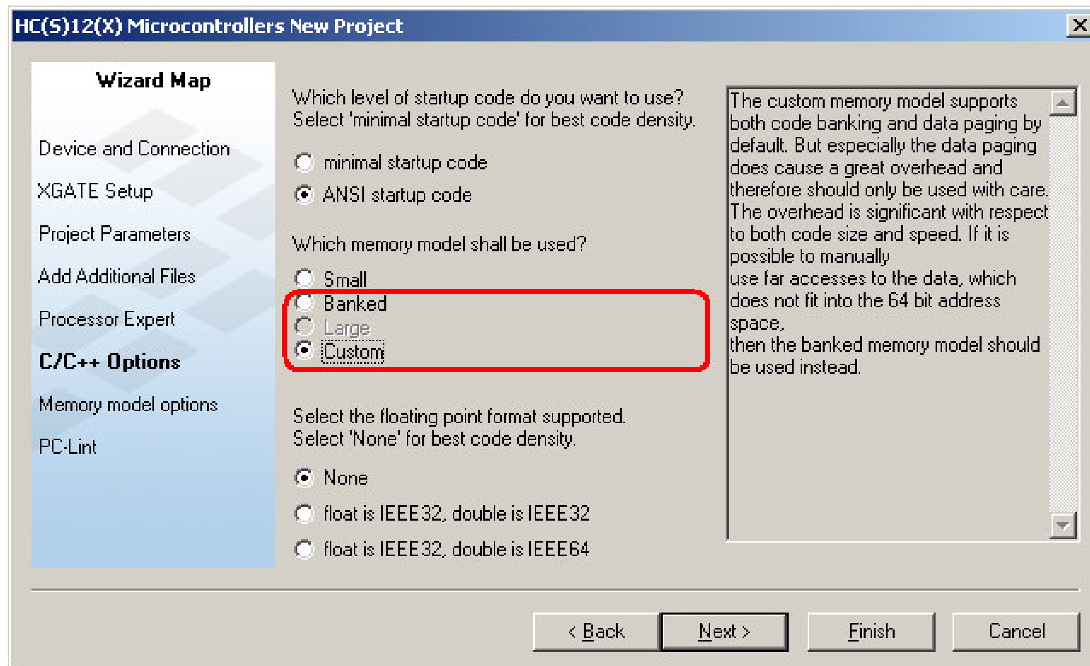

Project source code (*.c)
Compiler
MemoryBanker related Compiler Options:
-AddIncldistribution_support.h
-MemBanker
-D_DISTRIBUTE_CODE
-D_DISTRIBUTE_CONST
-D_DISTRIBUTE_DATA
Object files (*.o)
PRM file
Linker
MemoryBanker related Linker Options:
-Dist
-DistInfodata.txt
-DistSegDISTRIBUTE
-DistFile__MB_code.h
-ConstDist
-DataDist
-DataDistInfodata.txt
-DistSegCONST_DISTRIBUTE
-DataDistSegDATA_DISTRIBUTE
-DataDistFile__MB_data.h
Header files
Generated files:
__MB_data.h
__MB_code.h
Phase 1 (“Pass 1” target)
+
Project source code (*.c)
Compiler
MemoryBanker related Compiler Options:
-AddIncl__MB_code.h
-AddIncl__MB_data.h
Object files (*.o)
PRM file
Linker
Executable
Phase 2 (“Standard” target)

Step 1 – Create a New Project using Project Wizard

Select derivative and other parameters of the project according to project requirements with respect to MemoryBanker limitations. (See Limitations)

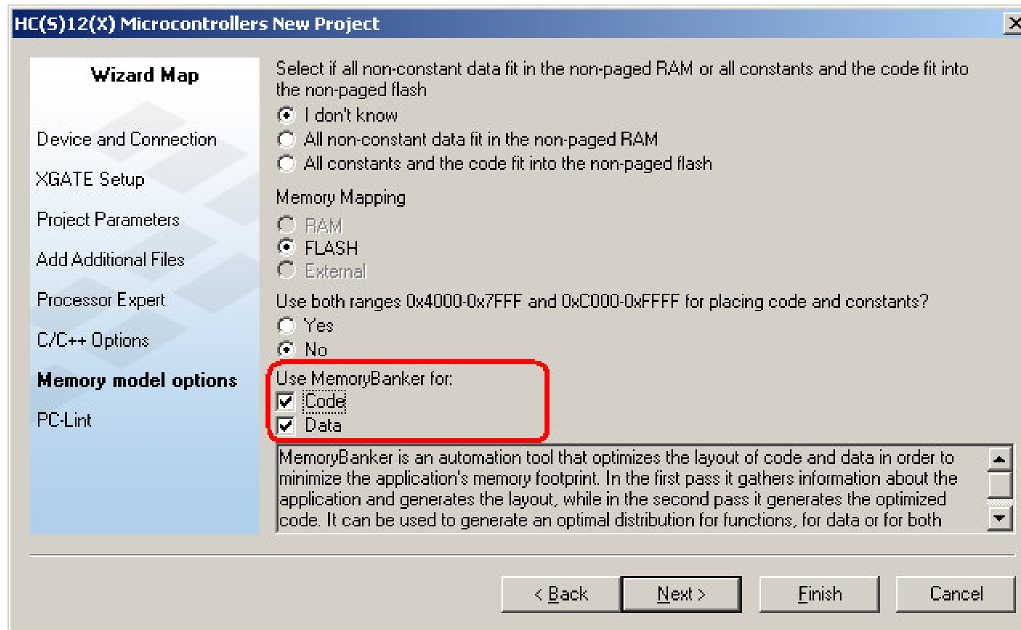
Step 2 – Select the Memory Model that Supports Paged Memory

The Custom memory model offers additional options that may generate more effective code in contrast to the Large memory model. The Banked memory model expects all data accesses to use non-banked memory. When using the Banked memory model, MemoryBanker optimization can be enabled only for code, not for data.



Step 3 – Enable the MemoryBanker Optimization

The Project Wizard allows MemoryBanker to be used for optimization of Code, Data (constant + non-constant) or both. There are two check boxes that control the MemoryBanker optimization. The availability of the “Data” option depends on the selected memory model and the memory mapping.

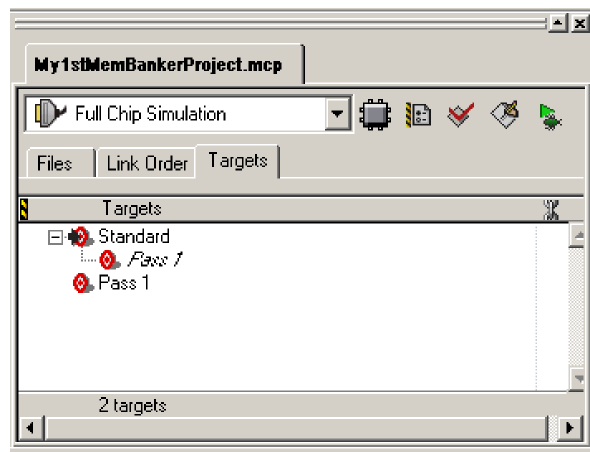


Step 3a – Adjust Custom Memory Model

There are several additional memory related options that could be used to increase optimization in special cases. (See Additional Optimizations if All Constant or All Non-Constant Data Fit in Local Memory)

Step 4 – Project is Ready to Use

The generated project includes specific structures, different to the project without MemoryBanker support. The target structure is the major difference. The project includes two targets named “Standard” and “Pass 1”. The “Pass 1” target is the sub-target of the “Standard” target. “Pass 1” is always built prior to “Standard” and represents Phase 1 of two pass build concept, whereas “Standard” target represents Phase 2 (see From Wizard (IDE)). In order to generate an executable file the “Standard” Target has to be selected.



NOTE

Project Wizard doesn't cover all possible combinations of MemoryBanker options. In the case of specific requirements other than available in the Project Wizard memory model dialog (e.g. enable MemoryBanker for constants only) it is necessary to adjust compiler/linker options manually for both Phase 1 and Phase 2 targets.

From the Command Line

Below are step-by-step instructions on how to modify an existing command line project to use MemoryBanker.
These instructions can also be used with any IDE projects. It may be more convenient to create new IDE project using project wizard and transfer the files from original to the new project.

Step 1 – Identify and Mark Objects that will be Distributed.

You have to define the name of one code section, one data section and one constant section that will be distributed over a set of memory pages. Let us suppose, you want to distribute the code section DISTRIBUTE, the constant section CONST_DISTRIBUTE and the data section DATA_DISTRIBUTE.

Step 1.1 – Identify and Mark Functions that will be Distributed.

Add the following line in front of each function that will be distributed by the MemoryBanker:

```
#pragma CODE_SEG DISTRIBUTE
```

Activating bank distribution for the whole application can easily be done by putting the pragma in a header file, which is included in each source file within the application. You can use the option -AddIncl to add a header file to the list of included files. CodeWarrior™ already has such a header file named "distribution_support.h" prepared and it is available in the directory {Install}\lib\hc12c\include\distribution_support.h, where {Install} refers to your CodeWarrior installation directory. This file is common for code/data/constant distribution.
To activate code distribution within "distribution_support.h" the macro _DISTRIBUTE_CODE needs to be defined. (E.g. compiler option -D_DISTRIBUTE_CODE can be added)

At this point it is important to identify functions in the application that cannot be distributed over the available memory pages.
Any function that must be allocated in non banked memory should not be placed in the DISTRIBUTE section. This is the case for functions attached to the CPU vector table, run time support functions …
If you have such functions in your application you need to make sure they stay in NON_BANKED memory and that you switch back to the DISTRIBUTE section after that.

This can be done as follows:

```
#pragma push
#pragma CODE_SEG __NEAR_SEG NON_BANKED
void myInterupt(void) {
  /* Interrupt code here*/
}
#pragma pop
```

In the same way, functions which are invoked from an assembly source file or from within a library module should not be placed in the DISTRIBUTE section.

Step 1.2 – Identify and Mark Variables that will be Distributed.

Add following line in front of each variable that shall be distributed by the MemoryBanker:

```
#pragma DATA_SEG DATA_DISTRIBUTE
```

Activating bank distribution for the whole application can easily be done by putting the pragma in a header file, which is included in each source file within the application. You can use the option –AddIncl to add a header file to the list of included files. CodeWarrior™ already has such a header file named "distribution_support.h" prepared and it is available in the directory {Install}\lib\hc12c\include\distribution_support.h, where {Install} refers to your CodeWarrior installation directory. This file is common for code/data/constant distribution.
To activate data distribution within "distribution_support.h" the macro _DISTRIBUTE_DATA needs to be defined. (E.g. compiler option -D_DISTRIBUTE_DATA can be added)

At this point it is important to identify variables in the application that cannot be distributed over the available memory pages.
Any variable that must be allocated in non banked memory should not be placed in the DATA_DISTRIBUTE section.
If you have such variables in your application you need to make sure they are defined in another section and that you switch back to DATA_DISTRIBUTE section after that.

This can be done as follows:

```
#pragma push
#pragma DATA_SEG MY_DATA
int criticalData;
#pragma pop
```

Any variable that is defined in a C source file and accessed within an assembly source file or within a library module must not be placed in DATA_DISTRIBUTE section.

Step 1.3 – Identify and Mark Constants that will be Distributed.

Add following line in front of each variable that shall be distributed by the MemoryBanker:

```
#pragma CONST_SEG CONST_DISTRIBUTE
```

Activating bank distribution for the whole application can easily be done by putting the pragma in a header file, which is included in each source file within the application. You can use the option –AddIncl to add a header file to the list of included files. CodeWarrior™ already has such a header file named "distribution_support.h" prepared and it is available in the directory {Install}\lib\hc12c\include\distribution_support.h where {Install} refers to your CodeWarrior installation directory. This file is common for code/data/constant distribution.
To activate constant distribution within "distribution_support.h" the macro _DISTRIBUTE_CONST needs to be defined. (E.g. compiler option -D_DISTRIBUTE_CONST can be added)

At this point it is important to identify constants in the application that cannot be distributed over the available memory pages.
Any constants that must be allocated in non banked memory should not be placed in the CONST_DISTRIBUTE section.

If you have such variables in your application you need to make sure they are defined in another section and that you switch back to CONST_DISTRIBUTE section after that.

This can be done as follows:

```
#pragma push
#pragma CONST_SEG MY_CONST
Const int criticalData = 0x2355;
#pragma pop
```

Any constant that is defined in a C source file and accessed within an assembly source file or within a library module must not be placed in CONST_DISTRIBUTE section.

Step 2 – Adjust Your PRM File to Enable Bank Distribution

The linker PRM file needs to be adjusted to enable bank distribution.
All memory areas which shall contain code (usually Flash and EEPROM memory) must be assigned the attribute IBCC_FAR or IBCC_NEAR.

Attribute	Description
IBCC_FAR	Inter-bank calling convention far. Applies to a segment of paged memory.
IBCC_NEAR	Inter-bank calling convention near. Applies to a segment of non-paged memory.

All memory area which shall contain data or constants (usually RAM, Flash and EEPROM memory) must be assigned the attribute DATA_FAR or DATA_NEAR.

Attribute	Description
DATA_FAR	Symbols are accessed using global addressing mode. Applies to segment of paged memory.
DATA_NEAR	Symbols are accessed using extended addressing. Applies to segment of non-paged memory.

You can find below an example of a PRM file's SEGMENTS block enabling bank distribution. See an example of a .prm file enabling MemoryBanker under (CodeWarrior_Examples)/HCS12X/ HSC12X_MemoryBanker_cmdline\prm\example.prm:

```
SEGMENTS
      RAM           = READ_WRITE  DATA_NEAR            0x2000 TO   0x3FFF;
      NEAR_DATA_ROM = READ_ONLY   DATA_NEAR IBCC_NEAR  0x4000 TO   0x4100;
      NEAR_CODE_ROM = READ_ONLY   DATA_NEAR IBCC_NEAR  0x4101 TO   0x7FFF;
      ROM_C000      = READ_ONLY   DATA_NEAR IBCC_NEAR  0xC000 TO   0xFEFF;
      RAM_F0        = READ_WRITE  DATA_FAR           0xF01000 TO 0xF01FFF;
      PAGE_C0       = READ_ONLY   DATA_FAR IBCC_FAR  0xC08000 TO 0xC0BFFF;
      PAGE_C1       = READ_ONLY   DATA_FAR IBCC_FAR  0xC18000 TO 0xC1BFFF;
END
```

Once you have added the above near and far attributes, you then need to specify in the PLACEMENT block where you want to place the DISTRIBUTE, CONST_DISTRIBUTE, DATA_DISTRIBUTE sections. In the PLACEMENT block you need to use the keyword DISTRIBUTE_INTO to place a section which will be distributed.
Sticking with our example above the placement block will look as follows:

```
PLACEMENT
      _PRESTART,
      STARTUP,
      ROM_VAR,
      STRINGS,
      VIRTUAL_TABLE_SEGMENT,
      NON_BANKED,
      COPY                  INTO ROM_C000;
      DEFAULT_ROM           INTO PAGE_C0, PAGE_C1;
      SSTACK,
      DEFAULT_RAM           INTO  RAM;
      DISTRIBUTE            DISTRIBUTE_INTO NEAR_CODE_ROM, PAGE_C0, PAGE_C1;
      CONST_DISTRIBUTE      DISTRIBUTE_INTO NEAR_DATA_ROM, PAGE_C0, PAGE_C1;
      DATA_DISTRIBUTE       DISTRIBUTE_INTO RAM, RAM_F0;
END
```

At least one segment within the each distribution placement (DISTRIBUTE/ CONST_DISTRIBUTE/ DATA_DISTRIBUTE) has to be NEAR otherwise the MemoryBanker optimization does not work and the linker issues an error.

Step 3 – Get the Linker to Evaluate Optimal Distribution

In order to get the linker to generate information for optimal bank distribution, you need to add the following options to your first pass linker options:

Option	Description
-Dist	Enable bank distribution for functions
-DistSeg	Specifies the name of the code section we want to distribute
-DistFile	Specifies the name of the optimization file for code. This is the file the compiler will use in the second pass
-DistInfo	Optional. Specifies the name of the Distribution information file for code. If specified the linker will generate a text file where it will document the gain in code size with the bank distribution
-ConstDist	Enable bank distribution for constants
-ConstDistSeg	Specifies the name of the constant section we want to distribute
-DataDist	Enable bank distribution for variables
-DataDistSeg	Specifies the name of the data section we want to distribute
-DataDistFile	Specifies the name of the optimization file for data and constants. This is the file the compiler will use in the second pass
-DataDistInfo	Optional. Specifies the name of the Distribution information file for data and constants. If specified the linker will generate a text file where it will document the gain in code size with the bank distribution

Except the options above there are the specific options that control if the option file containing additional compiler options for Phase 2 and the file containing the names of the libraries to be used in Phase 2 will be generated:

Option	Description
-Options	Enable generation of the compiler options for Phase 2
-OptionFile	Optional. Specifies the name of the option file
-LibOptions	Enable generation of the file that contains the names of the libraries to be used in Phase 2
-LibFile	Optional. Specifies the name of the file with the libraries names for Phase 2

For our example, the following options must be added to the linker command line:

```
-DataDist -DataDistInfodata.txt -DataDistSegDATA_DISTRIBUTE -
DataDistFiledata.h -ConstDist -ConstDistSegCONST_DISTRIBUTE -Dist -
DistInfocode.txt -DistSegDISTRIBUTE -DistFilecode.h
```

This will:

- Enable data distribution
- Distribute variables allocated in section DATA_DISTRIBUTE
- Generate a file data.h containing input data for the second pass build.
- Generate a file data.txt containing information about gain in code size for the optimization.
- Enable constant distribution
- Distribute constants allocated in section CONST_DISTRIBUTE
- Enable code distribution
- Distribute functions allocated in section DISTRIBUTE
- Generate a file code.h containing input data for the second pass build.
- Generate a file code.txt containing information about gain in code size for the optimization.

Step 4 – Re-Compile Application using Result of Step 3

Now you need to recompile your application, including the new header files generated from Step 3. In this purpose, you just need to include the optimization file in each source file within the application.
This can be done adding a –AddIncl option to your command line
For our example, the option we need to add to the compiler command line is:

```
-AddInclcode.h -AddIncldata.h
```

All the other code/data/constant distribution related options used during Phase 1 compilation (e.g. -AddIncldistribution_support.h , -D_DISTRIBUTE_CODE, …) are not used in Phase 2.

As the MemoryBanker might place functions which are implemented next to one another on a different page, you also need to disable the optimization which replaces a JSR op code by BSR.
So it is recommended to add the option –Onb=b to your compiler command line as well.
The compiler option file generated as the result of the options –Options and –OptionsFile should be used during Step 4.

Step 5 – Get Final Executable File

Finally you need to re-link your application without the –Dist options to generate the final executable file.
The file containing a list of libraries generated as a result of the options –LibOptions and –LibFile should be added using the options –ReadLibFile and -P2LibFile.

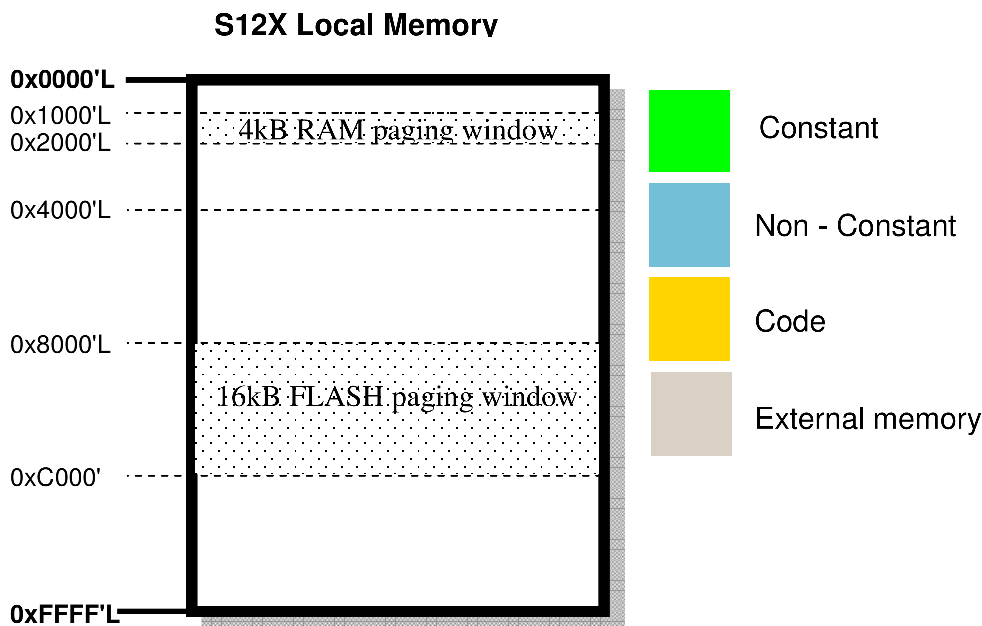
You find some example projects using the MemoryBanker in your {Install}
\(CodeWarrior_Examples)\HCS12X\HSC12X_MemoryBanker_cmdline directory where {Install}
refers to your CodeWarrior installation directory.

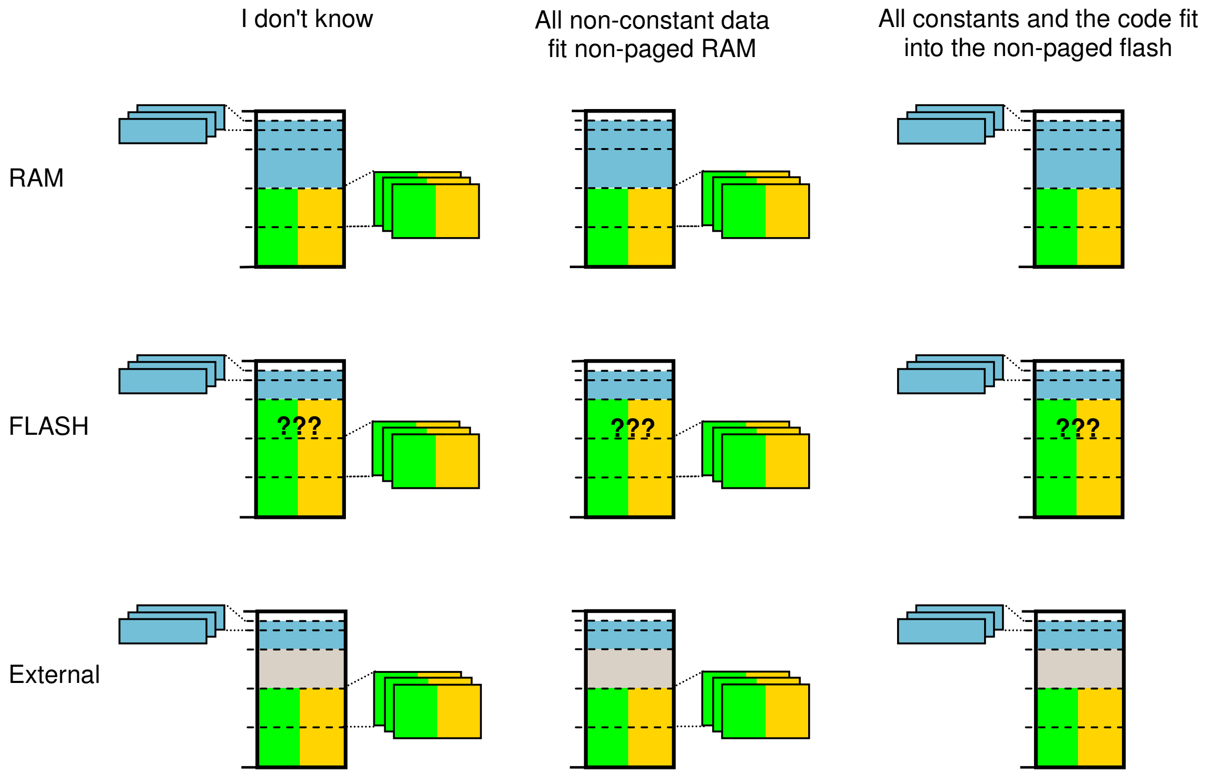
Additional Optimizations if All Constant or All Non-Constant Data Fit in Local Memory

CodeWarrior introduces new memory options that may additionally optimize memory accesses in specific cases (e.g. not use non-constant data paging if not necessary, use BSR instead of JSR if possible etc.). These options can be selected based on the code, constants and non-constant (variables, stack...) data estimations when an application is being created.

NOTE

MemoryBanker can be enabled only in the case where the Memory Map is set to FLASH (see the screenshot Step 3 – Enable the MemoryBanker Optimization).





??? – Depends if the option “Use both ranges 0x4000-0x7FFF and 0xC000-0xFFFF for placing code and constants” selected. The selection influences the linker parameter file. The .PRM file than contains both ranges in the placement block or just 0xC000-0xFFFF block.

If all constants (or all non-constant data) fit in local memory, MemoryBanker can force all accesses to constants (or all non-constant data) to be performed as accesses to near data. This feature is not integrated in the IDE. So it can only be used from the command line. The linker can automatically generate in the Phase 1 some compiler options for the Phase 2: -ConstQualiNear, -NonConstQualiNear or -Mb. (See Step 3 – Get the Linker to Evaluate Optimal Distribution) The options should be added to the rest compiler options for the Phase 2. These options are generated only when all the constant and/or non-constant data fits into the local memory map.

NOTE

As a consequence of automatic compiler option generation in the Phase 1 the libraries have to change if any of these option is generated. Linker is able to accomplish it in automated manner. (See the Step 3 – Step 5 sections in From the Command Line)

If one of the options is generated, then even through-pointer accesses are optimized, but not when they are used as function parameters. This is because it is quite a common practice among programmers to pass pointers to constant data as pointers to non-constant (and vice-versa). The only

way to get parameter passing optimized is still adding a __near qualifier, which means that the user takes full responsibility about what is actually passed to the function.

Converting Existing IDE Projects to Use MemoryBanker

First consider whether your existing application can benefit from the MemoryBanker before starting conversion.
Basically, there are two ways to enable MemoryBanker optimization on an existing project:

1. Modify the existing project to create a specific project structure (IDE or makefile) and configure the MemoryBanker settings manually.
2. Create a new project using the CodeWarrior Project Wizard and transfer all the files from the original project into the new one (IDE projects only).

Since the MemoryBanker requires significant changes in .prm file as well as changes in compiler/linker settings of two targets (Phase 1 and Phase 2) it's recommended to create a new project that enables MemoryBanker optimizations, instead of trying to modify the original one.
The advantage of this approach is that it allows easier transfer of projects created in previous versions of CodeWarrior.

Modifying the Memory Model of a MemoryBanker Project

During project development you can reach the point that the memory model and options selected are no longer suitable for your application. For example, the size of constants or variables do not fit into non-paged memory. Linker error messages usually occur in such cases.

The easiest way to change the memory model for an IDE based project is:

1. Create a completely new project using the Project Wizard, and configure the memory model and memory settings according to the updated object sizes estimations.
2. Transfer all the custom source files and custom build tools settings from the existing project into the new project.

Changing the memory model in Command Line projects (see From the Command Line) or direct modifications of existing IDE projects are more complex since all settings needs to be adjusted manually for both compilation phases. Below is the list of main actions that need to be considered if the memory model and/or memory options are being changed manually:

- Compiler settings of pass 1 target (named "Pass 1" by default) - adjust compiler options according to the new memory model and memory model settings.
- Compiler settings of pass 2 target (named "Standard" by default)- adjust compiler options according to the new memory model and memory model settings.
- ANSI Library – there are several different pre-compiled ANSI libraries. The one that matches with the new memory model settings shall be added into the project instead of the original one. (see {Install} lib\hc12c\readme.txt for more information about the available libraries)
- If any other libraries (e.g. 3rd party libraries) are included in the project they should be recompiled using the same memory model option (-Ml or -Mb) that the application uses. The MemoryBanker optimization cannot be applied to libraries where there is no source code available.

NOTE

To make this process easier you can generate a new temporary project using the Project Wizard. Select the new memory model options through the wizard dialogs. Finally, replace or add the compiler\linker options from the temporary project into your existing project, replacing the original settings. You can also check the ANSI library added by the wizard and use the same library in your project.

Removing MemoryBanker from a Project

Follow the steps below to disable MemoryBanker feature:

1. remove "Pass 1" sub-target in IDE project or remove Pass 1 compiler/linker lines in the makefile to disable multi pass compiling.
2. remove all MemoryBanker related compiler/linker options related to remaining "Standard" target (see the options in section From Wizard (IDE))

Limitations

- MemoryBanker can only be used when you have source code for the application. Optimization cannot be applied to third party libraries where you only have a .lib file.
- Functions whose addresses are inserted in the CPU vector table must be allocated in non-banked memory.
- Functions put into a DISTRIBUTE segment do not have additional calling convention information. E.g. the following is illegal:

```
#pragma CODE_SEG DISTRIBUTE
void near MyNearFunction(void) { …
```

- Special care should be taken when you are mixing C and assembler in the application. Symbols (variable, constant, function) defined in a C source file and accessed from an assembly source file should not be placed in a DISTRIBUTE section.
- MemoryBanker does not work for projects including ProcessorExpert source files.
- MemoryBanker does not work for OSEKturbo projects.
- At the moment the MemoryBanker does not work in multi-core XGATE projects. This is limitation of V5.0.
- MemoryBanker applies only to the S12X derivatives that includes MMC units with paging capability to support a global 8 Mbytes memory address space. Currently only the S12X family derivatives could take advantage of MemoryBanker (S12XA, S12XB, S12XD, S12XE, S12XF, S12XHZ, S12XS)

Checking Object Distribution and Amount of Memory Saved

When you are using the MemoryBanker, the linker is producing several text files allowing you to check:

- How did the linker distribute the objects among the available pages
- The amount of code saved due to this optimization

Checking Object Distribution

The files specified in option –DistFile and –DataDistFile contain information on how the linker distributed the code and data among the available pages of memory.

Here is an example of a distribution file describing code distribution:

```
#pragma REALLOC_OBJ "DISTRIBUTE0" main __INTERSEG_CC__
#pragma REALLOC_OBJ "DISTRIBUTE0" ("calc_banked.o") IsUnaerOp __NON_INTERSEG_CC__
#pragma REALLOC_OBJ "DISTRIBUTE0" ("calc_banked.o") Illegal __NON_INTERSEG_CC__
#pragma REALLOC_OBJ "DISTRIBUTE0" ("calc_banked.o") ReadInt __NON_INTERSEG_CC__
```

In the snippet above,

- __INTERSEG_CC__ identifies a function invoked using a CALL instruction (far calling convention).
- __NON_INTERSEG_CC__ identifies a function invoked using a JSR (Near calling convention).

Here is an example of a distribution file for data and constants

```
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  gint __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  hint __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  fint __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  iint __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  d1 __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  c1 __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE0"  e1 __NON_FAR_DAC__
#pragma REALLOC_OBJ "DATA_DISTRIBUTE1"  a1 __FAR_DAC__
```

In the snippet above,

- __FAR_DAC__ identifies a variable or constant allocated in banked memory area (Accessed using global addressing mode).
- __NON_FAR_DAC__ identifies a variable or constant allocated in non banked memory area (Accessed using extended addressing mode).

Checking Amount of Memory Saved by the Optimization

The files specified in option – DistInfo and –DataDistInfo contain information on code size saved through the MemoryBanker optimization.

Here is an example of info file describing code saved when optimizing code distribution:

```
main  old size: 97  optimized size: 92  calling convention: far
[calc.o]  IsUnaerOp  old size: 16  optimized size: 16  calling convention: near
[calc.o]  Illegal  old size: 8  optimized size: 8  calling convention: near
[calc.o]  ReadInt  old size: 146  optimized size: 146  calling convention: near
EVAL_Eval  old size: 262  optimized size: 252  calling convention: near
```

In the code snippet above, <old size> - <optimized size> provides you information about the amount of memory saved due to MemoryBanker optimization.