Table of Contents

W5100S 1
- ***Overview*** 1
- ***Pin MAP*** 2
- ***Features*** 3
- ***Target Application*** 3

Documents 4
- ***Data Sheet*** 4
- ***W5100S vs W5100 Comparison Sheet*** 4
- ***Application Note*** 4
- ***Hardware Design Guide*** 5

Driver 5
- ***1. ioLibrary_BSD*** 5
- ***2. ioLibrary*** 10
- ***3. BSD Type driver for W5200 User*** 11

Reference Schematic 11
- ***External Transformer Type*** 12
- ***RJ45 with Transformer Type*** 12

W5100S Application 13

W5100S TCP Function 13
- ***Initialization*** 14
  - Basic Setting 14
  - Setting Network Information 14
  - Set SOCKET n Buffer Information 14
- ***Data Communications*** 15
  - TCP 15
  - TCP SERVER 16
  - TCP CLIENT 22

W5100S UDP Function 24
- ***Initialization*** 24
  - Basic Setting 24
  - Setting network information 24
  - Set SOCKET memory information 25
- ***Data Communications*** 25
  - UDP 25
  - Unicast and Broadcast 26
  - Multicast 29

W5100S

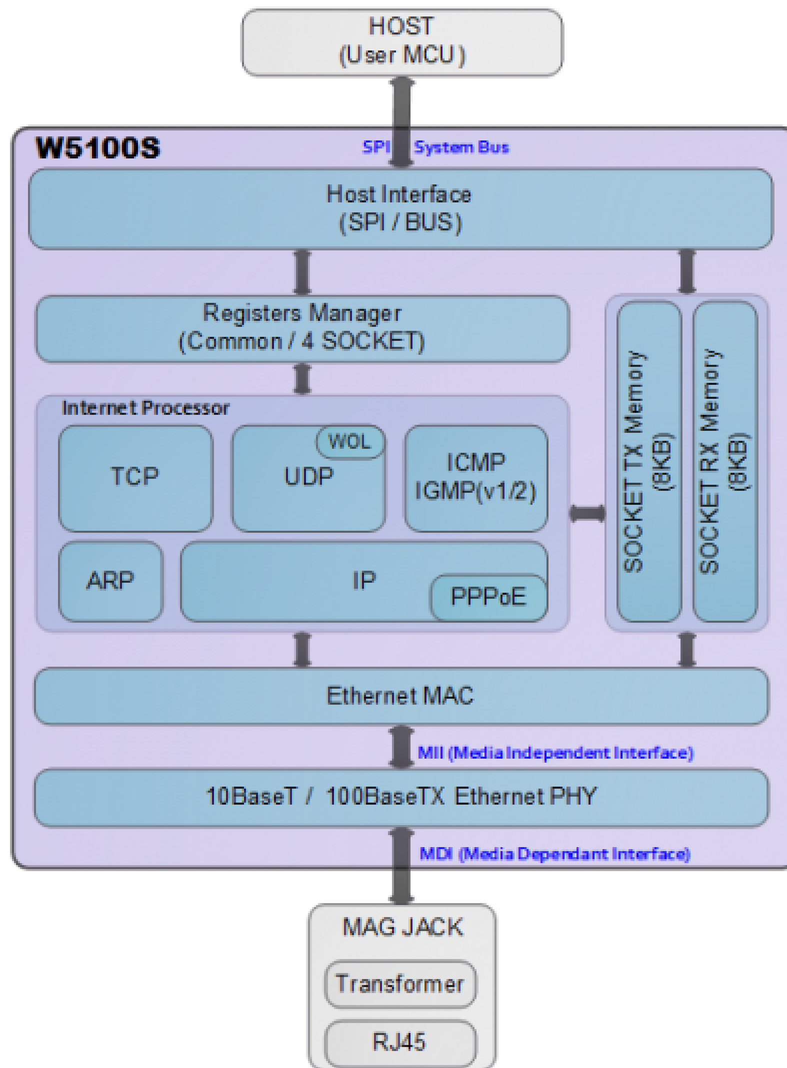
Overview

W5100S is an embedded Internet controller designed as a full hardwired TCP/IP with WIZnet technology. W5100S provides internet connectivity to your embedded system by using SPI (Serial Peripheral Interface) or Parallel System BUS. SPI and Parallel System BUS provide easy connection via external MCU to W5100S. The clock speed of W5100S SPI supports upto 70MHz and the Parallel System Bus supports higher speed network communication than SPI.

Since W5100S integrates the Hardwired TCP/IP stack with 10/100 Ethernet MAC and PHY, it is truly a one-chip solution for the stable internet connectivity. WIZnet's hardwired TCP/IP stack supports TCP, UDP, IPv4, ICMP, ARP, IGMP, and PPPoE - and it has been proven through various applications over the last decade.

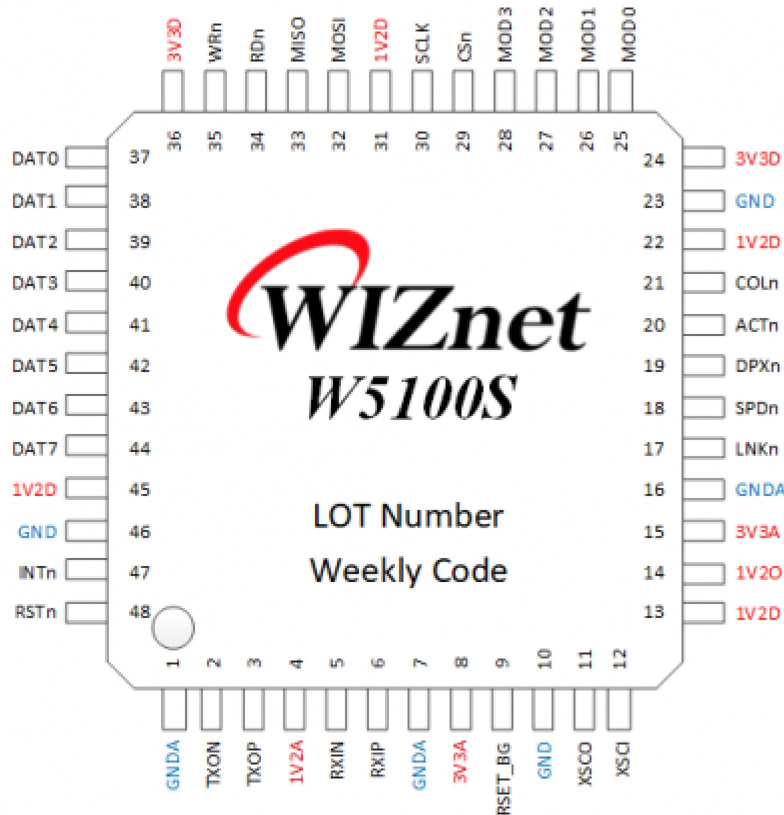
W5100S provides four independent SOCKETs to be used simultaneously and 16KB internal memory for data communication. Users can develop an Ethernet application easily by using the simple W5100S SOCKET program instead of handling a complex Ethernet controller. W5100S also provides WOL (Wake on LAN) and a Power Down Mode in order to reduce power consumption.

W5100S is a low-cost chip that exceeds its predecessor, W5100. Existing firmware using W5100 can be used on W5100S without modification. W5100S has two types of packages, 48 Pin LQFP & QFN Lead-Free Package.



—-

Pin MAP



Features

- Support Hardwired Internet Protocols: TCP, UDP, WOL over UDP, ICMP, IGMPv1/v2, IPv4, ARP, PPPoE
- Support 4 Independent Hardware SOCKETs simultaneously
- Support SOCKET-less Command: ARP-Request, PING-Request
- Support Ethernet Power Down Mode & Main Clock gating for power save
- Support Wake on LAN over UDP
- Support Serial & Parallel Host Interface: High Speed SPI(MODE 0/3), Parallel System Bus with 2 Address signal & 8bits Data
- Internal 16 Kbytes Memory for TX/ RX Buffers
- Not support IP Fragmentation
- Not Maintain ARP-cache Table
- 10BaseT/100BaseTX Ethernet PHY Integrated
- Support Auto Negotiation (Full/Half Duplex, 10/100 Speed)
- Support Auto-MDIX when Auto-Negotiation Mode.
- 3.3V operation with 5V I/O signal tolerance
- LED outputs (Full/Half Duplex, Link, 10/100 Speed, Active)
- Two types of packages: 48 Pin LQFP & QFN Lead-Free Package (7x7mm, 0.5mm pitch)

Target Application

- User product based on W5100 : No need to modify Firmware
- Home Network Devices: Set-Top Boxes, PVRs, Digital Media Adapters
- Serial-to-Ethernet: Access Controls, LED displays, Wireless AP relays, etc.
- Parallel-to-Ethernet: POS / Mini Printers, Copiers
- USB-to-Ethernet: Storage Devices, Network Printers

- GPIO-to-Ethernet: Home Network Sensors
- Security Systems: DVRs, Network Cameras, Kiosks
- Factory, Building, Home Automations
- Medical Monitoring Equipment
- Embedded Servers
- Internet of Things (IoT) Devices
- IoT Cloud Devices

2017/12/11 15:58 · Bang

Documents

Data Sheet

Korean

- W5100S Datasheet v1.2.5

English

- W5100S Datasheet v1.2.5

W5100S vs W5100 Comparison Sheet

Korean

- W5100S vs W5100 Comparison Sheet v1.1.0

English

- W5100S vs W5100 Comparison Sheet v1.1.0

Application Note

Korean

- W5100S IPRAW v1.0.0
- W5100S PPPoE v1.0.0
- W5100S SLC v1.0.0
- W5100S Interrupt v1.1.0

English

- W5100S IPRAW v1.0.0
- W5100S PPPoE v1.0.0
- W5100S SLC v1.0.0
- W5100S Interrupt v1.0.0

Hardware Design Guide

Korean

- Crystal Selection Guide v1.0.0

English

- Crystal Selection Guide v1.0.0
- IR Reflow Profile

2017/03/31 16:28 · 이상준

Driver

The ioLibrary means **"Internet Offload Library"** for WIZnet chip. It includes **drivers** and **application protocols**. There are three kinds of libraries explained on this page The first two drivers (ioLibrary_BSD, ioLibrary) can be used for W5100S application designs. These will be updated continuously. The former BSD-Type driver will not be updated, as it is only meant to be a migration help from W5200 to W5100S.

---

1. ioLibrary_BSD

2. ioLibrary

3. BSD Type driver for W5200 User

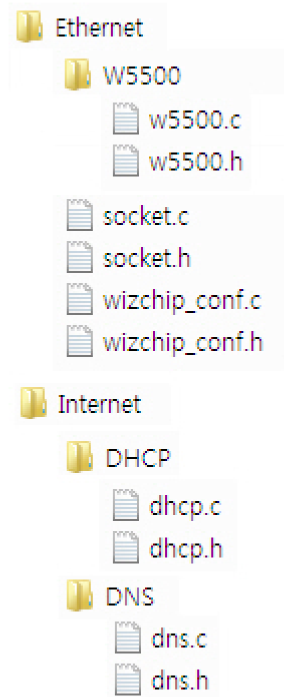
---

1. ioLibrary_BSD

Overview

This driver provides the Berkeley Socket type APIs. The function names of this ioLibrary_BSD are the same as the function names of the ioLibrary.

- Directory Structure



- Ethernet : SOCKET APIs like BSD & WIZCHIP(W5500,W5200 and etc) Driver
- Internet :
  - DHCP client
  - DNS client
  - Others will be added.

Download

< ioLibrary_BSD : latest version >

ioLibrary_BSD GitHub Repository
https://github.com/Wiznet/ioLibrary_Driver

< ioLibrary_BSD : old version >

	Type	Version	Note	Download Link
Source code	Ethernet (Berkeley Socket type APIs)	1.0.3	-	Click
		1.0.2	-	Click
		1.0.1	-	Click
		1.0.0	-	Click
	Internet (Application protocols)	1.1.1	-	Click
		1.1.0	-	Click
		1.0.0	-	Click
Documents	Socket APIs Help (chm, html)	1.0.3	-	Click
		1.0.2	-	Click
		1.0.1	-	Click
		1.0.0	-	Click

< Update History >

- ioLibrary_BSD
    - Ethernet : Berkeley Socket type APIs
        - Document (chm, html): Socket APIs Help
        - Revision History
            - ioLibrary_BSD will be update in github continuously.
            - V1.03 (Refer to 20140501)
                - wizchip_conf.c
                    1. ***Explicit type casting in wizchip_bus_readbyte() & wizchip_bus_writebyte()***
                    2. uint32_t type converts into ptrdiff_t first. And then reconverting it into uint8_t*. For remove the warning when pointer type size is not 32bit. ***If ptrdiff_t doesn't support in your complier, You should must replace ptrdiff_t into your suitable pointer type.***
                - w5500.c : ***Implicit type casting → Explicit type casting***
                    1. wizchip_read_data() & wizchip_write_data() : Fixed the problem on porting into under 32bit MCU
                - socket.h
                    1. Modify the comment : SO_REMAINED → PACK_REMAINED
                    2. Add the comment as zero byte udp data reception in getsockopt().
                - socket.c
                    1. ***Implicit type casting → Explicit type casting.***
                    2. replace 0x01 with PACK_REMAINED in recvfrom()
                    3. Validation a destination ip in connect() & sendto(): It occurs a fatal error on converting unint32 address if uint8* addr parameter is not aligned by 4byte address. Copy 4 byte addr value into temporary uint32 variable and then compares it.

- V1.02
- socket.c (Refer to 20131220)
    1. setsockopt() : Remove warning message (delete tmp variable)
- w5500.c (Refer to 20131220)
    1. WIZCHIP_READ_BUF() & WIZCHIP_WRITE_BUF() in _WIZCHIP_IO_MODE_SPI_FDM_ case
        1. Remove warning message
        2. Remove unnecessary 'for' loop
- V1.01
- socket.c (Refer to 20131104)
    1. sendto() : Add to clear the timeout interrupt status of socket(Sn_IR_TIMEOUT).
- V1.00
- First released.
- Internet : Application protocols
- Revision History
- V1.11 (2013-12-26)
- DHCP Client
    1. Modify variable declaration(dhcp_tick_1s) for code optimization in dhcp.c
- V1.10
- DHCP Client
    1. Optimize code
    2. Add reg_dhcp_cbfunc()
    3. Add DHCP_stop()
    4. Integrate check_DHCP_state() & DHCP_run() into DHCP_run()
    5. Don't care system endian
    6. Move unreferenced DEFINE to dns.c
    7. Remove the unused DEFINE
    8. Add comments
- DNS Client
    1. Remove secondary DNS server in DNS_run

        1. If 1st DNS_run failed, call DNS_run with 2nd DNS again
    2. DNS_timerHandler → DNS_time_handler
    3. Move unreferenced DEFINE to dns.c
    4. Remove the unused define
    5. Integrated dns.h dns.c & dns_parse.h dns_parse.c into dns.h & dns.c
- V1.00
- First released.
    - DHCP Client (Dynamic Host Configuration Protocol Client)
    - DNS Client (Domain Name System Client)

< Application code examples : latest version >

	Application	Update	Note	Download Link
STM32F103X CooCox CoIDE Project	Loopback Test		-	
	DHCP Client		-	
	DNS Client		-	
EnergyMicro Tiny GECKO(EFM32TG840F32) IAR Project	Loopback Test DHCP Client DNS Client		-	

< Application code examples : old version >

	Application	Update	Note	Download Link
STM32F103X CooCox CoIDE Project	Loopback Test		-	
	DHCP Client		-	
	DNS Client		-	}

These projects do not contain [Ethernet] and [Internet] codes. (Empty directory)

Please download ioLibrary_BSD APIs and Application protocols, and then insert to each of same named directory in provided project.

< History >

- Application code example
    - Example project was made by CooCox CoIDE with the STM32F103X Cortex-M3 platform.
        - Loopback Test

- DHCP Client

- DNS Client

Description

This driver provides BSD-type Socket APIs for W5100S. Because the function names of this driver are more user-friendly than those of the older drivers, …, current WIZnet chip users can easily migrate from their WIZnet chip application to the W5500 application. All drivers for W5100, W5200 and W5300 will be merged into the ioLibrary in the near future. All application protocols will also be merged into ioLibrary based on this BSD-type Socket APIs.

This table shows the differences between other BSD drivers and new W5500 driver.

Driver	Other BSD Drivers	W5100S Driver
Variables Type	type.h (made by wiznet) ex) uint16	
Register Naming	REGName + Index ex) SIPR0 , SIPR1, SIPR2, SIPR3	
Basic IO function	IINCHIP_READ IINCHIP_WRITE IINCHIP_READ_BUF IINCHIP_WRITE_BUF 16bit Address Space User should implement Functions MCU Dependent	
Register Function	IINCHIP_XXX can be used. Supports some getREG() & setREG() functions.	
Extra Functions	None	

Socket APIs	Other BSD Drivers	W5100S Driver
Return Value	void Success or Fail Transmit/Receive Size	
Error Code	None	
IO Mode	Block & Non-Block Fixed	
Block Function	send recv sendto recvfrom	
Non-Block Function	connect	
recvfrom	Should read data in received packet unit.	

- Socket APIs
  - Function Name
    - Same as the function name of previous drivers
  - Function Return value
    - Previous Drivers: Void or Success/Fail and Transmit/Receive Size
    - W5500 Driver: All functions return Success and Fail. In Fail case, operations are subdivided.
      - Success: SOCK_OK, Socket Number, Transmit and Receive Size
      - Fail: SOCK_BUSY, SOCKERR_XXX, SOCKFATAL_XXX (0 or Negative value)
        1. **SOCK_BUSY** : 0
        2. SOCKERR_SOCKNUM
        3. SOCKERR_SOCKOPT
        4. SOCKERR_SOCKINIT
        5. SOCKERR_SOCKCLOSED
        6. SOCKERR_SOCKMODE
        7. SOCKERR_SOCKFLAG
        8. SOCKERR_SOCKSTATUS
        9. SOCKERR_ARG
        10. SOCKERR_PORTZERO
        11. SOCKERR_IPINVALID
        12. SOCKERR_TIMEOUT
        13. SOCKERR_DATALEN
        14. SOCKERR_BUFFER
        15. **SOCKFATAL_PACKLEN**
  - Block / Non-Block IO mode
    - Previous Drivers : Block function and Non-Block function were mixed.
      - Block Function : send(), recv(), sento(), recvfrom()
      - Non-block Function : connect()

        - Blocking can be avoided by using getSn_SR(), getSn_TX_FSR(), and getSn_RX_RSR() properly.
    - W5500 Driver
        - Block / Non-Block IO mode can be selected by user. (Default: Block mode)
        - socket() with new flag SF_IO_NONBLOCK or setsockopt() with SO_SET_IOMODE Can be configured.
        - Block and Non-block Configurable Function
        - connect(), send(), recv(), sendto(), recvfrom()
        - **getSn_SR(), getSn_TX_FSR() and getSn_RX_RSR() functions can be used like … like previous drivers. They are not related to IO mode**

---

2. ioLibrary

Download

< ioLibrary with example project : latest version >

	Application	Version	Note	Download Link
Cookie board	Loopback test	1.0.2	-	Click

<Revision History>

- v102
    - socket.c(Refer to 2014-03-18)
        1. TCPReSend() : Remove this function and related codes because TCP send mechanism was changed.
        2. TCPReSendNB() : Remove this function and related codes because TCP send mechanism was changed.
        3. TCPSendCHK() : Modify return value.
        4. TCPSend() : Change return value to len.
    - loopback.c(Refer to 2014-03-18)
        1. Existing mechanism resend packet if don't send all received packet, but change not to resend.
- v100
    - First release

< ioLibrary : latest version >

	Description	Version	Note	Download Link
Driver Source code	ioLibrary source code	1.0.2	-	Click

< ioLibrary : old version >

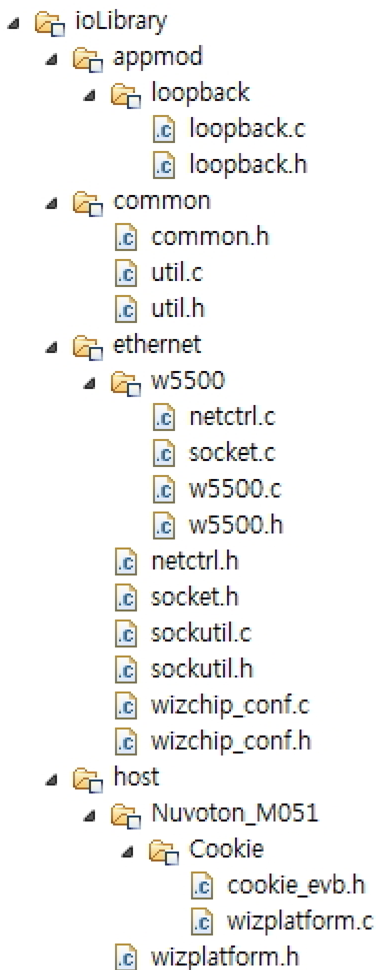
	Description	Version	Note	Download Link
Driver Source code	ioLibrary source code	1.0.0	-	Click
Driver documents	Socket APIs Help(chm, html) (To use html, open the index.html)	1.0.0	-	Click

This ioLibrary has basic I/O functions, socket register access functions, common register access functions, utilities and functions for setting up a platform and network This code has been evaluated on the CooCox Cookie Board with ARM Cortex-M0 MCU.

Please refer to this link for more details.

- How to use on cookie board.

The figure below shows the folder structure of this ioLibrary.



---

3. BSD Type driver for W5200 User

- Driver Source code : w5500_cortexm3_firmware_for_legacy.zip

This driver has the same BSD as the API for W5200 users. We have been evaluating this code on the **ARM-CortexM3(STM32F103 series)** chipset.

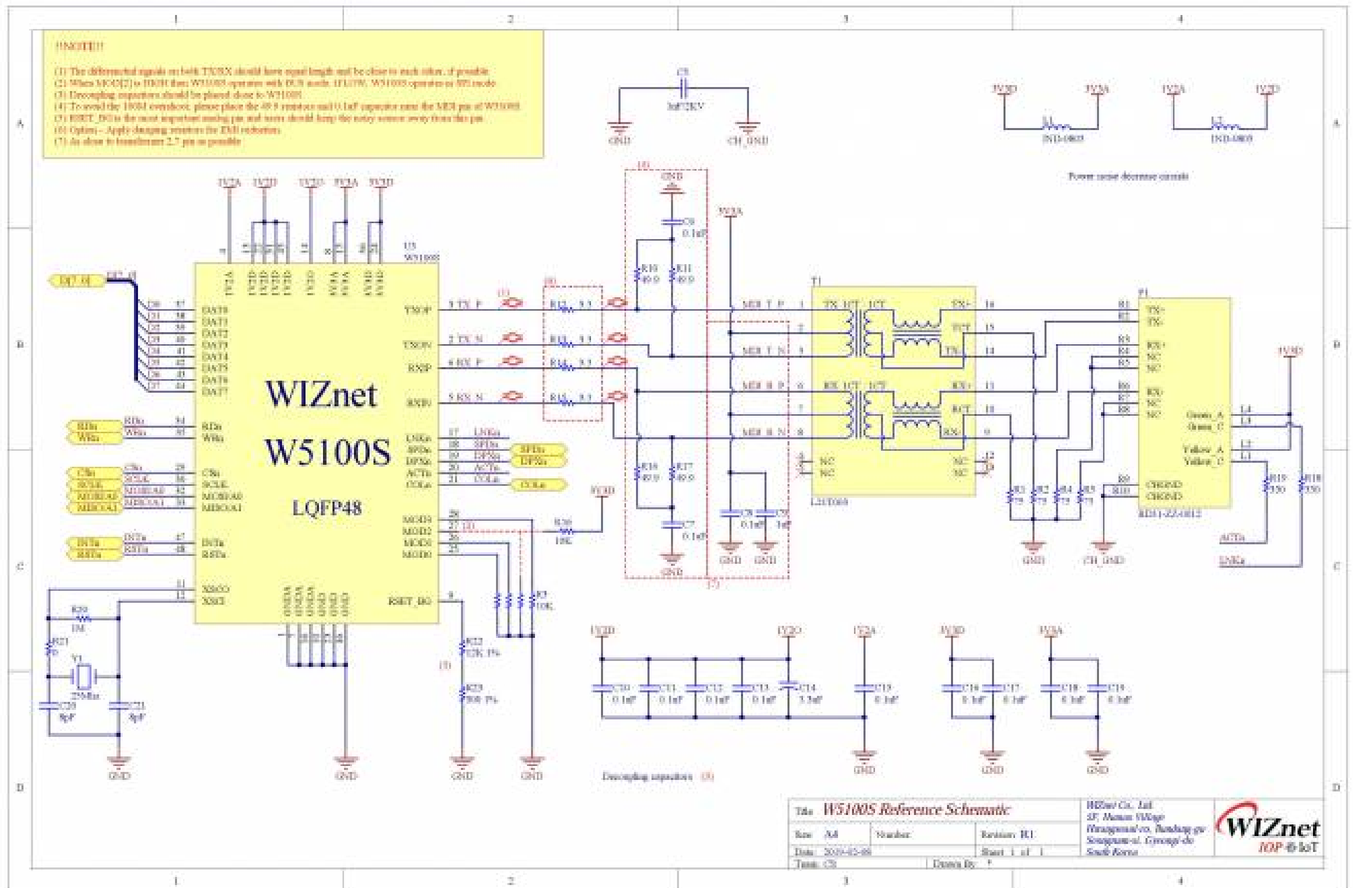
This type of driver is the final version. We will not update it later. Please use the new (well coded ☺) driver code for new projects.

2017/03/31 16:30 · 이상준

Reference Schematic

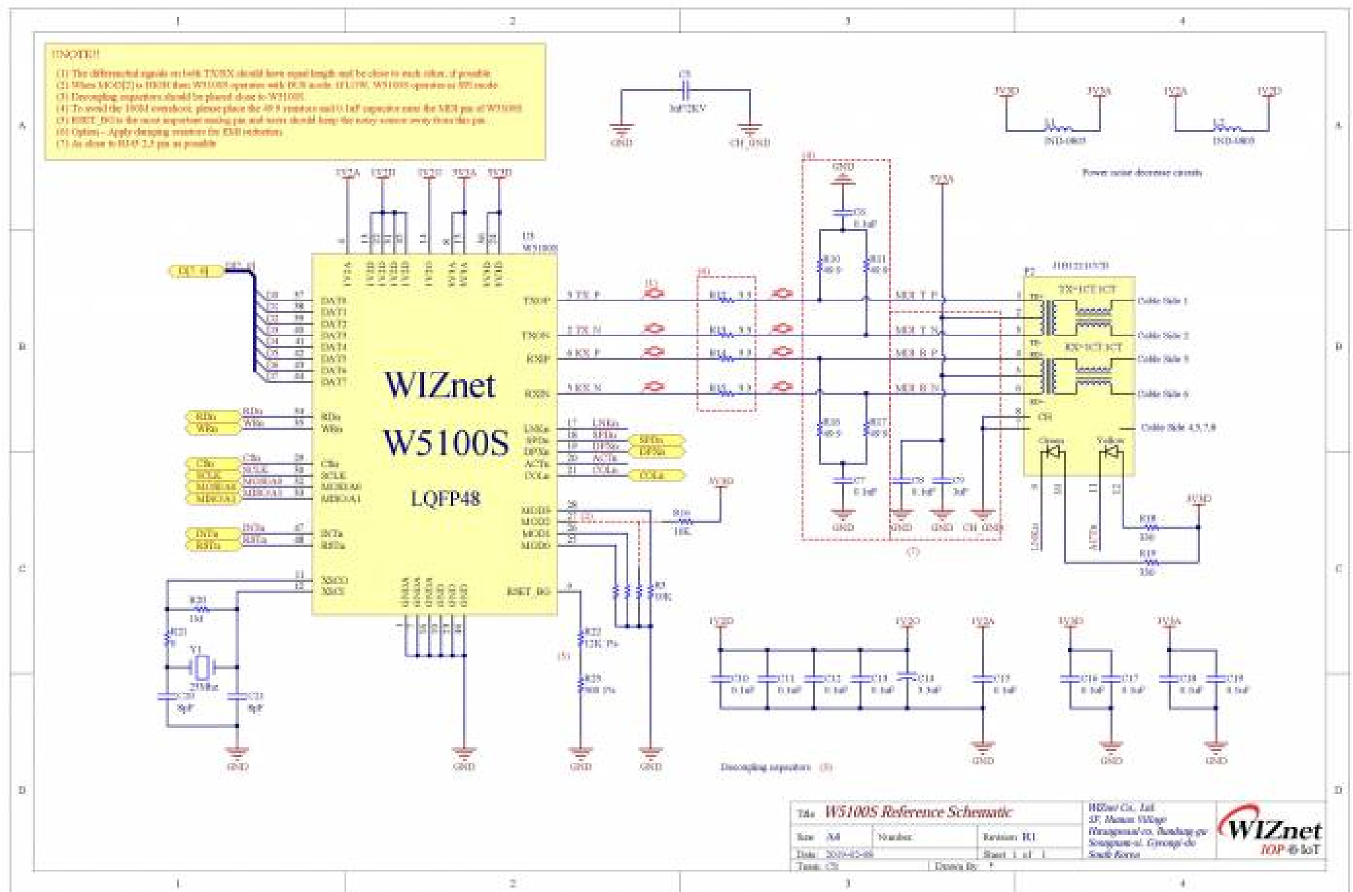
External Transformer Type

Go to github



RJ45 with Transformer Type

Go to github



2017/03/31 18:24 · Bang

W5100S Application

Refer to the following application examples.

- TCP
- UDP
- IPRAW
- PPPoE
- SOCKET-less Command
- Interrupt
- DMA

2017/12/11 16:32 · Bang

W5100S TCP Function

By setting some Registers and Memory Operation, W5100S provides Internet Connectivity. This Chapter

describes How to operate W5100S TCP Function.

Initialization

Basic Setting

For W5100S Operation, select and utilize appropriate Registers shown below.

1. Mode Register (MR)
2. Interrupt Mask Register (IMR)
3. Retry Time-value Register (RTR)
4. Retry Count Register (RCR)

For more Information of above Registers, refer to the “Register Descriptions” in W5100S Datasheet.

Setting Network Information

Basic Network Information setting for Communication: It must be set the basic Network Information.

1. SHAR(Source Hardware Address Register)
   - It is prescribed that the Source Hardware Addresses, which is set by SHAR, use unique Hardware Addresses (Ethernet MAC address) in the Ethernet MAC Layer. The IEEE manages the MAC address allocation. The manufacturer which produces the Network device allocates the MAC Address to product.
   - Details on MAC address allocation refer to the website as below.
   - http://www.ieee.org/
   - http://standards.ieee.org/regauth/oui/index.shtml
2. GAR(Gateway Address Register)
3. SUBR(Subnet Mask Register)
4. SIPR(Source IP Address Register)

Set SOCKET n Buffer Information

This stage shows SOCKET n TX/RX Buffer Information. The base Address and Mask Address of each SOCKET are set in this stage.

In case of, assign 2KB TX/RX Buffer per SOCKET

```
In case of, assign 2KB TX/RX Buffer per SOCKET
{
  // set Base Address of TX/RX Buffer for SOCKET n
  gS0_RX_BASE = 0x8000; // TX Buffer Block Base Address
  gS0_RX_BASE = 0xC000; // RX Buffer Block Base Address
  TxTotalSize = ; // For check the total size of SOCKET n TX Buffer
  RxTotalSize = ; // For check the total size of SOCKET n RX Buffer

  for (n=; n<3; n++) {
     Sn_TXBUF_SIZE = 2; // assign 2KB TX Buffer per SOCKET
     Sn_RXBUF_SIZE = 2; // assign 2KB RX Buffer per SOCKET
     // 0x07FF, for getting offset address within assigned SOCKET n TX/RX Buffer
     gSn_TX_MASK = (1024 * Sn_TXBUF_SIZE) – 1;
     gSn_RX_MASK = (1024 * Sn_RXBUF_SIZE) - 1;
```

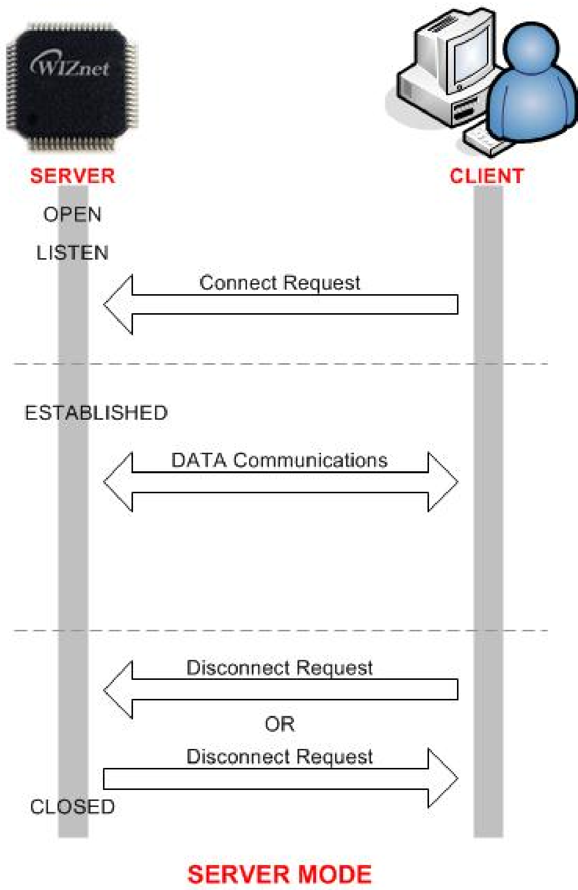
```
      if( n != ) {
         gSn_TX_BASE = gSn-1_TX_BASE + (1024 * Sn-1_TXBUF_SIZE);
         gSn_RX_BASE = gSn-1_RX_BASE + (1024 * Sn-1_RXBUF_SIZE);
      } // end if

   TxTotalSize = TxTotalSize + Sn_TXBUF_SIZE;
   RxTotalSize = RxTotalSize + Sn_RXBUF_SIZE;
   If( TxTotalSize > 8 or RxTotalSize > 8 ) goto ERROR; // invalid Total Size
   } // end for
}
```

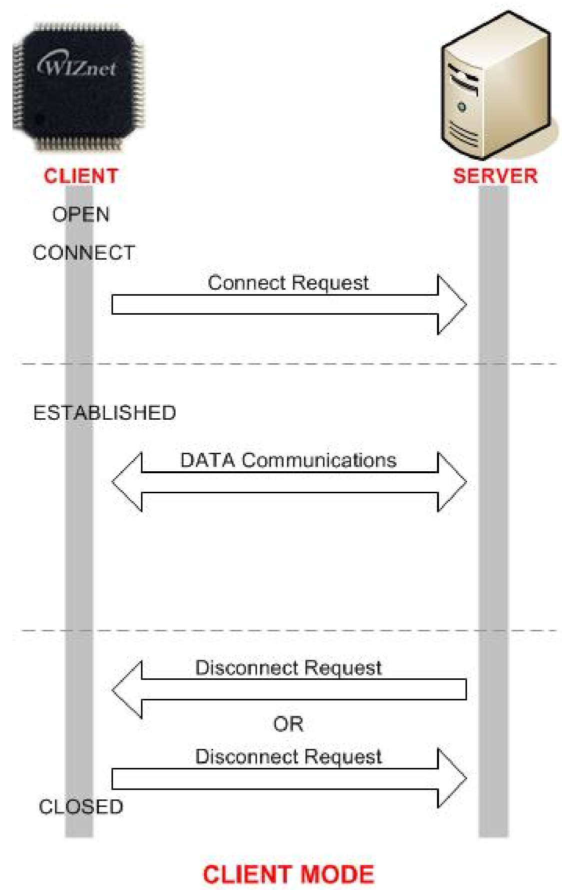
Data Communications

After Initialization Process, SOCKET is opened on TCP, UDP, IPRAW or MACRAW Mode and able to transmit and receive Data. This stage shows How to use SOCKET on TCP Mode.

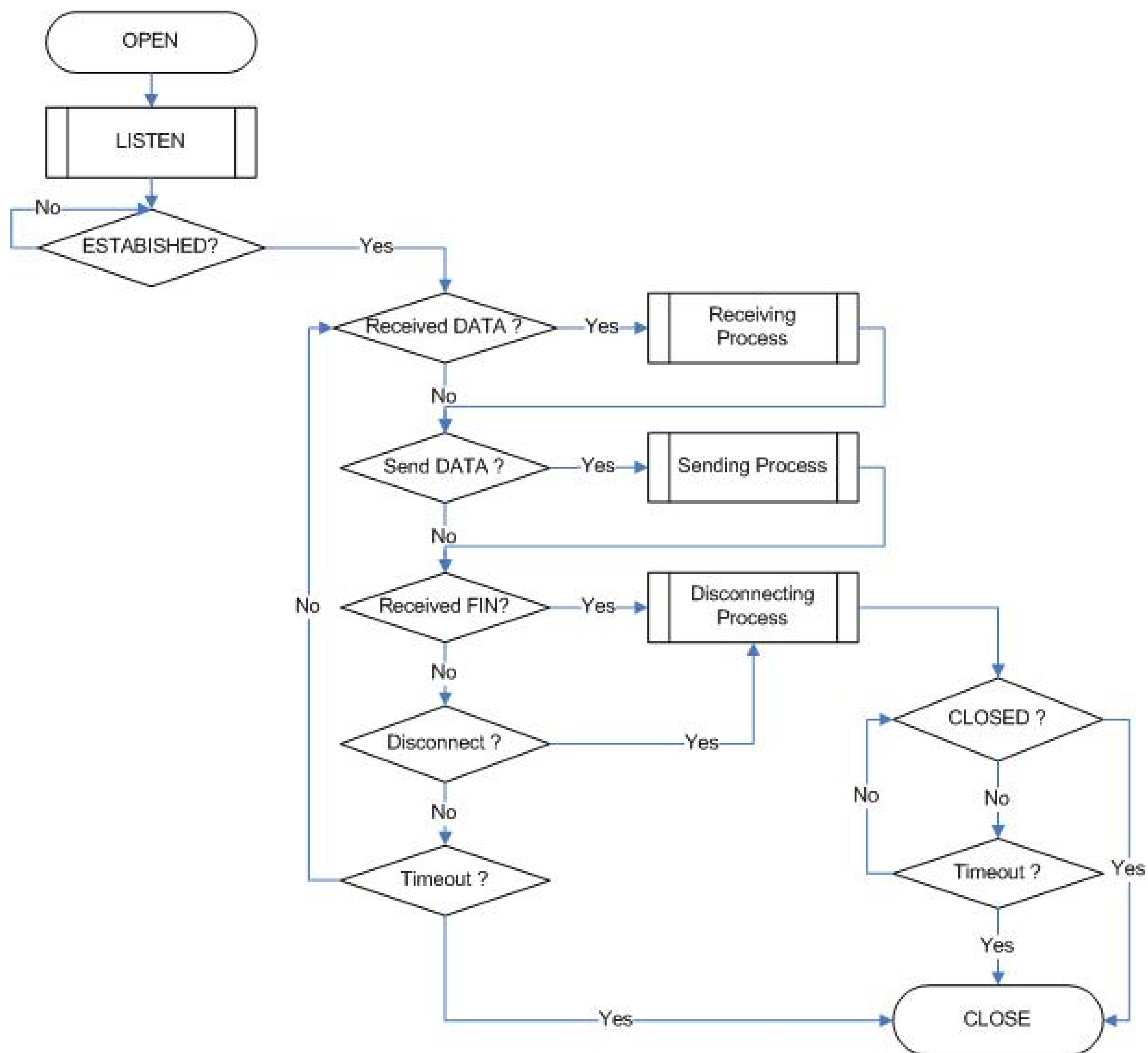
TCP

TCP (Transmission Control Protocol) is a bidirectional Data Transmission Protocol based on a 1:1 communication on Transport Layer. It also provides Communication between Applications by using Port Number. TCP 1:1 communication needs the Connection Process such as transmitting Connection Request to Peer or receiving Connection Request from Peer. In this Connection Process, the side transmitting Connection Request is 'TCP CLIENT' and the other side receiving Connection Request is 'TCP SERVER'. TCP also provides reliable, ordered and error-checked delivery of a stream Data between applications running on hosts communicating by an IP network. 'TCP SERVER' and 'TCP CLIENT' are maintaining transmit and receive Data until the TCP connection is terminated.





TCP SERVER



SOCKET Initialization

SOCKET Initialization is required for TCP Mode SOCKET. The Initialization consists of SOCKET Mode setting, SOCKET Port Number setting, SOCKET Option setting and SOCKET OPEN Command. 4 SOCKETs are all opened as TCP Mode. After OPEN Command(Sn_CR = OPEN), if the SOCKET status(Sn_SR) is changed to SOCKET_INIT, SOCKET Initialization is completed. This process is identically applied in “TCP SERVER” and “TCP CLIENT”.

```
{
START:

   Sn_MR[3:] = "0001"; // set TCP Mode
   Sn_PORTR0,1 = source_port; // sets source port number

   /* configure SOCKET Option when you need it */
   // Sn_MR[ND] = '1'; // set No Delay ACK

   Sn_CR = OPEN; // sets OPEN command
   /* wait until Sn_SR is changed to SOCK_INIT */
   if (Sn_SR != SOCK_INIT) Sn_CR = CLOSE; goto START;
}
```

LISTEN

Run as “TCP SERVER” by LISTEN Command.

```
{
   /* listen SOCKET */
   Sn_CR = LISTEN;
   /* wait until Sn_SR is changed to SOCK_LISTEN */
   if (Sn_SR != SOCK_LISTEN) Sn_CR = CLOSE; goto START;
}
```

ESTABLISHMENT

“TCP SERVER” keeps Sn_SR (SOCK_LISTEN) until received SYN Packet. If “TCP SERVER” receives SYN Packet from “TCP CLIENT”, it transmits SYN/ACK Packet to ‘TCP CLIENT’ and the Connection Process between “TCP SERVER” and “TCP CLIENT” is completed. If there is no response from Peer against of transmitted SYN Packet or SYN/ACK Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to ‘1’.

```
First method :
{
   /* check SOCKET Interrupt */
   if (Sn_IR[CON] == ‘1’)
   {
      /* clear SOCKET Interrupt */
      Sn_IR[CON] = ‘1’;
      goto Received DATA?; /* or goto Send DATA?; */
   }
   else if(Sn_IR[TIMEOUT] == ‘1’) goto Timeout?;
}

Second method :
{
   if (Sn_SR == SOCK_ESTABLISHED)
   {
      /* clear SOCKET Interrupt */
      Sn_IR[CON] = ‘1’;
      goto Received DATA? /* or goto Send DATA?; */
   }
   else if(Sn_IR[TIMEOUT] == ‘1’) goto Timeout?;
}
```

Receive DATA?

Whether SOCKET n Data is received is confirmed by Sn_IR [RECV] or Sn_RX_RSR.

```
First method :
{
   /* check SOCKET RX Memory Received Size */
   if (Sn_RX_RSR > ) goto Receiving Process;
}

Second method :
```

```
{
   if (Sn_IR[RECV] == ‘1’)
   {
      /* check SOCKET Interrupt */
      Sn_IR[RECV] = ‘1’; /* clear SOCKET Interrupt */
      goto Receiving Process;
   }
}
```

Receiving Process

Received Data is read from SOCKET n RX Buffer Block. The Read Offset Address of Received Data in RX Memory Block is calculated by gSn_RX_BASE, gSn_RX_MASK and Sn_RX_RD. After reading received Data, Sn_RX_RD must be increased by Data read Size and Sn_CR [RECV] must be set to ‘1’. If there is remain Data in SOCKET n RX Buffer Block after Sn_CR [RECV] Command, Sn_IR [RECV] is set to ‘1’. When Read Offset Address calculated, it is cautious to over the boundary Address (n=0,1,2 : gSn_RX_BASE ~ gSn+1_RX_BASE, n=3 : gS3_RX_BASE ~ 0xFFFF) of SOCKET n RX Buffer Block.

```
{
   /* get Received Size */
   get_size = Sn_RX_RSR;

   /* calculate SOCKET n RX Buffer Size & Offset Address */
   gSn_RX_MAX = Sn_RXBUF_SIZE * 1024;
   get_offset = Sn_RX_RD & gSn_RX_MASK;

   /* calculate Read Offset Address */
   get_start_address = gSn_RX_BASE + get_offset;

   /* if overflow the upper boundary of SOCKET n RX Buffer */
   If( (get_offset + get_size) > gSn_RX_MAX )
   {
      /* copy upper_size bytes of get_start_address to destination_address
         - destination_address is user data memory address */
      upper_size = gSn_RX_MAX – get_offset;
      memcpy(get_start_address, destination_address, upper_size);
      destination_address += upper_size;
      /* copy the remained size bytes of gSn_RX_BASE to destination_address */
      remained_size = get_size – upper_size;
      memcpy(gSn_RX_BASE, destination_address, remained_size);
   }
   else
   {
      /* copy get_size of get_start_address to destination_address */
      memcpy(get_start_address, destination_address, get_size);
   }

   /* increase Sn_RX_RD as get_size */
   Sn_RX_RD += get_size;

   /* set RECV Command */
   Sn_CR[RECV] = ‘1’;
   while(Sn_CR != 0x00); /* wait until RECV Command is cleared*/
}
```

Send DATA? / Sending Process

Written Data in SOCKET n TX Buffer Block is transmitted. The Write Offset Address in TX Memory Block is calculated by gSn_TX_BASE, gSn_TX_MASK and Sn_TX_WD. And Data to be transmitted from the Write Offset Address is written. After writing Data, Sn_TX_WD must be increased by Data Size and Data is transmitted by Sn_CR [SEND]. Before Sn_IR [SENDOK] = '1', next Data Transmission Process is not executed. After transmitting Data, the time length until Sn_IR[SENDOK] is depending on SOCKET Count, Data Size and Network Traffic. Also Sn_IR [TIMEOUT] could be occurred. When Write Offset Address calculated, it is cautious to over the boundary Address (n=0,1,2 : gSn_TX_BASE ~ gSn+1_TX_BASE, n=3 : gS3_TX_BASE ~ 0xC000) of SOCKET n TX Buffer Block. If there is no response from Peer against of transmitted Data Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to '1'.

```
{
   /* calculate SOCKETn TX Buffer Size & Offset Address */
   gSn_TX_MAX = Sn_TXBUF_SIZE * 1024;
   get_offset = Sn_TX_WR & gSn_TX_MASK;

   /* check the max size of DATA(send_size) & Free Size of SOCKETn TX
      Buffer(Sn_TX_FSR)*/
   if( send_size >gSn_TX_MAX ) send_size = gSn_TX_MAX;
   while(send <= Sn_TX_FSR); // Wait until SOCKET n TX Buffer is free */

   /* If you don't want to wait TX Buffer Free
      send_size = Sn_TX_FSR; // write Data as size of Free Buffer */
   /* calculate Write Offset Address */
   get_start_address = gSn_TX_BASE + get_offset;

   /* if overflow the upper boundary of SOCKET n TX Buffer */
   If( (get_offset + send_size) > gSn_TX_MAX )
   {
      /* copy upper size bytes of source_address to get_start_address
         - source_address is the start address of user data */
      upper_size = gSn_TX_MAX – get_offset;
      memcpy(source_address, get_start_address, upper_size);

      /* copy the Remained Size Bytes of source_address to gSn_TX_BASE */
      source_address += upper_size;
      remained_size = send_size – upper_size;
      memcpy(source_address, gSn_TX_BASE, remained_size);
   }
   else
   {
      /* copy send_size bytes of source_address to get_start_address
         - source_address is the start address of user data */
      memcpy(source_address, get_start_address, send_size);
   }

   /* increase Sn_TX_WR as send_size */
   Sn_TX_WR += send_size;

   /* set SEND Command */
   Sn_CR = SEND;
   while(Sn_CR != 0x00); /* wait until SEND Command is cleared*/

   /* wait until SEND Command is completed or TIMEOUT Interrupt is occurred*/
```

```
    while(Sn_IR[SENDOK] == ‘’ and Sn_IR[TIMEOUT] = ‘’);

    /* clear SOCKET Interrupt*/
    if(Sn_IR[SENDOK] == ‘1’) Sn_IR[SENDOK] = ‘1’;
    else goto Timeout?;
}
```

Received FIN (Passive Close)

When FIN Packet received from Peer.

```
First Method:
{
    If(Sn_SR == SOCK_CLOSE_WAIT) goto Disconnecting Process;
}

Second Method:
{
    If(Sn_IR[DISCON] == ‘1’) goto Disconnecting Process;
}
```

Disconnected (Active Close)

When FIN Packet transmitted to Peer.

```
{
    /* send FIN Packet */
    Sn_CR[DISCON] = ‘1’;
    while(Sn_CR != 0x00); /* wait until DISCON Command is cleared*/
    goto Disconnecting Process;
}
```

Disconnecting Process

In Passive Close, if FIN Packet is received from Peer and there is no Data to be transmitted, SOCKET transmits FIN Packet and it will be closed. If there is no response from Peer against of transmitted FIN Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to ‘1’. In Active Close, if SOCKET transmits FIN Packet to Peer, SOCKET waits for Peer FIN Packet. SOCKET will be closed after receiving FIN Packet from Peer. If there is no response from Peer against of transmitted FIN Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to ‘1’.

```
Passive Close: /* received FIN Packet from Peer */
{
    /* send FIN Packet */
    Sn_CR = DISCON;
    while(Sn_CR != 0x00); /* wait until DISCON Command is cleared*/

    /* wait unit ACK Packet is received*/
    while(Sn_IR[DISCON] == ‘’ and Sn_IR[TIMEOUT] == ‘’) ;
    if (Sn_IR[DISCON] == ‘1’)
    {
        /* clear Interrupt */
```

```
      Sn_IR[DISCON] = ‘1’;
      goto CLOSED;
   }
   else goto Timeout?;
}

Active Close : /* sent FIN Packet to Peer */
{
   /* wait until FIN Packet is received*/
   while(Sn_IR[DISCON] == ‘’ and Sn_IR[TIMEOUT] == ‘’) ;
   if (Sn_IR[DISOCN] == ‘1’)
   {
      /* clear Interrupt */
      Sn_IR[DISCON] = ‘1’;
      goto CLOSED;
   }
   else goto Timeout?;
}
```

Timeout?

If there is no response from Peer against of transmitted SYN or SYN/ACK or FIN or Data Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to ‘1’.

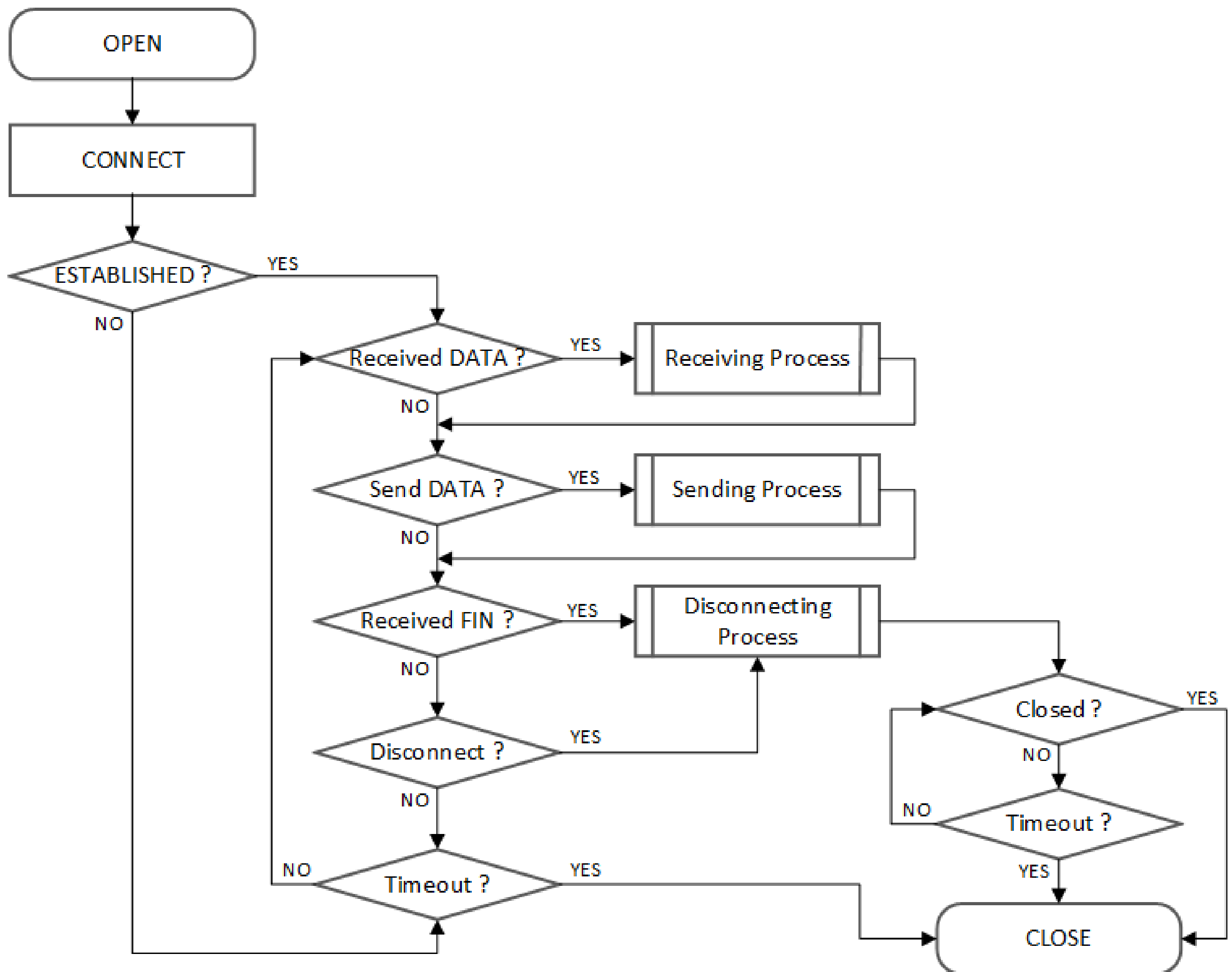
```
{
   /* check TIMEOUT Interrupt */
   if(Sn_IR[TIMEOUT] == ‘1’)
   {
      /* clear Interrupt */
      Sn_IR[TIMEOUT] = ‘1’;
      goto CLOSE;
   }
}
```

CLOSE

SOCKET n is closed by the Disconnect Process, Sn_IR[TIMEOUT] = '1' and Sn_CR[CLOSE] = '1'.

```
{
   /*wait until SOCKET n is closed*/
   while(Sn_SR != SOCK_CLOSED);
}
```

TCP CLIENT



OPEN

It is the same as “TCP SERVER”.

CONNECT

SOCKET n is operated as “TCP CLIENT” by Sn_CR[CONNECT]. SYN Packet is transmitted to “TCP SERVER” by Sn_CR[CONNECT].

```
{
   /* set Destination IP Address, 192.168.0.11 */
   Sn_DIPR[:3] ={ 0xC0, 0xA8, 0x00, 0x0B};

   /* set Destination PORT Number, 5000(0x1388) */
   Sn_DPORTR[:1] = {0x13, 0x88};

   /* set CONNECT Command */
   Sn_CR = CONNECT;
   while(Sn_CR != 0x00); /* wait until CONNECT Command is cleared*/
   goto ESTABLISHED?;
}
```

ESTABLISHED?

“TCP CLIENT” is in Sn_SR (SOCK_SYNSENT) until receiving SYN/ACK Packet from “TCP SERVER” against of SYN Packet transmitted. If SYN/ACK Packet is received from ‘TCP SERVER’, the Connection Process between ‘TCP SERVER’ and ‘TCP CLIENT’ is completed. If there is no response from Peer against of transmitted SYN Packet within the Retransmission Time, Sn_IR [TIMEOUT] is set to ‘1’.

Others flow

Refer to “TCP SERVER” flow.

2017/03/31 16:30 · 이상준

W5100S UDP Function

By setting some register and memory operation, W5100S provides internet connectivity. This chapter describes how it can be operated.

Initialization

Basic Setting

For the W5100S operation, select and utilize appropriate registers shown below.

1. Mode Register (MR)
2. Interrupt Mask Register (IMR)
3. Retry Time-value Register (RTR)
4. Retry Count Register (RCR)

For more information of above registers, refer to the “Register Descriptions.”

Setting network information

Basic network information setting for communication: It must be set the basic network information.

1. SHAR(Source Hardware Address Register)
    - It is prescribed that the source hardware addresses, which is set by SHAR, use unique hardware addresses (Ethernet MAC address) in the Ethernet MAC layer. The IEEE manages the MAC address allocation. The manufacturer which produces the network device allocates the MAC address to product.
    - Details on MAC address allocation refer to the website as below.
    - http://www.ieee.org/
    - http://standards.ieee.org/regauth/oui/index.shtml
2. GAR(Gateway Address Register)
3. SUBR(Subnet Mask Register)
4. SIPR(Source IP Address Register)

Set SOCKET memory information

This stage sets the socket TX/RX memory information. The base address and mask address of each SOCKET are fixed and saved in this stage.

In case of, assign 2Kbytes TX/RX memory per SOCKET

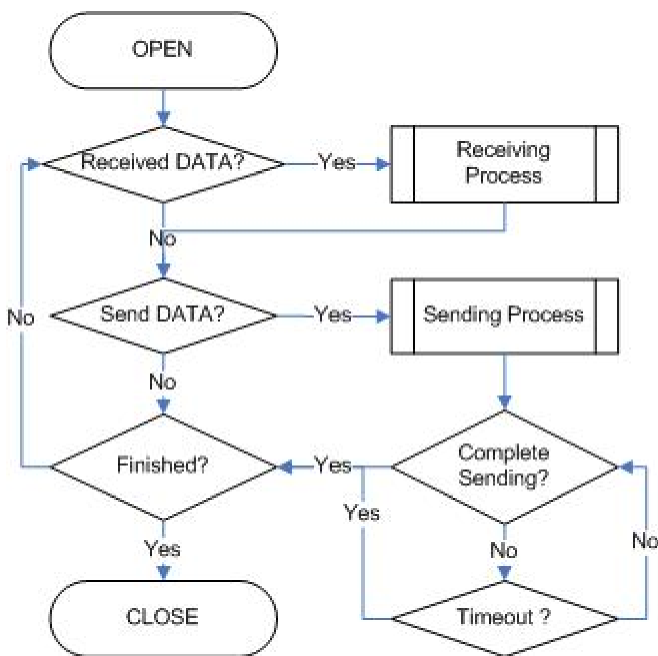
```
In case of, assign 2Kbytes TX/RX memory per SOCKET
{
Sn_RXMEM_SIZE(ch) = (uint8 *) 2; // Assign 2K rx memory per SOCKET
Sn_TXMEM_SIZE(ch) = (uint8 *) 2; // Assign 2K rx memory per SOCKET

/* Same method, set gS1_TX_BASE, gS1_TX_MASK, gS2_TX_BASE, gS2_TX_MASK,
gS3_TX_BASE, gS3_TX_MASK, gS4_TX_BASE, gS4_TX_MASK*/
}
```

Data Communications

After the initialization process, W5100S can transmit and receive the data with others by 'open' the SOCKET of TCP, UDP, IPRAW, and MACRAW mode. The W5100S supports the independently and simultaneously usable 4 SOCKETs. In this section, the communication method for each mode will be introduced.

UDP

The UDP is a Connection-less protocol. It communicates without "connection SOCKET". The TCP protocol guarantees reliable data communication, but the UDP protocol uses datagram communication which has no guarantees of data communication. Because the UDP does not use "connection SOCKET", it can communicate with many other devices with the known host IP address and port number. This is a great advantage, communication with many others by using just one SOCKET. But also it has many problems such as loss of transmitted data, unwanted data received from others, etc. To avoid these problems and guarantee reliability, the host retransmits damaged data or ignores the unwanted data which is received from others. The UDP protocol supports unicast, broadcast, and multicast communication. It follows the below communication flow.



Unicast and Broadcast

The unicast is one method of UDP communication. It transmits data to one destination at one time. On the other hand, the broadcast communication transmits data to all receivable destinations by using 'broadcast IP address (255.255.255.255)'. For example, suppose that the user transmits data to destination A, B, and C. The unicast communication transmits each destination A, B, and C at each time. At this time, the ARP_{TO} can also occur when the user gets the destination hardware address of destinations A, B and C. User cannot transmit data to destinations which have ARP_{TO}. The broadcast communication can simultaneously transmit data to destination A, B and C at one time by using "255.255.255.255" or "local address | (~subnet address)" IP address. At this time, there is no need to get the destination hardware address about destination A, B and C, and also $ARP_{TO}$is not occurred.

Note: Broadcast IP

⇒ The Broadcast IP address can be obtained by performing a bit-wise logical OR operation between the bit complement of the subnet mask and the host's IP address.
ex> If IP:"222.98.173.123" and the subnet mask:"255.255.255.0", broadcast IP is "222.98.173.255"

Description	Decimal	Binary
HOST IP	222.098.173.123	11011110.01100010.10101101.01111011
Bit Complement Subnet mask	000.000.000.255	00000000.00000000.00000000.11111111
Bitwise OR	-	-
Broadcast IP	222.098.173.255	11011110.01100010.10101101.11111111

SOCKET Initialization

For the UDP data communication, SOCKET initialization is required; it opens the SOCKET. The SOCKET open process is as followed. At first, choose one SOCKET among the 4 SOCKETS of W5100S, then set the protocol mode (Sn_MR(P3:P0)) of the chosen SOCKET and set the source port number Sn_PORT0 for communication. Finally, execute the OPEN command. After the OPEN command, the state of Sn_SR is changed to SOCK_UDP. Then the SOCKET initialization is complete.

```
{
START:
Sn_MR = 0x02; /* sets UDP mode */
Sn_PORT0 = source_port; /* sets source port number */
Sn_CR = OPEN; /* sets OPEN command */
/* wait until Sn_SR is changed to SOCK_UDP */
if (Sn_SR != SOCK_UDP) Sn_CR = CLOSE; goto START;
}
```

Check received data

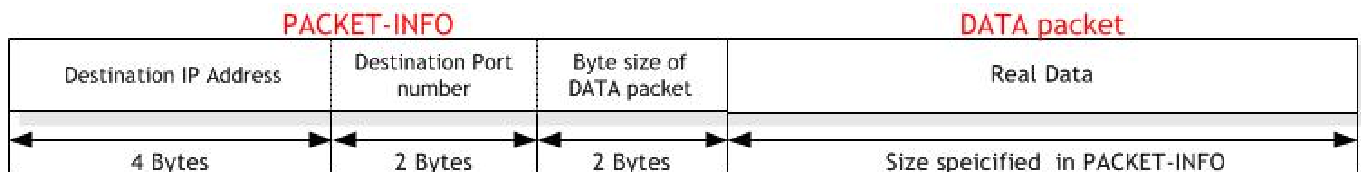
Check the reception of UDP data from destination. User can also check for received data via TCP communication. It is strongly recommended to use the second method because of the same reasoning from TCP. Please refer to the "TCP SERVER" section. TCP SERVER

```
First method :
{
if (Sn_IR(RECV) == '1') Sn_IR(RECV) = '1'; goto Receiving Process stage;
/* In this case, if the interrupt of Socket n is activated, interrupt occurs. Refer
```

```
to IR, IMR
Sn_IMR and Sn_IR. */
}
Second Method :
{
if (Sn_RX_RSR0 != 0x0000) goto Receiving Process stage;
}
```

Receiving process

Process the received UDP data in Internal RX memory.
The structure of received UDP data is as below.



The received UDP data consists of 8bytes PACKET-INFO, and DATA packet. The PACKET-INFO contains transmitter's information (IP address, Port number) and the length of DATA packet. The UDP can receive UDP data from many others. User can classify the transmitter by transmitter's information of PACKET-INFO. It also receives broadcast SOCKET by using "255.255.255.255" IP address. So the host should ignore unwanted reception by analysis of transmitter's information. If the DATA size of SOCKET n is larger than Internal RX memory free size, user cannot receive that DATA and also cannot receive fragmented DATA.

```
{
/* Get offset address */
src_ptr = Sn_RX_RD;
/* select RX memory, refer to RMSR(Rx Memory Size Register) */
cntl_byte = Socket_n_RX_Buffer
/* read head information (8 bytes) */
header_size = 8;
/* copy header_size bytes of get_start_address to header_address */
for(i=; i<header_size; i++)
{
  header[i] = W5100S_READ(src_ptr, header);
}
/* update src_ptr */
src_ptr += header_size;

/* save remote peer information & received data size */
peer_ip = header[ to 3];
peer_port = header[4 to 5];
get_size = header[6 to 7];

/* copy len bytes of src_ptr to destination_address */
for(i=; i<get_size; i++)
{
  *(dst_ptr+i) = W5100S_READ(addr, cntl_byte, src_ptr+1);
}
/* increase Sn_RX_RD as length of len+ header_size */
Sn_RX_RD += get_size;
/* set RECV command */
Sn_CR = RECV;
```

```
}
```

Check send data / sending process

The size of DATA that the user wants to transmit cannot be larger than Internal TX memory. If it is larger than MTU, it is automatically divided by MTU unit and transmitted. The Sn_DIPR0 is set “255.255.255.255” when user wants to broadcast.

```
{
/* first, get the free TX memory size */
FREESIZE:
freesize = Sn_TX_FSR0;
if (freesize<len) goto FREESIZE; // len is send size

/* Write the value of remote_ip, remote_port to the Socket n Destination IP Address
Register(Sn_DIPR), Socket n Destination Port Register(Sn_DPORT). */
Sn_DIPR0 = remote_ip;
Sn_DPORT0 = remote_port;

/* Get offset address */
dst_ptr = Sn_TX_WR;
/* select TX memory, refer to TMSR(Tx Memory Size Register) */
cntl_byte = Socket_n_TX_Buffer
/* copy len bytes of source_address to dst_ptr */
for(i=; i<len; i++)
{
  W5100S_WRITE(addr, cntl_byte, dst_ptr+i);
}
/* increase Sn_TX_WR0 as length of len */
Sn_TX_WR += len;
/* set SEND command */
Sn_CR = SEND;
}
```

Check complete sending / Timeout

To transmit the next data, user must check that the prior SEND command is completed. The larger the data size, the more time to complete the SEND command. Therefore, the user must properly divide the data to transmit. The ARP_{TO} can occur when user transmits UDP data. If ARP_{TO} occurs, the UDP data transmission has failed.

```
First method :
{
/* check SEND command completion */
while(Sn_IR(SENDOK)==‘’) /* wait interrupt of SEND completion */
{
/* check ARPTO */
if (Sn_IR(TIMEOUT)==‘1’) Sn_IR(TIMEOUT)=‘1’; goto Next stage;
}
Sn_IR(SENDOK) = ‘1’; /* clear previous interrupt of SEND completion */
}
Second method :
{
```

```
If (Sn_CR == 0x00) transmission is completed.
If (Sn_IR(TIMEOUT bit) == '1') goto next stage;
/* In this case, if the interrupt of Socket n is activated, interrupt occurs. Refer
to
Interrupt Register(IR), Interrupt Mask Register (IMR) and Socket n Interrupt
Register (Sn_IR).
*/
}
```

Check Finished / SOCKET close

If user doesn't need the communication any more, close the SOCKET n.

```
{
/* clear remained interrupts */
Sn_IR = 0x00FF;
IR(n) = '1';
/* set CLOSE command */
Sn_CR = CLOSE;
}
```

Multicast

The broadcast communication communicates with many and unspecified others. But the multicast communication communicates with many specified others who registered at a multicast-group. Suppose that A, B, and C are registered at a specified multicast-group. If user transmits data to multicast-group (contains A), B and C also receive the DATA for A. To use multicast communication, the destination list registers to multicast-group by using IGMP protocol. The multicast-group consists of 'Group hardware address,' 'Group IP address,' and 'Group port number.' User cannot change the 'Group hardware address' and 'Group IP address.' But the 'Group port number' can be changed.
The 'Group hardware address' is selected at the assigned range (From "01:00:5e:00:00:00"to "01:00:5e:7f:ff:ff") and the 'Group IP address' is selected in D-class IP address (From "224.0.0.0" to "239.255.255.255", please refer to the website; http://www.iana.org/assignments/multicast-addresses).
When selecting, the upper 23bits of 6bytes 'Group hardware address' and the 4bytes 'Group IP address' must be the same. For example, if the user selects the 'Group IP address' to "244.1.1.11," the 'Group hardware address' is selected to "01:00:5e:01:01:0b." Please refer to the "RFC1112" (http://www.ietf.org/rfc.html).
In the W5100S, IGMP processing to register the multicast-group is internally (automatically) processed. When the user opens the SOCKET n with multicast mode, the "Join" message is internally transmitted. If the user closes it, the "Leave" message is internally transmitted. After the SOCKET opens, the "Report" message is periodically and internally transmitted when the user communicates.
The W5100S support IGMP version 1 and version 2 only. If user wants use an updated version, the host processes IGMP directly by using the IPRAW mode SOCKET.

SOCKET Initialization

Choose one SOCKET for multicast communication among 4 SOCKETs of W5100S. Set the Sn_DHAR0 to 'Multicast-group hardware address' and set the Sn_DIPR0 to 'Multicastgroup IP address.' Then set the Sn_PORT0 and Sn_DPORT0 to 'Multicast-group port number.' Set the Sn_MR(P3:P0) to UDP and set the Sn_MR(MULTI) to '1.' Finally, execute OPEN command. If the state of Sn_SR is changed to SOCK_UDP after the OPEN command, the SOCKET initialization is completed.

```
{
```

```
START:
/* set Multicast-Group information */
/* set Multicast-Group H/W address(01:00:5e:01:01:0b) */
Sn_DHAR[:5] = {0x01,0x00,0x5e,0x01,0x01,0x0b};

/* set Multicast-Group IP address(211.1.1.11) */
Sn_DIPR[:3] = {211,1,1,11};

Sn_DPORT[:1] ={0x0B,0xB8}; /* set Multicast-GroupPort number(3000) */
Sn_PORT[:1] = {0x0B,0xB8}; /* set SourcePort number(3000) */
Sn_MR = 0x02 | 0x80; /* set UDP mode & Multicast on Socket n Mode Register */
Sn_CR = OPEN; /* set OPEN command */
/* wait until Sn_SR is changed to SOCK_UDP */
if (Sn_SR != SOCK_UDP) Sn_CR = CLOSE; goto START;
}
```

Check received data

Refer to the “Unicast & Broadcast.” section.

Unicast & Broadcast

Receiving process

Refer to the “Unicast & Broadcast.” section. Unicast & Broadcast

Check send data / Sending Process

Since the user sets the information about multicast-group at SOCKET initialization, user does not need to set IP address and port number for destination any more. Therefore, copy the transmission data to internal TX memory and executes SEND command.

```
{
/* first, get the free TX memory size */
FREESIZE:
freesize = Sn_TX_FSR;
if (freesize<len) goto FREESIZE; // len is send size
/* calculate offset address */
dst_mask = Sn_TX_WR0 &gSn_TX_MASK; // dst_mask is offset address
/* calculate start address(physical address) */
dst_ptr = gSn_TX_BASE + dst_mask; // dst_ptr is physical start address
/* if overflow SOCKETTX memory */
if ( (dst_mask + len) > (gSn_TX_MASK + 1) )
{
/* copy upper_size bytes of source_addr to destination_address */
upper_size = (gSn_TX_MASK + 1) ? dst_mask;
memcpy((0x0000 + source_addr), (0x0000 + dst_ptr), upper_size);
/* update source_addr*/
source_addr += upper_size;
/* copy left_size bytes of source_addr to gSn_TX_BASE */
left_size = len ? upper_size;
memcpy( source_addr, gSn_TX_BASE, left_size);
```

```
}
else
{
/* copy len bytes of source_addr to dst_ptr */
memcpy( source_addr, dst_ptr, len);
}
/* increase Sn_TX_WR as length of len */
Sn_TX_WR0 += send_size;
/* set SEND command */
Sn_CR = SEND;
}
```

Check complete sending / Timeout

Since the host manages all protocol process for data communication, timeout cannot occur.

```
{
/* check SEND command completion */
while(S0_IR(SENDOK)==‘’); /* wait interrupt of SEND completion */
S0_IR(SENDOK) = ‘1’; /* clear previous interrupt of SEND completion */
}
```

Check finished / SOCKET close

Refer to the “Unicast & Broadcast.” section. Unicast & Broadcast

2017/03/31 16:31 · 이상준

---

comparison

From:
http://wizwiki.net/wiki/ -
Document Wiki

Permanent link:
http://wizwiki.net/wiki/doku.php/products:w5100s:allpages

Last update: 2018/12/24 09:59