1.0 Scope
This document discusses accessing virtual registers in a system based on the AMD Geode™ GX and LX processors, and the AMD Geode™ CS5535 and CS5536 companion devices (hereafter referred to as Geode processor and Geode companion device).

2.0 Introduction
Virtual registers are a register set emulated by Virtual System Architecture™ (VSA) technology. The purpose of virtual registers is to provide a uniform communications interface between disparate software modules. Typical usage examples are:

- Customization of VSA software parameters and features by XpressROM
- Event notification between Virtual Support Modules (VSMs)
- Provide access to resources that are not normally accessible from Ring-3 code

Virtual registers are accessed as 16-bit index/data pairs. The 16-bit index is partitioned into two 8-bit fields. The eight MSBs of a virtual register index specify a functional class. The eight LSBs of the index identify a specific register within the class. Once a register has been indexed, a read of the register is performed with "IN AX, DX", and a write is performed with "OUT DX, AX".

A virtual register class may be implemented by either the system manager or one or more VSMs. Typically, a class is identified with a single VSM since both virtual register classes and VSMs usually are defined across similar functional lines. However, some classes (e.g., power management) are broadly defined and multiple VSMs may be registered for the same class.

3.0 Virtual Register Access
Accessing a virtual register is a three phase process:

- Unlock phase
- Index phase
- Data phase

A GLIU descriptor is used to generate an SMI upon access to the virtual register I/O range. In the following examples, the default virtual register I/O range of 0AC1Ch-0AC1Fh is used. The actual range is defined by the Northbridge F0 BAR0 value.

3.1 Unlock Phase
Virtual registers must be unlocked before they may be accessed. This feature is to prevent an operating system from erroneously detecting a nonexistent device when performing a hardware scan, or worse, causing side effects by writing to a virtual register. To unlock a virtual register, the following code is executed:

```
MOV DX, VRC_INDEX ; DX = 0AC1Ch
MOV AX, VRC_UNLOCK ; 0FC53h (Unlock code)
OUT DX, AX
```

If virtual registers are unlocked, but followed by an illegal access (e.g., undefined Class, Byte access, or misaligned access), the registers revert to the Locked state. Once a virtual register access is completed, the registers return to the Locked state.

**Note:** When the virtual registers are in the Locked state, DWORD accesses to the Index register are recognized. If the 16 MSBs of EAX equal VR_UNLOCK (FC53h), then AX is interpreted as the Class/Index value. That is, the Unlock and Index phases may be performed with either one DWORD access or two WORD accesses.
3.2 Index Phase
After virtual registers have been unlocked, the Class/Index is specified as follows:

```
MOV DX, VRC_INDEX         ; DX = 0AC1Ch
MOV AH, VRC_MISCELLANEOUS ; AH = 00h (Register Class)
MOV AL, MSR_ACCESS       ; AL = 07h (Register Index within the Class)
OUT    DX, AX
```

3.3 Data Phase
After the index phase, the virtual register may be either read or written. The effect of a read or a write is determined entirely by the register definition. Typically, a read retrieves information and a write provides information. However, an access may invoke defined side effects (e.g., MSR or memory access). There are no restrictions on what functionality the access causes.

When the data register is read or written, the system manager determines which VSM(s) have registered for the accessed register. An EVENT_VIRTUAL_REGISTER message is inserted into the VSMs’ message queues and the VSM(s) are scheduled for execution. The message parameters specify the Class/Index, and if an I/O write, the written data value. There are no restrictions on how the VSM interprets the access.

**Note:** Only WORD accesses are allowed to the Data register. BYTE or DWORD accesses are illegal.

4.0 Virtual Register Classes
This section describes each virtual register class and the registers defined within each class. Note that a VSM may create a new class of virtual registers as long as it does not interfere with existing classes. The classes described below are the currently defined classes.

**Note:** When sample code is shown, it is assumed that the register has been unlocked and indexed, and that General Purpose register DX contains the value 0AC1Eh.

4.1 VRC_MISCELLANEOUS (00h)
The system manager implements this class. It provides functionality that is not related to any particular hardware device or VSM.

4.1.1 VSA_VERSION_NUM (00h)
Reading this register returns the system manager’s version in AX. Writes are ignored.

4.1.2 HIGH_MEM_ACCESS (01h)
This register provides read/write access to any 32-bit address, be it physical memory or a memory-mapped location. The address is passed in EBX.

To read a memory location:

```
MOV EBX, <address> ; Physical address to read
IN    AX, DX
```

On return, EAX contains the value read from <address>.

To modify a memory location:

```
MOV EBX, <address> ; Physical address to modify
MOV ESI, 0000FFFFh ; AND mask
MOV EDI, 01230000h ; OR mask
OUT    DX, AX
```

The DWORD at <address> is read, ANDed with ESI, ORed with EDI, then rewritten to <address>.
4.1.3 GET_VSM_INFO (02h)
This register is reserved for use by the INFO utility.

4.1.4 SIGNATURE (03h)
This register returns the VSA2 technology signature ('VSA2') in EAX.

4.1.5 VSM_VERSION (05h)
This register is reserved for use by the INFO utility.

4.1.6 CTRL_ALT_DEL (06h)
This register should be written by the BIOS just before performing a warm boot. A MSG_WARM_BOOT is broadcast to each VSM. A VSM should respond to this message by putting its hardware into an inactive state.

4.1.7 MSR_ACCESS (07h)
There are occasions when it is necessary for software to read or write an MSR. This presents a problem for Ring-3 code since the RDMSR and WRMSR instructions are privileged instructions. To facilitate resolving this dilemma, a virtual register is defined to provide MSR access. Note, however, in some protected-mode operating systems, I/O may also be restricted. In such cases, a device driver or other mechanism is necessary to enable the virtual register access to proceed.

Note: No error checking is performed on the MSR address used with this feature. Use of an invalid MSR address causes a Trap 0Dh exception within VSA software, resulting in a system hang. Invalid MSRs in the Southbridge return invalid data (reads) and are ignored on writes. Hardware fixes are applied if appropriate.

The value of an MSR is returned by reading from this register. The following code demonstrates the method:

```
MOV ECX, <MSR_Address>
IN AX, DX
; On Return: EDX:EAX = MSR contents. EFLAGS is unchanged except the Carry Flag is toggled.
```

An MSR may be modified by writing to this virtual register. Register ECX contains the MSR address to write. ESI:EDI contains 64 bits of AND mask and EBX:EAX contains 64 bits of OR mask. The requested MSR is read, the AND mask is applied first, followed by the OR mask. The MSR is then rewritten. The following code demonstrates the method:

```
MOV ECX, <MSR_Address>
MOV ESI, <32-MSBs of AND mask>
MOV EDI, <32-LSBs of AND mask>
MOV EBX, <32-MSBs of OR mask>
MOV EAX, <32-LSBs of OR mask>
OUT DX, AX
```

4.1.8 GET_DESCR_INFO (08h)
This register is reserved for use by the INFO utility.

4.1.8.1 PCI_INT_AB (09h) and PCI_INT_AB(0Ah)
This register may be used by the BIOS to change the PCI interrupt pin assignments. By default, the PCI interrupts are assigned to GPIO pins as follows:

INTA#: GPIO0
INTB#: GPIO7
INTC#: GPIO12
INTD#: GPIO13

The GPIOs assigned to PCI interrupts are configured as level-sensitive, inverted inputs. If a platform uses a different configuration, the BIOS must notify VSA software of the new pin assignments.

Note: Changing GPIO pin assignments must be done before the Interrupt Steering register is written. They may not be changed more than once. Subsequent attempts to change them will be ignored.
To change the GPIOs assigned to INTA# and INTB#, virtual register PCI_INT_AB should be unlocked and indexed, then the following code may be executed:

```
MOV DX, VRC_DATA
MOV AL, 08h ; Use GPIO pin 8 for INTA#
MOV AH, 09h ; Use GPIO pin 9 for INTB#
OUT DX, AX
```

Similarly, to change INTC# and or INTD#, virtual register PCI_INT_CD should be unlocked and indexed.

If a platform does not require four PCI interrupts, the BIOS may reduce the number of PCI interrupts supported. This is done by assigning a pin # = 0FFh. If a virtualized PCI header uses an interrupt pin for which a GPIO pin is no longer assigned, that interrupt pin register is reduced to the next lower valid value. For example, if support for INTC# and INTD# are removed (by writing 0FFFFh to virtual register PCI_INT_CD), then all virtualized PCI headers that previously defined either INTC# or INTD# are changed to INTB#.

### 4.2 VRC_VG (02h)

This class is implemented by the graphics software VSM. These registers are written by the BIOS to convey video parameters to the graphics software. Refer to the [AMD Geode™ GX Processors Graphics Software Specification](PID# 31923) or the [AMD Geode™ LX Processors Graphics Software Specification](PID# 32478) for details.

### 4.3 VRC_APM (03h)

This class supports the APM interface. The APM VSM implements a generic APM interface. APM functionality that is platform dependent is handled via registers in the class.

#### 4.3.1 REPORT_EVENT (00h)

This register is used by the BIOS or a power management VSM (i.e., PMCore or battery VSM) to report events to the APM driver. The value written is the APM event value as defined by the APM Specification Revision 1.2. If read, the value returned is a mask of pending events.

#### 4.3.2 CAP ABILITIES (01h)

This register is used by the system BIOS or the APM BIOS to report capabilities to the APM driver. The value written is the APM capabilities mask as defined by the APM Specification Revision 1.2. Whenever this register is written, a Capabilities Change Notification event is posted. When read, the current capabilities mask is returned.

### 4.4 VRC_PM (04h)

This class is implemented to facilitate the implementation of power management. Most of the registers are related to legacy (timer-based) power management.

#### 4.4.1 POWER_MODE (00h)

This register specifies the current mode of power management: Disabled, legacy, APM, or ACPI.

Valid values (found in VSA_II\INC\APM.H) are:

- PM_DISABLED 00h
- PM_LEGACY 01h
- PM_APM 02h
- PM_ACPI 03h

#### 4.4.2 POWER_STATE (01h)

This register sets the system Power State (S0-S5).
4.4.3 **DOZE_TIMEOUT** (02h)
This register specifies the Legacy Doze timeout interval (in seconds). If there is no keyboard/mouse activity for the defined interval, the system begins clock throttling the CPU (C1). The CPU automatically returns to C0 upon the next keyboard/mouse interrupt. This power state is transparent to the user.

4.4.4 **STANDBY_TIMEOUT** (03h)
This register specifies the Standby timeout interval (in seconds). If there is no system activity for the defined interval, the system enters Standby (S1). System activity is defined as keyboard/mouse activity. If a non-zero VIDEO_ACTIVITY register value is specified, then the video controller must also be inactive for the system to be considered inactive.

4.4.5 **SUSPEND_TIMEOUT** (04h)
This register specifies the Suspend timeout interval (in seconds). If there is no system activity for the defined interval, the system enters Suspend. System activity is defined the same as for STANDBY_TIMEOUT. Suspend is defined as S2 if a Save-to-RAM VSM is not present, or S3 if a Save-to-RAM VSM is present.

4.4.6 **VIDEO_TIMEOUT** (07h)
This register specifies the interval (in seconds) that the video subsystem must be inactive before the system is considered inactive for entering Standby or Suspend. If this register is zero, then the video subsystem is not required to be inactive for the system to transition to a lower power state.

4.4.7 **DISK_TIMEOUT** (08h)
This register specifies the timeout interval (in seconds) for the ATA disk drive(s). If no ATA drive commands are issued for the defined interval, the disk drive(s) are commanded to spin down.

4.4.8 **FLOPPY_TIMEOUT** (09h)
This register specifies the timeout interval (in seconds) for the floppy drive(s). If no floppy commands are issued for the defined interval, a MSG_SET_POWER_STATE is broadcast to every VSM. The parameters of the message define the class/unit of device (DEVICE_STORAGE) and the D-state (normally D1_STATE) to which the device should be transitioned. Typically, the SuperI/O VSM puts the inactive floppy device into a lower power state.

4.4.9 **SERIAL_TIMEOUT** (0Ah)
This register specifies the timeout interval (in seconds) for the serial port(s) described by the BIOS Data Area (BDA) at 0040:0000-0007. If there is no I/O activity to the serial port for the defined interval, a MSG_SET_POWER_STATE is broadcast to every VSM. The parameters of the message define the class/unit of device (DEVICE_SERIAL) and the D-state (normally D1_STATE) to which the device should be transitioned. Typically, the SuperI/O VSM puts the inactive serial device into a lower power state.

4.4.10 **PARALLEL_TIMEOUT** (0Bh)
This register specifies the timeout interval (in seconds) for the parallel port(s) described by the BDA at 0040:0008-000F. If there is no I/O activity to the parallel port for the defined interval, a MSG_SET_POWER_STATE is broadcast to every VSM. The parameters of the message define the class/unit of device (DEVICE_PARALLEL) and the D-state (normally D1_STATE) to which the device should be transitioned. Typically, the SuperI/O VSM puts the inactive parallel device into a lower power state.

4.4.11 **IRQ_WAKEUP_MASK** (0Ch)
This register specifies the IRQ(s) that are to be enabled during S1 or S2. Activity on an enabled IRQ returns the system to S0. The register value is a mask. Bit 1 corresponds to IRQ1…Bit 15 to IRQ15. To make an IRQ a potential wakeup IRQ, set the corresponding bit to one. IRQ0 may not be specified as a wakeup IRQ.

4.4.12 **SUSPEND_MODULATION** (0Dh)
This register specifies the C1 clock throttling value (as a percent). If the C1 State is entered, this value specifies the percentage of time the CPU is to be clocked versus suspended.
4.4.13 SLEEP_PIN (0Eh)
This register is written by the system BIOS to specify the GPIO pin and PME number to be used for the Sleep button. If not defined, a Sleep button is not implemented. The eight LSBs specify the GPIO pin and the eight MSBs specify the PME.

4.4.14 SLEEP_PIN_ATTR (0Fh)
This register is written by the system BIOS to specify the attributes to be used for the Sleep button GPIO. Attributes may be one or more of the following:

- FALLING_EDGE
- RISING_EDGE
- DEBOUNCE
- PULLUP
- PULLDOWN
- INVERT

4.4.15 PM_S1_CLOCKS (12h) - PM_S5_CLOCKS (16h)
These registers are written by the system BIOS to specify how to control the external clock chip upon entering S1 - S5. Valid values are:

- Sx_CLOCKS_ON
- Sx_CLOCKS_OFF

4.4.16 PM_S0_LED (17h) - PM_S5_LED (1Ch)
These registers are written by the system BIOS to specify the period and duty cycle of the Standby LED during S0 - S5. The eight LSBs of the virtual register value are the duty cycle (units = percent). The eight MSBs of the value are the period (units = 100 ms).

4.5 VRC_ACPI (08h)
Most of the registers in this class are reserved for communication between the ACPI Source Language (ASL) code and ACPI VSM. The registers written by the system BIOS are in the following subsections.

4.5.1 SCI_IRQ (01h)
This register specifies which IRQ is used for the ACPI SCI.

4.5.2 ACPINVS_LO (02h) and ACPINVS_HI (03h)
These two registers specify the physical address of the FACS table.

4.5.3 RW_PIRQ (06h)
RW_PIRQ reflects the current value set in the hardware. This is used by the ASL functions: _SRS (Set Resource Setting) and _CRS (Current Resource Setting).

4.5.4 PIRQ_ROUTING (08h)
PIRQ_ROUTING holds the possible INTA - INTD routings as defined by the BIOS. This is used by the ACPI ASL code to initialize the _PRS (Possible Resource Settings) for INTA through INTD. The current implementation only allows one IRQ setting to be specified for each PCI interrupt. The BIOS writes this register with the routing definitions before the end of POST.
4.6 VRC_POWER (0Ah)
This class is to be implemented by a battery controller VSM. The APM VSM accesses these registers for reporting the power status of the system.

4.6.1 BATTERY_UNITS (00h)
This register is read to detect the number of battery units present.

4.6.2 BATTERY_SELECT (01h)
This register is written to select the desired battery for querying its status (see Section 4.6.3 "AC_STATUS (02h)" and Section 4.6.4 on page 7). The battery number is 1-based per the Get Power Status function defined in the APM specification.

4.6.3 AC_STATUS (02h)
This register returns the AC line status:
00h = Off-line
01h = On-line
02h = On backup power
FFh = Unknown

4.6.4 BATTERY_STATUS (03h)
This register is read to determine the status of the selected battery:
00h = High
01h = Low
02h = Critical
03h = Charging
FFh = Unknown

4.6.5 BATTERY_FLAG (04h)
This register reports the status of a given battery:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 = High</td>
</tr>
<tr>
<td>1</td>
<td>1 = Low</td>
</tr>
<tr>
<td>2</td>
<td>1 = Critical</td>
</tr>
<tr>
<td>3</td>
<td>1 = Charging</td>
</tr>
<tr>
<td>4</td>
<td>1 = Selected battery not present</td>
</tr>
<tr>
<td>7</td>
<td>1 = No system battery</td>
</tr>
</tbody>
</table>

A value of FFh represents unknown status.

4.6.6 BATTERY_PERCENTAGE (05h)
This register returns the remaining life of the selected battery (in percent of full charge). A value of FFh represents unknown.

4.6.7 BATTERY_TIME (06h)
This register returns the remaining life of the selected battery (in time). Bit 15 = 0 means time units are in seconds. Bit 15 = 1 means time units are in minutes. Bits [14:0] are the number of seconds or minutes. A value of FFFFh represents unknown.
4.7 VRC_OHCI (0Bh)
The Open Host Controller Interface (OHCI) VSM implements the registers in this class.

4.7.1 SET_LED (00h)
This register is used by the 8042 VSM to tell the OHCI VSM how to set the LEDs on the USB keyboard(s).

4.7.2 INIT_OHCI (01h)
Normally, a VSM initializes when VSA software is first installed. However, some BIOSes may not have initialized the keyboard controller at that stage of POST. If Legacy keyboard emulation is enabled when such a BIOS initializes the keyboard controller, the emulation may interfere with the initialization. Therefore, the OHCI VSM does not perform its initialization (and therefore enable legacy emulation) until the BIOS signals that it is ready. This register must be written by the BIOS after the keyboard controller has been initialized. Writing this register signals the legacy USB VSMs that it is OK to perform USB initialization.

4.7.3 VRC_KEYBOARD (0Ch)
The registers in this class are reserved for communication between the OHCI and 8042 VSMs.

4.8 VRC_STR (0Fh)
The Save-to-RAM VSM implements the register in this class.

4.8.1 RESTORE_ADDR (00h)
This register is read by the BIOS (after VSA software early initialization and before end of POST) to determine the physical address of the MSR restore table. VSA software returns a 32-bit flat pointer in EAX. Normally the BIOS saves the value returned in the EBDA. When waking from S3, the BIOS must restore the MSRs specified in the table before performing the SYS_RESUME_FROM_RAM software SMI. The table is terminated by a MsrAddr field of zero. The structure of the table is:

```c
typedef struct {
    unsigned long MsrAddr;
    unsigned long MsrData[2]; // LSBs first, then 32 MSBs
} BIOS_MSRS;
```
5.0  Virtual Register Definitions

Note:  The following is from \SYSMGR\INC\VR.H

#define VRC_INDEX 0xAC1C // Index register
#define VRC_DATA 0xAC1E // Data register
#define VR_UNLOCK 0xFC53 // Virtual register unlock code

#define VRC_MISCELLANEOUS 0x00 // Miscellaneous Class
  #define VSA_VERSION_NUM 0x00
  #define HIGH_MEM_ACCESS 0x01
  #define GET_VSM_INFO 0x02
  #define SIGNATURE 0x03
  #define VSM_VERSION 0x05
  #define CTRL_ALT_DEL 0x06
  #define MSR_ACCESS 0x07
  #define PCI_INT_AB 0x09
  #define PCI_INT_CD 0x0A

#define VRC_VG 0x02 // SoftVG Class
#define VRC_APM 0x03 // APM Class
  #define REPORT_EVENT 0x00
  #define CAPABILITIES 0x01
  #define APM_PRESENT 0x02

#define VRC_PM 0x04 // Legacy PM Class
  #define POWER_MODE 0x00
  #define POWER_STATE 0x01
  #define DOZE_TIMEOUT 0x02
  #define STANDBY_TIMEOUT 0x03
  #define SUSPEND_TIMEOUT 0x04
  #define PS2_TIMEOUT 0x05
  #define RESUME_ON_RING 0x06
  #define VIDEO_TIMEOUT 0x07
  #define DISK_TIMEOUT 0x08
  #define FLOPPY_TIMEOUT 0x09
  #define SERIAL_TIMEOUT 0x0A
  #define PARALLEL_TIMEOUT 0x0B
  #define IRQ_WAKEUP_MASK 0x0C
  #define SUSPEND_MODULATION 0x0D
  #define SLEEP_PIN 0x0E
  #define SLEEP_PIN_ATTR 0x0F
  #define SMI_WAKEUP_MASK 0x10
  #define PM_S1_CLOCKS 0x12
  #define PM_S2_CLOCKS 0x13
  #define PM_S3_CLOCKS 0x14
  #define PM_S4_CLOCKS 0x15
  #define PM_S5_CLOCKS 0x16
  #define PM_S0_LED 0x17
  #define PM_S1_LED 0x18
  #define PM_S2_LED 0x19
  #define PM_S3_LED 0x1A
  #define PM_S4_LED 0x1B
  #define PM_S5_LED 0x1C
#define VRC_ACPI 0x08 // ACPI Class
  #define ENABLE_ACPI 0x00
  #define SCI_IRQ 0x01
  #define ACPIUNUSED LO 0x02
  #define ACPIUNUSED HI 0x03
  #define GLOBAL LOCK 0x04
  #define ACPIUNUSED1 0x05
  #define RW_IRQ 0x06
  #define SLPB_CLEAR 0x07
  #define PIQ_ROUTING 0x08
  #define ACPIUNUSED2 0x09
  #define ACPIUNUSED3 0x0A
  #define PIC_INTERRUPT 0x0B
  #define ACPI_PRESENT 0x0C
  #define ACPIUNUSED COMMAND 0x0D
  #define ACPIUNUSED PARAM1 0x0E
  #define ACPIUNUSED PARAM2 0x0F
  #define ACPIUNUSED PARAM3 0x10
  #define ACPIUNUSED RETVAL 0x11

#define VRC_POWER 0x0A // Battery Class
  #define BATTERY_UNITS 0x00
  #define BATTERY_SELECT 0x01
  #define AC_STATUS 0x02
  #define BATTERYSTATUS 0x03
  #define BATTERY_FLAG 0x04
  #define BATTERY_PERCENTAGE 0x05
  #define BATTERY_TIME 0x06

#define VRC_OHCI 0x0B // OHCI Class
  #define SET_LED 0x00
  #define INIT_OHCI 0x01

#define VRC_KEYBOARD 0x0C // Keyboard Controller Class
  #define KEYBOARD_PRESENT 0x00
  #define SCANCODE 0x01
  #define MOUSE_PRESENT 0x02
  #define MOUSE_BUTTONS 0x03
  #define MOUSE_XY 0x04

#define VRC_DDC 0x0D // Video DDC Class
  #define VRC_DDC_ENABLE 0x00
  #define VRC_DDC IO 0x01

#define VRC_STR 0x0F // Save-to-RAM Class
  #define RESTORE_ADDR 0x00

#define VRC_SUPERIO 0x13 // SuperI/O Class
  #define VRC_SIO_CHIPID 0x00
  #define VRC_SIO_NUMLD 0x01
  #define VRC_SIO_FDCEN 0x02
  #define VRC_SIO_FDCIO 0x03
  #define VRC_SIO_FDCIRQ 0x04
  #define VRC_SIO_FDCDMA 0x05
  #define VRC_SIO_FDCFG1 0x06
  #define VRC_SIO_FDCFG2 0x07
  #define VRC_SIO_PP1EN 0x08
  #define VRC_SIO_PP1IO 0x09
  #define VRC_SIO_PP1IRQ 0x0A
  #define VRC_SIO_PP1DMA 0x0B


#define VRC_SIO_PP1CFG1 0x0C
#define VRC_SIO_SP1EN 0x0D
#define VRC_SIO_SP1IO 0x0E
#define VRC_SIO_SP1IRQ 0x0F
#define VRC_SIO_SP1CFG1 0x10
#define VRC_SIO_SP2EN 0x11
#define VRC_SIO_SP2IO 0x12
#define VRC_SIO_SP2IRQ 0x13
#define VRC_SIO_SP2CFG1 0x14
#define VRC_SIO_KBEN 0x15
#define VRC_SIO_KBIO1 0x16
#define VRC_SIO_KBIO2 0x17
#define VRC_SIO_KBIRQ 0x18
#define VRC_SIO_KBCFG1 0x19
#define VRC_SIO_MSEN 0x1A
#define VRC_SIO_MSIO 0x1B
#define VRC_SIO_MSIRQ 0x1C
#define VRC_SIO_RTCEN 0x1D
#define VRC_SIO_RTCIO1 0x1E
#define VRC_SIO_RTCIO2 0x1F
#define VRC_SIO_RTCIRQ 0x20
#define VRC_SIO_RTCCFG1 0x21
#define VRC_SIO_RTCCFG2 0x22
#define VRC_SIO_RTCCFG3 0x23
#define VRC_SIO_RTCCFG4 0x24
#define VRC_CHIPSET 0x14 // Chipset Class
#define VRC_CS_PWRBTN 0x00
#define VRC_CS_UART1 0x01
#define VRC_CS_UART2 0x02