1.245 SocketAccept - Accept an incoming connection

Usage

`SocketAccept` is used to accept incoming connection requests. `SocketAccept` can only be used for server applications.

Basic examples

The following example illustrates the instruction `SocketAccept`:

See also *More examples on page 700*.

Example 1

```
VAR socketdev server_socket;
VAR socketdev client_socket;
...
SocketCreate server_socket;
SocketBind server_socket,"192.168.0.1", 1025;
SocketListen server_socket;
SocketAccept server_socket, client_socket;
```

A server socket is created and bound to port `1025` on the controller network address `192.168.0.1`. After execution of `SocketListen` the server socket starts to listen for incoming connections on this port and address. `SocketAccept` waits for any incoming connections, accepts the connection request, and returns a client socket for the established connection.

Arguments

```
SocketAccept Socket ClientSocket [\ClientAddress] [ \Time ]
```

`Socket`

Data type: `socketdev`

The server socket that are waiting for incoming connections. The socket must already be created, bounded, and ready for listening.

`ClientSocket`

Data type: `socketdev`

The returned new client socket that will be updated with the accepted incoming connection request.

`[\ClientAddress]`

Data type: `string`

The variable that will be updated with the IP-address of the accepted incoming connection request.

`[\Time]`

Data type: `num`

The maximum amount of time [s] that program execution waits for incoming connections. If this time runs out before any incoming connection then the error handler will be called, if there is one, with the error code `ERR_SOCK_TIMEOUT`. If there is no error handler then the execution will be stopped.

Continues on next page

1.245 SocketAccept - Accept an incoming connection
Socket Messaging
Continued

If parameter \Time is not used then the waiting time is 60 s. To wait forever, use the predefined constant WAIT_MAX.

Program execution

The server socket will wait for any incoming connection requests. When accepting the incoming connection request the instruction is ready and the returned client socket is by default connected and can be used in SocketSend and SocketReceive instructions.

More examples

More examples of the instruction SocketAccept are illustrated below.

Example 1

```
VAR socketdev server_socket;
VAR socketdev client_socket;
VAR string receive_string;
VAR string client_ip;
...
SocketCreate server_socket;
SocketBind server_socket, "192.168.0.1", 1025;
SocketListen server_socket;
WHILE TRUE DO
  SocketAccept server_socket, client_socket
       \ClientAddress:=client_ip;
  SocketReceive client_socket \Str := receive_string;
  SocketSend client_socket \Str := "Hello client with ip-address
       " +client_ip;
  ! Wait for client acknowledge
  ...
  SocketClose client_socket;
ENDWHILE
ERROR
  RETRY;
UNDO
  SocketClose server_socket;
  SocketClose client_socket;
```

A server socket is created and bound to port 1025 on the controller network address 192.168.0.1. After execution of SocketListen the server socket starts to listen for incoming connections on this port and address. SocketAccept will accept the incoming connection from some client and store the client address in the string client_ip. Then the server receives a string message from the client and stores the message in receive_string. Then the server responds with the message "Hello client with ip-address xxx.xxx.x.x" and closes the client connection.

After that the server is ready for a connection from the same or some other client in the WHILE loop. If PP is moved to main in the program then all open sockets are closed (SocketClose can always be done even if the socket is not created).

Continues on next page

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

Name	Cause of error
ERR_SOCK_CLOSED	The socket is closed (has been closed or is not created). Use SocketCreate to create a new socket.
ERR_SOCK_TIMEOUT	The connection was not established within the time out time

Syntax

```
SocketAccept
  [ Socket ':=' ] < variable (VAR) of socketdev > ','
  [ ClientSocket ':=' ] < variable (VAR) of socketdev >
  [ '\' ClientAddress ':=' < variable (VAR) of string> ]
  [ '\' Time ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*, section *Socket Messaging*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Example of server socket application	*SocketReceive - Receive data from remote computer on page 713*

1.246 SocketBind - Bind a socket to my IP-address and port

Usage

`SocketBind` is used to bind a socket to the specified server IP-address and port number. `SocketBind` can only be used for server applications.

Basic examples

The following example illustrates the instruction `SocketBind`:

Example 1

```
VAR socketdev server_socket;

SocketCreate server_socket;
SocketBind server_socket, "192.168.0.1", 1025;
```

A server socket is created and bound to port `1025` on the controller network address `192.168.0.1`. The server socket can now be used in an `SocketListen` instruction to listen for incoming connections on this port and address.

Arguments

```
SocketBind Socket LocalAddress LocalPort
```

`Socket`

Data type: `socketdev`

The server socket to bind. The socket must be created but not already bound.

`LocalAddress`

Data type: `string`

The server network address to bind the socket to. The only valid addresses are any public WAN addresses or the controller service port address 192.168.125.1.

`LocalPort`

Data type: `num`

The server port number to bind the socket to. Generally ports 1025-4999 are free to use.

Program execution

The server socked is bound to the specified server port and IP-address.

An error is generated if the specified port is already in use.

Use the `SocketBind` and `SocketListen` instructions in the startup of the program to associate a local address with a socket and then listen for incoming connections on the specified port. This is recommended to do only once for each socket and port that is used (TCP/IP).

Use the `SocketBind` instruction if receiving data with `SocketReceiveFrom` (UDP/IP).

Continues on next page

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

Name	Cause of error
ERR_SOCK_CLOSED	The socket is closed (has been closed or is not created) Use SocketCreate to create a new socket.
ERR_SOCK_ADDR_INUSE	The address and port is already in use and cannot be used again. Use a different port number.

Syntax

```
SocketBind
  [ Socket ':=' ] < variable (VAR) of socketdev > ','
  [ LocalAddress ':=' ] < expression (IN) of string > ','
  [ LocalPort ':=' ] < expression (IN) of num > ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*
Receive data from remote computer	*SocketReceiveFrom - Receive data from remote computer on page 718*

1.247 SocketClose - Close a socket

Usage

`SocketClose` is used when a socket connection is no longer going to be used.

After a socket has been closed it cannot be used in any socket call except `SocketCreate`.

Basic examples

The following example illustrates the instruction `SocketClose`:

Example 1

```
SocketClose socket1;
```

The socket is closed and cannot be used anymore.

Arguments

```
SocketClose Socket
```

`Socket`

Data type: `socketdev`

The socket to be closed.

Program execution

The socket will be closed and its allocated resources will be released.

Any socket can be closed at any time. The socket cannot be used after closing. It can be reused for a new connection after a call to `SocketCreate`.

Limitations

Closing the socket connection immediately after sending the data with `SocketSend` can lead to loss of sent data. This is because `TCP/IP` socket has built-in functionality to resend the data if there is some communication problem.

To avoid such problems with loss of data, do the following before `SocketClose`:

- handshake the shutdown or
- `WaitTime` 2

Avoid fast loops with `SocketCreate ... SocketClose`, because the socket is not really closed until a certain time (TCP/IP functionality).

Syntax

```
SocketClose
  [ Socket ':=' ] < variable (VAR) of socketdev > ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*, section *Socket Messaging*
Create a new socket	*SocketCreate - Create a new socket on page 709*

Continues on next page

For information about	See
Connect to a remote computer (onlyclient)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*t
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Send data to remote computer	*SocketSendTo - Send data to remote computer on page 727*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*
Receive data from remote computer	*SocketReceiveFrom - Receive data from remote computer on page 718*

1.248 SocketConnect - Connect to a remote computer

Usage

`SocketConnect` is used to connect the socket to a remote computer in a client application.

Basic examples

The following example illustrates the instruction `SocketConnect`:

See also *More examples on page 707*.

Example 1

```
SocketConnect socket1, "192.168.0.1", 1025;
```

Trying to connect to a remote computer at ip-address `192.168.0.1` and port `1025`.

Arguments

```
SocketConnect Socket Address Port [\Time]
```

`Socket`

Data type: `socketdev`

The client socket to connect. The socket must be created but not already connected.

`Address`

Data type: `string`

The address of the remote computer. The remote computer must be specified as an IP address. It is not possible to use the name of the remote computer.

`Port`

Data type: `num`

The port on the remote computer. Generally ports 1025-4999 are free to use. Ports below 1025 can already be taken.

`[ \Time ]`

Data type: `num`

The maximum amount of time [s] that program execution waits for the connection to be accepted or denied. If this time runs out before the condition is met then the error handler will be called, if there is one, with the error code `ERR_SOCK_TIMEOUT`. If there is no error handler then the execution will be stopped.

If parameter `\Time` is not used the waiting time is 60 s. To wait forever, use the predefined constant `WAIT_MAX`.

Program execution

The socket tries to connect to the remote computer on the specified address and port. The program execution will wait until the connection is established, failed, or a timeout occurs.

Continues on next page

More examples

More examples of the instruction `SocketConnect` are illustrated below.

Example 1

```
VAR num retry_no := 0;
VAR socketdev my_socket;
...
SocketCreate my_socket;
SocketConnect my_socket, "192.168.0.1", 1025;
...
ERROR
  IF ERRNO = ERR_SOCK_TIMEOUT THEN
    IF retry_no < 5 THEN
      WaitTime 1;
      retry_no := retry_no + 1;
      RETRY;
    ELSE
      RAISE;
    ENDIF
  ENDIF
```

A socket is created and tries to connect to a remote computer. If the connection is not established within the default time-out time, i.e. 60 seconds, then the error handler retries to connect. Four retries are attemped then the error is reported to the user.

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`ERR_SOCK_CLOSED`	The socket is closed (has been closed or is not created). Use `SocketCreate` to create a new socket.
`ERR_SOCK_TIMEOUT`	The connection was not established within the time-out time.

Syntax

```
SocketConnect
  [ Socket ':=' ] < variable (VAR) of socketdev > ','
  [ Address ':=' ] < expression (IN) of string > ','
  [ Port ':=' ] < expression (IN) of num >
  [ '\' Time ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	Described in:
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*

Continues on next page

1.248 SocketConnect - Connect to a remote computer

Socket Messaging

Continued

For information about	Described in:
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*

1.249 SocketCreate - Create a new socket

Usage

`SocketCreate` is used to create a new socket for connection based communication or connectionless communication.

Both socket messaging of stream type protocol TCP/IP with delivery guarantee and datagram protocol UDP/IP is supported. Both server and client application can be developed. For datagram protocol UDP/IP, broadcast is supported.

Basic examples

The following example illustrates the instruction `SocketCreate`:

Example 1

```
VAR socketdev socket1;
...
SocketCreate socket1;
```

A new socket device using stream type protocol TCP/IP is created and assigned into the variable `socket1`.

Example 2

```
VAR socketdev udp_sock1;
...
SocketCreate udp_sock1 \UDP;
```

A new socket device using datagram protocol UDP/IP is created and assigned into the variable `udp_sock1`.

Arguments

```
SocketCreate Socket [\UDP]
```

`Socket`

Data type: `socketdev`

The variable for storage of the system's internal socket data.

`[ \UDP ]`

Data type: `switch`

Specifies that the socket should be of the type datagram protocol UDP/IP.

Program execution

The instruction creates a new socket device.

The socket must not already be in use. The socket is in use between `SocketCreate` and `SocketClose`.

Limitations

Any number of sockets can be declared but it is only possible to use 32 sockets at the same time.

Avoid fast loops with `SocketCreate ... SocketClose`, because the socket is not really closed until after a certain time (when using TCP/IP functionality).

Continues on next page

1.249 SocketCreate - Create a new socket

Socket Messaging

Continued

Syntax

```
SocketCreate
  [ Socket ':=' ] < variable (VAR) of socketdev >
  [ '\' UDP ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*, section *Socket Messaging*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Send data to remote computer	*SocketSendTo - Send data to remote computer on page 727*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*
Receive data from remote computer	*SocketReceiveFrom - Receive data from remote computer on page 718*

1.250 SocketListen - Listen for incoming connections

Usage

`SocketListen` is used to start listening for incoming connections, i.e. start acting as a server. `SocketListen` can only used for server applications.

Basic examples

The following example illustrates the instruction `SocketListen`:

Example 1

```
VAR socketdev server_socket;
VAR socketdev client_socket;
...
SocketCreate server_socket;
SocketBind server_socket, "192.168.0.1", 1025;
SocketListen server_socket;
WHILE listening DO;
  ! Waiting for a connection request
  SocketAccept server_socket, client_socket;
```

A server socket is created and bound to port `1025` on the controller network address `192.168.0.1`. After execution of `SocketListen` the server socket starts to listen for incoming connections on this port and address.

Arguments

```
SocketListen Socket
```

`Socket`

Data type: `socketdev`

The server socket that should start listening for incoming connections. The socket must already be created and bound.

Program execution

The server socket start listening for incoming connections. When the instruction is ready the socket is ready to accept an incoming connection.

Use the `SocketBind` and `SocketListen` instructions in the startup of the program to associate a local address with a socket and then listen for incoming connections on the specified port. This is recommended to do only once for each socket and port that is used.

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`ERR_SOCK_CLOSED`	The socket is closed (has been closed or is not created). Use `SocketCreate` to create a new socket.

Continues on next page

1.250 SocketListen - Listen for incoming connections

Socket Messaging

Continued

Syntax

```
SocketListen
  [ Socket ':=' ] < variable (VAR) of socketdev > ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*

1.251 SocketReceive - Receive data from remote computer

Usage

`SocketReceive` is used for receiving data from a remote computer.
`SocketReceive` can be used both for client and server applications.

Basic examples

The following example illustrates the instruction `SocketReceive`:

See also *More examples on page 715*.

Example 1

```
VAR string str_data;
...
SocketReceive socket1 \Str := str_data;
```

Receive data from a remote computer and store it in the string variable `str_data`.

Arguments

```
SocketReceive Socket [ \Str ] | [ \RawData ] | [ \Data ]
     [\ReadNoOfBytes] [\NoRecBytes] [\Time]
```

`Socket`

Data type: `socketdev`

In a client application where the socket receives the data, the socket must already be created and connected.

In a server application where the socket receives the data, the socket must already be accepted.

`[ \Str ]`

Data type: `string`

The variable in which the received `string` data should be stored. Max. number of characters 80 can be handled.

`[ \RawData ]`

Data type: `rawbytes`

The variable in which the received rawbytes data should be stored. Max. number of `rawbytes` 1024 can be handled.

`[ \Data ]`

Data type: `array of byte`

The variable in which the received byte data should be stored. Max. number of `byte` 1024 can be handled.

Only one of the optional parameters `\Str`, `\RawData`, and `\Data` can be used at the same time.

`[ \ReadNoOfBytes ]`

Read number of Bytes

Data type: `num`

Continues on next page

1.251 SocketReceive - Receive data from remote computer
Socket Messaging
Continued

The number of bytes to read. The minimum value of bytes to read is 1, and the maximum amount is the value of the size of the data type used, i.e. 80 bytes if using a variable of the data type `string`.

If communicating with a client that always sends a fixed number of bytes, this optional parameter can be used to specify that the same amount of bytes should be read for each `SocketReceive` instruction.

If the sender sends RawData, the receiver needs to specify that 4 bytes should be received for each `rawbytes` sent.

`[ \NoRecBytes ]`

Number Received Bytes

Data type: `num`

Variable for storage of the number of bytes needed from the specified `socketdev`.

The same result can also be achieved with

- function `StrLen` on varable in argument `\Str`
- function `RawBytesLen` on variable in argument `\RawData`

`[ \Time ]`

Data type: `num`

The maximum amount of time [s] that program execution waits for the data to be received. If this time runs out before the data is transferred then the error handler will be called, if there is one, with the error code `ERR_SOCK_TIMEOUT`. If there is no error handler then the execution will be stopped.

If parameter `\Time` is not used then the waiting time is 60 s. To wait forever, use the predefined constant `WAIT_MAX`.

Program execution

The execution of `SocketReceive` will wait until the data is available or fail with a timeout error.

The amount of bytes read is specified by the data type used in the instruction. If using a `string` data type to receive data in, 80 bytes is received if there is 80 bytes that can be read. If using optional argument `ReadNoOfBytes` the user can specify how many bytes that should be received for each `SocketReceive`.

The data that is transferred on the cable is always bytes, max. 1024 bytes in one message. No header is added by default to the message. The usage of any header is reserved for the actual application.

Parameter	Input data	Cable data	Output data
`\Str`	1 char	1 byte (8 bits)	1 char
`\RawData`	1 rawbytes	1 byte (8 bits)	1 rawbytes
`\Data`	1 byte	1 byte (8 bits)	1 byte

It is possible to mix the used data type (`string`, `rawbytes`, or `array of byte`) between `SocketSend` and `SocketReceive`.

Continues on next page

More examples

More examples of the instruction `SocketReceive` are illustrated below.

Example 1

```
VAR socketdev server_socket;
VAR socketdev client_socket;
VAR string client_ip;

PROC server_messaging()
  VAR string receive_string;
  ...
  ! Create, bind, listen and accept of sockets in error handlers
  SocketReceive client_socket \Str := receive_string;
  SocketSend client_socket \Str := "Hello client with
       ip-address"+client_ip;
  ! Wait for acknowlegde from client
  ...
  SocketClose server_socket;
  SocketClose client_socket;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    server_recover;
    RETRY;
  ELSE
  ! No error recovery handling
  ENDIF
ENDPROC

PROC server_recover()
  SocketClose server_socket;
  SocketClose client_socket;
  SocketCreate server_socket;
  SocketBind server_socket, "192.168.0.1", 1025;
  SocketListen server_socket;
  SocketAccept server_socket,
       client_socket\ClientAddress:=client_ip;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    RETURN;
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC
```

This is an example of a server program with creation, binding, listening, and accepting of sockets in error handlers. In this way the program can handle power fail restart.

Continues on next page

1.251 SocketReceive - Receive data from remote computer

Socket Messaging
Continued

In the procedure `server_recover`, a server socket is created and bound to port `1025` on the controller network address `192.168.0.1`. After execution of `SocketListen` the server socket starts to listen for incoming connections on this port and address. `SocketAccept` will accept the incoming connection from some client and store the client address in the string `client_ip`.

In the communication procedure `server_messaging` the server receives a string message from the client and stores the message in `receive_string`. Then the server responds with the message `"Hello client with ip-address xxx.xxx.x.x"`.

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`ERR_SOCK_CLOSED`	The socket is closed. Broken connection.
`ERR_SOCK_TIMEOUT`	No data was received within the time out time.

Limitations

There is no built-in synchronization mechanism in Socket Messaging to avoid received messages that are compounded of several sent messages. It is up to the programmer to handle the synchronization with “Ack” messages (one sequence of `SocketSend - SocketReceive` in the client or server program must be completed before next sequence of `SocketSend - SocketReceive`).

All sockets are closed after power fail restart. This problem can be handled by error recovery. See example above.

Avoid fast loops with `SocketCreate ... SocketClose` because the socket is not really closed until a certain time (TCP/IP functionality).

The maximum size of the data that can be received in one call is limited to 1024 bytes.

Syntax

```
SocketReceive
  [ Socket ':=' ] < variable (VAR) of socketdev >
  [ '\' Str ':=' < variable (VAR) of string > ]
  | [ '\' RawData ':=' < variable (VAR) of rawbytes > ]
  | [ '\' Data ':=' < array {*} (VAR) of byte > ]
  [ '\' ReadNoOfBytes ':=' < expression (IN) of num > ]
  [ '\' NoRecBytes ':=' < variable (VAR) of num > ]
  [ '\' Time ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*

Continues on next page

For information about	See
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Test for the presence of data on a socket.	*SocketPeek - Test for the presence of data on a socket on page 1394*

1.252 SocketReceiveFrom - Receive data from remote computer

Usage

`SocketReceiveFrom` is used for receiving data from a remote computer. `SocketReceiveFrom` can be used both for client and server applications.

`SocketReceiveFrom` is used for connectionless communication with datagram protocol UDP/IP.

Basic examples

The following example illustrates the instruction `SocketReceiveFrom`:

See also *More examples on page 715*.

Example 1

```
VAR string str_data;
VAR string RemoteAddress;
VAR num RemotePort;
...
SocketCreate \UDP;
SocketBind myUDPsock, "192.168.9.100", 4044;
SocketReceiveFrom socket1 \Str := str_data, RemoteAddress,
     RemotePort;
```

Receive data from a remote computer and store it in the string variable `str_data`. The address of the remote computer is stored in the string variable `RemoteAddress` and the port number is stored in the num variable `RemotePort`.

Arguments

```
SocketReceiveFrom Socket [ \Str ] | [ \RawData ] | [ \Data ]
     [\NoRecBytes] RemoteAddress RemotePort [\Time]
```

`Socket`

Data type: `socketdev`

A socket device identifying a bound socket.

`[ \Str ]`

Data type: `string`

The variable in which the received `string` data should be stored. Max. number of characters 80 can be handled.

`[ \RawData ]`

Data type: `rawbytes`

The variable in which the received rawbytes data should be stored. Max. number of `rawbytes` 1024 can be handled.

`[ \Data ]`

Data type: `array of byte`

The variable in which the received byte data should be stored. Max. number of `byte` 1024 can be handled.

Continues on next page

Only one of the optional parameters `\Str`, `\RawData`, and `\Data` can be used at the same time.

`[ \NoRecBytes ]`

Number Received Bytes

Data type: `num`

Variable for storage of the number of bytes needed from the specified `socketdev`.

The same result can also be achieved with

- function `StrLen` on varable in argument `\Str`
- function `RawBytesLen` on variable in argument `\RawData`

`RemoteAddress`

Data type: `string`

A string variable containing the source address of the remote computer.

`RemotePort`

Data type: `num`

A num variable containing the port used by the remote computer when sending the datagram package.

`[ \Time ]`

Data type: `num`

The maximum amount of time [s] that program execution waits for the data to be received. If this time runs out before the data is transferred then the error handler will be called, if there is one, with the error code `ERR_SOCK_TIMEOUT`. If there is no error handler then the execution will be stopped.

If parameter `\Time` is not used then the waiting time is 60 s. To wait forever, use the predefined constant `WAIT_MAX`.

Program execution

The execution of `SocketReceiveFrom` receives a datagram and stores the source address and source port. It will wait until the data is available or fail with a timeout error.

The amount of bytes read is specified by the data type used in the instruction. If using a `string` data type to receive data in, 80 bytes is received if there is 80 bytes that can be read.

The data that is transferred on the cable is always bytes, max. 1024 bytes in one message. No header is added by default to the message. The usage of any header is reserved for the actual application.

Parameter	Input data	Cable data	Output data
`\Str`	1 char	1 byte (8 bits)	1 char
`\RawData`	1 rawbytes	1 byte (8 bits)	1 rawbytes
`\Data`	1 byte	1 byte (8 bits)	1 byte

It is possible to mix the used data type (`string`, `rawbytes`, or `array of byte`) between `SocketSendTo` and `SocketReceiveFrom`.

Continues on next page

1.252 SocketReceiveFrom - Receive data from remote computer
Socket Messaging
Continued

More examples

More examples of the instruction `SocketReceiveFrom` are illustrated below.

Example 1

```
VAR socketdev udp_socket;
VAR string client_ip;
VAR num client_port;

PROC server_messaging()
  VAR string receive_string;
  ...
  ! Create and bind of sockets in error handlers
  SocketReceiveFrom udp_socket \Str := receive_string, client_ip,
       client_port;
  SocketSendTo udp_socket, client_ip, client_port \Str := "Hello
       client with ip-address"+client_ip;
  ...
  SocketClose udp_socket;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=SOCK_CLOSED THEN
    messaging_recover;
    RETRY;
  ELSE
  ! No error recovery handling
  ENDIF
ENDPROC

PROC messaging_recover()
  SocketClose udp_socket;
  SocketCreate udp_socket \UDP;
  SocketBind udp_socket, "192.168.0.1", 1025;
ERROR
  IF ERRNO=ERR_SOCK_CLOSED THEN
    RETURN;
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC
```

This is an example of a server program with creation and binding of sockets in error handlers. In this way the program can handle power fail restart.

In the communication procedure `server_messaging` the server receives a string message from the client and stores the message in `receive_string`. Then the server responds with the message `"Hello client with ip-address xxx.xxx.x.x"`.

Continues on next page

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable ERRNO will be set to:

Name	Cause of error
ERR_SOCK_CLOSED	The socket is closed.
ERR_SOCK_TIMEOUT	No data was received within the time out time.

Limitations

All sockets are closed after power fail restart. This problem can be handled by error recovery. See example above.

The maximum size of the data that can be received in one call is limited to 1024 bytes.

Syntax

```
SocketReceiveFrom
  [ Socket ':=' ] < variable (VAR) of socketdev >
  [ '\' Str ':=' < variable (VAR) of string > ]
  | [ '\' RawData ':=' < variable (VAR) of rawbytes > ]
  | [ '\' Data ':=' < array {*} (VAR) of byte > ]
  [ '\' NoRecBytes ':=' < variable (VAR) of num > ]
  [ RemoteAddress ':=' ] < variable (VAR) of string >
  [ RemotePort ':=' ] < variable (VAR) of num >
  [ '\' Time ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Send data to remote computer	*SocketSend - Send data to remote computer on page 723*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example client socket application	*SocketSend - Send data to remote computer on page 723*
Send data to remote computer	*SocketSendTo - Send data to remote computer on page 727*

Continues on next page

1.252 SocketReceiveFrom - Receive data from remote computer

Socket Messaging

Continued

For information about	See
Test for the presence of data on a socket.	*SocketPeek - Test for the presence of data on a socket on page 1394*

1.253 SocketSend - Send data to remote computer

Usage

`SocketSend` is used to send data to a remote computer. `SocketSend` can be used both for client and server applications.

Basic examples

The following example illustrates the instruction `SocketSend`:

See also *More examples on page 724*.

Example 1

```
SocketSend socket1 \Str := "Hello world";
```

Sends the message `"Hello world"` to the remote computer.

Arguments

```
SocketSend Socket [ \Str ] | [ \RawData ] | [ \Data] [ \NoOfBytes
    ]
```

`Socket`

Data type: `socketdev`

In client application the socket to send from must already be created and connected.

In server application the socket to send to must already be accepted.

`[ \Str ]`

Data type: `string`

The `string` to send to the remote computer.

`[ \RawData ]`

Data type: `rawbytes`

The `rawbytes` data to send to the remote computer.

`[ \Data ]`

Data type: `array of byte`

The data in the `byte` array to send to the remote computer.

Only one of the optional parameters `\Str`, `\RawData`, or `\Data` can be used at the same time.

`[ \NoOfBytes ]`

Data type: `num`

If this argument is specified only this number of bytes will be sent to the remote computer. The call to `SocketSend` will fail if `\NoOfBytes` is larger than the actual number of bytes in the data structure to send.

If this argument is not specified then the whole data structure (valid part of `rawbytes`) will be sent to the remote computer.

Continues on next page

1.253 SocketSend - Send data to remote computer
Socket Messaging
Continued

Program execution

The specified data is sent to the remote computer. If the connection is broken an error is generated.

The data that is transferred on the cable is always bytes, max. 1024 bytes in one message. No header is added by default to the message. The usage of any header is reserved for the actual application.

Parameter	Input data	Cable data	Output data
\Str	1 char	1 byte (8 bits)	1 char
\RawData	1 rawbytes	1 byte (8 bits)	1 rawbytes
\Data	1 byte	1 byte (8 bits)	1 byte

It's possible to mix the used data type (`string`, `rawbytes`, or `array of byte`) between `SocketSend` and `SocketReceive`.

More examples

More examples of the instruction `SocketSend` are illustrated below.

Example 1

```
VAR socketdev client_socket;
VAR string receive_string;

PROC client_messaging()
  ...
  ! Create and connect the socket in error handlers
  SocketSend client_socket \Str := "Hello server";
  SocketReceive client_socket \Str := receive_string;
  ...
  SocketClose client_socket;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    client_recover;
    RETRY;
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC

PROC client_recover()
  SocketClose client_socket;
  SocketCreate client_socket;
  SocketConnect client_socket, "192.168.0.2", 1025;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    RETURN;
```

Continues on next page

```
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC
```

This is an example of a client program with creation and connection of socket in error handlers. In this way the program can handle power fail restart.

In the procedure `client_recover` the client socket is created and connected to a remote computer server with IP-address `192.168.0.2` on port `1025`.

In the communication procedure `client_messaging` the client sends `"Hello server"` to the server and the server responds with `"Hello client"` to the client, which is stored in the variable `receive_string`.

Example 2

```
VAR socketdev client_socket;
VAR string receive_string;

PROC client_messaging()
  ...
  ! Send cr and lf to the server
  SocketSend client_socket \Str := "\0D\0A";
  ...
ENDPROC
```

This is an example of a client program that sends non printable characters (binary data) in a string. This can be useful if communicating with sensors or other clients that requires such characters.

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`ERR_SOCK_CLOSED`	`The socket is closed. Broken connection.`

Limitations

There is no built-in synchronization mechanism in Socket Messaging to avoid received messages that are compounded of several sent messages. It's up to the programmer to handle the synchronization with "Ack" messages (one sequence of `SocketSend - SocketReceive` in the client or server program must be completed before the next sequence of `SocketSend - SocketReceive`).

All sockets are closed after power fail restart. This problem can be handled by error recovery. See example above.

Avoid fast loops with `SocketCreate ... SocketClose` because the socket is not really closed until a certain time (TCP/IP functionality).

The size of the data to send is limited to 1024 bytes.

Continues on next page

1.253 SocketSend - Send data to remote computer

Socket Messaging

Continued

Syntax

```
SocketSend
  [ Socket ':=' ] < variable (VAR) of socketdev >
  [ \Str ':=' < expression (IN) of string > ]
  | [ \RawData ':=' < variable (VAR) of rawdata > ]
  | [ \Data ':=' < array {*} (IN) of byte > ]
  [ '\' NoOfBytes ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*
Use of non printable characters (binary data) in string literals.	*Technical reference manual - RAPID kernel*

1.254 SocketSendTo - Send data to remote computer

Usage

`SocketSendTo` is used to send data to a remote computer. `SocketSendTo` can be used both for client and server applications.

`SocketSendTo` is used for connectionless communication with datagram protocol UDP/IP.

Basic examples

The following example illustrates the instruction `SocketSendTo`:

See also *More examples on page 724*.

Example 1

```
VAR socketdev udp_socket;

SocketCreate udp_socket \UDP;
SocketSendTo udp_socket, Address, Port \Str := "Hello world";
```

Sends the message `"Hello world"` to the remote computer with IP address `Address` and port `Port`.

Arguments

```
SocketSendTo Socket RemoteAddress RemotePort [ \Str ] | [ \RawData
     ] | [ \Data] [ \NoOfBytes ]
```

`Socket`

Data type: `socketdev`

The socket must already been created.

`RemoteAddress`

Data type: `string`

The address of the remote computer. The remote computer must be specified as an IP address. It is not possible to use the name of the remote computer.

`RemotePort`

Data type: `num`

The port on the remote computer. Generally ports 1025-4999 are free to use. Ports below 1025 can already be taken.

`[ \Str ]`

Data type: `string`

The `string` to send to the remote computer.

`[ \RawData ]`

Data type: `rawbytes`

The `rawbytes` data to send to the remote computer.

`[ \Data ]`

Data type: `array of byte`

Continues on next page

1.254 SocketSendTo - Send data to remote computer
Socket Messaging
Continued

The data in the `byte` array to send to the remote computer.

Only one of the optional parameters `\Str`, `\RawData`, or `\Data` can be used at the same time.

`[ \NoOfBytes ]`

Data type: `num`

If this argument is specified only this number of bytes will be sent to the remote computer. The call to `SocketSendTo` will fail if `\NoOfBytes` is larger than the actual number of bytes in the data structure to send.

If this argument is not specified then the whole data structure (valid part of `rawbytes`) will be sent to the remote computer.

Program execution

The specified data is sent to the remote computer.

The data that is transferred on the cable is always bytes, max. 1024 bytes in one message. No header is added by default to the message. The usage of any header is reserved for the actual application.

Parameter	Input data	Cable data	Output data
`\Str`	1 char	1 byte (8 bits)	1 char
`\RawData`	1 rawbytes	1 byte (8 bits)	1 rawbytes
`\Data`	1 byte	1 byte (8 bits)	1 byte

It's possible to mix the used data type (`string`, `rawbytes`, or `array of byte`) between `SocketSendTo` and `SocketReceiveFrom`.

More examples

More examples of the instruction `SocketSendTo` are illustrated below.

Example 1

```
VAR socketdev client_socket;
VAR string receive_string;
VAR string RemoteAddress;
VAR num RemotePort;

PROC client_messaging()
  ...
  ! Create and bind the socket in error handlers
  SocketSendTo client_socket, "192.168.0.2", 1025 \Str := "Hello
       server";
  SocketReceiveFrom client_socket \Str := receive_string,
       RemoteAddress, RemotePort;
  ...
  SocketClose client_socket;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    client_recover;
```

Continues on next page

```
    RETRY;
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC

PROC client_recover()
  SocketClose client_socket;
  SocketCreate client_socket \UDP;
  SocketBind client_socket, "192.168.0.2", 1025;
ERROR
  IF ERRNO=ERR_SOCK_TIMEOUT THEN
    RETRY;
  ELSEIF ERRNO=ERR_SOCK_CLOSED THEN
    RETURN;
  ELSE
    ! No error recovery handling
  ENDIF
ENDPROC
```

This is an example of a client program with creation and bind of socket in error handlers. In this way the program can handle power fail restart.

In the procedure `client_recover` the client socket is created and bound to a remote computer server with IP-address `192.168.0.2` on port `1025`.

In the communication procedure `client_messaging` the client sends `"Hello server"` to the server and the server responds with `"Hello client"` to the client, which is stored in the variable `receive_string`.

Example 2

```
VAR socketdev udp_socket;

PROC message_send()
  ...
  ! Send cr and lf to the server
  SocketSendTo udp_socket, "192.168.0.2", 1025 \Str := "\0D\0A";
  ...
ENDPROC
```

This is an example the program sends non printable characters (binary data) in a string. This can be useful if communicating with sensors or other clients that requires such characters.

Error handling

The following recoverable errors are generated and can be handled in an error handler. The system variable `ERRNO` will be set to:

Name	Cause of error
`ERR_SOCK_CLOSED`	`The socket is closed.`

Continues on next page

1.254 SocketSendTo - Send data to remote computer
Socket Messaging
Continued

Limitations

All sockets are closed after power fail restart. This problem can be handled by error recovery. See example above.

The size of the data to send is limited to 1024 bytes.

Syntax

```
SocketSendTo
  [ Socket ':=' ] < variable (VAR) of socketdev >
  [ RemoteAddress ':=' ] < expression (IN) of string >
  [ RemotePort ':=' ] < expression (IN) of num >
  [ \Str ':=' < expression (IN) of string > ]
  | [ \RawData ':=' < variable (VAR) of rawdata > ]
  | [ \Data ':=' < array {*} (IN) of byte > ]
  [ '\' NoOfBytes ':=' < expression (IN) of num > ] ';'
```

Related information

For information about	See
Socket communication in general	*Application manual - Controller software IRC5*
Create a new socket	*SocketCreate - Create a new socket on page 709*
Connect to remote computer (only client)	*SocketConnect - Connect to a remote computer on page 706*
Receive data from remote computer	*SocketReceive - Receive data from remote computer on page 713*
Close the socket	*SocketClose - Close a socket on page 704*
Bind a socket (only server)	*SocketBind - Bind a socket to my IP-address and port on page 702*
Listening connections (only server)	*SocketListen - Listen for incoming connections on page 711*
Accept connections (only server)	*SocketAccept - Accept an incoming connection on page 699*
Get current socket state	*SocketGetStatus - Get current socket state on page 1391*
Example server socket application	*SocketReceive - Receive data from remote computer on page 713*
Receive data from remote computer	*SocketReceiveFrom - Receive data from remote computer on page 718*
Use of non printable characters (binary data) in string literals.	*Technical reference manual - RAPID kernel*

1.255 SoftAct - Activating the soft servo

Usage

`SoftAct` (*Soft Servo Activate*) is used to activate the so called “soft” servo on any axis of the robot or external mechanical unit.

This instruction can only be used in the main task `T_ROB1` or, if in a *MultiMove* system, in any motion tasks.

Basic examples

The following example illustrates the instruction `SoftAct`:

Example 1

```
SoftAct 3, 20;
```

Activation of soft servo on robot axis `3` with softness value `20%`.

Example 2

```
SoftAct 1, 90 \Ramp:=150;
```

Activation of the soft servo on robot axis `1` with softness value `90%` and ramp factor `150%`.

Example 3

```
SoftAct \MechUnit:=orbit1, 1, 40 \Ramp:=120;
```

Activation of soft servo on axis `1` for the mechanical unit `orbit1` with softness value `40%` and ramp factor `120%`.

Arguments

```
SoftAct [\MechUnit] Axis Softness [\Ramp]
```

`[ \MechUnit ]`

Mechanical Unit

Data type: `mecunit`

The name of the mechanical unit. If this argument is omitted then it means activation of the soft servo for specified robot axis in the current program task.

`Axis`

Data type: `num`

Number of the robot or external axis to work with soft servo.

`Softness`

Data type: `num`

Softness value in percent (0 - 100%). 0% denotes min. softness (max. stiffness), and 100% denotes max. softness.

`[ \Ramp ]`

Data type: `num`

Ramp factor in percent (>= 100%). The ramp factor is used to control the engagement of the soft servo. A factor 100% denotes the normal value; with greater

Continues on next page