Synaptics RMI Interfacing Guide

PN: 511-000099-01 Rev. E

Copyright

May 19, 2008.

Trademarks

Synaptics, the Synaptics logo, Synaptics OneTouch, ClearPad, EdgeMotion, LightTouch, LuxPad, MobileTouch, NavPoint, PalmCheck, RoundPad, ScrollStrip, TouchPad, and TouchStyk are trademarks of Synaptics Incorporated.

All other brand names or trademarks are the property of their respective owners.

Notice

Information contained in this publication is provided as-is, with no express or implied warranties, including any warranty of merchantability, fitness for any particular purpose, or non-infringement. Synaptics Incorporated assumes no liability whatsoever for any use of the information contained herein, including any liability for intellectual property infringement. This publication conveys no express or implied licenses to any intellectual property rights belonging to Synaptics or any other party. Synaptics may, from time to time and at its sole option, update the information contained herein without notice.

Conventions used in this document

The table below describes the documentation conventions. These conventions are used in all Synaptics technical literature.

Term	Meaning
$	Hexadecimal numbers are marked with a leading '$' sign: The number $7FF is equal to 2047 decimal.
—	Bits shown as "— " in register diagrams are equivalent to bits marked *Reserved.*
italics	*Italicized* words introduce a term described in the adjacent text or in the Glossary.
Reserved	*Reserved* is used to signify a bit or bit-field not currently used in any (published) way.
`Courier`	Courier font is used for text to be entered on a command line or in a program, or for text output from a device.

Contents

1. Introduction

This document defines a register-oriented protocol for use in Synaptics® embedded products. The overall protocol is known as RMI: the Register Mapped Interface. RMI uses a "register map" model that is convenient and familiar to host system developers.

The basic goals of RMI are:

1. To support a large and varied product line, with an emphasis on forward-, backward-, and cross-compatibility and consistency among Synaptics products;

2. To employ industry-standard I^2C, SMBus, and SPI-based interfaces, following the familiar and easy-to-use "register" model that is commonly found in devices with these interfaces; and

3. To be easy to document, easy to understand, and easy to use from the perspective of implementers of RMI drivers and systems incorporating RMI devices. RMI is designed so that any given RMI product can be documented concisely. For example, the numbering of functions and data sources is elaborate when considering the RMI protocol as a whole, but in each specific RMI device the resulting register map is straightforward and easy to use.

Each RMI product uses a particular physical interface (I^2C, SPI, or SMBus) to access a particular register set tailored to the product. But RMI itself is a platform protocol that ties together the common aspects of all physical interfaces and all register maps of Synaptics' various embedded products.

1.1. Conventions used in RMI documentation

Bits within a byte, register, or other quantity are numbered with bit 0 as the least significant bit of the register or quantity. Ranges of bits are denoted *n*:*m* for the field of bits numbered *n* down to *m*, inclusive. For example, bits 7:4 comprise the most significant four bits of an 8-bit register.

All signed quantities in RMI are expressed in two's complement binary notation, where the most significant bit is taken as a sign bit. For example, a signed 8-bit byte is $00 to encode 0, $7F to encode +127, $80 to encode –128, and $FF to encode –1. A signed 6-bit register field would be $00 to encode 0, $1F to encode +31 (the largest value that can be encoded in a signed 6-bit field), $20 to encode –32 (the smallest value that can be encoded), and $3F to encode –1.

For a tutorial about a particular device, or information about initiating communication, please see the relevant Quickstart guide.

Note: Not all features described in this RMI Interfacing Guide are supported by every device. Consult the Product Specification or other device-specific documentation to find out which options are supported by a particular device.

2. The structure of RMI

RMI, the Register Mapped Interface, is a communications interface for use with Synaptics modules. RMI is built upon industry-standard physical interfaces. Currently, RMI offers a choice of SPI, I^2C, or SMBus. RMI communications involve two entities: The *host*, typically the main system processor, is the master. The *device*, typically a chip or module supplied by Synaptics, is a slave.

For systems with multiple hosts or multiple devices, RMI relies on the arbitration and addressing mechanisms of the underlying physical interface. For example:

- In SPI-based RMI systems with multiple devices, the host might generate a separate SSB signal for each device.
- In I^2C or SMBus-based RMI systems with multiple hosts, the hosts might use bus arbitration and repeated-Start transactions to negotiate safe shared access to a device.

2.1. Registers

RMI is defined in terms of logical *registers*. The host communicates with the device by reading and writing the device's registers through physical interface transactions.

All registers in RMI are 8 bits wide. Quantities larger than a byte are held in several consecutive registers which are typically read or written as a group.

Certain multi-byte quantities may *require* that the host must write them as a group. For such quantities, writing to any byte but the last simply stores the data written in a holding area, and writing to the final (highest-addressed) byte of the quantity stores the byte written, plus the held data, into the multi-byte quantity. This restriction applies only to a rare few multi-byte quantities in RMI; most registers can be written as independent bytes even if they are part of larger structures.

Similarly, certain multi-byte quantities may require that the host must read them as a group. For such quantities, reading from the first (lowest-addressed) byte reads the entire quantity into a holding area and reports the first byte of the held data. Reading any byte after the first reports the saved data from the holding area. This restriction applies only to a rare few multi-byte quantities in RMI; most registers can be read as independent bytes even if they are part of larger structures.

Note: Currently, RMI does not include any registers that must be read as a group in this sense. The data registers have their own more specialized rules for ensuring data coherency; see section 2.5.6.

Registers are identified by 15-bit addresses. Where '$' signifies hexadecimal notation, the register addresses range from $0000 to $7FFF. Each address in this range potentially identifies one byte of register data. Only a sparse few of the 32768 potential addresses are actually implemented; other addresses are marked *reserved* as defined in section 2.3.

The register address space is divided into *pages* of related registers. Each page consists of 256 registers in the range $*xx*00–$*xx*FF.

2.1.1. Register map

All RMI devices organize their registers according to the general scheme shown in Table 1.

Address range	*Purpose*	*See page*
$0000–$00FF	Standard RMI control, command, and status registers	19
$0000	Device Control register	19
$0001	Interrupt Enable control register	22
$0002	Error Status register	22
$0003	Interrupt Request status register	24
$0004	Device Command register	24
$0200–$02FF	Standard RMI query registers	25
$0200–$0207	General product information and version queries	26
$0208–$020D	Product serialization queries	27
$0210–$021F	Product ID queries	29
$0300–$03FF	Function Presence query registers	29
$0400–$04FF	Data registers	17
$0500–$07FF	*Reserved for product-specific alternative data pages*	—
$0800–$0BFF	*Reserved for definition in future versions of RMI*	—
$0C00–$0FFF	Manufacturer-defined registers	—
$10*xx*–$7F*xx*	Function-specific register pages	30

Table 1. RMI register map overview

Addresses $0000–$00FF and $0200–$03FF are used for registers whose definitions are universal among all RMI devices. The first page contains standard control, command, and status registers, which are described in section 2.6. The other two pages contain standard queries, which are described in section 2.7.

Addresses $0100–$01FF are reserved for use by each physical layer for any registers specific to the physical layer (if any).

Addresses $0400–$04FF are used for data registers. Section 2.5 describes the general conventions for data registers, and the documentation for each function describes the specific layouts of the relevant data registers.

Addresses $0800–$0BFF are generally reserved for assignment by future versions of the RMI standard.

Addresses $0C00–$0FFF are reserved for use by the device manufacturer for proprietary registers, diagnostic mechanisms, etc.

Addresses $1000–$7FFF are assigned in pages of 256 registers to various optional *functions* of RMI devices. The properties of RMI functions are described generally in section 2.2, and the various standard RMI functions are documented in section 3.

2.2. RMI functions

The features and capabilities of an RMI device arise from one or more functions that are included in the device. RMI defines a standard set of functions such as 2-D TouchPad™ sensors and general-purpose input/output (GPIO) pins. A particular RMI-based product will include or omit each possible function depending on the needs of the product. Each function is largely independent of any other functions that might be present in the same device.

Functions are consistently defined among all devices that include them. For example, Function $22 always corresponds to brightness-controlled LEDs, and the registers associated with Function $22 are defined in a consistent way in all RMI devices that include LEDs. For devices that lack the brightness-controlled LED function, the register addresses associated with Function $22 are unused.

Some RMI functions are completely universal, with a fixed definition that is identical in any product that includes them. Other RMI functions represent more general capabilities, with parameters that may be chosen differently from one product to another. For example, the 1-D Sensor function (RMI Function $14) may involve one linear strip sensor in one product, and two closed-loop sensors in another.

Section 3 documents all the RMI functions.

2.2.1. Function numbers

RMI functions are identified by an informal name and a standardized identifying number. The number for a function is a 7-bit integer in the range $10–$7F.

Function numbers $7C through $7F are reserved for definition by particular products. Functions $7C–$7F are not required to be, and typically will not be, consistent from one RMI product to another. Instead, products with a unique feature not likely to appear in other products can implement this feature as functions in the $7C–$7F range without permanently crowding the space of standard function numbers. Multi-purpose drivers and diagnostic tools should treat Functions $7C–$7F as unknown, unrecognized functions unless they recognize the specific Product ID.

Function numbers less than $7C are intended to be universally consistent among all RMI devices: If two different products both contain a function numbered $*xx*, this means that both contain the “same” function in some sense, and the behavior of function number $*xx* in both devices follows the same general specification. Synaptics Incorporated (contact: sales@synaptics.com) is responsible for maintaining a registry of RMI function number assignments.

The function number forms a base address for registers associated with the function. The Function Presence query register for Function $*xx* goes at address $03*xx*, and any other registers that are specific to Function $*xx* go in the address range $*xx*00–$*xx*FF. For example, the Function Presence query register for LEDs (Function $22) has address $0322, and the other registers related to LEDs have various addresses in the range $2200–$22FF.

Note: Some physical interfaces, such as the RMI implementation of SMBus, provide facilities for accessing common registers using 8-bit addresses (see, for example, section 4.2.1). The last address of each page, $*xx*FF, is specifically reserved for use by the physical layer. For example, RMI on SMBus uses this address for a Page Selection register.

2.3. RMI physical layer operations

This section summarizes the basic operations that are supported by every RMI physical layer. For specific details of the physical layers currently defined for RMI, see section 4.

2.3.1. Writing registers

The host may write to any *N* consecutive device registers starting at any register address *R*. The write transaction always "succeeds" as far as the RMI protocol is concerned. A physical layer may have limits on the number of consecutive registers that may be written in a single operation. See the section describing your selected physical layer in section 4 for more information.

A single register write operation may not span more than one register page. In other words, the *N* consecutive registers covered by a given write operation must have addresses that differ only in the low 8 bits. It is an error if the host performs a write operation that spans multiple pages; the effect is undefined and implementation-dependent.

As a special case, a write operation that sets bit 0 of register $0004 to '1' (the Reset command) must write only to register $0004 and only to set bit 0. Most physical layers define a holdoff time after the Reset command before the host can again reliably access the RMI interface. (Non-resetting writes to register $0004, where bit 0 is written as '0', are no more constrained than other ordinary RMI write operations.)

The end of the *N*-register write transaction must be an event distinguishable by the device, in order to support register semantics such as those described in section 3.5.4. (For example, for SPI this is the rise of the SSB signal.)

2.3.2. Reading registers

The host may read from any *N* consecutive device registers starting at any register address *R*. Each read transaction reports the *Device Status register* in addition to the requested bytes. The read transaction always "succeeds" as far as the RMI protocol is concerned.

A physical layer may have limits on the number of consecutive registers that may be read in a single operation. See the section describing your selected physical layer in section 4 for more information. A single register read operation may not span more than one register page (in the sense defined above in section 2.3.1).

The beginning and end of the *N*-register read transaction must be events distinguishable by the device, for reasons described in section 2.5.6. (For example, in SMBus these events are the Start and Stop conditions, and for SPI these are the fall and rise of the SSB signal.)

Preferably, the physical layer will allow the host to choose to adjust dynamically the number *N* of registers that it reads based on the first few data bytes it has read. However, RMI never *requires* the host to perform a read operation of non-fixed size, because some hosts lack this ability (for example, because of restrictions in their OS drivers).

2.3.3. Signaling attention and interrupts

The device may signal the host for attention. From the host's point of view, the attention signal acts like a traditional interrupt signal. The host responds to the attention interrupt by reading the appropriate device registers to determine the reason for the attention request. See Section 2.5.4 for more information about how interrupt requests are handled in RMI.

2.3.4. Spontaneous resets

Every physical layer should provide for robust system operation in the presence of spontaneous resets at any time by either the device or the host.

Spontaneous resets on the device side are a fact of life in many capacitive touch sensor designs. Touch sensors, by their nature, are more exposed than most electronics to disruption by ESD (electrostatic discharges) during operation, particularly if the sensor area is framed by an open bezel instead of being under solid, uninterrupted plastic. Synaptics chips are designed to ensure that any such disruption results in a clean reset of the chip. The RMI protocol is designed to allow the system to recover gracefully in the event of such a spontaneous device reset. Together, these designs allow robust touch sensing even in ESD-prone or otherwise glitchy environments.

Spontaneous resets on the host side are also a fact of life in embedded systems. If the host restarts suddenly, it may not have had time to shut down the peripherals in an orderly way. An RMI device should be prepared to accept a "reset" command (in the case of RMI, a write of $01 to register address $0004) no matter what was happening earlier, even if a previous RMI physical layer transaction was interrupted partway through, and even if a special command or mode was already in progress on the device.

Several properties of RMI are designed to work together to allow the host to detect reliably when the device has reset itself spontaneously:

- Every data source asserts an interrupt request after reset.
- Every implemented interrupt enable bit is '1' after reset (see Section 2.6.2).
- Therefore, the attention signal is always asserted after device reset.
- The attention signal will prompt the host to read the data registers as if new data had arrived.
- When the host reads the Device Status register, it will see a '0' in the Configured Flag. Because the host knows it has already configured the device, it can infer that the device has lost its configuration due to a spontaneous reset.

The host should respond to a spontaneous device reset by fully reinitializing the device. Depending on the nature of the overall system, the host may even wish to use a spontaneous reset in one RMI device as a cue to reinitialize other parts of the system.

Hosts that operate the device in its default configuration, without ever writing to any control registers, will not necessarily be able to detect a spontaneous reset because the Configured Flag will always be '0'. However, these hosts need not be cognizant of spontaneous resets; from their point of view, the resetting device will simply pause briefly in its reporting of sensor activity and then resume normal operation.

2.4. Kinds of RMI registers

Although in principle an RMI device can do anything in response to a read or a write of any register, RMI defines several standard types of registers with well-specified behaviors. These are described in the following sub-sections.

2.4.1. Control registers

Control registers allow the host to initialize the device and control its functions. A control register generally looks like a readable and writable RAM location: The host can write data to the register, and the host can read the current contents of the register back without side effects.

Control registers generally do not change except when explicitly written by the host. However, this is merely a guideline and not a strict requirement: An RMI product or function may define control-like registers that change for other reasons, although often such registers might better be treated as status registers (see section 2.4.2).

Each control register has a defined reset value. When the device resets for any reason, all the control registers revert to their reset values.

Writing to a control register generally affects some aspect of the device's operation, either immediately or at a later time. Some control registers may also have side effects upon writing. For example, some registers may cause the touch sensors to be recalibrated as a side effect of being written by the host, and some registers may cancel and restart the sensor measurement cycle (the *report period* of section 2.6.1) when written. Any such side effects are described in the documentation for the register.

A control register typically holds some *parameter* that configures the device. Some parameters use several registers; for example, registers $1046 and $1047 together hold the Max Position parameter of a Function $10 sensor. Other parameters take just one or a few bits of a register; for example, register $0000 contains the Report Rate and Sleep Mode parameters in different bit fields of the register. Sometimes the fixed properties of a particular device, such as the number of strip sensors in Function $14, are also referred to in RMI as *parameters.* All the parameters (both fixed and adjustable) of an RMI device are together called the *configuration* of the device.

Some parts of a control register may be marked *reserved* or "—", and some whole registers in a group of control registers may be *reserved*. Reserved bits normally reset to '0'; these bits may harmlessly be written to '0', but the behavior of the device is undefined if the host writes a reserved bit to '1'. For example, some reserved control bits may activate undocumented or proprietary features of the device when written to '1', and those undocumented features may change without notice from one version of the product to another.

Similarly, some possible settings of a control register may be marked *reserved,* and the behavior of the device is undefined if the host sets a register to a reserved value.

Some control registers or parts of control registers are implemented only in some versions or models of a device. When a control register is *unimplemented*, it will reset to a suitable value reflecting the fixed behavior that is implemented in its place (for example, a fixed sensitivity setting, or a fixed '0' enable bit for an unimplemented mode). At the implementation's discretion, an unimplemented register or register bit may be treated as a query register (where writes are ignored regardless of the data written), or as a control register that does nothing (where writes change the contents of the register but have no other effect on the device). In all cases, writes to unimplemented control bits are harmless.

Similarly, some possible settings of a control register may be unimplemented in some versions or models of a device; writing a control register to a setting that is unimplemented on the device has an undefined effect, but the effect is generally harmless and most often corresponds to one of the implemented settings of the register.

2.4.2. Status registers

Status registers contain information that may change spontaneously to reflect the status of the device. They are readable by the host but ordinarily are not written by the host. It is implementation-defined whether a host write to a status register will visibly change the register, but in any case a write to a status register will have no other effect on the device.

Reserved status bits typically read as '0', but the host should ignore reserved status bits for forward compatibility with future RMI devices that might use these bits for additional status information.

2.4.3. Query registers

Query registers are read-only registers that allow the host driver to determine what kind of RMI device is attached and what features it includes. Most hosts in embedded systems will never need to read the query registers at all, but platform host drivers and diagnostic tools may find these registers useful.

Query registers typically report constant data read from ROM on the device, but some queries may depend on the way the device is configured (for example, the Sensor Resolution of Functions $10 and $14). For more about query registers, see section 2.7.

The host should not write to a query register, but in any case writes to query registers are ignored.

Reserved query bits typically read as '0', but the host should ignore reserved query bits for forward compatibility with future RMI devices that might use these bits for additional query information.

2.4.4. Data registers

Data registers report sensor readings and other input data to the host. Data registers can generate interrupt requests at certain times or at a certain rate. Reading a data register clears its interrupt request condition. The special structure and behavior of RMI data registers are detailed in section 2.5.

Some parts of some data registers are marked *reserved*; host software should always ignore these bits. RMI devices typically will report '0' in all *reserved* data register bits, but the host should not rely on this.

2.4.5. Command registers

Command registers allow the host to perform discrete commands or to signal discrete events on the device. Each bit in a command register corresponds to a possible command. The recommended way to use a command register is to write a byte consisting of a single '1' bit in the position for the desired command, with '0's in the rest of the bits of the byte. Thus, the byte written can be considered as a "command code" that happens always to be a power of two.

Command register bits are special in that the host can only write them from '0' to '1': Writing a '0' to a command bit leaves the state of the bit untouched by the write operation. Writing a '1' to a command bit issues the command. The command bit automatically clears to '0' when the command completes.

Some commands may complete instantaneously; their bits will never read as ‘1’. Other commands may take some time to complete; their bits read as ‘1’ while the command is in progress. After issuing a command, the host may read back the command register repeatedly, at an interval suitable for the command, if it wishes to see how long the command is taking to complete. Or, the host may proceed immediately knowing that the command will execute in due course. In many cases there is no technical reason that the host must know when the command completes; by allowing the host to proceed immediately in these cases, RMI allows host drivers to use a simple, largely “stateless” design.

Certain command register bits may also be set to ‘1’ by the device itself. RMI does not define what happens if a ‘1’ is written to a command bit that is still ‘1’ from a previous posting of the same command. The behavior depends on the particular command and function. For this reason, the host should never read the command register and write back a value derived from the data that was read. Instead, the host should typically write a “command code” as described above.

It is acceptable to write a ‘1’ to one command bit when other command bits are already ‘1’ due to previously posted, still-pending commands. The result is that several commands will be posted at once. The order in which the device executes these pending commands is implementation-defined, and may not be the same as the order in which the commands were posted. In situations where the order of command execution is significant, the host should read and wait until the command register becomes $00 before writing a new command.

The Reset bit of command register $0004 is irregular in that the device will reset, and the RMI host interface will go “off the air,” when the reset command is posted. Therefore, it is not meaningful to read register $0004 to wait for completion of the reset command, nor to post another command at the same time as a reset command is pending.

Writing $00 to a command register has no effect, and is harmless.

Unused bits in a command register are marked *reserved* or *unimplemented.* The host must never write a ‘1’ to a reserved or unimplemented command bit. For example, some reserved command bits may activate undocumented or proprietary features of the device when written to ‘1’ and these undocumented or proprietary features are subject to change without notice. Reserved or unimplemented command register bits, and wholly reserved or unimplemented command registers, are not required to behave as described in this section when written to ‘1’.

2.5. Data reporting

An RMI device may have any number of *data sources.* In general, a data source corresponds to one independent sensor or input device, but in some cases a group of similar sensors (such as an array of buttons) will be combined to form a single data source.

2.5.1. Data source numbering

In an RMI device with *N* data sources, the sources are numbered 0 through *N*–1. The data sources of each present function are numbered consecutively, in order of ascending function numbers. If a function defines several data sources, the function specification defines the order in which those sources are numbered within its group of consecutive data sources. Only those data sources actually present in a given build-time product configuration are assigned numbers. However, data source numbers for a given product are always fixed at run time, as are the addresses of data registers in page $04*xx*.

For example, as illustrated in Figure 1, if product *A* contains a 1-D strip sensor (Function $14), a group of capacitive buttons (Function $18), and a group of GPIOs (Function $20), these would be data sources 0-and-1, 2, and 3, respectively. (Each 1-D strip contributes two data sources, one absolute and one relative.) In a product *B* that was like *A* but lacking capacitive buttons, the GPIOs would be source 2 instead of source 3. In a product *C* that was like *A* but with *two* 1-D strip sensors, the two strips would be sources 0-and-1 and 2-and-3, respectively, the capacitive buttons would be source 4, and the GPIOs would be source 5.

Product *C* of this example would use these fixed data source numbers even if it were a OneTouch-style device with the *potential* for two strips, each of which could be enabled or disabled at run-time. The data registers for a disabled strip would still be present in page $04*xx*, although of course they would not contain interesting data when their associated sensor is disabled.

Figure 1 also illustrates the layout of the data registers on address page $04*xx* for these examples; data register addresses are described in section 2.5.5. Figure 1 assumes there are between 9 and 16 capacitive buttons and between 1 and 8 GPI pins, so that Function $18 has two data registers and Function $20 has one data register.

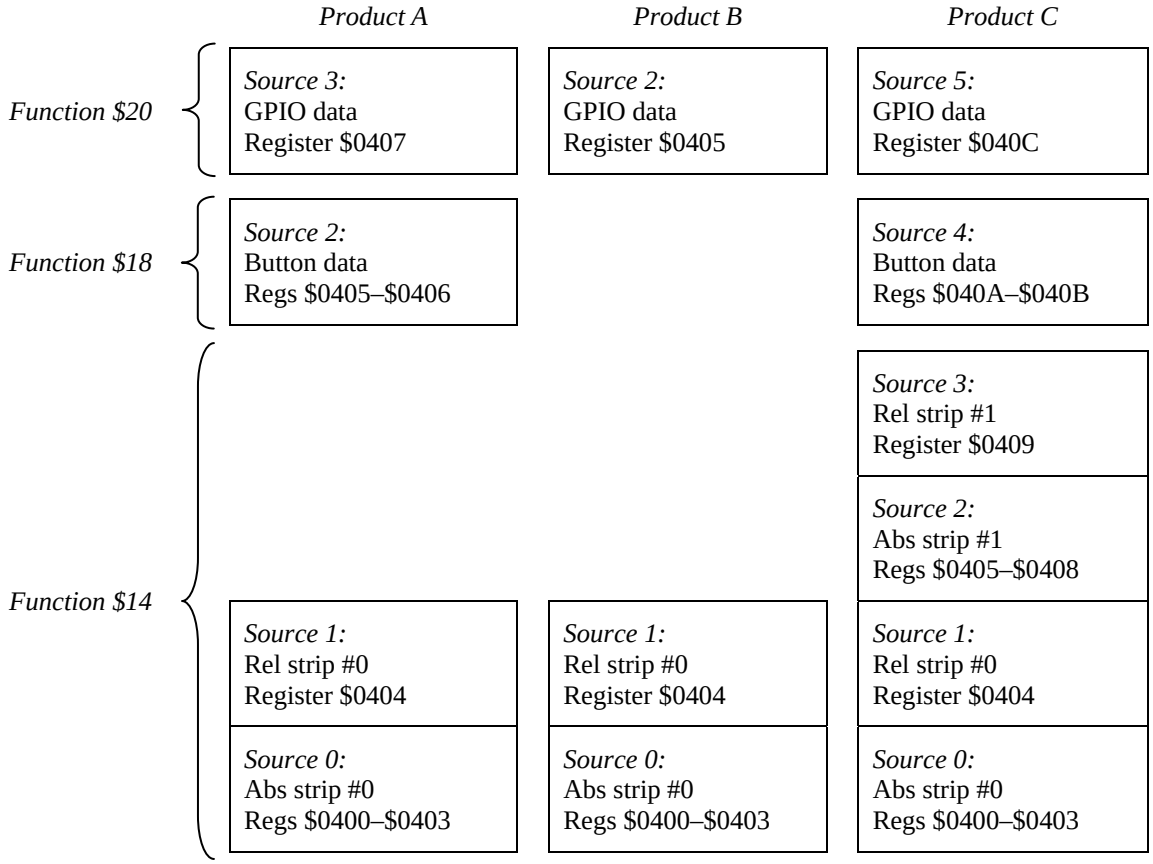


Figure 1. Numbering data sources and data registers (example)

2.5.2. Interrupt Request

RMI uses *interrupt requests* to signal whether or not a data source has information likely to be of interest to the host. Exactly what constitutes an interrupt, and when and at what rate new data will arrive, depends entirely on the kind of input device involved. Each RMI function defines the exact rules for signaling interrupt requests for each of its data sources.

Each data source maintains an independent interrupt request state bit. The Device Status register, described in section 2.5.3, reports the interrupt request bits either separately or in summary, depending on the total number of data sources in the device.

The interrupt request bit for a data source is ‘1’ if the data source has an interrupt request, or ‘0’ if there is no interrupt request for the source. Once a source has an interrupt request and its interrupt request bit has changed to ‘1’, the interrupt request bit remains at ‘1’ until the host reads data from the source, or (in some products) the host takes other actions that are documented to clear the interrupt request state of the data source.

The data registers always report meaningful data whether or not the interrupt request condition is present. Most often, an interrupt request state of ‘0’ implies that the data registers are unchanged since the last time the host read the data. However, some data sources may define a more selective “interrupt” criterion, so that certain “insignificant” data changes may occur without causing interrupt request to be asserted. Very simple hosts could even read the data registers by periodic polling, observing the data but completely ignoring the interrupt request bits and the attention signal.

After device reset and before the data registers have been read for the first time, every data source has an interrupt request status of ‘1’. This is true even for data sources corresponding to sensors with a “sensor enable” control that resets to the “disabled” state. (The reason for this rule is explained in section 2.5.4.)

2.5.3. Device Status register

The Device Status register reports the overall status of the device, such as which sources have interrupt requests and whether or not there is an error condition. Figure 2 shows the format of the Device Status register.

7	6	5	4	3	2	1	0
Error Flag	Configured Flag	—	interrupt request 4+	interrupt request 3	interrupt request 2	interrupt request 1	interrupt request 0

Figure 2. Device Status register

The bits of the Device Status register have the following meanings:

Error Flag (Device Status register bit 7)

This bit is ‘1’ if there is some kind of error on the device, or ‘0’ if there is no error. If the host sees a ‘1’ in the Error Flag bit, it can read the Error Status register (address $0002) to determine the type of error. The host’s typical response to any error is to reinitialize the device by rewriting the desired configuration into the device’s various control registers, possibly after resetting the device if appropriate.

Configured Flag (Device Status register bit 6)

This bit is '0' on a freshly reset RMI device. This bit changes to '1' upon the first host write to the Device Control register (address $0000), and it stays at '1' forever after (or until the device resets again for any reason). The host can examine the Configured Flag bit to detect when a spontaneous reset of the device during operation has caused the device to lose its configuration. (Hosts that merely power up the device and use it in its default configuration can safely ignore the Configured Flag.)

Reserved (Device Status register bit 5)

This bit is reserved for definition by future versions of the RMI standard. In devices that conform to the present RMI standard, this bit will always read as '0'.

Combined interrupt requests for data sources 4 and above (Device Status register bit 4)

This bit is '1' if any data source numbered 4 or higher has an interrupt request, or '0' if none of the high-numbered data sources have interrupt requests. For devices with fewer than five data sources, this bit always reads as '0'. For devices with more than five data sources, this bit effectively reads the logical OR of the high-numbered interrupt request bits. For devices with exactly five data sources, this bit is simply the interrupt request bit for data source 4.

Interrupt request for data sources 3, 2, 1, and 0 (Device Status register bits 3:0)

These bits report the interrupt request states of the four lowest-numbered data sources. If there are $N < 4$ data sources, the bits numbered N up through 3, inclusively, always read as '0'.

2.5.4. Attention signal

The host in an RMI system is in charge of all transactions with the device. When the device has an interrupt request to report to the host, it uses an *attention* mechanism to alert the host that it is time to read the data registers. Typically, a host system will connect the attention signal to an interrupt input. The attention signal is merely advisory; the host is free to read the data registers at any time. The form of the attention signal depends on the physical interface; for example, see section 4.3.5 for a description of the attention signal for RMI-on-SPI.

Synaptics can supply RMI devices in which the attention signal is active-high or active-low. The attention polarity is a build-time option, not run-time configurable. The attention signal is said to be *asserted* if it is in the state that alerts the host, which is high for active-high polarity or low for active-low polarity. The attention signal is in the *de-asserted* state when it is not asserted.

The attention signaling mechanism is compatible with both level-triggered and edge-triggered interrupt inputs on the host.

Register $0001 of every RMI device includes up to 8 interrupt enable bits, one bit per data source. For devices with more than 8 data sources, the interrupt enable bits of the higher-numbered data sources are held in registers in the address range $0020–$002F. Bits in registers $0001 and $0020–$002F that correspond to data source numbers not present in a given device are unimplemented control bits in the sense defined in section 2.4.1.

The attention signal is asserted whenever at least one data source has a ‘1’ in its bit of the interrupt enable mask and its interrupt request bit is also ‘1’. The attention signal may become asserted or de-asserted if the host writes to the control registers to change the interrupt enable bits.

Note that most data sources continue to work, and continue to operate their interrupt request bits, even if their interrupt enable bit is ‘0’. The interrupt enable bit merely controls whether interrupt requests on a source will send an attention signal to the host.

The host can disable attention by setting all the interrupt enable bits to ‘0’, thereby forcing the assertion signal to its de-asserted level. This gives the host a way to “disable interrupts” on the device side. For example, in a system where several devices’ attention pins have been logically OR’d together to drive a host interrupt pin, the host might wish to disable interrupts from some of the devices while remaining sensitive to other devices.

The attention signal is always asserted after device reset.

2.5.5. Data register page

Each data source reports information using a consecutive group of data registers within the $04*xx* address page. A data source may have one or more data registers, always a whole number of 8-bit bytes. The number of registers for a data source may vary from one product to another (for example, depending on the number of buttons supported). But the number of data registers for each source in any given product is fixed, independent of any run-time mode settings.

The data registers for data sources 0 through *N*–1 are assigned consecutive addresses starting at $0400. For example, if source 0 has 3 data registers, source 1 has 1 data register, and source 2 has 3 data registers, then source 0’s data registers have addresses $0400–$0402, source 1’s data register has address $0403, and source 2’s data registers have addresses $0404–$0406.

The remainder of the data register page (addresses $0407–$04FF in the example above), is unused. Reads to these unused data page addresses always return a dummy byte of $00.

To allow hosts to optimize their data accesses, some products may elect to provide additional data pages with the data sources arranged in a non-standard order, perhaps as either a build-time or a run-time option. For instance, in the example above, source 1’s data register might be visible at address $0500 as well as $0403, and source 0’s data registers might be visible at addresses $0501–$0503 as well as $0400–$0402. However, this feature is strictly optional, and RMI tools can rely on the fact that page $04*xx* is always present and always ordered in the way defined by the RMI standard. The RMI standard itself merely sets aside pages $05*xx*–$07*xx* for this general type of feature in case any product wishes to include one.

The act of reading any of the data registers for a data source clears the interrupt request bit for that source. When reading from the data register page, the reported Device Status register shows the interrupt request bits from before any the interrupt request bits are cleared as a result of the read operation.

2.5.6. Data coherence

When a read transaction spans several data registers of a given data source, it always reads the registers *coherently*. For example, if data source 0 includes registers $0400–$0402 and the host reads all three of these registers in one transaction, the data reported in register $0400 will always come from the same measurement as the data reported in registers $0401 and $0402.

WARNING: If the host reads the several registers of a data source using several *distinct* read transactions, the data are *not* guaranteed to be coherent. For example, if registers $0404–$0405 together encode a 16-bit number that changes from $12F5 to $1307, a read of register $0404 followed by a separate read of $0405 might return the very wrong value $1207. For this reason, at least for data sources where coherence among register bytes can be an issue, it is strongly recommended to read related data registers using a single multi-byte read transaction.

When a read transaction spans registers of more than one data source, each source will be read coherently, but the sources are not guaranteed to be coherent with one another. For example, if a device has two 1-D strip sensors as two sources, in some RMI device implementations a read transaction reading both strip sensors together might occasionally report older data for the first strip and newer data from a later measurement for the second strip.

2.6. Standard control, command, and status registers

2.6.1. Register $0000: Device Control register

The Device Control register contains bits that control the pace of processing in the device, and the conditions under which it can enter low-power states.

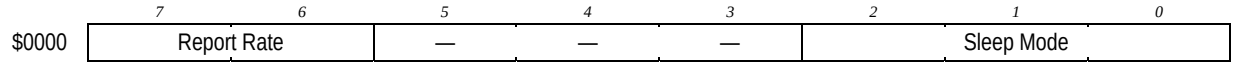


Figure 3. Device Control Register

The bits of this register are defined as follows:

Report Rate (Device Control Register $0000, bits 7:6)

This field sets the report rate for the device. It applies in common to all functions on the device that have a natural report rate.

Many RMI functions divide time into *report periods* that occur at a *report rate*, roughly analogous to the "packet rate" of a mouse. If there are several such functions on an RMI device that work in terms of report periods, all the functions schedule their operation to the common report period of the device.

The encoding of the Report Rate field is largely device-dependent. The RMI standard does not require any particular encoding. The reset value of the Report Rate field is also device-dependent, and corresponds to the normal or preferred report rate for the device.

In most Synaptics touch module products, the Report Rate value '10' encodes 80 report periods per second, '01' encodes 40 report periods per second, '00' sometimes encodes a device-specific slower rate, and '11' sometimes encodes a device-specific faster rate. The reset value of the Report Rate field is '10' in most touch module products, but it may be another value such as '01' in some products.

RMI functions that work in terms of report periods assert interrupt requests, and therefore also the attention signal, at most once per report period.

Usually, report periods happen at a steady rate. Some conditions may cause the report period in progress to be canceled and a new report period started. This will not result in any loss of data, but it will add a visible irregularity to the steady rate of reports. For example, depending on the device implementation, writes to some control registers will cancel and restart the report period.

Some RMI functions do not schedule their activity in terms of report periods; these are known as *asynchronous* functions. Asynchronous RMI functions may assert an interrupt request on any schedule depending on the needs of the function. If an RMI device contains only asynchronous functions, its Report Rate field is unimplemented and resets to '00' (as described in section 2.4.1).

Sleep Mode (Device Control Register $0000, bits 2:0)

This field controls power management on the device. This field affects all functions of the device together. Below are descriptions of possible sleep modes.

Note: Not all sleep modes are supported by every device. Consult the Product Specification or other device-specific documentation to find out which sleep modes are supported by the device.

Sleep Mode = '000': Force Fully Awake.

In this state, the device operates continuously, never entering a doze or sleep state. This mode consumes considerably more power than the others, and should be used only in special situations, for example in an unusual environment or usage pattern for which the product's normal automatic power management algorithms are ill-suited.

Sleep Mode = '001': Normal Operation.

In this state, the device automatically and invisibly switches between full operation and a doze state in which finger sensing happens at a reduced rate (typically a few tens of milliseconds). The doze sensing rate is chosen so that almost any human finger action, such as rapid tapping, will still perform well.

This setting merely authorizes the device to doze when it is able. Most products will be able to doze only under certain conditions, and will automatically remain fully awake, for example, when a finger is present or when pulse-width modulation for LEDs is active.

Most RMI devices can be left in the Normal Operation setting at all times.

Some Synaptics products do not support dozing; in these products, the Normal Operation state is identical to Force Fully Awake.

Sleep Mode = '010': Low-Power Operation.

This state is like the Normal Operation state, but with a longer doze sensing interval (typically a small fraction of a second) that conserves more power but that is likely to miss fast tapping actions. This setting may be preferable to Normal Operation in products whose known usage model never involves very rapid finger tapping gestures.

This Sleep Mode setting actually authorizes the device to switch automatically between the fully awake, normal dozing, and low-power dozing states depending on the pattern of sensor activity and other conditions.

Some Synaptics products do not support low-power operation; in these products, the Low-Power Operation state is identical to Normal Operation.

Sleep Mode = '011': Very-Low-Power Operation.

This state is like the Low-Power Operation state, but with an even longer doze sensing interval (typically on the order of one second) that conserves even more power but that is visibly slow to respond to user input after periods of prolonged inactivity.

This Sleep Mode setting actually authorizes the device to switch automatically between the fully awake, normal dozing, low-power dozing, and very-low-power dozing states depending on the pattern of sensor activity and other conditions.

Some Synaptics products do not support very-low-power operation; in these products, the Very-Low-Power Operation state is identical to Low-Power Operation.

Sleep Mode = '100': Sensor Sleep.

This state fully disables touch sensors and similar "analog" inputs on the device. All touch sensors report the "not touched" state regardless of any finger presence. Digital inputs, such as mechanical buttons attached to GPI pins, continue to operate even in the Sensor Sleep state.

Setting the Sleep Mode field to Sensor Sleep merely constitutes a request to the device to enter the sleeping state. The device may continue to operate in a higher-power state for a short time before it goes to sleep. In particular, finger presence may still be reported for one or two more report periods after Sleep Mode is set to Sensor Sleep.

All RMI devices support the Sensor Sleep state at least to the degree of forcing the touch sensors to the "not touched" state. In many devices, the Sensor Sleep state conserves additional power. In devices that do not support this deeper sleeping mode, the Sensor Sleep state is identical to Very-Low-Power Operation except for forcing the "not touched" state. Even then, this mode may help to conserve overall system power by interrupting the host less often.

Sleep Mode = '101,' '110,' and '111': Reserved.

These Sleep Modes are reserved for definition by future versions of RMI.

The reset state of the Sleep Mode bits depends on the device. Most products default to the Normal Operation state, but some (particularly those that do not implement fast tapping gestures) may default to the Low-Power Operation state.

Other bits of register $0000:

These bits are *reserved* for definition in future versions of the RMI protocol.

2.6.2. Register $0001: Interrupt Enable register

The Interrupt Enable control register determines which data sources are able to assert the attention signal.

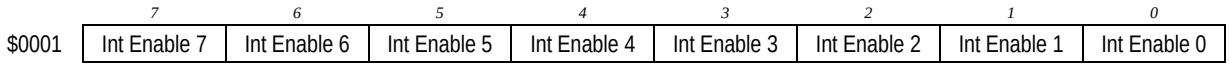


Figure 4. Interrupt Enable register

Each bit of this register controls whether the corresponding data source will assert attention when it has an interrupt request. Bit *n* of this register is '1' if the interrupt request on data source *n* should assert attention, or '0' if the interrupt request on source *n* should not affect the attention signal. See section 2.5.4.

Depending on the implementation, the effect on the Attention signal of a change to the interrupt enable bits may be either immediate or deferred until the next report period.

Every interrupt enable bit for a present data source resets to '1'. Interrupt enable bits corresponding to not-present data sources are unimplemented and reset to '0'.

Setting this field to all '0' bits effectively disables the attention signal altogether.

2.6.3. Register $0002: Error Status register

The Error Status register reports the reason for the most recent device reset or error condition.

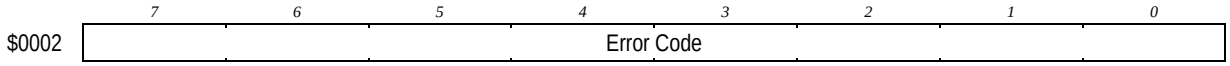


Figure 5. Error Status Register

If the device signals an error by reporting '1' in the Error Flag bit of the Device Status register, this register holds an *error code* byte in the range $01–$7F that identifies the kind of error. If several error conditions arise at once, this register reports one of the extant errors; which one it reports is implementation-dependent.

Error conditions created by host actions, such as invalid configurations of control bits, may be reported instantly or they might not be reported until a few milliseconds after the offending register was written (for example, error conditions might not be checked until the next report period). Similarly, if new data is written to the control registers to correct the error condition, it may take a few milliseconds for the Error Flag to clear.

Immediately after reset but before any error has occurred, the Error Flag will be '0' and the Error Status register will hold a *reset code* byte in the range $80–$FF. The reset code reports the reason that the device was last reset. Some RMI devices will be unable to distinguish some or all of the reset reasons, in which case they will report code $80 to show that the reason for the reset is unknown.

At any other time (when at least one error condition has occurred *and* been corrected since reset), the contents of this register are undefined and uninteresting.

RMI defines the following standard error codes and reset codes:

Code $00: Reserved.
This error code is reserved for definition by future versions of RMI, although well-behaved RMI devices can report $00 after an error condition has been cleared.

Code $01: Invalid Configuration.
This error signals a problem with the general configuration of the device, not specific to any one function. Many RMI devices will not implement this error at all; some devices might, for example, signal an error $01 if the Report Rate bits of register $0000 were set to an unsupported value.

Code $02: Device Failure.
This error signals a hardware problem with the device, not specific to any one function. Many RMI devices will not implement this error at all; some devices might, for example, signal an error $02 if the firmware fails a CRC self-check.

Codes $03–$0F: Reserved.
These error codes are reserved for definition by future versions of RMI.

Codes $10–$7F: Function-specific errors.
If Function $*xx* signals an error, the error code $*xx* is reported. Exactly what constitutes an error condition depends on the function; a typical example would be invalid settings in the control registers of the function. Many functions will not implement any error conditions at all.

Code $80: Reset occurred.
The Error Status register holds this code if no error has occurred since the last time the device was reset, but this RMI device does not implement reset-reason reporting so the exact nature of the reset is unknown.

Code $81: Power-on reset.
The Error Status register holds this value after a reset due to power cycling of the device.

Code $82: External reset.
The Error Status register holds this value after a reset that happened because an external RESET pin was asserted.

Code $83: Host-initiated reset.
The Error Status register holds this value after a reset that happened because the host explicitly reset the device by writing a ‘1’ to bit 0 of command register $0004.

Code $84: Fault reset.
The Error Status register holds this value after a reset due to some other erroneous condition in the device, such as an internal firmware assertion failure.

Code $85: Watchdog reset.
The Error Status register holds this value after a reset due to timeout of an on-chip watchdog timer in the device.

Codes $86–$FF: Reserved.
These error or reset codes are reserved for definition by future versions of RMI.

2.6.4. Register $0003: Interrupt Request status register

The Interrupt Request status register reports which data sources currently have interrupt requests.

	7	6	5	4	3	2	1	0
$0003	Interrupt Request7	Interrupt Request6	Interrupt Request5	Interrupt Request4	Interrupt Request3	Interrupt Request2	Interrupt Request1	Interrupt Request0

Figure 6. Interrupt Request Status Register

Bit *n* of this register is '1' if data source number *n* has an interrupt request, or '0' otherwise. Note that bits 3:0 of this register are identical to bits 3:0 of the Device Status register; register $0003 is primarily of interest when the device has many data sources and the host wishes to know exactly which of these sources have data to report.

2.6.5. Register $0004: Device Command register

The Device Command register is used to issue special commands to an RMI device. For general guidelines on the use and operation of command registers, see section 2.4.5.

	7	6	5	4	3	2	1	0
$0004	—	—	—	—	—	—	ReZero	Reset

Figure 7. Device Command Register

The bits of this register are defined as follows:

Reset command (Device Command Register $0004, bit 0)

Writing a '1' to this bit causes the device to reset exactly as if its RESET pin had been pulled low (except possibly for reporting a different reset reason in register $0002).

The device's host interface (SMBus or SPI pins) may not operate for a certain amount of time T_{RESC} (about 1 ms) after a reset command or a low level on the RESET pin. This delay will be much shorter than the delay T_{POR} before the host interface begins operating after a power-on reset. The T_{RESC} delay begins at the end of the write transaction that writes to register $0002 (for example, at the rise of SSB in the case of RMI on SPI).

Note: The device will assert attention as soon as it is capable of responding to an operation on its physical interface.

ReZero command (Device Command Register $0004, bit 1)

Writing a '1' to this bit causes all sensors on the device to revert to their "not touched" states. The host should never need to issue a ReZero command under normal conditions, because Synaptics devices handle zeroing completely automatically. Some hosts may wish to issue ReZero commands to inform the device that the physical environment has suddenly changed, for example, because a covering metal flap has just been lifted off the sensor. Depending on the particular product, the ReZero command may be instantaneous, or it may remain posted for several milliseconds until it is processed on the next report period.

Reserved (Device Command Register $0004, bits 2, 3 and 4)

These command bits are *reserved* for definition in future versions of the RMI protocol.

Reserved (Device Command Register $0004, bits 5 and 6)
These command bits are *reserved* for definition by the physical layer. Their meaning might not be consistent from one physical layer to another.

Reserved (Device Command Register $0004, bit 7)
This command bit is *reserved* for definition by the device manufacturer, typically for use with undocumented diagnostic functions. Its meaning is entirely at the discretion of the manufacturer; it may be consistent and universal for all the manufacturer's RMI products, or it may vary without notice from one version of the device to another.

2.7. Standard query registers

RMI defines a standard set of query registers that all RMI devices implement to describe the type, version, and capabilities of the device. The queries appear as read-only registers in address pages $02*xx* and $03*xx*.

Most RMI hosts will be designed to work with a specific device, and these hosts can completely ignore the query registers. The standard queries are more useful for diagnostic tools that need to work with a wide variety of RMI devices, and for multi-purpose host drivers that must support a varied platform of products or systems.

The standard RMI queries are shown in Table 2.

Address range	*Purpose*
$0200	RMI Protocol Version query
$0201	Manufacturer ID query
$0202	Physical Interface Version query
$0203	Product Properties query
$0204–$0207	Product Info queries
$0208–$020D	Device Serialization queries
$020E–$020F	*Reserved for future definition by RMI*
$0210–$021F	Product ID queries
$0220–$02FF	*Reserved for future definition by RMI*
$0300–$030F	*Reserved for future definition by RMI*
$0310–$037F	Function Presence queries
$0380–$03FF	*Reserved for future definition by RMI*

Table 2. RMI standard queries

The query registers are read-only registers: They will always read the same data for a given device configuration, and write operations to them by the host are ignored. In some products, certain queries may depend on the settings of control registers or feature strapping pins on the device, but the queries will be constant for a given setting of the control registers or strapping pins.

In devices that comply with the current version of RMI, queries marked *reserved* in Table 2 will read as $00 for forward compatibility with future versions of RMI.

2.7.1. Register $0200: RMI Protocol Version query

This byte reports the major and minor version numbers of the RMI protocol that the device implements. The RMI protocol version refers to the version number of the present document that the overall device conforms to. The RMI protocol version does *not* track the versions of the particular physical layer, particular RMI functions, or firmware on the device; these are tracked by other version queries described below.

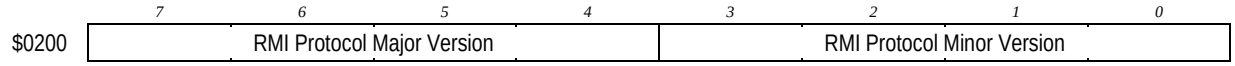


Figure 8. RMI Protocol Version Query register

Several RMI queries report version numbers of various aspects of an RMI product. Register $0200 reports the version number of the RMI protocol itself; other registers report the version of the product or of a particular function. In general, reporting version number *x.y* for an aspect of RMI means that the device conforms to version *x.y* of the specification for that aspect. The various aspects' version numbers might change independently; for example, for some reason a device might implement a newer version of Function $10 but an older version of the physical interface, or vice versa.

RMI version numbers generally consist of two unsigned integers, *major* and *minor*. Minor, mostly compatible changes would generally be expressed by increasing the *minor* version by one, holding the *major* version the same. Major or significantly incompatible changes would generally be expressed by increasing the *major* version by one and changing the *minor* version back to 0 or 1.

The *major* version can be 0 to indicate a "preliminary" version of the aspect being described (the *minor* version then identifies which of several preliminary drafts pertains). The *minor* version can be 0 to indicate a "preliminary" version of a specific major version (with no way to distinguish among drafts). If *major* and *minor* are both non-zero, the version represents a final shipping specification. The version number 0.0 is never used. (Note that, while register $*xx*00 will probably report $00 if $*xx* is a not-present function number, register $*xx*00 is in this case an unimplemented register, not a true Function Version query register.)

RMI diagnostic tools will display version numbers as "*major*.*minor*" where *major* and *minor* are each decimal integers from 0 to 15.

2.7.2. Register $0201: Manufacturer ID query

This byte reports the identity of the manufacturer of the RMI device.

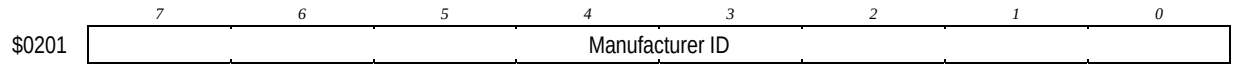


Figure 9. Manufacturer ID Query register

Synaptics RMI devices report a Manufacturer ID of $01.

2.7.3. Register $0202: Physical Interface Version query

This byte reports the major and minor version of the RMI physical layer. For example, in RMI-on-SMBus devices, register $0202 reports the version number of the SMBus interface specification contained in section 4.1.

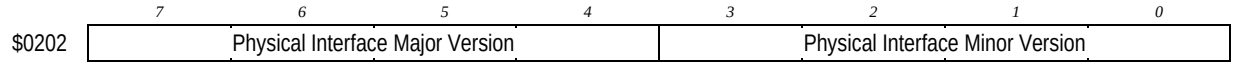


Figure 10. Physical Interface Version Query register

The version of each physical layer is included in the specification document for the physical layer. Section 4 describes the standard RMI-on-I^{2}C, RMI-on-SMBus, and RMI-on-SPI physical layers.

2.7.4. Register $0203: Product Properties query

This byte contains bits that describe whether the RMI product has various optional properties.

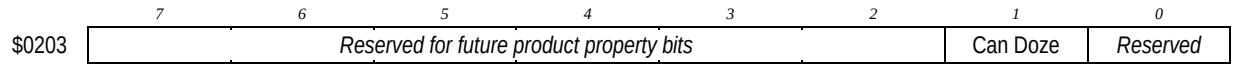


Figure 11. Product Properties Query register

Each property bit is '1' if the product has the associated property, or '0' if the product does not have the associated property. Reserved property bits report as '0', but they may report as '1' in devices that comply with a future version of RMI.

At present, only one bit of register $0203 has a defined meaning.

Can Doze property (Register $0203, bit 1)

This bit is '1' if the product is able to reduce the amount of power it draws in response to higher settings of the Sleep Mode field of register $0000.

2.7.5. Registers $0204–$0207: Product Info query

These bytes report information for the product.

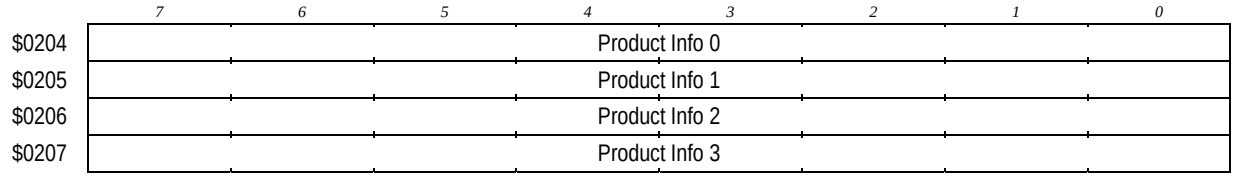


Figure 12. Product Info Query registers

Each byte is an unsigned integer in the range from 0 to 255. The contents of these registers are product specific and are specified when the product is ordered.

2.7.6. Registers $0208–$020D: Device Serialization queries

These six bytes optionally record the individual identity of the device. Some RMI devices are not serialized at the factory; for unserialized devices, all six of these bytes report $00.

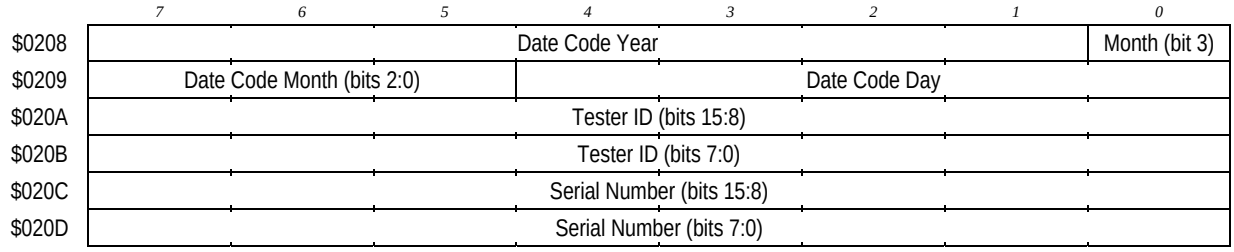


Figure 13. Device Serialization Query registers

The various Device Serialization queries are defined as follows:

Date code (Registers $0208–$0209)

These 16 bits are intended to record the date on which the module was manufactured. The actual interpretation is up to the manufacturer, but RMI diagnostic tools will display this field assuming the bits are divided into year, month, and day fields as shown in Figure 13. The year code is a number from 1 to 127 to indicate years 2001–2127. The month code is a number from 1 to 12 to indicate the months January through December. The day code is a number from 1 to 31 to indicate the day of the month. The day field can be 0 to indicate "unknown day of the month"; the day and month fields can both be 0 to indicate "unknown day of the year"; the entire 16 bits can be zero to indicate "unknown manufacturing date."

If the month code is $E or $F, the date code is formatted in a different way that is also recognized by RMI diagnostic tools. In this format, bits 5:0 of register $0209 encode a week number within the year. The remaining possible month code $D is reserved for future definition by RMI.

Tester ID (Registers $020A–$020B)

These 16 bits are intended to identify the equipment used to manufacture or test the RMI device. The value $0000 conventionally means "unknown tester." The actual interpretation of these registers is up to the device manufacturer.

Serial number (Registers $020C–$020D)

These 16 bits are intended to record a unique serial number for a given tester and day. The value $0000 conventionally means "no serial number recorded." The actual interpretation of these registers is up to the device manufacturer. RMI diagnostic tools will display the tester ID and serial number in the form "*tester*:*serial*", where *tester* and *serial* each are four hexadecimal digits.

If the Serial Number query (registers $020C–$020D) is not $0000, then the combination of Manufacturer ID, Product ID, Date Code, Tester ID, and Serial Number should be completely unique among all manufactured RMI devices. (Nothing in the RMI standard itself uses the serialization information, but hosts and tools may rely on this uniqueness property when the Serial Number is non-zero.)

2.7.7. Registers $0210–$021F: Product ID queries

These bytes report the identity of the particular RMI device or product.

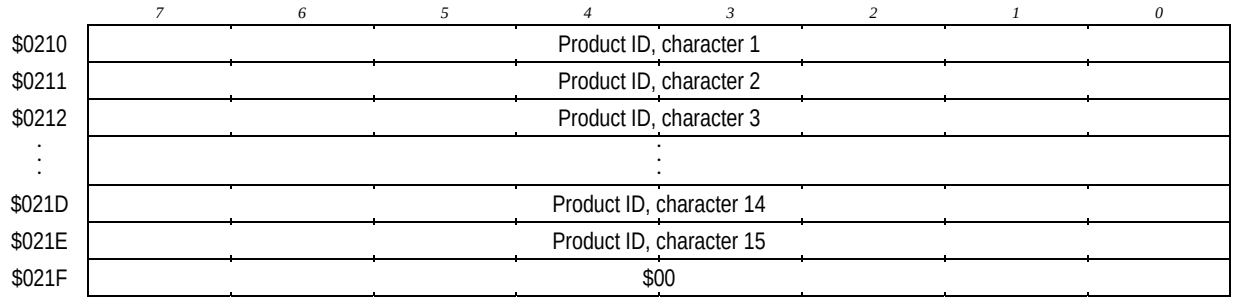


Figure 14. Product ID Query registers

These registers form a null-terminated string that identifies the product. If the string is of length *N* characters, then registers $0210 through $0210 + *N* – 1, inclusive, encode printable ASCII characters in the range from $20 to $7E, and registers $0210 + *N* through $021F, inclusive, are $00.

The form of the Product ID string depends on the product manufacturer. The exact Product ID format for Synaptics products will vary from one product family to another. For custom touch sensing modules, the Product ID currently is of the form, for example, "`TM605`" or "`TM605-2`", but this format is subject to change in the future.

2.7.8. Registers $0310–$037F: Function Presence queries

This group of registers can be used to identify all functions present on a device. The low 8 bits of the register address correspond to a function number from $10 to $7F. Function Presence query register $03*xx* is $00 if Function $*xx* is not present on the device, or it is a non-$00 value if Function $*xx* is present.

Note: RMI informally reserves the address range $0380–$03FF in case a future extension of RMI doubles the range of function numbers.

The Function Presence query register for each present function is formatted as follows:

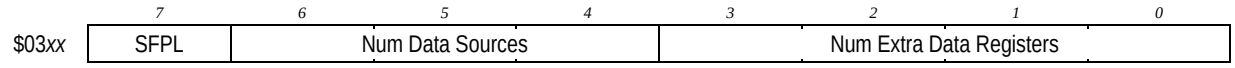


*Figure 15. Function Presence Query register (if Function $*xx *is present)*

Num Data Sources (Register $03xx, bits 6:4)

These bits of the Function Presence Query register report the total number of data sources defined by the function in the current product, as an integer from 0 to 7.

Num Extra Data Registers (Register $03xx, bits 3:0)

Each data source contributes at least one data register to the function. If Num Data Sources is non-zero, then bits 3:0 of the Function Presence Query register report the total additional number of data registers defined by the function beyond the assumed one data register per data source, as an integer from 0 to 15. Thus, (Num Data Sources + Num Extra Data Registers) is the total number of data registers defined by the function.

If Num Data Sources is zero, then the function must necessarily have no data registers, and bits 3:0 of the query instead report $F. If the number of data sources or the number of data registers is too large to be represented in this format, bits 6:4 instead report zero and bits 3:0 report $E. Note that bits 6:0 of the Function Presence Query are guaranteed to be not-all-zero for any present function.

Special Function Page Layout (Register $03xx, bit 7)

If this bit is '1' for Function $*xx*, then RMI diagnostic tools can assume that the register page $*xx*00–$*xx*FF has a somewhat standardized layout as described in section 2.8. If this bit is '0', the function-specific register page is irregular and tools should make no assumptions unless they specifically recognize the function number.

RMI devices must return $00 when the host reads any address in the $0310–$037F range corresponding to a function that is not present in the device.

Diagnostic tools, and multi-purpose host drivers, can use the defined properties of the Function Presence Query registers to enumerate all the functions present on an unknown device, and to correctly interpret the bits of the Device Status register and the data registers of page $04*xx* even if they do not recognize some of the included functions. (Note that if any Function Presence query reports $0E in bits 6:0, diagnostic tools will not be able to interpret the device's data without recourse to outside information about the device.)

2.8. Function-specific registers

Function number $*xx* is associated with a page of 256 registers whose 15-bit addresses are $*xx* in their high 7 bits. For example, register addresses $1400–$14FE are reserved for definition by Function $14.

However, a function can set the Standard Function Page Layout bit (bit 7) of its Function Presence query (see section 2.7.8) to indicate that it follows certain conventions about the layout of its register page. Diagnostic tools and platform drivers may be able to use this fact to provide a basic interface to functions they do not recognize.

If the Standard Function Page Layout bit is '1' for Function $*xx*, then registers $*xx*00–$*xx*FF generally follow the conventions shown in Table 3.

Address range	*Purpose*
$*xx*00	Function Version query
$*xx*01–$*xx*3F	Other function-related queries
$*xx*40	Command register, if present
$*xx*41–$*xx*7F	Control and status registers
$*xx*80–$*xx*FE	Other registers
$*xx*FF	Reserved for use by the physical interface

Table 3. Standard function page layout

If the Standard Function Page Layout bit is '0' for Function $*xx*, or if Function $*xx* is not present, then tools should not assume anything about the contents, purpose, or behavior of registers $*xx*00–$*xx*FF.

3. Standard RMI functions

This section describes the RMI functions currently defined by Synaptics.

The standard function numbers are shown in Table 4.

Function	*Purpose*	*See page*
$10	2-D sensors (TouchPads)	32
$13	Scroller (typically used with TouchPad)	45
$14	1-D sensors (strips and rings)	49
$18	Capacitive buttons	53
$20	Digital GPIOs (including mechanical buttons)	57
$22	Simplified LEDs	62
$40–$79	Reserved	—
$7C–$7F	*Reserved for unique/specialty devices*	—

Table 4. Standard function numbers

These standard RMI functions are described in the following sections.

3.1. Function $10: 2-D TouchPad sensors

Function $10 implements one or several two-dimensional touch position sensors, such as Synaptics TouchPad™ or ClearPad™ products.

3.1.1. Number of 2-D sensors

A Function $10 device may include any number of 2-D sensors. The sensors are identified by consecutive numbers starting with 2-D sensor #0. Each 2-D sensor has two data sources. The first data source reports the absolute finger position, and the second data source reports the relative finger motion.

The Num Data Sources field of register $0310 expresses how many distinct 2-D sensors are present in the device. When Function $10 is present with 1, 2, or 3 2-D sensors, its Num Data Sources value will be 2, 4, or 6, respectively. If Function $10 has more than three 2-D sensors, its register $0310 will be $8E, indicating that the standard queries are unable to express the number of 2-D sensors (see section 2.7.8). Tools that operate a device with more than three 2-D sensors must rely on information beyond the device's own queries to interpret the device.

For example, in a device with two 2-D sensors, source 0 is absolute position for 2-D sensor #0, source 1 is relative motion for 2-D sensor #0, source 2 is absolute position for 2-D sensor #1, and source 3 is relative motion for 2-D sensor #1.

When more than one 2-D sensor is present in the device, all the 2-D sensors operate independently. Each 2-D sensor reports its data in separate data registers with separate interrupt request and interrupt enable bits. Each 2-D sensor's properties are described by a separate set of queries, and each 2-D sensor is configured by a separate set of control registers.

3.1.2. Register page layout

Function $10 implements the Standard Function Page Layout, as shown in Table 5.

Address range	*Purpose*
$1000	Function Version query
$1001	General 2-D Properties query
$1002–$1009	Queries for 2-D sensor #0:
$1002–$1003	Sensor Properties
$1004–$1005	Sensor X Max Position
$1006–$1007	Sensor Y Max Position
$1008	Sensor Resolution
$1009	*Reserved for future definition*
$100A–$1011	Queries for 2-D sensor #1
$1012–$1019	Queries for 2-D sensor #2
$101A–$103F	*Reserved for future definition*
$1040	*Reserved for future command register*
$1041	General 2-D Control register
$1042–$1043	*Reserved*
$1044–$1047	Sensor control registers for 2-D sensor #0:
$1044	Sensor Sensitivity
$1045	*Reserved for future definition*
$1046–$1047	Sensor Max Position
$1048–$104B	Sensor control registers for 2-D sensor #1
$104C–$104F	Sensor control registers for 2-D sensor #2
$1050–$107F	*Reserved for future control registers*
$1080–$10FF	*Reserved for future definition*

Table 5. Function $10 register page

3.1.3. Query registers

Register $1000 reports the version number of the Function $10 specification that the device implements.

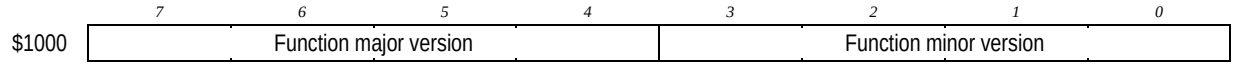


Figure 16. Function $10 Version query register

Register $1001 contains general information about the 2-D sensor function.

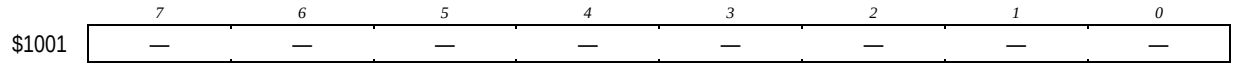


Figure 17. Function $10 General 2-D Properties query register

All bits of this query are *reserved*.

Registers $1002–$1019 describe information about each of up to three 2-D sensors. For devices with fewer than three 2-D sensors, the query registers corresponding to higher-numbered 2-D sensors are unused and read as $00. For devices with more than three 2-D sensors, the higher-numbered sensors might not be described by device queries.

The queries for a given 2-D sensor are shown in Figure 18:

	7	6	5	4	3	2	1	0
$1002	—	—	—	—	—	—	—	—
$1003	—	—	Has 2D Scrollers	Has Scroller	Has Enhanced Gestures	Has Multi Finger	Has Palm Detect	—
$1004	—	—	—	Sensor X Max Position (bits 12:8)				
$1005	Sensor X Max Position (bits 7:0)							
$1006	—	—	—	Sensor Y Max Position (bits 12:8)				
$1007	Sensor Y Max Position (bits 7:0)							
$1008	Sensor Resolution							
$1009	—	—	—	—	—	—	—	—

Figure 18. Function $10 per-sensor query registers (addresses shown for sensor #0)

The contents of these registers are defined as follows:

*Has Palm Detect (Register $1003+8*N, bit 1)*

This bit is ‘1’ if the device is capable of measuring finger width, or ‘0’ if not. (See the Width data register field in section 3.1.5.)

*Has Multi Finger (Register $1003+8*N, bit 2)*

This bit is ‘1’ if the device is capable of counting the number of fingers on the pad, or ‘0’ if not. (See the Sensor Status data register field in section 3.1.5.)

*Has Enhanced Gestures (Register $1003+8*N, bit 3)*

This bit is ‘1’ if the device is capable of supporting enhanced gestures, or ‘0’ if not. (See the Gesture data registers in section 3.1.5.2)

*Has Scroller (Register $1003+8*N, bit 4)*

If this bit is ‘1’, there is a Function $13 in the device that represents either a mechanical scroll wheel, or a scrolling zone or strip, associated with this 2-D sensor. By convention, the *N*th 2-D sensor with a ‘1’ bit in either of these bits is associated with the *N*th data source of Function $13.

If the 2-D sensor has a separate (non-virtual) scroll strip associated with it, the designer of the device may elect either to associate the strip with the 2-D sensor in the same way as a virtual scrolling zone (above), or to treat the strip as a separate 1-D sensor. The former case is more appropriate if the strip is to be used strictly as a relative scroller. The latter case allows the host to obtain absolute position and Z data for the strip. In the latter case, the 2-D sensor’s Has Scroller bit will be ‘0’, and the 1-D sensor will be treated as an independent Function $14 in the device.

*Has 2D Scrollers (Register $1003+8*N, bit 5)*

If this bit is ‘1’, the 2-D sensor is associated with two consecutively-numbered data sources of Function $13 which combine to report general two-dimensional scrolling deltas; the first of these two sources always reports horizontal scrolling, and the second always reports vertical scrolling. If the Has 2D Scrollers bit is ‘0’, then either the scroller is only able to scroll in one axis (vertical scrolling only), or it can scroll in only one axis at a time (disjoint horizontal and vertical virtual scroll zones on the same pad); in this case, a single scroller data source is used.

Many RMI devices, such as custom-designed TouchPad modules, have sensors with a known physical size and aspect ratio. On these devices, each 2-D pad has X Max Position, Y Max Position, and Sensor Resolution queries that report the physical properties of the pad. These queries are computed as a function of the Max Position control registers. If the host writes to the Max Position registers, it may take up to two report periods (as defined in section 2.6.1) to recalculate the X Max Position, Y Max Position, and Sensor Resolution queries; those query registers' contents are untrustworthy for up to that much time after any write to Max Position. The data registers for the pad are also untrustworthy for up to two report periods after any change to Max Position.

*X Max Position and Y Max Position (Registers $1004+8*N – $1007+8*N)*

These queries describe the largest values that will be reported in the X Position and Y Position data registers, respectively. The X and Y Positions are always scaled the same as each other in units-per-millimeter terms, so that a 100-unit change in X Position represents the same physical distance on the pad as a 100-unit change in Y Position.

If the pad has landscape orientation (in other words, it is wider than it is high, like a traditional TouchPad mounted in a laptop computer), then X Max Position is the same as Max Position, and Y Max Position is a smaller number depending on the aspect ratio of the 2-D pad. For a portrait-oriented pad (one that is mounted with the sensor higher than it is wide), Y Max Position is the same as Max Position, and X Max Position is smaller. For a perfectly square pad, both X Max Position and Y Max Position are equal to Max Position.

*Sensor Resolution (Register $1008+8*N)*

This query reports the scale of the X and Y Position values, measured in position units per millimeter, rounded to the nearest 8-bit unsigned integer. If the calculated resolution would exceed 255 units per millimeter, the Sensor Resolution query reports 255.

Thus, if X Width represents the known physical X-axis width of the sensor active area in millimeters, and Y Height represents the physical Y-axis height in millimeters, then the queries are calculated as:

X Max Position = (Max Position) * (X Width) / max(X Width, Y Height);
Y Max Position = (Max Position) * (Y Height) / max(X Width, Y Height);
Sensor Resolution = X Max Position / X Width = Y Max Position / Y Height.

The X Width and Y Height values are not directly reported in Function $10 queries, but the host can derive them from the other queries by solving the above equations, as follows:

X Width = X Max Position / Sensor Resolution;
Y Height = Y Max Position / Sensor Resolution.

If Sensor Resolution is on the order of 100 units per millimeter, then these formulas will yield width and height values accurate to about 1%. The aspect ratio of the sensor is simply the ratio of X Width to Y Height, or, equivalently, the ratio of X Max Position to Y Max Position.

In other RMI devices, such as Synaptics OneTouch™ products, the device firmware does not know the physical size and aspect ratio of the sensor. In such devices, the X Max Position, Y Max Position, and Sensor Resolution query registers are unimplemented and all reset to $00 for each 2-D pad on the device. For purposes of the definitions below, the X Max Position and Y Max Position of such a device are both equal to the Max Position control register value.

3.1.4. Control registers

Register $1041 contains global settings that control all of the 2-D sensors together.

	7	6	5	4	3	2	1	0
$1041	—	—	No Filter	—	Reduced Reporting	No Clip Z	No Decel	—
$1042	*Reserved*							
$1043	*Reserved*							

Figure 19. Function $10 General 2-D Control register

Register $1041 resets to $00, and can be left at its reset setting in most applications. Its bits are defined as follows:

No Filter (Register $1041, bit 5)

If this bit is set to ‘1’, then the position filtering (if any) that the sensor normally applies to its X Position and Y Position data is disabled. This may cause the position values to be more jittery or susceptible to electrical noise, but it may be advantageous to set No Filter in unusual applications for which the standard filtering performs poorly.

Reduced Reporting (Register $1041, bit 3)

In general terms, a sensor will assert the ATTN interrupt at a nominal 80 Hz rate for the entire duration that one or more fingers is present on the sensor. Setting this register to '1' means that sensors will assert their ATTN interrupt only under the following circumstances:

- If the finger count changes and the sensor status does not indicate a transitional finger (it is not equal to '111') or
- if the gesture bit changes.

For RMI devices that do not support multiple fingers, setting this control bit means that ATTN will be asserted once when a finger lands (sensor status changes from '000' to '001'), and once again when the finger leaves (sensor status changes from '001' to '000'). For RMI devices that sense multiple fingers, ATTN will be asserted each time an additional finger lands, and each time that any finger leaves. If a user were to tap the sensor, the ATTN interrupt would be asserted whenever the gesture bit changed state. See Section 2.5.4 for more information about attention signals.

No Clip Z (Register $1041, bit 2)

If this bit is ‘0’ (the default setting), then when the Sensor Status field of the data registers reports no finger present, the Z data register is forced to zero and the Position data registers report the last valid finger position.

If this bit is set to ‘1’, then capacitance readings too small to qualify as a finger may still report a non-zero Z. Depending on the physical design of the sensor, this mode may allow a limited form of proximity detection. However, it will cause the sensor to report interrupt requests occasionally when no real finger is near, which may interfere with power management in the host system.

No Deceleration (Register $1041, bit 1)

If this bit is '0' (the default setting), the device automatically reduces the scale factor it uses to compute the relative motion deltas when the finger moves especially slowly; this feature assists with fine positioning of a cursor on a large display. If this bit is set to '1', the deceleration feature is disabled and the scale factor of the motion deltas is constant (except as controlled by Relative Acceleration, described below).

Reserved (other bits of register $1041, Registers $1042, $1043)

These bits are reserved for definition by future versions of RMI Function $10.

Registers $1044–$104F control the operation of up to three 2-D sensors. For devices with more than three 2-D sensors, the higher-numbered 2-D sensors are configured in a device-dependent way. For devices with fewer than three 2-D sensors, the higher-numbered control registers in this range are unimplemented and reset to $00.

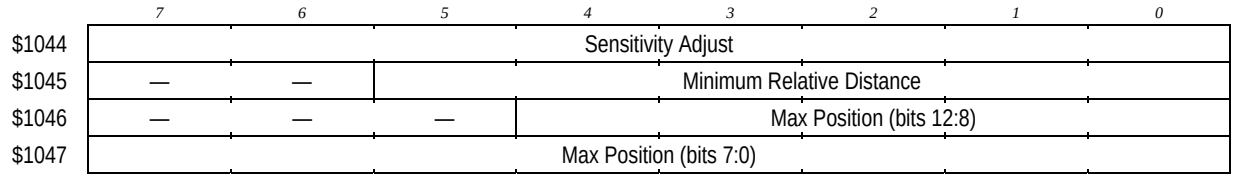


Figure 20. Function $10 per-sensor 2-D Control registers (addresses shown for sensor #0)

The contents of these registers are defined as follows:

*Sensitivity Adjust (Register $1044+4*N)*

This register resets to $00, and can be left at its reset setting in most applications. It holds a signed 8-bit integer, where positive values increase the sensitivity to light touches, and negative values decrease the sensitivity to firm touches. The Sensitivity Adjust value is scaled in the same units as those of the Z data register.

Minimum Relative Distance (Register $1045, bits 5:0)

This register resets to $00 and can be left at its reset value for most applications. The initial relative data interrupt is suppressed until relative motion exceeds this value. If the finger continues to move, subsequent relative data interrupts will occur for any amount of motion. The value of this field may be 0..63, which is interpreted as follows:

0	A relative data interrupt occurs for any amount of relative motion.
1	Suppress relative data interrupts until 2 units of relative motion have occurred.
63	Suppress relative data interrupts until 64 units of relative motion have occurred.

Setting the Minimum Relative Distance to a non-zero value will affect the user's ability to make small controlled movements.

*Max Position (Registers $1046+4*N – $1047+4*N)*

This parameter sets the largest value that the X and Y Position registers are permitted to report, as an unsigned integer from $0002 to $1FFF. The behavior of the device is undefined if Max Position is set to a number less than $0002.

The actual reporting ranges are indicated by the X and Y Max Positions, described above in section 3.1.3. The Max Position control register resets to a device-dependent default.

For devices with a known physical sensor size (such as custom modules), the default Max Position is chosen if possible to cause the default Sensor Resolution to come to around 80 units per millimeter (around 2000 units per inch). For devices that do not know the physical sensor size (such as Synaptics OneTouch products), the Max Position control register generally defaults to $1FFF, the largest representable range.

After any change to the Max Position or Sensitivity Adjust registers for a pad, the data registers for the pad should be considered untrustworthy for up to the next two report periods.

3.1.5. Data registers

Each 2-D sensor in Function $10 has two data sources, the first one reporting the absolute finger position on the sensor, and the second reporting relative finger motion. Both groups of registers are present for every 2-D sensor. If a 2-D sensor in Function $10 supports the ‘Enhanced Gestures’ feature, it will also have a third data source to report the current gesture state for that 2-D sensor.

The host may choose to read a subset of the data source registers if it desires only one kind of data. The host may also choose to set the interrupt enable bit for a subset of the potential data source interrupts if it is only interested in being alerted to the corresponding type of finger activity.

For example, a host that plans to read the absolute data registers could set either the absolute or the relative interrupt enable bit depending on whether it wishes to be interrupted continuously when the finger is present, or only when the finger moves appreciably. A host that was only interested in knowing where the finger was when a tap gesture was performed could enable just the gesture interrupt enable bit. This would prevent a stream of interrupts from the absolute data source if a finger was touching the pad, but not tapping.

The absolute data source for a 2-D sensor has six data registers, as shown in Figure 21.

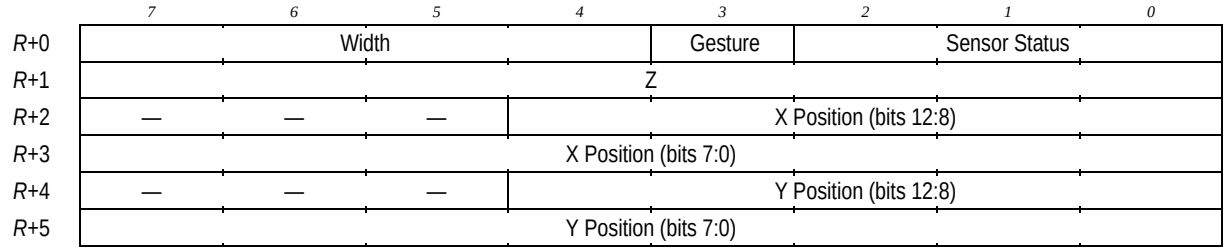


Figure 21. Function $10 data registers for absolute-2-D-position source

The fields of these data registers are defined as follows:

Width (Register R+0, bits 7:4)

This field reports the estimated finger width as an unsigned integer, where 0 represents an extremely narrow finger and 15 represents an extremely wide contact such as a palm laid flat on the sensor. If no finger is touching the pad, Width reports 0. If more than one finger is touching the pad, the value reported for Width is unpredictable.

Not all Synaptics 2-D pads implement finger width detection. For those that do not implement finger width (for example, those that report Has Palm Detect = ‘0’ in the queries), the Width field will report 0 regardless of the kind of finger contact.

Gesture (Register R+0, bit 3)

This bit is '1' if the "virtual mouse button" is currently "clicked" due to a tap, double-tap, or tap-and-slide gesture. Hosts that wish to implement tapping gestures can logically OR this bit into the signal representing that a "mouse button" is currently pressed. Hosts uninterested in tapping gestures, or that do their own tap gesture detection, can safely ignore the Gesture bit.

Sensor Status (Register R+0, bits 2:0)

This field reports the number of fingers present on the sensor. Not all Synaptics 2-D pads implement multiple-finger counting. For those that do not implement finger counting (i.e., those that report Has Multi Finger = '0' in the queries), any number, one or more, of fingers will be reported as Sensor Status '001', and the status values '010' and '011' will never be reported. Multi-finger contact on such devices will usually lead to a larger-than-usual reported Width value.

Sensor Status = '000': No finger.

This value indicates that there is no finger present on the sensor.

Sensor Status = '001': One finger.

This value indicates that there is one finger present on the sensor.

Sensor Status = '010': Two fingers.

This value indicates that there are two fingers present on the sensor.

Sensor Status = '011': Three or more fingers.

This value indicates that there are three or more fingers present on the sensor.

Sensor Status = '100', '101', '110': Reserved.

These values are reserved for use by future versions of RMI Function $10.

Sensor Status = '111': Transitional finger count.

This value indicates that the number of fingers on the pad is indistinct, typically because a second or third finger has just touched down or just lifted from the pad. The X and Y Positions may jump discontinuously during a report period that reports Sensor Status '111'; hosts that do their own relative motion computations based on the reported absolute positions should suppress motion due to these discontinuous jumps.

Some Synaptics products may generate discontinuous jumps in reported position for other reasons; these products may report a Sensor Status of '111' to signal a discontinuous jump in Position even if the reason is unrelated to multiple fingers.

Z (Register R+1)

This field reports the amount of finger contact or finger signal strength, which often serves as a rough estimate of finger pressure. The calculation of Z is inherently approximate; actual reported Z values will vary from one pad to another and from one user to another. In fact, because capacitance is influenced by environmental effects such as the moisture of the skin, Z measurements can even vary from day to day for the same device and user.

Although Z is used to determine finger presence, if a measure of finger pressure is not required, the Z value can be ignored and the Sensor Status field of the first data byte can be used to determine finger presence using the device's built-in algorithms.

When Z = 0, the position cannot be measured and the X and Y Position registers are left unchanged. By default Z is taken as 0 whenever the device's built-in algorithms determine that no finger is present. If the No Clip Z bit of register $1041 is set to '1', then positive but very small Z values will be reported when the device measures a faint capacitance signal; in this case, the position will be reported but it may not be very accurate.

X Position (Register R+2 bits 12:8, Register R+3 bits 7:0)

This field reports the horizontal position of the finger on the pad. It always reports a value between $0000 and the value of the X Max Position query, inclusive. If the X Position is $0000, the finger is somewhere to the left of the leftmost extreme of the sensitive area. If the X Position is equal to X Max Position, the finger is somewhere to the right of the rightmost extreme. (Not all 2-D sensors are capable of sensing these extreme values; those that cannot, will never report an X Position equal to $0000 or X Max Position.) If the X Position is a value in the range from $0001 to (X Max Position – 1), it indicates the position of the finger within the sensor active area, where $0001 represents the left extreme and (X Max Position – 1) represents the right extreme.

Y Position (Register R+4 bits 12:8, Register R+5 bits 7:0)

This field reports the vertical position of the finger on the pad. It always reports a value between $0000 and the value of the Y Max Position query, inclusive. If the Y Position is $0000, the finger is somewhere below the bottommost extreme of the sensitive area. If the Y Position is equal to Y Max Position, the finger is somewhere above the uppermost extreme. (Not all 2-D sensors are capable of sensing these extreme values; those that cannot, will never report a Y Position equal to $0000 or Y Max Position.) If the Y Position is a value in the range from $0001 to (Y Max Position – 1), it indicates the position of the finger within the sensor active area, where $0001 represents the bottommost extreme and (Y Max Position – 1) represents the uppermost extreme.

When no finger is present on the sensor, the X and Y Positions report the last known valid finger position.

3.1.5.1. Relative data registers

The relative data source for a 2-D sensor has two data registers, as shown in Figure 22.

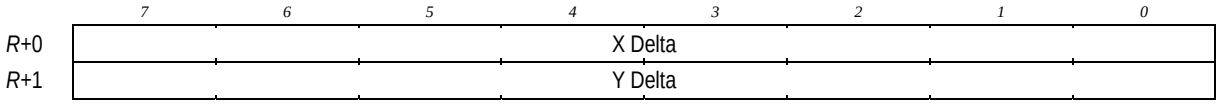


Figure 22. Function $10 data registers for relative-2-D-motion source

These data registers are applicable only for the first finger. These data registers are defined as follows:

X Delta (Register R+0)
This byte reports the amount of horizontal finger motion during a reporting period, as a signed 8-bit integer where a positive X Delta represents rightward motion (toward increasing absolute X Positions).

Y Delta (Register R+1)
This byte reports the amount of vertical finger motion, where a positive Y Delta represents upward motion (toward increasing absolute Y Positions).

The delta registers accumulate motion until the host reads the delta registers. The motion accumulators report +127 or –128 if so much motion occurs between host reads that the registers overflow their 8-bit signed range.

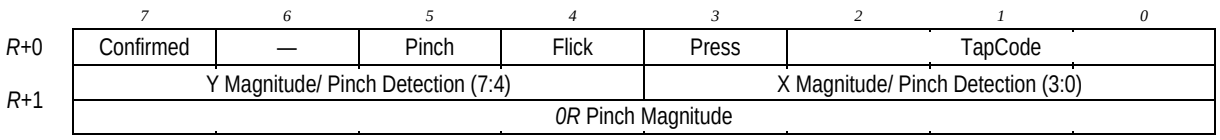
Note: The device may implement the relative motion accumulators as "sticky" 8-bit signed integers, which hold at +127 or –128 once reaching either of those values until they are cleared by reading. Or, the accumulators may be implemented as larger accumulators whose values are clipped to +127 or –128 when expressed in the data registers; in this case, they will not appear "sticky" if an overflowing motion is countered by a reverse motion before the host reads the data registers.

Any read transaction that reads at least one register of the relative data source (those registers shown in Figure 22) clears the X and Y Delta accumulators. Thus, reading relative data registers twice in rapid succession is likely to return $00 deltas on the second read. Note that this means the X and Y deltas must be read in a single multi-byte read transaction; if separate one-byte read transactions were used for these two bytes, the second delta byte would be cleared to $00 before it was read.

3.1.5.2. Gesture data registers

Note: Some sensors do not support gestures. Consult the Product Specification to find out which gestures, if any, are supported by the device.

The gesture data source for a 2-D sensor has two data registers, as shown below.



Function $10 data registers for a gesture data source

These data registers are applicable only for the first finger, and are defined as follows:

Confirmed (Register R+0, bit 7)

This bit is '1' to indicate that a gesture is confirmed. This is helpful to hosts that either lack or do not want to use the resources to perform timing since some gestures occur over a period of time. When the Confirmed bit is '1' it indicates that all criteria necessary for this gesture (including timing) have been met. Once the confirmed flag becomes '1' it will remain set until one of the following occurs:

- In the case of a single tap, double tap or a flick gesture, the gesture data source is read.
- A new gesture occurred (in this case the host was very slow to respond to an interrupt).

If register R+0 is non-zero, but the Confirmed bit is '0,' it indicates that a gesture has started and been initially identified, but may evolve into a different gesture after additional finger activity or time.

Pinch (Register R+0, bit 5)

This bit is '1' if the sensor has detected a *pinch* gesture. A pinch gesture is defined to be a simultaneous two-fingered gesture involving a user's thumb and forefinger, where the two fingers are either moving towards each other, or moving away from each other. If the Pinch bit is '1', the direction and magnitude of the pinch is reported in the Pinch Magnitude field of the second data register (see below). When a pinch occurs, the Pinch bit will become '1' and will remain set until one of the following occurs:

- The gesture data source is read.
- A new gesture occurs (in this case the host was very slow to respond to an interrupt).

A pinch can only occur when exactly two fingers are present. The Pinch Magnitude field contains pinch information only when the Pinch flag is set. The Confirmed flag is always set for this gesture.

Flick (Register R+0, bit 4)

This bit is '1' if the sensor has detected a *flick* gesture. A flick gesture is defined to be a rapid finger press and release, similar to a tap, but with a significant change in position between the press location and release location. If the Flick bit is '1', the direction and magnitude of the flick is reported in the X & Y Magnitude fields of the second data register. When a flick occurs the flick bit will become '1' and remain set until one of the following occurs:

- The gesture data source is read.
- A new gesture occurred (in this case the host was very slow to respond to an interrupt).

A flick can only occur when a single finger is detected. The X and Y Magnitude fields contain flick information only when the Flick flag is set. The Confirmed flag is always set for this gesture.

Press (Register R+0, bit 3)

This bit is '1' if the sensor has detected a *press* gesture. A press gesture is defined to be a finger that lands, and then stays in the same approximate position for approximately 500ms or longer (a period significantly greater than would be expected for a tap gesture). When a press occurs the Press bit will become '1' and remain set until one of the following occurs:

- The finger moves.
- The finger is lifted.

The Confirmed flag is always set for this gesture.

TapCode (Register R+0, bits 2:0)

This field encodes several tap-related gestures, as follows:
0 = No gesture
1 = Single Tap
2 = Tap and Hold (Tap and Slide)
3 = Double Tap
4 – 7 = Reserved

TapCode 1 (Single Tap)

This code indicates that the sensor has detected a *tap* gesture. A tap gesture is defined to be a rapid finger press and release with minimal position change while the finger is touching the sensor. When a tap occurs the TapCode will become '1' and will remain so until:

- The gesture changes (for example to a tap and hold gesture).
- The gesture data source is read after Confirmed is '1,' in which case both Confirmed and TapCode revert to '0.'
- A new gesture occurred (in this case the host was very slow to respond to an interrupt).

TapCode 2 (Tap and Hold)

This code indicates that the sensor has detected a *tap and hold* gesture. A tap and a hold gesture is defined to be a tap event that is rapidly followed by a finger press-and-hold. When a tap and hold occurs, the TapCode will become '2' and remain so until one of the following occurs:

- The gesture changes (for example to a double tap).
- The finger is lifted.
- A new gesture occurred (in this case the host was very slow to respond to an interrupt).

TapCode 3 (Double Tap)

This code indicates that the sensor detected a *double tap* gesture. A double tap gesture is defined to be two taps that occur in rapid succession. When a double tap occurs the TapCode will become '3' and will remain so until one of the following occurs:

- The gesture data source is read after Confirmed is '1,' in which case both Confirmed and TapCode revert to '0.'
- A new gesture occurred (in this case the host was very slow to respond to an interrupt).

Y Magnitude and X Magnitude (Register R+1, bits 7:4 and 3:0), or Pinch Magnitude (bits 7:0)

These fields report the X and Y magnitudes of the current flick gesture or the Pinch Magnitude of the current Pinch gesture, if there is one. These fields report values of $00 whenever there is no flick or Pinch gesture in effect. The values reported in these fields are signed relative motion rather than an actual distance. As the names suggest the Y Magnitude field indicates the component of the flick that was in the vertical direction, while the X Magnitude field indicates the component of the flick that was in the horizontal direction. The Pinch Magnitude is an 8-bit signed number which indicates the magnitude of the Pinch In (negative values) or Pinch Out (positive values) gesture.

Both X and Y Magnitudes report values in the range –7..+7, which are described below:

- Flick
  - –7 a fast flick along the negative X or Y axis.
  - –1 a slow flick along the negative X or Y axis.
  - +1 a slow flick along the positive X or Y axis.
  - +7 a fast flick along the positive X or Y axis.

Pinch Magnitude reports values in the range –128..+127, which are described below:

- Pinch
  - –128 pinching out very quickly.
  - –1 pinching out very slowly.
  - +1 pinching in very slowly.
  - +127 pinching in very quickly.

The host may make its own interpretation of the meaning of pinch out and pinch in to suit its particular needs. Some possible meanings might be:

- Pinch out = increase the volume.
- Pinch in = decrease the volume.

Or:

- Pinch in = make the image smaller.
- Pinch out = make the image larger.

3.1.6. Interrupt Requests

The absolute data source of a 2-D sensor asserts an interrupt request on every report period for which at least one finger is present, or when a finger is sufficiently close so that Z > 0. In the default mode, Z > 0 only if a finger is present and thus Sensor Status is non-zero. However, if the No Clip Z mode bit is set, Z > 0 may merely indicate a finger proximity that is not enough for Sensor Status to report as a true finger. The absolute data source also asserts an interrupt request on any report period for which Z changes to zero from a non-zero value (for example, when the finger lifts off the pad).

The relative data source of a 2-D sensor asserts an interrupt request on every report period that adds a non-zero amount of motion to the X Delta and/or Y Delta motion accumulators. Note that because interrupt request bits are sticky, an interrupt request will remain at ‘1’ even if a later backward finger motion subtracts the deltas down to zero again by the time the host reads the deltas; therefore, the host should be prepared occasionally to see ($00, $00) deltas even when an interrupt request is reported. Conversely, some finger motions might not generate X and Y Deltas in some products; for example, finger motion in a virtual scroll zone will generate deltas on the associated scroller instead of on the 2-D pad’s relative data source.

3.2. Function $13: Scroller

Function $13 implements a purely relative scrolling function, typically attached to a 2-D TouchPad™ (Function $10) sensor. However, Function $13 could be used to describe any input device, such as a mechanical wheel, that expresses relative scroll-like motions but not absolute position.

3.2.1. Number of scrollers

A Function $13 device may include any number of scrollers. The scrollers are identified by consecutive numbers starting with scroller #0. Each scroller has one data source.

The Num Data Sources field of register $0313 expresses how many distinct scrollers are present in the device. If Function $13 has more than seven scrollers, its register $0313 will be $8E, indicating that the standard queries are unable to express the number of scrollers (see section 2.7.8). Tools that operate a device with more than seven scrollers must rely on information beyond the device's own queries to interpret the device.

3.2.2. Register page layout

Function $13 implements the Standard Function Page Layout, as shown in Table 6.

Address range	*Purpose*
$1300	Function Version query
$1301	General Scroller Properties query
$1302–$1309	Scroller-specific Properties queries
$130A–$133F	*Reserved for future definition*
$1340	*Reserved for future command register*
$1341–$1348	Sensor-specific control registers
$134A–$137F	*Reserved for future control registers*
$1380–$13FF	*Reserved for future definition*

Table 6. Function $13 register page

3.2.3. Query registers

Register $1300 reports the version number of the Function $13 specification that the device implements.

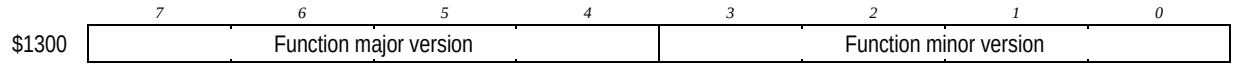


Figure 23. Function $13 Version query register

Register $1301 contains general information about the scroller function.

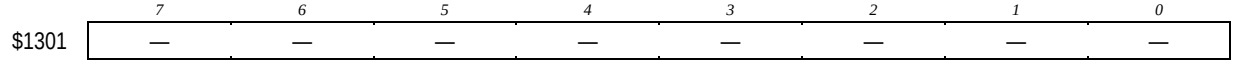


Figure 24. Function $13 General Scroller Properties query register

All bits of this query are *reserved*.

Registers $1302 through $1309 contain specific information about up to eight scrollers. Register $1302 + *N* describes scroller #*N*. If the device has fewer than eight scrollers, the higher-numbered queries in this group are reserved. If the device has more than eight scrollers, the higher-numbered scrollers might not be described by queries.

	7	6	5	4	3	2	1	0
$1302	—	—	—	—	—	—	—	—

Figure 25. Function $13 per-scroller query registers (address shown for scroller #0)

In the present version of Function $13, all bits of this query are *reserved*.

3.2.4. Control registers

Registers $1341–$1348 control the operation of up to eight Scroller sensors. For devices with more than eight Scroller sensors, the higher-numbered Scroller sensors are configured in a device-dependent way. For devices with fewer than eight Scroller sensors, the higher-numbered control registers in this range are unimplemented and reset to $00.

	7	6	5	4	3	2	1	0
$1341	—	—	—	—	—	—	Edge Motion	Tap Enable

Figure 26. Function $13 per-Sensor Scroller Control register

This register resets to $03, and can be left at $03 in most applications. These bits are defined as follows:

Edge Motion (Register $1341 + N, bit 1)

If this bit is set to ‘1’, the edge motion is enabled on the device. When a finger stays at the end zone of the Scroller sensor after a swipe or after a press and hold of some duration, enabling the edge motion will either increase or decrease Scroll Delta value depending on the location of the end zone. For example, when a finger swipes to the left and stays on the left end zone of the Scroller, Scroll Delta will keep decreasing until after the finger is lifted up.

Tap Enable (Register $1341 + N, bit 0)

This bit is set to ‘1’ to enable tapping gesture. When enabled, tapping at the end zone of the Scroller sensor will either increase or decrease the Scroll Delta value, depending on which end zone is being tapped. Tapping on the left end or top end zone will decrease the value of Scroll Delta while tapping on the right end or bottom end zone will increase the Scroll Delta value.

Function $13 also uses the standard RMI interrupt enable control bits (see section 2.6.2) in a special way. For scrollers that report a virtual scrolling zone on a 2-D pad, the interrupt enable bit for the scroller enables the scroll zone itself: If interrupt enable is ‘0’ for the scroller, then the entire 2-D pad area is used for relative motion sensing, but if interrupt enable is ‘1’ for the scroller, then certain finger motions within a designated zone on the 2-D pad generate scrolling deltas instead of relative motion deltas.

3.2.5. Data registers

Function $13 includes one data source for each scroller present in the device. Each data source defines a single data register, as shown in Figure 27:

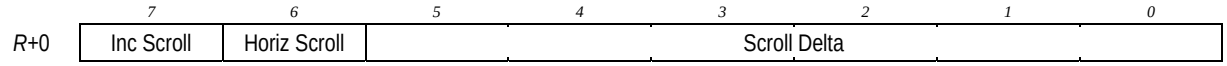


Figure 27. Function $13 data register for scroller

The fields of this data register are defined as follows:

Inc Scroll (Data Register, bit 7)

This bit is '1' if the scrolling is due to an incremental action, such as a tap gesture, a press and hold, a scroll wheel with detents, or an "up" or "down" button. This bit is '0' if the scrolling is due to a continuous action such as a swipe in a virtual scroll zone, scroll strip, or ring.

Some scrollers, such as up/down buttons, will always report incremental motion; others, such basic scroll strips, will always report continuous motion. Other scrollers may generate both kinds of motions in response to different inputs; for example, a scroll strip that enables tapping gestures or EdgeMotion to scroll by increments.

Horiz Scroll (Data Register, bit 6)

This bit is '1' if the scrolling operation specifies horizontal scrolling, or '0' if the operation specifies vertical scrolling or does not specify a direction.

Some scrollers, such as linear scroll strips, will report always horizontal scrolling or always vertical scrolling. Others, such as 2-D pads with right- and bottom-edge virtual scroll zones, are able to report scrolling in either direction in response to different inputs.

Scroll Delta (Data Register, bits 5:0)

This field is a signed 6-bit number in the range –32 through +31 indicating a relative amount of scrolling. A positive value represents "forward" scrolling, such as downward vertical scrolling or rightward horizontal scrolling.

The scale of the scroll delta for incremental scrolling (such as a tap, a tap-and-hold, or one detent of a wheel) is product-dependent. The scale of the scroll delta for continuous scrolling also depends on the particular device.

The scroll delta register accumulates motion until the host reads the scroller data register. The scrolling accumulator reports +31 or –32 if so much scrolling occurs between host reads that the register overflows its 6-bit signed range. The device may implement the scroll delta as being backed by a larger accumulator with a wider range, whose value is clipped to –32 to +31 range for reporting in the data register.

Any read transaction that reads the scroller data register clears the scroller's delta accumulator. Thus, reading the scroll data register twice in rapid succession is likely to return a $00 delta on the second read.

3.2.6. Interrupt Requests

The data source of a scroller asserts an interrupt request on every report period that adds a non-zero amount to the Scroll Delta accumulator. Note that because interrupt request bits are sticky, an interrupt request will remain at ‘1’ even if a later backward scrolling action subtracts the delta down to zero again by the time the host reads the delta; therefore, the host should be prepared occasionally to see a $00 delta even when an interrupt request is reported.

3.3. Function $14: 1-D strip and ring sensors

Function $14 implements one or several one-dimensional touch position sensors, such as Synaptics ScrollStrip™ or ring sensor products.

3.3.1. Number of 1-D sensors

A Function $14 device may include any number of 1-D sensors. The sensors are identified by consecutive numbers starting with 1-D sensor #0.

Each 1-D sensor has two data sources. The first data source reports the absolute finger position, and the second data source reports the relative finger motion.

The Num Data Sources field of register $0314 expresses how many distinct 1-D sensors are present in the device. When Function $14 is present with 1, 2, or 3 1-D sensors, its Num Data Sources value will be 2, 4, or 6, respectively. If Function $14 has more than three 1-D sensors, its register $0314 will be $8E, indicating that the standard queries are unable to express the number of 1-D sensors (see section 2.7.8). Tools that operate a device with more than three 1-D sensors must rely on information beyond the device's own queries to interpret the device.

For example, in a device with two 1-D sensors, source 0 is absolute position for 1-D sensor #0, source 1 is relative motion for 1-D sensor #0, source 2 is absolute position for 1-D sensor #1, and source 3 is relative motion for 1-D sensor #1.

When more than one 1-D sensor is present in the device, all the 1-D sensors operate independently. Each 1-D sensor reports its data in separate data registers with separate interrupt request and interrupt enable bits. Each 1-D sensor's properties are described by a separate set of queries, and each 1-D sensor is configured by a separate set of control registers.

3.3.2. Register page layout

Function $14 implements the Standard Function Page Layout, as shown in Table 7.

Address range	*Purpose*
$1400	Function Version query
$1401	General 1-D Properties query
$1402–$1409	Sensor-specific Properties queries
$140A–$143F	*Reserved for future definition*
$1440	*Reserved for future command register*
$1441	General 1-D Control register
$1442–$1443	*Reserved*
$1444–$1453	Sensor-specific Control registers
$1454–$147F	*Reserved for future control registers*
$1480–$14FF	*Reserved for future definition*

Table 7. Function $14 register page

3.3.3. Query registers

Register $1400 reports the version number of the Function $14 specification that the device implements.

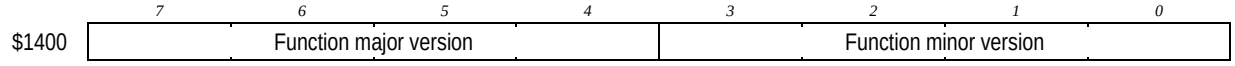


Figure 28. Function $14 Version query register

Register $1401 contains general information about the 1-D sensor function.

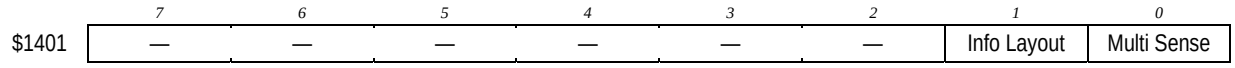


Figure 29. Function $14 General 1-D Properties query register

The fields of this register are defined as follows:

Multi Sense (Register $1401, bit 0)

For RMI devices with more than one 1-D sensor, the Multi Sense bit reports whether the device is able to sense finger contact on more than one 1-D sensor at once. If the Multi Sense bit is '1', each 1-D sensor of Function $14 operates effectively independently; if the Multi Sense bit is '0', only one 1-D sensor at a time will report finger contact. For devices with only one 1-D sensor, the Multi Sense bit is always '0'.

Info Layout (Register $1401, bit 1)

This bit is '1' if the Info Orient and Info Location bits of the various 1-D sensors of Function $14 are reliable. This bit is '0' if the layout and orientation of the sensors is unknown when the device was built, and thus the Info Orient and Info Location bits are not reliable.

Registers $1402–$1409 describe information about each of up to four 1-D sensors. For devices with fewer than four 1-D sensors, the query registers corresponding to higher-numbered 1-D sensors are unused and read as $00. For devices with more than four 1-D sensors, the higher-numbered sensors might not be described by device queries.

The queries for a given 1-D sensor are shown in Figure 30:

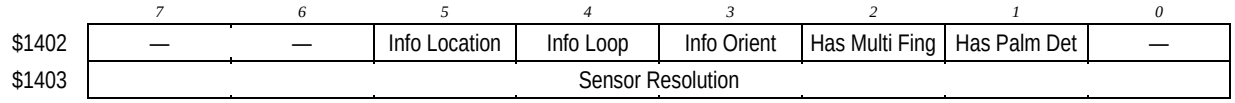


Figure 30. Function $14 per-sensor query register (addresses shown for sensor #0)

The fields of these registers are defined as follows:

*Info Location (Register $1402 + 2*N, bit 5)*

This bit is '1' if this sensor is generally below and maybe to the left of the previous sensor, or '0' if this sensor is generally to the right of the previous sensor. This bit allows the host to roughly determine the arrangement of the sensors, such as a horizontal row, a vertical column, or a 2 × 2 array of sensors. The Info Location bit of 1-D sensor #0 is always '0'.

*Info Loop (Register $1402 + 2*N, bit 4)*

This bit is '1' if the 1-D sensor is a closed loop, or '0' if the sensor is an open-ended strip.

*Info Orient (Register $1402 + 2*N, bit 3)*

For closed-loop sensors, the Info Orient bit is ‘0’ if the loop path is substantially circular, or ‘1’ if the loop path is oval or otherwise non-circular. For open strip sensors, the Info Orient bit is ‘1’ if the strip is substantially vertical, or ‘0’ if the strip is horizontal (or otherwise non-vertical).

The Has Multi Finger and Has Palm Detect bits are defined in the same way as for Function $10, and the Sensor Resolution is computed from Max Position in a way analogous to Function $10. As in Function $10, the Sensor Resolution is $00 for devices with unknown physical dimensions.

3.3.4. Control registers

Registers $1441–$1443 control the operation of the 1-D sensors. When multiple 1-D sensors are present, these registers control all of them identically.

	7	6	5	4	3	2	1	0
$1441	—	—	No Filter	—	—	No Clip Z	No Decel	—
$1442	*Reserved*							
$1443	*Reserved*							

Figure 31. Function $14 General 1-D Control registers

Register $1441 resets to $00, and can be left at $00 in most applications. Its bits are defined in the same way as those of Function $10 register $1041. Note that if a device contains both 2-D pads and 1-D strips, all the pads on the device are controlled together by register $1041, and all the strips on the device are controlled together by register $1441.

Registers $1444–$1453 control the operation of up to four 1-D sensors. For devices with more than four 1-D sensors, the higher-numbered 1-D sensors are configured in a device-dependent way. For devices with fewer than four 1-D sensors, the higher-numbered control registers in this range are unimplemented and reset to $00.

	7	6	5	4	3	2	1	0
$1444	Sensitivity Adjust							
$1445	—	—	—	—	—	—	—	—
$1446	—	—	—	Max Position (bits 12:8)				
$1447	Max Position (bits 7:0)							

Figure 32. Function $14 per-sensor 1-D Control registers (addresses shown for sensor #0)

The Sensitivity Adjust and Max Position registers are defined analogously to the per-sensor control registers of Function $10.

3.3.5. Data registers

Each 1-D sensor of Function $14 has two data sources, the first reporting the absolute finger position and the second reporting relative finger motion.

The absolute data source for a 1-D sensor has four data registers, as shown in Figure 33.

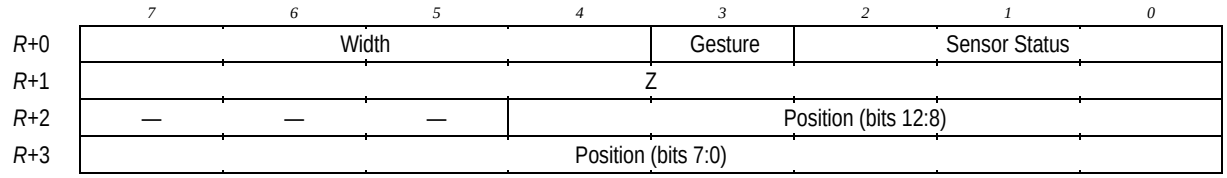


Figure 33. Function $14 data registers for absolute-1-D-position source

The Width, Gesture, Sensor Status, Z, and Position fields are defined analogously to the corresponding data fields of the Function $10 absolute data source.

For strip sensors, the Position is equal to $0000 or Max Position when the finger is past the limits of the sensor area; not all strips will be able to sense these positions. The Position is between $0001 and (Max Position – 1) when the finger is touching the strip. The position $0000 corresponds to the leftmost or bottommost end of the strip if the device is a product to be mounted in a known orientation. For products (such as Synaptics OneTouch) whose orientation is not known by the device firmware, Position $0000 corresponds to the lowest-numbered sensor electrode.

For closed-loop sensors, the Position is equal to $0000 when the finger is at the twelve o'clock position on the loop (for products with a known orientation) or when the finger is over the lowest-numbered sensor electrode (for products with unknown orientation). The Position values range from $0000 to Max Position, inclusive, where the position values $0001 and Max Position are immediately adjacent to position $0000. The Position values increase from $0000 with motion in a clockwise direction (for products with a known sensor electrode layout) or toward higher-numbered sensor electrodes (for products with unknown layout).

The relative data source for a 1-D sensor has one data register, as shown in Figure 34.

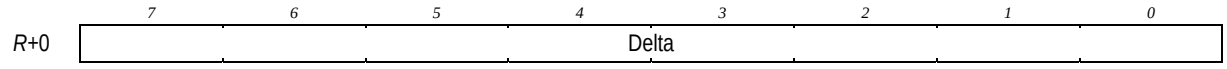


Figure 34. Function $14 data registers for relative-1-D-position source

The 1-D sensor Delta is defined analogously to the corresponding data field of the Function $10 relative data source. A positive Delta represents increasing Position values as defined above. Any read transaction that reads the relative data source register shown in Figure 34 clears the Delta accumulator. Thus, reading the relative data register twice in rapid succession is likely to return a $00 delta on the second read.

3.3.6. Interrupt Requests

Generally speaking, the data sources of a 1-D sensor assert an interrupt request under analogous conditions to those of a Function $10 2-D sensor.

3.4. Function $18: Capacitive buttons

Function $18 implements a group of capacitive buttons.

3.4.1. Number of capacitive buttons

A Function $18 device may include up to 255 capacitive button sensors. The sensors are identified by consecutive numbers starting with button #0.

Regardless of the number of capacitive buttons, Function $18 always has exactly one data source that reports the states of all the buttons. However, the number of data registers in the data source varies depending on the number of buttons, as described below.

3.4.2. Register page layout

Function $18 implements the Standard Function Page Layout, as shown in Table 8.

Address range	*Purpose*
$1800	Function Version query
$1801	Number of Capacitive Buttons query
$1802–$183F	*Reserved for future definition*
$1840	*Reserved for future command register*
$1841–$1842	General Button Control registers
$1843–$187F	*Reserved for future control registers*
$1880–$18FF	*Reserved for future definition*

Table 8. Function $18 register page

3.4.3. Query registers

Register $1800 reports the version number of Function $18 that the device implements.

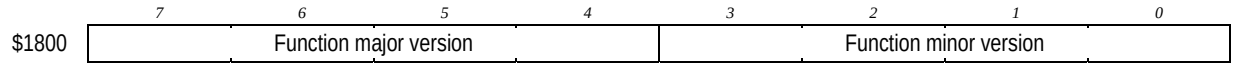


Figure 35. Function $18 Version query register

Register $1801 reports the number of capacitive buttons on the device.

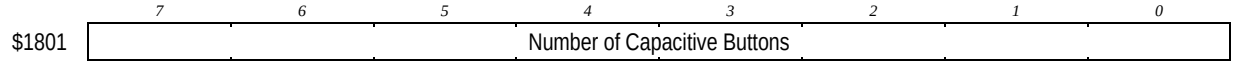


Figure 36. Function $18 Number of Capacitive Buttons query register

The Number of Capacitive Buttons field is an integer from 1 to 255 reporting the number of capacitive buttons present.

3.4.4. Control registers

Registers $1841–$1842 control the operation of the capacitive buttons.

	7	6	5	4	3	2	1	0
$1841	—	Button Usage		Heavy Filtering	—	—	—	—
$1842	Sensitivity Adjust							

Figure 37. Function $18 General Button Control registers

These registers each reset to $00, and can be left at $00 in most applications. Their bits are defined as follows:

Heavy Filtering (Register $1841, bit 4)

Capacitive buttons are always filtered to reduce the effects of electrical noise. If this bit is set to ‘1’, the device applies an extra-heavy filtering algorithm to all the capacitive buttons.

Notes: The majority of button products should never require enabling this feature. For systems that may produce excessive amounts of electrical noise, enabling this feature may help distinguish button events from electrical noise. Enabling the Heavy Filtering feature will modestly impact the responsiveness of the capacitive buttons. This impact may affect the speed at which button presses are registered, the speed at which button releases are registered, and/or the ability to catch fast taps on the buttons.

Button Usage (Register $1841, bits 6:5)

This two-bit field provides guidance about how the capacitive buttons are expected to be used in typical operation of the device:

Button Usage = ‘00’: Unrestricted usage.

This value indicates that the user may touch the buttons in any combination.

Button Usage = ‘01’: Not all buttons at once.

This value indicates that the user might touch multiple buttons at once, but the user would never be expected to touch all the capacitive buttons at the same time during correct operation of the device.

This setting is strictly advisory; the device is permitted to implement button usage ‘01’ in exactly the same way as button usage ‘00’.

However, some devices with more than one capacitive button may use button usage settings ‘01’, ‘10’, and ‘11’ as a cue to reject or recalibrate to capacitance measurements that affect all buttons at once. This cue may help reduce the buttons’ sensitivity to hover, accidental palm contact, and environmental changes such as temperature drift.

When button usage is ‘01’, ‘10’, or ‘11’, the capacitive button data registers may report unpredictable data if all buttons are touched simultaneously. Also, if this simultaneous touch is more than brief, then the button data may remain unpredictable for as long as any buttons remain touched thereafter. However, every Function $18 implementation must be able to recover quickly and resume correct operation once all fingers have been removed from the buttons.

Button Usage = ‘10’: Strongest button only.

This value indicates that the user is expected to touch only one capacitive button at a time. At most one button’s data bit will be reported as ‘1’ in this mode. If the finger is proximate several buttons at the same time, the device will attempt to choose the button with the strongest finger signal.

Button Usage = ‘11’: First button only.

This value indicates that the user is expected to touch only one capacitive button at a time. At most one button’s data bit will be reported as ‘1’ in this mode. If the finger is proximate several buttons at the same time, the device will attempt to choose the first button that was touched for as long as that button remains touched. If the first button touched is lifted while other buttons are still touched, it is undefined which of the still-touched buttons will be reported next.

Button usages ‘10’ and ‘11’ are advisory in the sense that some devices might not fully implement the “strongest button” or “first button” rules. In such devices, button usages ‘10’ and ‘11’ may be treated the same. However, all Function $18 devices must ensure that no more than one button data bit is reported as ‘1’ at the same time whenever the Button Usage field is set to ‘10’ or ‘11’.

For devices with only one capacitive button, the Button Usage field is effectively ignored.

Button Sensitivity Adjustment (Register $1842)

This register is a signed 8-bit byte whose usage is analogous to the sensitivity adjustment of RMI Function $10. It affects all the capacitive buttons of the device together. It can be left at $00 in most cases.

3.4.5. Data registers

Function $18 always has one data source. The number of data registers in the data source varies depending on the number of buttons. If there are 1–8 buttons, Function $18 has one data register. If there are 9–16 buttons, Function $18 has two data registers. In general, if there are *N* buttons, Function $18 has int((*N*+7) / 8) data registers.

Buttons 7–0 are reported in bits 7:0 of the first capacitive button data register. Buttons 15–8 are reported in bits 7:0 of the second data register, buttons 23–16 are reported in the third data register, and so on. In general, button *N* is reported in bit (*N* mod 8) of data register number int(*N* / 8).

Each bit in the capacitive button data registers is ‘1’ if the button is being touched, or ‘0’ if the button is not being touched.

The data registers of a capacitive button sensor are shown in Figure 38, with the example of 18 buttons.

	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
R+0	Btn7	Btn6	Btn5	Btn4	Btn3	Btn2	Btn1	Btn0
R+1	Btn15	Btn14	Btn13	Btn12	Btn11	Btn10	Btn9	Btn8
R+2	—	—	—	—	—	—	Btn17	Btn16

Figure 38. Function $18 data registers for capacitive buttons (for example of 18 buttons)

Note that if a device includes both mechanical and capacitive buttons, these are treated in RMI as two separate Functions with two separate data sources and data registers.

3.4.6. Interrupt Requests

Function $18 asserts an interrupt request on its data source whenever any button bit changes from a ‘0’ to a ‘1’ or from a ‘1’ to a ‘0’. Because an interrupt request is a “sticky” bit, interrupt requests read as ‘1’ if any button bit has changed since the last time the data registers were read, even if the button bit has changed back to its previous value since that time. An interrupt request is asserted synchronously with the report rate of other capacitive sensors in the device.

3.5. Function $20: Digital GPIOs

Function $20 implements a group of general-purpose digital input/output pins, as might be used for input buttons, simple LED control, and so on. The GPIO facility supports two distinct models:

- In the *non-integrated model* of GPIOs, the GPIs and GPOs are treated as completely separate facilities. A device may have only GPIs, or only GPOs, or some of each type. GPI #*N* is usually a different, unrelated chip pin to GPO #*N*, and if a pin is both a GPI and a GPO, its GPI number and GPO number need not be the same. The non-integrated model is well-suited to special-purpose custom modules that want to have straightforwardly numbered mechanical button inputs as well as straightforwardly numbered LED outputs.

  Note: The non-integrated model, as defined, is flexible enough to encompass practically any combination of GPI, GPO, and GPIO functionality. In practice, care should be taken when specifying an RMI device to avoid confusing choices. For example, if a certain pin is both a GPI and a GPO, it is best to assign it equal GPI number and GPO number even though this is not strictly required. Some GPIO combinations may not be supported in a given product line; for example, current Synaptics firmware supports only the integrated model and a “strict” non-integrated model in which each pin is either a GPI or a GPO, never both.

- In the *integrated model* of GPIOs, the GPIs and GPOs are simply the input and output functions of the same set of pins. GPI #*N* reports the input voltage sampled on pin #*N*, and GPO #*N* controls the output voltage driven onto the same pin #*N*. The integrated model is well-suited to general-purpose products that can benefit from maximum flexibility.

3.5.1. Number of GPIOs

A Function $20 device may include up to 240 GPIs and up to 240 GPOs. The GPIs are identified by consecutive numbers starting with GPI #0, and the GPOs are identified separately by consecutive numbers starting with GPO #0. (If the device follows the integrated GPIO model, GPI numbering and GPO numbering coincide; in the non-integrated model, GPI numbering and GPO numbering are completely separate.)

A device might contain only GPIs, in which case the number of GPOs is zero. A device might also have only GPOs, in which case the number of GPIs is zero. (Function $20 itself would be present only if at least one GPI or GPO exists.)

If there are no GPIs, then Function $20 has no data sources. If there are one or more GPIs, Function $20 has exactly one data source that reports the states of all the GPIs. The number of data registers in the data source varies depending on the number of GPIs, as described below.

3.5.2. Register page layout

Function $20 implements the Standard Function Page Layout, as shown in Table 9.

Address range	*Purpose*
$2000	Function Version query
$2001–$2003	General GPIOs queries
$2004–$203F	*Reserved for future definition*
$2040	*Reserved for future command register*
$2041	General GPIO Control register
$2042–$207F	GPO control registers
$2080–$20FF	*Reserved for future definition*

Table 9. Function $20 register page

3.5.3. Query registers

Register $2000 reports the version number of the Function $20 specification that the device implements.

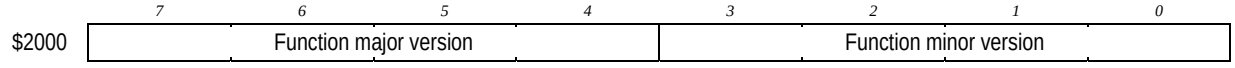


Figure 39. Function $20 Version query register

Registers $2001–$2003 report the number of GPIOs on the device, and their overall properties.

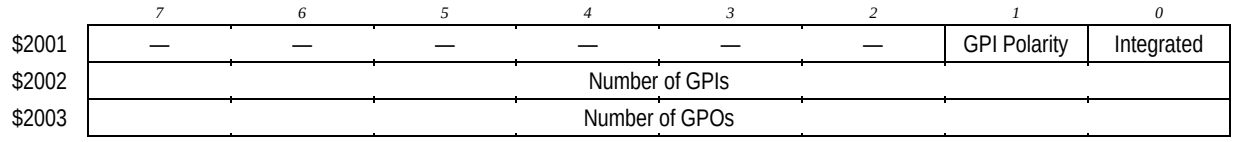


Figure 40. Function $20 General GPIOs query registers

The bits of these registers are defined as follows:

Integrated (Register $2001, bit 0)
This bit is ‘1’ if the GPIOs use the integrated model, or ‘0’ if the GPIOs use the non-integrated model (as defined above).

GPI Polarity (Register $2001, bit 1)
The GPI Polarity bit is ‘0’ if the GPIs use positive polarity as defined in section 3.5.5, or if there are no GPIs. The GPI Polarity bit is ‘1’ if the GPIs use negative polarity.

Number of GPIs (Register $2002)
This field is an integer from 0 to 240 reporting the number of GPIs (for the non-integrated model) or the number of GPIOs (for the integrated model).

Number of GPOs (Register $2003)
This field is an integer from 0 to 240 reporting the number of GPOs (for the non-integrated model). For the integrated model, the Number of GPOs query is unused and reserved.

3.5.4. Control registers

Register $2041 controls the overall operation of the GPIOs.

	7	6	5	4	3	2	1	0
$2041	—	—	—	Debounce	—	—	—	—

Figure 41. Function $20 General GPIO Control register

This register resets to $00, and can be left at $00 in most applications. In the present version of Function $20, this register contains only a Debounce bit:

Debounce (Register $2041, bit 4)
If this bit is set to '1', the device applies a debouncing algorithm to all the GPI inputs. The exact debouncing algorithm is device-specific, but in general debouncing strives to eliminate brief glitches in the input signal due, for example, to "bounce" in the contacts of a mechanical switch. Enabling debouncing will modestly impact the responsiveness of the GPIs; depending on the debouncing algorithm, this impact may affect the speed at which high-to-low transitions are registered, the speed at which low-to-high transitions are registered, and/or the ability to catch fast low or high pulses.

Registers $2042 onward each control the output modes of a group of eight consecutive GPOs. The number of control registers depends on the number of GPOs. Each group of eight GPOs is controlled by a pair of consecutive control registers. In general, if there are *N* GPOs, then there are 2 * int((*N*+7) / 8) GPO control registers. If a device has no GPOs (but Function $20 is still present because there are GPIs), then there are no GPO control registers at all.

Figure 42 shows an example for the case of 19 GPOs.

	7	6	5	4	3	2	1	0
$2042	Dir7	Dir6	Dir5	Dir4	Dir3	Dir2	Dir1	Dir0
$2043	Data7	Data6	Data5	Data4	Data3	Data2	Data1	Data0
$2044	Dir15	Dir14	Dir13	Dir12	Dir11	Dir10	Dir9	Dir8
$2045	Data15	Data14	Data13	Data12	Data11	Data10	Data9	Data8
$2046	—	—	—	—	—	Dir18	Dir17	Dir16
$2047	—	—	—	—	—	Data18	Data17	Data16

Figure 42. Function $20 GPO Output Control registers (for example of 19 GPOs)

The Data*N* and Direction*N* bits together control the state of GPO #*N*, as follows:

Direction*N*	Data*N*	State of GPIO #*N*
0	0	Input mode, high-impedance
0	1	Input mode, with weak pull-up resistor
1	0	Output mode, driving digital '0'
1	1	Output mode, driving digital '1'

Table 10. GPO control settings (integrated-model conventions)

Direction*N*	Data*N*	State of GPO #*N*
0	0	High-impedance state
0	1	Weak pull-up resistor
1	0	Driving digital ‘0’
1	1	Driving digital ‘1’

Table 11. GPO control settings (non-integrated-model conventions)

The GPO Data and Direction control registers may be written individually or in pairs or groups. However, if a single RMI write transaction covers both the Data and Direction registers for a given GPO #*N*, then the two parts of the control state of GPO #*N* will be updated coherently: Register writes may change the Data*N* and Direction*N* bits from any one to any other of the four states in Table 10 without creating a glitch on the pin.

Depending on the device implementation, writing to a GPO control register may have an immediate effect on the controlled GPOs, or the effect may be delayed until the next internal reporting period, typically a few milliseconds.

The reset states of the GPO control registers are up to the device; in many products, they will reset to non-$00 values.

Chip pins that are GPIs but not GPOs are not controllable through the RMI interface. They typically are either permanently in high-impedance mode, or permanently in weak pull-up mode, depending on the design of the particular device.

3.5.5. Data registers

When GPIs are present, Function $20 has one data source. The number of data registers in the data source varies depending on the number of GPIs. If there are 1–8 GPIs, Function $20 has one data register. If there are 9–16 GPIs, Function $20 has two data registers. In general, if there are *N* GPIs, Function $20 has int((*N*+7) / 8) data registers. If there are GPOs but no GPIs, then Function $20 is present with no data sources or data registers.

GPIs 7–0 are reported in bits 7:0 of the first GPI data register. GPIs 15–8 are reported in bits 7:0 of the second data register, GPIs 23–16 are reported in the third data register, and so on. In general, GPI number *N* is reported in bit (*N* mod 8) of data register number int(*N* / 8).

A device may define its GPIs to have either positive or negative polarity. Positive polarity means each GPI data bit follows the digital level on the associated pin, either ‘1’ for a high voltage or ‘0’ for a low voltage. Positive polarity is most appropriate for general-purpose I/Os. Negative polarity means each GPI data bit follows the complement of the digital level on the associated pin. Negative polarity is most appropriate for input pins connected to mechanical buttons with pull-up resistors.

The GPI data registers are shown in Figure 43, with the example of 19 GPIs.

	7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
R+0	GPI7	GPI6	GPI5	GPI4	GPI3	GPI2	GPI1	GPI0
R+1	GPI15	GPI14	GPI13	GPI12	GPI11	GPI10	GPI9	GPI8
R+2	—	—	—	—	—	GPI18	GPI17	GPI16

Figure 43. Function $20 data registers for GPIs (for example of 19 GPIs)

Note that if a device includes both mechanical and capacitive buttons, these are treated in RMI as two separate Functions with two separate data sources and data registers.

Chip pins that are GPOs but not GPIs do not have their states reported through the RMI interface, even if they are set in high-impedance mode.

3.5.6. Interrupt Requests

If GPIs are present, Function $20 asserts an interrupt request on its data source whenever any GPI data bit changes from a ‘0’ to a ‘1’ or from a ‘1’ to a ‘0’. Because an interrupt request is a “sticky” bit, interrupt requests read as ‘1’ if any GPI has changed since the last time the data registers were read, even if the GPI has changed back to its previous value since that time. The device may choose to sample the GPI pins and assert an interrupt request for GPIs either asynchronously, or synchronously with the report rate of the capacitive sensors in the device. In practice, current Synaptics products always sample and report GPIs synchronously.

3.6. Function $22: Simplified LEDs

Function $22 implements LEDs with adjustable brightness, as might be implemented using either pulse width modulation (PWM) or adjustable-current output pins. For brevity, RMI sometimes refers to Function $22 devices as “LEDs.” However, nothing about Function $22 requires that the component being controlled is a light-emitting diode; also, simple LEDs that do not require brightness control might be implemented more easily using Function $20 GPOs.

Depending on the device, chip pins that control Function $22 LEDs may or may not also be accessible as Function $20 GPIOs. A general-purpose device such as Synaptics OneTouch™ might allow the same pin to be controlled via either function; a custom product would typically implement each pin as either an LED, a GPI, or a GPO depending on the pin’s intended purpose.

3.6.1. Number of LEDs

A Function $22 device may include up to 16 brightness-controlled LEDs. The LEDs are identified by consecutive numbers starting with LED #0. The total number of LEDs is reported in query register $2201 (described below).

3.6.2. Register page layout

Function $22 implements the Standard Function Page Layout, as shown below.

Address range	*Purpose*
$2200	Function Version query
$2201–$2202	General LED queries
$2203–$223F	*Reserved for future definition*
$2240	*Reserved for future command register*
$2241	*Reserved for future control register*
$2242–$2243	LED Enable Command register
$2244–$2245	General LED Control registers
$2246–$224F	*Reserved for future control registers*
$2250–$225F	Per-LED control registers
$2260–$22FF	*Reserved for future definition*

Table 12. Function $22 register page

3.6.3. Query registers

Register $2200 reports the version number of the Function $22 specification that the device implements.

	7	6	5	4	3	2	1	0
$2200	Function major version				Function minor version			

Figure 44. Function $22 Version query register

Registers $2201–$2202 report the number of LEDs on the device:

	7	6	5	4	3	2	1	0
$2201	Number of LEDs							
$2202	—	—	—	—	—	—	—	—

Figure 45. Function $22 General LEDs query registers

No bits in register $2202 have defined meanings.

3.6.4. Control registers

Registers $2242 and $2243 control whether or not each LED is enabled. Bit *N* of register $2242 controls the state of LED #*N*, and bit *N* of register $2243 controls the state of LED #(*N* + 8).

	7	6	5	4	3	2	1	0
$2242	LED #7	LED #6	LED #5	LED #4	LED #3	LED #2	LED #1	LED #0
$2243	LED #15	LED #14	LED #13	LED #12	LED #11	LED #10	LED #9	LED #8

Figure 46. Function $22 Enable Control registers

Registers $2242 and $2243 reset to $00. The host should write registers $2242 and $2243 to set the device in the desired state of each LED.

LED Enable (Registers $2242 and $2243)

Writing the LED Enable *N* bit to ‘1’ sets LED #*N* to the Enabled state. The LED will turn on over a specified period of time, or animate in a specified pattern. The amount of LED current corresponding to the “on” state for an LED is set by the Per-LED Control registers.

Writing the LED Enable *N* bit to ‘0’ sets LED #*N* to the Disabled state. The LED will turn off either immediately or over a specified period of time. In RMI Function $22, the “off” state is always represented by zero sink current on the LED pin.

Registers $2244–$2245 control the speed of the ramps and animations of the LEDs.

	7	6	5	4	3	2	1	0
$2244	Ramp Period A							
$2245	Ramp Period B							

Figure 47. Function $22 LED Period Control registers

These registers reset to $00. The fields of these registers are defined as follows:

Ramp Period A (Register $2244)

This field determines the ramp rate for selected LEDs. Ramp Period A is used by any LED whose Pattern is '000', '010', '100', '110', or '111'. The period is specified in multiples of approximately 10 ms, from 0 ms to 2550 ms:

- $00 indicates an instantaneous transition to the new target intensity.
- $01 - $FE specifies multiples of approximately 10 ms. The basic ramp period unit of 10 ms is accurate to ±10%.
- $FF causes the transition to take approximately 2.5 seconds.

Writing to register $2244 while any LED is currently ramping based on Ramp Period A will have an undefined (but not drastic) effect on the LED.

Ramp Period B (Register $2245)

This field determines the ramp rate for selected LEDs. It is defined analogously to Ramp Period A. Ramp Period B is used by any LED whose Pattern is '001', '011', '101', '110', or '111'.

Each LED also has one register that controls the LED specifically. The register for LED #*N* is at address $2250 + *N*:

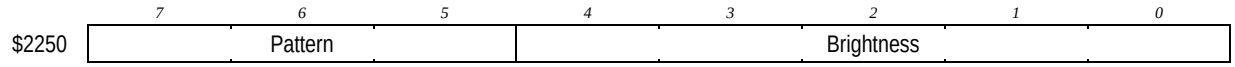


Figure 48. Function $22 per-LED Control register (address for LED #0 shown)

These registers control the intensity of the selected LED:

Brightness (Register $2250, bits 4:0)

The Brightness field indicates the target intensity level when the LED is enabled:

- $00 represents fully off,
- $01 - $1E represent intermediate intensities evenly or approximately-evenly spaced between fully off and fully on, and
- $1F represents fully on.

Pattern (Register $2250, bits 7:5)

The Pattern field takes one of the following values:

Pattern = '000': Rise and fall period A.

In this setting, when the LED is enabled by setting the LED Enable *N* bit in register $2242 or $2243, the LED ramps to the intensity indicated by the Brightness field over a time determined by Ramp Period A (register $2244) and then holds at that intensity. When the LED is disabled by clearing the LED Enable *N* bit, the LED ramps down to the "off" state over a time determined by Ramp Period A and then remains "off."

The ramp begins immediately after the write to the LED Enable control registers or the Per-LED Control register, and may be unsynchronized (out of phase) with other ongoing ramps or animations. To ramp several LEDs synchronously, write all the LED Enable bits at once in the same RMI write transaction.

Pattern = '001': Rise and fall period B.
This pattern is like pattern '000', except that the Ramp Period B parameter (register $2245) determines the ramp rate instead of Ramp Period A.

Pattern = '010': Rise period A, fast fall.
In this setting, when the LED is enabled the LED ramps to the intensity indicated by the Brightness field over a time determined by Ramp Period A and then holds at that intensity. When the LED is disabled, the LED switches immediately to the "off" state.

Pattern = '011': Rise period B, fast fall.
This pattern is like pattern '010', except that the Ramp Period B parameter (register $2245) determines the ramp rate instead of Ramp Period A.

Pattern = '100': Fast rise, fall period A.
In this setting, when the LED is enabled the LED switches immediately to the intensity indicated by the Brightness field and then holds at that intensity. When the LED is disabled, the LED ramps down to the "off" state over a time determined by Ramp Period A and then remains "off."

Pattern = '101': Fast rise, fall period B.
This pattern is like pattern '100', except that the Ramp Period B parameter (register $2245) determines the ramp rate instead of Ramp Period A.

Pattern = '110': Ramping animation.
In this setting, when the LED is enabled the LED ramps to the intensity indicated by the Brightness field over a time determined by Ramp Period A, and then holds at that intensity for a time determined by Ramp Period B. The LED then ramps down to the "off" state over Ramp Period A and remains off for Ramp Period B. This cycle repeats continuously for as long as the LED is enabled.

When the LED is disabled, the LED switches immediately to the "off" state.

Pattern = '111': Pulsed animation.
In this setting, when the LED is enabled the LED pulses between the "on" and "off" states. The LED switches immediately to the intensity indicated by the Brightness field, then remains at the target intensity for a time determined by Ramp Period B. Then the LED switches immediately to the "off" state and remains "off" for a time determined by Ramp Period A. This cycle repeats continuously for as long as the LED is enabled.

When the LED is disabled, the LED switches immediately to "off" state.

Function $22 interacts with the power management features described in section 2.6.1. The device will be unable to doze if any ramping or animation is ongoing. Also, for devices that use pulse width modulation (PWM) to control LED brightness, the device will not doze if any LED is resting at an intensity other than “off” or fully on (Brightness = $1F). Devices that use adjustable current to control LED brightness will be able to doze when LEDs rest at intermediate intensities, but the dozing power usage of the device may be considerably higher when any LED is at any intensity other than “off.”

3.6.5. Data registers

RMI Function $22 has no data sources or data registers.

3.6.6. Interrupt Requests

Because RMI Function $22 has no data sources, it has no interrupt request state and does not affect the Attention signal.

4. Standard RMI physical layers

RMI is defined so that it may be implemented atop a variety of physical interfaces.

RMI is defined for three physical layers:

- RMI-on-I^{2}C. See section 4.1.
- RMI-on-SMBus: See section 4.2.
- RMI on four-wire SPI: See section 4.3.

4.1. I^{2}C physical interface

Synaptics RMI-on-I^{2}C devices are suitable for connecting directly to an industry-standard I^{2}C host interface. This section describes the I^{2}C physical layer. The RMI-on-I^{2}C interface has been developed using version 2.1 of *The I^{2}C Bus Specification*, dated January 2000. This document can be found at http://www.nxp.com/. The remainder of this section assumes that the reader has familiarity with *The I^{2}C Bus Specification* document.

4.1.1. I^{2}C transfer protocols

To communicate with an RMI-on-I^{2}C device, a host needs to be able to:

1. Read one or more RMI registers starting from some RMI register address.
2. Write one or more RMI registers starting from some RMI register address.

The I^{2}C bus specification imposes no limit to the number of registers that the host can read in a single transfer. However, RMI does not permit a physical layer transfer to cross a page boundary, so the practical limit on the maximum length of a transfer would be the number of registers in an RMI address page, or 256.

The I^{2}C bus specification imposes no limit on how long a transfer can take, or how slowly a bus master is allowed to clock the bus. To meet this specification, RMI devices will not impose any timeouts during any I^{2}C transfer.

4.1.1.1. I^{2}C transfer details

The following terms are used in the definition of the I^{2}C transfer protocols:

S	Indicates an I^{2}C Start event
P	Indicates an I^{2}C Stop event
Sr	Indicates an I^{2}C Repeated Start event
A	Indicates an I^{2}C ACK bit
N	Indicates an I^{2}C NAK bit

SlaveAddr	The 7-bit Slave Address field in an I²C header byte
Wr	The 1-bit 'write' field in an I²C header byte (a Write always has the value 0)
Rd	The 1-bit 'read' field in an I²C header byte (a Read always has the value 1)

4.1.2. RMI-on-I²C register addressing

RMI defines a 15-bit RMI register address space. When RMI is implemented using the I²C physical layer, the size of the I²C Command Code effectively limits the size of a register address to 8 bits. As a result, RMI-on-I²C devices define a *Page Select register* to supply the upper 7 bits of the 15-bit RMI address, while the Command Code in each I²C transfer supplies the lower 8 bits of the 15-bit address.

To make most efficient use of the I²C paged addressing scheme, Synaptics RMI-on-I²C devices define that for all commonly used RMI device registers, there will be a duplicate *aliased* register located at a new RMI address. The entire set of aliased register addresses are grouped into a single page of the RMI address space. This enables user software to access all commonly used RMI registers without ever having to rewrite the Page Select register.

The aliased addresses occupy page \$04*xx* in the general RMI address map. At reset, all RMI-on-I²C devices initialize their Page Select register to the value \$04. This means that by default, all I²C register accesses will access the RMI aliased address space.

For example: After a device reset, the Page Select register is defined to default to page address \$04. If the host sends an RMI-on-I²C device a Read Byte command to write address \$F0, the device will access page \$04 at offset \$F0, or address \$04F0. From the table below, it can be seen that the aliased address \$04F0 corresponds to the unaliased address \$0000, or the RMI Device Control register.

The aliased address space takes the following general form:

Aliased Address	*General RMI Address*	*Register Groups*
\$0400 - \$041F	\$0400 - \$041F	RMI Data Registers and Device Status register (actual addresses are product dependent). See section 2.5.3 for a description of the Device Status register.
\$0420 - \$043F	\$*xx*40 - \$*xx*5F	Command/Control/Status registers for the first RMI function.
\$0440 - \$045F	\$*xx*40 - \$*xx*5F	Command/Control/Status registers for a second RMI function.
\$0460 - \$047F	\$*xx*40 - \$*xx*5F	Command/Control/Status registers for a third RMI function.
\$0480 - \$049F	\$*xx*40 - \$*xx*5F	Command/Control/Status registers for a fourth RMI function.
\$04E0 - \$04E7	\$0200 - \$0207	RMI Product ID queries
\$04F0 - \$04F4	\$0000 - \$0004	RMI Control, Command, and General Status registers
\$04FF	\$*xx*FF	Page Select register (default value is \$04)

Table 13. General RMI Aliased Address Map

Note: The exact Aliased Address Space address assignments are product-dependent. All products are shipped with documentation that describes their specific address map. See section 4.4 for a sample Product Address Map.

4.1.3. Block read operations

The Block Read operation allows a host to read one or more RMI registers starting from a specified RMI address. The device starts reading from the RMI address specified by the read operation, and continues to send registers from consecutively incrementing RMI addresses until the host finally NAKs the transfer.

Below is an example of a Block Read operation, where the host reads 1 RMI register from address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Sr	Slave Addr	Rd	A	Register N	N	P

Below is an example of a Block Read operation where the host reads 4 consecutive RMI registers starting from address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Sr	Slave Addr	Rd	A	Register N	A	Register N+1	A

Register N+2	A	Register N+3	N	P

It is not permitted to perform an I^2C read operation that is not preceded by an I^2C write of the low-order 8 bits of the RMI register address.

4.1.3.1. Repeated starts

The Repeated Start separating the write of the RMI register address from the read of the register data ensures correct operation on an I^2C bus that supports multiple bus masters. For a host bus that either does not require multi-master support or cannot generate Repeated Start events, RMI-on-I^2C devices permit a host to replace a Repeated Start with a Stop event followed by a Start event.

4.1.4. Block write operations

The Block Write operation allows a host to write one or more RMI registers starting at a specified RMI address. The device starts writing data to the RMI address specified by the write operation, and continues to write registers to consecutively incrementing RMI addresses as long as the host keeps sending data. An RMI device will ACK every byte that it receives.

Below is an example of a Block Write operation where the host writes data to a single RMI register at address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Register N	A	P

Below is an example of a Block Write operation where the host writes 3 consecutive RMI registers starting with the register at address N:

S	SlaveAddr	Wr	A	Low 8 bits of register addr N	A	Register N	A	Register N+1	A	Register N+2	A	P

4.1.5. Synaptics module I²C protocol compliance

The Synaptics I²C interface is designed to comply with the basic I²C protocol as described in the *I²C Bus Specification*, Version 2.1 by Philips. Conforming to this specification ensures the following:

- Synaptics modules correctly recognize and respond to Start events, Repeated Start events, and Stop events.
- Synaptics devices properly generate SCL "clock stretching" as a slave device.
- Synaptics modules support the 7-bit addressing mode.

4.1.5.1. Addressing modes

Synaptics devices do not support the 10-bit addressing extension to I²C. Synaptics Master-Slave modules master their transmissions to a host device using 7-bit addresses.

Synaptics modules can co-exist on the same I²C bus with other devices that support the 10-bit extended addressing mode. In addition, while Synaptics Slave-Only modules have 7-bit addresses, they can respond as slave device to a host/master that has a 10-bit address.

4.1.5.2. Data rate and clock stretching

Synaptics I²C devices can maintain either the "Fast Mode" data rate in the *I²C Bus Specification* at 400K bits per second, or the "Normal Mode" data rate of 100K bits per second while a byte is being clocked over the bus. In either case, an RMI device may be required to perform an I²C Clock-Stretch operation in between the bytes of a transfer.

4.1.6. I²C electrical compliance

Synaptics I²C modules meet the electrical specifications of both standard mode (100K bits per second) and fast mode (400K bits per second) except as described in the following sections. For more information on those aspects, see the *I²C Bus Specification*.

4.1.6.1. I^2C bus V_{DD}

In typical host systems, the I^2C bus V_{DD} will be the same as the Synaptics ASIC V_{DD}. However, the *I^2C Bus Specification* allows the I^2C bus V_{DD} to be different than the V_{DD} level of the ASICs that are connected to the bus. The Synaptics ASIC implementation clamps the I^2C bus signals SDA and SCL to the Synaptics ASIC V_{DD} through some ESD (electrostatic discharge) protection diodes.

This leads to a few system implications if the I^2C bus V_{DD} does not match the Synaptics ASIC V_{DD}:

- The I^2C bus V_{DD} must be no greater than the Synaptics ASIC V_{DD}.
- The host system must supply the pullup resistors from SDA and SCL to the I^2C bus V_{DD}.

4.1.6.2. Powering down a Synaptics I^2C device

If V_{DD} is removed from a Synaptics I^2C device to power it down, the ESD diodes built into the SDA and SCL pins of the Synaptics device will clamp those I^2C bus signals to ground. This means that all I^2C communication on that bus segment will cease until power is restored to the Synaptics module. This will only be an issue if a host desired to power down a Synaptics RMI device without powering down the other I^2C devices on the same bus.

4.1.6.3. I^2C input levels

The V_{IH}/V_{IL} levels used by the Synaptics ASIC are referenced to its own V_{DD}, not to the I^2C bus V_{DD}. The Synaptics ASIC input thresholds are 1.8V-compatible if the ASIC V_{DD} supply is below 3.0V. The ASIC input thresholds are TTL-compatible if the ASIC V_{DD} supply is between 3.0V – 5.5V.

4.1.6.4. Fast mode hysteresis

The *I^2C Bus Specification* requires a hysteresis of 10% of bus V_{DD} if bus V_{DD} is less than 2.0V. In the Synaptics implementation, the fast-mode Schmitt-trigger hysteresis is 5% of ASIC V_{DD}.

4.1.6.5. Noise rejection filters

The Synaptics SDA and SCL inputs do not implement the fast-mode spike-suppression filters precisely as described in the *I^2C Bus Specification*. Instead, the Synaptics implementation of the I^2C SDA and SCL inputs use different mechanisms to provide comparable noise rejection capabilities.

4.2. SMBus physical interface

This section describes the SMBus physical layer. (The version of the physical layer of an RMI device can be found by reading register $0202, described in section 2.7.3.). The RMI-on-SMBus interface has been developed using version 2.0 of the *System Management Bus (SMBus) Specification*, dated August 3, 2000. This document can be found at http://www.smbus.org/.

Synaptics RMI-on-SMBus devices are suitable for connecting directly to an industry-standard SMBus host interface. The SMBus transaction protocols supported by the Synaptics RMI-on-SMBus interface are defined in section 4.2.3.

4.2.1. RMI-on-SMBus register addressing

RMI defines a 15-bit RMI register address space. When RMI is implemented using the SMBus physical layer, the size of the SMBus Command Code effectively limits the size of a register address to 8 bits. As a result, RMI-on-SMBus devices define a *Page Select register* to supply the upper 7 bits of the 15-bit RMI address, while the Command Code in each SMBus transfer supplies the lower 8 bits of the 15-bit address.

To make most efficient use of the SMBus paged addressing scheme, Synaptics RMI-on-SMBus devices define that for all commonly used RMI device registers, there will be a duplicate *aliased* register located at a new RMI address. The entire set of aliased register addresses are grouped into a single page of the RMI address space. This will enable user software to access all commonly used RMI registers without ever having to rewrite the Page Select register.

The aliased addresses occupy page $04*xx* in the general RMI address map. At reset, all RMI-on-SMBus devices initialize their Page Select register to the value $04. This means that by default, all SMBus register accesses will access the RMI aliased address space.

For example: After a device reset, the Page Select register is defined to default to page address $04. If the host sends an RMI-on-SMBus device a Read Byte command to write address $F0, the device will access page $04 at offset $F0, or address $04F0. From the table below, it can be seen that the aliased address $04F0 corresponds to the unaliased address $0000, or the RMI Device Control register. The aliased address space takes the following general form:

Aliased Address	*General RMI Address*	*Register Groups*
$0400 - $041F	$0400 - $041F	RMI Data Registers and Device Status register (actual addresses are product dependent). See section 2.5.3 for a description of the Device Status register.
$0420 - $043F	$*xx*40 - $*xx*5F	Command/Control/Status registers for the first RMI function.
$0440 - $045F	$*xx*40 - $*xx*5F	Command/Control/Status registers for a second RMI function.
$0460 - $047F	$*xx*40 - $*xx*5F	Command/Control/Status registers for a third RMI function.
$0480 - $049F	$*xx*40 - $*xx*5F	Command/Control/Status registers for a fourth RMI function.
$04E0 - $04E7	$0200 - $0207	RMI Product ID queries
$04F0 - $04F4	$0000 - $0004	RMI Control, Command, and General Status registers
$04FF	$*xx*FF	Page Select register (default value is $04)

Table 14. General RMI Aliased Address Map

Note: The exact aliased address space address assignments are product-dependent. All products are shipped with documentation that describes their specific address map. See section 4.4 for a sample Product Address Map.

4.2.2. Page Select register

The Page Select register is defined to exist at address $*xx*FF in every page of the general 15-bit RMI address space. This means that the Page Select register can be accessed as the last register on every page in the entire RMI address space.

The Page Select register is a control register that supplies the upper 7 bits of the 15-bit RMI address. The default value of the Page Select register is $04. This means that by default, SMBus transfers will access the aliased address space. Since all of the important RMI registers have been defined to exist on page $04*xx*, typical SMBus hosts can treat RMI-on-SMBus devices as though they only have 8-bit addresses.

4.2.3. SMBus transfer protocols

To communicate with an RMI-on-SMBus device, a host needs to be able to do the following things:

1. Read one or more RMI registers starting from some RMI register address.
2. Write one or more RMI registers starting from some RMI register address.

RMI-on-SMBus supports the standard SMBus transfer protocols to read and write bytes, and to read and write words. Since all RMI registers are 8-bit registers, reading or writing a word really means to read or write a pair of sequential RMI byte-wide registers. The following subsections describe the SMBus read and write commands, using the notation found in the *System Management Bus (SMBus) Specification*, Version 2.0 of August 3, 2000.

4.2.3.1. SMBus Byte Write

The SMBus *Byte Write* command writes a byte to a single register in an RMI-on-SMBus device. In its general form, the SMBus *Byte Write* command has this format:

S	SlaveAddr	Wr	A	Command Code	A	Data Byte	A	P

- The *Command Code* is the address of the RMI register to be written.
- The *Data Byte* is the value to write to the RMI register.

4.2.3.2. SMBus Byte Read

The SMBus *Byte Read* command reads the contents of single register in an RMI-on-SMBus device. In its general form, the SMBus *Byte Read* command has this format:

S	SlaveAddr	Wr	A	Command Code	A	Sr	Slave Addr	Rd	A	Data Byte	N	P

- The *Command Code* is the address of the RMI register to be read.
- The *Data Byte* is the value that was read from the RMI register.
- All SMBus Read operations terminate with a NAK bit followed by a STOP bit.

4.2.3.3. SMBus Word Write

The SMBus *Word Write* command writes a pair of bytes to a sequential pair of registers in an RMI-on-SMBus device. In its general form, the SMBus *Word Write* command has this format:

S	SlaveAddr	Wr	A	Command Code	A	Data Byte 0	A	Data Byte 1	A	P

- The Command Code is the address of the first RMI register to be written.
- Data Byte 0 will be written to the RMI Command Code address.
- Data Byte 1 will be written to the RMI Command Code address+1.

4.2.3.4. SMBus Word Read

The SMBus *Word Read* command reads the contents of a sequential pair of registers in an RMI-on-SMBus device. In its general form, the SMBus *Word Read* command has this format:

S	SlaveAddr	Wr	A	Command Code	A	Sr	Slave Addr	Rd	A	Data Byte 0	A	Data Byte 1	N	P

- The Command Code is the address of the first RMI register to be read.
- Data Byte 0 is the contents of the register at the RMI Command Code address.
- Data Byte 1 is the contents of the register at the RMI Command Code address+1.
- All SMBus Read operations terminate with a NAK bit followed by a STOP bit.

4.2.4. Repeated starts

For the Read Byte and Read Word transfer protocols, the SMBus specification requires that a Repeated Start event must be used to separate the writing of the Command Code byte from the reading of the data byte(s). The Repeated Start ensures correct operation in host systems that support multiple bus masters. For host systems that either do not require multi-master support or cannot generate Repeated Start events, Synaptics RMI-on-SMBus devices will permit a host to replace a Repeated Start with a Stop event followed by a Start event.

4.2.5. Multi-register read/write operations

To improve bus utilization, Synaptics RMI-on-SMBus devices permit special multi-register read and write operations as an extension to the SMBus specification. If a host performs a standard Byte Read operation but simply keeps reading more bytes, the device will continue to send registers from consecutively incrementing RMI addresses until the host finally NAKs the transfer.

Below is an example of a multi-register Read operation, where the host reads 4 consecutive RMI registers starting from address N:

S	SlaveAddr	Wr	A	Command Code (Register Addr N)	A	Sr	Slave Addr	Rd	A	Register N	A	Register N+1	A

Register N+2	A	Register N+3	N	P

In similar fashion, if the host performs a standard Byte Write operation but simply keeps writing more bytes, the RMI-on-SMBus device will write the extra bytes to subsequent register addresses.

Below is an example of a multi-register Write operation, where the host writes 3 consecutive RMI registers starting with register address N:

S	SlaveAddr	Wr	A	Command Code (Register Addr N)	A	Register N	A	Register N+1	A	Register N+2	A	S

4.2.6. SMBus compliance

The SMBus Specification Version 2.0 describes a wide variety of features. Not all of these features are required to be supported for a particular device to be considered to be SMBus-compliant. Full information on the SMBus can be found in the document titled *System Management Bus (SMBus) Specification*, Version 2.0 of August 3, 2000. This document can be found at http://www.smbus.org/.

SMBus is described in terms of *layers.* Synaptics SMBus compliance issues will be dealt with on a layer-by-layer basis.

4.2.6.1. Layer 1: physical layer

In general, the SMBus physical layer looks like a standard I^2C physical layer interface. To solve certain problems inherent in a typical I^2C interface, SMBus imposes some additional restrictions on the I^2C interface that it uses as its physical layer. These restrictions typically have to do with the timing and length of the transfers that are permitted. The important restrictions imposed by the SMBus specification on a Synaptics RMI slave device are:

- 10 KHz minimum bus operating frequency,
- 25 mSec (min), 35 mSec (max) clock-low timeout period, and
- 500 mSec (max) Time in which a device must be operational after power-on reset.

Note: Synaptics RMI-on-SMBus devices do not support the SMBus SMBALERT# signal. This means that Synaptics SMBus devices will not respond to the SMBus Alert Response Address. Synaptics devices support a general purpose Attention signal (ATTN) that can either be used as an interrupt input to a host processor or as an input that the host can poll. As an order-time option, the ATTN signal can be configured as being either active high or active low.

4.2.6.2. Layer 2: data link layer

Synaptics RMI-on-SMBus devices implement the Data Link layer as described in the *SMBus Specification*. Section 4.3.3 of the *SMBus Specification* Version 2.0 defines that SMBus devices must implement "clock-low extending" (also known as I^2C "clock-stretching"). In particular, the *SMBus Specification* V2.0 defines that *all* devices on a SMBus (both masters and slaves) must be able to tolerate both periodic and random clock stretching.

Synaptics RMI-on-SMBus slave devices can tolerate both random and periodic clock-stretching imposed by any other device sharing the SMBus. Synaptics SMBus slave devices will stretch the clock for short periods of time in random fashion, but they will never stretch the clock long enough to violate the 10 KHz (min) bus transfer frequency.

4.2.6.3. Layer 3: SMBus network layer

The SMBus Specification defines eleven different command protocols that can be used to transfer data. The specification states that a slave device does not need to support all eleven protocols in order to be SMBus compliant.

Synaptics RMI-on-SMBus devices support the following four SMBus transfer protocols:

- Byte Read, Byte Write
- Word Read, Word Write

Synaptics SMBus devices do *not* support the following SMBus protocols:

- Quick Command
- Send Byte, Receive Byte
- Process Call
- Block Write Process Call, Block Read Process Call
- Block Read, Block Write

Note: Synaptics SMBus devices do not support the ARP functionality to assign bus addresses. Synaptics SMBus devices implement a fixed, 7-bit I^2C addressing mechanism. The 7-bit I^2C slave address for a given Synaptics SMBus device will be fixed at the time that the product is ordered. It is the implementer's responsibility to choose a SMBus address that will not conflict with other SMBus devices in their system. If desired, RMI-on-SMBus modules can be ordered with an address-strapping option. This allows a host system to select a module's I^2C address among two or more preconfigured I^2C addresses by strapping module IO pins.

Note: Synaptics devices do not support the SMBus Packet Error Check (PEC) byte. Hosts should not expect Synaptics devices to generate a PEC byte during read operations. Hosts should not send PEC bytes to a Synaptics device during write operations.

4.2.7. Sample SMBus transfers

All of these examples assume that the RMI-on-SMBus device has been assigned I^2C address $20.

Write a RESET Command:

- Uses SMBus *Write Byte* protocol.
- RMI *Device Command* register is at SMBus address $F4.
- The *Reset* command bit is bit 0 in the *Device Command* register.

S	$40	A	$F4	A	$01	A	P

Read Button Data:

- Uses SMBus *Read Word* transfer protocol.
- RMI *Button Data* registers are located sequentially at SMBus addresses $00 and $01.
- Example data assumes that buttons 0, 7, and 9 are being pressed, and all others are released:

S	$40	A	$00	A	Sr	$41	A	$81	A	$02	N	P

Write Sleep Mode Control Register:

- Uses SMBus *Write Byte* protocol.
- RMI *Device Control* register is located at SMBus address $F0.

  We will write the value $81 (binary 10---001).

  - The *Report Rate* bits ‘10’ indicate normal 80Hz operation.
  - The *Sleep Mode* bits ‘001’ will configure normal operation. This means that the device will doze to save power, but it will wake up to process any button presses.

S	$40	A	$F0	A	$81	A	P

4.3. SPI physical interface

This section describes the RMI-on-SPI physical layer. (The version of the physical layer of an RMI device can be found by reading register $0202, described in section 2.7.3.)

4.3.1. SPI signals

RMI-on-SPI uses the industry-standard four-wire SPI interface. The SPI signals include:

- SSB, a device-select signal driven by the host. In some SPI systems this signal is known as Slave Select, $\overline{SS}$, or Chip Select, $\overline{CS}$. SSB is an active-low signal that goes low when an RMI transaction is in progress.
- SCK, a clock signal driven by the host. Several clocking conventions are supported, as described in section 4.3.2.
- MOSI (master out / slave in), a data signal driven by the host.
- MISO (master in / slave out), a data signal driven by the RMI device. The device drives MISO only when SSB is low; when SSB is high, the device floats its MISO pin. This allows multiple RMI devices to be connected with SCK, MOSI, and MISO all tied in parallel, using separate SSB wires to address the various devices.
- ATTN, an *optional* attention signal driven by the RMI device. This pin is not present on devices that use the SRQ mechanism to signal attention (see section 4.3.5 below).
- RESET, an *optional* reset signal driven by the host. If a device provides a RESET pin, the pin is an active-low input with a pull-up resistor on the RMI device. This allows RESET to be left unconnected when not needed.

4.3.2. SPI clocking

“SPI” is actually a loosely defined family of standard interfaces. The clock polarity and clock phase (often denoted CPOL and CPHA) vary from one SPI system to another. Synaptics can supply RMI devices that use the standard clocking conventions that correspond to CPHA = ‘1’. The clocking conventions that correspond to CPHA = ‘0’ are not currently supported by the RMI standard.

CPOL defines the idle level of SCK. CPOL is ‘0’ if SCK is low between transactions, or ‘1’ if SCK is high between transactions. The usual clocking convention for RMI-on-SPI devices is CPOL = ‘1’; upon request, Synaptics can also supply RMI devices that use the CPOL = ‘0’ clocking convention.

CPHA defines on which SCK edge the MOSI and MISO data are sampled by their respective receivers. CPHA is ‘0’ to sample on the edge where SCK leaves its idle level (the falling edge if CPOL is ‘1’), or CHPA is ‘1’ to sample on the edge where SCK returns to its idle level (the rising edge if CPOL is ‘1’). Thus, all current RMI-on-SPI devices sample MOSI, and expect the host to sample MISO, on the trailing SCK edge where SCK returns to its idle level.

RMI always transmits each byte most-significant-bit first, following the convention of most chips’ SPI interfaces. (In other words, RMI follows the DORD = ‘0’ convention of AVR microcontrollers.)

RMI always changes both MISO and MOSI on the same clock edge, and it always samples both MISO and MOSI on the same (opposite) clock edge. (In other words, RMI follows the SMP = '0' convention of PIC microcontrollers.)

4.3.3. SPI transaction format

The host (the SPI master) drives the SSB pin high between transactions, and low during a transaction (see Figure 49 and Figure 50). When SSB is high, the device (the SPI slave) floats its MISO pin and ignores the MOSI and SCK pins; this allows multiple devices to share the MISO, MOSI, and SCK pins provided that each device receives a separate SSB signal. When SSB is low, the device drives the MISO pin, and it samples MOSI and changes MISO in response to the clock waveform on SCK. SSB edges delimit a transaction; therefore, the host must hold SSB low continuously throughout a transaction.

During the first two bytes (16 SCK pulses) after the fall of SSB, the host transmits an *address word* on MOSI, most significant byte first. In the address word, bit 15 is '1' for a read transaction and '0' for a write transaction. Bits 14:0 hold the register address *R*. During the first byte of the address word, the device transmits undefined data on MISO; then during the second byte of the address, the device transmits the Device Status register.

For a read transaction, in subsequent bytes (groups of 8 SCK pulses), the device transmits on MISO the contents of consecutive registers starting from the addressed register, and MOSI is ignored.

For a write transaction, in subsequent bytes, the host transmits on MOSI the write data for consecutive registers starting from the addressed register, and the device transmits undefined data on MISO. During a write transaction that writes several consecutive registers, the writing action to each register occurs as the transfer of the data byte for the register completes (except for a very few multi-byte quantities that are written only when the final byte is written, as described in section 2.1).

Note: This addressing mechanism is different from that of RMI-on-SMBus, but it is more consistent with the types of mechanisms most often used on SPI devices.

The transaction ends when the host raises SSB. If the host raises SSB during or after either byte of the address word, no transaction occurs. If the host raises SSB during a write transaction when only a fraction of the 8 bits of a data byte have been transmitted, the register corresponding to that data byte is not written (but any registers written earlier in the transaction will already be committed).

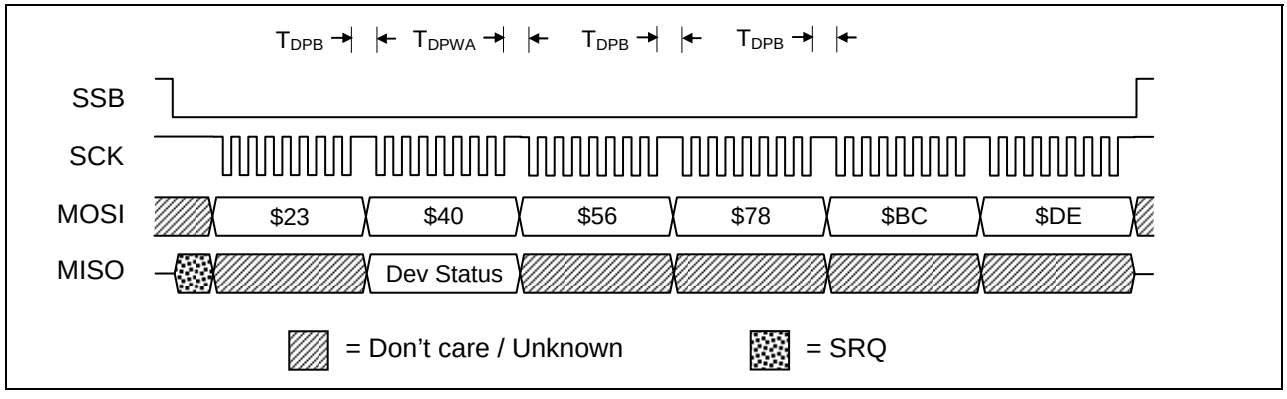


Figure 49. RMI-on-SPI write transaction (assuming CPOL = CPHA = '1'). Host writes $56, $78, $BC, and $DE to registers $2340–$2343, respectively.

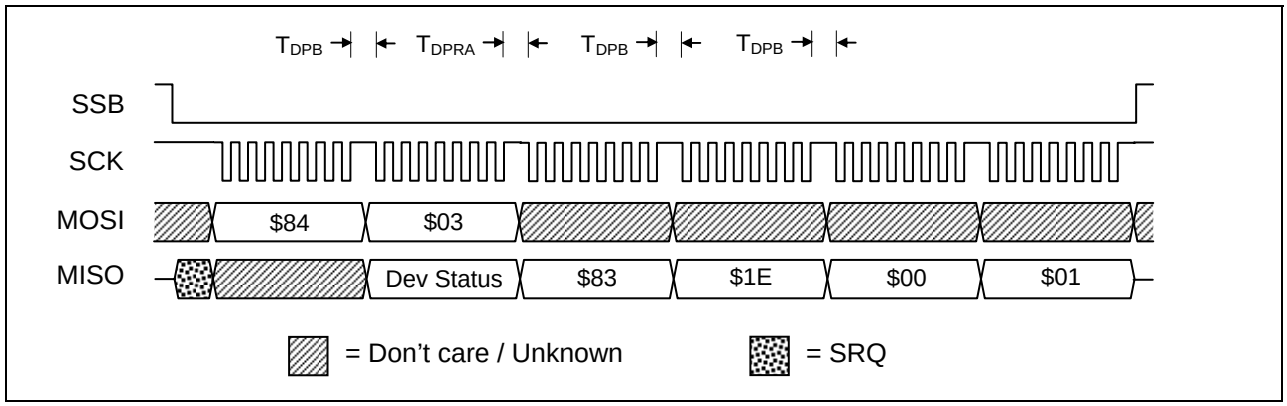


Figure 50. RMI-on-SPI read transaction (assuming CPOL = CPHA = '1'). Host reads $83, $1E, $00, and $01 from registers $0403–$0406, respectively.

4.3.4. SPI timing

The host may clock SCK at a rate F_{SCK} of up to [TBD] MHz (see Figure 51 and Table 15). The maximum rate will be at least 2 MHz for RMI devices based on Synaptics chips.

Note: The RMI protocol attempts to allow for RMI devices implemented on chips not made by Synaptics. For example, a Synaptics module including both a touch sensing chip and another chip might choose to use the other chip for host communication. Non-Synaptics chips implementing RMI devices may impose a lower maximum clock rate such as 1 MHz.

The minimum SCK high and low times, T_{CH} and T_{CL}, are each half of the minimum overall SCK period. RMI places no constraints on the *maximum* SCK high and low periods; the host may clock the device as slowly and as irregularly as it wishes.

Other timing parameters of the SPI port are TBD. For RMI devices based on Synaptics chips, the input-related parameters T_{SS}, T_{SH}, T_{DS}, and T_{DH} are expected to be no more than a few tens of nanoseconds, and the output-related parameters T_{SO}, T_{SZ}, and T_{DO} are expected to be no more than a few hundreds of nanoseconds under reasonable loads.

To allow time for the device to process each byte received and to generate the next byte for transmission, the host must provide a brief delay T_{DP} between bytes during an SPI transaction. A delay between bytes means a minimum time between the trailing SCK edge of the last bit of one byte and the leading SCK edge of the first bit of the following byte. The host must delay for $T_{DPRA} \geq 100$ µs between the second address byte and the first data byte in a read transaction, and it must delay for $T_{DPWA} \geq 100$µs between the second address byte and the first data byte in a write transaction. The host must delay for $T_{DPB} \geq 100$ µs between any other two bytes in a transaction. These are minimum delays; the host is free to delay longer

To allow time for the device to complete a transaction and prepare for the next transaction, the host must also provide a brief delay $T_{BT} \geq 100$ µs between the rising edge of SSB that completes one transaction and the falling edge of SSB that begins the next transaction.

Note: These delays were chosen to be sufficient for implementing RMI with a wide variety of features in a wide variety of chips. The actual minimum required delays might be reduced for a specific device implementation.

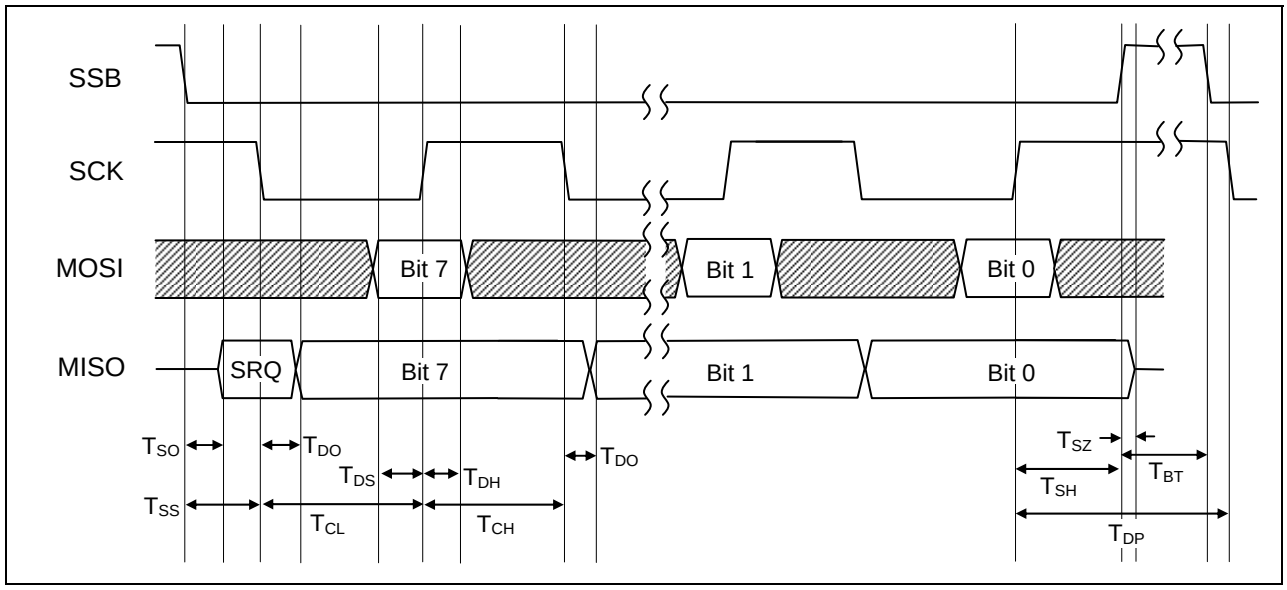


Figure 51. RMI-on-SPI timings (assuming CPOL = CPHA = '1')

Parameter	*Description*	*Condition*	*Min*	*Max*	*Unit*
F_{SCK}	Clock frequency		0	2	MHz
T_{CH}	SCK high period		250		ns
T_{CL}	SCK low period		250		ns
T_{SS}	SSB fall to SCK edge		TBD		ns
T_{SH}	SCK edge to SSB rise		TBD		ns
T_{DS}	MOSI setup time before SCK edge		TBD		ns
T_{DH}	MOSI hold time after SCK edge		TBD		ns
T_{SO}	SSB fall to MISO driven			TBD	ns
T_{SZ}	SSB rise to MISO floating			TBD	ns
T_{DO}	SCK edge to MISO change			TBD	ns
T_{DPB}	Interbyte delay between bytes		100		µs
T_{DPWA}	Interbyte delay after write address		100		µs
T_{DPRA}	Interbyte delay after read address		100		µs
T_{BT}	Delay between transactions		100		µs
T_{POR}	Delay after power-on reset		TBD		ms
T_{RESC}	Delay after reset command		TBD		ms

Table 15. RMI-on-SPI timings

4.3.5. SPI attention mechanism

During the second address byte that begins a transaction, the device transmits the Device Status register (see section 2.5.3). The host can examine this byte to determine whether any data registers contain interrupt requests.

Synaptics can offer RMI devices using either of two additional attention mechanisms. One mechanism uses a non-standard extension of the SPI interface called a "service request" (SRQ) bit. The other uses a separate fifth ATTN pin to hold the attention signal.

Note: The option for a separate attention pin accommodates hosts that cannot handle interrupts and MISO signals on the same host pin. It also may help in case RMI device interfaces must be implemented in non-Synaptics chips that cannot generate an SRQ-like bit on MISO.

In the SRQ option, the RMI device drives the attention signal onto MISO as soon as SSB falls (see Figure 49 and Figure 50).

The SRQ attention signal is “live” on MISO in the interval between the fall of SSB and the first SCK edge: The host may lower SSB, leave SCK at its idle level, and watch MISO using an interrupt to wait for an interrupt request report from the device. After the first SCK edge, it is undefined whether MISO continues to follow the “live” attention state, or freezes at the state of the attention signal at the time of the SCK edge.

The SRQ signal may be “live,” but its behavior is relatively simple: The only change to MISO that can possibly occur during the SRQ period is one transition from the inactive to the active attention level, because attention, once asserted, can be deasserted only by a host transaction that reads the data registers or writes the Interrupt Enable bits. For RMI devices implemented using Synaptics’ SPI-compatible chips, MISO will freeze after the first SCK edge.

Synaptics is also able to supply RMI devices that transmit the attention signal on a fifth ATTN pin. In this option, ATTN can be ordered either as an active-high push-pull output pin, or as an active-low open-drain pin for which the host must supply an external pull-up resistor. (The latter option allows multiple RMI devices’ ATTN pins to be merged in a wired-OR configuration. Because MISO is a fully driven push-pull output, the ATTN attention mechanism is better suited than SRQ to multi-device RMI systems.)

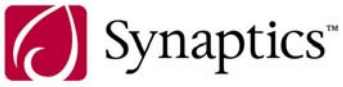
4.4. Sample ControlBar product address map

This section defines a sample RMI product designed for ControlBar usage. This sample ControlBar product is assumed to have ten capacitive buttons and eight current-controlled LEDs. The register map in the following table describes the aliased addresses and bit assignments for a sample product:

Aliased Address	*RMI Address*	*Register Name*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
$00	*$0400*	Func18: Button Data 0	Button 7	Button 6	Button 5	Button 4	Button 3	Button 2	Button 1	Button 0
$01	*$0401*	Func18: Button Data 1	—	—	—	—	—	—	Button 9	Button 8
$02	*N/A*	Device Status register	Error	Configured	—	—	—	—	—	F18 Button
$21	*$1841*	Func18: Button Control Reg 0	—	Button Usage			Heavy Filter	—	—	—	—
$22	*$1842*	Func18: Button Control Reg 1	Button Sensitivity Adjust							
$42	*$2242*	Func22: LED Enable	Enable 7	Enable 6	Enable 5	Enable 4	Enable 3	Enable 2	Enable 1	Enable 0
$44	*$2244*	Func22: Ramp Period A	Ramp Period A							
$45	*$2245*	Func22: Ramp Period B	Ramp Period B							
$50	*$2250*	Func22: LED #0	Pattern				Brightness			
$51	*$2251*	Func22: LED #1	Pattern				Brightness			
$52	*$2252*	Func22: LED #2	Pattern				Brightness			
$53	*$2253*	Func22: LED #3	Pattern				Brightness			
$54	*$2254*	Func22: LED #4	Pattern				Brightness			
$55	*$2255*	Func22: LED #5	Pattern				Brightness			
$56	*$2256*	Func22: LED #6	Pattern				Brightness			
$57	*$2257*	Func22: LED #7	Pattern				Brightness			
$E4	*$0204*	Product Major Version	Major Version (Product Family)							
$E5	*$0205*	Product Minor Version	Minor Version (Product Number)							
$F0	*$0000*	Device Control Register	Report Rate		—	—	—	Sleep Mode		
$F1	*$0001*	Interrupt Enable Control Register	—	—	—	—	—	—	—	F18 Button
$F2	*$0002*	Error Status Register	Error Code							
$F3	*$0003*	Interrupt Request Status Register	—	—	—	—	—	—	—	F18 Button
$F4	*$0004*	Device Command Register	—	—	—	—	—	—	Rezero	Reset
$FF	*$xxFF*	Page Select Register	—	High 7 bits of 15-Bit RMI Page Address						

Contact Us

To locate the Synaptics office nearest you, visit our website at www.synaptics.com.



CORPORATE HEADQUARTERS
SYNAPTICS INCORPORATED
3120 SCOTT BLVD.
SANTA CLARA, CA 95054
USA

P +1 408 454 5100
F +1 408 454 5200
sales@synaptics.com