White Paper
Sam Fleming
Technical Marketing Engineer
Intel Corporation

Accessing the Real Time Clock Registers and the NMI Enable Bit

A Study in I/O Locations 0x70-0x77

January 2009

Executive Summary

There are two banks of Real Time Clock (RTC) registers that are located inside the silicon of most Intel® chipsets, including the Intel® EP0579 and Intel® System Controller Hub US15W chipset components. The registers are accessed via I/O addresses 0x70-0x77. The bit at 0x70[7] also happens to be the NMI Enable bit. These registers are commonly accessed by BIOS, operating systems, and sometimes custom applications. Accessing these registers, however, is not trivial. These registers are aliased in various ways depending on which RTC register bank is being accessed. Additionally, the chipset must be put in a different mode called Alt-Access Mode in order to successfully read some of the registers in some of the modes.

These registers are commonly accessed by BIOS and operating systems. Accessing these registers, however, is not trivial.

This paper will discuss the full process software needs to follow to successfully access these registers. It will also provide some sample code showing the necessary steps software needs to take in order to successfully read and write to these registers.

The simplest way of reading the RTC registers is to use the register pair located at I/O locations 0x74 and 0x75 to access the standard bank and the register pair located at I/O locations 0x72 and 0x73 to access the extended bank. The NMI Enable bit nominally located at 0x70[7] is best accessed ONLY when the chipset is put into Alt-Access Mode. This guarantees successful read and write cycles regardless of the settings of the other registers.§

Contents

Background

The registers needed to access the two banks of RTC registers at I/O locations 0x70-0x77 have been present since the early days of personal computers (PC). They were originally located in the Real Time Clock circuitry itself before being absorbed into Intel silicon as PC designs matured. As such, it retains a lot of the legacy limitations inherent with earlier architectures (aliasing, etc).

Accessing these registers can be difficult. The NMI Enable bit (NMI_EN = I/O 0x70[7]) is especially troublesome as a straight read of this register will return all 0xFF data, although writes work fine. This paper will explain the details and the necessary steps that software needs to perform to access these registers.

Real Time Clock (RTC) Battery Backed Up RAM

There are two banks of 128 bytes each of battery backed up RAM locate in most Intel chipsets, typically in the “Southbridge” or ICH# component. One bank is called the Standard Bank, the other is called the Extended Bank. In the Standard Bank, fields 0x00-0x0D (first 14 bytes) are reserved for various functions (see Table 1.).

Table 1. RTC Standard and Extended RAM Bank

Index	Standard Bank	Extended Bank
0x00	Seconds	
0x01	Seconds – Alarm	
0x02	Minutes	
0x03	Minutes – Alarm	
0x04	Hours	
0x05	Hours – Alarm	
0x06	Day of Week	
0x07	Day of Month	
0x08	Month	
0x09	Year	

Index	Standard Bank	Extended Bank
0x0A	Register A	
0x0B	Register B	
0x0C	Register C	
0x0D	Register D	
0x0E-0x7F	114 Bytes of User RAM	
0x80-0xFF	Do not use or access	

This data is accessed through a standard Index/Data Register Pair, which will be discussed further in the next section. There is nominally a separate set of Index/Data Register Pairs for each bank, but aliasing of these I/O locations and some Read/Write limitations makes accessing this data somewhat complex.

Registers A-D are not really RAM locations. Instead, these are the four registers used to access and program the functions of the Real Time Clock. Details of these four registers are beyond the scope of this document.

Real Time Clock I/O Registers

There are four sets of Index/Data register I/O location pairs that are used to access the standard and extended bank RAM locations. Due to aliasing, it is much more complex than this.

Table 2. Index/Data I/O Register Pairs Used to Access RTC RAM

Index Register	Data Register
0x70[1]	0x71
0x72	0x73
0x74	0x75
0x76	0x77

1. I/O Location 0x70 operates differently from the other Index Registers. This register is fully writeable, but it can only be read when Alt-Access Mode is enabled.

There are two different ways that the registers get aliased. Which method of alias depends on the setting of the 128E - Upper 128 Byte Enable bit (D31:F0-D8[2]). Setting this bit to a 1b enables the upper, or Extended Bank, of RTC Memory. When this bit is set to a 0b, all four sets of registers in Table 2 alias to I/O locations 0x70 and 0x71. See Table 3 and Table 4 for details on the aliasing.

Table 3. I/O Register Aliasing When Bit 128E=1b (Standard and Extended Banks Enabled)

	Standard Bank Index Register	Standard Bank Data Register	Extended Bank Index Register	Extended Bank Data Register
Primary I/O Address	0x70[1]	0x71	0x72	0x73
Aliased I/O Address	0x74	0x75	0x76	0x77

1. I/O Location 0x70 operates differently from the other Index Registers. This register is fully writeable, but it can only be read when Alt-Access Mode is enabled.

Table 4. I/O Register Aliasing When Bit 128E=0b (Only Standard Bank Enabled)

	Standard Bank Index Register	Standard Bank Data Register	Extended Bank Index Register	Extended Bank Data Register
Primary I/O Address	0x70[1]	0x71	Disabled	
Aliased I/O Address	0x72	0x73		
	0x74	0x75		
	0x76	0x76		

1. I/O Location 0x70 operates differently from the other Index Registers. This register is FULLY writeable, but it can only be read when Alt-Access Mode is enabled.

Most programmers tend to use I/O pair 0x74 and 0x76 when accessing the Standard Bank RTC RAM registers. I/O 0x74 can be read and written normally (with the exception of 0x70[7], or any of its aliases).

Note: Reads to aliases of 0x70[7] always return 0b in any mode.

Note: ANY write to 0x70[7] or an alias does work, even when NOT in Alt-Access Mode.

NMI Enable – 0x70[7]

The Non-Maskable Interrupt (NMI) Enable bit is 0x70[7] (and its aliases). This bit must be 0b to enable the Intel® chipset to send a Non-Maskable Interrupt. When set to a 1b, NMI's are disabled. This bit is commonly accessed by applications, BIOS, and even the operating system since it is used to block NMI assertions when sensitive code is executing.

Any write to 0x70[7] or an alias will set/clear the bit - in any mode. The problem software has is in trying to read the bit 0x70[7] is only readable

when in Alt-Access Mode, and only when I/O location 0x70 is read. Reads of an alias of 0x70[7] will always return a 0b. Alt-Access Mode is enabled by setting D31:F0-D0[6]=1b.

Note: The test code locks up if you leave the Alt-Access Bit enabled for any length of time. Set it, read 0x70[7], and quickly disable Alt-Access.

Sample Code

Figure 1 is a simple DOS program written in C which demonstrates everything discussed in this paper. The program performs read and writes to all of the RTC access I/O Registers (I/O locations 0x70-0x77) in all of the various modes discussed above:

- Lower Bank of RTC (U128E disabled)
- Upper Bank of RTC (U128E enabled)
- Alt-Access Mode

It also performs a demonstration of how locking of the two banks works.

Figure 1. Sample Code Demonstrating How to Access I/O Locations 0x70-0x77

```
#define PROGRAM_STRING "Port70"
#define COPYRIGHT_STRING "Copyright 2008, Intel Corporation"
#define VERSION_NUMBER_STRING "2.0"

#define EXE_NAME "PORT70.EXE"

#include <conio.h>      // For clrscr();
#include <stdlib.h>     // Needed for exit(0);
#include <stdio.h>      // Needed for printf
#include <string.h>     // Needed for strlen
#include <ctype.h>      // For toupper() function

#include "basedef.h"
#include "pci_acc.h"  // Needed for PCI Access Routines
#include "mem_acc.h"

//=========================================================
// Local Function Declarations (Forward References)
//=========================================================
void U128EnableTest();
void U128DisableTest();
void AltAccessTest();
void Upper128ByteLockTest();
```

```
void Lower128ByteLockTest();
void InitRegs();
void dumpstatus();
void dumpreg();

//*************************************************************
//*************************************************************
void main()
{
   int tempbyte, i, j;

   j=0; j=j+1;
   i=0; i=i+1;
   tempbyte=0; tempbyte=tempbyte+1;

   char option;
   BOOL quit = FALSE;

   while (!quit)
   {
      clrscr();
      cout <<   "         Main Menu            " << endl;
      cout <<   "1. U128 (E)nable Test         " << endl;
      cout <<   "2. U128 (D)isable Test        " << endl;
      cout <<   "3. (A)lt Access Test          " << endl;
      cout <<   "4. (U)pper 128 Byte Lock Test " << endl;
      cout <<   "5. (L)ower 128 Byte Lock Test " << endl;
      cout <<   "X. E(x)it " << endl;
      cout <<   "Option# > " ;

      option = toupper(getche());  // Get char and make it upper
case.
      if ( !quit )
      {
         switch(option)
         {
            case '1':
            case 'E':
               U128EnableTest();
               break;
            case '2':
            case 'D':
               U128DisableTest();
               break;
            case '3':
            case 'A':
               AltAccessTest();
               break;
            case '4':
            case 'U':
                Upper128ByteLockTest();
               break;
            case '5':
            case 'L':
```

```
                Lower128ByteLockTest();
               break;
            case 'X':
            case 13 : // <CR> - I always like this to exit.
            case 27 : // <ESC. - Another key I like to cause an
exit.
               quit = TRUE;
               break;
            default:
               break;
         } // end of main_menu switch
      } // end of main menu quit
   }  //end of main_menu while
   exit(0);
}

//*********************************************************
//*********************************************************
void U128EnableTest()
{
   UCHAR  Bdata;

   clrscr();
   // ================= Enable U128E =====================
   cout << "\n*** Enabling U128E = 1b ***\n";
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata | 0x04;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   InitRegs();

   dumpstatus();
   dumpreg();

   cout << "\n*** Write 0x70<=0xB7   Write 0x71<=0x42 ***";
           outportb(0x70, 0xB7);
                           outportb(0x71, 0x42);
   cout << "\n*** Write 0x76<=0xA9   Write 0x77<=0x21 ***\n\n";
           outportb(0x76, 0xA9);
                           outportb(0x77, 0x21);
   dumpreg();

   cout << "\nLessons Learned:  ";
   cout << "\n  1) When U128E enabled, 0x70/71 aliases to
0x74/0x75. 0x72/0x73 to 0x76/77.";
   cout << "\n  2) I/O 0x70 is NOT readable when Alt Access is
disabled.";
   cout << "\n  3) I/O 0x70/72/74/76[7] is WRITEABLE, but not
READABLE when Alt-Acc Disabled.";
   cout << "\n     (Test 3 proves this conclusively).";

   getch();
}

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

```
void U128DisableTest()
{
   UCHAR  Bdata;

   clrscr();
   // ================== Disable U128E ====================
   cout << "\n*** Disabling U128E = 0b ***\n";
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0xFB;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   InitRegs();

   dumpstatus();
   dumpreg();

   cout << "\n*** Write 0x70<=0x9C   Write 0x71<=0x35 ***\n";
           outportb(0x70, 0x9C);
                              outportb(0x71, 0x35);
   dumpreg();

   cout << "\nLessons Learned:  ";
   cout << "\n  1) When U128E disabled, all I/O ranges alias to
the I/O 0x70 and I/O 0x71.";
   cout << "\n  2) I/O 0x70 is NOT readable when Alt Access is
disabled.";
   cout << "\n  3) I/O 0x70/72/74/76[7] is WRITEABLE, but not
READABLE when Alt-Acc Disabled.";
   cout << "\n     (Test 3 proves this conclusively).";
   getch();
}
//*********************************************************
//*********************************************************
void AltAccessTest()
{
   UCHAR  Bdata;

   clrscr();
   // ================== Disable U128E ====================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0xFB;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   InitRegs();

   dumpstatus();
   cout <<   "*** Write 0x74<=0x97   Write 0x71<=0x12 ***\n";
                outportb(0x74, 0x97);
                                   outportb(0x71, 0x12);
   dumpreg();

   cout << "\n*** Enabling Alt Access Mode ***\n";
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD0, &Bdata);
   Bdata = Bdata | 0x40;
```

```
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD0, &Bdata);
   dumpstatus();

   dumpreg();

   // Turn off Alt-Access Mode (Or system locks)
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD0, &Bdata);
   Bdata = Bdata & 0xBF;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD0, &Bdata);

   cout << "\nLessons Learned:  ";
   cout << "\n  1)  ALT-ACCESS permits READS to I/O 0x70[7:0].
Even 0x70[7] is readable.";
   cout << "\n  2)  STILL can't read I/O 0x72/74/76[7]";
   cout << "\n  3)  Note:  Writes to I/O 0x70/72/74/76[7] DO work
BEFORE Alt-Acc";
   getch();
}

//*********************************************************
//*********************************************************
void Lower128ByteLockTest()
{
   UCHAR  Bdata;

   clrscr();
   // ================== Disable U128E ====================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0xFB;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   InitRegs();
   dumpstatus();

   cout << "*** Write 0x70<=0x3A  Write 0x71<=0x11 ***\n";
               outportb(0x70, 0x3A); outportb(0x71, 0x11);
   dumpreg();

   // ================== Enable L128LOCK =============+++++
   cout << "*** Enabling L128LOCK = 1b ***\n";
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata | 0x08;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   dumpstatus();

   cout << "*** Write 0x70<=0x3A  Write 0x71<=0xEE ***\n";
             outportb(0x70, 0x3A); outportb(0x71, 0xEE);
   dumpreg();

   // Renable U128E
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata | 0x04;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   cout << "Lessons Learned:  ";
```

```
   cout << "\n  1) L128 Lock works for ranges 0x38-0x3F.";
   cout << "\n  2) Both Reads and Writes locked to I/O 0x38-0x3F.
Reads return garbage.";
   cout << "\n  3) Once locked, region cannot be unlocked.
Reboot to unlock.";
   getch();
}

//*********************************************************
//*********************************************************
void Upper128ByteLockTest()
{
   UCHAR  Bdata;

   clrscr();

   // ================== Upper U128E Enable ===============
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata | 0x04;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   InitRegs();
   dumpstatus();

   // Note, I/O 0x38-0x3F are blocked.
   cout << "*** Write 0x72<=0x38  Write 0x73<=0x11\n";
            outportb(0x72, 0x38);
                              outportb(0x73, 0x11);
   dumpreg();

   // ================== Enable U128LOCK ==================
   cout << "\n*** Enabling U128LOCK = 1b ***\n";
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata | 0x10;
   PCI_WriteCfgByte(0, 0x1F, 0, 0xD8, &Bdata);

   dumpstatus();

   cout << "*** Write 0x72<=0x38  Write 0x73<=0xEE ***\n";
               outportb(0x72, 0x38);
                                 outportb(0x73, 0xEE);
   dumpreg();

   cout << "\nLessons Learned:  ";
   cout << "\n  1) U128 Lock works for ranges 0x38-0x3F.";
   cout << "\n  2) Both Reads and Writes locked to I/O 0x38-0x3F.
Reads return garbage.";
   cout << "\n  3) Once locked, region cannot be unlocked.
Reboot to unlock.";
   getch();
}

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

```
void dumpreg()
{
   UCHAR  Bdata;
   int tempbyte, i;
   char * string;
   string = new char[80];

   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0x04;
   if (Bdata)     // U128 Enabled
      {
      cout << "
*** Reads ***                \n";
      cout << "                                    LB Index    LB
Target    UB Index    UB Target\n";
      cout << "                                    0x70=>0x";
      tempbyte=inportb( 0x70);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                        "0x71=>0x";
      tempbyte             = inportb( 0x71);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16));

      cout << "   0x72=>0x";
      tempbyte=inportb( 0x72);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                        "0x73=>0x";
      tempbyte             = inportb( 0x73);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";
      cout << "                                    0x74=>0x";
      tempbyte=inportb( 0x74);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                        "0x75=>0x";
      tempbyte             = inportb( 0x75);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16));
      cout << "   0x76=>0x";
      tempbyte=inportb( 0x76);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                        "0x77=>0x";
      tempbyte             = inportb( 0x77);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";
```

```
      }
   else          //  U128 Disabled
      {
      cout << "
*** Reads ***     \n";
      cout << "                                                    LB
Index    LB Target\n";
      cout << "
0x70=>0x";
      tempbyte=inportb( 0x70);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                          "0x71=>0x";
      tempbyte             = inportb( 0x71);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";

      cout << "
0x74=>0x";
      tempbyte=inportb( 0x74);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                          "0x75=>0x";
      tempbyte             = inportb( 0x75);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";

      cout << "
0x72=>0x";
      tempbyte=inportb( 0x72);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                          "0x73=>0x";
      tempbyte             = inportb( 0x73);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";

      cout << "
0x76=>0x";
      tempbyte=inportb( 0x76);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "  ";
      cout <<                          "0x77=>0x";
      tempbyte             = inportb( 0x77);
      for (i = 1; i<=(2-strlen(itoa(tempbyte,string,16))); i++)
         cout << "0";
      cout << strupr(itoa(tempbyte, string, 16)) << "\n";
```

```
      }
}

//*********************************************************
//*********************************************************
void dumpstatus()
{
   UCHAR  Bdata;

   // ============== ALT ACCESS MODE =====================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD0, &Bdata);
   Bdata = Bdata & 0x40;
   if (Bdata)
      cout << "AltAccess=ENABLED    ";
   else
      cout << "AltAccess=DISABLED   ";

   // =================== U128E ==========================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0x04;
   if (Bdata)
      cout << "U128E=ENABLED    ";
   else
      cout << "U128E=DISABLED   ";

   // =================== L128LOCK =======================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0x08;
   if (Bdata)
      cout << "L128LOCK=ENABLED    ";
   else
      cout << "L128LOCK=DISABLED   ";

   // =================== U128LOCK =======================
   PCI_ReadCfgByte(0, 0x1F, 0, 0xD8, &Bdata);
   Bdata = Bdata & 0x10;
   if (Bdata)
      cout << "U128LOCK=ENABLED    \n";
   else
      cout << "U128LOCK=DISABLED   \n";
}

//*********************************************************
//*********************************************************
void InitRegs()
{
   // Filling Registers with Junk to validate Demo
   outportb(0x70, 0x19);   outportb(0x71, 0x11);
   outportb(0x72, 0x22);   outportb(0x73, 0x33);
   outportb(0x74, 0x44);   outportb(0x75, 0x55);
   outportb(0x76, 0x66);   outportb(0x77, 0x77);
}
```

Results

The results of running each of the commands in the sample code are shown in Figure 2:

Figure 2. Output of Sample Code

```
         Main Menu

1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > 1
================================================================================
*** Enabling U128E = 1b ***
AltAccess=DISABLED   U128E=ENABLED    L128LOCK=DISABLED
U128LOCK=DISABLED
                                              *** Reads ***
                                LB Index    LB Target    UB Index    UB
Target
                                0x70=>0xFF  0x71=>0x55   0x72=>0x66
0x73=>0x77
                                0x74=>0x44  0x75=>0x55   0x76=>0x66
0x77=>0x77

*** Write 0x70<=0xB7   Write 0x71<=0x42 ***
*** Write 0x76<=0xA9   Write 0x77<=0x21 ***

                                              *** Reads ***
                                LB Index    LB Target    UB Index    UB
Target
                                0x70=>0xFF  0x71=>0x42   0x72=>0x29
0x73=>0x21
                                0x74=>0x37  0x75=>0x42   0x76=>0x29
0x77=>0x21

Lessons Learned:
  1) When U128E enabled, 0x70/71 aliases to 0x74/0x75; 0x72/0x73 to
0x76/77.
  2) I/O 0x70 is NOT readable when Alt Access is disabled.
  3) I/O 0x70/72/74/76[7] is WRITEABLE, but not READABLE when Alt-Acc
Disabled.
     (Test 3 proves this conclusively).
================================================================================
         Main Menu
```

```
1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > 2
================================================================================
*** Disabling U128E = 0b ***
AltAccess=DISABLED   U128E=DISABLED   L128LOCK=DISABLED
U128LOCK=DISABLED
                                                    *** Reads ***
                                                LB Index    LB Target
                                                0x70=>0xFF  0x71=>0x77
                                                0x74=>0x66  0x75=>0x77
                                                0x72=>0x66  0x73=>0x77
                                                0x76=>0x66  0x77=>0x77

*** Write 0x70<=0x9C   Write 0x71<=0x35 ***
                                                    *** Reads ***
                                                LB Index    LB Target
                                                0x70=>0xFF  0x71=>0x35
                                                0x74=>0x1C  0x75=>0x35
                                                0x72=>0x1C  0x73=>0x35
                                                0x76=>0x1C  0x77=>0x35

Lessons Learned:
  1) When U128E disabled, all I/O ranges alias to the I/O 0x70 and I/O
0x71.
  2) I/O 0x70 is NOT readable when Alt Access is disabled.
  3) I/O 0x70/72/74/76[7] is WRITEABLE, but not READABLE when Alt-Acc
Disabled.
     (Test 3 proves this conclusively).
================================================================================


         Main Menu
1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > 3
================================================================================
AltAccess=DISABLED   U128E=DISABLED   L128LOCK=DISABLED
U128LOCK=DISABLED
*** Write 0x74<=0x97   Write 0x71<=0x12 ***
                                                    *** Reads ***
                                                LB Index    LB Target
                                                0x70=>0xFF  0x71=>0x12
                                                0x74=>0x17  0x75=>0x12
                                                0x72=>0x17  0x73=>0x12
```

```
                                                0x76=>0x17  0x77=>0x12

*** Enabling Alt Access Mode ***
AltAccess=ENABLED    U128E=DISABLED   L128LOCK=DISABLED
U128LOCK=DISABLED
                                                    *** Reads ***
                                                LB Index    LB Target
                                                0x70=>0x97  0x71=>0x12
                                                0x74=>0x17  0x75=>0x12
                                                0x72=>0x17  0x73=>0x12
                                                0x76=>0x17  0x77=>0x12

Lessons Learned:
  1)  ALT-ACCESS permits READS to I/O 0x70[7:0].  Even 0x70[7] is
readable.
  2)  STILL can't read I/O 0x72/74/76[7]
  3)  Note:  Writes to I/O 0x70/72/74/76[7] DO work BEFORE Alt-Acc
================================================================================
=======
         Main Menu
1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > 4
================================================================================
=======
AltAccess=DISABLED   U128E=ENABLED    L128LOCK=DISABLED
U128LOCK=DISABLED
*** Write 0x72<=0x38  Write 0x73<=0x11
                                                *** Reads ***
                                    LB Index    LB Target    UB Index    UB
Target
                                    0x70=>0xFF  0x71=>0x55   0x72=>0x38
0x73=>0x11
                                    0x74=>0x44  0x75=>0x55   0x76=>0x38
0x77=>0x11

*** Enabling U128LOCK = 1b ***
AltAccess=DISABLED   U128E=ENABLED    L128LOCK=DISABLED
U128LOCK=ENABLED
*** Write 0x72<=0x38  Write 0x73<=0xEE ***
                                                *** Reads ***
                                    LB Index    LB Target    UB Index    UB
Target
                                    0x70=>0xFF  0x71=>0x55   0x72=>0x38
0x73=>0x00
                                    0x74=>0x44  0x75=>0x55   0x76=>0x38
0x77=>0x00

Lessons Learned:
  1) U128 Lock works for ranges 0x38-0x3F.
```

```
  2) Both Reads and Writes locked to I/O 0x38-0x3F.  Reads return
garbage.
  3) Once locked, region cannot be unlocked.  Reboot to unlock.
================================================================================
=======


         Main Menu
1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > 5
================================================================================
=======
AltAccess=DISABLED   U128E=DISABLED   L128LOCK=DISABLED
U128LOCK=ENABLED
*** Write 0x70<=0x3A  Write 0x71<=0x11 ***
                                                  *** Reads ***
                                               LB Index    LB Target
                                               0x70=>0xFF  0x71=>0x11
                                               0x74=>0x3A  0x75=>0x11
                                               0x72=>0x3A  0x73=>0x11
                                               0x76=>0x3A  0x77=>0x11
*** Enabling L128LOCK = 1b ***
AltAccess=DISABLED   U128E=DISABLED   L128LOCK=ENABLED
U128LOCK=ENABLED
*** Write 0x70<=0x3A  Write 0x71<=0xEE ***
                                                  *** Reads ***
                                               LB Index    LB Target
                                               0x70=>0xFF  0x71=>0x00
                                               0x74=>0x3A  0x75=>0x00
                                               0x72=>0x3A  0x73=>0x00
                                               0x76=>0x3A  0x77=>0x00
Lessons Learned:
  1) L128 Lock works for ranges 0x38-0x3F.
  2) Both Reads and Writes locked to I/O 0x38-0x3F.  Reads return
garbage.
  3) Once locked, region cannot be unlocked.  Reboot to unlock.
================================================================================
=======
         Main Menu
1. U128 (E)nable Test
2. U128 (D)isable Test
3. (A)lt Access Test
4. (U)pper 128 Byte Lock Test
5. (L)ower 128 Byte Lock Test
X. E(x)it
Option# > x
================================================================================
=======
```

Conclusion

Although successfully reading and writing to the RTC registers located in 0x70-0x77 can be difficult (especially the NMI Enable Bit at I/O location 0x70[7]), it can be done if the programmer follows a few key steps:

1. **How I/O locations 0x70-0x77 get aliased depends on how the 128E enable bit is set** (see Table 3and Table 4 above). Most programmers just use I/O pair 0x74/5 to read from the standard bank (works when U128E is 0b or 1b), and 0x72/3 to read from the extended bank (when U128E is 1b). This works fine for all bits except 0x70[7].

2. **In order to read 0x70[7], the chipset must be in Alt Access Mode**. When Alt Access IS enabled, 0x70[7] can now be read normally, although the aliases of 0x70[7] still always return 0b. Thus, software must read the NMI Enable bit from 0x70[7], not one of the aliases.

3. **0x70[7] and 0x74[7], etc. are writeable at all times** - even when Alt-Access Mode is disabled. This is the source of tons of software grief. If a programmer isn't careful to read/mask/write bit 7 when accessing 0x70 and the aliases, the NMI bit can be set or cleared inadvertently.

Authors

Sam Fleming is a Technical Marketing Engineer with the Applications Design-In Center at Intel in Folsom, California.

Acronyms

ICH#: This term refers to the piece of Intel silicon that typically contains functionality NOT associated with the CPU and/or memory controller. As Intel continues to further integrate its chipsets, this definition is getting fuzzier and fuzzier.

For the sake of this document, the ICH# refers the Intel silicon that contains the legacy RTC functionality (the registers located at I/O locations 0x70-0x77). This would include the full line of ICH products (ICH, ICH2, ICH4, etc), the 6300ESB, ESB2, and the EP0579 and Intel® System Controller Hub US15W Chipset components.

South Bridge: Same as the ICH#

RTC: Real Time Clock