e4thcom-0.6.1 -- A Terminal for Embedded Forth Systems

Abstract

e4thcom for the Linux OS (32 Bit executables for X86 and Raspberry/Raspbian) is a terminal program for embedded Forth Systems. The current release 0.6.1 supports

- *editable command line buffer with history and tab selection* ***(new since 0.5.4 :** see \index **)***
- *editor interface for debugging* ***(new since 0.5.4 :** see #edit and #ls **)***
- *data transmission via serial line (UDP network connection* only *with 0.5.3)*
- *conditional and unconditional uploading of source code*
    - *redefined in 0.5.4,* **full- and half-duplex since 0.5.5 :** see --half-duplex
    - *path for file access changed in 0.5.4 : see Uploading of Source Code Files*
    - *commands for break and exit changed in 0.5.4 : '\' = break, '\ [Enter]' = exit*
- *plug-ins (Forth Source Code files) for target specific configuration*
    - ***anyForth**, **430CamelForth**, **430eForth**, **4e4th**, **AmForth**, **Mecrisp**, **Mecrisp-Stellaris*** and ***noForth***
- *plug-ins (Forth Source Code files)* for *cross-assembling and cross-disassembling*
    - *MSP430 and AmForth targets*
- *loading target specific resource identifiers (register and bit identifiers) from Forth files*

Forth on Embedded Systems

Forths interactivity is an important advantage on embedded systems. Any simple terminal program can be used to get in touch with the system and explore its structure and capabilities.

But - as soon as the first program is written and ready for testing - an easy way to upload the source code files is missing. A tool to overcome this deficiency is the e4thcom terminal program.

The e4thcom Terminal

e4thcom is a terminal program for the Linux OS. It sends terminal input without a local echo via serial line (or UDP 0.5.3 only) to the target (Forth System) and displays characters received from the target in the terminal window. This is the terminal-mode of the program.

Entering the character '#' or '\' as the first one of a new line activates the control-mode of the program. Terminal input is then not forwarded to the target system but catched until the enter key is pressed and is then interpreted as a terminal directive.

Uploading of Source Code Files

Two terminal directives are defined for uploading of source code files. They are modelled after the corresponding words *include* and *require* from the Forth Standard and named **#include** and **#require**. Both expect to get the name of a source code file as the only parameter:

```
#include <filename>  \ unconditional uploading

#require <filename>  \ conditional uploading
```

Both directives can also be used in source code files, one directive per line, the rest of the line is silently discarded.

When directives are entered at the terminal, the abbreviations **#i** and **#r** can be used.

The file name lookup is done at the path **`cwd:cwd/mcu:cwd/target:cwd/lib`** . CWD is the directory, that was the active one (current working directory) at program start.

The underlaying concept is, to store project-specific source code in **cwd**, target-forth-specific in **cwd/target**, hardware-specific in **cwd/mcu** and target-independent code in **cwd/lib**.

Unconditional Uploading

Source code files are read line by line and the text is interpreted as a terminal directive or uploaded to the target.

After sending a line of source code to the target, the terminal waits for the target to process the line and return with a message. If no error occures, the next line is read from the source. Otherwise uploading is aborted with an error message.

Comment lines starting with a '\' char are not uploaded but only displayed in the terminal window. The same is true for multi-line comments inside curly braces.

When the end of a source code file is reached, the file is closed and control is given back to the calling level. Finally, when the last file is closed, the terminal mode is activated again.

Conditional Uploading

Before uploading a file, the terminal checks, whether a word, that equals the file name, already exists in the current search order of the targets dictionary. The upload directive is ignored, if that is the case. Otherwise the file is uploaded.

After uploading a file, the terminal again checks if now a word exists in the target dictionary, that equals the file name. If that is not the case, a NOOP-Word with that name is created in the target dictionary.

With this approach, the number of dictionary entries for file names can be minimized, by using - whenever adequate - the name of a prominent word, defined in a file, as the name for that file.

Cancelling Uploading of Source Code

Uploading can be cancelled by pressing the TAB key at the keyboard. (In versions <0.5.4 ^B was used.)

Please be aware, that the target may be in compile mode when canceling a running upload. Typing **`[`** `[Enter]` *will then bring the target back to interpret mode.*

Data Transmission Modes

Since version 0.5.5 two data transmission modes are supported, full- and half-duplex. Default is **full-duplex**. To use half-duplex start e4thcom with the **--half-duplex** option.

Up to version 0.5.3 only half-duplex was implemented (uploading a byte and waiting for the echo). That worked fine in general but was unacceptable slow with the MSP430 FR5969 Launchpad. This unit has a more sophisticated USB interface and only works with acceptable transmission speed if block-transfer is used to upload source code.

There may be targets or transmission channels, that are not capable to handle full-duplex data transmission. Then try the --half-duplex option.

Error Editing

In e4thcom-0.5.4 the terminal directive **`#edit`** `<file-name> [Enter]` was added to open a source code file in an external editor of your choise.

If uploading is aborted with an error message, typing **`#edit`** `[Enter]` or **`#e`** `[Enter]` at the command line will open the uploaded file with the cursor placed in the line that raised the error.

The editor command is defined in the config file `.e4thcom-x.y.z` . Change it to your needs. Search order for the file is `project-dir:e4thcom-dir` .

To list the files available in the current working directory (project directory) the directive **`#ls`** `[OPTION]... [FILE]...` can be used to execute the OS command ls (see the ls man page).

Buffered Command Line

Command line input is no longer directly send to the target but buffered until the [Enter] key is pressed. This means it is no longer possible to send a single char to the target without a trailing CR. If you need to send a single char without a CR, enter it with a leading '\' char in an empty command line,

e.g. `\x [Enter]` sends the char x to the target without a trailing CR.

Command Line History

e4thcom uses an internal history buffer of limited size. Command line input is added to that buffer, throwing out oldest entries, if required. A limited TAB completion (TAB selection) ist supported:
• Pressing the TAB key displays those words from the history, matching the first character of the command line.
• Pressing the Esc key removes all chars from the command line buffer.

The command line history can be extended from within source code files, using the terminal directive \index:

```
\index  phrase1  phrase2  ...  phraseN
```

The directive must be the first word in a line. Phrases must be preceeded with two spaces as separators. A phrase can be a word or a sequence of words, separated by a space.

Installing e4thcom

e4thcom is distributed as a tar.gz archive. For testing or as a user without admin rights unpack the archive in a directory of your choise, e.g. in your home directory or on the desktop. You will then find the e4thcom binary and related files in a new directory named e4thcom-x.y.z. The e4thcom binary in this directory is the one for X86 Linux Systems.

Please be aware, that absolute path- and filenames are restricted to 64 chars (bytes) and relative path- and filenames (relative to the e4thcom directory or to a project directory) are restricted to 32 chars (bytes).

When installing e4thcom on a Raspberry with Raspbian OS please copy the binary from the Raspbian subdirectory to the e4thcom directory.

On the Raspberry B please use the half-duplex mode (see option --half-duplex). Full-duplex does not work properly. May be it's to fast for the USB channels.

As an admin you can make e4thcom available to all system users by unpacking the archive to the /opt directory, changing owner and group with `chown -R root:root /opt/e4thcom-x.y.z` and creating a link named e4thcom in /usr/local/bin to the corresponding file in the /opt/e4thcom-x.y.z directory.

Starting e4thcom

To start the e4thcom terminal program, open a terminal window in a (project) folder of your choice and enter the command

```
/path/to/e4thcom-x.y.z/e4thcom [options]
```

The following options are supported:

-t [target]	The target system to connect to. Loads the plug-in file **target.efc** from the e4thcom directory at program start. See the chapter Plug-Ins which targets are supported.
-d [device]	The serial device, e.g. ttyS0, ttyUSB0, ttyACM0 ... for a serial line connection

~~or ip-address:port for an UDP connection~~ (only 0.5.3)

-b [baudrate]	The baudrate for a serial line connection. Supported values are: B1200, B2400, B4800, B9600, B19200, B38400, B57600, B115200
-h	Displays the terminals help text.
--half-duplex	The data transmission mode. Default is full-duplex.

The connection to a target system via a serial line is established during the program start. This may fail if the chosen device is not available or used by another process (e.g. by a modem manager). The program is then terminated with an error message. After fixing the cause of failure or waiting for the modem manager to give up, a successful start of the program should be possible.

Target Connections via UDP (only e4thcom-0.5.3)

From the Forth-Tagung in April 2015 I came back with a TI Tiva Connected Launchpad with TM4C1294NCPDT, Mecrisp-Stellaris and the Ethernet driver contributed by Bernd Paysan. This made me extend the e4thcom terminal to support communication vai UDP.

The terminals UDP mode is enabled by assigning an IP-Address and a port number to the -d option (see section Starting e4thcom), e.g. :

```
    -d ip-address:port   e.g.: -d 192.168.2.107:4201
```

By starting two e4thcom instances, it's possible now to connect to a TMC4.... via serial line and UDP at the same time.

Closing e4thcom

To close the terminal enter `\ [Enter]` at the command line (in versions <0.5.4 the esc key was used.).

Plug-Ins

Target systems are different concerning the messages (the prompts) returned after processing a line of uploaded code.

Up to version 0.3.4 e4thcom was split in an executable file and a number of target specific image files, named after the targets. Starting with version 0.4 there is only one image file for all targets. The target specific code was factored out and is now loaded as a plug-in at program start.

The target specific plug-ins are small Forth Source Code Files so that support for other targets can be easily added by experienced Forth Users.

The current e4thcom release comes with the following plug-ins:

`anyforth.efc`	\ dump uploader without any error checking, only unconditional uploading with #i[nclude] <file name> is supported
`430camelforth.efc`	\ generic CamelForth plug-in
`430camelforth-xas.efc`	\ generic + MSP430 cross-assembler and -disassembler ¹)
`430eforth.efc`	\ generic 430eForth plug-in
`430eforth-xas.efc`	\ generic + MSP430 cross-assembler and -disassembler ¹)
`4e4th.efc`	\ generic 4e4th plug-in
`4e4th-xas.efc`	\ generic + MSP430 cross-assembler and -disassembler ¹)
`amforth.efc`	\ generic amForth plug-in
`amforth-xas.efc`	\ generic + ATmega cross-assembler ²)
`mecrisp.efc`	\ generic Mecrisp plug-in
`mecrisp-msp430xas.efc`	\ generic + MSP430 cross-assembler and -disassembler ¹)
`mecrisp-st.efc`	\ generic Mecrisp-Stellaris plug-in

```
noforth.efc                \ generic noForth plug-in
noforth-xas.efc            \ generic + MSP430 cross-assembler and -disassembler ¹)
```

¹) The cross-assembler and -disassembler are based on code from noForth.
²) The cross-assembler is based on the amforth ATmega assembler from Lubos Pekny. (Needs further testing.)

The reliability of the plugins depends on the targets prompts (and of course on my ideas, to evaluate them). Not all are perfect yet.

The noforth plug-in (testet with noForth C/V 141218 ... 160401)

works perfect whith ACK/NAK control chars enabled on the target (**OK HEX 8000 TO OK FREEZE**).

- ^ACK (06) received: noForth is ready to receive a new line. Terminal sends the next line.
- ^NAK (15) received: noForth is ready to receive a new line but there was an error. Terminal stops uploading.
  No ^ACK/^NAK received: noForth is busy or waiting for user input. Terminal waits for ^ACK/^NAK. Terminal impút is send to the target while waiting.

The mecrisp and mecrisp-stellaris plug-ins (testet with Mecrisp 1.x ... 2.x and Mecrisp-Stellaris 2.1.1 for TM4C1294)

use a combination of timeout detection and text analysis:

- ' ok.^NL' received : It's assumed that mecrisp is ready to receive a new line. Terminal sends the next line.
- Timeout after 'message^NL' : It's assumed that mecrisp is ready to receive a new line, but an error occured. The terminal stops uploading.
- Timeout without 'message^NL' : It's assumed that mecrisp is busy or waiting for user input. The terminal waits for ^NL, terminal imput is send to the target while waiting.

The plugins are not perfec but work fine. It's not very likely to meet remaining ambiguous condition.

The amforth plugin (testet with amforth 4.9 , 5.1, 5.5, 6.0)

uses a combination of timeout detection and text analysis:

- ' ok' received : It's assumed that amforth is ready to receive a new line. The terminal sends the next line.
- ' ^CR^NL' received : It's assumed that amforth is ready to receive a new line but an error occured. The terminal stops uploading.
- Timeout : It's assumed that amforth is busy or waiting for user input. The terminal waits for ' ok' or ' ^CR^NL'. Terminal imput is send to the target while waiting.

Again, this plugin is not perfect but works fine, as long as the strings, used for error and ok detection, are not send by your code, while the target is still busy.

The 4e4th plug-in (testet with 4e4th 0.34 and debug version 150925)

is based on text analysis only:

- '?^XON^CR^NL' or '?^CR^NL^XON^ CR^NL' received : It's assumed that 4e4th is ready to receive a new line but an error occured. The terminal stops uploading.
- '^XON^CR^NL' without a leading '?' received : It's assumed that 4e4th is ready to receive a new line. The terminal sends the next line.

This plugin has the following restriction:

◦ Error messages created with abort", that need to be detected while uploading, must be terminated with a question mark.

The 430CamelForth plug-in (tested with 430CamelForth v0.5)

is based on timeout detection and text analysis. It's not perfect but close to. An ambiguous condition exists, if the target waits for user input during uploading. Then uploading is aborted.

The 430eForth plug-in (tested with 430eForth v4.3)

is based on timeout detection and text analysis. It's not perfect but close to. An ambiguous condition exists, if the target waits for user input during uploading. Then uploading is aborted. Error messages should end with a '?' char.

Finally it's safe to say that the most reliable plugin is that for noForth. And it is the most simple one. In other words, it would be very nice to have an ACK/NAK option in the other Forth systems too.

Cross Assembling

Starting with version 0.4 e4thcom comes with a cross assembler interface. A target specific cross assembler, written in Forth, can be included by the target specific plugin at program start. Then the terminal interprets the words **code** and **end-code**, **label**, **x[** and **\xas** as terminal directives when uploading source code from files.

code and **end-code**

A line of source code that starts with the string **code** is uploaded to the target and afterwards the terminals cross-assembler mode is enabled. The next lines are then assembled by the terminals cross-assembler and the resulting code is send to and comma-compiled at the target, until the string **end-code** is found in the source. Then the terminals cross-assembler mode is disabled again and the string **end-code** is send to the target. The words **code** and **end-code** must be defined in the targets dictionary but do not need to have any e4thcom related properties.

```
code led1.on ( -- )
   BIT0 # P1OUT & .b bis next end-code
```

label LB[01,05]

Assign the targets next free code address to one of the global labels LB01...LB05, which are predefined in the cross assembler dictionary, e.g.:

```
code min ( n1 n2 -- n1|n2 )
   sp )+ w mov tos w cmp  label LB01  >? if, w tos mov then, next
end-code

code max ( n1 n2 -- n1|n2 )
   sp )+ w mov w tos cmp  LB01 jmp  end-code
```

\xas

A line of source code, that starts with **\xas** is not uploaded to the target but interpreted by the cross assembler (the terminals internal Forth interpreter). This can be used to extend the cross assembler, e.g. to add short macro definitions or to load a file with target specific resource identifiers, e.g.:

```
\xas  MCU: name        Loads the file name.efr.
```

The search order to find name.efr is **project-dir:e4thcom-dir**.

x[..........]

A new feature of the cross assembler interface was added to version 0.5. The cross assembler can be temporarily disabled to send text to the target and read a cell-sized value back. This is required to write code that needs to access data or words defined at the target, e.g.:

```
#require code

\xas MCU: MSP430G2553

\xas  : dup, ( -- ) tos sp -) mov ;     \ some macros
\xas  : push, ( x -- ) dup, tos mov ;   \

code base@ ( -- u )
   x[ base ] & push, next end-code
```

The **number base in the cross assembler mode** is hex. The cross assembler is case sensitive. I prefere lower case for operators and upper case for register and bit identifiers.

Cross Disassembling

A target specific cross disassembler can be included by the target specific plug-in. Then the terminal directive **#das** (abbr. **#d**) is supported at the terminals command line.

#das name
\ Starts disassembling name (at its xt).

#das
\ Reads an address from the targets TOS and starts disassembling at this address.

Using Resource Files

Modern MCUs have lots of peripheral modules. A host of parameters and descriptors is required to access, configure and use this modules. e4thcom can load resource data from Forth files, to be used in cross assembler definitions or to be uploaded to the target:

\res

A line of source code, that starts with `\res` is not uploaded to the target but interpreted by the terminals internal Forth interpreter. This can be used to define resource descriptors in the terminals dictionary or to load some from target specific resource files.

`\res  <single-number> equ name`
\ Creates the resource descriptor name (a constant) in the terminals dictionary.

`\res  export  name1 name2 ... nameN`
\ Exports resource descriptors (as constants) from the terminals dictionary to the target.

`\res  MCU: name`
\ Loads the (resource) file name.efr , e.g. `\res MCU: MSP430G2553` .

The search order to find name.efr is **project-dir:e4thcom-dir**.

With a little help ...

Up to version 0.5.3 e4thcom did not have a command line history and did not support extended command line editing. It was intentionally designed minmalistic because I used it with a GTK+ GUI called ForthBox which added the missing features.

Recently I noticed, that the ForthBox can not be used on 64 Bit Linux Systems because it depends on the 32 Bit libvte, which is not available in the repositories of the current 64 Bit Linux distributions. So I decided not to bundle the ForthBox any longer with the e4thcom terminal. You can still use the one from e4thcom-0.5.3 (on 32 Bit systems) and I'll probably make the latest ForthBox version available as a separate release, if anyone is interested in.

What's left to do :

• Adding device locking to prevent concurrent device access from different processes. If another process uses the same device as the e4thcom terminal, it may snap away received chars so that the communication with the target will not work properly.

The problem here seems to be, that not all programs, that use serial devices, use device locking or don't do it the same way.

• Adding more targets and cross assemblers.

You can do it yourself and/or contribute. *The new plug-in structure should make it easy. The plug-in files and the cross assembler file that come with this release should be useful examples.*

Annotation:

The e4thcom distribution comes with ABSOLUTELY NO WARRANTY. It is free software under the terms of the GNU General Public License as published by the Free Software Foundation; either version 3 of the License, or any later version, see http://www.gnu.org/licenses .

Questions, error notes and suggestions for improvements or additional targets are welcome and can be forwarded to `firstname dot lastname  @ forth-ev.de` .

e4thcom was implemented with MINFORTH Plus, a MINFORTH derivative. *Thanks to Andreas Knochenhauer for creating MINFORTH, which was a great starting point for my MINFORTH Plus development environment.*

Thanks to Albert Nijhof and Willem Ouwerkerk for supporting me to make e4thcom ready for noForth and for agreeing that the noForth cross assembler could be adapted to and can be distributed with e4thcom.

Last Revision: MM-170208