

# Moving from C6805 to C6808

Byte Craft Limited  
A2-490 Dutton Drive  
Waterloo, Ontario  
Canada N2L 6H7

Phone: 519 888 6911 • fax: 519 746 6751

Email: [info@bytecrafter.com](mailto:info@bytecrafter.com) • <http://www.bytecrafter.com>



The information in this document is intended for use  
with the Byte Craft Limited C6808 Code Development System.

## Revision history:

Version	Date	Initials	Description
0.1	14/11/06	KZ	Initial version

## Introduction

The Motorola/Freescale HC05 architecture has had a long life in embedded systems. However, the HC08, HCS08, and new RS08 architectures have superceded HC05. Your upgraded design needs to make use of one of these newer parts, and can take advantage of the enhanced instruction sets (in HC08 and HCS08) or the reduced size and cost (of RS08).

Developers porting programs generally see a simpler and more effective transition for C programs than that needed for assembly programs. The compiler completely re-evaluates its programs at each recompilation, taking advantage of every resource the target has to offer. Assembly programs written for HC05 can be reassembled for all three current architectures, but with different concerns for each:

- HC08 and HCS08 are binary-compatible but not instruction-cycle compatible with HC05, and legacy assembly programs won't take advantage of the new processor features.
- RS08 is not binary-compatible, so programs will need some rework even with the pseudo-instructions specified to implement HCS08-compatible behaviour.

Porting even a C program to a new target part requires rewriting source code: it's just common sense. On the other hand, our customers have cited various reasons why touching the source code of an existing project is undesirable: regulations and certifications, license terms with clients or vendors, and budget constraints are just a few. As much as possible, our suggestions will have no impact on your source code.

We've put together some C-specific tips on moving an embedded project from C6805 and HC05 to C6808 and HC08/HCS08/RS08. This document complements Freescale's application notes on moving from HC05 to HC08; they are listed at the end.

## Migrating to the new part

### *One last compile with C6805*

Start with a known baseline: make sure you can compile old code with C6805 and get the executable you expect. Use the listing file to identify the version of the compiler that generated the latest builds, and

Migrating to the new part confirm the generated code. Different versions of the compiler can generate slightly different code as new optimizations are introduced. Likewise, abandoned revisions to the project may have altered the source code of a previously released design.

Ensure that all your libraries can be recompiled from source. *Newer compilers cannot use object files or libraries from previous versions.*

**Cycle counts:** The cycle counts for instructions differ between HC05 and HC08/HCS08. For convenient comparison between old and new generated code, enable the cycle count feature in C6805. Add the directive

```
#pragma option k0;
```

to the source file and recompile with C6805. This will cause the compiler to insert a field in the listing file that shows how many cycles each instruction takes. Rename the listing file to indicate it was generated with C6805. Leave the directive in when compiling under C6808. Compare the new listing file with the previous one when recalibrating loops or verifying other time-dependent behaviour.

## Compare old and new peripherals

Get to know the differences between peripherals on the old and new parts. In a few cases, code for peripheral access will compile without change. However, in most others some rewriting will be necessary. We have seen port names change from one part to the next (with otherwise identical peripherals), and bit flags and fields move around within I/O registers on different parts.

## Resolving naming conflicts

In our device header files, we adhere very closely to Freescale documentation for each part. Identifiers for I/O registers and their bits follow Freescale's data sheets. On occasion, Freescale names two I/O ports or bit fields the same. In this case, we must rename symbols to avoid duplicate declarations. Such changes are noted in the header file comments.

When you redesign a program for a new part, the device header files may declare identifiers that conflict with those in your existing code. For instance:

```
/* In new device header file */
#pragma portrw FLCR @ 0xFE08; /* Flash control register */

/* In existing software */
//FLCR: Fine level control read command
#define FLCR 0xAA

/* ... */

PORTA = FLCR;
```

The compiler will issue the error “Inconsistent duplicate macro definition” when compiling this example.

It's possible to change the device header file to resolve the conflict, but this will require you to repeat the same change with each new Code Development System upgrade. In some environments this will lead to problems with other projects. It's best to put a shim in the source code to re-declare the conflicting new peripherals for the current project only.

Add a re-declaration block similar to this:

```

/* In new device header file */
#pragma portrw FLCR @ 0xFE08; /* Flash control register */

/* Redefinition block */
#pragma portrw NEWFLCR @ & FLCR;
#undef FLCR

/* In existing software */
//FLCR: Fine level control read command
#define FLCR 0xAA

/* ... */

PORTA = FLCR;

```

You can now use the existing source, and the new peripheral, without substantial revisions. Place the re-declaration block in a header file if you anticipate using BLink: the new declarations must accompany the device header file that BLink needs to read as part of the linker command file.

### Watch out for different optimizations

The move from C6805 to C6808 will introduce new optimizations into generated code that may conflict with your original intentions for the program. If you have used any of the following, check the generated code extra carefully:

- **Pointers and arrays.** Compilers can optimize pointers and indexes to overcome hardware limitations. Bank switching on RS08 is a particular concern because it didn't exist before in '08 parts. Watch out for indirect accesses in your code that won't generate the expected effective address.
- **Self-modifying code.** It is tempting to trust that self-modifying code will work as expected because the HC(S)08 is binary compatible with the HC05. However, new optimizations used by the new compiler may change the code generated for (among other things) flow control. The compiler will replace subroutine calls with jumps or branches where possible to save program memory and stack space. Don't assume that your code's `RTS` will break the return stack, however.

The compiler makes use of the H register in HC(S)08 in long operations. Your interrupt service routines may need to preserve its state if they perform long operations themselves. Use the `PSHH()`; and `PULH()`; intrinsics to do so.

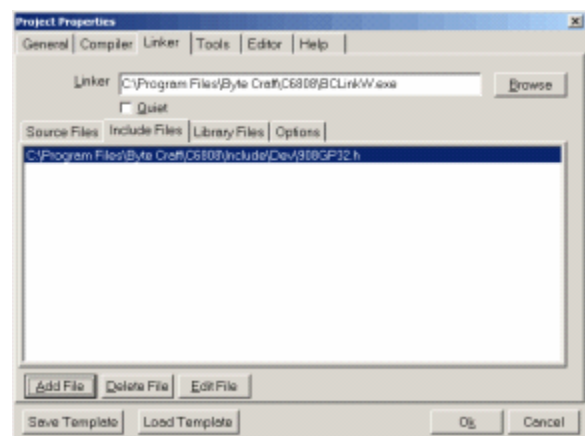
## Step-by-step guide

This is a list of steps to take to recompile an HC05 program for a HC08, HCS08, or RS08 part.

At each step, recompile to check if the next step is necessary.

1. Change the device header file `#include` in your main program to use the HC(S)08/RS08 header for the new target part.

If you're using BLink, make the change in every source file/library file, and in the settings for BLink. Choose **Project|Properties**, select the



Let BLink know about the new device header file.

**Linker** tab, and add the file in the **Include Files** list.

2. Look in your program for symbols duplicated in the device header file. Use new declarations to capture their addresses and then `#undef` them.

If you're using BLink, ensure these declarations are available to the linker, probably through a header file `#included` after the device header.

3. Check for global symbols unexpectedly hidden by local ones within C functions. If you've declared an identifier within a function written for HC05, one that also appears as a global in an HC(S)08/RS08 device header file, the global definition will be hidden in that function.
4. Check the code written for each peripheral against the data sheet. Mask option registers, system integration modules, timers, and COP timers will show the greatest changes between generations of parts.
5. Migrate configuration options from the old part to the new part. Configuration options from the original program probably used the `#pragma mor` or `#pragma mori` directives. These directives are still available, but deprecated. Instead, use these more general directives:

- a. For configuration registers in Flash (with no RAM components), use `#pragma fill`:

```
#pragma fill @ CONFIG_x = <configuration value>;
```

- b. For configuration registers in RAM, simply assign a value, either in the user-supplied function `__STARTUP()` or very early in `main()`:

```
void __STARTUP(void)
{
    CONFIG_x = <configuration_value>;
}
```

## For future reference

If you've retargeted an HC05 design to HC(S)08/RS08, the same thing may happen again in the future with another new architecture. Here are some hints on how to simplify the next transition.

- If you have any left-over Flash, embed the time of compilation in the executable itself with the `__TIME__` and/or `__DATE__` macros:

```
#pragma memory ORG @ <somewhere out of the way>;
const char * __product_compiled = __TIME__;
```

Do this early in the development process: backups will indicate to future developers that the date or time will probably appear in the final executable, and the rough location. They can then match the information in the executable against the timestamp on files recovered from archives. If the two are very different, more research will be required to recover the correct files.

- Use device-independent I/O methods. We have in the past generated I/O libraries that abstract input and output functions away from specific bit values. Consider the specific instance of data direction for bidirectional I/O ports:

```

/* Macros for use in DDR() and DDR_MASKED() assignments */
#define OOOOOOOO 0b11111111
#define OOOOOOOI 0b11111110
#define OOOOOOIO 0b11111101
#define OOOOOOII 0b11111100
//and so on

/* Mask macros for use in DDR_MASKED() assignments */
#define _____ 0b00000000
#define _____C 0b00000001
#define _____C_ 0b00000010
#define _____CC 0b00000011
//and so on

/* Macros for adjusting the data direction */
#define DDR(PORT,VAL) \
    if(&PORT==&PORTA) DDRA=VAL; \
    else if(&PORT==&PORTB) DDRB=VAL

#define DDR_MASKED(PORT,MASK,VAL) \
    if(&PORT==&PORTA) DDRA=((DDRA&~MASK)|(VAL&MASK)); \
    else if(&PORT==&PORTB) DDRB=((DDR&~MASK)|(VAL&MASK))

```

If the underlying hardware changes, both the `DDR()` macros and the I/O definitions can be changed to leave the source code intact.


## References

### *Freescal Application Notes*


AN1218/D: HC05 to HC08 Optimization.

AN2717: M68HC08 to HCS08 Transition. A good look at the way peripherals can change between devices.

**The '08 has shrunk.  
Do the same for your code.**



Over twenty years of code generation experience goes into the C6808 Code Development System.



02A4  
02A6  
02A9

# C6808

**Code Development System  
for HC08, HCS08, and RS08**