AN140

16-127173

Copley Drive Web Interface

Introduction

Starting with firmware version 4.34, most Copley EtherCAT drives can be configured to use standard Ethernet protocols such as UDP/IP and TCP/IP. When configured in this way, the drives include a simple internal web server which can serve customer generated web pages. This feature allows the drive to be controlled through a web browser using programs written in JavaScript.

Supported drives

The web server feature is available on all of Copley's EtherCAT drives in the *plus* family. This includes all Copley EtherCAT drives except for the AEP drive and the ARM based EtherCAT drives (IES and NES).

Required firmware

The web interface as described here was first added to the plus family firmware starting with version 4.34. That firmware version or later will be required to use the features described here.

Due to limitations in the amount of RAM available in most Copley drives, the EtherCAT network and web interface can not both be supported by the same firmware image. To enable the web interface described in this document the drive firmware will have to be updated to a new version which removes EtherCAT networking features and adds the TCP/IP and web server features. This is true for all drives designed before the AEV. The AEV and future EtherCAT drives have a larger internal RAM which allows both network protocols to be supported in the standard firmware.

Please contact Copley customer support to obtain the web enabled firmware for your drive.

Enabling the web server

Drive parameter 0x121 (network configuration) is used to enable the TCP/IP stack and web server in the drive. Enabling these features will also disable the EtherCAT networking layer in the drive, EtherCAT can not be used at the same time the web server is being used.

Setting bit 9 of parameter 0x121 will enable the web interface and disable the EtherCAT interface.

For drives which require special firmware to add support for the web interface, bit 9 of parameter

0x121 will be permanently set and can not be cleared without changing the drive firmware. For drives such as the AEV which support both EtherCAT and the web interface in the same firmware, bit 9 of parameter 0x121 is used to toggle between these two modes.

To communicate with the drive over TCP/IP, the drive will need to be assigned an IP address. This can either be done manually by setting parameter 0x11F, or can be done using a DHCP server on the local network.

IP addresses are normally written out as a series of four decimal values separated by periods such as:

192.168.1.1

When passed to parameter 0x11F, the four decimal values should be packed into a single 32-bit value in little endian format. That is, the right most digit in the IP address is the most significant byte in the 32-bit value. The IP address 192.168.1.1 would be formatted as 0x0101A8C0.

To automatically assign an IP address to the drive, set bit 0 of parameter 0x121. When bit 0 is set, the drive will request an IP address from a DHCP server on the network. The resulting IP address can be read from the drive at parameter 0x11F.

Initial web landing page

Once the web interface has been enabled and a valid IP address has been assigned, it should be possible to access a simple default web page on the drive using a web browser on the same local network. This is done by entering the drive's IP address in the browsers address bar formatted as a URL.

For example, if the IP address assigned to the drive were 192.168.1.1, then the following URL should load the default web page of the drive:

http://192.168.1.1

This default web page is primarily used to confirm that the web server has been successfully enabled and to upload custom web data to the drive.

At the bottom of the default web page there is a button which will allow a file containing custom web content to be uploaded to the drive. The data should be contained in a zip file and will be saved in the drive as a file named web.zip.

Once a custom set of web pages has been successfully loaded onto the drive, the default landing page will no longer come up when the drive is targeted by a web browser. To get back to the default page after loading web content, use the following URL (substituting the correct IP address of the drive):

http://192.168.1.1/_default

Note that there's an underscore character before the word default.

This default page can then be used to replace the web.zip file loaded onto the drive with a new file.

web.zip

As mentioned above, the web pages stored on the drive should all be packaged in a single zip archive file named web.zip. Any file that's uploaded to the drive using the default web interface will automatically be checked to confirm that it's a zip file and will be renamed web.zip after uploading. Any existing web.zip already on the drive will be replaced.

The web.zip file can contain files in any content; html, JavaScript, css, etc. One file that should always be included in the web.zip archive is a file named index.html. This is the top level web page that will be served up if no other page is specified.

A zip archive is a standard file format that can be used to hold one or more other files using various compression methods. The Copley drive firmware which reads these zip files has some limitations which must be taken into account when creating the file.

The zip file format supports several different compression methods for the files it contains. Copley drives are currently only capable of extracting files which use either the *deflate* algorithm, or those that are stored in the zip file uncompressed. The deflate algorithm is the standard algorithm used to compress files in a zip archive, so this limitation is generally not a problem.

A more significant limitation is the amount of RAM available on the drive to extract files from a zip archive. Drives which require special firmware to support the web interface (i.e. those drives developed before the AEV model) only have a 8kbyte available for extracting compressed files. Files that are stored in the zip archive using the deflate algorithm must be decompressed into this buffer before they can be used, so this puts an upper limit on the size of the decompressed files to 8k. The AEV and later drives use a 32k buffer, so they can extract files up to that size.

There's no restriction on the size of files stored uncompressed in a zip archive, so larger files can be stored in a zip file without compression if necessary.

There is an efficient way to store larger files in the web.zip file although it's a bit non-intuitive. A file compression method known as gzip is a standard way of sending compressed data from web servers to browsers. Files can be converted to gzip format first, then stored in the web.zip archive file without further compression. The drive firmware can extract these files from the web.zip file and send them to the browser as they are (still in gzip format) and the browser will handle the decompression. This is actually the most efficient method to handle large files as they don't need to be decompressed by the drive and are transmitted in a compressed format over the network.

Files are converted to the gzip file format using a utility program of the same name (gzip). This utility is a standard feature of most Linux systems and a Windows version can be found with a quick Google search. When a file is compressed to gzip format, the extension .gz is added to the file name. For example, if a file named index.html were compressed using gzip, the resulting compressed file would be named index.html.gz.

When the drive searches for files in the web.zip archive it will look first for the file name exactly as it's requested by the web browser (such as index.html), and if that's not found it will look for the gzipped version of the file (ex. index.html.gz). If the gzipped version of the file is found it will be extracted and sent to the requesting web browser with the necessary flags to indicate that it's gzip compressed content. This will allow the browser to decompress it to get the original uncompressed file.

Dynamic content

Serving up static web pages isn't in itself a very useful feature for the drive. The thing that makes the web interface really useful is the drive's ability to handle dynamic requests to read/write it's parameters and perform other actions. This allows a complex user interface to be designed using HTML and JavaScript which will run in the web browser and interact with the drive using these dynamic requests.

Dynamic content in the drive is accessed using JSON (JavaScript Object Notation). This is a simple and efficient method of sending requests and receiving responses from the drive using the web interface.

For example, requesting the current motor position using the jquery library in JavaScript would be done with a line like this:

```
$.getJSON( "_cmd", {type: "get", ids: [0x07]}, callback );
```

The callback function specified in that line (named *callback* here) will be called with the drive's response neatly formatted as a valid JavaScript object.

The details of using jquery to send JSON requests is well beyond the scope of this document, but there are many useful web resources available to get you started.

Getting drive parameters:

To request the values of one or more drive parameters, send a JSON formatted request to the drive's _cmd web page. The JSON object sent must have a property called `type` set to the string "get". It also needs a property called `ids`. The value of the `ids` property should be an array of up to 10 parameter numbers which are requested.

The drive will respond to this request with a JSON object with a property named `values`. The value of this property will be an array of the parameter values requested in the same order they were passed in. In the event that there was an error processing one of the parameter requests, the value for that parameter will be replaced with an error code in the form `err_`*`n`* where *`n`* is an integer error number

Parameter values are formatted exactly like they would be when sent to the drive using the binary serial interface. There's a separate applications note on that interface, but in brief the parameter ID is a 16-bit number which is bit-mapped as follows:

Bits	Meaning
0-8	Hold the parameter ID number
9-11	Reserved for future use (must be zero)
12	0 to access a parameter in working memory (RAM). 1 to access the parameters value saved to flash.

Bits	Meaning
13-14	Give the axis number (0-3) on multi-axis drives
15	reserved

The values returned by the drive will have various formats depending on the type of parameter being accessed. Most parameter are simple integer values (either 16 or 32-bit). The drive will return these as signed decimal integer values. String parameters are returned as ASCII strings by the drive.

Other parameters such as filter settings and output pin configuration will be returned as an array of values. The exact details of what these parameter values mean is documented in the parameter dictionary document.

Setting drive parameters

To set the value of one or more drive parameters, send a JSON formatted request to the drive's _cmd web page. The JSON object sent must have a property called `type` set to the string "set". It also needs a property called `params`. The value of the `params` property should be an array of up to 10 JSON objects each holding a parameter ID (`id` property) and value (`value` property).

For example, one could set the destination position for a move (parameter 0xCA) to a value of 12345 encoder counts, and also set the max velocity for the move (parameter 0xCB) to a value of 3456.7 cts/sec using the following JSON request:

```
$.getJSON( "_cmd", {type: "set", params:[ {0xca,12345}, {0xcb,34567}], callback );
```

The drive will set each listed parameter to the passed value. The JSON object that the drive will return will be passed to the callback function. This object will have a property named `status` which will hold an array of *N* values where *N* is the number of parameters set. Those values will either be "ok" on success, or an error number "err_n" on failure.

Generic binary serial commands

It's possible to execute an arbitrary binary serial port command through the web interface. Binary serial commands are documented in Copley application note AN112. That document should be referenced for a more complete explanation of that interface. In short, every binary serial command consists of an 8-bit opcode and zero or more 16-bit data words.

To send a binary serial command to the drive, a JSON formatted object should be sent to the _cmd web page on the drive. This object must have a `type` property set to "binary". The object also needs a property called `cmd` set to the opcode value, and optionally a `data` property set to a comma separated list of 16-bit data words.

Here’s an example of a binary command to the drive which will start a move. The command value is 0x11 which is the opcode for the trajectory command (used to start/stop/update moves), and the data consists of a single value (1). Trajectory sub-command 1 is used to start a new move.

```
$.getJSON( "_cmd", {type: "binary", cmd: 0x11, data:1 } );
```

The drive will respond to this command with a JSON formatted object with an error property giving the error code resulting from the command. An error code of 0 is success.

Working with multiple drives

A single web application can send JSON formatted requests to multiple drives on the network by specifying their IP address in the URL. Rather then just accessing the “_cmd” web page, one would access the “_cmd” page of a specific drive. i.e. $.getJSON(“http://192.168.1.2/_cmd”, …)

Sending JSON requests to different URLs will often cause problems due to browser security measures which are designed to prevent one web page from accessing content on a different server. This is called the browsers cross-domain policy. Copley drives get around this problem by explicitly allowing such cross-domain access in the header fields that they send with the HTTP requests.

This also means that it’s not necessary to host the web files on the drive itself. The web data could be located on the local PC that is running the web browser, or on a remote server on the network. This is helpful in the case of very large web applications since the maximum size of the web.zip file that a drive can hold is limited to about 800kbytes due to the size of the drive’s internal file system.

Revision History

Date	Version	Revision
5/5/2020	Rev 00	Initial release