RX62N Group

Multi-Slave RSPI Driver for YRDKRX62N

R01AN0772EU0100
Rev.1.00
Sep 24, 2011

Introduction

The YRDKRX62N has a number of on-board devices that communicate over the SPI bus such as the LCD, the Micron phase-change flash memory, and the microSD Card socket, with the RSPI peripheral in the RX62N MCU serving as the controller. This document describes the design and operation of an RSPI driver that employs standard C code to directly control the RSPI registers.

This driver was originally designed to support streaming of uncompressed 44Khz stereo audio data stored in the SDcard or the PCM flash memory. Performance of the driver has been verified to be sufficient to achieve the minimum 1411 kbps (1.4mbps) transfer rate needed to successfully stream 44kHz 16-bit stereo PCM audio data to a host PC when combined with the overhead of the USB Mass Storage function. The driver also supports communication with the Okaya LCD on the YRDKRX62N.

Together with this application note there is RSPI driver level c-code, *rx62n_rspi_driver.h/c*, written specifically for the YRDK62N board.

Target Audience: Those developing applications that use RSPI to control multiple slave devices from Renesas RX MCUs, and who need examples that demonstrate configuration and operation of the RSPI peripheral at the register level.

Target Device

RX62N on YRDKRX62N board.

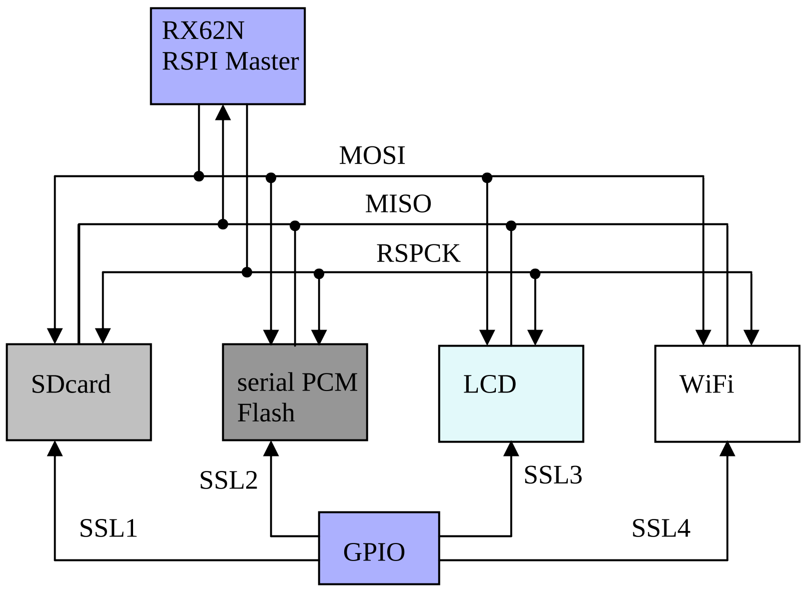

Figure 1: RSPI connected peripherals on the YRDKRX62N

Contents

1. General description

The RX62N includes two independent channels of Serial Peripheral Interface (RSPI) that are capable of full-duplex high-speed serial communications with multiple processors and peripheral devices. The YRDKRX62N demonstration board has four peripheral slave devices connected to the RSPI channel 0 bus: a microSD card slot, a Micron 16MB flash memory, the Okaya LCD module, and a general purpose application header that can support an external Redpine WiFi controller. This application note describes a software code module, the YRDKRX62N RSPI Driver, that supports the use of these devices at the RSPI hardware level with a single set of low-level driver functions.

The driver functions described in this document constitute the lowest level of code that directly manages the Renesas RSPI bus controller peripheral in the RX microcontroller. To complete actual communications with the devices attached to the RSPI bus on the YRDKRX62N board, there are higher level protocols needed that are specific to each device. Those higher protocol layers must call the primitive RSPI driver routines found in the YRDKRX62N RSPI Driver. Discussion of the higher level protocols is outside the scope of this document.

2. Review of SPI Basics

The Serial Peripheral Interface (SPI) is a three or four wire bidirectional synchronous serial data link standard. SPI operates in full-duplex mode with a "master" device initiating communications and a "slave" device responding. Data is passed in "data frames" which may consist of various number and size of data words. Each signal line of the SPI bus is unidirectional, with the primary signals being SCLK (serial clock), MOSI (master-out, slave-in), and MISO (master-in, slave-out). Therefore, a minimal system consists of a single master and a single slave device. Multiple slave devices may be present on the bus simultaneously with the individual slave enabled by the use of an SSL (slave select) signal. Each slave will normally have its own unique select line.

The specifications for signal levels vary among devices using SPI. It is common for the signals to idle either high or low, however many SPI devices ignore the idle state of the signal lines and only depend on the clock active edge transition. Clock phase to data, and polarity also vary, but typically the data is clocked in on the rising edge of SCLK. Therefore, it is necessary to carefully review the specifications for each device connected to the SPI bus in order to determine the settings required.

3. The YRDKRX62N RSPI Driver

The RSPI driver for the YRDKRX62N is provided with a set of functions that support initialization of the RSPI peripheral, write a block of data, read a block of data, and control the slave select signals for the four supported devices (LCD, on-board serial flash, SD card slot, and application header/WiFi port). To maintain interchangeability with existing YRDKRX62N applications created using the HEW project generator, this RSPI driver implementation preserves the function names found in the older version of the driver.

Table 1 List of YRDKRX62N RSPI functions

Driver Function name	Purpose
YRDKRX62N_RSPI_Init	Prepares the RSPI peripheral for operation as SPI master.
YRDKRX62N_RSPI_Select	Asserts slave select signal on one of the four SPI slave devices.
YRDKRX62N_RSPI_Deselect	De-asserts slave select signal of one of the four SPI slave devices.
YRDKRX62N_RSPI_Read	Reads n number of bytes from a SPI slave device.
YRDKRX62N_RSPI_Write	Writes n number of bytes to a SPI slave device.
YRDKRX62N_RSPI_Lock	Acquires a lock on the RSPI channel.
YRDKRX62N_RSPI_Unlock	Releases the resource lock on the RSPI channel
YRDKRX62N_RSPI_TxReady	Waits until RSPI transmit register is empty.

4. Detailed description of YRDKRX62N RSPI Driver functions

YRDKRX62N_RSPI_Init

Prepares RSPI channel for communication.

This function sets mode, data frame size, bit-rate, configures I/O pins, and sets up interrupts.

Prototype

```
void YRDKRX62N_RSPI_Init(uint8_t rspi_channel);
```

Arguments

`rspi_channel`	the channel number, 0 or 1.	Channel 0 is used for all YRDKRX62N RSPI slaves

Return value

None

Comments

Must be called prior to performing any data transfers over RSPI

Settings summary

Data format: single transfer -8 bits data length

Bit order: MSB first

RSPI clock idle state: 1 (high) when idle

MOSI (data out) idle state: idle at 0 (low).

Modes:

- SPI operation (4-wire)
- Full-duplex.
- Master mode
- SPTI and SPRI enabled

RSPI I/O pins

- RSPI Pin group A
- RSPCKA-A (clock) is output
- MOSIA-A (master out) is output
- MISOA-A (master in) is input

Clock: Use bit rate divide by 1

Bit rates: 0 = 24 Mbps, 1 = 12Mbps

Example

```
/* Initialize the RSPI bus */
YRDKRX62N_RSPI_Init(RSPI_CHANNEL_0);
```

YRDKRX62N_RSPI_Read

Transfers a block of data from a SPI slave device to the RSPI master.

This function reads a block of data from the RSPI bus into a predefined destination buffer.

Prototype

```
bool YRDKRX62N_RSPI_Read(uint8_t *pDest, uint16_t usBytes );
```

Arguments

`pDest`	Pointer to buffer for the received data. The buffer must have space for the number of bytes in `usBytes`.
`usBytes`	number of bytes to be received.

Return value

true, success; false, failure

Comments

This function is used to read raw data from any of the four RSPI connected slave devices. The SSL signal line must be asserted on the source device by calling the `YRDKRX62N_RSPI_Select` function before calling this function. An arbitrary number of bytes may be read, limited only by the size of the read-data destination buffer and the value of the byte count parameter which is an unsigned 16-bit type. As data is read, the index into the destination buffer will automatically increment. It is up to the calling program to allocate space for the destination buffer and to provide any upper level communication protocols needed.

See the chapter on Data Transfer for details of the internal theory of operation.

Example usage

```
/* read a block of 512 bytes into sector read buffer */
YRDKRX62N_RSPI_Read(buffer, 512);
```

YRDKRX62N_RSPI_Write

Transfers a block of data from RSPI master to SPI slave device.

This function writes a block of data in memory onto the RSPI bus.

Prototype

```
bool YRDKRX62N_RSPI_Write(uint8_t *pSrc, uint16_t usBytes );
```

Arguments

`pDest`	Pointer to buffer containing the data to be transmitted
`usBytes`	number of bytes to be received.

Return value

true, success; false, failure

Comments

This function is used to write raw data to any of the four RSPI connected slave devices. The SSL signal line must be asserted on the destination device by calling the `YRDKRX62N_RSPI_Select` function before calling this function. An arbitrary number of bytes may be written as specified by the value of the byte count parameter which is an unsigned 16-bit type. As data is written, the index into the source data buffer will automatically increment. It is up to the calling program to provide any upper level communication protocols needed.

See the chapter on Data Transfer for details of the internal theory of operation.

Example usage

```
    /* Write the data start token to the SD card */
    YRDKRX62N_RSPI_Write(&data_start_token, 1);

    /* Write a block of 512 bytes from sector source buffer */
    YRDKRX62N_RSPI_Write(wbuffer, 512);
```

YRDKRX62N_RSPI_Select

Asserts SPI slave select signal for specified device.

Prototype

```
void YRDKRX62N_RSPI_Select(uint16_t chip_select);
```

Arguments

`chip_select` Enumerated SPI slave device number.

Legal device number macro definitions:

```
/* enumeration for chips selects associated with YRDKRX62N RSPI-0 */
typedef enum {
    NO_DEVICE_SELECTED=0,
    SDMICRO_SELECTED,
    FLASH_SELECTED,
    WIFI_SELECTED,
    LCD_SELECTED
} device_selected ;
```

Return value

None.

Comments

Slave select signals are output under program control using GPIO pins. The RX62N RSPI peripheral does have integrated slave select outputs, however these are not used in this application because the YRDKRX62N board has other GPO pins routed to the attached SPI slave devices. Because of this, it is up to the application to manage slave selects by calling the YRDKRX62N_RSPI_Select( ), or YRDKRX62N_RSPI_Deselect( ) at the appropriate times.

Example usage

```
/* Assert SD slot chip select. */
YRDKRX62N_RSPI_Select(SDMICRO_SELECTED);
```

YRDKRX62N_RSPI_Deselect

De-asserts SPI slave select signal for specified device.

Prototype

```
void YRDKRX62N_RSPI_Deselect(uint16_t chip_select);
```

Arguments

`chip_select` Enumerated SPI slave device number.

Legal device number macro definitions:

```
/* enumeration for chips selects associated with YRDKRX62N RSPI-0 */
typedef enum {
    NO_DEVICE_SELECTED=0,
    SDMICRO_SELECTED,
    FLASH_SELECTED,
    WIFI_SELECTED,
    LCD_SELECTED
} device_selected ;
```

Return value

None.

Comments

Slave select signals are output under program control using GPO pins. The RX62N RSPI peripheral does have integrated slave select outputs, however these are not used in this application because the YRDKRX62N board has other GPO pins routed to the attached SPI slave devices. Because of this, it is up to the application to manage slave selects by calling the YRDKRX62N_RSPI_Select( ), or YRDKRX62N_RSPI_Deselect( ) at the appropriate times.

Example usage

```
/* Done. Deselect the sdcard. */
YRDKRX62N_RSPI_Deselect(SDMICRO_SELECTED);
```

YRDKRX62N_RSPI_Lock

Acquires a lock on the RSPI channel.

Used to cooperatively manage access to the RSPI bus.

Prototype

```
bool YRDKRX62N_RSPI_Lock(uint16_t pid);
```

Arguments

`pid`	unique program ID.

Return value

true = lock acquired;   false = lock not acquired

Comments

To acquire the lock on the RSPI bus, pass a unique program ID in a call to function YRDKRX62N_RSPI_Lock. On success, the call will return true, or false if the lock is currently owned by another process.

Example usage

```
/* Try to get the RSPI lock */
if (YRDKRX62N_RSPI_Lock(myPid))
{
      ; //got the lock so now free to use the RSPI bus
}
else;
{
      ; //RSPI bus busy. Try again or do something else.
}
```

YRDKRX62N_RSPI_Unlock

Releases the resource lock on the RSPI channel.

Prototype

```
    bool YRDKRX62N_RSPI_Unlock(uint16_t pid);
```

Arguments

`pid`	unique program ID that owns the lock.

Return value

true = correct pid, unlock successful

false = pid not owner of lock, lock not released.

Comments

To release the lock on the RSPI bus, pass the unique program ID of the process that owns the lock in a call to function YRDKRX62N_RSPI_Unlock. On success, the call will return true, or false if the lock is currently owned by another process.

Example usage

```
/* Release the RSPI lock */
if (!YRDKRX62N_RSPI_Unlock(myPid))
{
      ; //Handle error, I thought I was the owner.
}
```

YRDKRX62N_RSPI_TxReady

Waits until RSPI transmit register is empty.

A utility spin-loop used by some legacy applications.

Prototype

```
void YRDKRX62N_RSPI_TxReady(void);
```

Arguments

None.

Return value

None.

Example

```
/* wait for any RSPI transmit operation to complete */
YRDKRX62N_RSPI_TxReady();
```

5. Driver interrupt service routines (ISR).

These callback functions are used internally by the RSPI Driver. They are not used outside of the driver module.

RSPI0_SPRI0

Receive buffer full interrupt.

Sets a global flag `g_SPI_RXdata_avail` to indicate that the receive data register contains data and is ready to be read.

Arguments

none

Return Value

none

Declaration

```
#pragma interrupt RSPI0_SPRI0(vect=VECT(RSPI0, SPRI0))
static void RSPI0_SPRI0(void);
```

RSPI0_SPTI0

Transmit buffer empty interrupt.

Sets a global flag `g_SPI_TXbuff_empty` to indicate that the transmit data buffer register is ready to be written.

Arguments

none

Return Value

none

Declaration

```
#pragma interrupt RSPI0_SPTI0(vect=VECT(RSPI0, SPTI0))
static void RSPI0_SPTI0(void);
```

6. Operation overview of YRDKRX62N RSPI Driver

This RSPI driver operates under "Single-Master/Multi-Slave" mode as described in the RX62N Group User's Manual. For complete details of the RSPI peripheral operation please refer to the RX62N Group User's Manual.

Prior to performing any data transfers over RSPI, the application must initialize the RSPI peripheral by making a call to function YRDKRX62N_RSPI_Init(0). Channel 0 is used for the YRDKRX62N. The application must then call the YRDKRX62N_RSPI_Select function to assert the desired slave select before beginning data transmission. However, if there is any possibility that the RSPI bus may be busy with communication to another slave device, then the YRDKRX62N_RSPI_Lock function should be called to acquire the lock before asserting slave select. Then data transfer may begin by calling the read or write functions YRDKRX62N_RSPI_Read, or YRDKRX62N_RSPI_write. Finally, after the transfer is complete, the slave select may be deselected and the lock may be released.

6.1 SPI Slave Select

Four SPI slave devices are defined for connection to the RSPI bus in this implementation. These are:

1. Micro-SDcard socket.
2. Micron phase-change flash memory.
3. WiFi / Application header
4. Okaya LCD module

Slave select signals are output under program control using GPO pins. The RX62N RSPI peripheral does have integrated slave select outputs, however these are not used in this application because the YRDKRX62N board has other GPO pins routed to the attached SPI slave devices. Because of this, it is up to the application to manage slave selects by calling the YRDKRX62N_RSPI_Select( ), or YRDKRX62N_RSPI_Deselect( ) at the appropriate times.

6.2 Managing multiple RSPI access conflicts with Lock and Unlock

To prevent multiple applications from attempting to take control of the RSPI driver at the same time, which would result in conflicts on the SPI bus, a simple locking mechanism is provided in which an application can acquire a "lock" on the RSPI channel. This lock is not enforced, but rather must be used cooperatively by calling applications. In situations where multiple application operations are using the RSPI bus for different devices, an application should request a lock before attempting to use the RSPI bus. This should be done before asserting the slave select signal to avoid having multiple devices selected simultaneously. If the lock is granted, then it becomes "owned" by the calling application program ID (pid). Then if another application tries to obtain the lock, it is denied and the application should wait until the lock is released by the application that currently owns it. Only the owner pid can release the lock.

Checking the lock

Before asserting the slave select signal, a process should check ownership of the RSPI bus lock by attempting to acquire the lock. If the attempt fails, the process must not take control of the RSPI bus. After a process is finished using the RSPI bus, it should release its lock to make it available to another process. It is up to the application to cooperatively operate within this method.

Acquiring RSPI lock

To acquire the lock on the RSPI bus, pass a unique program ID in a call to function YRDKRX62N_RSPI_Lock. On success, the call will return true, or false if the lock is currently owned by another process.

Releasing RSPI lock

To release the lock on the RSPI bus, pass the unique program ID of the process that owns the lock in a call to function YRDKRX62N_RSPI_Unlock. On success, the call will return true, or false if the lock is currently owned by another process.

Data transfer

6.3 Sending data from RSPI Master

Data is written by the RSPI Master on the MOSI (master out, slave in) line. The RSPI driver write function YRDKRX62N_RSPI_Write performs this operation.

An RSPI Transmit buffer empty interrupt ISR is implemented to provide faster throughput when interrupts are enabled. The RSPI_Write function operates in a polling fashion while waiting for the TX buffer to become available. RSPI_Write waits for a local global flag that is set by the transmit buffer empty ISR before going on to write another byte. Because of the possibility that this function may be called from within some other interrupt callback function, the RSPI transmit buffer empty interrupt may not occur, due to interrupts being disabled while processing the "outer" interrupt. To handle this situation, RSPI_Write also checks the transmit buffer empty IR flag in the ICU while it's waiting for the g_SPI_TXbuff_empty flag. e.

An arbitrary number of bytes may be transmitted in the RSPI_Write function, limited only by the size of the source-data buffer and the size of the byte count parameter which is an unsigned 16-bit value. As data is written, the index into the source-data buffer will automatically increment.

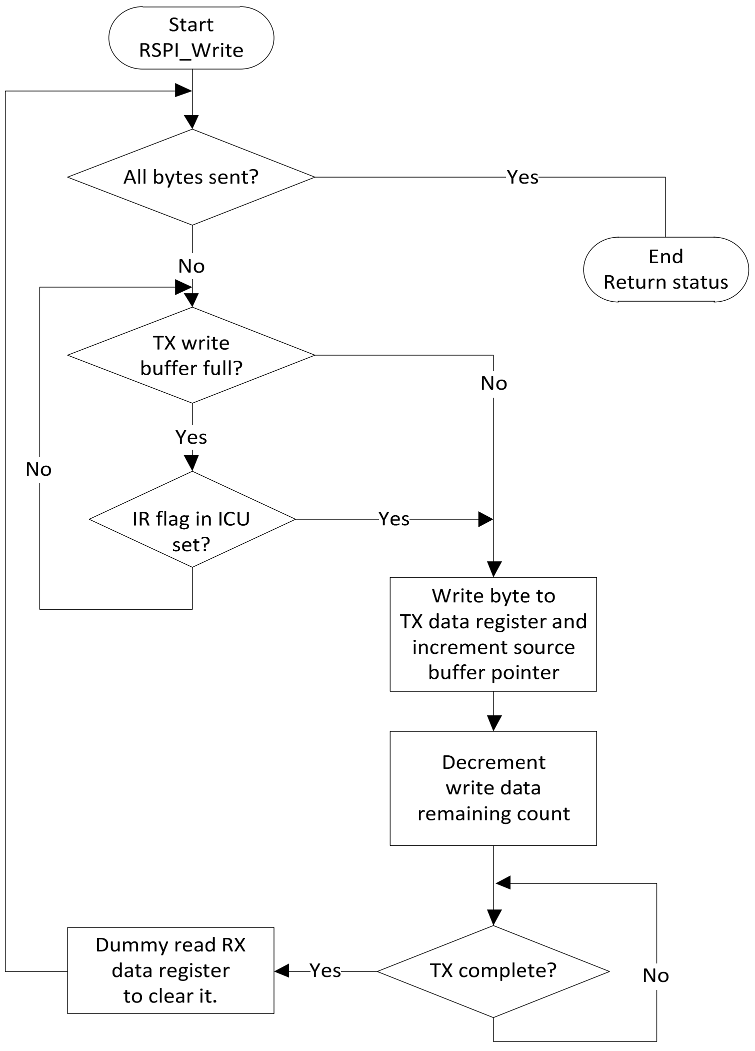
Start RSPI_Write
All bytes sent?
Yes
End Return status
No
TX write buffer full?
No
Yes
No
IR flag in ICU set?
Yes
Write byte to TX data register and increment source buffer pointer
Decrement write data remaining count
Dummy read RX data register to clear it.
Yes
TX complete?
No

Figure 2: RSPI_Write function flow diagram

6.4 Receiving data from RSPI Slave

Data is received by the RSPI Master on the MISO (master in, slave out) line. With the RSPI peripheral configured as SPI bus Master, it must be set for full-duplex operation in order to receive data from a slave device on the SPI bus. This requires the RSPI Master to output clocks to the Slave. Clocks are output only when the RSPI Master is sending data. Therefore, in order to read data from the SPI bus, the master must also simultaneously write data. This can either be actual data that needs to be transmitted (if the slave is capable of full-duplex communication), or it can be dummy data that is ignored by the slave. In the case of this driver implementation the read data is clocked by dummy writes with 0xFF data pattern. The slave device's protocol specifies data patterns that indicate the status or meaning of data it writes to the bus.

In this driver implementation the RSPI read function YRDKRX62N_RSPI_Read performs this operation.

An RSPI receive buffer full interrupt ISR is implemented to provide faster throughput when interrupts are enabled. The YRDKRX62N_RSPI_Read function operates in a polling fashion. YRDKRX62N_RSPI_Read waits for a local global flag that is set by the receive buffer full ISR before reading the receive buffer register. Because of the possibility that this function may be called from within some other interrupt callback function, the RSPI receive buffer full interrupt may not occur, due to interrupts being disabled while processing the "outer" interrupt. In this case the receive buffer may become full without the g_SPI_RXdata_avail flag ever becoming set. To handle this situation, YRDKRX62N_RSPI_Read also checks for the presence of the receive buffer full IR flag in the ICU while it's waiting for the g_SPI_RXdata_avail flag.

An arbitrary number of bytes may be read in the YRDKRX62N_RSPI_Read function, limited only by the size of the read-data destination buffer and the value of the byte count parameter which is an unsigned 16-bit type. As data is read, the index into the destination buffer will automatically increment.

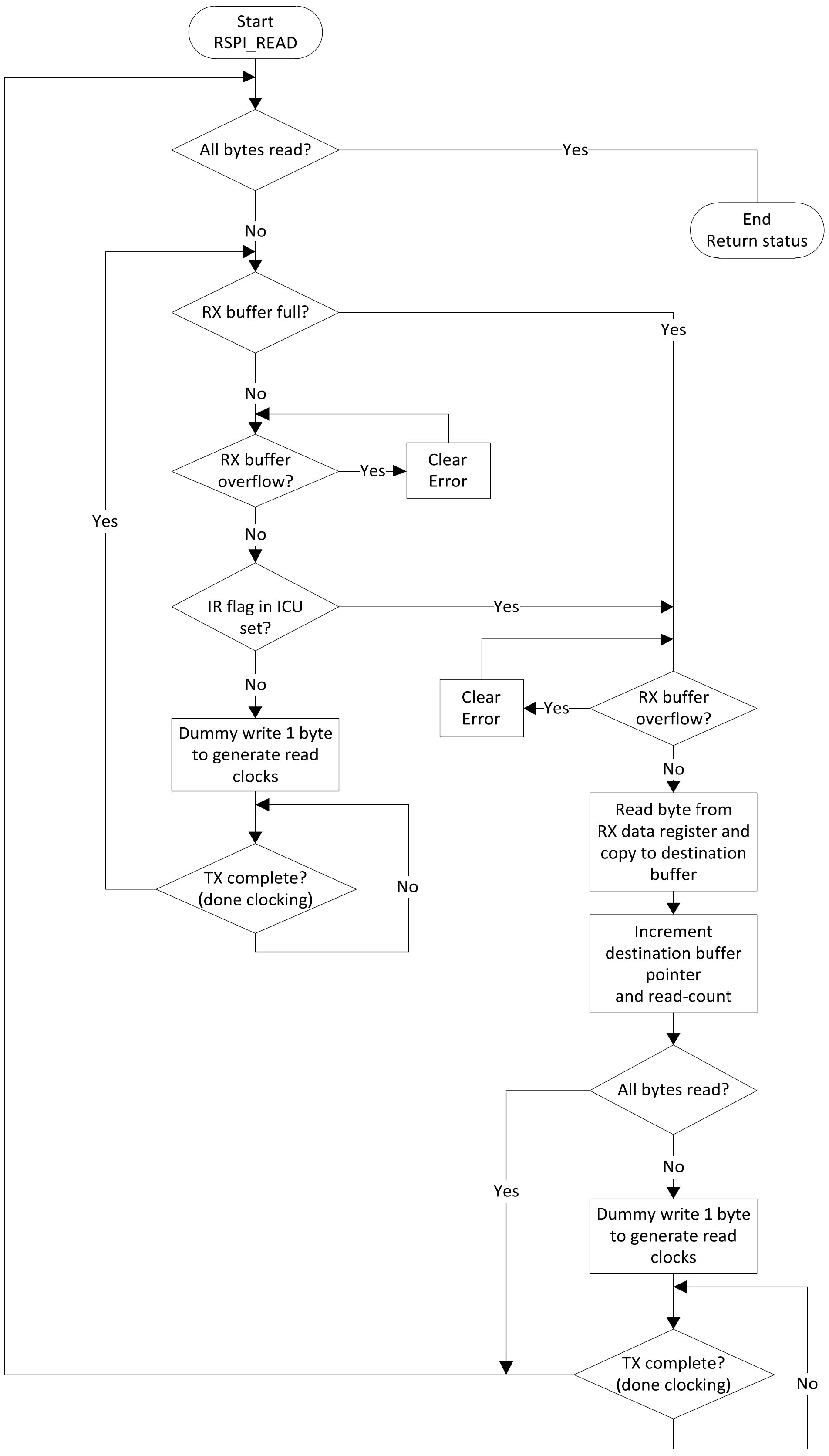
Start
RSPI_READ
All bytes read?
Yes
End
Return status
No
RX buffer full?
Yes
No
RX buffer overflow?
Yes
Clear Error
No
Yes
IR flag in ICU set?
Yes
No
Clear Error
Yes
RX buffer overflow?
Dummy write 1 byte to generate read clocks
No
Read byte from RX data register and copy to destination buffer
TX complete? (done clocking)
No
Increment destination buffer pointer and read-count
All bytes read?
No
Yes
Dummy write 1 byte to generate read clocks
TX complete? (done clocking)
No

Figure 3 RSPI_READ function flow diagram

Website and Support <website and support,ws>

Renesas Electronics Website
 http://www.renesas.com/

Inquiries
 http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Sep 24, 2011	—	First edition issued

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.

   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan, R.O.C.
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141