Oct 1996
Lit. No. 21-098A

H8 Microcontroller Series Application Notes Collection

HITACHI

H8 Microcontroller Series Application Notes

Content

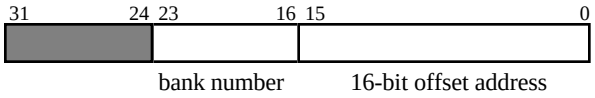
App.Note Number	CPU, Title	Pages
APPS/001/1.0	H8/300, Physical and Logical Address Space	5
APPS/003/1.0	H8/325, Configuring the 8-bit timer to create PWM waveforms	3
APPS/004/1.0	H8/300, Development and Debugging Code using the Banked Memory models for the H8/300	4
APPS/005/1.0	H8 Series, Using the S-Record file splitting utility ‘splitter’	4
APPS/008/1.0	H8/300, Serial communications interface	6
APPS/009/1.0	H8/300, Analogue to digital converter 4 APPS/010/1.0 H8/300, Digital to analogue converter	2
APPS/011/1.0	H8, EPROM security on the H8/300 and H8/500 Families	3
APPS/012/1.0	H8/300H, DMA request and transfer time for H8/300H	2
APPS/013/1.0	H8/300H, Interfacing various bit-size DRAM to the 300H	5
APPS/014/1.0	H8/3003, Adding the damping resistor to the oscillator	1
APPS/015/1.0	H8/300H, Enabling DMA end-of-transfer interrupts	2
APPS/016/1.0	H8/304X, Multiplexed I/O functions on the 3003 and 304X	4
APPS/019/1.0	H8/300, ‘C’ code framework example program	9
APPS/022/1.0	H8, Memory checking and initialisation program	7
APPS/026/1.0	H8/325, Interfacing to LCD character modules	4
APPS/036/1.0	H8/300H, Software UART implementation using 2 timer and 2 DMA channels	13
APPS/002/1.0	A Mechanism for Banking Data on the H8/300	6
APPS/006/1.0	Example Assembler Fuzzy Driver Routine, which can be called from C	6
APPS/007/1.0	Writing Downloadable C code using the IAR C Compiler	13
APPS/029/1.1	H8/300 C Code Demonstration Program	4
APPS/032/1.1	H8-3042 Framework Program	9
APPS/037/1.1	H8/300H Software UART implementation using 2 Timer and 2 DMA channels	12
APPS/043/1.0	Flat Panel Displays - EMI Considerations	4
APPS/044/1.0	H8/300H Direct Memory Access Channel (DMAC) example	4
APPS/045/1.0	H8/300H Direct Memory Access Channel (DMAC) - Serial communication example	5
APPS/048/1.1	LCD character module control using H8_300H	10
APPS/049/1.1	Producing Optimised C for 300/300H Controllers	12
APPS/050/1.2	Memory Access Speed on the H8/300H	2
APPS/051/1.0	Real time low power sheduler for the 300L series.	7

HITACHI

H8/300 Physical and Logical Address Space

The H8/300 C compiler can be used to produce code which is greater in size than the total address space of the microcontroller. The banked (b) and mini-banked (m) memory models make use of an I/O port to extend the address bus by up to 8 bits *for code only*. Each time a given function is called a bank switch may have to be undertaken to access the memory device / area containing it, thus a table of addresses must be available to be able to perform the bank switch, this is created for you by the compiler in the segment FLIST. Each vector in FLIST has the following structure:

Figure 1: Structure of FLIST vector in banked models



The default I/O port used is Port 4, but this can be changed by modifying the CSTARTUP and L07 assembler source files.

The start address, size, separation and type of banked segments is defined at link time using the following format:

-b<storage class>(<segment type>)<segment name>=<start>,<size>,<offset>

where:

<storage class>	is	≅# for physical addressing
	or	blank for logical addressing
<segment type>	is	≅CODE to define the correct segment type
<segment name>	is	the comma separated list of segment names to be included in the banks
<start>	is	the start address in hexadecimal
<size>	is	the bank size in hexadecimal
<offset>	is	the *logical* offset between the start addresses of successive banks

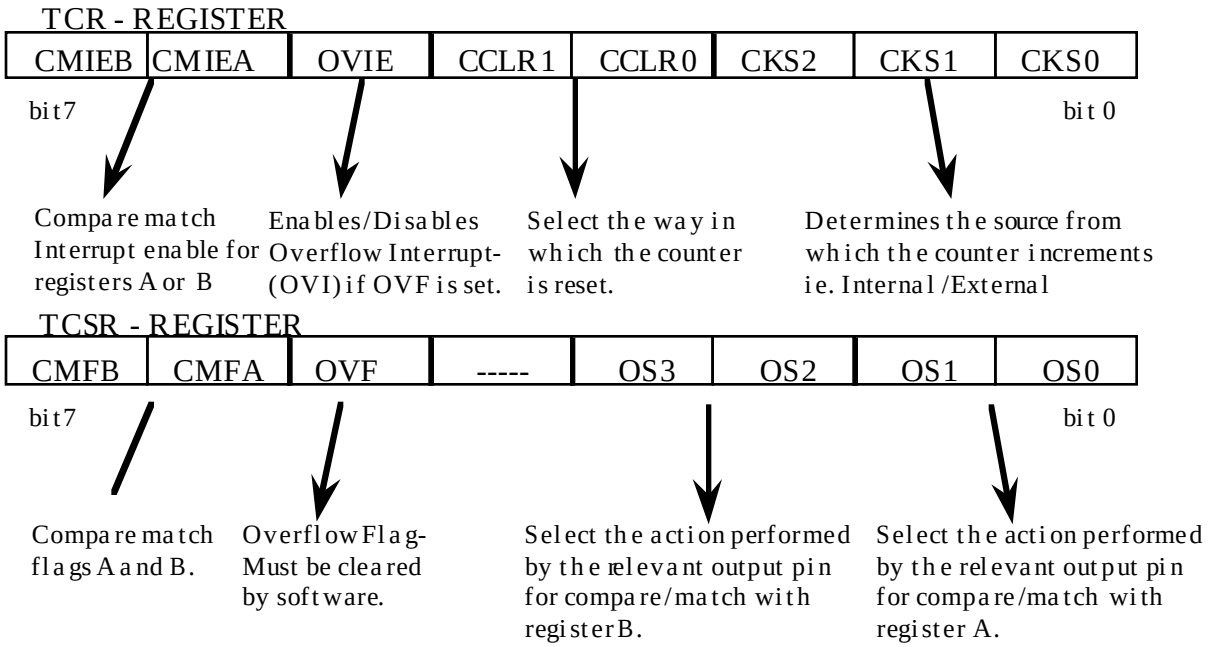
There are two ways to allocate memory to banked segments at link time - either Logically or Physically the difference is defined below, using the example of a three code bank system, where the banks are of size H 4000.

Configuring the 8 - bit timer on the H8-325 to create PWM waveforms

The H8/325 series have one 16 bit free-running timer and two multifunctional 8-bit timers on board. Each of the two 8 bit timers have two registers; TCORA and TCORB associated with it. The values contained in these registers are compared with the timer register -TCNT. If a match occurs then an interrupt will be requested. An interrupt request will also occur if the timer overflow (FFH > 00H) occurs. If enabled, the interrupt requests will cause the program to vector to the appropriate interrupt service routine.

TCORA and TCORB are both set to FFH after reset, whilst the timer control and status registers are depicted below.

Figure 1 - 8-bit Timer Control/Status Registers



Creating a PWM waveform

By using the compare/match facility to toggle an output pin, PWM can be achieved using the 8-bit timers. The following code example demonstrates this effect and creates a waveform of 50% duty cycle, see figure2, below. Software intervention is not required, hence no interrupt service routines are used.

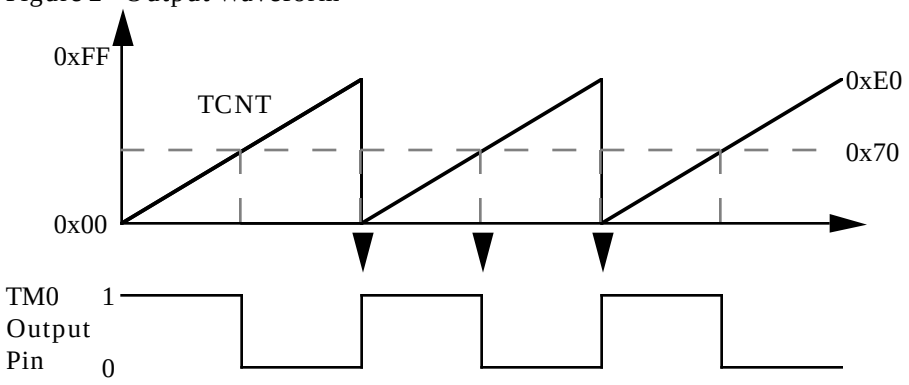
NB. By changing the initial values of TMR0_TCORA and TMR0_TCORB, the

duty cycle can be altered.

Program

```
#include <ioh8325.h>        /* include I/O header files */

void init_timer (void)
 {
 TMR0_TCORA = 0xE0;     /* sets pulse rate              */
 TMR0_TCORB = 0x70;     /* sets pulse width             */
 TMR0_TCSR = 0x06;      /* o/p '0' on compare B         */
                        /* o/p '1' on compare A         */
 }

int main (void)
 {
 init_timer();          /* initialise the 8-bit timer ctrl registers    */
 TMR0_TCR |= 0x0B;      /* start timer with phi/1024 internal clock          */
                    /* timer reset on comp/match with TCORA        */
 while(1);              /* main code sequence ... repeat ≅forever       */
 }
```

Figure 2 - Output Waveform



Developing and Debugging Code using the Banked memory models for the H8/300

The H8/300 series of microcontrollers have a linear 64KByte address range. The C compiler does directly support an extended ≅banked address range for code. The banked memory model used by the compiler extends the effective address bus by utilising a number of I/O pins to switch between banked external memory (see the H8/300 C tools tutorial for more information).

The linker provided with the C compiler produces a single output file in the format specified by the user (usually Motorola S-Record format). The S-Record file contains the following:

1, Constant ROM segments (vectors, constants and initialisers)
2, Startup code ROM segments.
3, Library code ROM segments.
4, Non-banked user code ROM segments.
5, Banked user code ROM segments.
6, Various data RAM segments.

Having generated the S-Record files the user then may decide to debug the code using an evaluation board, or an in-circuit emulator, or to program an H8 device and try to run the code directly.

In Circuit Emulators:

The Hitachi in circuit emulators PCE83XX do not directly support the banked memory models as the I/O lines used to switch banks are user defined and may be changed dynamically in some systems. Thus the full S-Record file cannot be loaded into the device as the banked segments contained in the object code will be outside the addressing range of the PCE83XX. One approach is to separate the banked segments from the object code, and then load them into external memory on the target board (i.e. not emulator memory). There are two hurdles to overcome in doing this, firstly the object code needs to be dissected , and secondly the target memory needs to be programmed.

i, Dissecting the S-Record Object file:

The user may dissect a S-Record file by hand using an EPROM programmer to load in the whole file, then selectively save blocks to separate files. This however can take some time to do, and will have to be repeated each time the user has a new revision of code to debug. The process as a whole is prone to errors and will increase the development time unnecessarily.

ii, Moving the S-Record addresses:

Given that the banks of code exist in separate files, the base file may be loaded into an In-Circuit Emulator (ICE) for debugging. The memory on the target board which is to contain the banked code segments must then be programmed. Usually the finished product will use EPROM devices to store the banks. Typically many banks will typically exist in one EPROM, thus the EPROM will need to have the banks contiguously loaded from address 0. To move logically spaced S-Records into one contiguous range suitable for loading into a single EPROM the user will need to cut and paste all of the banks using an EPROM programmer. Once again the process will take some time and is fraught with possible errors.

iii, Choosing the storage medium:

Once the user has generated separate files which contain the code / data to exist on the H8 microcontroller and one or more banked code segment files, the data needs to be programmed into the target memory. Usually this has been achieved using EPROMs, as this matches the final production run for the target hardware, however this will be a slow process as the EPROMs will have to be erased and then reprogrammed. A quicker solution is to use SRAM on the target and program it directly from the ICE. This can be achieved using the following method:

1, Compile and link the banked code using the IAR tools using logical addressing.

2, Use the Hitachi ≅splitter routine to segment the file in the following manner:

a, Place all H8 files into a single output file without moving them.
b, Place bank 0 code into another file without moving the bank base address.
c, Place all successive banks in separate files and move the base address of each to the start address of bank 0.

3, Use the ICE to set up a series of macros to perform the following:

a, Set the I/O lines responsible for addressing bank 0
b, load in the S-record file for bank 0

...and repeat for banks 1..N

The ICE will effectively switch the address decoding on the target hardware to point to the correct SRAM bank start address, and then the bank will be loaded into area using the ICE to load the S-Record file (which has the correct lower 16-bit addresses).

In short the ≅splitter utility is used to separate all banks into separate files and then move all banks so that they occupy the same address range (the lower 16-bits of the actual bank address). The full bank address is reconstructed by the ICE just before loading of the bank from the file using the I/O ports as defined by the bank switching routine L07.

The Hitachi utility ≅splitter is available on request from Hitachi, and is supplemented by applications note APPS/005.

Using the S-Record file splitting utility ≅splitter

Hitachi has produced a utility to enable code developers to reformat the object code produced by the IAR C compiler / linker. The ≅splitter utility has been written for customers who wish to develop and debug code which is to reside in target system memory (i.e. not on an in-circuit emulator). The decoding logic used to define the target memory map may require that the logical addresses put out by the H8 processor do not correspond to the actual addresses the code resides in.

Splitter Specification

The splitter utility allows the user to take a single S-Record object file and split it into a number of separate files, the features are:

1, The splitter utility uses a textual format file to define the input file (i.e. linker produced S-Record file), the names of all of the output files, and the addresses and ranges of the segments to move.

2, Each output file may have a number of segment areas associated with it.
3, Each segment area may have its base address moved to any address.
4, Segments which need to be ≅shadowed can be copied to any number of output files.
5, Each shadow segment may have its base address moved.

6, In performing the split, it may be necessary to change the base address for many S-Records, and in addition certain S1 records may need to be translated into S2 records to handle a wider address range. This is all done automatically by splitter with the appropriate recalculation of checksums where necessary.

6, ≅splitter checks the given format file for segment clashes and invalid ranges. In addition if an S-Record exists in the input file which has no defined output destination the user is informed of the error and the splitting process stops.

7, On completion ≅splitter informs the user of the total number of S-Records processed.

Format File Specification

A line in the format file may take one of the following forms:

1, Input file name specification line contains ≅-I,whitespace, then the filename in double quotes:

e.g. -I ∋input.obj

2, Output file specification for a segment ≅-O plus filename, the segment range (in hexadecimal or decimal) plus an optional new base address:

e.g. -O ∋out1.obj 0x1000 0x1FFF
e.g. -O ∋out2.obj 0x1000 0x1FFF BASE 0x20000

If no BASE value is specified the S-Records are copied to the new file directly, without modification.

3, Copied segment specification ≅-C ,the output filename, the range of addresses defining the segment and an optional new base address for the segment.

e.g. -C ∋out3.obj 0x0000 0x0100

4, Any line not containing one of ≅-I, ≅-O or ≅-C is treated as a comment.

Example address Mapping and format file:

The user has a program for the H8/300, compiled using the banked memory model. The system can be described in logical addresses as:

0000 to 7FFF	*Vectors, constants, startup code and libraries.*
8000 to BFFF	*Bank 0 user code*
C000 to FFFF	*User data.*
18000 to 1BFFF	*Bank 1 user code*
28000 to 2BFFF	*Bank 2 user code*
38000 to 3BFFF	*Bank 3 user code*

The banked code will actually reside in a single EPROM on the target, with each bank segment being contiguous in memory. Thus the actual S-Record addresses need to be:

0000 to 7FFF	*Vectors, constants, startup code and libraries. (H8)*
0000 to 3FFF	*Bank 0 user code (EPROM)*
C000 to FFFF	*User data. (H8)*
4000 to 7FFF	*Bank 1 user code (EPROM)*
8000 to BFFF	*Bank 2 user code (EPROM)*
C000 to FFFF	*Bank 3 user code (EPROM)*

Thus we need two output files, one for the H8 and the other for the EPROM. The format file is then:

define input file
-I ∋aout.a21
H8 output file segments
-O ∋h8.a21 0x0000 0x7FFF

-O эh8.a21 0xC000 0xFFFF
EPROM output file segments
-O эeprom.a21 0x8000 0xBFFF BASE 0x0000
-O эeprom.a21 0x18000 0x1BFFF BASE 0x4000
-O эeprom.a21 0x28000 0x2BFFF BASE 0x8000
-O эeprom.a21 0x38000 0x3BFFF BASE 0xC000

To invoke the splitter utility, enter ≅splitter with the filename of your format file (if no extension is specified the default extension ≅.fmt is used).

e.g. c:\> *splitter example.fmt*

Two output files will then be created, with the desired S-Record configuration.

THE H8/300 SERIAL COMMUNICATIONS INTERFACE

The serial communications interface (SCI) can perform either synchronous or asynchronous communications, and data rates of up to 2.5Mbit/sec are possible. Each channel of SCI has its own baud rate generation, so the use of serial communications does not impact the number of timers available in the microcontroller.
By programming a control register and loading the baud rate register, a wide range of data rates can be achieved from one source of microcontroller clock. In asynchronous mode, several data formats are catered for, including the provision of odd or even parity.
To indicate various conditions occurring within the SCI, a status register is provided which contains flags for receive buffer full, transmit buffer empty or receive error. Each of these flags has an interrupt associated with it to indicate the occurrence of such a condition. Each of the SCI data registers (TDR and RDR) is double buffered so it is possible to transmit and receive data in a "Back-to-back" manner.

The following three code examples show how to initialise and use both the transmit and receive interrupts. Two types of transmission are provided, synchronous and asynchronous.

MODULE TO TRANSMIT A MESSAGE USING SYNCHRONOUS COMMUNICATIONS

```
#pragma language=extended   /* ALLOW NON ANSI SPECIFIC EXTENSIONS */
#include <inth8337.h>
#include <ioh8337.h>

#define  BRR_SETUP 0x0C
#define  SMR_SETUP 0x80
#define  SCR_SETUP 0xCF
#define CLR_TDRF   0x7F
#define  SRR_SETUP 0xC8
#define SET_TE     0x20
#define TEND_SET   0x40
#define CLR_TE     0xDF

char message[5] = "HELLO";   /* TRANSMISSION MESSAGE */
int  index = 0;          /* INDEX VARIABLE FOR MESSAGE TRANSMISSION */
```

```
interrupt [SCI_TXI0]    void tx_isr (void)
{
 SCI0_TDR = message[index++];     /* TRANSMIT NEXT CHARACTER        */
 SCI0_SSR &= CLR_TDRE;           /* CLEAR THE TDRE BIT TO '0'      */
 if (index > 4)                 /* ENSURE ALL BITS HAVE BEEN TRANSMITTED */
 {
  while (SCI0_SSR &= TEND_SET != TEND_SET);   /* WAIT FOR TEND BIT IN
                                                 SSR TO BE SET (TO '1') */
  SCI0_SCR &= CLR_TE;               /* CLEAR TE BIT IN THE SCR          */
 }
}

  /* DELAY OF 105 us */
void wait_for_1_bit (void)
{
 int a;
 for (a=0;a<100;a++);

}

void init_sci (void)
{
 SCI0_SCR &= SCR_SETUP;        /* CLEAR TE AND RE BITS IN SCR TO '0'      */

 SCI0_SMR = SMR_SETUP;         /* SETUP SYNCH, NO PARITY, 8 BIT, 1 STOP
BIT  */
                        /* INTERNAL CLOCK                                  */

 SCI0_BRR = BRR_SETUP;         /* 9600 BAUD (FOR AN 8 Mhz XTAL)         */

 wait_for_1_bit();         /* WAIT FOR AT LEAST THE TIME TAKEN TO TX OR RX
1 BIT */


 SCI0_SCR |= SET_RE;        /* SET THE RECEIVE ENABLE (RE) BIT AND
                                  INTERRUPT ENABLE */
}


void main (void)
{

 /* INITIALISE THE SERIAL COMMUNICATIONS INTERFACE */
 init_sci();
 set_interrupt_mask(0);    /* DISABLE THE INTERRUPT MASK */
```

```
 while (1);                    /* DO FOREVER */
}

 MODULE TO TRANSMIT A MESSAGE USING ASYNCHRONOUS
COMMUNICATIONS

#pragma language=extended   /* ALLOW NON ANSI SPECIFIC EXTENSIONS */
#include <inth8337.h>
#include <ioh8337.h>

#define  BRR_SETUP 0x0c
#define  SMR_SETUP 0x00
#define  SCR_SETUP 0xcf
#define SET_TE    0x20
#define CLR_TDRE  0x7f
#define TDRE_SET  0x80

char message[5] = "HELLO"; /* TRANSMISSION MESSAGE */

  /* DELAY OF 105 us */
void wait_for_1_bit (void)
{
 int a;
 for (a=0;a<100;a++);
}

void init_sci (void)
{
 SCI0_SCR &= SCR_SETUP;        /* CLEAR TE AND RE BITS IN SCR TO '0'     */

 SCI0_SMR = SMR_SETUP;          /* SETUP ASYNCH, NO PARITY, 8 BIT, 1 STOP
                                   BIT  */
                        /* INTERNAL CLOCK                                 */

 SCI0_BRR = BRR_SETUP;          /* 9600 BAUD (FOR AN 8 Mhz XTAL           */

 wait_for_1_bit();         /* WAIT FOR AT LEAST THE TIME TAKEN TO TX OR RX
                                   1 BIT */

 SCI0_SCR |= SET_TE;         /* SET THE TRANSMIT ENABLE (TE) BIT        */
}

void transmit_msg (int index)
{
/* REPEAT POLLING UNTIL TDRE = '1' */
 while ((SCI0_SSR &= TDRE_SET) == TDRE_SET);
```

```
 SCI0_TDR = message[index];          /* TRANSMIT CHARACTER  */
 SCI0_SSR &= CLR_TDRE;               /* CLEAR TDRE TO '0'   */
}

void  main (void)
{
 int i;

 init_sci();

 /* TRANSMIT SERIAL DATA */
 for (i=0;i<5;i++)
 {
  transmit_msg(i);
 }
 while (1);                  /* DO FOREVER */
}
```

MODULE TO RECEIVE A 255 CHARACTER MESSAGE USING
ASYNCHRONOUS COMMUNICATIONS

```
#pragma language=extended   /* ALLOW NON ANSI SPECIFIC EXTENSIONS */
#include <inth8337.h>
#include <ioh8337.h>

#define  BRR_SETUP 0x0C
#define  SMR_SETUP 0x00
#define  SCR_SETUP 0x50
#define CLR_RDRF   0xBF
#define  SRR_SETUP 0xC8
#define SET_RE     0x10

#pragma language=extended    /* SPECIFY NON ANSI EXTENSIONS      */

#include <ioh8337.h>        /* REQUIRED I/O HEADER FILE        */
#include <inth8337.h>        /* REQUIRED INTERRUPT HEADER       */

#define DATA_PACKETS 255     /* MAX NO. OF DATA TO BE RECEIVED   */

char data_in[DATA_PACKETS];  /* DATA TO BE RECEIVED             */
int  i=0;                    /* POINTER TO THE DATA-INPUT BUFFER  */

interrupt [SCI_RXI1] void rx_isr (void)
{
 data_in[i++] = SCI1_RDR;    /* READ DATA FROM REGISTER */
```

```
 SCI1_SSR &= CLR_RDRF;       /* CLEAR RDRF FLAG IN SSR  */
}



interrupt [SCI_ERI1] void rx_error_isr (void)
{
 /* ERROR RECOVERY CODE OR  */
 /* SIGNAL THE END OF RECEPTION */
 /* AND INITIATE TRANSMISSION --SEE AS_TX.C APPLICATION CODE  */

 SCI1_SSR &= SSR_SETUP;  /* CLEAR THE ORER, FER AND PER BITS TO '0' */
}

  /* DELAY OF 105 us */
void wait_for_1_bit (void)
{
 int a;
 for (a=0;a<100;a++);
 }


void init_sci (void)
{
 SCI0_SCR &= SCR_SETUP;       /* CLEAR TE AND RE BITS IN SCR TO '0'     */
SCI0_SMR = SMR_SETUP;         /* SETUP ASYNCH, NO PARITY, 8 BIT, 1 STOP
                              BIT  */

                              /* INTERNAL CLOCK                         */
 SCI0_BRR = BRR_SETUP;        /* 9600 BAUD (FOR AN 8 Mhz XTAL)        */

 wait_for_1_bit();       /* WAIT FOR AT LEAST THE TIME TAKEN TO TX OR RX
                              1 BIT */

 SCI0_SCR |= SET_RE;       /*   SET THE RECEIVE ENABLE (RE) BIT AND
                              INTERRUPT ENABLE */
}
void main (void)
{
 int i;
 init_sci();
 set_interrupt_mask(0);    /* DISABLE THE INTERRUPT MASK */
 while (1)
 /* DO FOREVER */
 ;
}
```

THE H8/300 SERIES ANALOGUE TO DIGITAL CONVERTER

Some devices within the H8/300 family have A/D facility, these converters have the following features:-

* Eight bit resolution
* Eight channels of analogue inputs - specified using a multiplexer.
* Conversion times as low as 12.2 us (at 10Mhz)
* Conversion can be triggered by an external signal
* A CPU interrupt (ADI) is generated once A/D conversion is completed
* Selectable modes
    - Single Mode
    - Scan Mode

Single Mode - A/D conversion begins when the ADST bit in the ADCSR is set to 1. When conversion is completed, the completion flag(ADF) is set. If the interrupt enable bit (ADIE) is also set, an A/D conversion end interrupt (ADI) is requested so that the result can be processed.

Scan Mode - This mode is used to monitor inputs on up to four channels - selected by the CH0 - 2 bits in the Analogue to Digital Control Status Register (ADCSR). When the ADST bit is set to 1, A/D conversion starts on the first selected channel.
As soon as conversion of the first channel is completed, conversion of the next channel begins. The selected channels are repeatedly converted until the ADST bit is cleared. The converted results for each channel are transferred to and stored in the data registers ADDRA to ADDRD.
The ADST bit can be set to 1 by software, or the external trigger signal (ADTRG).

The following code examples show how to configure the on-board A/D in both Single and scan modes

EXAMPLE OF THE A/D CONVERTER USING AN INTERRUPT TO SAMPLE ONE CHANNEL

```
#pragma language=extended   /* ALLOW NON ANSI SPECIFIC EXTENSIONS */
#include <inth8327.h>
#include <ioh8327.h>
```

```
/* VARIABLE TO HOLD THE CONVERTED DATA */
unsigned char a2d_data ;

#define CLR_ASDT 0xdf  ;
#define DIS_EXT 0x7f   ;
#define  A2D_SETUP 0x61 ;
#define CLR_ADF 0x7f   ;

 /* THE A/D CONVERTS THE VOLTAGE AT AN1 TO AN 8-BIT VALUE    */
 /* AT THE END OF THE CONVERSION, THE RESULT IS IN ADDRB     */
 /* THE ADF FLAG IS THEN CLEARED. THE ADST IS CLEARED
AUTOMATICALLY */
 /* AND THE CONVERSION HALTS */

interrupt [AD_ADI] void a_to_d_isr (void)
{
 a2d_data = AD_ADDRB;        /* READ THE A TO D DATA    */
 AD_ADCSR &= CLR_ADF;        /* CLEAR THE ADF FLAG */
}

void init (void)
{
 AD_ADCSR &= CLR_ASDT;     /* CLEAR ADST TO '0' IN THE STATUS
CONTROL REG */
 AD_ADCR = DIS_EXT;       /* DISABLE EXTERNAL TRIGGER FROM THE
CONTROL REG */
 AD_ADCSR = A2D_SETUP;     /* SET TO 'SINGLE MODE'. CLOCK AND
CHANNEL ARE ALSO SETUP*/
 /* ALTERING THESE VALUES DURING CONVERSION MAY LEAD TO
ERRORS */

}


int main (void)
{
 init();                        /* SET UP CONTROL / STATUS REGISTERS */


 while (1)
 /* TO RESUME DATA CONVERSION, SOFTWARE MUST SET THE ADST BIT
TO '1' */
 ;
}
```

EXAMPLE OF AN A/D INTERRUPT TO SCAN SEVERAL ANALOGUE INPUT

```
CHANNELS
#pragma language=extended   /* ALLOW NON ANSI SPECIFIC EXTENSIONS */
#include <inth8327.h>
#include <ioh8327.h>

 /* VARIABLES TO HOLD THE CONVERTED DATA */
unsigned char a2d_data0, a2d_data1, a2d_data2, a2d_data3;
#define CLR_ASDT 0xdf  ;
#define DIS_EXT 0x7f   ;
#define  A2D_SETUP 0x73 ;
#define CLR_ADF 0x7f   ;
  /* THE A/D CONVERTS THE VOLTAGES AT AN0-3 TO AN 8-BIT VALUE*/
 /* AT THE END OF THE CONVERSION, THE RESULT IS IN ADDRA-D     */
 /* THE ADF FLAG IS THEN CLEARED */

interrupt [AD_ADI] void a_to_d_isr (void)
{
 a2d_data0 = AD_ADDRA;       /* INPUT FIRST A/D CHANNEL  */
 a2d_data1 = AD_ADDRB;       /* INPUT SECOND A/D CHANNEL */
 a2d_data2 = AD_ADDRC;       /* INPUT THIRD A/D CHANNEL  */
 a2d_data3 = AD_ADDRD;       /* INPUT FOURTH A/D CHANNEL */
 AD_ADCSR &= CLR_ADF ;       /* CLEAR THE ADF FLAG      */
}

void init (void)
{
 AD_ADCSR &= CLR_ASDT;     /* CLEAR ADST TO '0' IN THE STATUS
                              CONTROL REG */
 AD_ADCR = DIS_EXT;       /* DISABLE EXTERNAL TRIGGER FROM THE
                              CONTROL REG */
 AD_ADCSR = A2D_SETUP;     /* SET TO 'SCAN MODE'. CLOCK AND
                              CHANNEL ARE ALSO SETUP*/
 /* ALTERING THESE VALUES DURING CONVERSION MAY LEAD TO
ERRORS */
}

int main (void)
{
 init();                        /* set up control/status registers */

 while (1)
 /* TO HALT DATA CONVERSION, SOFTWARE MUST CLEAR THE ADST BIT
TO '0' */
 ;
}
```

THE H8/300 DIGITAL TO ANALOGUE CONVERTER

Some members of the H8/300 series have an on-chip Digital to Analogue converter. Analogue signals can be output on up to two channels. The on-chip D/A has the following features:-

* 8 bit resolution
* Maximum conversion time of only 10us
* Output voltage in the range 0 Volts to AVcc

The D/A is enabled when the D/A 'Enable' bit is set to 1. Once enabled, the DADR contents are continuously converted and output. The output value is calculated by:- D/A OUTPUT VOLTAGE = (DADR/256) x AVcc

D/A CONVERTER EXAMPLE

```
#pragma language=extended   /* allow non ANSI specific extensions */
#include <inth8327.h>
#include <ioh8327.h>

#define DA_DACR_SETUP 0x40

 /* DELAY OF 105 us */
void wait (void)
{
 int a;
 for (a=0;a<100;a++);
 }
int main (void)
{
 int i;                    /* LOOP VARIABLE */
 DA_DACR = DA_DACR_SETUP;          /* SET UP ANALOGUE CHANNEL 0 IN
                                    D/A CONTROL REGISTER */
 while(1)
  {
   for (i=0;i<256;i++)
    {
     DA_DADR0 = i;             /* PROVIDE DATA TO BE CONVERTED */
     wait();                   /* DELAY TO ALLOW A CONVERSION TO AN
                                  ANALOGUE OUTPUT */
    }
  }
}
```

EPROM Security on the H8/300 and H8/500 families

The H8/300 and H8/500 microcontrollers have an EPROM security feature that can be used by the application programmer. This feature allows the user of the microcontroller to protect parts (or all) of the code programmed into the on chip EPROM of the device from being read by means other than his or her own program. Due to the nature of this feature it cannot be tested by Hitachi and is therefore not guaranteed. It is up to the user to determine whether or not to implement the features of this function and accept sole responsibility for its outcome.

Memory Configuration:

The memory matrix of the H8 microcontroller is configured as a dual matrix, one with even addresses and one with odd addresses. The configuration of each matrix appears as lines of memory 32 bytes wide. (32 x 8, 256 bits). This configuration allows an individual memory line to consist of 64 bytes of data (including both even and odd addressing). Each memory line has 1 security bit thus allowing every 64 byte segment to have the option of the security feature. The address of this security bit is the same as the starting address for the memory line.

Security Functions:

The security function has two different operations depending on the mode of operation that the device is placed into; EPROM programming mode or CPU operation mode.

EPROM Programming mode:

In the EPROM programming mode, the ability of the EPROM programmer to read the EPROM contents is limited by the state of the security bit.

If the security bit is a ‘ 1 ’ (unprogrammed state), then the data in the EPROM can always be read. If the security bit is a ‘ 0 ’ (programmed state), then any read operation to the EPROM will result in a ‘00’ being read. This indicates that once the security bit is programmed, the user will be unable to verify the contents of the EPROM.

security bit 1 EPROM data can be read (normal)

security bit 0 “00” data can be read.

CPU operating mode:

In the CPU operating mode, the ability of any device to read the EPROM contents is limited by the state of the security bit.

If the security bit is a ‘ 1 ’ (unprogrammed state), then the data in the EPROM can always be read. If the security bit is a ‘ 0 ’ (programmed state), then the read state of the EPROM (from the CPU), depends upon where the instruction execution is occurring from.

security bit	1	EPROM data can be read (normal)
security bit	0	After Reset, the CPU can read EPROM data until it executes an instruction outside the internal EPROM area (either external memory or internal RAM). Once an instruction is executed outside the internal EPROM memory area, then the EPROM becomes disabled and cannot be accessed any further. This prohibits an external program from being able to “dump” the contents of the internal on chip EPROM.

Programming the Security bit:

There exists two EPROM programming modes; Normal and Security. The Normal programming mode allows the user to program the code/data area of the on-chip memory of the device. The Security programming mode allows the user to program the security bits, thus implementing the security feature. The security function is implemented by programming a ‘0’ into the address corresponding to the memory line location. Setting the programming mode is accomplished by setting certain I/O port pins as shown in the following table.

H8/300 Family Programming modes

Device	Programming Mode	Port Pins
H8/325 Family	Normal Security	$P7_0 = 1$ $P7_1 = 1$ $P7_0 = 1$ $P7_1 = 0$
H8/330	Normal Security	$P8_0 = 1$ $P8_1 = 1$ $P8_0 = 1$ $P8_1 = 0$
H8/338 Family	Normal Security	$P6_4 = 1$ $P6_3 = 1$ $P6_4 = 1$ $P6_3 = 0$
H8/350	Normal Security	$P8_0 = 1$ $P8_1 = 1$ $P8_0 = 1$ $P8_1 = 0$

H8/500 Family Programming modes

Device	Programming Mode	Port Pins
H8/520	Normal Security	$P5_0 = 1$ $P5_1 = 1$ $P5_0 = 1$ $P5_1 = 0$
H8/532	Normal Security	$P6_0 = 1$ $P6_1 = 1$ $P6_0 = 1$ $P6_1 = 0$
H8/534	Normal Security	$P6_0 = 1$ $P6_1 = 1$ $P6_0 = 1$ $P6_1 = 0$
H8/536	Normal Security	$P6_0 = 1$ $P6_1 = 1$ $P6_0 = 1$ $P6_1 = 0$

Again, this feature cannot be tested by Hitachi and therefore cannot be guaranteed. It is up to the user to determine whether or not to implement the function of this feature and accept sole responsibility for its outcome.

DMA Request and Transfer Time for H8/300H

The DMA Controller module of the H8/300H family of microcontrollers can be activated in 3 possible ways:

1. By internal interrupts from either the ITU (upon compare-match or input capture conditions), or the SCI (upon transmit-end or receive-end).
2. By external request via a falling edge or low level at the DREQ pin.
3. In software by register setup (auto-request).

The table below shows the activation methods, transfer direction, and the bus modes for each DMA transfer type.

TRANSFER TYPE	BUS MODE	ACTIVATION	Address or Register Length	
			SOURCE	DESTINATION
Short Address Modes - /O Mode - Idle Mode - Repeat Mode	Cycle-Steal	Compare-Match, Input-Capture, or TxI	24 Bits	8 Bits
		RxI	8 Bits	24 Bits
		External Request	24 Bits	8 Bits
Normal Mode	Cycle-Steal	External Request	24 Bits	24 Bits
	Cycle-Steal or Burst	Auto Request		
Block Transfer Mode	Burst	Compare-Match, Input-Capture, or External Request		

During Cycle-Steal DMA modes, only one byte (or word) of data is transferred at each request, and the CPU and DMAC share the data bus by alternating CPU cycles with DMA transfer cycles. During Burst DMA modes, a whole string of data is transferred at each request; consequently, the DMAC keeps the bus until each data block is transferred while the CPU is idling. This Tech Note will describe how to calculate the minimum time between 2 consecutive byte or word DMA transfer requests by each of the following activation sources (assuming no wait-states are added into the alternating CPU cycles).

Interrupt requests:

The minimum time between consecutive transfer requests activated by an internal interrupt, and with the DMAC operating in either the Short Address Modes or the Normal Mode, can be calculated by adding the following timing

parameters:

1. The time it takes to request the DMA, ie., the time between the interrupt-causing event and the internal interrupt request signal. If the interrupt is caused by a compare-match event, the interrupt acknowledge time between the compare-match signal and the internal interrupt request signal (IMI) is 1 Timer clock period, as can be seen in the ITU section of the the H8/3003 Hardware Manual. If the interrupt is caused by an input-capture event, the acknowledge time between the input-capture pin toggling and the DMA request input-capture signal is 1 clock period, and the interval between the input-capture signal and the internal interrupt request signal is 1.5 clock periods, which makes it a total of 2.5 Timer clock periods, as can be observed from the figures in the same section in the manual. If the interrupt source is an SCI transmit or receive-end, there is no delay between the event occurence and the interrupt request signal.
2. The DMA "latency" time, ie., the time it takes from the transfer request until the DMA controller starts operating. The H8/3003 Hardware Manual specifies a minimum latency time of 4 T-states.
3. One DMA "dummy" cycle (Td), which lasts for 1 T-state.
4. The time it takes the DMA to read the contents of the source address. Depending upon the bus-controller settings and/or what memory area is being accessed, this process can take a minimum of either 2 or 3 T-states (assuming no wait states are induced in the cycle).
5. If the DMAC operates in the Block Transfer Mode, the time it takes the DMA to write data at the destination address must be added in the above calculation, since the interrupt request is sampled at the end of this cycle. Depending upon the bus controller settings and/or what memory area is being accessed, this process can take a minimum of either 2 or 3 T-states (again assuming no wait states are induced in this cycle).
6. Additional CPU states occurring during the following CPU cycle.

As an example, figure 1 below shows the timing between consecutive input-capture activated DMA cycles operating in the Short Address Modes performing transfers from a 16-bit 2 T-state access area to an 8-bit 3 T-state access area. The Timer as well as the DMAC is assumed to operate at the system clock frequency. The Timer is set up so that input-capture events are triggered on both edges at the input-capture pin. The minimum time between consecutive input-capture signals will be 7 T-states between the first 2 toggle actions at the input-capture pin, and 8 T-states between the following requests, assuming the inserted CPU cycle is only 2 T-states.

Interfacing various bit-size DRAMs to the H8/300H

The H8/300H microcontroller family facilitates the task of interfacing various types of memory devices within its linearly addressed memory map. In particular, dynamic random access memories can be easily connected given the addition of a Refresh Controller module into the H8/300H chip which provides properly timed control signals (RAS, CAS, WE, RD) as to insure smooth access and refresh cycles. In addition, external device decoding logic is kept at a bare minimum since the microcontroller provides chip select signals for each memory block area. The H8/3003 Hardware Manual, Application Note AE-0043, and technote TN-0131 discuss in detail how x16 DRAMs can be interfaced to the H8/3003 AS WELL AS THE NECESSARY TIMING CONSIDERATIONS. THIS TECHNOTE WILL ILLUSTRATE HOW OTHER DRAM PARTS CAN BE PROPERLY CONNECTED TO THE MICROCONTROLLER.

DRAMs of x1, x4, and x8 bit-size can also be directly (or with a minimum amount of glue logic) attached to the H8/300H family. Figure 1 below shows bit-by-bit how the Refresh Control Register (RFSHCR) must be programmed for these particular interfaces. The CAS/WE bit can be programmed either way since no byte control is required. If this bit is set, then the CAS signal will be provided through both HWR and LWR pins, and the WE signal will be provided via the RD line. If the CAS/WE bit is cleared, the CAS signal will be available at the RD pin, and the WE signal can be wired from either HWR or the LWR pins of the H8/300H. The M9/M8 bit must be set to 1 for DRAM devices that use 9-bit or higher column address mode; most x1, x4, and x8 available DRAMs use this mode.

7	6	5	4	3	2	1	0
SRFMD	PSRAME	DRAME	CAS/WE	M9/M8	RFSHE	Reserved	RCYCE
X	0	1	0 or 1	0 or 1	X	1	X

Figure 1. Refresh Control Register Settings.

Figures 2-8 show the following examples of generic x4 and x8 bit-size DRAM

connections via either 8-bit or 16-bit data bus along with its corresponding memory map for the 16MByte expanded modes:

Example 1 (fig. 2-3):

A Hitachi HM514800 512k x 8 is interfaced to the H8/300H via an 8-bit data bus. Since the DRAM uses a 10 rows x 9 columns addressing mode, lines A0 - A9 will provide the row address, and lines A1 - A9 will output the column address. A0 is connected to pin A9 of the DRAM since its state remains unchanged during both row and column addressing. The CAS signal is obtained from the HWR line of the microcontroller, and the WE signal is provided at the RD pin (assuming the CAS/WE bit of RFSHCR is set). The RAS pin of the H8/300H is connected to the CAS pin of the DRAM since these signals are multiplexed inside the microcontroller. Given this hardware configuration, the addressable DRAM memory occupies 512KBytes between H'600000 and H'67FFFF.

Example 2 (fig. 3-4):

Four Hitachi HM514256 256K x 4's are connected to the H8/300H via a 16-bit data bus. The RAS, CAS and WE signals are provided as described in the example above. Since the DRAM uses a 9 rows x 9 columns addressing mode, lines A1 - A9 will provide both row and column address. The A0 line form the H8/300H is not connected since word accesses are performed (see explanation in AE-0043). The addressable DRAM memory occupies the same boundaries as in the previous example.

Example 3 (fig. 5-6):

Two Hitachi HM514800 512K x 8's are interfaced to the H8/300H via a 16-bit data bus. The RAS, CAS and WE signals are provided as described in the previous examples. The DRAMs use a 10 row x 9 column addressing mode. One device will store the upper bytes, and the other will contain the lower bytes. A0 is not connected since word accesses are performed. Therefore, the row address will be provided through line A1 - A9 and A19, while the column address via lines A1 - A9. The memory is mapped identically to the previous examples.

Example 4 (fig. 7-8):

Two Hitachi HM514800 512K x 8's are connected to the H8/300H via an 8-bit data bus. The CAS and WE signals are provided as described in the previous examples. Since the first DRAM will contain the 512KBytes between H'600000 and H'67FFFF, and the second device will contain the remaining 512KBytes between H'680000 and H'6FFFFF, additional decoding logic is necessary. The first device must be enabled for either byte transfers (if A19 = 0 and CS3 goes low), or for refresh cycles (if A19 = 0 and both CS3 and RFSH to low). The second DRAM must be enabled for byte transfers or refresh cycles if A19=1. The address lines are connected as described in example 1.

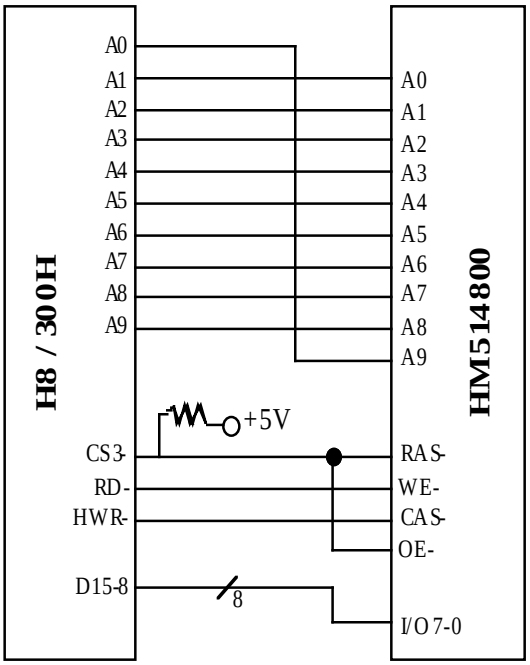


Fig. 2. DRAM Connection.

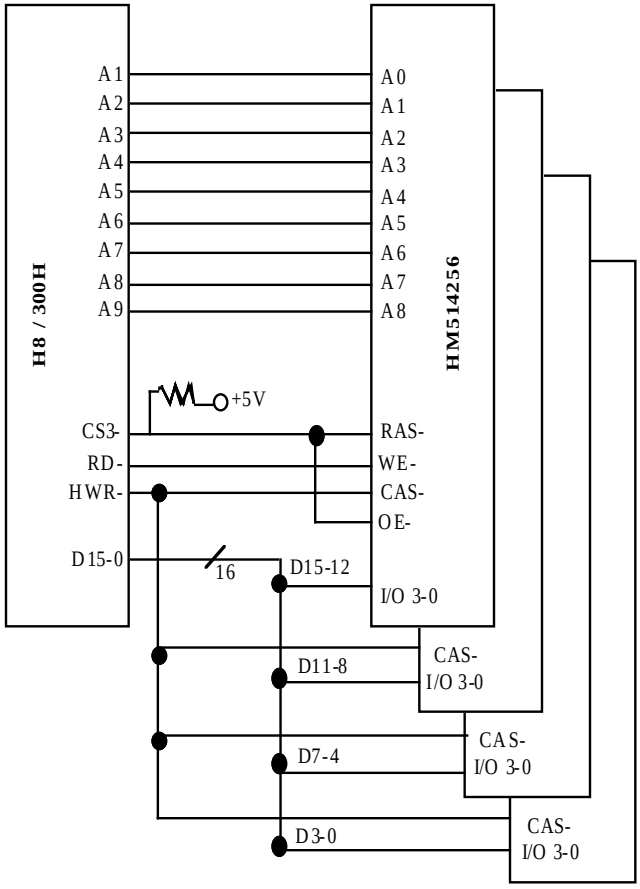


Fig. 4. DRAM Connection

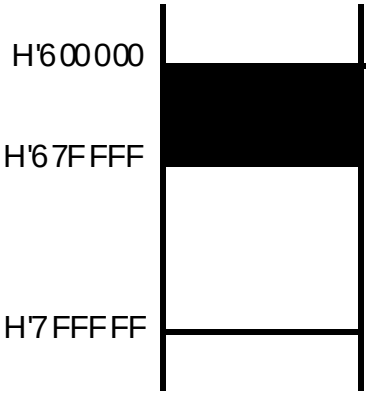


Fig. 3. Memory Mapping.

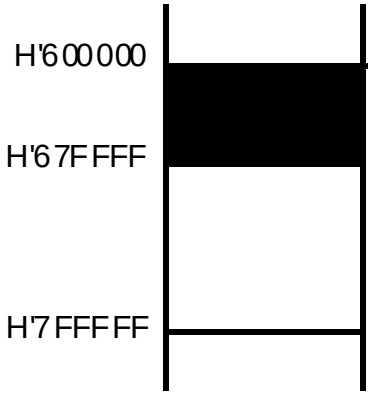


Fig. 5. Memory Mapping

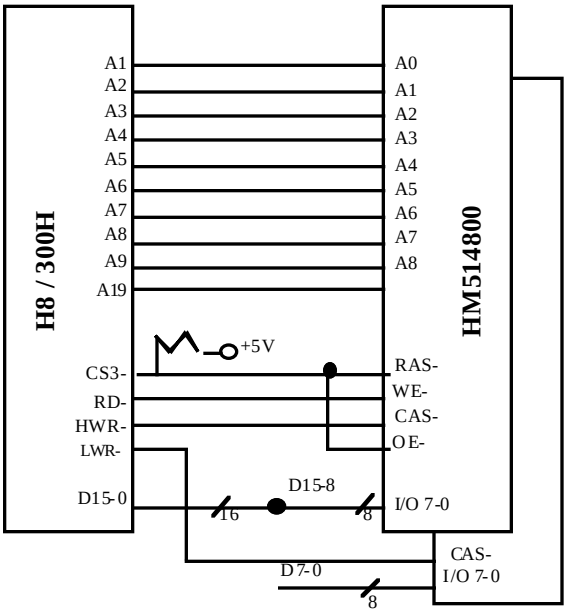


Fig. 6. DRAM Connection

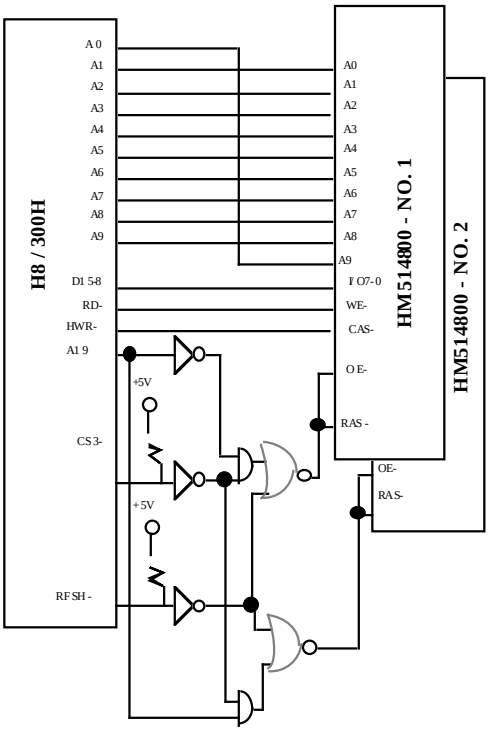


Fig. 8. DRAM Connection

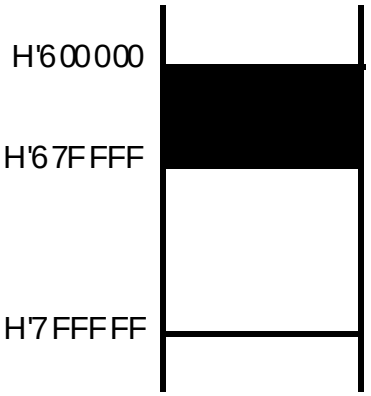


Fig. 7. Memory Mapping.

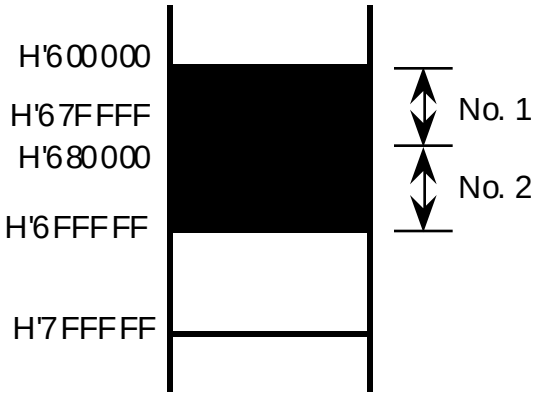
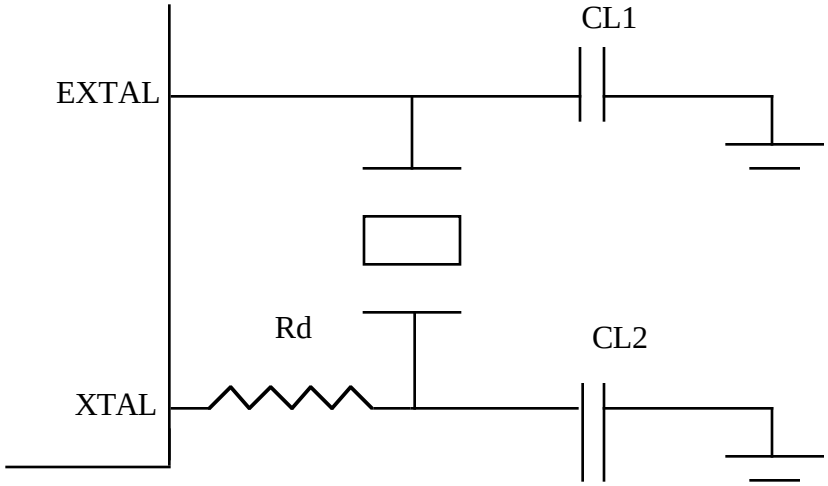


Fig. 9. Memory Mapping.

Adding the damping resistor to the oscillator line of the H8/3003

The H8/3003 hardware manual, ADE-602-055 (O), talks about a damping resistance Rd in connection with the crystal resonator on page 506, section 16.2.1 and table 16-1. However, this resistance is not shown in figure 16-2. The damping resistance Rd should be placed between the XTAL and the bottom end of the crystal as shown in the figure below:



HITACHI

Enabling DMA end-of-transfer interrupts on the H8/300H

All members of the 16-bit H8/300H family feature an on-chip DMA controller providing fast and direct data transfers between memory and I/O locations or between memory locations without CPU involvement. A maximum of 8 DMA channels can operate simultaneously in memory-to-I/O transfer mode, and a maximum of 4 DMA channels can operate simultaneously in memory-to memory operation mode (on the H8/3003). For each DMA mode of operation, an end-of-transfer interrupt (DEND) may be requested upon a byte (or word) transfer completion or, if the block transfer DMA mode is used, upon transfer completion of a memory block of data. In order to ensure proper operation, the user must adhere to a sequenced setup procedure for each DMA mode of operation, as shown in the hardware manuals. However, extra care must be exercised when enabling the end-of-transfer interrupts in each mode of operation.

The Data Transfer Control Register (DTCR) in the memory-to-I/O modes, and its counterpart in memory-to-memory modes (DTCRA), contain 2 control bits that determine a DMA-end interrupt is correctly issued. These are are the data transfer enable (DTE) bit, and the data transfer interrupt enable (DTIE) bit. The function of the DTE bit is to enable or disable data transfer, and the function of the DTIE bit is to enable or disable a DEND interrupt at the end of the transfer. According to the setup procedure for each DMA transfer type outlined in the H8/3003 hardware manual, the DTE bit should be cleared and the DTIE bit should be set (thus enabling a DEND interrupt at end-of-transfer) by writing the DTCR (or DTCRA) register in one write cycle. This step must be undertaken although the DTE bit has an initialization value of 0.

If, during the setup procedure, the user programs the DTCR (A) with the DTIE bit cleared to 0 (thus initially disabling end-of-transfer interrupts) and, later on in the program, after a series of DMA transfers have been executed, the user decides to enable end-of-transfer interrupts by setting the DTIE bit to 1 (using the BSET instruction), an end-of-transfer DEND interrupt will be issued right away before the DMA transfer starts. In order to avoid this situation, 2 options are available:

1. Clear the DTE bit again, as the DTIE bit is set in the same write cycle (that is, use a MOV instruction).

2. Set the DTIE bit right after the DTE bit is set, that is after the DMA transfer is enabled.

Also, the programmer must make sure that the DTCR (or DTCRA) must be read while the DTE bit is cleared before the DTE bit is set to start the DMA transfer. Using bit manipulation instructions to program any bits in DTCR (A) except DTIE should take care of reading the register, Alternatively, a MOV instruction can be used to write the content of DTCR into a CPU register, and then write it back to DTCR.

Multiplexed I/O Functions on the H8/3003 and H8/304X

The H8/300H microcontroller family combines a 32-bit register architecture CPU with high peripheral function integration in a small, 112-pin QFP package. Consequently, the need of multiplexing signals belonging to various on-chip modules arises. Virtually all I/O Port Pins are multiplexed with 1, 2 or 3 other input/output signals that control various peripheral and CPU functions. The H8/3003 has 9 I/O Ports with a total of 58 I/O pins, and the H8/3042 has 11 I/O Ports with a total of 78 I/O pins. All but 2 I/O bits are multiplexed on the H8/3003, and all I/O lines of the H8/3042 have at least dual functions. Although the distribution of peripheral function signals among the available I/O pins have at least dual functions. Although the distribution of peripheral function signals among the available I/O pins has been carefully optimized, the high degree of pin function-sharing poses some constraints on the operation conditions of the chip. This technote will elucidate these constraints for each I/O Port. Also refer to Appendix C - I/O Port Block Diagrams in the Hardware Manual for more information on the I/O Ports structure.

Port 8 (lines 0-3):

P80/RFSH- /IRQ0-
P81/CS3- / IRQ1-
P82/CS2- / IRQ2-
P83/CS1- / IRQ3-

If the corresponding interrupt enble bit for IRQ1, IRQ2 and/or IRQ3 in the Interrupt Enable Register (IER) is set (and the corresponding I-bit in CCR is cleared) while the microcontroller is interfaced to external devices mapped in memory area 3, 2 and/or 1 respectively, an interrupt request will occur every time the corresponding chip select signal is asserted. The reason is that the output chip select signal is fed back to the interrupt controller inputs through a non-tristatable buffer and an inverter, as illustrated in the Port 8 Block Diagram of Appendix C in the Hardware Manual. Since in most cases this situation is undesirable, a user cannot employ an external interrupt signal if it is multiplexed with a chip select signal used to address an external device. Similarly, IRQ0- cannot be used if the microcontroller is interfaced to PSRAM or DRAM devices that uses the RFSH- line for its refresh cycle.

Port 9 (lines 4 and 5):

P94 / SCK0 / IRQ4-
P95 / SCK1 / IRQ5-

If the corresponding interrupt enable bit for IRQ4 and/or IRQ5 in IER is set (and the I-bit in CCR is cleared) while the SCI is driven from an external serial clock or outputs the internal serial clock, an interrupt request will occur on each falling edge of low level at the SCK0 and/or SKC1 pins respectively (and regardless of the DDR-bits settings). In order to avoid these occurrences, and external interrupt cannot be used if it is multiplexed with an I/O pin used for the SCI serial clock.

Port A (lines 0-7):

PA0 / TP0/ TEND0- / TCLKA
PA1 / TP1 / TEND1- / TCLKB
PA2 / TP2 / TIOCA0 / TCLKC
PA3 / TP3 / TIOCB0 / TCLKD
PA4/ TP4 / TIOCA1 / A23 (only for H8/304X in modes 3 and 4)
PA5 / TP5 / TIOCB1 / A22 (only for H8/304X in modes 3 and 4)
PA6 / TP6 / TIOCA2 / A21 (only for H8/304X in modes 3 and 4)
PA7 / TP7 / TIOCB2 / A20 (only for H8/304Z in modes 3 and 4)

If the Integrated Timer Unit (ITU) is configured for external clock input via TCLKA or TCLKB, and the DMA module is activated by external request and uses signals TEND0 or TEND1 as end-of-DMA cycle flags, the ITU will be driven from the corresponding TEND signal (and regardless of the DDR-bits settings). Since this situation is not desirable in most applications, the ITU should not be driven by an external clock if it is multiplexed with an I/O oun used by the DMA controller to provide an end-of-cycle flag.

The ITU should not be set to use TCLKC or TCLKD as external clock inputs while using TIOCA0 or TIOCB0 respectively, as compare-match outputs since the corresponding I/O Port outputs are fed back to the counter clock inputs (see Appendix C in the Hardware Manuals). If the ITU is configured to use TCLKC or TCLKD while set up for input-capture trigger action at TIOCA0 or TIOCB0 respectively, an input-capture event will occcur on the selected exernal clock transitions. Unless this particular situation is needed, do not set up the ITU to be driven from an external clock input pin that is multiplexed with an input-capture pin used the timer network.

The Timing Pattern Controller (TPC) cannot use ouputs TP2-3 to be triggered by an ITU input-capture event at pins TIOCA0 or TIOCB0 (if TPC is operating in the non-overlapping mode) since TP2 and TP3 are function multiplexed with TIOCA0 and TIOCB0 respectively. Likewise , outputs TP4 and TP6 cannot be triggered by input-capture action at pins TIOCA1 and TIOCA2 respectively because they are function-multiplexed on the same I/O pins. Also, outputs TP5 and TP7 cannot be utilized if input-capture occurs at TIOCB1 and TIOCB2 respectively, if TPC is operating in the non-overlapping mode.

Also, the ITU cannot be driven from an external clock via TCLKD, TCLKC, TCLKB, or TCLKA if TPC outputs are desired from TP3, TP2, TP1, or TP0 respectively.

Port B (lines 0-3, 6-7):

PB0 / TP8 / TIOCA3
PB1 / TP9/ TIOCB3
PB2 / TP10 / TIOCA4
PB3 / TP11 / TIOCB4
PB6 / TP14 / DREQ0-
PB7 / TP15 / DREQ1- / ADTRG-

The TPC cannot use outputs TP8 or TP10 if they are triggered by an input-capture action via TIOCA3 or TIOCA4 respectively, since the same pins are used for both functions. Similarly, ouputs TP9 or TP11 cannot be used if they are triggered by an input-capture action via TIOCB3 or TIOCB4 respectively (and if the TPC is operating in the non-overlapping mode).

If the TPC is configured to use outputs TP14 and/or TP15 while the DMA controller is set up to be activated upon an external trigger at DREQ0- or DREQ1-, DMA transfer operations will occur every time a high-to-low transition happens at the corresponding TP outputs (since the I/O port output is fed back to the input line (see Appendix C in the Hardware Manual).

If the DMA controller is configured to perform transfers upon a falling edge at DREQ1- while the A/D converter is setup to accept start-of-conversion upon a falling edge at ADTRG-, both operations will start simultaneously since they are triggered at the same pin (PB7).

Port C (lines 2-5) and only for H8/3003:

PC2 / TEND2- / CS4-
PC3 / DREQ2- / CS5-
PC4 / TEND3- / CS6-
PC5 / DREQ3- / CS7-

If the DMA controller is set up to be activated upon an external trigger at the DREQ2- or DREQ3- pins while the microcontroller is interfaced to external devices mapped in memory areas 5 or 7 respectively, DMA transfers will occur every time the corresponding multiplexed chip select signal (CS5- or CS7-) is asserted. This situation can be avoided by using only one of the pin functions. Similarly, if the DMA uses the TEND2- or TEND3- pins to signal end-of-transfer, the microcontroller should not be interfaced to external devices mapped in memory areas 4 and 6, and vice versa.

H8-300 'C' Code Framework Example Program

This is a 'Starter' example program for the H8-300 family of processors, and in particular the H8-325. It is intended to indicate a typical framework program which can be run on a 300 series processor. The code performs a number of functions which are described in the code header. The idea is that, if the code is thought relevant, then it can be added to or modified, to produce a user application. In addition the code can be compiled, linked, down loaded to a suitable emulator and run. The code header describes a number of simple functions that are performed in the code which can be checked using an oscilloscope on the emulator header, no target hardware is required.

The code can be compiled and linked without an alteration and the following compile and link command lines can be used:-

To Compile

icch8300 h325frm -ms -r -L -q -s -C -P

This compiles code for the small memory model (-ms), produces a list file which 'C' and assembler are interspersed (-L -q), includes debug information (-r), optimises for speed (-s), allows nested comments (-C) and generates promable code (-P).

To Link

xlink h325frm -f h325 -r -o h325frm -l h325frm.map -xsme

This links the code according to the memory assignments in the h325.xcl memory mode (-f h325.xcl), produces debug information (-r), outputs a down loadable debug file h325frm.d20 (-o h325frm) and generates a map file h325frm.map (-l h325frm.map -xsme).

The h325frm.d20 file can be downloaded to an emulator such as a PCE that is running 'C' Spy, ISDT or CIDE. If required to run under ICE the h325frm.obj file is used.

For further details on compiling and linking please refer to the H8 C Tools Tutorial for the H8/300 family.

The h325.xcl file which is based on the small memory model xcl file , LNKH83S.XCL, is shown below.

```
 -!                -LNKH83S.XCL-

    XLINK 4.xx command file to be used with the H8300 C-compiler V2.xx
    using the -ms option (small stack memory model).
    Usage: xlink your_file(s) -f lnkh83s -o your_output_file

    First define CPU  -!

-cH8300

-!  Allocate segments which should be loaded -!

-Z(BIT)BITVARS=0

-!  First allocate the read only segments . -!

-Z(CODE)INTVEC=0
-Z(CODE)IFLIST,RCODE,CODE,CDATA,ZVECT,CONST,CSTR,CCSTR=100

-!  Then the writeable segments which must be mapped to a RAM area
    C000 was here supposed to be start of RAM -!

-Z(DATA)DATA,IDATA,UDATA,ECSTR,WCSTR,TEMP,CSTACK+40=FB80
-Z(DATA)SHORTAD=FF00-FF7F
-!  NOTE: In case of a RAM-only system, the two segment lists may be
    connected to allocate a contigious memory space. I.e. :
            -Z...CCSTR,DATA...=start_of_RAM                -!

-! See configuration section concerning printf/sprintf -!
 -e_small_write=_formatted_write

-! See configuration section concerning scanf/sscanf -!
 -e_medium_read=_formatted_read

-!  Now load the 'C' library -!
clh83s          -! or clh83sd for -2 option -!

-! Code will now reside in file aout.a20 in Motorola 'S' format -!
```

```
/*
```

H8-300 Framework Program

This application code example is intended for use as a framework program for H8-300 series microcontrollers. The program has the following functions:

1.0 Provides a 5ms, 50ms, 100ms, 250ms and 500ms scheduler based on a 5ms interrupt from the 8bit timer.

2.0 Initialistation functions for the 8bit timer and serial port 1.

3.0 A simple state machine skeleton utilising switch statements.

The example is based on the H8-325, but is relevant for most H8-300 microcontrollers. The code can be compiled, linked and downloaded to a PCE or LEV. To prove that everything is OK, portx, pin x is toggled every 5ms from the scheduler, resulting in a 100Hz waveform. The H8-325 is working in single chip, mode 3, at an internal system clock frequency of 10MHz.

Ver 1.0 ---------- 17 April 94.

```
*/
#pragma language = extended     /* extentions to Ansii 'C' */

/*#include "ioh8325.h"    /* port definitions */*/
#include "ioh8sfr.h"     /* special function register definitions */
#include "inth8325.h"    /* interrupt vector identifiers */

/* Typedefs ! */

/* The StateN would be replaced by user description */

typedef enum { State1, State2, State3, State4 } State;

/* Function prototypes */

void    Initialise(void);
void    Init_Ports(void);
void    Init_Timer_0(void);
void    Init_Serial_0(void);
State   State_Mc(void);
void    Mainloop(void);
void    User_5ms_Func1(void);
void    User_5ms_Func2(void);
void    User_5ms_Func3(void);
```

```
void    User_50ms_Func1(void);
void    User_50ms_Func2(void);
void    User_50ms_Func3(void);
void    User_100ms_Func1(void);
void    User_100ms_Func2(void);
void    User_100ms_Func3(void);
void    User_250ms_Func1(void);
void    User_250ms_Func2(void);
void    User_250ms_Func3(void);
void    User_500ms_Func1(void);
void    User_500ms_Func2(void);
void    User_500ms_Func3(void);

/* Definitions */
#define OSC                 20                      /* 10 MHz system clock */
#define BAUD_RATE           31250
#define BAUD_VALUE          (OSC * 1000000/(64*BAUD_RATE)-1) /* for n=0 */

/* Global variables - Note all globals are of format g_xxxxx */

unsigned char g_rxd_buffer0[32];

unsigned char g_Message[] = "Faster than a speeding bullit !";

void main(void)
{

Initialise();
Mainloop();

}


void Mainloop(void)
{
/* This is were the main code is called */
/* Scheduler interrupts in background */

while (1)
        {
        /* infinte loop calling user functions */
        /* user loop functions called here ..... */



        }

}

void Initialise(void)
/* This routine calls the peripheral initialisation routines and
   enables the global interrupt bit.
*/
{
Init_Ports();                   /* Initialise output ports */
```

```
Init_Timer_0();              /* Initialise 8 bit timer 0 */
Init_Serial_0();             /* Initialise Serial port 0 */
set_interrupt_mask(0);       /* enable global interrupts - this is a pre-defined
                                assembler function */
}



void Init_Ports(void)
{
/* Set Port 4 & 7 as outputs */

P4DDR = 0xFF;
P7DDR = 0xFF;

/* Tx and Rx port pins set automatically */
}



void Init_Timer_0(void)
{
/* set up to produce a 5ms interrupt on compare match A of 8 bit timer 0 */

TMR0_TCR = 0x4B;     /* int on CMIEA, clear on A, clock/1024 */
TMR0_TCSR = 0x00;    /* no change on compare match */
TMR0_TCORA = 48;     /* value to give compare match every 5.0176mS */
}


void Init_Serial_0(void)
{
/* This initialises the serial port to asynch mode, 8bit data, no parity,
   1 stop, 1 start and 31250 baud. An interrupt will be generated on
   character receive but not on transmit.
*/
SCI0_SMR = 0;
SCI0_SCR = 0x70;       /* Rxd int enable, Rxd, Tx enabled, int clock */
SCI0_BRR = BAUD_VALUE;

}



State State_Mc(void)
{
/* This is a simple switch based state machine. Typical use, keyboard
   decode, structured decision making - more ordered than 'if', 'else'
   structure.
*/

static State l_state;
```

```
switch (l_state)

        {
        case State1:    /* user function and new state
                        l_state = StateN */
                        break;

        case State2:    /* user function and new state
                        l_state = StateN */
                        break;

        case State3:    /* user function and new state
                        l_state = StateN */
                        break;

        case State4:    /* user function and new state
                        l_state = StateN */
                        break;

        /* .... and any number of unit states */

        default:        /* default state - error handler ! */
                        break;

        }

}



interrupt [TMR_CMI0A] Scheduler(void)
/* Note - variables cannot be passed or
   returned on interrupt functions */
{

static unsigned char t50ms=9, t100ms=1, t250ms=4, t500ms=1, index=0;

/* clear compare match flag */

/* Use 'sfr' keyword for bit access to timer control status register */
TMR0_TCSR.6 = 0;

/* This interrupt is called every 5ms */

/* user functions to be called every 5ms - note combined run time
   of these functions must be < 5ms */

User_5ms_Func1();
User_5ms_Func2();
User_5ms_Func3();

/* Toggle port pin for square wave output at 100Hz (10mS) */

P4DR.7 = (!P4DR.7) ? 1 : 0;     /* toggle pin */
```

```
        /* Test routine - this routine outputs a message from the transmit
           of serial port0 to the receive of serial port 0 (the tx and rx
           pins required to be linked). A character is output every 5ms.
           Delete this routine as required */

while (!SCI0_SSR.7);    /* wait until tx buffer free */
SCI0_TDR = g_Message[index++];
SCI0_SSR.7 = 0;         /* reset TDRE flag */

if (!g_Message[index]) index=0;



if (!t50ms)
        {
        /* user functions to be called every 50ms - note combined run time
        of these functions must be < 50ms */

        /* 50ms tick */

        User_50ms_Func1();
        User_50ms_Func2();
        User_50ms_Func3();

        /* reload 50ms tick */
        t50ms = 9;

        /* user functions to be called every 100ms - note combined run time
        of these functions must be < 100ms */

        /* 100ms tick */

        if (!t100ms)
                {
                /* Toggle port pin for square wave output at 5Hz (200mS) */

                P4DR.5 = (!P4DR.5) ? 1 : 0;     /* toggle pin */


                User_100ms_Func1();
                User_100ms_Func2();
                User_100ms_Func3();

                /* reload 100ms tick */
                t100ms = 1;

                /* user functions to be called every 500ms - note combined
                run time of these functions must be < 500ms */

                /* 500ms tick */

                if (!t500ms)
                        {
                        User_500ms_Func1();
                        User_500ms_Func2();
```

```
                User_500ms_Func3();

                /* reload 500ms tick */
                t500ms = 4;
                }
            else t500ms--;

            }
        else t100ms--;

        /* user functions to be called every 250ms - note combined run time
        of these functions must be < 250ms */

        /* 250ms tick */

        if (!t250ms)
            {

            User_250ms_Func1();
            User_250ms_Func2();
            User_250ms_Func3();

            /* reload 250ms tick */
            t250ms = 4;

            }

        else t250ms--;
        }

else t50ms--;

}



interrupt [SCI_RXI0] Receive_Port0(void)
/*
        Interrupt handler for receive interrupt generated by port 0
*/
{
static unsigned char Index=0;

/* read character into circular buffer */

g_rxd_buffer0[Index++] = SCI0_RDR;

/* Reset RDRF flag - using sfr structure */
SCI0_SSR.6 = 0;

if (Index >= 32) Index = 0;

}
```

```
/* User Functions */

void    User_5ms_Func1(void){}
void    User_5ms_Func2(void){}
void    User_5ms_Func3(void){}
void    User_50ms_Func1(void){}
void    User_50ms_Func2(void){}
void    User_50ms_Func3(void){}
void    User_100ms_Func1(void){}
void    User_100ms_Func2(void){}
void    User_100ms_Func3(void){}
void    User_250ms_Func1(void){}
void    User_250ms_Func2(void){}
void    User_250ms_Func3(void){}
void    User_500ms_Func1(void){}
void    User_500ms_Func2(void){}
void    User_500ms_Func3(void){}
```

Sample program to illustrate the low power modes and LCD drive of the H8-38XX family of microcontrollers

This is an example program to implement a simple counter using 5 character, seven segment LCD display driven directly from the onboard LCD drive on the H8-38XX family of microcontrollers. The low power features of the controller are also demonstrated by performing direct transfers from active high speed to active medium speed to sub-active and back to active high speed. As would be expected the counter updates at a speed dependent on the mode of operation.

The program source is supplied on disk complete with a sample 'xcl' for linking.

```
 /*********************************************************

LCD Counter Using the H8/3834 Microcontroller in three different operating modes:

        Full_speed - running off the system clock
        Active_medium - 1/8 of system clock
        Sub_active - 32KHz sub_clock

  The effect of the different modes is to cause the counter to count at different rates.

*********************************************************/

 /* Version 1.0 */

#include "c:\icch8300\inc\ioh83834.h"   /* H8-3834 IO labels */

#pragma language=extended

interrupt [0x28] void direct_transfer(void);

void active_medium(void);
void sub_active(void);
void full_speed(void);


void pause(long delay);

/* ********* */
/* Variables */
/* ********* */
#pragma memory = dataseg(LCDRAM)        /* Define LCD RAM area as
```

```
                                                relocatable code */
char ram[20];

#pragma memory = default

char value[] ={0x3f,0x0c,0x76,0x5e,0x4d,0x5b,0x7b,0x0e,0x7f,
                    0x5f,0x3f};

char dig1,dig2,dig3,dig4,dig5;

main()
{
        RLCTR = 0xfc;  /* locate LCD Ram in area 0xF740 to 0xF77F */
        LPCR = 0x0f;    /* No expansion, but all segments in use. static
          duty */
        LCR = 0xf0;      /* resistor ladder on, lcd controller on, blank off,
                            frame frequency = 128Hz */

        dig1=0;
        ram[0]=ram[1]=ram[2]=ram[3]=ram[4]=value[dig1];

        IENR2 = IENR2 | 0x80; /* Interrupt enable register 2,
                                 Enable direct transfer interrupt mode */

        TMA = TMA | 0x08;       /* Timer Mode register A ,set for 1 second
                                   time base interrupt */

        set_interrupt_mask(0);     /* Clear I bit */



        for (;;)
        {
        active_medium();            /* Change to active medium mode */
        sub_active();               /* Change to sub_active mode */
        full_speed();               /* Change to full_speed (active full) */
        }
}


interrupt [0x28] void direct_transfer(void)
/* Direct transfer available between high, medium and sub-active mode
   by executing the sleep instruction when the DTON bit in SYSCR2 is set.
   The sleep instruction generates an interrupt that causes the processor
   to change to the next mode.
*/
{
                IRR2 = IRR2 & 0x7f;  /* Clear interrupt request flag */
}



void active_medium(void)
{
```

```
/* set for transistion from active high speed to active  medium speed mode via sleep mode */

SYSCR1 = SYSCR1 & 0x77;    /* clear SSBY & LSON bits */
SYSCR2 = SYSCR2 | 0x0c;    /* set MSON & DTON bits */
sleep();                   /* Initiate mode change */
pause(12000);              /* count time */
}



void sub_active(void)
{
                /* set up for transistion from active medium speed to
                   subactive mode via watch mode */

SYSCR1 = SYSCR1 | 0x88;    /* set SSBY & LSON bits */
sleep();                   /* Initiate mode change */
pause(160);
}




void full_speed(void)
{

 /* Transistion from sub-active to active high speed via watch mode */

SYSCR1 = SYSCR1 | 0x80; /* set SSBY bit */
SYSCR1 = SYSCR1 & 0xf7; /* reset LSON bit */
SYSCR2 = SYSCR2 & 0xFB; /* reset MSON bit */

sleep();          /* change to high speed mode via watch after delay set by bits
                     STS2 to STS0 in SYSCR1 - 8192 states */

pause(60000);
}



void pause(long delay)
{
long loop;

for (loop=0;loop<delay;loop++)
        {
        /* While pausing in mode, update the LCD counter */
        ram[4] = value[++dig1];
        if (dig1==10) { dig1=0; ram[3]=value[++dig2]; }
        if (dig2==10) { dig2=0; ram[2]=value[++dig3]; }
        if (dig3==10) { dig3=0; ram[1]=value[++dig4]; }
        if (dig4==10) { dig4=0; ram[0]=value[++dig5]; }
        if (dig5==10) { dig5=0; }
```

```
        }
}
```

Memory Checking and Initialisation Program for the H8 Range of Microcontrollers

This application code is designed to run on any Hitachi H8 Microcontroller compiled under IAR C compiler. The source code is supplied on disk with a sample 'xcl' file to enable linking.

The code has been written to check an area of either 8 or 18 bit memory and then clear to zero. The memory check is performed by writing then reading the Hex values 55 and AA to the specified memory area. The check is continued until the memory area has been completed or an error has been found. If an error is found the address and the data pattern is reported. Upon a successful memory check the memory area is initialised to 0.

Program Details:

Function:- mem_status *Chk_Clr_Mem(unsigned int, unsigned int)

Passed: Start address and end address of memory area to be checked.

Returns: pointer to structure mem_status.

```
/*
    This function checks and then clears an area of memory
    bounded from a start address to an end address. The
    start and end address are passed in the function call.
    A pointer to a structure is returned which reports on
    the result of the memory check and initialisation.

    18 July 1994.

*/

typedef unsigned char Bool;

/* define state values used in memory check and initialisation routine */

typedef enum { END=0, WRITE_55, CHECK_MEM_55, WRITE_AA, CHECK_MEM_AA,
               PASS, FAIL, CLEAR } memory_state;

/* define structure for memory status after memory area check */
```

```
typedef struct
        {
        unsigned int address;
        unsigned int value;
        Bool         status;
        } mem_status;

mem_status *Chk_Clr_Mem(unsigned int,  unsigned int);

/*mem_status memory;*/


void main(void)       /* This was used to test the function */
{
static mem_status *mem_results;

mem_results = (mem_status *)Chk_Clr_Mem(0x8000, 0xA000); /* check this area */

}



mem_status *Chk_Clr_Mem(unsigned int start, unsigned int end)
/* Function to check and clear an area of memory. Passed start and end
   address of memory area to be checked. Returns a pointer to a structure
   that contains the status of the check, the address at which it failed
   and what value it failed on. If the check has passed than the address
   and data value are not required.

*/
{
static unsigned char *start_ptr;

static memory_state memory_check;

static mem_status memory;       /* declare return structure */

/* Perform memory check by writing and reading 0x55 and 0xAA
   to specified areas */

start_ptr = (unsigned char *)(start);


/* state machine for checking memory */

memory_check= WRITE_55;              /* initial state */

while (memory_check)
{
switch (memory_check)

        {

        case WRITE_55:          /* write data to memory */
```

```
                while (start_ptr <= (unsigned char *)end)
                        *(start_ptr++) = 0x55;

                /* re-load start pointer */
                start_ptr = (unsigned char *)(start);

                /* next state */
                memory_check = CHECK_MEM_55;

                break;


case CHECK_MEM_55:      /* read memory and compare with 0x55 */

                while ((start_ptr <= (unsigned char *)end) &&
                     (*(start_ptr++) == 0x55));
                /* if pointer did not reach end error detected */

                /* next state ? */
                memory_check = (--start_ptr == (unsigned char *)end) ?
                                                        WRITE_AA : FAIL;

                memory.value = 0x55;
                break;


 case WRITE_AA:         /* write data to memory */

                start_ptr = (unsigned char *)(start);

                while (start_ptr <= (unsigned char *)end)
                        *(start_ptr++) = 0xAA;

                /* re-load start pointer */
                start_ptr = (unsigned char *)(start);

                /* next state */
                memory_check = CHECK_MEM_AA;

                break;


case CHECK_MEM_AA:      /* read memory and compare with 0xAA */

                while ((start_ptr <= (unsigned char *)end) &&
                     (*(start_ptr++) == 0xAA));
                /* if pointer did not reach end error detected */

                /* next state ? */
                memory_check = (--start_ptr == (unsigned char *)end) ?
                                                        PASS : FAIL;
                memory.value = 0xAA;
                break;
```

```
        case PASS:              /* return status */
                                memory.status =0;       /* Pass */
                                memory_check = CLEAR;
                                break;



        case CLEAR:             /* clear memory */

                                start_ptr = (unsigned char *)(start);

                                while (start_ptr <= (unsigned char *)end)
                                        *(start_ptr++) = 0x00;
                                memory_check = END;
                                break;



        case FAIL:              /* return status */
                                /* where did it fail ? */
                                memory.address = (unsigned int)start_ptr;
                                /* what value ? */
                                memory.status =1;       /* Fail */
                                memory_check = END;
                                break;


        default:                break;

        }

}

return &memory;  /* return pointer to structure */

}
```

Interfacing to LCD Character Modules

Configuring the HD44780 LCD controller / driver which is built onto the range of Hitachi Character Liquid Crystal Display Modules.

The HD44780 gives the user the ability to display alphanumerics, Kana characters, symbols and also the facility to generate custom characters. The interface to a microcontroller/processor can be either 8-bit or 4-bit wide. The HD44780 can drive upto 16 characters, configured as 1 line by 16 characters or 2 lines by 8 characters. However, built into the device is a display data RAM area which can support upto 80 characters maximum, so by attaching 9 off HD44100's (40-channel segment driver) for a 1 line display with a duty factor of 1/8 or 1/11, or 4 off HD44100 for a 2 line display with a duty factor of 1/16 you will be able to drive upto 80 characters.

The functions below have been extracted from a program which is designed to drive any Hitachi Character LCD Module which has the HD44780 controller/driver built-in. This display module is controlled via the Hitachi H8/325 8-bit single chip microcontroller. To enable customers to fully evaluate the H8/325, we have produced a low cost evaluation board, the following code has been writen to run on this EV board (LEV8325). The displays control and data lines are controlled via a PIA.

```
/* This string of characters is to be displayed on the character display module. */

const char char_screen [ ] = {'T','h','i','s',' ',' C','h','a','r','a','c','t','e','r',' ',' M','o','d','u','l','e',' ',
'i','s',' ',' t','h','e',' ',' L','M','0','9','2','L','N',' ',' w','h','i','c','h',' ', 'i','s','',' a',' 2',' b','y',' 4','0', 'D','i','s','p','l','a','y',0x00,0x01,0x02};


/* This functon reads back the status of the data bus and is called during the 'Busy
Flag' check.*/

char read_lcd ( char reg )
{
char  temp, read_val;

write_E_port (PIA1_CRA,0);              /* enable DDRA access */
write_E_port (PIA1_DRA,0);              /* port A all inputs */
write_E_port (PIA1_CRA,0x04);           /* enable DRA access */
write_E_port (PIA1_CRB,0x04);
temp = read_E_port (PIA1_DRB);          /* read control signal values */
if ( reg )                              /* if reg is true then access lcd data reg */
   temp = temp | 0x03;                  /* set RS and R/W pins */
else
   {
   temp = temp | 0x02;                  /* set R/W pin only */
```

```
   temp = temp & 0xfe;                    /* ensure RS pin = 0 */
   }
write_E_port (PIA1_DRB,temp);             /* output control signals RS and R/W */
temp = temp | 0x04;                       /* set E clock pin */
write_E_port (PIA1_DRB,temp);             /* output E clock */
for ( read_val = 0; read_val <= 10; read_val ++ )
   { }                                    /* delay to allow data set up from lcd*/
read_val = read_E_port (PIA1_DRA);        /* get data */
temp = temp & 0x0b;                       /* clear E clock pin */
write_E_port (PIA1_DRB,temp);             /* output E clock */
temp = temp & 0xf8;                       /* clear RS and R/W pins */
write_E_port (PIA1_DRB,temp);
return (read_val);
}
```

/* This function is used to send data to the HD44780, to set up the various registers and also to send the character information to the display data RAM. */

```
void write_lcd ( char reg, char write_val )
{
char  temp,i;

write_E_port (PIA1_CRA,0);                /* enable DDRA access */
write_E_port (PIA1_DRA,0xff);             /* port A all outputs */
write_E_port (PIA1_CRA,0x04);             /* enable DRA access */
write_E_port (PIA1_DRA, write_val);       /* output the data */
temp = read_E_port (PIA1_DRB);            /* read control signal values */
if ( reg )                                /* if reg is true then access lcd data reg */
   {
   temp = temp | 0x01;                    /* set RS pin */
   temp = temp & 0xfd;                    /* clear R/W pin */
   }
else
   temp = temp & 0xfc;                    /* ensure R/W and RS pins = 0 */
write_E_port (PIA1_DRB,temp);             /* output R/W and RS signals */
temp = temp | 0x04;                       /* set E pin */
write_E_port (PIA1_DRB,temp);             /* output E signal */
for ( i = 0; i <=10; i++ )
   { } /* short delay */
temp = temp & 0xfb;                       /* clear E pin */
write_E_port (PIA1_DRB,temp);             /* output E signal */
temp = temp & 0xf8;                       /* clear RS and R/W pins */
write_E_port (PIA1_DRB,temp);             /* output RS and R/W signals */
}
```

/* This function checks the status of the 'Busy Flag'. If DB7 is High ('1'), this indicates that the HD44780 is busy processing the previous instruction. The controller can only accept the next instruction when DB7 is Low ('0').*/

```
void wait_lcd ( void )
{
while  (read_lcd (lcd_control) & 0x80)    /* wait for busy flag = '0' */
{ }
}
```

```
/* Before writing any screen data to the display you need to initialise it in terms of number of
display lines, cursor, font etc. This function sets up the controller to drive a 2 line display, 5*7
dots, cursor and blink ON, with increment. */

void set_up_lcd ( void )
{
char temp;
long i;

/* first set up the PIA */
write_E_port (PIA1_CRB,0x04);          /* allow access to DR */
temp = read_E_port (PIA1_DRB);
temp = temp & 0xf8;                    /* ensure RS, R/W and E are inactive */
write_E_port (PIA1_DRB,temp);
write_E_port (PIA1_CRB,0x00);          /* select access to DDRB */
write_E_port (PIA1_DRB,0x07);          /* RS, R/W and E to outputs */
write_E_port (PIA1_CRB,0x04);          /* allow access to DRB */
write_E_port (PIA1_CRA,0x04);          /* allow access to DRA */
write_E_port (PIA1_DRA,0x00);          /* data bus to 0 */
write_E_port (PIA1_CRA,0x00);          /* allow access to DDRA */
write_E_port (PIA1_DRA,0xff);          /* all port A are outputs */
write_E_port (PIA1_CRA,0x04);          /* allow access to DRA */
for (i = 0; i <= 10000; i++)
   { }

write_lcd ( lcd_control,0x38);         /* initialise display function set, need to
                                          send thrice to ensure correct initialisation */
for (i=0; i <= 10000; i++);            /* delay */
for (i=0; i <= 10000; i++);            /* delay */
write_lcd (lcd_control,0x38);          /* send again! */
for (i=0; i <=10000; i++);             /* delay */
write_lcd ( lcd_control,0x38);         /* and again! */
wait_lcd ( );                          /* now check 'Busy Flag' */
write_lcd (lcd_control,0x01);          /* clear display */
wait_lcd ();
write_lcd ( lcd_control,0x0f);                 /* display ON, cursor ON with Blink */
wait_lcd ();
write_lcd (lcd_control,0x02);          /* return to home position */
}


void custom ( void )
{
set_cgram_addr(0x40);                  /* address location for custom chars */
wait_lcd();                            /* check status of busy flag */
write_lcd(lcd_data,0x11),              /* 1st byte of bit pattern for  1st custom char */
write_lcd(lcd_data,0x0f),
write_lcd(lcd_data,0x0f),
write_lcd(lcd_data,0x11),
write_lcd(lcd_data,0x1e),
write_lcd(lcd_data,0x1e),
write_lcd(lcd_data,0x01),
write_lcd(lcd_data,0x00),

write_lcd(lcd_data,0x00),              /* 1st byte of bit pattern for  2nd custom char */
```

```
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x00),

write_lcd(lcd_data,0x1f),                    /* 1st byte of bit pattern for 3rd custom char */
write_lcd(lcd_data,0x15),
write_lcd(lcd_data,0x1f),
write_lcd(lcd_data,0x1b),
write_lcd(lcd_data,0x1f),
write_lcd(lcd_data,0x11),
write_lcd(lcd_data,0x1f),
write_lcd(lcd_data,0x00),
}

void main ( void )
{

int n;

set_up_lcd ();                               /* initialise the display */
wait_lcd ( );
n=0;
while ( char_screen[n] ) {
        write_lcd ( lcd_data, char_screen [n++]);    /* send the character  string to the
                                                        LCM*/
        }


}
```

H8/300H Software UART implementation using 2 timer and 2 DMA channels

Most variants of the H8/300H family of microcontrollers support two hardware UARTS at standard. However in a lot of detailed embedded applications today there is an increasing requirement for more than two. Detailed here is a method where a further full duplex UART channel can be implemented using two on board timers and two of the DMA channels. For the transmission side there is some overhead required to set up the transmission table and pattern and on the receive side there are two CPU interrupts, one at reception of the first start pulse edge (approximately 8µs) and one when the packet has been received (approximately 64µs).
The outer layer of the application code has been written in C but the bit bashing routines are shown here in assembler for speed purposes, but there is no reason why these cannot be converted to Ansi C.

Asynchronous Communications:

An asynchronous data packet will be of one of the following formats:

<start bit >	<data>		<stop bit>
<start bit>	<data>		<stop bit> <stop bit>
<start bit>	<data>	<parity bit>	<stop bit>
<start bit>	<data>	<parity bit>	<stop bit> <stop bit>

The data content may be, typically, between five and eight bits. Parity may be even, odd or disabled. There may be one or two stop bits. In standard use the user selects the format required for the communications protocol in use, and then the UART remains set at a standard format. This example assumes that the user has only one protocol required, and that this is known at compile time, thus the code may be written specifically for the format required, and may therefore have any generalisations removed to increase execution speed and reduce code overhead.

The selected packet format is: 9600 bps, 8 data bits, no parity, 1 stop bit.

To implement a UART two separate operations must be catered for:

• The UART must be able to create data packets on an I/O line which conform to the required bit rate and packet format.
• The UART must be able to detect transitions on a separate I/O line and disassemble the transitions into a valid data value.

Both the above operations require the UART simulation software be able to time I/O control at a rate which is proportional to the bit rate of the asynchronous interface. In addition data must be transferred to/from the I/O lines in synchronisation with the time steps. It is feasible to implement such timing and data transfer operations in a number of ways:

• CPU performs a timing loop and transfers data.
• Timers generate timeout interrupts, the CPU then transfers data on interrupts occurring.
• Timers generate timeout interrupts, data is transferred by DMA on timeout events.

The above three options represent a decreasing requirement on the CPU, with the timer/DMA option requiring CPU intervention only to set up and start transmission, or to disassemble the received data. It is this option which has been selected for this example, as it represents the best use of the H8/300H peripherals to minimise the CPU overhead in implementing a UART.

Transmitter Implementation:

The algorithm used to implement the transmission part of the software UART reduces the number of DMA transfers to the absolute minimum required, with the slight CPU overhead of creating a timer set up table before invoking the timer/DMA transmission system. The algorithm uses the following basic principle:

• if a bit pattern is exclusive-ORed with itself divided by 2, the resulting pattern represents the 0 to 1, or 1 to 0 transitions in the original bit pattern.

for example:

Original Pattern:	01110100
Pattern / 2	00111010
XOR	01001110

i.e. a bit transition occurs at (reading from left to right) bits 1, 4, 5 and 6.

This transition data may then be used to determine the exact time at which the transmit line must be toggled, which will not usually be on every bit of the

packet. Thus worst case, the transition method of forming data packets will tend to that of transferring bit data at the bit rate, while usually the number of transfers will be significantly reduced (four transitions in the above example).

The set up procedure for transmission is described below:

• Calculate the transition pattern.
• Form a table of transition times based upon the bit rate. The table should contain multiples of the bit rate count for the frequency of the timer channel used.
• Set the timer channel to compare-match on the first transition timeout and to toggle an I/O line on each compare-match event. The timer channel should also trigger a DMA transfer on each compare-match event.
• Set a DMA channel to transfer successive data from the transition data channel, to the compare-match register on each successive compare-match event, until all transitions have occurred.

Thus once the transition table has been created and the timer and DMA initialised, then the timer will control the I/O line using a compare-match event, and the DMA will reload the compare-match register automatically with the next compare-match value. When the required number of transitions have occurred the DMA can be used generate a CPU interrupt, which may then stop any further timer and DMA operations. This represents a completely self-contained system which will only interrupt the CPU on completion of transmission .

Receiver Implementation:

The receiver needs to detect the start of a data packet (denoted by the low start bit), it then must sample the receive data line at a minimum of the bit rate to determine the incoming data bits. It is recommend that the received data line is sampled at a frequency greater than the bit rate, so that more than one sample is available for each data bit. This will then enable the receiver to remove any spurious readings from the incoming bit stream, by taking the majority of the readings for each bit.

The implementation described below uses a sampling rate which is four times the bit rate. The start bit is detected by a timer channel using the input capture function. This may then be used to interrupt the CPU, which may then start the sampling process. A timer channel may then be used to set a sampling interrupt which may be used to trigger a DMA transfer from the receiver I/O line to a receive data buffer.

Having finished the sampling process the CPU may then examine the sampled data to determine the received value. This is accomplished by taking the majority level of each four samples as the received bit level. Once the data value has been determined any parity checking may be performed, if required.

To summarise the operations:

Setup - Timer & DMA

- Set the sample rate into a compare-match register.
- Enable input-capture interrupts for the timer channel
- start the timer channel
- set the source and destination DMA transfer addresses
- set the DMA transfer count
- set the DMA operation mode

On Input-capture Interrupt:

- Stop timer channel
- Disable input capture interrupt & clear flag
- Enable compare match-interrupts
- Enable DMA transfers on compare-match interrupt and enable DMA.
- Start timer channel.

On DMA completion interrupt:

- Stop the timer channel & clear flags
- Disable DMA transfers
- Disassemble receive buffer using majority samples to determine data bits.
- Set receiver flag and store data.

The above process may then be used to receive multiple data bytes at the desired bit rate.

Please note that this method uses two DMA channels to perform the data transfer, as normal address mode is required. If these are not available, the sample transfer could be implemented as a compare-match CPU interrupt. However, this would have more impact on CPU bandwidth.

UART design notes:

The UART uses the following hardware:

Transmitter -	One channel of the ITU
	One channel of the DMA
Receiver -	One channel of the ITU
	Two channels of the DMA.

However if the UART is not required to transmit at the same time as data may be received, then the total requirement for the UART will be one channel of the ITU

and two channel of the DMA.

In this example the UART has been set up assuming that reception and transmission will not occur simultaneously

UART Benchmark Timings:

The code was timed on an E7000PC emulator for the H8/3042 device in advanced single chip mode (mode 6) operating at 16MHz system clock. The functions/interrupt handlers are listed below with execution times. In addition timings were taken from the invocation of the transmit function, until the last data bit was transferred, and the handler returned. This will give an indication of the total transmit time which the CPU requires - please note that the CPU will not be performing this code exclusively during this period.

Function:	Time taken (µsec):
Receiver	64 µsec
Transmit	21 to 25 µsec
InitialiseUART	21 µsec
ITUCaptureInterrupt	8 µsec
DMAIsr	2 µsec

```
////////////////////////////////////////////////////////////
//
//  UART.C - software UART implementation for H8/3042
//
////////////////////////////////////////////////////////////

#include "ioh83042.h"  // I/O register defintions
#include "inh83042.h"  // Interrupt vector addresses
#include <inh83.h>     // In-line functions

#pragma language=extended

////////////////////////////////////////////////////////////
// System Constants

#define GRA0    0x6A
#define DRA     0xD3

////////////////////////////////////////////////////////////
// Union for C access to DMAC MAR registers

union u_tag {
  unsigned long full;
  struct w_tag {
    unsigned short hw;
    unsigned short lw;
  } word;
  struct b_tag {
    unsigned char rb;
    unsigned char eb;
    unsigned char hb;
    unsigned char lb;
  } byte;
} MAR;

////////////////////////////////////////////////////////////
//Global data and transmit/receive buffers

unsigned short dma_buff[20];
unsigned short period;
unsigned short dma_count;
unsigned char rx_buff[40];

////////////////////////////////////////////////////////////
// Assembler based fast comms routine

void Transmit (unsigned char);

////////////////////////////////////////////////////////////
// Globals used for Flags

unsigned char SCI2_SMR;
unsigned char SCI2_SSR;
unsigned char SCI2_SCR;
unsigned char SCI2_RDR;
```

```
//////////////////////////////////////////////////////////
// Function to set the ITU and DMAC channels to their initial
// settings, the ITU is run for a short time to allow a
// compare match to occur so that the output line may be set to
// a logic high.
//
// The transmitter uses ITU channel 0 (CMA) and DMA channel 0A
// (in I/O mode).
//
// The receiver uses ITU channel 1 (ICA then OCA) and DMA channels
// 1A and 1B (in block transfer full address mode)

void InitialiseUART (void)
{
  period = 417;          //sys/4 @16MHz -> 9600 bps
  //////////////////////////////////////////////////////////
  // set up for Transmit, initialise the timer output to high...
  ITU_TCNT0 = 0x0000;
  ITU_TCR0  = 0xA2;      //CCLRA, sys/4
  ITU_TIOR0 = 0x8A;      // output 1 on CMA
  ITU_GRA0  = 10;        // compare match value
  ITU_TSR0 &= 0xF8;      // clear flags
  ITU_TSTR |= 0x01;      // start the timer channel
  while ((ITU_TSR0 & 0x01) != 0x01); //wait on compare match
  ITU_TSTR &= 0xFE;      // stop channel 0 with output = 1
  ITU_TSR0 &= 0xF8;      // clear flags
  //////////////////////////////////////////////////////////
  //now set up the timer and DMA channels to perform the transmission
  ITU_TIOR0 = 0x8B;      // toggle O/P on CMA
  ITU_TIER0 = 0xF9;      // enable interrupts on CMA
  DMAC_IOAR0A = GRA0;    // set I/O destination to GRA0 register
  DMAC_DTCR0A = 0x40;    // word, inc MAR, I/O, CMA
  //////////////////////////////////////////////////////////
  // set up ITU for receive
  ITU_TCNT1 = 0x0000;    //reset counter
  ITU_TCR1 = 0xA0;       //CCLRA, sys
  ITU_TIOR1 = 0x8E;      //ICA falling edge
  ITU_TSR1 &= 0xF8;      //clear flags
  ITU_TIER1 = 0x01;      //enable interrupt on ICA
  //set up DMA for receive
  DMAC_DTCR1A = 0x07;    //byte, fixed source, block mode
  DMAC_DTCR1B = 0x19;    //inc dest, source is block, trigger on CMA1
  DMAC_ETCR1AH = 0x01;   //1 byte per block
  DMAC_ETCR1AL = 0x01;
  DMAC_ETCR1BH = 0x00;   //36 transfers to complete
  DMAC_ETCR1BL = 0x24;
  DMAC_MAR1AE = 0x00;    //port A is the source address
  DMAC_MAR1AH = 0xFF;
  DMAC_MAR1AL = DRA;
  MAR.word.lw = (unsigned short)&rx_buff; //set destination
  DMAC_MAR1BE = MAR.byte.eb;
  DMAC_MAR1BH = MAR.byte.hb;
  DMAC_MAR1BL = MAR.byte.lb;
}
```

```
////////////////////////////////////////////////////////////
// Interrupt service routine to fire off the sampling process
// for a packet of data, given that the low edge of a start
// bit has been detected by the input capture logic of the
// timer channel.

interrupt [ITU_IMIA1] void ITUCaptureInterrupt (void)
{
  ITU_TSTR &= 0xFD;     //stop channel 1
  ITU_TCNT1 = 0x0000;   //reset counter
  ITU_TIOR1 = 0x88;     //CMA, no output
  ITU_GRA1 =  417;      //sys @ 16MHz -> 4 * 9600 samples/sec
  DMAC_DTCR1A |= 0x80;  //enable DMA
  DMAC_DTCR1B |= 0x80;  //DMA master enable
  DMAC_DTCR1A |= 0x08;  //enable DMA interrupts
  ITU_TSR1 &= 0xF8;     //clear flags
  ITU_TSTR |= 0x02;     //start channel 1
}

////////////////////////////////////////////////////////////
// main function to test that the transmit and receive operations
// work - here the transmit line may be fed back to the recieve
// line to allow simultaneous transmission and receipt.

int main (void)
{
  int i;
  set_interrupt_mask(0);
  InitialiseUART();
  while (1)
  {
    Transmit(0x74);        // transmit the byte 0x74
    for (i=0;i<4000;i++);  // delay for a bit between packets
  }
  return (0);
}
```

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;
      ;; RXTX - assembler routines for H8/3042 software UART
      ;;
      ;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

       NAME rxtx

       CASEON


      PUBLIC  Transmit
      PUBLIC  Receiver
      PUBLIC  DMAisr
      EXTERN  period
      EXTERN  dma_buff
      EXTERN  dma_count
      EXTERN  rx_buff
      EXTERN  SCI2_RDR
      EXTERN  SCI2_SSR

       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       ;; Set the CPU control register constants

ITU_TCNT0       equ     H'FF68
ITU_GRA0        equ     H'6A
ITU_TCR0        equ     H'64
ITU_TIOR0       equ     H'65
ITU_TSR0        equ     H'67
ITU_TIER0       equ     H'66
ITU_TSTR        equ     H'60
DMA_MAR0AM      equ     H'20
DMA_MAR0AS      equ     H'21
DMA_MAR0AB      equ     H'22
DMA_MAR0AY      equ     H'23
DMA_IOAR0A      equ     H'26
DMA_ETCR0AH     equ     H'24
DMA_ETCR0AL     equ     H'25
DMA_DTCR0A      equ     H'27
DMA_DTCR1A      equ     H'37
DMA_DTCR1B      equ     H'3F


       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       ;; Bit constants

RXERFLG         equ     6



      RSEG CODE
```

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;; Transmit function to form transition table and then
      ;; set the ITU and DMA off to transmit the data packet
      ;;
      ;; Parameters: char to transmit will be in R6L
      ;;
      ;; Register usage:

Transmit:
      ;;set up a work environment
      PUSH.L  ER6
      PUSH.L  ER5
      PUSH.L  ER4
      PUSH.W  R3
      PUSH.W  R0

      MOV.B   R6L,R6H         ;get a copy of the input char
      SHLL.B  R6H             ;right shift
      XOR.B   R6L,R6H         ;form transition byte

      ;initialise data
      MOV.L   #dma_buff:32,ER4;buffer pointer
      SUB.L   ER5,ER5         ;zero out ER5
      MOV.B   #H'08,R3L       ;set loop count to 8
      MOV.W   @period:24,E6   ;load bit period to E6
      MOV.W   E6,R5           ;initial setting for dma timeout

      ;form timeout table
label02:
      SHLR.B  R6H             ;is there a transition ?
      BCC     label01
      MOV.W   R5,@ER4         ;if yes then buffer the time period
      INC.L   #2,ER4
      SUB.W   R5,R5           ;clear period count
      INC.W   #1,E5           ;inc count
label01:
      ADD.W   E6,R5           ;add next bit period to count
      DEC.B   R3L
      BNE     label02         ;do eight times...

      MOV.W   R5,@ER4         ;buffer the time period
      INC.W   #1,E5           ;inc count

label03:

      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;; Finally load the stop bit timeout into the dma table
      ;; here there is one stop bit in a packet
      INC.L   #2,ER4          ;point to next entry in the transition table
      MOV.W   E6,@ER4         ;set stop period in transition table
      INC.W   #1,E5           ;increment dma transfer count
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;now send it out using DMA & ITU

SUB.W   R0,R0                   ;load ITU_TCNT with 0
MOV.W   R0,@ITU_TCNT0:16
MOV.L   #dma_buff,ER4
MOV.W   #20,R0                  ;load GRA with start offset
MOV.W   R0,@H'FF6A:16
BCLR    #0,@ITU_TSR0:8          ;clear IMIA flag
MOV.B   R4H,@DMA_MAR0AB:8
MOV.B   R4L,@DMA_MAR0AY:8
MOV.W   E4,R4
MOV.B   R4L,@DMA_MAR0AS:8
MOV.W   E5,R5                   ;ECTR0A data =
MOV.B   R5H,@DMA_ETCR0AH:8      ;no of transfers
MOV.B   R5L,@DMA_ETCR0AL:8


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;finally start the timer channel 0 to tx the data
BSET    #0,@ITU_TSTR:8
BSET    #7,@DMA_DTCR0A:8        ;enb dmas
BSET    #3,@DMA_DTCR0A:8        ;enb dma ints

POP.W   R0
POP.W   R3
POP.L   ER4
POP.L   ER5
POP.L   ER6

RTS



 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
 ;; need to place received char is placed in SCI2_RDR

Receiver:

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; assume that the data is in rx_buff
; 4 samples per bit, 9 bits...
PUSH.L  ER0
PUSH.W  R1
PUSH.W  R2
PUSH.W  R6

MOV.B   #H'FD,R0L
MOV.B   R0L,@ITU_TSTR:8   ;stop the timer channel
BCLR    #3,@DMA_DTCR1A:8  ;disable DMA interrupts
BCLR    #7,@DMA_DTCR1A:8  ;disable DMA transfers
BCLR    #7,@DMA_DTCR1B:8  ;ditto
```

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;now disassemble the received data

      MOV.L   #rx_buff,ER0
      ADD.L   #4,ER0  ;zip past start bit
      SUB.B   R2H,R2H ;set bit count to 0
      SUB.B   R6L,R6L ;initilaise return value
receive6:
      SUB.W   R1,R1   ;initialise zero & 1 counts
      MOV.B   #04,R2L ;sample count
receive0:
      BTST    #4,@ER0
      BEQ     receive1
      INC.B   R1H     ;if its a 1 inc 1s count
      BRA     receive2
receive1:
      INC.B   R1L     ;if its a 0 inc 0s count
receive2:
      INC.L   #1,ER0  ;point to next item
      DEC.B   R2L     ;dec sample counter
      BNE     receive0
      CMP.B   #3,R1H
      BGE     receive3
      CMP.B   #3,R1L
      BGE     receive4
      BRA     rxerror
receive3:                      ;if majority was 1
      BSET    R2H,R6L         ;set bit in O/P data register
      BRA     receive5
receive4:                      ;if majority was 0
      BCLR    R2H,R6L         ;clear bit in O/P register
receive5:
      INC.B   R2H
      CMP.B   #08,R2H         ;are all 8 bits done
      BNE     receive6
receive7:
      MOV.B   R6L,@SCI2_RDR

      POP.W   R6
      POP.W   R2
      POP.W   R1
      POP.L   ER0
      RTE

rxerror:
      MOV.B   #H'FF,R6L
      MOV.L   #SCI2_SSR,ER0
      BSET    #RXERFLG,@ER0
      BRA     receive7
```

```
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;; Interrupt Handler for DMA on transmit completion

DMAIsr:
      BCLR.B          #3,@DMA_DTCR0A:8 ;disable DMA interrupts
      BCLR.B          #7,@DMA_DTCR0A:8 ;disable DMA transfers
      BCLR            #0,@ITU_TSTR:8   ;stop timer channel 0
      RTE


      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;; Interrupt vector definitions for the isrs
      ;;

      COMMON  INTVEC
      DS.B    88
      DC.W    DMAIsr
      DS.B    2
      DC.W    Receiver


      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      END
```

A Mechanism for Banking Data on the H8/300

The bank switching mechanism used by the H8/300 C compiler uses a look up table to determine the pseudo 32-bit address for each function as it is called. This results in both a data and a code overhead as each function call will in fact be surplanted by a call to a bank switcher which reads the relevant 32-bit value from the look up table store in the FLIST segment. The bank switching mechanism for user code function calls is selected at compile time by selecting one of the banked memory models. The size and position of the banks is determined at link time (however please not that each code module cannot exceed the size of a bank as it is treated as a non-divisible object at link time!).

To implement a secondary data switching mechanism the user will have to set up the data banks as a separate data segments (either constant data (ROM) or static data (RAM). To do this the #pragma directives should be used in the following way:

Constant Data Segments

```
#pragma memory = constseg (NAME1)
unsigned int = 450;
char str[] = "Please Enter Password";          /* string constant */
char *strptr = str'                            /* ptr to string constant
char array[] = {'0',0x56,'2','3','4','5'};     /* initialised array */
#pragma memory = default
```

Non-Initialised Data Segments

```
#pragma memory = dataseg(NAME2)
int a;
char *b;
struct tag {
        char a;
        struct tag *next;
} item;
#pragma memory = default
```

There are a few points to note from the above fragments of C code.

1. Firstly all ROM variables should of course be initialised to their default values (or they will take the value 0).
2. Please note that to initialise a pointer to a character string a two stage process has to be undergone - firstly define an array of characters which is initialised to the desired string. Then declare a pointer to a character which is initialised to the start address of the array (as above for str and strptr).This process is required to force the string constant into the user defined constant segment. If the code read 'char *str = "Please Enter Password"; then the compiler would recognise a pointer to a constant string and place the string into the constant string segment CSTR!
3. Static (volatile) data variables cannot be initialised at start-up due to the way that the cstart-up code has been written (i.e. there is no mechanism for recognising user segments and their locations). This could be coded into cstart-up but a ROM shadow area would have to be created to do it (and so you still wouldn't define the initialisation values in the above fragments).

Switching Banks

Now that the data itself has been defined the user has to note the position of each data bank so that the switch can be made. Figure 1 shows the scenario, the code banking scheme is being used with four code banks residing in the window H'D000 to H'FB00, the switch table has been located after the vector table (here at H'00C0). This in fact has all been done for you by the compiler.

In this example the code banks are physically separated by H'10000 (i.e. a full 64kbyte page) so that the code bank 1 will be between addresses H'1D000 and H'2FB80 etc.

This was achieved by the linker command:

```
-b(CODE)CODE=D000,2B00,10000
```

To implement the data bank switching scheme you must decide on the address decoding that will be required to select the relevant bank to be examined.

For example here the data banks are at logical addresses H'C000 TO H'D000, when banked they must not clash with the logical addresses of the code banks. Perhaps explicitly stated in the linker command file, e.g.

```
-Z(DATA)NAME1=C000-D000
-Z(DATA)NAME2=1C000-1D000
-Z(DATA)NAME3=2C000-2D000     (e.t.c).
```

The various external banks may then be switched by a couple of I/O lines forming the most significant two bits of the address (more bits will give you more banks). Naturally the user may decide to use the same port as the one used for code bank switching (but different pins!) to switch. In this case the following code may suffice when bits 4 and 5 or port 4 are used:

```
#define DBDDR    P4DDR
#define DBDR     P4DR
#define BANK(x) DBDR=( DBDR & 0xCF) | (x <<4))
```

The macro definition for BANK will simply compile down to four assembler lines for each invocation of BANK () and will allow the user to swiftly switch between data banks. For example

```
In C:           BANK (3);     ....will be....

In assembler:   MOV.B         @P4DR,R1L
                AND.B         #207,R1L
                OR.B          #48,R1L
                MOV.B         R1L,@P4DR
```

N.B. You must also define the relevant port to be an output port (DBDDR =) if it isn't the same one as used for banked code.

Judicious use of Multiple Data Banks

The most efficient use of the data bank scheme, where possible is to have each data bank large enough to handle the data which is local to, and shared between, a number of banked code modules. Thus if five functions all access the same data structure, then fitting all of the data for those functions into one bank will reduce the total number of switches required. This may mean that one code bank has one associated data bank, or more than one code bank will use a given data bank. The only time multiple switches may be required will be when given functions require access to more than one data bank at the same time. If carefully written the code will not cause any unsurvivable clashes between accesses - this means that intermediate 'overlapping' variables must be stored in non-banked memory. To do this the data must be 'shadowed'.

Shadowing Data across Multiple Data Banks

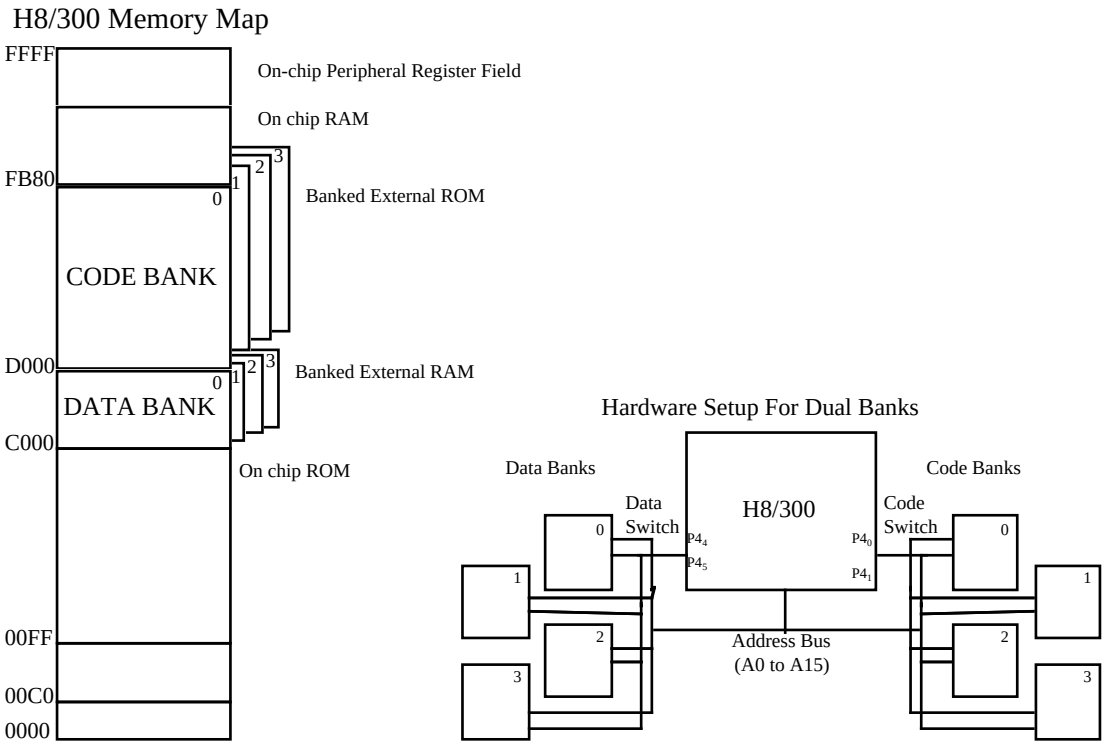
If your system requires multiple data banks to be accessed, at the same time, and this proves to be practicably impossible to avoid, then a shared 'shadow' area of memory must be maintained.

The simplest method will be to use non-banked memory to act as a temporary 'scratch pad' for intermediate sets of variables. This may be fine if the data structures are quite small, and not too much copying back and forth is required. However, when dealing with large systems this cache approach may result in a significant overhead with multiple reads, writebacks and then re-reads.

The functions 'malloc' and 'free' can be used to maintain a dynamically created scratch pad within the data heap on page 0.

Figure 1: H8/300 Dual Banked Memory Allocation

H8/300 Memory Map



Example Assembler Fuzzy Driver Routine,

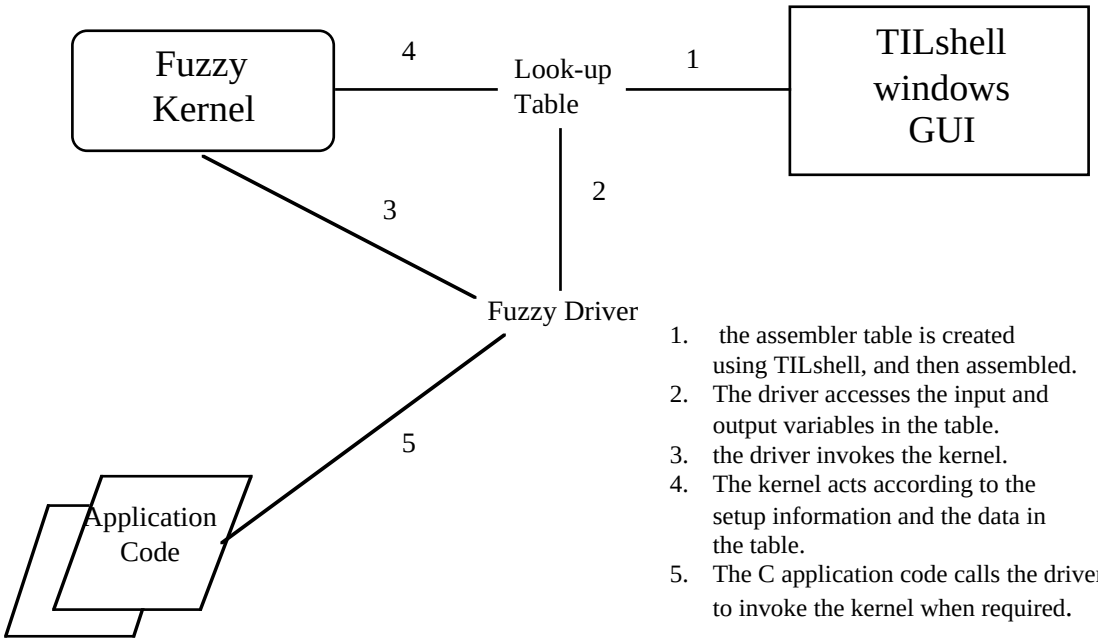
which can be called from C

The Fuzzy Logic development tools available from Hitachi use the Togai Infra Logic, TILshell, windows based development environment to define test and simulate fuzzy logic control systems. The system comes with a copy of the fuzzy logic kernel look-up table to determine how it should act.

A simplified example of a design and test process is as follows:

1. The user defines the fuzzy input and output variables, with associated membership sets and defines the fuzzy rule base (using TILshell).
2. The user then uses the TILshell to test the fuzzy system using a combination of:
   a. test input values
   b. 3-D control surface plot
   c. system simulation using a 'C like' test harness
3. The defined system is compiled (using TILshell) to produce an assembler source look-up table.
4. An assembler driver is written to interface to the application code's variables, point to the fuzzy look-up table, invoke the fuzzy kernel and finally return the calculated outputs.
5. The application code is written using the assembler driver to interface to the fuzzy kernel.

Figure 1:



1. the assembler table is created using TILshell, and then assembled.
2. The driver accesses the input and output variables in the table.
3. the driver invokes the kernel.
4. The kernel acts according to the setup information and the data in the table.
5. The C application code calls the driver to invoke the kernel when required.

Example Fuzzy Driver:

The example is from a simple red/green light traffic control system. For this example the fuzzy system is defined thus:

The Input variables:

TILshell name	Allocated symbol	Type
traffic_density	_traffic_density	unsigned char
traffic_speed	_traffic_speed	unsigned char

The Output Variables:

TILshell name	Allocated symbol	Type
green_light	_green_light	unsigned char
red_light	_red_light	unsigned char

Rule Base:

TILshell name	Allocated symbol	Type
traffic	_traffic	

Required C interface;

As the system reutrns more than one variable, it would seem sensible to pass pointers to structures to the fuzzy driver, one structure would contain the input variables, the second would be filled by the driver to return the output variables. Thus C type definitions could be:

```
typedef struct i_vals {
        unsigned char speed;
        unsigned char density;
}FUZZ_IN;

typedef struct o_vals {
        unsigned char green_time;
        unsigned char red_time;
}FUZZ_OUT;
```

And the function prototype for the driver might be:

```
void fuzzy_driver (FUZZ_IN *, FUZZ_OUT *);
```

The Assembler Driver

This must take the input variables from the structure and write them to the fuzzy input variable locations, then setup a pointer to the fuzzy look-up table, call the kernel and finally place the returned variables into the output structure. Here the assembler is for the H8/300 family. The code might then be:

```
;FUZZY kernel interface routine enabling data passed via a pointer to
;be passed to the kernel (param 1), and returning data from the kernel
;at the address pointed to by a second pointer (param2)

            NAME        fuzzy
            PUBLIC      fuzzy_driver

            EXTERN      _traffic_density        ;variables used by fuzzy table
            EXTERN      _traffic_speed
            EXTERN      _green_light
            EXTERN      _red_light
            EXTERN      Traffic
            EXTERN      _H300RTME               ;fuzzy kernel

            RSEG RCODE

fuzzy_driver:
            MOV.W       R0,@-R7
            MOV.W       R1,@-R7
            MOV.W       R2,@-R7
            MOV.W       R3,@-R7
            MOV.W       R4,@-R7
            MOV.W       R5,@-R7
            MOV.W       R6,@-R7
            MOV.B       @R1+,R0L                ;put vals from FUZZ_IN into rule
                        base
            MOV.B       R0L,@_traffic_speed
            MOV.B       @R1,R0H
            MOV.B       R0H,@_traffic_density
            MOV.W       #_Traffic:16,R0         ;point to rule base
            MOV.W       #_H300RTME:16,R1        ;point to kernel
            JSR         @R1                     ;invoke fuzzy kernel
            MOV.W       @(H'0010:16,R7),R0      ;pointer to OP var1 (green_time)
            MOV.W       R0,R1
            ADDS.W      #1,R1                   ;pointer to OP var2 (red_time)
            MOV.B       @_green_light,R3L
            MOV.B       @_red_light,R3H
            MOV.B       R3L,@R0                 ;saves values in structure
                        FUZZ_OUT
            MOV.B       R3H,@R1
            MOV.W       @R7+,R6                 ;restore old register values
            MOV.W       @R7+,R5
            MOV.W       @R7+,R4
            MOV.W       @R7+,R3
            MOV.W       @R7+,R2
```

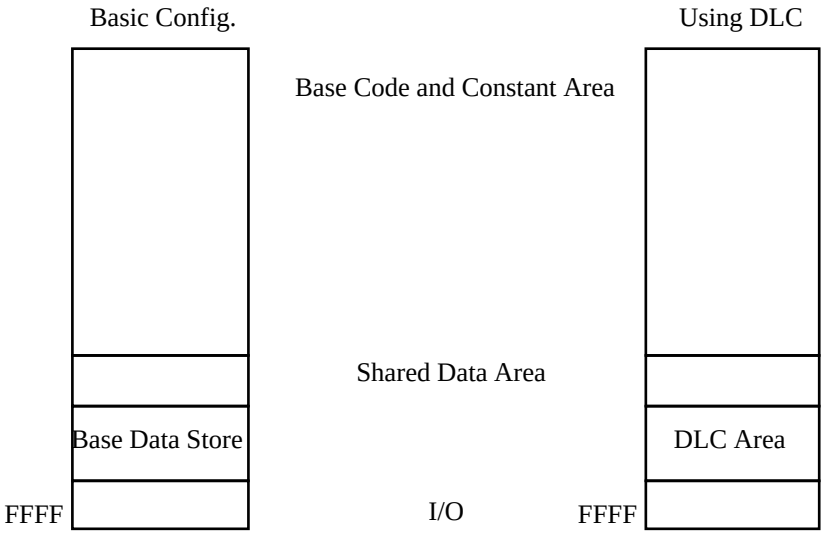
```
MOV.W     @R7+,R1
MOV.W     @R7+,R0
RTS
END
```

Writing Downloadable C code using the IAR C Compiler

In certain systems the user may require a different set of functions to be called according to the situation or mode of operation that the H8 microcontroller is in. The code will need to be downloaded into RAM and then executed from there.

In most systems RAM space is at a premium and so the user cannot leave an area of RAM free for the worst case scenario of downloading a large amount of code. One approach is to overwrite obsolete/irrelevant data with the code. In addition the interface between the base code and the dynamically loaded code DLC) needs to be standardised - i.e. if the base code calls dynamically loaded code (or vice-versa) the addresses need to have been fixed at link time. If there is to be some shared data area this needs to have accesses fixed at link time. However the linker (xlink) will not allow the user to have two segments located at the same address at link time, therefore some other action must be taken to ensure that both the base code and the DLL can access each others functions and shared data in a sensible manner. Figure 1 shows the possible interaction required by code used in such a system.

Figure 1: An example of a shared RAM system (First 64Kbytes)



Proposed Solution Using Splitter

Base Object Code Creation

In one case the object code created at link time has to have the base code, base data and shared data located at the correct addresses so that the default configuration can run correctly. The dynamically loadable code segments may be located in an unused area of the address map. This should then be done in linker stage 1.

Thus the interface between the base code and all data is correctly set-up. The object code created for the dynamically loadable code segments will however contain incorrect addresses and needs to be discarded.

Dynamically Loadable Code Creation

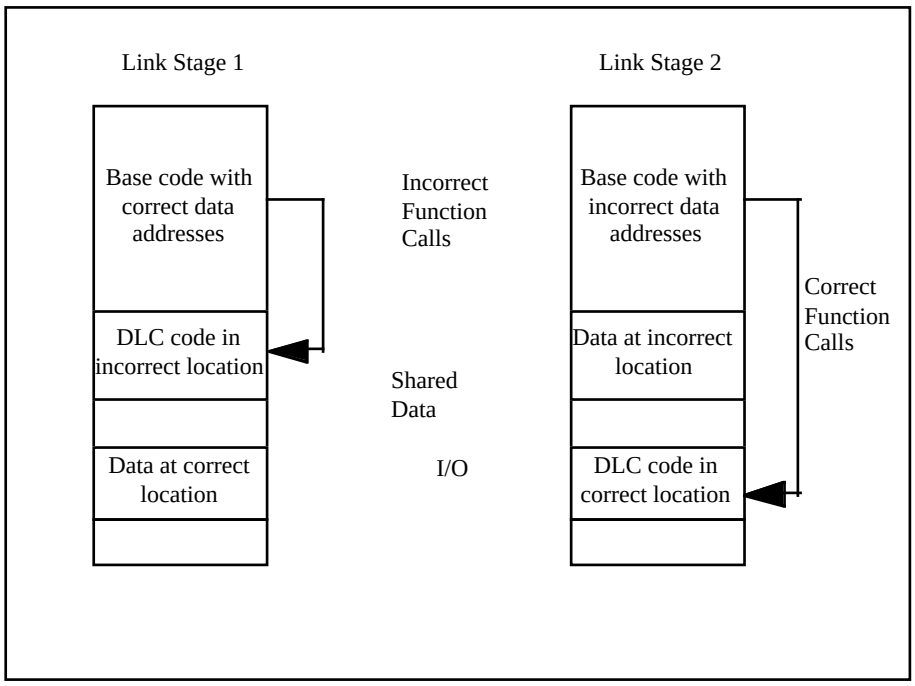
In each successive case for dynamically linked code the dynamically linked code must reside in the correct place, as should be shared data and the base code. However it is not necessary to have the discardable data in the correct place as this is not ever addressed by the dynamically loaded code. Thus another link should take place to create the correct object code for each successive dynamically loadable code segment. Thus linker stages 2 to N need to be performed for N sets of loadable code.

Thus the dynamically loadable code and its shared data will be located at the correct addresses. However the base code will be incorrect as it will contain accesses to base data, which will be at incorrect addresses and thus needs to be discarded.

The remaining problem

In any one of the above link stages the interface is correct between the base code and data, or the dynamically linked code and data. However, in linker stage 1 (creation of base code) the interface from the base code to the dynamically loadable code is incorrect (see Figure 2). Thus lookup function address tables will need to be created for calls to dynamically loadable code segments. As this information is different for each dynamically loadable segment it would be sensible to have a fixed address access table which was also downloaded with the code.

Figure 2: Function access Problems

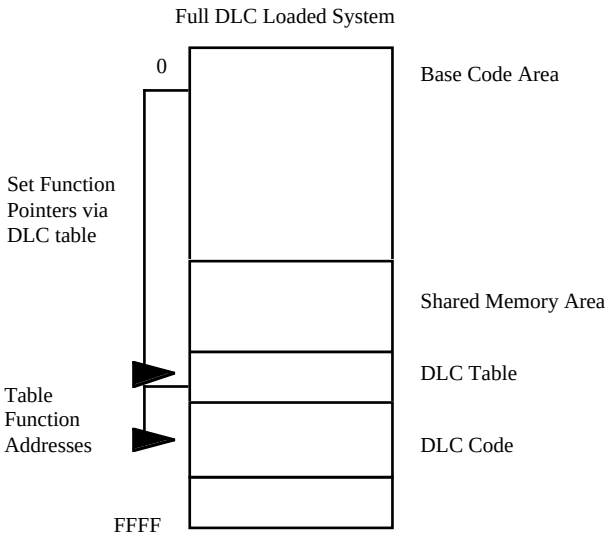


Setting up function Access Tables

For the base code the correct addresses of the dynamically loadable functions need to be accessible in a table format. This needs to be set at linker stages 2..N when the dynamically loadable code segments are correctly positioned.

Thus a series of function pointers would need to be created for accesses to each function (see Figure 3).

Figure 3: Function Pointer access of DLC from base code



Splitting Valid Object Code from Invalid Object Code

At each linker stage the linker will create code and perhaps data which will not be located at the correct addresses. To create a fully correct system the user will have to split the valid data from the invalid data. In addition the down loadable segments will probably exist in some external medium (EPROM for example) and thus will need to have the physical addresses of the Srecords moved so that they fit within the EPROM, whilst still maintaining the integrity of the code which will be downloaded to a different address. The Hitachi utility Splitter albws the user to perform these functions (see application note 005 for Splitter usage instructions).

Example of Dynamically Linked Code

The following example shows a process which may be undergone to load in a single DLC. The DLC will reside in an EPROM between the addresses 0000 and 1E7F.

Base System Specification

```
#include <stdio.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

/* Function prototypes */
void LoadDLC                (char *);
void SetupPointers          (char *);

#pragma memory  = dataseg (SHARED)
char *array;
#pragma memory = default

/* Table access pointers Table base address here is E000 */
#define TABLE(x)            (0XEE000 + (size of(void *) *x))

void ** pfpTestStr = (void **)TABLE(0);
void ** pfpFillStr =  (void **)TABLE(1);

/* Actual function pointers for DLC */

int (*pTestStr)             (char *);
int (*pFillStr              (char *, char *)

void LoadDLC (char *DLCName)

{
        /* your DLC loader code goes here */
}

void set-up pointers (char *DLCName)
{
```

```
{
        /*having loaded the table, set the function pointers to
        point to the functions in the DLC using the table entries in
TABLE segment*/
        if (!strcmp (DLCName, "DLC1"))
{
        pTestStr = *pfpTestStr;
        pFillStr = *pfpFillStr;
        return;
}
else if (!strcmp (DLCName, "DLC2"))
{
/* etc */
}
}
int main (void)
{
int res;
array = "Hello There";
LoadDLC ("DLC1")
SetupPointers ("DLC1");
res = pTestStr ("Hello There");
res = pFillStr (array, "Hello again");
return (0);
}
```

DLC Code

```
#define TRUE 1
#define FALSE 0

int TestStr                     (char *str);
int FillStr                     (char *, Char *);

/*
-------------------------------------SHARED DATA AREA-----------------------------------

*/

#pragma memory = dataseg (SHARED)
extern char *array;
#pragma memory = default

/*

--------------------SET-UP TABLE FOR FUNCTION ADDRESSES---------------------


#pragma memory = constseg (TABLE)
void *table [] = {
(void *) TestStr,
(void *) FillStr
};
#pragma memory = default
```

```
/*
----------DLC CODE----------

*/

#pragma codeseg (DLC)

int TestStr (char *str)
{
int i = 0
while (*str != '\0')
{
if  (*str ! = array [i++])
return (FALSE);
str++;
}
return (TRUE);
}
int FillStr (char *source, char *dest)
{
while (*source ! = '\0')
{
*dest = *source;
dest++;
source++;
}
return (TRUE);
}
```

System Build Batch File

```
@echo off
echo compiling base
icch8500 base -P-s-e-L-q
echo compiling dlc
icch8500 dlc -P-s-e-L-q

echo linking for base addresses + data
xlink dlc base -f base.xcl -o base -FPentica-bm -1 base.map -xsme
echo linking for dll
xlink dlc base -f dlc.xcl -o dlc -FPentica-bm -l dlc.map -xsme

echo splitting S-Records


splitter base
splitter dlc

del base.obj
del dlc.obj
rename main.obj base.obj
rename loader.obj dlc.obj
```

Linker Command Files

1,BASE.XCL

```
-!                                  BASE.XCL
This file defines the correct addresses for the data segments and the  code
-!
-! First define CPU -!
-cH8500

-! CORRECT BASE CODE SEGMENT AREAS -!
-Z(CODE)INTVEC, CODE, RCODE, CDATA, ZVECT, CONST,  CSTR,
CCSTR=0

-! TEMPORARY PLACE TO PUT DLL -!
-Z(CODE) TABLE, DLC=B000-CFFF

-! DLL AND BASE AREA SHARED VARIABLE STORAGE -!
-Z(DATA) SHARED=d000-dfff

-! CORRECT BASE VARIABLE AREA -!
-Z(DATA) DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP,
CSTACK+200=E000-FEFF
-Z(DATA) SHORTAD=FF00-FF7F

-DREG_BR=FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write
-e_medium_read=_formatted_read
-! Now load the 'C' library -!
c1h85s
```

2, DLL.XCL

```
-!                                  DLL.XCL
This file defines the correct addresses for the data segments and the  code.
-!
-! First define CPU -!
-cH8500

-! CORRECT BASE CODE SEGMENT AREAS -!
-Z(CODE) INTVEC, CODE, RCODE, CDATA, ZVECT, CONST, CSTR,
CCSTR=0

-! CORRECT PLACE TO PUT DLL -!
-Z(CODE) TABLE, DLC=E000-FEFF

-! DLL AND BASE AREA SHARED VARIABLE STORAGE -!
-Z(DATA) SHARED=d000-dfff

-! TEMPORARY BASE VARIABLE AREA -!
-Z(DATA) DATA, IDATA, UDATA, ECSTR, WCSTR, TEMP,
CSTACK+200=B000-CFFF
```

```
-Z(DATA) SHORTAD=FF00-FF7F

-DREG_BR=FF

-! See configuration section concerning printf/sprintf -!
-e_small_write=_formatted_write
-e_medium_read=_formatted_read

-! Now load the 'C' library -!
c1h85s
```

Splitter Command Files

1,BASE.FMT

```
BASE FORMAT FILE
-I "base.obj"

CODE SPLIT
-O "main.obj" 0x0000 0xAFFF
DATA SPLIT
-O "main.obj" 0xD000 0xFF7F
JUNK THE REST
-O "JUNK.OBJ" 0XB000 0XCFFF
```

2, DLC.FMT

```
DLL FORMAT FILE
-I "dlc.obj"

CODE SPLIT
-O "loader.obj" 0xE000 0xFEFF BASE=0x0000
JUNK THE REST
-O "JUNK.OBJ" 0x0000 0xDFFF
```

Created Output Files

A number of files have been created by the above process. In fact only the following files will be required for use by the end system:

dlc.obj - DLC code plus table (in EPROM format).
base.obj - base code, base data and shared data.

Final Comments:

The above example is intended as a guide to how a shared RAM system may be implemented using the IAR C compiler and Splitter. However the example shown here is a generalised one, and the user may find ways to improve the speed and storage usage of the system.

When using this document, keep the following in mind,

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during the operation of the user's unit according to this document.
4. Circuitry and other examples described herein are meant only to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples therein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorised for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant sales office when planning to use the products in MEDICAL APPLICATIONS.

APPLICATION NOTE **DATE : 14/11/95**

C Code Demonstration Program

This code illustrates some of the key features of the H8-300 C compiler V3.11/2 by IAR running on the H8-3834.

The program configures timer A as a RTC to generate a 1s interrupt. This 'Tick' interrupt increments variables seconds, minutes and hours, resulting in a clock function. The code also shows examples of general cpu initialisation such as interrupt handling and port configuration.

When not processing the processor is placed in either 'Sleep' or 'Watch' mode. The code for both modes is include but commented out for 'Watch', this can be re-selected if desired.

Internal CPU clock frequency = 5Mhz.

Compiler options:

1.0 -v0 300 cpu
2.0 -mL large memory model, normal intrinsics
3.0 -1id precision - 2byte ints, 4 byte doubles
4.0 -C nested comments
5.0 -K C++ // comments
6.0 -W(No) allows No size of trash on stack before clearing
7.0 -s(0-4) speed optimiser with level
8.0 -r debug information
9.0 -L -q list file with embedded assembler
10.0 -P PROMable code

```
---------------------------------------------------------------------------------------------------------------------------
*/


#include <ioh83834.h>   /* H8/3834 IO labels */
#include <inh83834.h>   /* and the interrupts */
#include <inh83.h>      /* include interrupt std fn's*/

#define EVER ;

/* ********* */
/* Functions */
/* ********* */

void init_port_4(void);         /* Function to initialise port 4*/
void init_timer_a(void);        /* Function to initialise timer A as a RTC*/
void watch(void);               /* Function to configure watch mode */
void toggle_port_4(void);       /* Configure port 4 to outputs */
void do_tasks(void);
tiny_func void Clock(void);     /* Clock defined as a tiny function */
interrupt [TIMER_A] void schedule(void); /* Interupt Service routine */



/***********************************/
/* Example's variables declarations*/
/***********************************/

bit set ;                                   /* bit variable which can only exist in tiny address range*/
```

```
bit clear ;

no_init unsigned char cal_vals[20];          /* example of non initialised variables*/

/* Data placement to user defined segments */
#pragma memory= dataseg(USER1)

unsigned char image[20];
unsigned short val1;
float fpval1;

#pragma memory=default                       /* back to the default type*/

near unsigned short far *ptr1;               /* pointing to far segment pointer in near area */

/* ********* */
/* Variables */
/* ********* */
#undef PCR4
sfr PCR4 = 0xFFE7;

#undef SYSCR2
sfr SYSCR2 = 0xFFF1;          /* Examples of register field definition using bits variables*/
bit NELSEL = SYSCR2.4;
bit DTON = SYSCR2.3;                  /* Direct transfere on flag */
bit MSON = SYSCR2.2;                  /* Medium speed on flag */
bit SA1  = SYSCR2.1;          /* SA1 and SA0 = Sub active mode clock select*/
bit SA0 =  SYSCR2.0;

unsigned char seconds=0, minutes=0, hours=0;
unsigned short ctick=0;

C_task void main(void)        /* C-task does not save registers which saves stack
                                 space. Only root functions should be declared
                                 like this. */

{
        init_port_4();          /* make port 4 an output */
        init_timer_a();         /* initialise timer a as RTC */
        IENR1 |= 0x80;          /* enable interrupt on timer A */
        IENR2 |= 0x80;          /* enable direct transfer */
        set_interrupt_mask(0);  /* Clear I bit, demonstartes the
                                     use of an intrinsic function */


        for (EVER)
        {
                /*  after the interrupt has woken up the
                    processor, the prog will execute the isr
                    and return here */

        watch();    /* bed time !*/

        }

}

void do_tasks(void)
{                       /* This is the kernal from which all of */
```

```
                        /* The tasks are called when the */
                        /* processor is in active mode */
unsigned char count1,count2;
        Clock();
        toggle_port_4(); /* delay to give indication to outside world via
                            LED */
        for(count1=0;count1<10;count1++)
         {
           for(count2=0;count2<10;count2++)
             {
             }
         }

}


void init_port_4(void)
{
        PCR4 = 0x7;  /*bottom 3 bit output*/
}

void init_timer_a(void)
{
                TMA = 0x1e;             /* reset tca and psw */
                TMA = 0x18;             /* bits 7,6,5 set o/p to 1Hz */
                                        /* bit 4 reserved */
                                        /* bits 3,2,1,0 set overflow to 1s */
}

void watch(void)
{           /* CONFIGURATION FOR WATCH MODE */

              /*  SYSCR1 &= 0x7f;     /* reset LSON bit (bit3)*/
                  SYSCR2 |= 0x04;     /* set MSON bit (bit 2)*/
                  SYSCR1 &= 0xb0;     /* reset bit 6 */
                  SYSCR1 |= 0xb0;     /*set bit 7 ssby bit, and bits 4,5*/
                  IENR1 |= 0x80;      /*enable interrupt on timer a*/
              */


              /* CONFIGURATION FOR SLEEP MODE */
                  SYSCR1 = 0x37;
                  NELSEL = 0;
                  DTON   = 0;         /*configure transition to sleep*/
                  MSON   = 1;         /*set active medium mode */
                  SA1    = 0;         /*subactive mode = clock/8 */
                  SA0    = 0;
                 /* SYSCR2 = 0xE4; */  /* Alternative method to above*/
                  IENR1  = 0x80;       /* enable interrupt on timer A*/

                sleep();                        /*Shut down*/
}
```

```
void toggle_port_4(void)
{
                PDR4 ^= 0x07;
}


#pragma function=tiny_func          /* all functions will use tiny function calling mechanisms */
void Clock(void)
/*      This function updates the seconds, minutes and hours. The standard
          function calling mechanism is overridden by 'tiny_func'. Will now
          use memory indirect addressing mode.
*/
{
if (ctick >= 1) { ctick = 0; seconds++;}
if (seconds>=60) { seconds=0; minutes++;}
if (minutes>=60) { minutes=0; hours++;}
}

#pragma function=default            /* back to default calling mechanism */

interrupt [TIMER_A] void schedule(void)
{
                IRR1 &= 0x7f;  /* reset the interrupt request */
                ctick++;
                do_tasks();         /* execute tasks */
                                    /* this lot must be <1 S  */
                                    /* The processor will return here */

}

/* Program End */
```

APPLICATION NOTE **DATE : 14/11/95**

H8-3042 Framework Program

This is a framework program for use on the H8-300H family of 16bit microcontrollers. It can be used as a basis for other programs and also to examine the various features of the 300H Peripherals. The program has been written for the H8-3042 but can be easily adapted to other members of the 300H family.

There are 7 basic functions for the program:-

1.0 Read 4 AD values, scale and output to serial port 0. The message is formatted via the 'sprintf' function and output to the serial port via a scheduler. The scheduler is called every 5mS from an interrupt generated by Ch0 of the ITU. A character is passed to the serial port every 5mS. Port B lines are toggled at multiples of 5mS to check correct operation.

2.0 Serial port 0 is set for 9600 baud, asynch, 8 data, 1 stop, no parity.

3.0 Configure ITU Ch1 to generate an output PWM of frequency 25KHz at 75% duty cycle. This can then be varied by adjusting the varying the voltage on analogue channel 0. This part is commented out to allow TPC operation.

4.0 Configure ITU Ch2 to measure the PWM frequency by using the input capture facilty.

5.0 Configure ITU Ch0 to generate a 5ms interrupt to act as a time base for a simple scheduler.

6.0 Configure ITU Ch3 to generate a 2mS interrupt that triggers DMAC channel 0A to load a byte of data to the TPC.

7.0 The DMAC is configured to short address I/O mode. Eight values are read from a look up table and transferred to the TPC. After these values have been transferred a cpu interrupt is generated and the cpu resets the DMA source address to point to the start of the look up table.

8.0 The TPC is configured so that outputs TP0 to TP7 are enabled, the TPC being set for non-overlap mode.

9.0 Configure the watchdog, for watchdog mode with a 65 mS timeout.

10.0 Configure the Refresh Controller for compare match interval mode, to give a match every 20mS. Upon a compare match, an interrupt is generated. The interrupt handler is used to reset the watchdog timer to prevent overflow and hence the MCU from reseting.

Specification Ends.
Software start date 11 January 95. - Version P1.0.

----------*/

```
/* Program starts here */

#pragma language = extended          // enable extensions
#include "inh83.h"                   // in-line functions
#include "ioh83042.h"                // port cast definitions
#include "inh83042.h"                // interrupt vector identifiers
#include "stdio.h"                                  // for sprintf

//---------------------------------------------------------------------------
/*      Function Prototypes      */

void Init_Interrupts(void);
void Init_ITU(void);
void Init_DMAC(void);
void Init_TPC(void);
void Init_AD(void);
void Init_Serial(void);
void Init_Ports(void);
void Init_WDog(void);
void Init_Refresh(void);

//---------------------------------------------------------------------------

/*      Definitions      */

#define EVER    ;;

//---------------------------------------------------------------------------

/*      Global Variables      */

unsigned char tx_msg_buf1[60];  // buffer for 'sprintf'
unsigned short  An0,An1,An2,An3,msg_ptr=0;
bit msg_sending_flag;
unsigned char step_out[8] = {0x95, 0x65, 0x59, 0x56, 0x95, 0x65, 0x59, 0x56};
unsigned short schedule_count=0;

unsigned long dmac_test=0;

//---------------------------------------------------------------------------

/*      'sfr' & 'sfrp' declarators here      */

sfr ITU0_TSR0 = 0xFFF67;
sfr PB_DR = 0xFFFD6;
sfr SCI0_SSR0 = 0xFFFB4;
sfr RTM_CSR = 0xFFFAD;

//---------------------------------------------------------------------------

/*      Code starts here      */

C_task main(void)       // C_task forces function not to save
{                       // registers on entry
unsigned char x;

/* Initialise Peripherals */
```

```
Init_Interrupts();
Init_Ports();
Init_DMAC();
Init_TPC();
Init_ITU();
Init_AD();
Init_Serial();
Init_Refresh();
Init_WDog();

for(EVER)
        {
        while (msg_sending_flag);                       // wait until last message sent
        for (x=0; x<=59; tx_msg_buf1[x++]=0);           // clear buffer
        sprintf(tx_msg_buf1, "Analogue Ch
            Values:An0=%d,An1=%d,An2=%d,An3=%d\n\r",An0,An1,An2,An3);
        msg_sending_flag = 1;                           // start transmission

       /* Read A-D buffers */

        An0 = ADDRA>>6;                                 // shift down to botton of short, '0's moved in
        An1 = ADDRB>>6;
        An2 = ADDRC>>6;
        An3 = ADDRD>>6;

        /* Scale in tenths of a volt */

        An0 = (An0*50)/1024;
        An1 = (An1*50)/1024;
        An2 = (An2*50)/1024;
        An3 = (An3*50)/1024;

        /* Use An0 to vary the PWM ratio on ITU Ch 0 */

        ITU_GRA1 = (An0*640)/50;

        }

}




void Init_Interrupts(void)
{
/* Configue SYSCR and CCR first */

SYSCR = 0x03;   // set UI bit in CCR as an interrupt mask bit

and_ccr(0x3F);    // enable interrupts, set UI bit to 0 - inline function

/* or alternatively this could be used to enable interrupts   */

set_interrupt_mask(0);   // enable all interrupts


/* Set priorities, Refresh controller and DMAC interrupt have priority. */
```

```
IPRA = 0x08;     // Priority for refresh controller
IPRB = 0x20;     // Priority for DMAC interrupts
IER = 0;         // all external ints disabled as default
ISCR = 0x3F;     // all external ints triggered by falling edge input

}



void Init_ITU(void)
{

/* Ch 0 - 5ms timebase interrupt for scheduler */

ITU_TCR0 = 0x21;        // clear comp' match GRA, internal clock/2
ITU_TIOR0 = 0;          // no output at compare match
ITU_TIER0 = 0x01;       // interrupt on compare match GRA
ITU_GRA0 = 0x9c3F;              // 40000 - to give 5ms compare match

/* Ch 1 - 25Khz PWM with 75% on duty cycle */

// Commented out for TPC operation

/*ITU_TCR1 = 0x40;      // clear comp' match GRB, internal clock
ITU_TIOR1 = 0;          // ignored PWM mode set in TMDR register
ITU_TIER1 = 0;          // no interrupts
ITU_GRA1 = 160;         // Go high
ITU_GRB1 = 640;         // Go low*/

/* Ch 2 - Input capture */

ITU_TCR2 = 0x40;        // clear on i/p cap on GRB
ITU_TIOR2 = 0x45;       // i/p capture GRB +ve edge, GRA -ve edge
ITU_TIER2 = 0;          // no interrupts

/* Ch 3 - 2mS interrupt to trigger DMA */

ITU_TCR3 = 0xC3;        // counter clear compare match GRB, clk/8
ITU_TIOR3 = 0;          // set GRA & GRB to output compares
ITU_TIER3 = 0x01;       /* interrupt on compare match A, triggers DMA & TPC,
                                interrupt flag, automatically cleared by DMAC */
ITU_GRA3 = 400;         // set non-overlap margin (10uS @ 16Mhz)
ITU_GRB3 = 4000;        // set period (2mS @ 16Mhz)



/* set registers common to all ITU channels */

ITU_TSNC = 0;           // channels operate independently - not synch'd
// Commented out for TPC operation
/*ITU_TMDR = 0x02;              // Ch 1 set for PWM mode*/
ITU_TFCR = 0;           // Ch 4 not used
ITU_TOER = 0;           // Ch 4 not used, pins used for normal i/o
ITU_TOCR = 0xFF;                // not used as default
ITU_TSTR = 0x0D;        // start channels 0, 2 and 3 - 1 stopped for TPC

}
```

```
void Init_DMAC(void)
{
/* configure DMAC for short addressing mode. Channel 0A set
   for I/O mode with a cpu interrupt generated after 8 transfers */

unsigned char temp;

DMAC_MAR0A = (unsigned long)step_out;  // address of look up table
DMAC_IOAR0A = 0xA5;            // NDRA same output trigger for TPC groups 0 & 1
DMAC_ETCR0AH = 0;              // high byte
DMAC_ETCR0AL = 8;              // Low byte, transfer count of 8

temp = DMAC_DTCR0A;            /* dummy read to allow DTE bit to be set to 1 */

DMAC_DTCR0A = 0x8B;            /* enable transfers, byte size, increment MAR,
                                  I/O mode, enable CPU interrupt at end of transfer
                                  count, trigger compare match A ITU 3 - DTCRB only
                                  used for full address mode */
}

void Init_TPC(void)
{
/* configure for non overlap with byte output on TPC groups 0 & 1. Groups
   are triggered by same timer, ITU Ch3. */

TPC_TPCR = 0xFF;               // groups 0 & 1 triggered by ITU Ch3 compare match
TPC_TPMR = 0x03;               // non overlap mode for groups 0 & 1
TPC_NDERA = 0xFF;      // enable transfer of NDERA data to PADR
}




void Init_AD(void)
{
/* Configure for scan mode on channels 0 to 3  */

ADCSR = 0x3B;   // Scan mode, ch's 0 to 3, start, no interrupt

}




void Init_Serial(void)
{
/* configure serial port for 9600baud, 1 stop, 8 data, no parity,
   interrupt on receive */

// Serial Port 1 not used

SCI0_SMR = 0;              // multiprocessor bit not used
SCI0_SCR = 0x70;           // interrupt on receive , start tx and rx
SCI0_BRR = 51;             // set for 9600 baud
}
```

```
void Init_Ports(void)
{
/* Ports set for following:-

      PortB = Output.

*/

PADDR = 0xFF;          // output
PBDDR = 0xFF;          // output

}


void interrupt [DMAC_DEND0A] DMAC0A_Tfr_End(void)
{
/* interrupt generated by DMAC0A at end of transfer count */
unsigned char temp;
DMAC_ETCR0AH = 0;                    // re-write transfer count - high byte
DMAC_ETCR0AL = 8;                    // Low byte, transfer count of 8
DMAC_MAR0A = (unsigned long)step_out;  // reset MAR to point to start of table
temp = DMAC_DTCR0A;                  // dummy read
DMAC_DTCR0A |= 0x80;                 // re-enable DTC transfers
}




void interrupt [ITU_IMIA0] ccr_mask[0x02FF] Scheduler(void)
/*
      Interrupt Service routine for compare match on
      ITU Ch 0. Note ccr_mask shown for example only
      as mask operation and value is set to have no
      effect on ccr.

*/
{
static unsigned char t50ms=9, t100ms=1, t250ms=4, t500ms=1, index=0;
ITU0_TSR0.0 = 0;          // reset GRA compare match interrupt flag
                          // read modify write required ie a bit operation

/* This interrupt is called every 5ms */


/* Toggle port pin for square wave output at 100Hz (10mS) */

PB_DR.5 = (!PB_DR.5) ? 1 : 0;     /* toggle pin */

schedule_count++;

if (schedule_count >= 5000) schedule_count=0;

/* transmit message from buffer */

if (msg_sending_flag)
```

```
        {
        while (!SCI0_SSR0.7);         /* wait until tx buffer free */
        SCI0_TDR = tx_msg_buf1[msg_ptr++];
        SCI0_SSR0.7 = 0;              /* reset TDRE flag */
        }

if (!tx_msg_buf1[msg_ptr])
        {
        msg_ptr=0;
        msg_sending_flag = 0;   // message sent
        }

/* user functions to be called every 5ms - note combined run time
   of these functions must be < 5ms */


/*      User_5ms_Func1();
        User_5ms_Func2();
        User_5ms_Func3(); */


if (!t50ms)
        {
        /* user functions to be called every 50ms - note combined run time
        of these functions must be < 50ms */

        /* 50ms tick */

/*      User_50ms_Func1();
        User_50ms_Func2();
        User_50ms_Func3(); */

        /* reload 50ms tick */
        t50ms = 9;

        /* user functions to be called every 100ms - note combined run time
        of these functions must be < 100ms */




        /* 100ms tick */

        if (!t100ms)
                {
                /* Toggle port pin for square wave output at 5Hz (200mS) */

                PB_DR.6 = (!PB_DR.6) ? 1 : 0;     /* toggle pin */

/*              User_100ms_Func1();
                User_100ms_Func2();
                User_100ms_Func3();*/

                /* reload 100ms tick */
                t100ms = 1;

                /* user functions to be called every 500ms - note combined
                run time of these functions must be < 500ms */
```

```
            /* 500ms tick */

            if (!t500ms)
                    {
/*                  User_500ms_Func1();
                    User_500ms_Func2();
                    User_500ms_Func3();*/

                    /* reload 500ms tick */
                    t500ms = 4;
                    }
            else t500ms--;

            }
      else t100ms--;

      /* user functions to be called every 250ms - note combined run time
       of these functions must be < 250ms */

       /* 250ms tick */



      if (!t250ms)
            {

/*          User_250ms_Func1();
            User_250ms_Func2();
            User_250ms_Func3();*/

            /* reload 250ms tick */
            t250ms = 4;

            }
      else t250ms--;
      }
else t50ms--;

}



void Init_Refresh(void)
{
RFSHCR = 0;             // set for interval timer mode
RTCOR = 78;             // set for a compare match every 20mS
RTMCSR = 0x7F;          // interrupt, clk/4096
}




void interrupt [CMI] Refresh_Int(void)
/* Interrupt handler for refresh controller when in interval mode */
{
RTM_CSR.7 = 0;                  // reset compare match flag

PB_DR.7 = (!PB_DR.7) ? 1 : 0;           /* toggle pin */
```

```
/* reset watchdog timer to prevent overflow */

WDT_TCNT = 0x5A00;              // '5A' password - reset timer to zero

}



void Init_WDog(void)
{
/* configure for watchdog mode - will reset MCU at overflow */

WDT_TCSR = 0xA57F;              // 'A5' password, watchdog mode, clk/4096, started
                                //  will overflow every 65mS if not reset
}
```

APPLICATION NOTE **DATE : 8/5/96**

H8/300H Software UART implementation using 2 Timer and 2 DMA channels

Most variants of the H8/300H family of micro-controllers support two hardware UARTS as standard. However in a lot of detailed embedded applications today there is an increasing requirement for more than two. Detailed here is a method where a further full duplex UART channel can be implemented using two on board timers and two of the DMA channels. For the transmission side there is some overhead required to set up the transmission table and pattern and on the receive side there are two CPU interrupts, one at reception of the first start pulse edge (approximately 8 µs) and one when the packet has been received (approximately 64 µs).

The outer layer of the application code has been written in C but the bit bashing routines are shown here in assembler for speed purposes, but there is no reason why these cannot be converted to Ansi C.

Asynchronous Communications:

An asynchronous data packet will be of one of the following formats:

<start bit>	<data>		<stop bit>	
<start bit>	<data>		<stop bit>	<stop bit>
<start bit>	<data>	<parity bit>	<stop bit>	
<start bit>	<data>	<parity bit>	<stop bit>	<stop bit>

The data content may be, typically, between five and eight bits. Parity may be even, odd or disabled. There may be one or two stop bits. In standard use the user selects the format required for the communications protocol in use, and the UART remains set at a standard format. This example assumes that the user has only one protocol required, and that this is known at compile time, thus the code may be written specifically for the format required, and may therefore have any generalisations removed to increase execution speed and reduce code overhead.

The selected packet format is: 9600 bps, 8 data bits, no parity, 1 stop bit.

To implement a Uart two separate operations must be catered for:

- The UART must be able to create data packets on an I/O line which conform to the required bit rate and packet format.
- The UART must be able to detect transitions on a separate I/O line and disassemble the transitions into a valid data value.

Both the above operations require the UART simulation software to be able to time I/O control at a rate which is proportional to the bit rate of the asynchronous interface. In addition data must be transferred to/from the I/O lines in synchronisation with the time steps. It is feasible to implement such timing and data transfer operations in a number of ways:

- CPU performs a timing loop and transfers data.
- Timers generate time-out interrupts, the CPU then transfers data on interrupts occurring.
- Timers generate time-out interrupts, data is transferred by DMA on time-out events.

The above three options represent a decreasing requirement on the CPU, with the timer/DMA option requiring CPU intervention only to set up and start transmission, or to disassemble the received data. It is this option which has been selected for this example, as it represents the best use of the H8/300H peripherals to minimise the CPU overhead in implementing a UART.

Transmitter Implementation:

The algorithm used to implement the transmission par of the software UART reduces the number of DMA transfers to the absolute minimum required, with the slight CPU overhead of creating a timer set up table before invoking the timer/DMA transmission system. The algorithm uses the following basic principle:

- If a bit pattern is exclusive-ORed with itself divided by 2, the resulting pattern represents the 0 to 1, or 1 to 0 transitions in the original bit pattern.

For example:

Original Pattern:	01110100
Pattern / 2	00111010
XOR	01001110

i.e. a bit transition occurs at (reading from left to right) bits 1,4,5 and 6.

This transition data may then be used to determine the exact time at which the transmit line must be toggled, which will not usually be on every bit of the packet. Thus worst case, the transition method of forming data packets will tend to that of transferring bit data at the bit rate, while usually the number of transfers will be significantly reduced ((four transitions in the above example).

The set up procedure for transmission is described below:

- Calculate the transition pattern.
- Form a table of transition times based upon the bit rate. The table should contain multiples of the bit rate count for the frequency of the timer channel used.
- Set the timer channel to compare-match on the first transition time-out and to toggle an I/O line on each compare-match event. The timer channel should also trigger a DMA transfer on each compare-match event.
- Set a DMA channel to transfer successive data from the transition data channel, to the compare-match register on each successive compare-match event, until all transitions have occurred.

Thus once the transition table has been created and the timer and DMA initialised, then the timer will control the I/O line using a compare-match event, and the DMA will reload the compare-match register automatically with the next compare-match value. When the required number of transitions have occurred the DMA can be used generate a CPU interrupt, which may then stop any further timer and DMA operations. This represents a completely self-contained system which will only interrupt the CPU on completion of transmission

Receiver Implementation

The receiver needs to detect the start of a data packet (denoted by the low start bit), it then must sample the receive data line at a minimum of the bit rate to determine the incoming data bits. It is recommended that the received data line is sampled at a frequency greater than the bit rate, so that more than one sample is available for each data bit. This will then enable the receiver to remove any spurious readings from the incoming bit stream, by taking the majority of the readings for each bit.

The implementation described below uses a sampling rate which is four times the bit rate. The start bit is detected by a timer channel using the input capture function. This may then be used to interrupt the CPU, which may then start the sampling process. A timer channel may then be used to set a sampling interrupt which may be used to trigger a DMA transfer from the receiver I/O line to a receive data buffer.

Having finished the sampling process the CPU may then examine the sampled data to determine the received value. This is accomplished by taking the majority level of each four samples as the received bit level. Once the data value has been determined any parity checking may be performed, if required.

To summarise the operations

Set-up - Timer & DMA

- Set the sample rate in to a compare-match register
- Enable input-capture interrupts for the timer channel
- Start the timer channel
- Set the source and destination DMA transfer addresses
- Set the DMA transfer count
- Set the DMA operation mode

On Input-capture Interrupt:

- Stop timer channel
- Disable input capture interrupt & clear flag
- Enable compare match-interrupts
- Enable DMA transfers on compare-match interrupt and enable DMA
- Start timer channel

On DMA completion interrupt:

- Stop the timer channel & clear flags
- Disable DMA transfers
- Disassemble receive buffer using majority samples to determine data bits.
- Set receiver flag and store data.

The above process may then be used to receive multiple data bytes at the desired bit rate.

Please note that this method uses two DMA channels to perform the data transfer, as normal address mode is required. If these are not available, the sample transfer could be implemented as a compare-match CPU interrupt. However, this would have more impact on CPU bandwidth.

UART design notes:

The UART uses the following hardware:

Transmitter - One channel of the ITU
One channel of the DMA
Receiver - One channel of the ITU
Two channels of the DMA

However if the UART is not required to transmit at the same time as data may be received, then the total requirement for the UART will be one channel of the ITU and two channel of the DMA.

In this example the UART has been set up assuming that reception and transmission will not occur simultaneously.

UART Benchmark Timings:

The code was timed on an E7000PC emulator for the H8/3042 device in advanced single chip mode (mode 6) operating at 16Mhz system clock. The functions/interrupt handlers are listed below with execution times. In addition timings were taken from the invocation of the transmit function, until the last data bit was transferred, and the handler returned. This will give an indication of the toal transmit time which the CPU requires - please note that the CPU will not be performing this code exclusively during this period.

Function:	Time Taken (µsec):
Receiver	64 µsec
Transmit	21 to 25 µsec
Initialise UART	21 µ sec
ITU Capture Interrupt	8 µsec
DMA Isr	2 µsec

UART.C - Software UART implementation for H8/3042.

```
#include “ioh83042.h”      //      I/O register definitions
#include “inh83042.h”      //      Interrupt vector addresses
#include <inh83.h>                  //      In-line functions

#pragma language=extended

//System Constants

#define GRA0            0x6A
#define DRA             0xD3

//Union for C access to DMAC MAR registers

union u_tag {
        unsigned long full;
        struct w_tag {
                unsigned short hw;
                unsigned short lw;
        } word;
        struct b_tag {
                unsigned char rb;
                unsigned char eb;
                unsigned char hb;
                unsigned char lb;
        } byte
} MAR

//Global data and transmit/receive buffers

unsigned short dma_buff[20]
unsigned short period;
unsigned short dma_count
unsigned char rx_buff[40];

//Assembler based fast comms routine

void Transmit (unsigned char);

//Globals used for Flags

unsigned char SCI2_SMR;
unsigned char SCI2_SSR;
unsigned char SCI2_SCR;
unsigned char SCI2_RDR;
```

Function to set the ITU and DMAC channels to their initial settings, the ITU is run for a short time to allow a compare match to occur so that the output line may be set to a logic high.

The transmitter uses ITU channel 0 (CMA) and DMA channel OA (in I/O mode).

The receiver uses ITU channel 1 (ICA then OCA) and DMA channels 1A and 1B (in block transfer full address mode)

void Initialise UART (void)

```
{

period = 417;                   //sys/4 @16MHz -> 9600 bps

//set up for Transmit, initialise the timer output to high...

ITU_TCNT0      = 0x0000;
ITU_TCR0       = 0xA2;          //CCLRA, sys/4
ITU_TIOR       = 0x8A;          //output 1 on CMA
ITU_GRA0       = 10;            //compare match value
ITU_TSRO &= 0xF8;               //clear flags
ITU_TSTR |     = 0x01;          //start the timer channel
while ((ITU_TSRO & 0x01) ! = 0x01); //wait on compare match
ITU_TSTR &    OxFE;             //stop channel 0 with output = 1
ITU-TSRO &    = 0xF8;           //clear flags

//now set up the timer and DMA channels to perform the transmission

ITU_TIOR0      = 0x8B;          //toggle O/P on CMA
ITU_TIER0      = 0xF9;          //enable interrupts on CMA
DMAC_IOAR0A = GRAO              //enable interrupts on CMA
DMACIOAR0A   = GRA0;            //set I/O destination to GRA0 register
DMAC_DTCR0A= 0x40; //word, inc MAR, I/O CMA

//set up ITU for receive

ITU_TCNT1     =0x0000; //reset counter
ITU_TCR1      =0xA0;            //CCLRA, sys
ITU_TIOR1     = 0x8E;           //ICA falling edge
ITU_TSR1 &    = 0xF8;           //clear flags
ITU_ TIER1    = 0x01;           //enable interrupt on ICA
//set up DMA for receive
DMAC_DTCR1A           = 0x07;  //byte, fixed source, block mode
DMAC_DTCR1B           = 0x19;  //inc dest, source is block, trigger on CMA 1
DMAC_ETCR1AH          = 0x01;  //1 byte per block
DMAC_ETCR1AL          = 0x01;
DMAC_ETCR1BH          = 0x00;  //36 transfers to complete
DMAC_ ETCR1BL         = 0x24;
DMAC_MAR1AE           = 0x00;  //port A is the source address
DMAC_MAR1AH           = 0xFF;
DMAC_MAR1AL           = DRA;
MAR.word.lw     = (unsigned short) &rx_buff; //set destination
DMAC_MAR1BE           = MAR.byte.eb;
DMAC_MAR1BH           = MAR.byte.hb;
DMAC_MAR1BL           = MAR.byte.lb;

}
```

Interrupt service routine to fire off the sampling process for a packet of data, given that the low edge of a start bit has been detected by the input capture logic of the timer channel.

```
interrupt [ITU_IMIA1] void ITUCaptureInterrupt (void)
```

```
{

ITU_TSTR &    = 0xFD;          //stop channel 1
ITU_TCNT1     = 0x0000;                //reset counter
ITU_TIOR1     = 0x88;                  //CMA, no output
ITU_GRA1      = 417;                   //sys @ 16Mhz -> 4 * 9600 samples/sec
DMAC_DTCR1A | = 0x80;                  //enable DMA
DMAC_DTCR1B | = 0x80;                  //DMA master enable
DMAC_DTCR1A | = 0x08                   //enable DMA interrupts
ITU_TSR1 &= 0xF8;                      //Clear flags
ITU-TSTR |= 0x02;                      //start channel 1

}
```

Main function to test that the transmit and receive operations work - here the transmit line may be fed back to the receive line to allow simultaneous transmission and receipt.

```
int main (void)
{
int i;
set_interrupt_mask (0);
InitialiseUART();
while (1)
{
Transmit (0x74);               //transmit the byte 0x74
for (i=0;i<4000;i++);                  //delay for a bit between packets
}
return (0);
}
```

RXTX - assembler routines for H8/3042 software UART

```
NAME rxtx

CASEON

PUBLIC      Transmit
PUBLIC      Receiver
PUBLIC      DMAisr
EXTERN      period
EXTERN      dma_buff
EXTERN      dma_count
EXTERN      rx_buff
EXTERN      SCI2_RDR
EXTERN      SCI2_SSR
```

Set the CPU control register constants

```
ITU_TCNT0       equ     H'FF68
ITU_GRA0        equ     H'6A
ITU_TCR0        equ     H'64
ITU_TIOR0       equ     H'65
ITU_TSRO        equ     H'67
ITU_TIERO       equ     H'66
ITU_TSTR        equ     H'60
DMA_MAR0AM      equ     H'20
DMA_MAR0AS equ      H'21
DMA_MAR0AB equ      H'22
DMA_MAR0AY equ      H'23
DMA_IOAR0A  equ     H'26
DMA_ETCROAH     equ     H'24
DMA_ETCROAL     equ     H'25
DMA_DTCROA equ      H'27
DMA_DTCR1A equ      H'37
DMA_DTCR1B equ      H'3F
```

Bit constants

```
RXERFLG         equ     6
```

```
RSEG CODE
```

Transmit function to form transition table and then set the ITU and DMA off to transmit the data packet.

Parameters: char to transmit will be in R6L

Register usage:

Transmit:

set up a work environment

```
PUSH.L    ER6
PUSH.L    ER5
PUSH.L    ER4
PUSH.W    R3
PUSH.W    R0



MOV.B     R6L,R6H       ;get a copy of the input char
SHLL.B    R6H           ;right shift
XOR.B     R6L,R6H       ;form transition byte

initialise data
MOV.L     #dma_buff:32,ER4;buffer pointer
SUB.L     ER5,ER5;zero out ER5
MOV.B     #H'08,R3L     ;set loop count to 8
MOV.W     @period:24,E6 ;load bit period to E6
MOV.W     E6,R5         ;initial setting for dma timeout

form timout table

label02;

SHLR.B    R6H           ;is there a transition?
BCC       label01
MOV.W     R5,@ER4       ;if yes then buffer the time period
INC.L     #2,ER4
SUB.W     R5,R5         ;clear period count
INC.W     #1,E5         ;inc count

label01:

ADD.W     E6,R5         ;add next bit period to count
DEC.B     R3L
BNE       label02       ;do eight times...

MOV.W     R5,@ER4       ;buffer the time period
INC.W     #1,E5         ;inc count

label03:
```

Finally load the stop bit timeout into the dma table here there is one stop bit in a packet.

```
INC.L     #2,ER4        ;point to next entry in the transition table
MOV.W     E6,@ER4       ;set stop period in transition table
INC.W     #1,E5         ;increment dma transfer count
```

Now send it out using DMA & ITU

```
SUB.W     R0.R0                    ;load ITU_TCNT with 0
MOV.W     R0,@ITU_TCNT0:16
MOV.L     #dma_buff,ER4
MOV.W     #20,R0                   ;load GRA with start offset
MOV.W     R0,@H'FF6A:16
BCLR      #0,@ITU_TSRO:8           ;clear IMIA flag
MOV.B     R4H,@DMA_MAR0AB:8
MOV.B     R4L,@DMA_MAR0AY:8
MOV.B     E4,R4
MOV.B     R4L,@DMA_MAR0AS:8
MOV.W     E5,R5                    ;ECTR0A data =
MOV.B     R5H,@DMA_ETCR0AH:8       ;no of transfers
MOV.B     R5L,@DMA_ETCR0AL:8
```

finally start the timer channel 0 to tx the data

```
BSET      #0,@ITU_TSTR:8
BSET      #7,@DMA_DTCR0A:8    ;enb dmas
BSET      #3,@DMA_DTCR0A:8    ;end dma ints


POP.W     R0
POP.W     R3
POP.L     ER4
POP.L     ER5
POP.L     ER6

RTS
```

need to place received char is placed in SCI2_RDR

Receiver:

Assume that the data is in rx_buff
4 samples per bit, 9 bits...

```
PUSH.L    ER0
PUSH.W    R1
PUSH.W    R2
PUSH.W    R6

MOV.B     #H'FD,R0L
MOV.B     R0L,@ITU_TSTR:8          ;stop the timer channel
BCLR      #3,@DMA_DTCR1A:8    ;disable DMA interrupts
BCLR      #7,@DMA_DTCR1A:8    ;disable DMA transfers
BCLR      #7,@DMA_DTCR1B:8    ;ditto
```

Now disassemble the received data

```
MOV.L     rx_buff,ER0
ADD.L     #4,ER0          ;zip past start bit
SUB.B     R2H,R2H         ;set bit count to 0
SUB.B     R6L,R6L ;initialise return value

receive6:

SUB.W     R1,R1           ;initialise zero & 1 counts
MOV.B     #04,R2L  ;sample count

receive0:

BTST      #4,@ER0
BEQ       receive1
INC.B     R1H             ;if its's a 1 inc 1s count
BRA       receive2

receive1:

INC.B     R1L             ;if it's a 0 inc 0s count

receive2:

INC.L     #1,ER0          ;point to next item
DEC.B     R2L             ;dec sample counter
BNE       receive0
CMP.B     #3,R1H
BGE       receive3
CMP.B     #3,R1L
BGE       receive4
BRA       rxerror

receive3:                 ;if majority was 1

BSET      R2H,R6L         ;set bit in O/P data register
BRA       receive5

receive4:                 ;if majority was 0

BCLR      R2H,R6L         ;clear bit in O/P register

receive5:

INC.B     R2H
CMP.B     #08,R2H ;are all 8 bits done
BNE       receive6
```

```
receive7:

MOV.B       R6l,@SCI2_RDR

POP.W       R6
POP.W       R2
POP.W       R1
POP.L       ER0
RTE

rxerror:

MOV.B       #H'FF,R6L
MOV.L       #SCI2_SSR,ERO
BSET        #RXERFLG,@ER0
BRA         receive7
```

Interrupt Handler for DMA on transmit completion

```
DMAIsr:

BCLR.B      #3,@DMA_DTCR0A:8    ;disable DMA interrupts
BCLR.B      #7,@DMA_DTCR0A:8    ;disable DMA transfers
BCLR        #0,@ITU_TSTR:8              ;sto timer channel 0
RTE
```

Interrupt vector definitions for the isrs

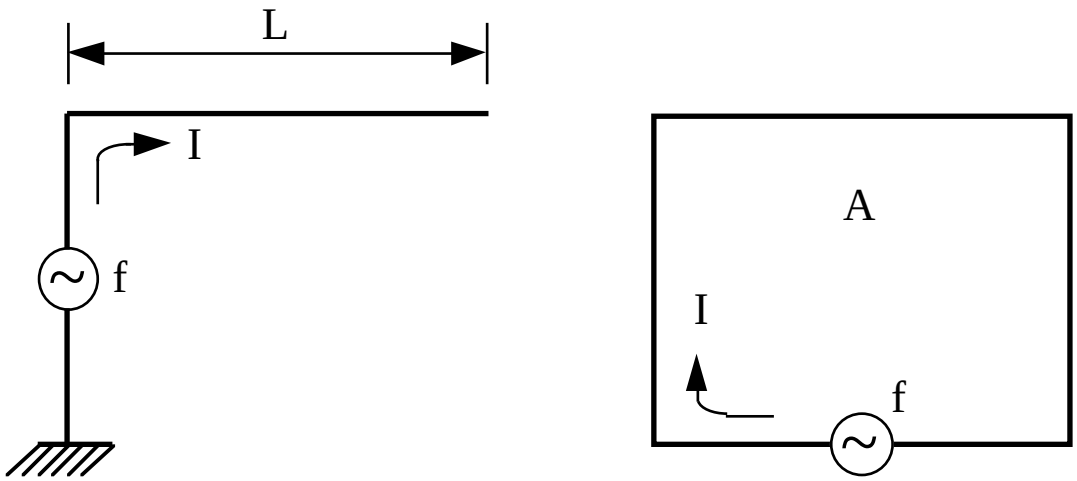
```
COMMON      INTVEC
DS.B        88
DC.W        DMAIsr
DS.B        2
DC.W        receiver


END
```

APPLICATION NOTE **DATE : 28/07/95**

Flat Panel Displays - EMI Considerations

The following application note describes EMI considerations on Thin Film Transistor(TFT) and Super Twist Neumatic (STN) displays. The considerations cover the equivalent circuit of the display, what harmonics of operating noise will affect the display and also suggest methods by which electro-magnetic interference can be reduced.

1. Antena Model and Estimation



Common Mode
$E \alpha fLI$

Differential Mode
$E \alpha f^2 AI$

	Number of Colour	Dot Clock (MHz)	Number of Data Line	EMI Strength (Differential Mode)
VGA TFT	512 colour	25 MHz	9 bit	0 dB (base)
	262K colour	25 MHz	18 bit	+ 6 dB
SVGA TFT	262K colour	38 MHz	18 bit	+ 13 dB
XGA TFT	16M colour	33 MHz	48 bit	+ 19 dB

Estimation of EMI strength based on the differential model

2. Harmonics

$= a_0 + a_1 \sin wt + a_2 \sin 2wt + a_3 \sin 3wt + a_4 \sin 4wt + ...$
$+ b_1 \cos wt + b_2 \cos 2wt + b_3 \cos 3wt + b_4 \cos 4wt + ...$

Test Result Example

Freq. (MHz)	Peak	Q. Peak	Limit	Delta	
48.38	25.4		40.0	14.60 dB	Horizontal
193.31	32.8		43.6	10.80 dB	
48.38	33.76		40.0	6.24 dB	
51.36	39.93	33.93	40.0	6.07 dB	
87.05	28.32		40.0	11.68 dB	
120.02	33.1		43.6	10.50 dB	Vertical
193.28	37.2		43.6	6.40 dB	
200.08	38.6	39.0	43.6	4.60 dB	
233.31	35.48		46.0	10.52 dB	
270.67	38.9	40.23	46.0	5.77 dB	

Dot clock frequency = 38.66 MHz

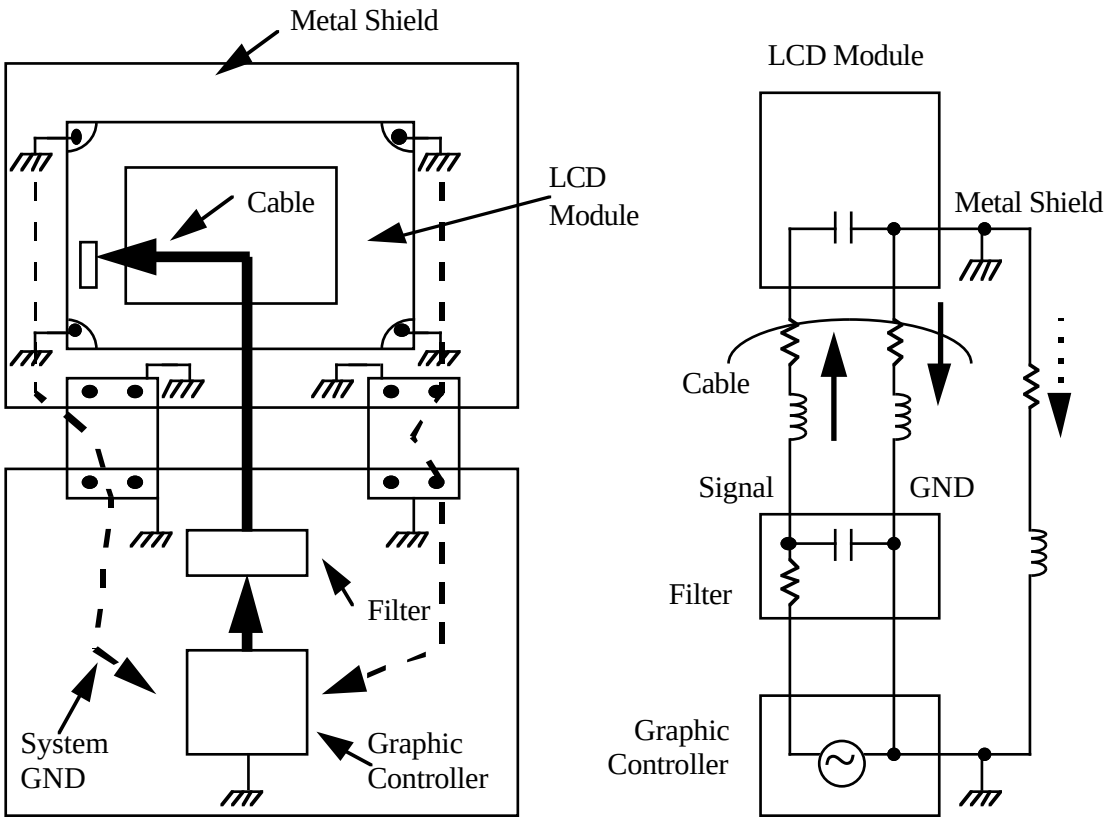
Data frequency = 9.665 MHz

Generated by graphic controller

Analysis

48.38 / 9.665 = 5.00	-	5th harmonic of data
51.36 / 9.665 = 5.31	-	not related to video frequency
87.05 / 9.665 = 9.00	-	9th harmonic of data
120.02 / 9.665 = 12.41	-	not related to video frequency
193.31 / 9.665 = 20.00	-	20th harmonic of data or 5th harmonic of dot clock
200.08 / 9.665 = 20.70	-	not related to video frequency
233.31 / 9.665 = 24.14	-	not related to video frequency
270.67 / 9.665 = 28.00	-	28th harmonic of data or 7th harmonic of dot clock

3. Actual System Configuration



Graphic Controller	:	EMI Generator
Filter	:	Help to reduce harmonics
Cable, LCD Module	:	Antenna
Metal Shield	:	Should completely cut EMI radition ... but, it creates a signal return path ie. creating differential mode noise

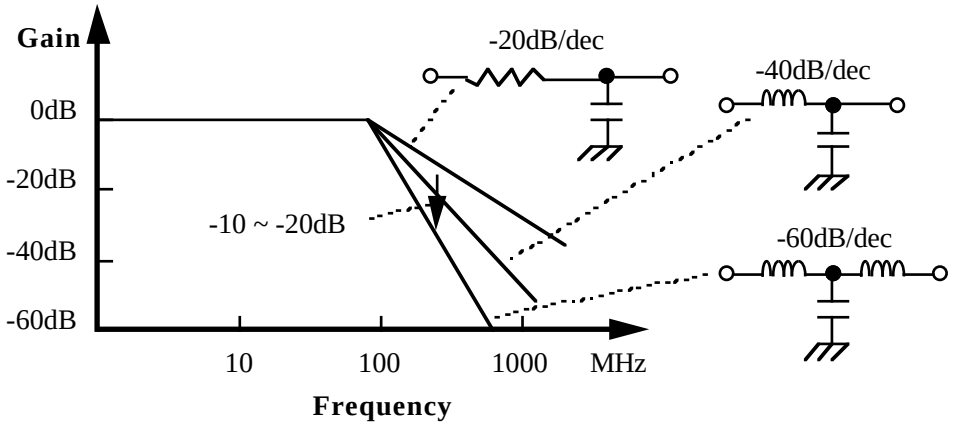
Need to reduce the return current on Metal Shield and case

4. How to reduce EMI?

(1) Apply 3V signal voltage

$$20 \log \frac{3V}{5V} = -4.4 \text{ dB}$$

(2) Replace RC filter to Inductive filter

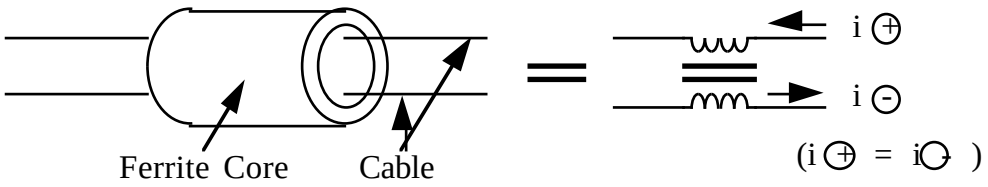


(3) Use flat cable instead of wire cable

Impedance of wire cable is not too low
Well designed flat cable is effective in lowering GND impedance

→ 2 layer cable { 1 layer - signal
1 layer - plain GND

(4) Ferrite Core helps to decrease GND impedance



H8/300H DIRECT MEMORY ACCESS CHANNEL (DMAC) EXAMPLE

The following application note describes the way in which the DMAC is configured to move information on the H8/3042. Data is passed from a location in memory to an output Port every time a value matches the state of an on board timer.

DMAC CHANNEL 0A USING REPEAT TRANSFER ON TIMER 3 COMPARE MATCH WITH REGISTER GRA.

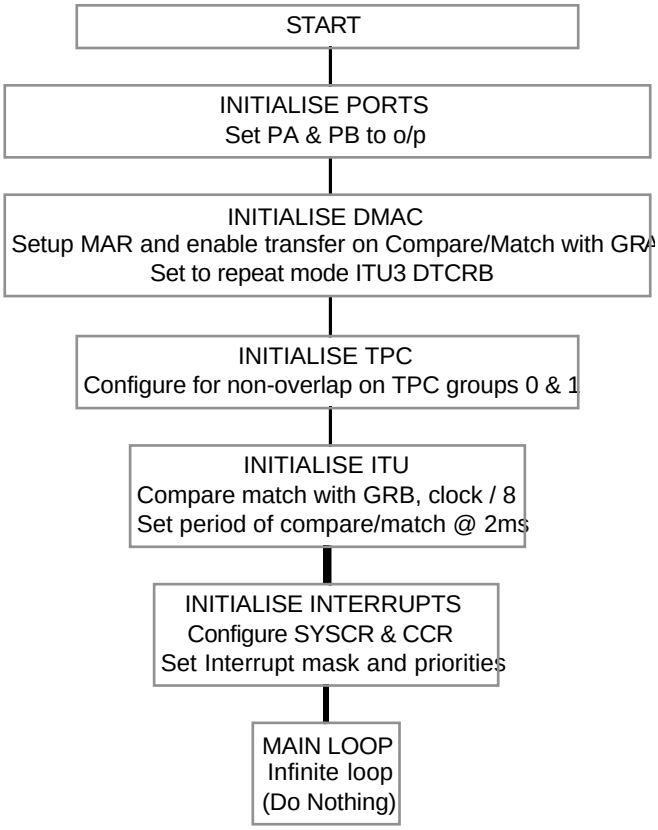
The following software example sets up the following functions:-

- A DMAC channel in repeat mode to place a value (from 0x11 to 0x88) into the NDRA register. The DMAC performs this operation on a compare match of GRA with Timer 3.

- The value held in NDRA is output to the Port A data register (PADR) on a compare/match of GRB with Timer 3.

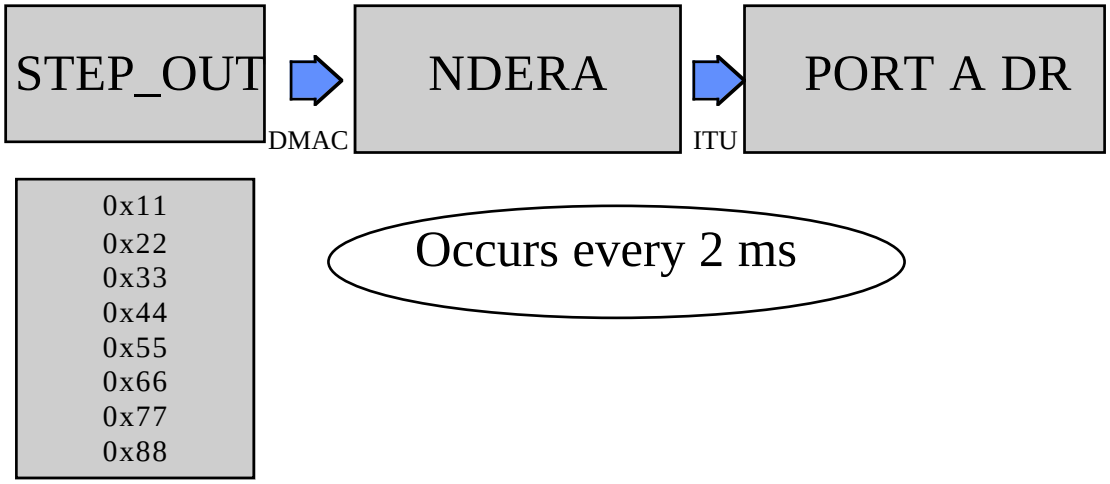
Both of the above functions are performed by peripheral blocks in 'background'. When an emulator 'break' occurs, the CPU is halted, however both functions will continue to run unaffected.

SOFTWARE FLOWCHART



DMAC OPERATION

The DMAC sends the contents of the STEP_OUT array into NDRA on a compare/match with GRA. This is then sent to PADR, outputting the byte to the port data register on compare/match with GRB.



When DMAC channel 0A has completed all its transfers, the MAR is re-initialised and the value 8, placed in the transfer count register.

CODE LISTING

```
/* Software start date 11 January 95. - Version P1.0.
------------------------------------------------------------------------
*/


/* Program starts here */

#pragma language = extended     // enable extensions
#include "ioh83042.h"           // port cast definitions
#include "inh83042.h"           // interrupt vector identifiers
#include "inh83.h"              // in-line functions

//------------------------------------------------------------------------
/*      Function Prototypes      */

void Init_Interrupts(void);
void Init_ITU(void);
void Init_DMAC(void);
void Init_TPC(void);
void Init_Ports(void);

 //------------------------------------------------------------------------
/*      Definitions      */
#define EVER    ;;
//------------------------------------------------------------------------
```

```
/*     Global Variables       */

unsigned char step_out[8] = {0x11,0x22,0x33,0x44,0x55,0x66,0x77,0x88};

//#pragma memory=dataseg(TEST)    // user defined data segment, location
                        // specified by linker
unsigned long dmac_test=0;

//#pragma memory=default

//---------------------------------------------------------------------------
/*     'sfr' & 'sfrp' declarators here        */

sfr PB_DR = 0xFFFD6;

//---------------------------------------------------------------------------
/*     Code starts here       */
C_task main(void)       // C_task forces function not to save
{                       // registers on entry
unsigned char x;

/* Initialise Peripherals */
Init_Interrupts();
Init_Ports();
Init_DMAC();
Init_TPC();
Init_ITU();
for(EVER)
     {
       }
}

void Init_Interrupts(void)
{
/* Configue SYSCR and CCR first */
SYSCR = 0x03;   // set UI bit in CCR as an interrupt mask bit
and_ccr(0x3F);  // enable interrupts, set UI bit to 0 - inline function

/* or alternatively this could be used to enable interrupts   */
set_interrupt_mask(0);   // enable all interrupts

/* Set priorities, Refresh controller and DMAC interrupt have priority. */
IPRA = 0x08;    // Priority for refresh controller
IPRB = 0x20;    // Priority for DMAC interrupts
IER = 0;        // all external ints disabled as default
ISCR = 0x3F;    // all external ints triggered by falling edge input

}
```

```
 void Init_ITU(void)
{
/* Ch 3 - 2mS interrupt to trigger DMA */
ITU_TCR3 = 0xC3;       // counter clear compare match GRB, clk/8
ITU_TIOR3 = 0;         // set GRA & GRB to output compares
ITU_TIER3 = 0x01;      /* interrupt on compare match A, triggers DMA & TPC
                  interrupt flag, automatically cleared by DMAC */
ITU_GRA3 = 400;        // set non-overlap margin (10uS @ 16Mhz)
ITU_GRB3 = 4000;       // set period (2mS @ 16Mhz)
ITU_TSTR = 0x08;       // start channels 0, 2 and 3 - 1 stopped for TPC
}


void Init_DMAC(void)
{
/* configure DMAC for short addressing addressing mode. Channel 0A set
   for I/O mode with a cpu interrupt generated after 8 transfers */
unsigned char temp;
DMAC_MAR0A = (unsigned long)step_out;  // address of look up table
DMAC_IOAR0A = 0xA5;     // NDRA same output trigger for TPC groups 0 & 1
DMAC_ETCR0AH = 8;       // high byte
DMAC_ETCR0AL = 8;       // Low byte, transfer count of 8
temp = DMAC_DTCR0A;     /* dummy read to allow DTE bit to be set to 1 */
DMAC_DTCR0A = 0x93;     /* enable transfers, byte size, increment MAR,
                  repeat mode, trigger compare match A ITU 3 - DTCRB only
                  used for full address mode */
}

void Init_TPC(void)
{
/* configure for non overlap with byte output on TPC groups 0 & 1. Groups
   are triggered by same timer, ITU Ch3. */

TPC_TPCR = 0xFF;       // groups 0 & 1 triggered by ITU Ch3 compare match
TPC_TPMR = 0x03;       // non overlap mode for groups 0 & 1
TPC_NDERA = 0xFF;      // enable transfer of NDERA data to PADR
}

void Init_Ports(void)
{
/* Ports set for following:-
      PortA = Output
      PortB = Output.
*/
PADDR = 0xFF;          // output
PBDDR = 0xFF;          // output
}
```

APPLICATION NOTE **DATE : 14/11/95**

H8/300H DIRECT MEMORY ACCESS CHANNEL(DMAC) - SERIAL COMMUNICATION EXAMPLE

The following example demonstrates how to set-up and use the DMAC for serial communication on the H8/300H series of microcontrollers

DMAC CHANNEL 0A USING REPEAT TRANSFER ON SCI0_TDR (To Transmit) AND DMAC CHANNEL 0B USING REPEAT TRANSFER ON SCI0_RDR (To Receive)

The following software example sets up the following functions:-

- DMAC channel 0A in repeat mode to place a value (from A TRANSMIT BUFFER 'SCI_out') into the SCI0_TDR. The DMAC performs this operation when the transmit buffer is empty.

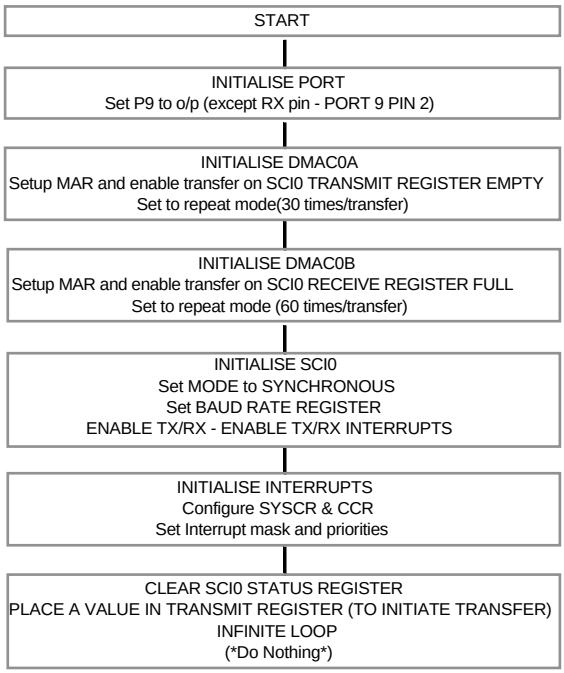
- DMAC channel 0B in repeat mode to place a value from SCI0_RDR into a receive array 'SCI_in'. The DMAC performs this operation when the receive data register is full.

Both of the above functions are performed by peripheral blocks in 'background'. When an emulator 'break' occurs, the CPU is halted, however both functions will continue to run unaffected. The transmit buffer is thirty characters long. The receive buffer is sixty characters long, therefore when the program is run, the receive buffer should contain the transmit message twice.

The serial communication channel can be set up for synchronous communication at the baud rate required eg:-

FAST:- 1Mbaud SMR CK=0 BRR=3
SLOW:- 250 baud SMR CK=3 BRR=249

SOFTWARE FLOWCHART

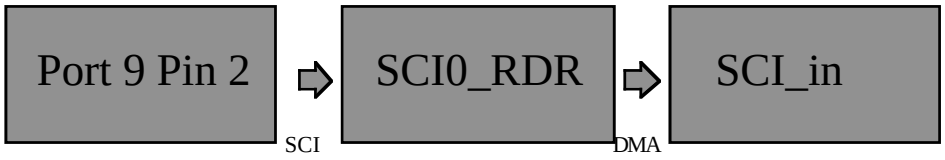


DMAC OPERATION

The DMA Channel 0A sends the contents of the 'SCI_out' array into SCI0_TDR when the transmit data register is empty. The DMA Channel 0B sends the contents of the SCI0_RDR into the 'SCI_in' array when the receive data register is full.

DMA Channel 0A

DMA Channel 0B



When DMAC channel 0A has completed all its transfers, the MAR is re-initialised and the value 30 is placed in the transfer count register.

When DMAC channel 0B has completed all its transfers, the MAR is re-initialised and the value 60 is placed in the transfer count register.

CODE LISTING

```
/*          Software start date 11 January 95. - Version P1.0.
------------------------------------------------------------------------*/


/* Program starts here */

#pragma language = extended     // enable extensions
#include "ioh83042.h"           // port cast definitions
#include "inh83042.h"           // interrupt vector identifiers
#include "inh83.h"              // in-line functions

//---------------------------------------------------------------------------
/*      Function Prototypes      */

void Init_Interrupts(void);
void Init_ITU(void);
void Init_DMAC(void);
void Init_SCI(void);
void Init_Ports(void);

//---------------------------------------------------------------------------
/*      Definitions      */

#define EVER    ;;

//---------------------------------------------------------------------------
/*      Global Variables       */

unsigned char SCI_out[30] = {" HITACHI MICRO SYSTEMS EUROPE "};
unsigned char SCI_in[60] ;
unsigned long dmac_test=0;
//---------------------------------------------------------------------------
/*      Code starts here       */

C_task main(void)       // C_task forces function not to save
{               // registers on entry
unsigned char dummy ;

/* Initialise Peripherals */
Init_Ports();
Init_DMAC();
Init_SCI();
Init_Interrupts();

dummy = SCI0_SSR;
SCI0_SSR = 0x00 ;
SCI0_TDR = 0x20;

for(EVER)
{
}

}
```

```
void Init_Interrupts(void)
{
/* Configue SYSCR and CCR first */
SYSCR = 0x03;   // set UI bit in CCR as an interrupt mask bit

and_ccr(0x3F);  // enable interrupts, set UI bit to 0 - inline function

/* or alternatively this could be used to enable interrupts   */
set_interrupt_mask(0);   // enable all interrupts

/* Set priorities, Refresh controller and DMAC interrupt have priority. */
IPRA = 0x00;
IPRB = 0x20;    // Priority for DMAC interrupts
IER = 0;        // all external ints disabled as default
ISCR = 0x3F;    // all external ints triggered by falling edge input
}

void Init_SCI(void)
{
SCI0_SMR = 0x80;
SCI0_BRR = 0x03;
SCI0_SCR = 0xF0;
}

void Init_DMAC(void)
{

/* configure DMAC0 for short addressing addressing mode. Channel 0A set
   for repeat mode with a cpu interrupt generated after 30 transfers */
/* configure DMAC0 for short addressing addressing mode. Channel 0B set
   for repeat mode with a cpu interrupt generated after 60 transfers */

unsigned char temp;

DMAC_MAR0A = (unsigned long)SCI_out;  // address of look up table
DMAC_IOAR0A = 0xB3;     // SCI0_TDR empty
DMAC_ETCR0AH = 0x1F;       // high byte, transfer count of 31
DMAC_ETCR0AL = 0x1E;       // Low byte, transfer count of 30

temp = DMAC_DTCR0A;     /* dummy read to allow DTE bit to be set to 1 */

DMAC_DTCR0A = 0x94;     /* enable transfers, byte size, increment MAR,
                           repeat mode, trigger TDR empty */


DMAC_MAR0B = (unsigned long)SCI_in;  // address of look up table
DMAC_IOAR0B = 0xB5;     // SCI0_RDR full
DMAC_ETCR0BH = 0x3D;       // high byte, transfer count of 61
DMAC_ETCR0BL = 0x3C;       // Low byte, transfer count of 60

temp = DMAC_DTCR0B;     /* dummy read to allow DTE bit to be set to 1 */

DMAC_DTCR0B = 0x95;     /* enable transfers, byte size, increment MAR,
                           repeat mode, trigger RDR full */
}



void Init_Ports(void)
```

```
{
/* Ports set for following:-
        Port9 = Output -- (Except receive pin)
*/
P9DDR = 0xFB;          // pin 2 (RD0) is an input
P9DR = 0xFB ;
}
```

APPLICATION NOTE **DATE : 22/1/96**

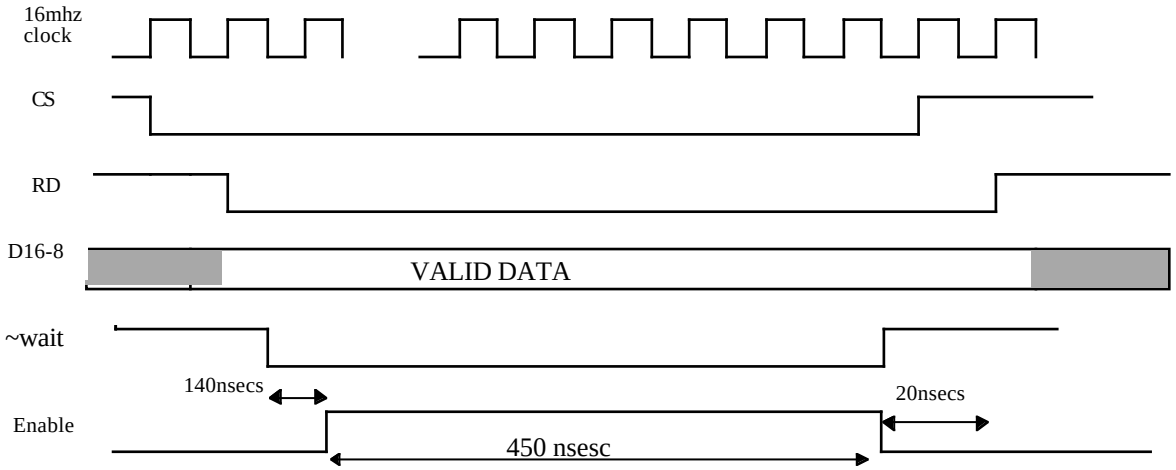
LCD character module control using H8_300H

The following application note details the requirements to interface an LCD module to a typical microcontroller, in this case a 300H processor. The technical note will show details of hardware, firmware and software requirements for the LCD interface. This particular application note will show the LCD interface solution shown as a mixture of hardware and software that should provide the most timely and efficient mechanism of controlling the module. However, it should be realised that the module could be directly controlled from software by using a mechanism of bit bashing and polling input pins. This procedure will not be shown in this application note.

System Specification.

The following specifications of timing requirements are valid for any of the 300H microcontrollers with an LCD module under the control of the HD44780 LCD driver and controller.
The timings show a 300H processor running at 16Mhz with the low level control returning the wait signal to stall the 300H.

Figure 1 : LCD timming for HD44780



Note ; Enable Pulse cycle > 1usecs

The manner in which the 300H will, address and control the LCD module is to use the address and chip select lines to decode the enable line to the LCD module, data will be sent on the high lines of the data bus under direct control of the RD pin. The wait pin signal will again be generated by the decode logic to sufficiently stall the 300H for the period of 610nsecs after the initial access.

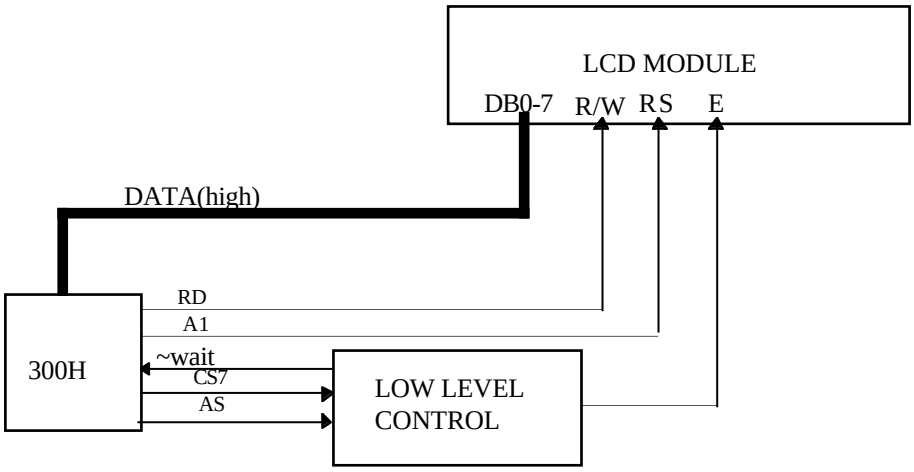
The address of the LCD display shall be designated to be within byte wide memory area 7 with A1 (lowest address line for byte wide memory) used to select between the instruction and data registers.

For completeness the watchdog timer has been used to ensure that the LCD display is not accessed, either read or write, within 1µsecond. The manner by which this is done is to stall the software until the watchdog counts up to 1µsecond. Note this could easily be done by using a simple a simple wait or NOP loop.

Hardware Specification

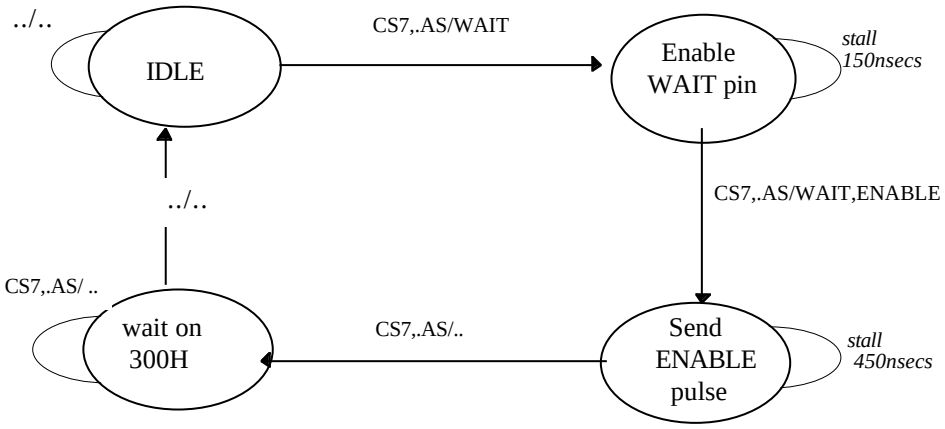
The schematic below shows the hardware involved in the control of the LCD. However, it should be noted that the choice of low level control is up to the final design although the logic architecture must provide registers to enable the construction of a counter and state machine, as shown later.

Figure 2 : Hardware schematics



The low level control requirements are simply to provide the enable and wait signal pulses to the LCD module and 300H respectively. The operation of the logic can be simply defined using the state diagram overleaf.

Figure 3 : Low level control operation



To provide the timing requirements from the initial application of RS and R/W (A1 and RD) to the ENABLE signal at the LCM a simple counter mechanism can be used. For example the counter can be enabled when the address strobe signal is asserted, thereby indicating A1 is valid along with RD, from which the states can be counted until the necessary time

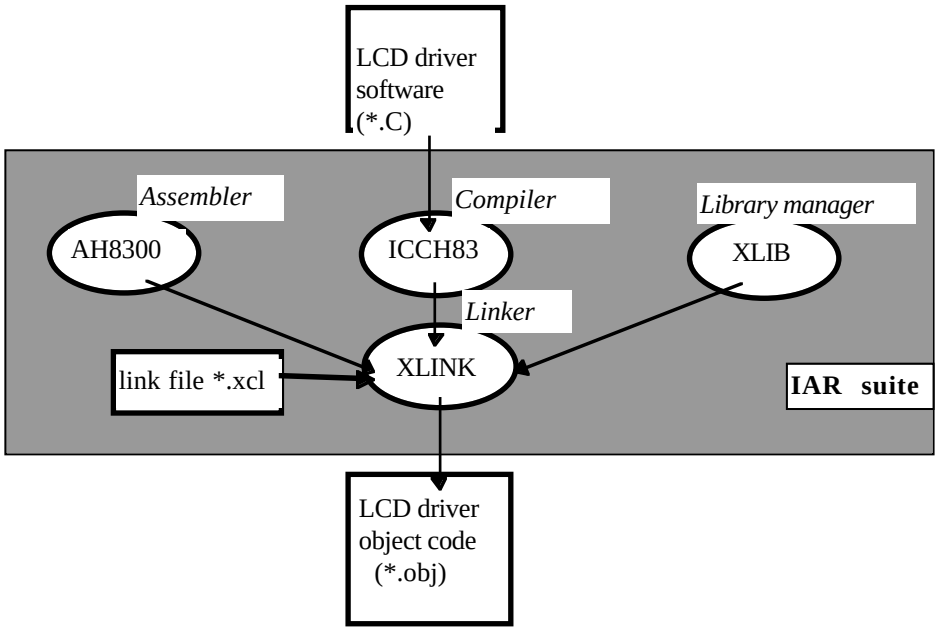
Software Specification

The software requirements for the interface to the LCD screen are shown in the following sections which describe not only the functionality of the code but the generation of it.

Generation of code

The tools that are used for the code generation are that of the IAR compiler/linker system as shown in the figure overleaf.

Figure 4 : IAR code system.



For the purposes of the control of the LCD driver software the program was built using the following options. It should be noted that when choosing development options they should be replicated right across the IAR system build. Since failure to do so could cause errors at the linker stage.

1. Processor options : v3 > H8_300H with a 16Mbyte address space
2. Memory model : ml > far function calls with default data type of huge, stack size is unlimited
3. Floating point precision : -1id > ints are 2 bytes long and doubles are used as floats(4 bytes long)
4. De-bugging level : NA > High level de-bugging not required (C-spy or CIDE not used)
5. Memory configuration :
    - INTVEC - 0-FF > Interrupt vector table
    - RCODE, CODE - 100 - FFFF > Vector handling code and normal code in area0 of device
    - Data, IData : FFFD10 - FFFE00 > Initialised and non-initialised data kept in ON-chip Ram.
6. Stack size : Stack set as 120 bytes
7. Code optimisation : Using standard speed and size optimisation s3 and z4

Consequently, the following commands were used to firstly build the library , compile the code and then linking together.

BUILDLIB -v3 -ml -1id -s3 -z4 ------> Outputs library file CLHSF3.r20

ICCH83 -v3 -ml -1id -s3 -z4 -P Lcd_Drv.c -l Lcd_Drv.lst -----> Outputs Lcd_Drv list and r20 files
XLINK -f linkfile.xcl ------> Outputs executable Lcd_Drv object file

It should be noted that the format of linkfile.xcl is as shown below :

```
lcd_drv
-ch8300h
-o   lcd_drv.obj
-l   lcd_drv.lst
clh83sf3
-FPENTICA-BM
-xsme

-Z(CODE)INTVEC,IFLIST,FLIST=0-ff
-Z(CODE)RCODE,CODE,CDATA0,CDATA1,CDATA2,CDATA3,ZVECT,CONST,CSTR,CCSTR=100

-!  Then the writeable segments which must be mapped to a RAM area
    C000 was here supposed to be start of RAM -!

-Z(huge)DATA3,IDATA3,UDATA3,ECSTR,WCSTR,TEMP=fffd10
-Z(huge)CSTACK+120=Fffde0
-Z(FAR)DATA2,IDATA2,UDATA2=fffd10
-Z(FAR)DATA1,IDATA1,UDATA1=fffd10
-Z(FAR)DATA0,IDATA0,UDATA0,SHORTAD,DATA=fffd10
```

Software functionality

The algorithm below shows the progression of control from reset or power-on, through initialisation to the writing of an ASCII string to the display.

1. Initialise On-chip registers
    - Set-up bus control registers for area 7 - 8-bit, 3 accesses with wait controller enabled
    - Set-up wait state controller - Pin wait mode 1 with 1 wait state automatically included
    - Enable CS7 on port C
    - Set-up watchdog timer to count 1µsecond
2. Initialise LCD display to correct mode
    - Delay for 15msecs after power-on (minimum)
    - Set function interface - 8bits
    - Delay for 4.1msecs to next write to LCD display
    - Set function interface to 8bits
    - Delay for 100usecs
    - Set function to 8-bits (Busy flag now valid to check)
    - Set LCD function/type to 8-bits, 2 lines with char. format of 5x7.
    - Set LCD display,cursor and blink to off.
    - Clear LCD display and return cursor to home position.
    - Set entry mode to increment cursor with accompanied shift left
3. Set pointer to ASCII text string to display
4. Clear LCD display
5. Return cursor to home
6. while not(end of string) do
7. check not end of current LCD row
8. write ASCII character to LCD screen
9. increment pointer to LCD screen
10. increment count for cursor
11. endwhile

It should be noted that for each access to the LCD screen whether reading or writing to either the instruction or data register the following procedure is followed to ensure that the LCD screens operating procedure is not violated (i.e. writing to screen when still busy and making an access to screen within 1µsecond of last access).

1. while (LCD flag still busy)
2. wait for watchdog to count down 1µsecond
3. read busy flag from LCD module
4. endwhile

To access the busy flag is simply a matter or reading the instruction register and checking bit 7 which indicates the busy flag status.

As can be seen from the above algorithm the functionality of the code should be very simple once the LCD module and 300H controller has been properly initialised, thus requiring minimal CPU control. Also note that the watchdog timer should be turned off when in a period of inactivity in accessing the LCD screen as the watchdog timer can use a great deal of processing cycles, since it is interrupt driven.

Register set-up

As shown in the previous section care must be taken in setting up the 300H microcontroller to ensure the LCD module is correctly and effectively interfaced to.

[1] Bus controller set-up

As noted earlier the LCD module has been addressed in area 7 of the 300H memory map consequently a number of registers must be set-up to enable the module to be accessed in 8-bit mode using CS7 and enabling the use of A0 to differentiate between the instruction and data registers respectively.

The above requirements are provided through the following register set-up

ABWCR - Bus width control register

ABW7	ABW6	ABW5	ABW4	ABW3	ABW2	ABW1	ABW0
0	X	X	X	X	X	X	1

ASTCR - Area access state control register

AST7	AST6	AST5	AST4	AST3	AST2	AST1	AST0
1	0	0	0	0	0	0	1

PCDDR - Port C data direction register

PCDDR7	PCDDR6	PCDDR5	PCDDR4	PCDDR3	PCDDR2	PCDDR1	PCDDR0
0	0	1	0	0	0	0	0

[2] Wait state controller

The wait state controller has been enabled in area 7 to provide wait mode 1 which automatically inserts 1 wait state and then thereafter depends if the input wait signal is low at the next low to high transition of the clock. This mechanism should enable the LCD module to stall the microcontroller by the required amount (around 500nsecs), through the low level hardware control.

WCR - Wait control register

X	X	X	X	WMS1	WMS2	WC1	WWC0
X	X	X	X	1	0	0	1

WCER - Wait control enable register

WCE7	WCE6	WCE5	WCE4	WCE3	WCE2	WCE1	WCE0
1	0	0	0	0	0	0	0

[3] Watchdog timer set-up

The following watchdog timer setup should enable the microcontroller to ensure accesses to the LCD module are no more frequent that 1µsecond access cycle. (Clock ticks at 0.125usecs thus count for 8 at least before enabling access)

It should be noted that in the 300H microcontrollers the watchdog registers are password protected and cannot be written to normally.

If using a microcontroller without a watchdog simply stall within the loop with NOP commands.

TCSR - Timer control/status register

OVF	WT/IT	TME	X	X	CKS2	CKS1	CKS0
0	0	1	X	X	1	1	1

TCNT - Timer counter register

bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
1	1	1	1	0	1	1	1

LCD driver code

The attached C-code shows example code that can control the LCD display as specified throughout this document. The code shows example text which can be replaced by any ASCII or a sub-set of KANJI codes to send to the screen.

```
/*************************************************************************
        LCD MODULE DRIVER SOFTWARE.

        Purpose : Software designed to integrate 300H processor to a typical Hitachi LCD
                  module. Applicable units are all the ones listed in the character display
                  catalog.
                  Input text can be derived from serials ports or constant strings, text
                 shown here gives an example of ASCII and KANJI text.
*************************************************************************/

#pragma language = extended
#include <ioh83003.h>              /*300h port defines*/
#include <INH83003.h>              /*300h interrupt vector table*/
```

```
#define   END_ROW1     20                /*Define of length of LCD row*/
#define  NEW_LINE      0x40    /*Define of the cursor position of the 2nd row*/
#define  NULL          0x00     /*ASCII null char. which terminates string*/
#define   CR           0x0d     /*ASCII equivalent of carriage return*/

unsigned char *LCMIR = 0 ;        /*Pointer to the LCD module instruction register*/
unsigned char *LCMDR = 0 ;        /*Pointer to the LCD module data register*/
unsigned char Access_Made = 0 ;   /*Global flag that signifys if it is okay to access LCD display*/

unsigned char *text_string ;      /*Pointer to string to output to LCD display*/

interrupt [WDT_WOVI] void LCD_count(void)

/* Purpose is to count from last LCD access and enable a flag if more than 1µsecond*/
{
unsigned char temp=0;

        Access_Made = 0 ;         /*Clear flag to indicate that it is okay to re-access LCD*/
        /*Setup TCNT register again to count 1µsecond*/
        /*10 ticks of a 8Mhz clock is sufficient for 1µsecond */
        temp = (char)WDT_TCNT & 0xF5 ;
        WDT_TCNT = 0x5A|temp ;          /*Write value to TCNT register*/

}

void reg_setup()

/* Purpose : Procedure to set-up internal 300H registers*/
{
  unsigned char temp =0 ; /*Temp var used to write values to watchdog register*/

        /*Wait state enable register*/
        WCER=0x80 ;    /*Wait control only enabled in area 7 accesses*/
        /*Wait state control register*/
        WCR=0xf9 ;      /*Wait mode 1 with 1 wait state automatically inserted*/
        /*Bus width control register*/
        ABWCR=0x80 ;  /*Area 7 accessed as an 8-bit area - only high data bus*/
        /*Access state control register*/
        ASTCR=0xff ;   /*Area 7 (and all the rest) accessed in a 3 state access*/
        /*Port C data direction control register*/
        PCDDR=0x20 ;  /*CS7 (PC5) enabled as an output - all the rest as generic inputs*/

        /*Set-up watchdog timer - register locked by password*/
        /*10 ticks of a 8Mhz clock is sufficient for 1µsecond */
        temp = (char)WDT_TCNT & 0xF5 ;
        WDT_TCNT = 0x5A|temp ;          /*Write value to TCNT register*/
        /*temp = (char)WDT_TCSR & 0x38 ;  /*Enable timer operation using 8Mhz increment clock*/
        WDT_TCSR = 0xA500|temp ;   /*Use password to write value*/ */


}

void delay(long *delay_val)

/* Purpose : Delay loop for waiting on LCD display - passed value indicates length of delay*/
{
  long times ;

  times = 0 ;
```

```
  while (times < *delay_val)
    {
            times = times + 1 ;
    }

}

void poll_lcd_busy()

/* Purpose : Function to determine if the LCD module is ready for another access*/

{
        while(( *LCMIR & 0x80 )== 0x80 )        /* Check bit 7 of Instruction register*/
                {
                  /*Wait for LCD busy flag to go low*/
                        Access_Made = 0 ;                       /*Included for test */
                        while (Access_Made)
                        { /*Wait for 1µsecond from last access to LCD display*/
                        }
                        Access_Made = 1 ;           /*Clear flag to show an access will be made*/
                }
}


void lcdinit()

/* Routine to initialse LCD display into the required setup - dependant on type */

{
long delay_val=0;                               /*Delay variable for initial initialisation*/

        delay_val=200000;           /*Delay for 15msecs after powerup - 16Mhz clock*/
        delay(&delay_val);
        *LCMIR = 0x30;              /* Funciton Set  - 8bit interface*/
        delay_val=65000;            /* Delay for 4.1msecs*/
        delay(&delay_val);
        *LCMIR = 0x30;              /* Funciton Set - 8bit interface */
        delay_val=2000;             /* Delay for 100usecs*/
        delay(&delay_val);
        *LCMIR = 0x30;              /* Funciton Set - 8bit interface*/
        poll_lcd_busy();            /*Wait for LCD module*/
        *LCMIR = 0x30;                  /* Function Set */
        /* BUSY FLAG CAN NOW BE CHECKED FROM THE LCD MODULE */
        poll_lcd_busy();
        *LCMIR = 0x08;              /* Function set to 8bits, 2lines and 5x7character form*/
        poll_lcd_busy();
        *LCMIR = 0x08;                  /* Display on/off control enable */
        poll_lcd_busy();
        *LCMIR = 0x01;                  /* Clear display and return cursor to home */
        poll_lcd_busy();
        *LCMIR = 0x06;        /* Entry Mode Set as increment with accompanying shift */
        poll_lcd_busy();
        *LCMIR = 0x0C;                  /* Display set to on, cursor and blink off*/
        Access_Made = 0 ;               /*Access to display enabled*/

        /*LCD MODULE NOW READY TO ACCEPT DATA FOR DISPLAYING*/
}
```

```
void text_out()

/*Purpose : Sub-routine to handle passing of char to module - checks line over run*/

{
unsigned char char_cnt = 0;            /*Count of characters sent to LCD screen*/

        char_cnt = 0 ;
        while (*text_string != NULL)
          { /*While text_string pointing to ASCII char*/
                poll_lcd_busy() ;              /*Proceed no further if display busy*/
                if ((char_cnt == END_ROW1)||(*text_string==CR))
                  { /*If new line required from line overrun or carriage return*/
                        *LCMIR = NEW_LINE;  /*Set cursor position to new line position*/
                        char_cnt = 0 ;
                        if(*text_string != CR)
                          { /*If actual character to show on display*/
                                *LCMDR=*text_string ;
                          }
                  }
                else
                  {  /*Simply display next char in next position*/
                        *LCMDR =*text_string;
                        char_cnt++;
                  }
                text_string++ ;                /*Next character to display*/
         }

}



void display_txt()

/*Purpose : Routine to set-up display ready for a new text string*/

{

        *LCMIR = 0x01;            /*Clear display and return cursor to home position*/
        poll_lcd_busy();
        *LCMIR = 0x0C;                  /*Display is set on */
        poll_lcd_busy();
        text_out();                     /*Output text string to LCD module*/
}




main()
{

        /*Set-up the 300H control by initialising registers*/
        reg_setup();

        /*Setup the pointers to the LCD display*/
        LCMIR = ( unsigned char *)0xE04000;     /*Instruction register address*/
        LCMDR = ( unsigned char *)0xE04002;     /*Data register address*/

        /*Initialise the display*/
```

```
        lcdinit();
        /*Write text strings to the LCD display*/
        text_string = "HITACHI  EUROPE  LTD.  LCD MODULE TEXT" ;
        /*Display string over two lines of LCD module*/
        display_txt() ;

}
```

APPLICATION NOTE **DATE : 8/5/96**

Producing Optimised C for 300/300H Controllers

The following application note has been produced to demonstrate features of the IAR toolsuite to enable efficient C coding for all of the 300 and 300H microcontrollers.

The notes in this document will apply to all versions of the compiler and are applicable to all of the microcontrollers in the 3 something range. Assumptions have been made that a certain degree of competence has been obtained in the IAR toolsuite. Although appendix A details a list of contacts and references for the IAR toolset for any teething troubles, the 300 compiler tutorial is recommended as a quick reference.

The following aspects will be covered :

1.0 General C coding practice
- **1.1 Use of modula coding**
- **1.2 Globals vs Local variables**
- **1.4 General C-guidelines**

2.0 Memory Models and Processor options

3.0 Data and Code segment location.

4.0 Processor Bit Representations

5.0 Processor dependent code
- **5.1 Memory indirect addressing**
- **5.2 Inline functions**

6.0 Compiler Optimisation switches
- **6.1 Size and speed optimisations**
- **6.2 Register Utilisation**

As an introduction to the comments on C-coding it is worth refreshing the CPU architectures of the 300 and 300H and their respective instruction sets. The reasoning behind this is that it should make it clear why certain programming techniques should be avoided to obtain maximum performance from the microcontroller.

300 / 300H CPU architecture

In general both of these architectures are based upon a Load/Store type utilising compact instruction sets based around mostly register operations. A further point to note that the architecture addresses in linear fashion

1.0 General C-Coding Practices

1.1 Modula coding principles

When programming to any C-environment it is recommended that a modular approach is used. The reasons for this are :

- Ease of Refining complex problem to code solution
- Maintainability can be obtained by enabling distinct modules to be easily swapped out or changed when necessary.
- Readability for new engineers joining the project
- Enabling portability to other micontrollers.
- Makes possible more refined and localised testing mechanisms.

A further point to note about this is that if code is grouped into similar execution code (code that runs at similar times) it may be possible to reduce the jumps to simple local ones and thus reduce the amount of full address reads required to run the code and thus reduce execution time.

1.2 Global Vs Local variables

In general it is much better programming style and practice to make use of local variables wherever possible. The reason for this is quite simply to reduce the amount of RAM required for DATA structures. The more local a variable is the more feasible it will be to store the value on STACK or in General Purpose Registers. Thus avoiding the need to obtain variable address, read from address and write back value to address, which could possibly be stored as a long pointer requiring 32-bits of data.

For further proof of this it can be seen that by examining the CPU instruction table in the HW manuals operations are about twice as efficient when using registered or immediate addressing modes rather than indirect addressing modes.

Therefore where possible local variables should be used and always passed by reference through functions, thus ensuring that is only one copy of the respective variable.

As an example of this code has been included below and then compiled using the IAR compiler with the achieved data densities as shown below :

Version 1 : Automatic variables utilised

```
char demo(void);

void main(void)
{
char x;

x=demo();

}

char demo(void)
```

Code size = 20 bytes

```
{

char a=5, b=78, c;

return(c=a*b+5);

}
```

Version 2 : Global static variables utilised

```
char demo(void);
char a=5, b=78, c;

void main(void)
{
demo();
}

char demo(void)
{
c=a*b+5;
}
```

Code size = 32bytes

Thus, as illustrated above careful choice of programming style, such as using localised automatics, can have a dramatic effect on the code density produced for the 300 or 300H. Later on example benchmark code will be timed to show how effect of programming style can effect the speed of execution.

1.3 General programming style comments

The following list although by no means definitive gives some final points on programming style that made aid in obtaining Robust, Readable and Correct code for an embedded application which can quite often be critical enough without allowing possible compiler/code bugs to be entered :

- Group all externs required for C file in an include *.h file
- Use static to give a variable file scope rather than making it global to the file
- Create fewer special purpose functions to stop thrashing between code areas.
- Always uses reference names that are meaningful to the system purpose
- Always name any constants that maybe required.
- Never use GOTO unless in a critical error situation.

Repeatedly used code should be grouped into a function to avoid redundancy.

2.0 Memory and Processor options

This section will deal with choosing the correct processor and memory models for a particular application. Careful choice of both these options is essential to achieve the optimum usage of the processor data and code areas. Also enabling the IAR compiler to generate the most efficient code by choosing the most appropriate addressing modes.

Processor options are the simplest to choose, effectively only two factors to determine. Firstly, actual processor of choice (H8_300/300L/300H) and secondly the maximum address range (H8_300H only).

The choice of processor option is simplified even more by the fact that the H8_300 and H8_300L have identical CPU cores with both having a maximum address range of 64K bytes. Consequently the only choice for 300 or 300L users is : v0.

However, for the 300H there is a choice of 3 address ranges that can be chosen : 64K, 1M and 16M; The choice of range depends not only depends on the suitability for the software program but most certainly on the choice regarding the hardware design. For instance it may be a requirement for the hardware to have the full address range of 0 to FFFFFF where in terms of the software it would make much better sense to compile for a 1M address range (0 to FFFFF). However, what ever choice is made the software and hardware modes must be compliant with each other.

Careful choice will be required between modes as this effects the size and location of pointers and function calling mechanisms. For instance a 16M address range requires pointers of a default size range of 32 bits to enable each address in the range to be accessed. Where a 64K address range would not require such a large pointer and could make do with a word pointer.

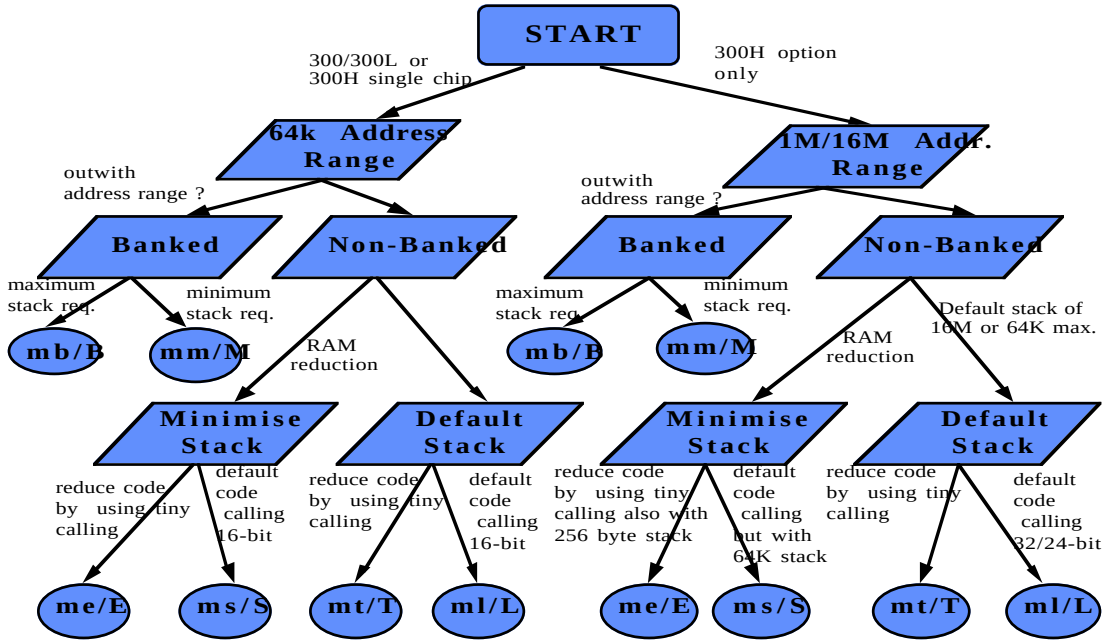
- **v0 - H8_300 OR H8_300L - ADDRESS RANGE = 64K**
- **v1 - H8_300H - ADDRESS RANGE = 64K**
- **v2 - H8_300H - ADDRESS RANGE = 1M**
- **v3 - H8_300H - ADDRESS RANGE = 16M**

Memory model options are much more vast and give the user further control of code generation process. For ease of choice included below is a flowchart which should enable the most efficient choice for a particular application.

When examining the flowchart it should be remembered that the choice of memory model is effected by the choice of processor option. The reason for this being that the processor options specifies the maximum address range and thus the default of the calling mechanisms, details of this is shown in the flowchart.

Further to the above choice variations is that it is possible to specify whether intrinsic library modules are addressed by the tiny_func mechanism or by the default mechanism of the memory model of choice. The case of the memory model specifies whether the space saving mechanism is employed or not. (lower case tiny_func calling mechanism will be employed). A discussion of the tiny_func calling mechanism is made later in this document. However, in general by using the tiny_func mechanism allows the compiler to generate a jsr rather than a bsr and thus save 2 bytes of code size.

Figure 1 : Memory model choice plan



Consequently, although there is a positive wealth of choices they can be quickly tailored down to a choice of around 2 or 3 for each application if the flowchart is followed correctly.

Note that although the choice of maximum stack size will effect directly the amount of RAM required for an application it will also have a considerable effect on the code requirements as well. The reason for this being that if the stack is specified as a maximum of 256 bytes then it can always be accessed as byte, R7L and similarly 64K would be word operations, R7 and 16M would be long operations, ER7. Thus,the code size requirements reduce as the size of maximum stack reduces.

The final choice of model is usually down to the requirements of data addressing, function calling mechanism and stack requirements. The points below give a summary on how the models effect the compiler generation of code.

- **LARGE MODEL -** Default option using standard calling mechanism and the stack calling mechanism represents that the stack can be a size up to the complete address range. No real code and data calling optimisations implemented
- **SMALL MODEL -** Code calling mechanism reflects maximum address range. However, stack size is reduced by a factor that enables the stack to be addressed as a word instead of a long or a byte instead of a word.
- **TINY MODEL -** Stack size is the default of a maximum as large as complete addressing range, word or long addressing. Code addressing is reduced by using tiny_func calling mechanism, as long as the function calling fits into the first 256 addresses of the memory map, where the interrupt vectors reside.
- **EXTRA SMALL -** Code optimisations as tiny model, along with further RAM savings by reducing the stack size to a minimum of 256 bytes. Thus all stack operations are byte wide.

Therefore, although the processor options defines the maximum range it is possible to still configure the software in a manner that makes use of ROM and RAM most efficiently.

3.0 Data and Code Segment direction

Although generally conceived as a linker procedure it is possible to control the final generation of placing code and data at the compiler routine. When referring to placing code and data what is really meant is the assigning of addresses to constant code areas and the addresses of data locations. Consequently, it is very important to achieve the most efficient mapping since it can generally mean the difference between addressing something using a word value or a long value. Not only will the mechanism of addressing data be examined but also procedures in calling functions.

As described in the previous section final address mapping of code can be altered depending on the choice of memory model. The choice of memory model will in general provide the default mechanism to be used by the compiler and linker to produce the object code. However, it is not the sole instrument in storage class description. it is in general possible to use overrides of memory directives where a programmer can determine a particularly time critical part of code that requires to run at optimum speed. Consequently, it may be possible to override the standard memory model with a more suitable one that may translate into assembly and thus machine code that will take less CPU cycles to run.

To enable this the IAR toolset enables non-ANSI functions or #pragmas to directly alter the memory usage. Simply what can be done is to specify memory to be tiny, near, far, huge and back to default as the user may seem fit.

Further to this the programmer has available 3 further #pragmas. Firstly there are two pragmas to enable direct naming of code or constant areas and data areas. The reason for doing this is such that it enables complete control at the linker stage where the actual code or data has to be placed, simply be specify the segment name and specify an actual address (-Z(DATA)DIFF_POSITION=0x900000). Secondly, the programmer can override the data initialisation mechanism by specifing certain variables as no_init. Consequently, this means that no extra ROM area is required for initialising values for data, also as a by product the CSTARTUP routine will obviously run quicker although this is usually not a concern in most systems.

In general the most popular pragmas will be to control what functions can be addresses in a tiny_func mechanism. This is true as often programs are too large to be completely compiled as a tiny model so instead a default mechanism such as large or small can be used and then selectively choose critical modules to be compiled using the tiny_func mechanism i.e. #pragma tiny_func.

The final technique available to the programmer to effect code generation is the use function attribute pragmas. Function attributes enable each individual module to be specified into a particular addressing scheme and also how it is called itself. The availble function attributes fall into two distinct variations =, firstly the ones that detail

4.0 Processor Bit Representation

In many applications the effective representation of bit variables are essential. Either for use in pin manipulation, register set-up or simply as an efficient mechanism of holding status flags.

To provide bit storage and manipulation there are two techniques of representation available to the programmer. Firstly, there is the standard keyword bit that can be used to access any variable in any area of memory as a bit variable and thus used the bit manipulation instructions. Secondly, we can define variables using the no-ANSI standard sfr and sfrp routines. Using these extended language features enables the actual bit to placed in the memory address area specifically set out for bit manipulations, 0xFFFF00 to 0xFFFFFF.

The difference between both these mechanisms although both syntactically correct is the efficiency of representation in terms of both speed of operation and size of object code created. For instance all bit operations must be within the address range 0xFFFF00 to 0xFFFFFF since the addressing range of all of them are tiny, i.e. 0 to FF. Consequently, if data is specified to be of bit type and is not within the tiny addressing range then before the bit operations is completed the data must be transferred to a general purpose register. Example code is shown below detailing the size of object code produced and the number of CPU cycles to complete :

Example Code Using SFRs

```
#pragma language=extended

sfr right_way = 0xFFFF00 ;

main ()
{
        /*Do some bit manipulations */
        right_way.1 = 1 ;
        right_way.2  =  0 ;
        right_way.3 = right_way.2 & right_way.1
         if (right_way.0 == 1)
         {
                right_way.4 = 1 ;
         }
}
```

```
        NAME   bitr(31)
        RSEG    CODE(1)
        PUBLIC main
        EXTERN          ?CL83LD3000_3_22_L00
        RSEG    CODE
main:
        BSET.B          #1,@0:8
        BCLR.B          #2,@0:8
        BLD.B           #1,@0:8
        BAND.B          #2,@0:8
        BST.B           #3,@0:8
        BLD.B           #0,@0:8
        BCC             ?0001
?0000:
        BSET.B          #4,@0:8
?0001:
        RTS
        END
```

Speed of Execution = 7useconds
Code Size = 948 Bytes

Example code using Unsigned chars and Bit

```
/* Bit Testing routine using unsigned chars*/
#pragma extended=language

struct
{
unsigned char Flag1 : 1 ;
unsigned char Flag2 : 1 ;
unsigned char Flag3 : 1 ;
unsigned char Flag4 : 1 ;
unsigned char Flag5 : 1 ;
} Byte_of_Bits;
```

```
          BSET.B          #0,@ER6
          BCLR.B          #1,@ER6
          MOV.L           ER6,ER5
          MOV.L           ER6,ER4
          BLD.B           #1,@ER4
          BAND.B                  #0,@ER5
          BST.B           #2,@ER6
          MOV.B           @Byte_of_Bits,R6L
          BTST.B          #3,R6L
          BNE             ?0001
?0000:
          MOV.L           #Byte_of_Bits,ER6
          BCLR.B          #4,@ER6
?0001:
          RTS
          RSEG    DATA3
Byte_of_Bits:
          DC.B    0,0
          END
```

```
main()
{
        Byte_of_Bits.Flag1 = 1 ;
        Byte_of_Bits.Flag2 = 0 ;
Byte_of_Bits.Flag3 =
     Byte_of_Bits.Flag1 & Byte_of_Bits.Flag2 ;
        if (Byte_of_Bits.Flag4 == 0)
        {
                Byte_of_Bits.Flag5 = 0 ;
        }
}
```

```
NAME  bitw(31)
      RSEG   CODE(1)
      RSEG   DATA3(1)
      PUBLIC Byte_of_Bits
      PUBLIC main
      EXTERN         ?CL83LD3000_3_22_L00
      RSEG   CODE
main:
      MOV.L          #Byte_of_Bits,ER6
```

Speed of Execution = 14useconds
Code size = 1030 bytes

Thus from the code comparisons it can be seen that both speed and size efficiency is gained through using the provided extended compiler functions of SFRs. Not only efficiency is gained but also the ease of use in programming.

5.0 Processor Dependant Code

The following sections now deal with more architecture dependant features that will directly effect the efficiency of representation of the C code.

5.1 Memory Indirect Addressing

The memory indirect addressing mode is an effective way of reducing the overall code implementation of software. The reason for this is by using memory indirect addressing it is possible to use the JSR command rather than a BSR which in turn uses 2 less bytes. However, the only constraint of this is that the function that is being called is within the tiny addressing range. To demonstrate the procedure in making effective use of memory indirect addressing the following code has been included.

```
/*  Addressing Test Program */

#pragma language=extended
#pragma function=tiny_func
void func_tiny()
{
static unsigned char i;
        i ++;
}

#pragma function=default
void norm_func()
{
static unsigned char j;
        j++;
}

main ()
{
        /*Call normal Function*/
        norm_func();
        /*Call tiny Function*/
        func_tiny();
}
```

```
NAME  ftest(31)
        RSEG   CODE(1)
        RSEG   FLIST(1)
        RSEG   DATA3(1)
        PUBLIC ?Flist_func_tiny
        PUBLIC func_tiny
        PUBLIC main
        PUBLIC norm_func
        EXTERN         ?CL83LD3000_3_22_L00
        RSEG   CODE
func_tiny:
        PUSH.W              R6
        MOV.B          @?0000,R6L
        INC.B          R6L
        MOV.B          R6L,@?0000
        POP.W          R6
        RTS
norm_func:
        PUSH.W              R6
        MOV.B          @?0001,R6L
        INC.B          R6L
        MOV.B          R6L,@?0001
        POP.W          R6
        RTS
main:
        BSR            norm_func
        JSR            @@?Flist_func_tiny
        RTS
```

Thus, it can be seen that by using the pragma directive to set-up a function as a tiny rather than the default (Huge in this case) then the JSR command can be used. Which as already discussed uses 2 less bytes in implementation. than a BSR with addressing mode @(d,PC). Further, to this point it is worth comparing the size of object code created. If tiny_func is used the size of code is , where if no tiny_func mode was used the size of code is

5.2 Inline Functions

Intrinsic functions have been provided as extensions to the ANSI library functions to provide more device specific utilities. For a full list it will be necessary to examine the IAR C manual although detailed in the included code below shows specifics of some of the routines.

In general the routines allow control of registers that would only be made possible by jumping to an assembly level module, such as altering the interrupt mask level on the CCR.

The code below not only shows the action of some of the routines it shows that the intrinsic functions are actually Inline. This meaning that it is not necessary to jump to a distinct routine to complete the action, but simply include the necessary assembly instructions. This will give a saving on code size and code execution. The feature of making the functions Inline is even more useful when examination shows how little code is required to complete the operation. more code would be involved in jumping to/from a function rather than executing it.

```
main ()
    8
    9       {
```

```
\  00000000          main:
  10
  11                /*Set interrupt mask in ccr to level 2*/
  12                set_interrupt_mask(2) ;
\  00000000  0480            ORC.B         #128,CCR
\  00000002  06BF            ANDC.B        #191,CCR
  13
  14                /* XOR CCR to value shown*/
  15                xor_ccr(8) ;
\  00000004  0508            XORC.B        #8,CCR
  16
  17
  18                /* Execute sleep command*/
  19                sleep() ;
\  00000006  0180            SLEEP
  20
  21                /* Execute a no-operation*/
  22                no_operation() ;
\  00000008  0000            NOP
  23
  24              /*etc.. */
  25
  26        }
\  0000000A  5470            RTS
\  0000000C                  END
```

Assembly functions included within created object code. No call or jump to a function.

Thus, the main point to consider is that wherever possible use a supplied Inline function, rather than creating your own function to complete the operation. Since, this will result in a much better implementation in terms of code density and speed.

Also note that setting a function as Inline is non-ANSI standard and the IAR compiler only enables the already defined intrinsic functions to be Inline. The reason for this is that complete corruption of code can occur if registers are altered within Inline code without saving the previous value used within the normal code.

6.0 Optimisation Switches at Compile Time

The following section detail switches that can be used when the compiler ICCH83 is invoked to make use of certain compilation algorithms to maximise the efficiency of the code.

6.1 Size and Speed Optimisation

The size and speed optimisation algorithm can be implemented at a number of levels to produce optimised code and data usage. The levels of optimisation will determine the correspondence between the C-code and the generated assembly code. It should be noted that the algorithms are incompatible and thus cannot be used in the same execution of the compiler.

Levels of optimisation range from 0 (min) to 8/9 (max.) with the following effects :

- Level 0-2 - No optimisation and thus can be used for debug processes
- Level 3 - debuggable optimisations only
- Level 4-7 - Medium complex optimisations and thus may not be fully debuggable
- Level 8 - Complex optimisations
- Level 9 - New optimisations for a particular version and thus maynot be fully tested and thus can improve in later versions

The method to choose the optimisation level should be based on a trial basis by re-compiling the code a number of times with different options. Typical results obtained are as follows, although the level of optimisation in general will improve code density to different degrees depending on the code structures. Note that in this case optimising for speed not only achieves the quickest execution speed it also obtains the smallest code size. Although, speed optimisation will not always produce the most compact code.

6.2 Register Utilisation

The final run-time compiler technique that can be used is the register utilisation switches, -u and -W. They give the programmer further control on how the generated code uses registers to clean up and trash.

Using these mechanisms can have a dramatic effect on the code density required for a procedure

Stack Clean Up

The stack clean up option -W directs the compiler in how often it should clean up temporary variables from the stack. Temporary variables are used in complex operations to store intermediary results. The -W option plus a number up to 50 specifies how much of the stack can be used by garbage before resetting the SP back again. With a high -W value less cleans up will occur and thus the code will execute quicker, although to offset this more stack will be required, specifically if the program has a high degree of function nesting.

i.e.

Complex Instruction	Bytes of Stack required	Clean ups : -W0	Clean ups : -W18
A	4	4->0	4
B	6	6->0	10
C	8	8->0	18->0
D	2	2->0	2
E	6	6->0	8

Register trashing conventions

The -u option effects the manner in which registers are used to store parameters and which registers need not be saved by a function and thus can be trashed.

For instance the general procedure is to store the first function parameter in R6 and the rest in the stack with all registers exempt from trashing. This can be extended to store parameters in ER4, ER5 and ER6 with register trashing enabled on ER4, ER5 and ER6. This is the highest level and will provide the most code compact solution.

The option is used as follows :

-ur***x***u***y*** where ***x*** chooses the level of register parameter storing and ***y*** the level of register trashing.

- r0 - Normal procedure R6 stores first parameter
- r1 - ER6 used to store parameter
- r2 - ER5, ER6 used to store parameters
- r3 - ER4 ER5 and ER6 used to store parameters
- u0 - No registers trashed in function call
- u1 - R6 trashed
- u2 - ER6 trashed
- u3 - R5 and ER6 trashed
- u4 - ER5 and ER6 trashed
- u5 - R4, ER5 and ER6 trashed
- u6 - ER4, ER5 and ER6 trashed

Thus, after proving the functionality of the code further levels of optimization can be introduced to reduce code size.

Please remember the library routines must be built with the same optimizations levels or it will not link!

Access Speed on the H8/300H

Minimum access times for external memory of H8/300H series are given in table 1. These minimum times are for two-, three-, and four-state accesses. Note that H8/3048 Series have different minimum access times and these times are presented in table 2.

System clock	8MHz	10MHz	12MHz	16MHz
Min Clock Cycle Time	125.0	100.0	83.3	62.5
Min 2-State Access Time	110	100	80	55
Min 3-State Access Time	230	200	160	115
Min 4-State Access Time	355	300	245	180

Table 1. Minimum access times for external memory of H8/300H Series. Times are in ns. Note that H8/3048 times differ from these (Table 2.).

System clock	8MHz	13MHz	16MHz	18MHz
Min Clock Cycle Time	125.0	76.9	62.5	55.5
Min 2-State Access Time	120	60	60	50
Min 3-State Access Time	240	140	120	105
Min 4-State Access Time	365	220	185	160

Table 2. Minimum access times for external memory of H8/3048 Series. Times are in ns.

Simple example how 8-/16-bit bus width and 2-/3-state access affects performance is shown in Table 3, where a word from external memory address h'6FFFFF is loaded to register R0, incremented by 1 and stored back to external memory address h'6FFFFF. Table 3 presents the amount of states required and table 4 the time in ns required to execute these instructions.

Instruction	On-Chip Memory	8-Bit Bus, 2-State Access	8-Bit Bus, 3-State Access	16-Bit Bus, 2-State Access	16-Bit Bus, 3-State Access
mov.w @aa,R0	8	16	24	8	12
inc.w #1,R0	2	4	6	2	3
mov.w R0,@aa	8	16	24	8	12
States Total	18	36	54	18	27

Table 3. Number of states required for instruction execution with different bus width and different state accesses. Advanced mode.

Instruction	On-Chip Memory	8-Bit Bus, 2-State Access	8-Bit Bus, 3-State Access	16-Bit Bus, 2-State Access	16-Bit Bus, 3-State Access
mov.w @aa,R0	444	888	1332	444	666
inc.w #1,R0	111	222	333	111	167
mov.w R0,@aa	444	888	1332	444	666
Time Total	999	1998	2997	999	1499

Table 4. Time in ns required for instruction execution with different bus width and different state accesses. Advanced mode, 16MHz clock speed.

Real time low power sheduler for the 300L series.

INTRODUCTION

The following example is a framework for a real-time sheduler which makes use of the very low power capabilities of the 300L family.
The 300L family of microcontrollers feature a dual crystal solution; one main oscillator between 0.2 and 10Mhz (0.1-5Mhz internal) and a subclock which uses a standard 32Khz watch crystal. This example uses **full speed** mode running off the main oscillator and **watch** mode using the 32Khz sub clock.

Watch mode is a software triggered standby mode where the main clock is shut down and a few key peripherals are kept running. All other registers and the on board RAM is retained. The members of the 300L family which feature LCD drive can also keep the display active during watch mode.
The key to watch mode is Timer A which functions as the time base timer, again running off the 32Khz subclock. When this timer overflows (Period selectable between $^1/_{32}$, $^1/_8$ $^1/_4$ or 1 second overflow) an interrupt is generated which will wake the 300L up into active mode.
The scheduler can then execute the real time tasks at full speed and instantly switch back to watch mode on completion. The power consumption reduction by switching modes can be as much as from 10mA down to 15µA, which is particularly applicable to battery operation.

The example considered here uses a quarter second overflow period, and by using two software counters a simple scheduler providing $^{1/}_4$, one second and ten second schedule periods although these could be adjusted to any value within reason. To demonstrate the operation of the code each sheduler period function toggles a port (port 2) which can be connected to an L.E.D. via a buffer.

FUNCTIONS OVERVIEW

main()

This contains only the watch mode call which always happens as soon as the real time scheduler has finished its current tasks.

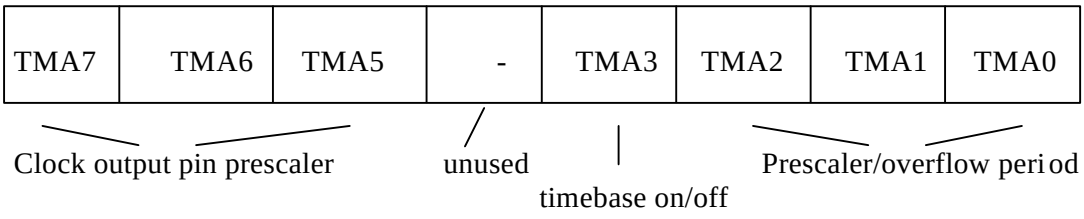
interrupt [TIMER_A] void schedule (void)

This is the interrupt service routine for timer A. Its purpose is to reset the interrupt request (to disable continual looping of the routine) and then call the scheduled functions.

void initialisation (void)

Here the ports controlling the LED's are initialised to outputs and timer A is set up as the timebase. Also timer A interrupt is enabled to allow it to wake the CPU up form watch mode on and overflow. The final and important initialisation is using the in-line function set_interrupt_mask(0) which clears the I bit in the CCR. This globally enables interrupts by reseting the mask from the default (1).

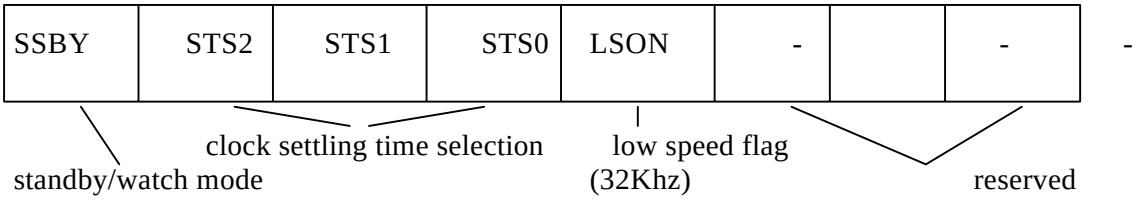
TMA- Timer A control register set to HEX ba = 1011 1010



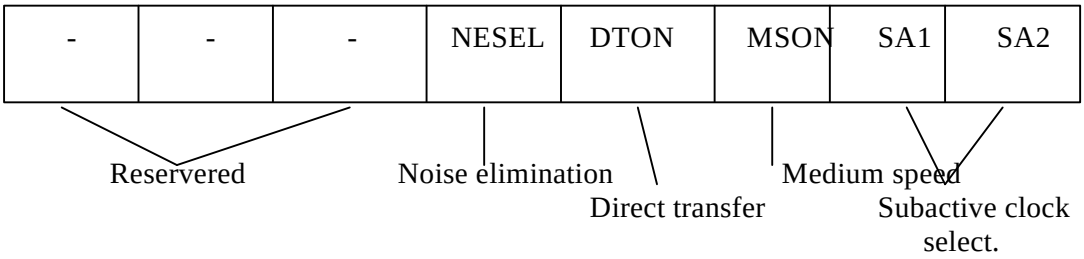
void watch(void)

The watch function sets the system control registers to tell the CPU to enter watch mode when the sleep instruction is called. SLEEP is another in-line extension to ANSI C provided by the IAR C compiler.

SYSCR1 System control Register 1 - set to HEX b0 = 1011 0000



SYSCR2 System control register 2 - set to HEX 00 - 0000 0000



void quarter_second_tasks(void)
void one_second_tasks(void)
void ten_second_tasks(void)

These functions are where the user should place or call the scheduled tasks. The only two restrictions to be considered are:-

1) The worst case when all 3 task calls are entered (once every ten seconds), the total execution time should be less than the schedule period.

2) Other interrupts will not be serviced as the schedule calls are themselves called from the interrupt service routine. This means the I bit will be set, preventing any further interrupts until it has finished. One way to avoid this is to use a global flag to request a schedule task, with the schedule call only setting the flag.

300LRTC.C

```
/*****************************************************************
This program is an a simple example of a low power, real time scheduler.It can
wake up and execute tasks on a periodic basis.
The 300L initially sit in WATCH mode (see main) until timer A overflows and
causes it to wake up into full speed execution mode. Three schedule time periods
are shown here, 1/4 second, 1 second and 10 seconds. Each schedule function
flashes an led to prove the operation of the program.
*****************************************************************/


#include "c:\h8\inc\ioh83834.h"   /* H8/3834 IO labels */
#include "c:\h8\inc\inh83834.h"   /*and the interrupts */
#include <inh83.h>                /*include interrupt std functions*/

/* ********** */
/* Variables  */
/* ********** */

unsigned char one_second,ten_seconds;



/* ********** */
/* Functions  */
/* Declaration*/
/* ********** */

interrupt [TIMER_A] void schedule(void);
void initialisation(void);
void watch(void);
void quarter_second_tasks(void);
void one_second_tasks(void);
void ten_seconds_tasks(void);



void main(void)
{
              initialisation(); /*set up timer A overflow interrupt*/
                                /*and the port registers           */


              for (;;)
              {


            /*The processor return to this section of the code as
      soon as the scheduler has finished its tasks.
            When an interrupt occurs the 300L will wake up and jump          to
the relevant interrupt service routine.*/

              watch();
              }
}
```

```
interrupt [TIMER_A] void schedule(void)
{
              IRR1 &= 0x7f;   /*reset the interrupt request.  */
                              /*this lot must be <250mS!!!!!!!*/
                              /*otherwise the scheduler will  */
                              /*loose synchronism             */


              quarter_second_tasks();
                              /*call this function every time as
                              the processor will only reach here
                              once every quarter second*/


              one_second++;         /*increment one_second count*/

              if (one_second==4) one_second_tasks();

                              /*when the one second count has reached
                              four then the time is 4*0.25 = 1 second
                              If this has happened then the
                  one_second_tasks function is called
      which contains the tasks which need to
                              be done once a second*/

              ten_seconds++;  /*increment ten_seconds count*/

              if (ten_seconds==40) ten_seconds_tasks();

                              /*when the one second count has reached
                              forty then the time is 40*0.25 = 10
                  seconds. If this has happened then the
      one_second_tasks function is called                                  which
contains the tasks which need to                              be done once a
second*/

}
void initialisation(void)
{
one_second = 0;         /*reset one_second count                */
ten_seconds = 0;        /*reset ten_seconds count               */

PMR2 = 0xf8;            /*Set port 2 bottom 3 bits to i/o       */
PCR2 = 0x07;            /*Set all bits in port 2 to outputs     */
PDR2 = 0x00;            /*switch ledS ofF initially             */


IENR1 |= 0x80;          /*enable interrupt on timer a           */
TMA = 0xba;             /*bits 7,6,5 set o/p to 2Khz            */
                        /*bit 4 reserved                        */
                        /*bits 3,2,1,0 set overflow period to 0.25s*/

set_interrupt_mask(0);  /* Clear I bit to enable global interrupts*/

}

void watch(void)
```

```
{
            SYSCR2 = 0x00;  /*set all bits low (medium speed off)*/
            SYSCR1 = 0xb0;  /*set bit 7 ssby bit, and bits 4,5 for
                64k states*/
            sleep();        /*go for it!*/
}


/************************************************
The three fuctions here are the scheduler functions.
Depending on how often you whish to call a certain
fiunction, place it in the quarter,one or ten second
scheduler function.
************************************************/

void quarter_second_tasks(void)
{
           /*Toggle Led to prove operation     */
if(PDR2 & 0x01) PDR2 &= 0xfe;
           /*if led1 is on, switch it off      */
else            PDR2 |= 0x01;
           /*otherwise switch it on !!   */
}


void one_second_tasks(void)
{
one_second=0;     /*clear one second count to zero*/
if(PDR2 & 0x02) PDR2 &= 0xfd;
                  /*if led2 is on, switch it off  */
else             PDR2 |= 0x02;
                  /*otherwise switch it on !!     */
}
```

```
void ten_seconds_tasks(void)
{

ten_seconds=0;    /*clear ten second count to zero*/
if(PDR2 & 0x04) PDR2 &= 0xfb;
                  /*if led3 is on, switch it off  */
else             PDR2 |= 0x04;
                  /*otherwise switch it on !!     */
}
```