Xchars
or
Unicode in Forth

First Experiences

M. Anton Ertl[*]
TU Wien

Bernd Paysan

Abstract

When dealing with different scripts at the same time (e.g., Latin, Greek, Cyrillic), or with Chinese ideograms, 8-bit fixed-width characters are too narrow. However, many Forth programs have an environmental dependency on `1 chars` = 1, so just making Forth characters wider would cause quite a lot of portability problems. We propose to add xchars for dealing with potentially wider, variable-width characters. This extension is relatively painless, requiring changes in only those program parts that work with individual characters, if they should work with the extended characters; uses of string words need no changes to work with extended characters. The xchar words can also be implemented on 8-bit-only Forth systems, so programs written to use xchars can also work on such systems.

1 Introduction

Most Forth systems today support character sets fitting into 8 bits, such as ASCII (7 bits) and its 8-bit extensions like ISO Latin-1.

However, such 8-bit character sets are not sufficient to support Chinese, Japanese, and Korean Han ideographs, or to express a text that contains, say, German, Russian, *and* Greek words. To address this problem, Unicode[1] was developed. Unicode is a universal character set.

There are several alternative encodings of Unicode characters: In UTF-32 each character consists of 32 bits, in UTF-16 each character consists of 1–2 16-bit entities, in UTF-8 each character consists of 1–4 8-bit entities. I.e., UTF-8 and UTF-16 are variable-width encodings.

How can Forth accomodate Unicode? ANS Forth only allows ASCII or only graphic ASCII characters in many contexts. However, ANS Forth also supports fixed-width, but large characters; e.g., a Forth system could use 32-bit characters to support the UTF-32 encoding of Unicode; since the codes for the ASCII characters are the same in Unicode, this would actually be a fully compliant ANS Forth implementation. Indeed, Jax4th was one of the first ANS Forth implementations and implemented characters as fixed-width 16-bit characters (for the then-current 16-bit (subset of) Unicode).

However, most Forth programs, even if they are otherwise mostly ANS Forth compliant, assume that `1 chars` produces 1.[2] These Forth programs would not work correctly on a system where `1 chars` produces 4, as would be the case with UTF-32 characters on a byte-addressed machine. And it is relatively hard to find all the places where one forgot to insert `chars` or `1 chars /` or where one used `1+` instead of `char+`. So going to UTF-32 characters would be a rather painful option.

Fortunately, Forth programs usually do not work with individual characters (with, e.g., words like `emit`) in many places. They work much more often with strings of characters (with, e.g., words like `type`). So if we find a way to deal with Unicode where string-handling code would continue to work, and only character-handling code needed changing, that solution would require much less porting effort for most programs than using UTF-32 with the existing character words and an appropriate `chars` size.

In this paper, we propose such a solution based on a new character type (xchars) and words for dealing with that type. In the following paper, we explain and discuss the new data types and words (Section 2), look at scenarios for using various en-

[*]Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; `anton@mips.complang.tuwien.ac.at`

[1]Actually, there were two standards: ISO 10646 and Unicode, produced by two different organizations, resulting in two standards documents; fortunately, the two documents define the same character set. We use the name Unicode throughout this paper.

[2]Since all widely used ANS Forth systems have the property that `1 CHARS` produces 1, it is pretty much impossible to test that a program does not have this environmental dependency.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2005 Conference Proceedings, p. 16--20

codings and character sizes in Forth systems (Section 3), give some examples of using these words (Section 4), report our experience with implementing these ideas in Gforth (Section 5), and compare our work to that of others.

2 Glossary

The following set of words is not final. It contains a number of redundant words, and we might decide to recommend a smaller set for widespread adoption. Conversely, there might also be words that are useful and that we missed or are still undecided (see Section 2.4).

If you are missing string words (like `type`), that's because you can use the ANS Forth string words on strings containing xchars.

2.1 Data types

xc An xchar (extended character) on the stack. For Unicode characters this will typically be the (decoded) Unicode number of the character.

xc-addr The address of an xchar in memory. Xchar addresses are character-aligned. An xchar can be represented (encoded) in memory differently than on the stack.

xc-addr u A string containing xchars. *u* counts the chars, not the xchars (or tha aus) in the string. All ANS Forth string words can be used on such a string.

2.2 Words

xchar+ (xc-addr1 – xc-addr2) Corresponds to `char+`. xc-addr2 is the address of the xc after xc-addr1.

xchar- (xc-addr1 – xc-addr2) Corresponds to `char-`. xc-addr2 is the address of the xc before xc-addr1.

+x/string (xc-addr1 u1 – xc-addr2 u2) Corresponds to `1 /string`.

-x/string (xc-addr1 u1 – xc-addr2 u2) Corresponds to `-1 /string`.

xc@ (xc-addr – xc) Corresponds to `c@`. Fetch the xchar at xc-addr onto the stack.

xc@+ (xc-addr1 – xc-addr2 xc) Fetch xc from xc-addr1; xc-addr2 is the address of the next xchar.

xc@+/string (xc-addr1 u1 – xc-addr2 u2 xc) Fetch xc from xc-addr1 and also perform the action of `+x/string`.

xc!+? (xc c-addr1 u1 – c-addr2 u2 f) If the buffer at c-addr1 u1 is big enough for xc, store xc there, f is true and c-addr2 u2 describe the rest of the buffer. If the buffer is too small, f is false and c-addr2 u2 is the same as c-addr1 u1.[3]

xc-size (xc – u) U is the number of chars that xc takes when stored in memory.

-trailing-garbage (c-addr u1 – c-addr u2) Given a string c-addr1 u1 containing xchars and further chars that do not form a complete xchar, c-addr u2 is the same string with only the complete xchars.

wcwidth (xc – u) U is the display width of xc on a monospaced display. Currently this word can produce the values 0, 1, 2.

display-width (xc-addr u – u2) u2 is the display width of the string xc-addr u on a monospaced display.

Ambiguous conditions exist, if the xchar(s) read from memory by `xchar+ xchar- +x/string -x/string xc@ xc@+ xc@+/string display-width` are not properly encoded xchars[4], or if the count would underflow (for `+x/string xc@+/string -trailing-garbage`).

In addition, words like `key`, `emit`, `char` and `[char]` have to be extended to work with xchars.

2.3 Requirements and Guarantees

An encoding to be used with the xchar words must have the following properties:

1. The length of an xchar can be determined in forward processing (every encoding has that property).
2. The length of an xchar can be determined in backwards processing (not every encoding has this property, but UTF-8, UTF-16, some encodings for Asian languages, and of course fixed-width encodings have it).
3. Partial xchars can be recognized (this is usually a consequence of backwards processability).
4. The on-stack representation of ASCII characters is the ASCII number (so that `char`, `emit` etc. work as expected).

[3] Bernd Paysan's reference implementation also cantains a word `xc!+ ( xc xc-addr1 -- xc-addr2 )`, but this word is cumbersome to use safely and easy to use not safely, leading to buffer overflows (like C's `strcat()`).

[4] E.g., in UTF-8, if an ASCII character is followed by a character in the range $80–$bf, or if the xchar is not encoded in the shortest possible sequence.

In addition, the following property is needed to ensure that all ANS Forth programs work on a system with xchars when processing ASCII-only strings (which is the only case that ANS Forth actually covers):

5. The in-memory encoding of ASCII characters is the same for xchars and chars.

2.4 Input and output

The xchars words were designed for having one universal encoding used throughout the Forth system. However, Forth code might have to deal with other encodings on I/O.

For I/O of text files (and other things supported by the Forth system with a file-like interface) in a specific encoding, the encoding of the external text could be specified in the fam (file access mode) parameter of `open-file` (with a `bin`-like word), and the reading and writing words would perform the conversion between the external and the internal encoding. One consequence of this conversion is that you usually cannot use file positions for such files in calculations to compute other file positions (because the size of the data in the file has little relation to the size of the data in the Forth system).

For text fields in binary files (e.g., Java `.class` files), the file has to be read/written in binary mode, and the text fields have to be converted between the external and the internal encoding with string conversion words.

These ideas have not been implemented in Gforth yet, and there are no word specifications yet.

2.5 Multiple internal encodings

Some people have suggested words for changing the Forth-internal encoding at run-time. We did not design xchars for such an environment, and would probably design an extension for such an environment differently. The way to deal with different encodings in the outside world in the xchars context is to convert them all to a universal encoding in the Forth system, and convert back on output.

The technical problem with switching between the internal encodings is that existing strings will continue to be in the old encoding, and interpreting them in the context of the new encoding will produce wrong results. So the program would have to keep track of which strings are in which encodings and always switch around between encodings, which is cumbersome and error-prone. And if two strings containing different encodings have to be used in the same operation (e.g., in `compare`), there is no way to set the switch right (and actually, with our xchars proposal `compare` does not encoding-dependent work).

3 Implementation scenarios

3.1 8-bit xchars and 8-bit chars

That is very easy to implement on top of current systems. It may appear pointless, but it allows to run code that uses xchars on systems that only deal with 8-bit characters. And it allows developing code on such systems that should work on systems with more featureful xchar implementations (although one should probably still test on a more featureful system). Gforth implements this scenario.

3.2 UTF-8 xchars and 8-bit chars

This combination satisfies all the requirements above (including requirement 5), as well as satisfying the widespread environmental dependency on `1 chars` = 1 (on byte-addressable machines). Moreover, the memory representation of a non-ASCII xchar consists only of non-ASCII chars; this means that even some programs working on individual characters will work on strings containing non-ASCII xchars, e.g., if the program searches for an ASCII character. Gforth implements this scenario.

3.3 UTF-32 xchars and 32-bit chars

This scenario satisfies all the requirements above (including requirement 5), but (on byte-addressable machines) not the environmental dependency on `1 chars` = 1. Xchars don't make much sense in that scenario, classical ANS Forth characters do everything they do.

3.4 UTF-32 xchars and 8 bit chars

This scenario satisfies all the requirements above except requirement 5; in addition it satisfies the environmental dependency on `1 chars`= 1. While such a system does not conform to ANS Forth (because requirement 5 is not satisfied), it probably takes less effort to port most programs to such a system than to a system like that in Section 3.3.

If this scenario would become the standard scenario, it would make sense to define a different wordset optimized for fixed-width wchars for it rather than our xchars wordset, which is designed for dealing with variable-width characters.

3.5 Other scenarios

Scenarios involving UTF-16 have similar tradeoffs to the UTF-32 scenarios, except that UTF-16 is a variable-width encoding.

4 Code examples

Here we present some examples of using the xchars words.

One thing that we noticed is that it is actually not that easy to find examples where characters are dealt with individually (instead of in strings).

The following word works like `type`, but prints the string back-to-front.

```
: revtype1 ( xc-addr u -- )
    over >r + begin
        dup r@ u> while
            xchar- dup xc@ emit
    repeat
    r> 2drop ;
```

One other thing we noticed is that often, instead of converting an xchar to the on-stack representation, it can just as well be treated as a string (and this is often more efficient):

```
: revtype2 ( xc-addr u -- )
    over >r + begin
        dup r@ u> while
            0 -x/string over swap type
    repeat
    r> 2drop ;
```

Here is another example, implementation of the widely-available word `scan` that searches for a character in a string. First, here is an xchar variant of the non-xchar version in Gforth:

```
: scan1 ( xc-addr1 u1 xc -- xc-addr2 u2 )
    >r
    BEGIN
        dup
    WHILE
        over xc@ r@ <>
    WHILE
        +x/string
    REPEAT  THEN
    rdrop ;
```

And here is a version that deals with the xc as string:

```
: xc->s ( xc -- xc-addr u )
    \ convert xc into ALLOCATEd
    \  in-memory representation
    dup xc-size  dup chars allocate throw
    swap ( xc xc-addr u )
    2dup 2>r xc!+? 0= abort" bug"
    2drop 2r> ;

: scan2 ( xc-addr1 u1 xc -- xc-addr2 u2 )
    xc->s 2dup 2>r search 0= if \ no match
        dup /string then
    2r> drop free throw ;
```

In many cases, the programmer can also provide the xchar as string and call `search` directly instead of through `scan2`.

Finally, here is a primitive implementation of `accept` for xchars.

```
: accept1 ( c-addr +n -- +n2 )
    over >r begin
        key dup #cr <> while ( c-a1 u1 xc )
            dup 2swap xc!+? >r rot r> 0= if
                drop #bell then
            emit
    repeat
    2drop r> - ;
```

5 Experience

We have implemented Xchars and UTF-8 support in the Gforth development version in December 2004 and January 2005, during the course of a month. The xchars addition itself took only a week (after earlier work on an UTF-8 specific wordset).

The main code changes were the addition of a 156-line file for UTF-8 handling, an 80-line file for generic xchar handling and for the 8-bit implementation, changes in `accept` (20 deleted lines, 117 lines added), and changes of less than 100 lines overall in about five other files.

Overall, these changes were relatively painless, and certainly much easier than the changes we would expect had we tried to change the char size.

One interesting challenge was that we did not implement `display-width`, and had to work around that lack in two places:

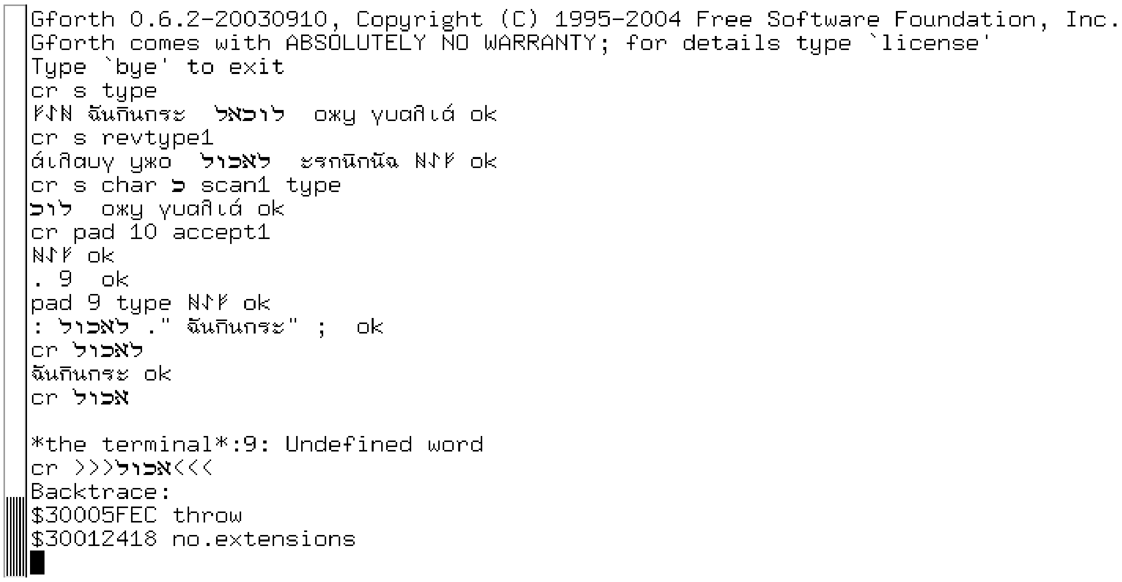
We use a pretty sophisticated editor for `accept`, where the user can move the cursor back and edit there without deleting the text. In order to achieve this without `display-width`, `accept` now always jumps to the start of the line, draws the part of the line before the cursors, remembers the cursor position, then draws the rest of the line and restores the cursor position to the remembered value.

The other problem was indicating where in an input line an error had happened. Originally Gforth did this by having a second line below the first with `^^^` characters pointing out the word. Now Gforth indicates the word by surrounding it with `>>>` and `<<<`.

Figure 1 gives an idea of how Gforth processing Unicode looks, including a case where an error message is shown.

6 Related work

Jax4th for Windows NT by Jack Woehr was one of the first dpANS Forth systems. It supported



```
Gforth 0.6.2-20030910, Copyright (C) 1995-2004 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
cr s type
ᚠᛇᚻ ฉันกินกระ לאכול φάω γυαλιά ok
cr s revtype1
άιλαυγ ωάφ לוכאל ะรกนิกนัฉ ᚻᛇᚠ ok
cr s char כ scan1 type
כול φάω γυαλιά ok
cr pad 10 accept1
ᚻᛇᚠ ok
. 9  ok
pad 9 type ᚻᛇᚠ ok
: לאכול ." ฉันกินกระ" ;  ok
cr לאכול
ฉันกินกระ ok
cr אכול

*the terminal*:9: Undefined word
cr >>>אכול<<<
Backtrace:
$30005FEC throw
$30012418 no.extensions
```

Figure 1: Gforth processes Unicode characters

the then-16-bit Unicode by making characters 16-bit in size, making use of the freedom that ANS Forth had given to Forth systems in this area (by making `1 chars` = 2). However, Jax4th was not used widely, and all widely-used systems implement `1 chars` = 1. More importantly, many near-ANS programs have an environmental dependency on `1 chars` = 1 and would break on systems like Jax4th. Therefore we decided to take a different approach in Gforth and introduced xchars.

Pelc and Knaggs [PK01] identified the same problems as we did (in particular the widespread environmental dependency on `1 chars` = 1), and similar to us propose adding new words for dealing with wider characters: They propose adding wide-character versions of the existing character and string words, for use with wide fixed-width encodings; the system and old-style applications would continue to use the regular character and string words, but applications could be converted to use these wide-character words. In contrast, we propose adding words that support variable-width encodings, but only for words that deal with individual characters; the string words work just as well for strings containing extended characters as for strings containing classical characters. Our approach requires less conversion work, so we propose applying it throughout the system instead of just to application data.

Java uses Unicode as character set and UTF-16 as internal character encoding.

Kuhn compiled an excellent resource on UTF-8 and Unicode [Kuh05], which is highly recommended for anyone having to deal with these issues and contains many links to other documents on the topic.

7 Conclusion

Xchars allow Forth systems to support Unicode (in particular in its UTF-8 encoding) in a relatively compatible way: String words (and programs using them) continue to work without changes; words dealing with individual characters work as usual with ASCII characters, but have to be adapted for working with extended characters.

Xchars can also be implemented easily on systems that only support 8-bit character encodings, so programs using xchars are not restricted to systems with Unicode (or other wide character) support.

Xchars have been implemented in Gforth and bigForth. The Gforth porting experience was relatively painless, requiring adding or changing only a few hundred lines of Forth code.

References

[Kuh05] Markus Kuhn. UTF-8 and Unicode FAQ for Unix/Linux. http://www.cl.cam.ac.uk/~mgk25/unicode.html, 2005.

[PK01] Stephen Pelc and Peter Knaggs. ANS Forth and large characters. http://www.mpeforth.com/arena/i18n.widechar.v7.PDF, 2001.