BrianHetrick.com Application Note AN-3

A Real-Time Application Framework for the Freescale HC08 Microcontroller Family

Goal

The goal of this Application Note is to present a framework for hobby real-time application software on the Freescale HC08 family of microcontrollers, with particular reference to the MC68HC908QTxA and MC68HC908QYxA microcontrollers.

Introduction

Most microcontroller programs have a simple goal: to control or report on some specified set of conditions in the environment. The conditions of interest might be the temperature and pressure associated with an industrial process, the rotational speed of a device subject to varying loads, the rate of precipitation at a specific location, or so forth.

Although the overall mission of the microcontroller program might be simply expressed, the implementation of microcontroller programs is generally less simple. Generally the microcontroller must receive commands regarding the conditions to be maintained; must acknowledge its receipt of such commands; must periodically make status reports regarding the conditions being maintained; must take actions to cause the set of conditions to reveal themselves; must interpret a limited amount of information available from sensors to deduce the conditions; must perform such self-checking as is possible; and must do all of these simultaneously, or very nearly so.

Microcontroller programs do not generally have an exit condition. Traditional computer programs perform a specific computation and terminate their execution, or support a user activity and exit when the activity is finished. In contrast, microcontroller programs generally go on "forever." Although a microcontroller may be commanded to shut down a process it is controlling, the microcontroller itself must maintain its readiness to accept new commands to, for example, restart the controlled process. Generally the only way to cause a microcontroller program to cease processing entirely is to remove power from the microcontroller.

There is no requirement that microcontroller programs follow any particular outline. However, it is useful to extract common features and implement these separately as a framework or scaffolding. This permits a microcontroller program to comprise "plug in" modules to the framework. At one extreme, this approach yields a real time operating system. At the opposite extreme, this approach yields a few useful routines which wrap only the commonest operations. This latter approach is taken in this Application Note.

Framework Features

The framework described in this Application Note has the following features:

- It is usable across the entire line of Freescale's popular HC08 microcontrollers. It is optimized for the low-end MC68HC908QTxA 8-pin and MC68HC908QYxA 16-pin microcontrollers. These microcontrollers are perfectly suited to hobby use due to their DIP packaging, low pin count, high degree of peripheral integration, low hardware cost, and no-cost ANSI C development environment.
- It uses the popular CodeWarrior development environment for the HC08 microcontrollers.
- It gives the application program control of all setup and configuration options.
- It permits the application program to have three "levels:" interrupt servicing, interrupt associated processing, and background. Interrupt servicing can be limited to actions which must occur at the time of an interrupt. Interrupt associated processing consists of actions which must occur before

the next interrupt of that type. Background consists of actions that take place without regard to interrupt deadlines.

- It contains a main interrupt loop, analogous to the main event loop of an event-driven application. The Computer Operating Properly (COP) module is serviced as part of this loop.
- It consists of only two files, a header file `framework.h` and a code file `framework.c`. No configuration of the framework is necessary; the files need only be included in the CodeWarrior project to become effective.

Choice of Microcontroller

A microcontroller is generally a single package containing a small computer, some memory, some devices such as oscillators and counters, and digital and analog input/output (I/O) pins. There are many families of microcontrollers available: popular families include the 8051 architecture devices, ARM architecture devices, and the Atmel AVR devices, among others. Each of these has advantages relative to the others. Choosing one or another when designing a product can be a very difficult decision.

The requirements for a microcontroller used in a hobby, however, differ from those used in a product. Hobby circuits are typically hand-assembled on wireless breadboards, or hand-soldered on generally handmade printed circuit boards. SOIC (small outline integrated circuit) packages can reliably be soldered by a hobbyist with good eyesight and a steady hand; denser packages such as PGA (pin grid array) or TSOP (thin small outline package) effectively cannot be used in hand assembly. Of the commercially available package styles, the most friendly to breadboard prototyping and hand assembly is the DIP (dual in-line package).

A further consideration is the number of pins in the package. Microcontrollers communicate with their environment through I/O pins: each I/O pin provides a digital or analog input, or a digital or analog output, or some combination of these. While a microcontroller in a 40 pin package can sample or generate a large number of signals, a 40 pin package is both unwieldy and expensive. Smaller packages, such as 8 and 16 pin packages, are generally more appropriate for hobbyist use. This package size also generally implies an 8 bit processor, rather than a more computationally capable 16 or 32 bit processor.

A final consideration is the ease and expense of programming the device. An inexpensive or free ANSI C compiler, inexpensive or readily built programming hardware, and reprogrammable program memory in the device are all highly desirable.

After a review of available microcontrollers and their development environments, the author has identified the Freescale HC08 microcontrollers – in particular the MC68HC908QTxA 8 pin DIP and MC68HC908QYxA 16 pin DIP parts – as the most attractive for hobby use. These parts are $2 to $3 (quantity 1) retail, and recently about $0.5 (quantity 100) surplus, well within many hobby budgets. These von Neumann architecture, 1.5 to 8 binary kilobyte (KiB[1]) flash memory, 8 megahertz (MHz) bus microcontrollers can be programmed in C from a Windows PC with a serial port using no-charge software and an easily built, inexpensive circuit. These are not the most powerful microcontrollers available, or the least expensive, even when considering only those in 8 pin and 16 pin DIP packages. However, in the author's opinion, these particular devices hit a "sweet spot" of price, ease of use, and performance that makes them highly attractive for hobbyists.

Choice of Development Tools

Freescale offers the CodeWarrior series of development tools for its microcontrollers. The HC08 microcontrollers are supported by the CodeWarrior for Microcontrollers product. Of special note is the *Special Edition: CodeWarrior for Microcontrollers* edition of this product, which is available for no charge. This edition limits the object code resulting from C language modules to 32 KiB. However, the microcontrollers of interest in this Application Note have at most 8 KiB of program storage, making this limitation irrelevant.

The CodeWarrior for Microcontrollers product enables programming in both assembly language and ANSI C. Assembly language programs may be absolute or relocatable. Programs may also combine assembly language and C. The product also contains a sophisticated linker that allows fine-grained placement of storage and that removes unused storage and code from the final product.

1 "KiB," or "kibibyte," means 1024 bytes. The "Ki" is the International Electrotechnical Commission recommended prefix to ensure multiples of 2^{10}, 1024, are not confused with multiples of 10^3, 1000.

The CodeWarrior for Microcontrollers product is a complete turnkey application development tool suite and integrated development environment for the HC08 microcontroller line. It supports downloading programs to and debugging programs on the microcontrollers using the host computer's serial ports, USB ports, and so forth. The CodeWarrior suite was chosen as the development tool chain for the framework due to its support by the microcontroller manufacturer, its overall capabilities, its turnkey nature, and its no-cost availability.

There are other development tools for the HC08 series of microcontrollers. These tools include both commercial offerings from third parties and open source alternatives. It may be possible to adapt the framework described in this Application Note to these other development tools. However, this is not investigated in this Application Note.

Setup and Configuration Options

Microcontrollers in general, and the HC08 family of microcontrollers in particular, are highly integrated parts. In addition to a CPU and memory, they contain modules providing a variety of other services. These services can include event timing, watching for specific conditions appearing on specific I/O pins, providing analog to digital (A/D) conversion of voltage levels appearing on I/O pins, and so forth. Configuration and setup of these devices is frequently critical to system performance.

The framework described in this Application Note imposes no configuration or setup requirements on the application author. In particular, the Computer Operating Properly (COP) module can be enabled or disabled; any memory model can be chosen; and either ANSI or minimal startup code can be chosen. The framework provides the application initialization opportunities both before and after interrupts become enabled.

Application Levels

A rich source of complexity in real-time programming is the coordination between interrupt level processing and non-interrupt level processing. In general, only direct interrupt servicing is desirable at interrupt level. Frequently such servicing need consist of only a few instructions: setting an output bit to a predetermined state, recording the state of an input bit, or capturing a timer channel count or analog to digital (A/D) converter value. Such simple interrupt routines execute quickly and keep interrupt latency low.

Keeping the interrupt servicing simple, however, requires that associated processing occur in a timely fashion. Setting an output bit to a predetermined value requires the value to be predetermined: in particular, computing the value to be output at the time of the next interrupt must be complete before that interrupt occurs. Similarly, interpreting a captured timer channel count or an input bit state must be complete by the time of the next timer channel count capture or input bit capture.

Finally, there is processing which is related only tangentially to interrupts: composing a status message, for example. This essentially free-floating processing must be done while remaining responsive to the demands of interrupt servicing and interrupt associated processing.

The major purpose of the framework is providing a structure providing for these three levels, and a discipline by which application code can fit into these three levels. This is covered in more detail in the section, "Using the Framework."

COP Servicing

The COP module provides a watchdog timer for the application program. Depending on the value written to the CONFIG1 register, the COP will reset the CPU when either 8176 or 262128 bus cycles elapse since it was last serviced. With the default 3.2 MHz bus clock, these periods correspond to 2.5 milliseconds (ms) and 81.9 ms. With the maximum 8 MHz bus clock derived from an external 32 MHz clock, these periods correspond to 1.0 ms and 32.7 ms.

The framework services the COP at each pass through the interrupt associated processing loop. If the COP is disabled, servicing it services no purpose, but is harmless. If the COP is enabled, the application is responsible for requesting associated processing frequently enough to keep the COP from resetting the CPU.

Framework Implementation

The framework consists of two C language files: `framework.h` and `framework.c`. The header file `framework.h` contains all declarations needed by client applications. These declarations include prototypes both for the framework-supplied utility functions and for the client-supplied functions invoked by

the framework. This header file further includes the CodeWarrior-supplied header file `derivative.h`. This in turn includes the microcontroller-specific header file for the microcontroller to which the project is targeted. This header file defines symbolic names for all control and status registers for the CPU and microcontroller-specific peripheral devices. The `framework.h` header file should generally be included by all code files in the client application.

The `framework.c` file includes all definitions for the framework. This includes the `void main (void)` entry point. The entry point is invoked by the CodeWarrior-supplied startup code.The framework also provides several other routines that may be of use in the client application.

The client application must provide five functions with specific names and prototypes. The framework calls these functions in specific circumstances. Client applications use the framework by providing interrupt service routines for the interrupts they use, and providing these five specific function.

Using the Framework

Introduction

A client application of the framework generally consists of four parts:

- Initialization functions;
- Interrupt service functions;
- Interrupt associated processing functions; and
- Background processing functions.

Initialization is performed once, and includes enabling interrupts. Following initialization, the framework enters a loop waiting for interrupt associated processing to be requested. Client-supplied interrupt service functions execute transparently as their interrupts occur. When one of these interrupt service functions requests associated processing, the framework forwards this request to a client-supplied associated processing function. Finally, when all associated processing has been performed, the framework provides the client-supplied background processing function the opportunity to execute. This opportunity lasts until the background processing function indicates it has no more work to do, or until more interrupt associated processing has been requested.

Execution Levels

There are three levels at which application tasks may be carried out: at the interrupt service level, at the associated processing level, and at the background level. These levels differ in how they are scheduled and in the type of processing appropriate to each.

Interrupt service functions are dispatched by the CPU accepting an interrupt. These functions execute with interrupts disabled. Interrupts which would otherwise be dispatched while the interrupt service function is executing wait until the interrupt service function returns. This may increase interrupt latency, the delay between the time when the conditions giving rise to an interrupt occur and the time when the resulting interrupt occurs. Interrupt routines should generally include as little processing as possible, to avoid increasing the latency of other interrupts.

Interrupt associated processing functions are “follow-up” functions requested by the interrupt service functions. Their execution does not increase interrupt latency, so they need not execute particularly quickly. They generally need to complete before the next occurrence of the interrupt with which they are associated. Functions which require multiple occurrences of the associated interrupt can be implemented as state machines.

Finally, background processing functions are executed after all associated processing functions are complete. Background processing is assumed to be implemented as a state machine with short-duration states. While background processing does not increase interrupt latency, it does increase the latency of associated processing. Associated processing is dispatched by the framework, so the background processing function must return control to the framework at regular intervals. If there is no associated processing requested, the framework again invokes the background processing function.

The framework initializes the background state number to 0, passes this state counter to the first invocation of the background processing function, and uses the background processing function’s return value as the next state number. Should the background processing function return 0, the framework does not invoke the background processing function until after associated processing functions are next executed.

When the background processing function returns 0, indicating it is not to be immediately invoked again,

and there is no associated processing requested, the framework executes a WAIT instruction. This instruction halts program execution until an interrupt occurs.

Framework Provided API

The link between interrupt service functions and the interrupt associated processing functions consists of two parts: the interrupt service function requesting associated processing, and the framework forwarding that request to client software. The interrupt service routine requests associated processing through the `RequestFollowup` function. This function has the prototype:

```
void RequestFollowup (word flag)
```

The `flag` parameter to this function is provided to identify the associated processing desired. This parameter is intended to be a vector of boolean (bit) flags, each indicating a particular type of followup processing desired. The logical OR of these values is later passed to the client-supplied associated processing routine, where they can be used to select associated processing.

The use of a `word` (`unsigned int`) parameter provides 16 individual bit flags which can be associated with desired associated processing functions. In applications intended for microcontrollers with more than 16 interrupt sources, multiple interrupt associated processing functions might be identified by a single bit and disambiguated through other means.

The framework also supplies the following convenience functions for use in special circumstances:

`void DisableInterrupts (void)` – This function disables CPU interrupts. CPU interrupts remain disabled until one of `EnableInterrupts` or `WaitForInterrupt` is invoked. Generally, applications should disable CPU interrupts only to manipulate data structures also manipulated by interrupt service functions, and should immediately re-enable interrupts thereafter. CPU interrupts should be disabled for the shortest length of time possible.

`void EnableInterrupts (void)` – This function enables CPU interrupts. Generally, application should invoke this as soon as possible after invoking `DisableInterrupts`.

`void ResetCOP (void)` – This function resets the COP. Generally, applications should not call this function, but let the framework reset the COP.

`void WaitForInterrupt (void)` – This function enables CPU interrupts and waits until an interrupt occurs. Generally, applications should not call this function, but let the framework wait for interrupts when appropriate.

Client Provided API

An application using the framework must provide interrupt routines for all interrupts it enables, and five additional specific functions required by the framework. Three of these specific functions are initialization functions, as follows:

`void InitializeCPU (void)` – This application function is the first client-supplied routine invoked by the framework. It is responsible for performing all initialization required by the application before interrupts can be enabled. On the microcontrollers considered by this Application Note, such initialization frequently includes CPU configuration (`CONFIG1` and `CONFIG2`), OSC configuration (`OSCSC` and `OSCTRIM`), and I/O port configuration and initial values (`DDRA`, `PTAPUE`, `DDRB`, `PTBPUE`, `PTA`, `PTB`). This function executes with interrupts disabled.

`void InitializeProgram (void)` – This application function is invoked by the framework shortly after the `InitializeCPU` function returns. This function is responsible for performing all initialization required by the application before the first `WAIT` instruction is executed. The `WAIT` instruction halts execution until an interrupt occurs. On the microcontrollers considered by this Application Note, such initialization frequently includes enabling and configuring peripherals and off-chip circuit elements, and possibly initializing static storage. The framework enables interrupts before calling this function.

`void ShowDistress (byte SRSRValue)` – This application function may be invoked after the `InitializeCPU` and `InitializeProgram` functions. This function is invoked if and only if the SIM Reset Status Register (SRSR) indicates the reason for the reset was one of an illegal operation, an illegal address, or a failure to service the COP. These resets may be regarded as evidence of software failures. The contents of the SRSR are provided to this function in the `SRSRValue` parameter. This function may return or not. If this function does not return, it must service the COP (if the CONFIG values enable the COP). Should this function return, pro-

cessing will continue as with a typical (power on) reset.

The major activity of the client application, however, is almost always in the interrupt associated processing work. All interrupt associated processing work is done by a single client-supplied function, `FollowupInterrupts`. This function has the prototype:

```
void FollowupInterrupts (word flags)
```

The `flags` parameter to this function is the logical OR of the `flag` values passed by interrupt service functions to the `RequestFollowup` routine since the most recent invocation of `FollowupInterrupts`. That is, each invocation of `FollowupInterrupts` is given the set of requests that have been made since the previous invocation.

Interrupts are enabled while `FollowupInterrupts` is executing. The interrupt service function for any interrupts that occur are immediately called as a result of normal interrupt dispatching by the CPU. These interrupt service functions might request associated processing. Such requests for associated processing are accumulated while `FollowupInterrupts` is executing. The requests for associated processing will be forwarded to a new invocation of `FollowupInterrupts` immediately after the executing invocation of `FollowupInterrupts` returns.

It may happen that associated processing for one interrupt takes so long that two instances of another interrupt occur. If both instances of the other interrupt request the same type of associated processing, only one request for associated processing will be passed to the `FollowupInterrupts` function. This situation is called an "overrun," and it is the client application's responsibility to either avoid it (by ensuring the situation does not occur) or handle it (by recognizing and dealing with the situation). The background processing capability provided by the framework can be used for lengthy associated processing.

The final client-supplied function is `BackgroundProcessing`. This function has the prototype:

```
word BackgroundProcessing (word state)
```

At the first invocation of `BackgroundProcessing`, the value of `state` is 0. At subsequent invocations, the value of `state` is the most recent return value from `BackgroundProcessing`. The framework invokes the `BackgroundProcessing` function repeatedly until one of two conditions occur:

- A request for associated processing is made; or
- The `BackgroundProcessing` function returns 0.

In either case, the framework returns to waiting for associated processing to be requested, or to invoking the `FollowupInterrupts` client-supplied function.

The source code files (`framework.h` and `framework.c`) for the real-time application framework are included in the ZIP file that contained this Application Note. If your copy of this Application Note did not come packaged with these files, see the next section. That section contains a URI through which the ZIP file can be downloaded.

Further Information

This paper and the source code for the programs described herein are available at http://www.brianhetrick.com/tr/AN-3.zip.

The definitive documentation for the Freescale 68HC908QTxA/QYxA microcontrollers is *MC68HC908QY4A MC68HC908QT4A MC68HC908QY2A MC68HC908QT2A MC68HC908QY1A MC68HC908QT1A Data Sheet*, available at http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC908QY4A.pdf.

The *ExpressSCH* circuit schematic software used for Figure 4 is available at no charge from ExpressPCB, through the page at http://www.expresspcb.com/.

The *Special Edition: CodeWarrior for Microcontrollers* software used for the HC08 programs in this Application Note is available at no charge from Freescale, through the page at http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01272600610BF1.

Disclaimer

The author believes all content of this Application Note and the software it describes is a straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes, or any portion of either, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License