4tH, the *friendly* Forth compiler

J.L. Bezemer

25th January 2008

Contents

List of Figures

List of Tables

Chapter 1

What's new

1.1 What's new in version 3.5c, release 2

Words

- None.

Functionality

- A default 4tH directory can be defined by setting an environment variable.
- Support for creating custom 4tH implementations.
- *4tsh* is scriptable now.

Bugfixes

- None.

Developer

- The library files concerning ANS Core Extensions, table searching and interpretation have been rewritten or replaced.
- A superfluous `#define` was removed from `4th.h`.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5c programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly.

Library reorganization

Some words have been moved to another library file, so you might have to change your includes according to the following table:

Word	v3.5c	v3.5c, release 2
WITHIN	`anscext.4th`	`ranges.4th`
BETWEEN	`comus.4th`	`ranges.4th`
SAVE-INPUT	`anscext.4th`	`evaluate.4th`
RESTORE-INPUT	`anscext.4th`	`evaluate.4th`

Interpreter

The inclusion of `interprt.4th` has to be done at the *very beginning* of the program, like all other include files. "NotFound" now always uses the same stack diagram: it leaves the address/count string on the stack that could not be interpreted. "NotFound" is now a deferred word with default behaviour, so defining it is optional. Either remove the definition or change it from, e.g.:

```
: NotFound type ."  is not defined" cr ;
```

To:

```
:noname type ."  is not defined" cr ; is NotFound
```

The "dictionary" table used to be mandatory. Change it from e.g.:

```
create dictionary
```

To:

```
create wordlist
```

After you've *completely* defined the table add this line:

```
wordlist to dictionary
```

Your program should compile and run correctly now.

Table search

Both `find.4th` and `lookup.4th` have been superseded by `row.4th`. Since "ROW" works slightly different, you might have to do some rewriting. Please consult the primer if you're unsure how. If you're not willing to do that, there are two options:

1. Use the `find.4th` and `lookup.4th` from a previous version of 4th;
2. Use the following definitions:

```
: find
  ['] skey= is key= >r row
  if nip nip r> cells + @c true
  else r> drop drop false
  then
;

: lookup
  ['] nkey= is key= >r row
  if nip r> cells + @c true
  else r> drop drop false
  then
;
```

Reserved words

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[1]) ANS-Forth standard, except for porting purposes.

1.2 What's new in version 3.5c

Words

- The words 'C,' and 'OFFSET' have been added.

Functionality

- Binary string constants can be defined.

Bugfixes

- None.

Developer

- MakeSymbol() has been added to comp_4th().

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5b, release 2 programs to V3.5c shouldn't be any problem. Most of them will only need recompilation. There is one thing to consider:

[1] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'C,' and 'OFFSET'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[2]) ANS-Forth standard, except for porting purposes.

1.3 What's new in version 3.5b, release 2

Words

- Renamed 'FIELD' to '+FIELD'. The words '[NEGATE]', 'CHOP' and '/STRING' have been added.

Functionality

- None.

Bugfixes

- The word '->' allocated slightly more memory than needed. This has been fixed.

Developer

- The function hgen_4th() has been removed from the API.
- The library files have been updated and expanded.

Documentation

- All documentation now reflects the functionality of the current version. A section on the 4tH shell (*4tsh*) has been added.

Hints

Porting your V3.5b programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Renamed words

If you used 'FIELD' in your programs, you'll have to replace it by '+FIELD'. No other changes are necessary.

[2] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[NEGATE]', 'CHOP' and '/STRING'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[3]) ANS-Forth standard, except for porting purposes.

1.4 What's new in version 3.5b

Words

- The words '.|' and 'SYNC' have been added.

Functionality

- Output buffers can be flushed.

Bugfixes

- None.

Developer

- The CODE() and NEXT macros have been added to allow easy modification of exec_4th().

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5a release 2 programs to V3.5b shouldn't be any problem. Most of them will only need recompilation. There is one thing to consider:

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '.|' and 'SYNC'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[4]) ANS-Forth standard, except for porting purposes.

[3] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

[4] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

1.5 What's new in version 3.5a, release 2

Words

- Renamed 'SLEEP' to 'PAUSE'. The word 'FILES' has been added.

Functionality

- None.

Bugfixes

- A bad mode string disabled pipes in the Unix version. This has been fixed.

Developer

- None.

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.5a programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Renamed words

If you used 'SLEEP' in your programs, you'll have to replace it by 'PAUSE'. No other changes are necessary.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved word is 'FILES'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[5]) ANS-Forth standard, except for porting purposes.

[5] A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

1.6 What's new in version 3.5a

Words

- The words 'WORD', '"', 'TOKEN', 'COPY', 'TEXT' and 'WAIT' have been discarded.
- The words 'NUMBER', 'ARGS', 'IS', 'REPEAT', 'AGAIN' and 'UNTIL' have been changed.
- Renamed '@" to '@C', 'SKIP' to 'OMIT' and 'RESULT' to 'OUT'.
- The words '@GOTO', '+CONSTANT', 'SOURCE-ID', 'CIN, 'COUT', 'PARSE-WORD', 'IMMEDIATE', 'NOT', 'INCLUDE', '[UNDEFINED]', '4TH#', 'SLEEP', ',"', '2DUP', '2DROP', '2SWAP', '2>R', '2R>', 'S|', ',|', '+PLACE', '-ROT', 'BOUNDS', '2R@', 'R'@', 'UNLOOP', 'SOURCE', 'SOURCE!', 'DEFER@', 'DEFER!', '>BODY', 'SCONSTANT', ':THIS', 'DOES>', 'STRUCT', 'END-STRUCT', '->', 'FIELD', 'ENUM', 'SEEK', 'TELL', 'AKA', 'ALIAS' and 'HIDE' have been added.

Functionality

- The execution of a 4tH program can be suspended.
- A suspended 4tH program can be saved and reloaded.
- A 4tH program can be embedded in a MS batch file.
- User defined words can be made private.
- User defined words can be aliased.
- User defined terminal input buffers are supported.
- Complete, ANS-Forth compatible redesign of all string handling words.
- Multiple WHILEs are supported with REPEAT, AGAIN and UNTIL.
- Support for structures and enumerations has been added.
- Files can now be opened in read/write mode.
- File pointers can be interrogated and repositioned.
- Limited DOES> support has been added.
- More ANS-Forth, COMUS and TOOLKIT words have been added.

Bugfixes

- Several small bugs in the editor were fixed.
- A small bug in 'FILL' was fixed.
- A bug in hgen_4th.c that caused SEGFAULT was fixed.
- A security vulnerability in 4th.c was fixed.

Developer

- Several changes in exec_4th(), comp_4th(), save_4th() and load_4th() to support suspension.
- The function inst_4th() has been renamed to fetch_4th().
- The function store_4th() has been added.
- The Hcode structure has been expanded with the members CellSeg, UnitSeg and Offset.
- PAD has been converted to a circular buffer for temporary strings.
- Most of the string handling and all file functions in exec_4th() have been rewritten.
- The entire virtual machine was rewritten and its performance significantly improved.
- All internal 4tH variables are now located in a hidden area of the Integer Segment.
- The performance of 'MOVE' has been significantly improved.
- The library files have been updated and significantly expanded.

Documentation

- All documentation now reflects the functionality of the current version.
- There is now one single manual.

Hints

Porting your V3.3d release 2 programs to V3.5a may require some effort. In previous versions, string support was quite a mess (IMHO), requiring awkward words like 'COPY'. Some words returned or expected an address, others an address/count pair. With version 3.5a string support was completely redesigned. Consequently, source files using strings or arrays of string constants will have to be partially rewritten in order to make them compile and run properly. There are several things to consider:

Strings

There has been a conversion to the format recommended by the ANS-Forth standard. *All* strings are now represented by an address/count pair, with the exception of string variables and string addresses returned by '@C'. For this purpose, 'WORD' has been replaced by 'PARSE-WORD'. 'NUMBER' and 'ARGS' now return an address/count pair. Parsed strings are no longer copied to PAD, but remain in TIB and are *not* zero-terminated. However, since parsed strings are now represented by an address/count pair this should not be a problem.

Most programs we examined used constructions like this:

```
[char] ; word count type
s" 567" drop number
1 args count my_variable place
```

Those can easily be converted to:

```
[char] ; parse-word type
s" 567" number
1 args my_variable place
```

As a rule of the thumb, we advise you to use 'COUNT' *only* on string variables and string addresses returned by '@C'. You might find after a while, that these are the *only* situations where 'COUNT' is actually required. In all other situations, you use the count on the stack. Special operators like '2DUP', '2DROP' and '2SWAP' have been added to make manipulation of address/count pairs easier.

Please note that 'OPEN' already required an address/count pair, but simply discarded the count. In version 3.5a the count is *required*. If you didn't program properly, this might cause errors now. Well designed programs will continue to function properly.

We advise against the use of 'MOVE' or 'CMOVE' for moving strings. Most of these constructions will continue to work, but some may fail. In any case, they are not portable. Use 'PLACE' and '+PLACE' wherever you can.

PAD

PAD has been converted into a circular string buffer. Because some routines directly interface with their C counterparts, temporary zero-terminated strings are stored in PAD. When the buffer overflows it wraps around, overwriting whatever is there. Some previously correctly running programs may corrupt the PAD this way. If this happens, you can solve this by storing the overwritten string into a string variable. The reason for all this is that this now works:

```
s" This is not overwritten" s" By this string" compare
```

Number representations are not clobbered unless you use extremely long number formats.

Arrays of string constants

Consider this construction:

```
16 string weekday

create weekdays
  " Monday" ,
  " Tuesday" ,
  " Wednesday" ,
  " Thursday" ,
  " Friday" ,
  " Saturday" ,
  " Sunday" ,

weekdays 4 th @' weekday copy count type cr
```

'@" returns an address in the String Segment. 'COPY' is the only word in pre-3.5a versions that can access the String Segment. It copies the string from the String Segment to an address in the Character Segment and returns that address . In version 3.5a and up, '@" has been replaced by '@C'. '@C' is a lot smarter. It copies the zero-terminated string from the String Segment to PAD and returns that address. A 'COUNT' is still needed, but since all temporary strings in PAD are zero-terminated, this can safely be done:

```
create weekdays
  ," Monday"
  ," Tuesday"
  ," Wednesday"
  ," Thursday"
  ," Friday"
  ," Saturday"
  ," Sunday"

weekdays 4 th @c count type cr
```

Note that the string variable is no longer needed and the resulting code is much cleaner! String constants are now declared by a simple ',"'. "" has been discarded. '@C' also works for integer constants and behaves like '@".

Deferred words

Deferred words are now fully COMUS compatible. You have to change your programs only slightly:

```
defer my-vector

: do-nothing ;
' do-nothing is my-vector
my-vector execute
```

Just remove the 'EXECUTE':

```
defer my-vector

: do-nothing ;
' do-nothing is my-vector
my-vector
```

Please note that 'IS' is no longer an alias for 'TO'. If you have used illegal constructions like that, you'll have to correct them.

Library files

Note the library files have been revised and expanded. Some words have been renamed or placed into another file. Note that the 'toolbelt.4th' and 'comus.4th' library files are primarily intented for porting purposes. The 'easy.4th' library file is intended to port 4tH programs to other Forth compilers. Note that this only works for ANS-Forth compliant programs.

Dropped words

'WAIT' has been dropped and replaced by 'MS'. You'll have to 'INCLUDE' the library file 'ansfacil.4th' in order to use it. Note that this implementation is *very crude* and may vary between 0 and +1999 milliseconds[6].

'TOKEN' has been replaced by 'PARSE', which returns an address/count pair. 'WORD' has been replaced by 'PARSE-WORD', which returns an address/count pair. 'COPY' has been incorporated into '@C'.

'TEXT' has been dropped. If you treat a file as a text file, it will be handled as a text file. Just remove 'TEXT':

```
s" textfile.txt" input text + open
```

So now it reads:

```
s" textfile.txt" input open
```

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '@C', 'OMIT', 'OUT', '@GOTO', '+CONSTANT', 'SOURCE-ID', 'CIN, 'COUT', 'PARSE-WORD', 'IMMEDIATE', 'NOT', 'INCLUDE', '[UNDEFINED]', '4TH#', 'SLEEP', ',"', '2DUP', '2DROP', '2SWAP', '2>R', '2R>', 'S|', ',|', '+PLACE', '-ROT', 'BOUNDS', '2R@', 'R'@', 'UNLOOP', 'SOURCE', 'SOURCE!', 'DEFER@', 'DEFER!', '>BODY', 'SCONSTANT', ':THIS', 'DOES>', 'STRUCT', 'END-STRUCT', '->', 'FIELD', 'ENUM', 'SEEK', 'TELL', 'AKA', 'ALIAS' and 'HIDE'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list, TOOLBELT list or (proposed[7]) ANS-Forth standard, except for porting purposes.

1.7 What's new in version 3.3d, release 2

Words

- The word 'C"' has been discarded. The words '[NEEDS' and '[DEFINED]' have been added.

Functionality

- Source files can be included at compile time.
- The existence of words in the dictionary can be checked at compile time.
- More COMUS words have been added.
- The 4tH program allows you to enter parameters in the menu.
- The Linux module 'binfmt_misc' is supported.

[6]The "Forth Programmers Handbook" states that 'MS' should be *at least* the duration plus *twice* the resolution of the system (which is one second in 4tH).

[7]A proposed ANS-Forth standard is usually published on comp.lang.forth (usenet) by an ANS-Forth committee member.

Bugfixes

- None.

Developer

- Function open_4th() has been rewritten.
- The parser in comp_4th() has been changed significantly and is now much more transparant.
- There is an extra option in the menu of 4th.c

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.3d programs to release 2 shouldn't be any problem. All executables will run without recompilation. However, you might have to change a few source files in order to make them compile properly. There are two things to consider:

Dropped words

If you used 'C"' in your programs, you'll have to replace it by '"'. No other changes are necessary.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '[NEEDS' and '[DEFINED]'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the COMUS list or ANS-Forth standard, except for porting purposes.

1.8 What's new in version 3.3d

Words

- The words 'FILE' and 'TTY' have been discarded. The words 'FILE', 'AS', 'USE', 'DEFER', 'IS', 'STDIN' and 'STDOUT' have been added. The words 'INPUT', 'OUTPUT', 'OPEN' and 'CLOSE' have been changed.

Functionality

- Multiple files can be opened concurrently.
- More COMUS words have been added.

Bugfixes

- A segment violation was caused in 4th.c when an invalid sequence of commands was issued. This has been fixed.
- Better errorhandling when a pipe cannot be opened.

Developer

- The file support in function exec_4th() has been rewritten.
- Added DoInitValue().

Documentation

- All documentation now reflects the functionality of the current version.

Hints

Porting your V3.3c programs to V3.3d shouldn't be any problem. Most of them will only require recompilation, except when files are manipulated. There are three things to consider:

Using files

The new 4tH file handling module adds the concepts of streams and channels. You have two channels, an input channel and an output channel. In (standard) 4tH you have eight streams (you can increase this when you compile 4tH), two are already taken by the system (stdin and stdout). At startup the stdin stream is connected to the input channel and the stdout stream is connected to the output channel.

You can open additional streams by using the 'OPEN' word:

```
OPEN (a n fmod -- handle)
```

E.g.

```
s" ls" input pipe + open
```

This is not a significant deviation from V3.3c in which 'OPEN' returned only a flag. You can still interpret the handle as a flag since 'OPEN' returns zero when it failed.

To use the handle you only have to connect it to the appropriate channel. In V3.3c this was done by using:

```
input file
```

In V3.3d, you use the word 'USE'. 'USE' takes a handle and connects the stream to the appropriate channel.

```
file ls
s" ls" input pipe + open dup as ls
0= abort" Cannot open pipe"

ls use
```

In V3.3c you had to close a file by closing the channel, while the stream was still connected:

```
s" ls" input pipe + open
0= abort" Cannot open pipe"

input file
input close
```

In V3.3d you have to close the stream:

```
file ls

s" ls" input pipe + open dup as ls
0= abort" Cannot open pipe"

ls use
ls close
```

The default stream is reconnected to the channel, *even* if another stream was currently connected to that channel. We give you an example how 4tH now handles files in respect to the previous version:

VERSION 3.3C

```
s" hello.txt" output text + open
0= abort" Cannot open file"
output file
." Hello world" cr
output close
```

VERSION 3.3D

```
file hello
s" hello.txt" output text + open dup as hello
0= abort" Cannot open file"
hello use
." Hello world" cr
hello close
```

I hope you can appreciate the extended possibilities of 4tH and the way we tried to minimize breaking existing code.

Using 'INPUT' and 'OUTPUT'

Two new constants have been added to 4tH: 'STDIN' and 'STDOUT'. Before you could use 'INPUT' and 'OUTPUT' as follows:

```
input  file
output tty
```

Using 'INPUT' and 'OUTPUT' this way is *depreciated* and should be replaced by:

```
stdin  use
stdout use
```

Please use 'INPUT' and 'OUTPUT' *only* as flags for OPEN.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'AS', 'USE', 'DEFER', 'IS', 'STDIN' and 'STDOUT'

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

1.9 What's new in version 3.3c

Words

- The word '+UNDER' has been discarded. The words 'PIPE', 'PLACE', 'TOKEN', 'SKIP', 'PARSE', '/CELL', '/CHAR', 'ABORT"', '[ABORT]' and '[=]' have been added.

Functionality

- A complete mini-IDE has been added.
- Parsing has been enhanced significantly.
- The Unix version now supports pipes.
- More CORE words implemented.
- Some environmental dependancies can be checked at compiletime.

Bugfixes

- Reentry of several 4tH functions was seriously flawed, most notoriously in 'comp_4th()'. This has been fixed.

Developer

- Several new functions have been added, most significantly in the area of C source generation.
- The loading of sourcefiles is now done by open_4th(); fload() can still be used, but is no longer supported.
- The function save_4th() has been optimized. HX files are up to 50% smaller compared to those created by previous versions.
- The function dump_4th() has two extra arguments, allowing partial decompilation.
- The file support in function exec_4th() has been rewritten and now supports popen() and pclose().
- The demonstration program 4th.c has been completely rewritten.

Documentation

- All documentation now reflects the functionality of the current version.
- A document describing a sample session in 4tH interactive mode has been added.
- Several documents have been merged.

Hints

Porting your V3.3a programs to V3.3c shouldn't be any problem. Most of them will only require recompilation. There are two things to consider:

Programs using '+UNDER'

Which is no longer supported. If you have such programs, just add this definition at the top:

```
: +UNDER ROT + SWAP ;
```

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'PIPE', 'PLACE', 'TOKEN', 'PARSE', 'SKIP', '/CELL', '/CHAR', 'ABORT"', '[ABORT]', and '[=]'.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

1.10 What's new in version 3.3a

Words

- The words 'APPEND', 'TEXT', 'S"', '[*]', '[+]', '[NOT]' and '#!' have been added.
- The word 'OPEN' has been changed.

Functionality

- An output file can now be opened in "append" and "text" mode.
- A 4tH program can now be run from the shell.
- More CORE words implemented.
- Compiletime calculation is possible now.

Bugfixes

- When reallocation of the segments during compilation fails, resources are freed.
- When memory allocation of the header during the loading of an HX file fails, the file is closed.

Developer

- Dropped the EasyC syntax.
- Added the proper 'int main()' declarations.
- Modern prototypes, local include files and no stricmp() function are now the default behaviour.
- Dropped stricmp() from the distribution and added MatchName() to comp_4th().
- Added CompileString().

Documentation

- All documentation now reflects the functionality of the current version.
- The Developers Guide has been enhanced.

Hints

Porting your V3.2e programs to V3.3a shouldn't be any problem. Most of them will only require recompilation. There are two things to consider.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are '#!', 'S"', 'APPEND', '[*]', '[+]', '[NOT]' and 'TEXT'.

Changed words

The word 'OPEN' now takes an extra value from the stack. If you used an construction like this:

```
64 string filename
" myfile.dat" filename copy
input open
```

Change it to this:

```
s" myfile.dat" input open
```

If you used a construction like this:

```
refill drop
bl word
input open
```

Change it to this:

```
refill drop
bl word count
input open
```

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

1.11 What's new in version 3.2e

Words

- The words 'I'', 'R', 'QUERY', 'ENDIF', 'END', 'MINUS', 'NOT', 'ASCII', '2+' and '2-' have been discarded.
- The words ':NONAME', '?DO', 'BLANK', 'ERASE', 'CMOVE>', 'NIP', 'TUCK', '+UNDER' 'REFILL', 'D>S', 'RSHIFT', 'CATCH' and 'MAX-RAND' have been added.
- Renamed '-TRAIL' to '-TRAILING', 'STACK' to 'STACK-CELLS', '#PAD' to '/PAD', '#TIB' to '/TIB' and 'LIMIT' to 'MAX-N'.

Functionality

- The Character Segment is now unsigned, so no more negative characters.
- Vectored execution has been enhanced.
- Better implementation of 'RECURSE'.
- Better ANS-Forth compatibility by adding some commonly used words.
- Compatibility with Forth-79 has been dropped.

Bugfixes

- load_4th() closes the file when memory allocation failed.
- An error in GetImmediate(), GetConstant() and GetWord() has been fixed.
- DoRecurse() can now detect the use of 'RECURSE' outside a colon definition.

Developer

- Complete redesign of the parser. The whole parser now consists of the functions: ParseText(), ParseString() and ParseDirective(). Inline macros are supported.
- MoveString() does not require any arguments anymore.
- A textmode has been added to accept().
- Removed and added several tokens.
- The names of all internal words are now pointers instead of sized arrays, which means name can have any length now.

Documentation

- All documentation now reflects the functionality of the current version.
- The Developers Guide has been enhanced.
- The Porting Guide has been enhanced.

Hints

Porting your V3.1d programs to V3.2e shouldn't be any problem. There are three things to consider.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are ':NONAME', '-TRAILING', '?DO', 'BLANK', 'ERASE', 'CMOVE>', 'NIP', 'TUCK', '+UNDER', 'REFILL', 'D>S', 'RSHIFT', 'CATCH', '/PAD', '/TIB', 'STACK-CELLS', 'MAX-N' and 'MAX-RAND'. Most likely you have used these names for compatibility purposes, e.g.:

```
: rshift negate shift ;
```

In that case you can simply remove these definitions.

Dropped words

The Forth-79 words 'R', '2-', '2+', 'QUERY', 'ENDIF', 'END', 'MINUS', 'NOT', 'ASCII' and 'I" are no longer supported. '-TRAIL', '#PAD', '#TIB', 'LIMIT' and 'STACK' have been renamed. If you have programs that use these words then either modify them or add the following definitions:

```
: 2+ 2 + ;
: 2- 2 - ;
: i' r> r> r> dup >r rot rot >r >r ;
: r r> r> swap over >r >r ;
: query input tty refill drop ;
: minus negate ;
: not invert ;
: -trail -trailing ;
: #pad /pad ;
: #tib /tib ;
: limit max-n ;
: stack stack-cells ;
```

Unfortunately, you still have to replace the following words by their ANS-Forth equivalent, since there is no colon definition available for them:

CHANGE:	TO:
ASCII	CHAR, [CHAR]
END	AGAIN
ENDIF	THEN

Table 1.1: Forth-79 to ANS conversion

Unsigned characters

If a character with an ASCII value greater than 127 was fetched from the Character Segment, it was converted to a negative value. 'C@' will now return a positive value. This means that you can remove patches like these:

```
: c@' c@ dup 0< if 256 + then ;
```

On the other hand, if you have programs that rely on this negative value (e.g. by storing "-1" in a character), then you have to modify them.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

1.12 What's new in version 3.1d

Words

- The words 'AT' and 'ALLOT' have been discarded
- The words 'ARRAY', 'TABLE', '.(', 'ABORT', 'S>D', '"', 'RECURSE', '[IF]', '[THEN]', 'ARGS' and 'ARGN' have been added.

Functionality

- Better ANS-Forth compatibility (thank you, Wil Baden)
- Commandline arguments are now supported
- Nested assertions are now supported
- Conditional compilation is now supported.

Bugfixes

- ASCII bug has been fixed
- Several bugs in ParseText() and ParseStrings() have been fixed.

Developer

- Added SkipSource(), DecodeSymbol() and DecodeLiteral()
- Added two more arguments to exec_4th()
- Added two more tokens to cmds_4th.h
- Moved <limits.h> to 4th.h
- Added an extra compilation option "LOCAL_H" for those who cannot access /usr/include.

Documentation

- All documentation now reflects the functionality of the current version
- A 'Porting Guide' has been added
- A 'What's New' bulletin has been added
- The 'Developers Guide' has been enhanced
- The 'Primer' has been enhanced.

Hints

Porting your V3.1c programs to V3.1d shouldn't be any problem. There are five things to consider.

AT

'AT' has been discarded. Simply replace all occurences of 'AT' by 'STRING'. If you used the 'CHARS' keyword, you can leave right there since it doesn't have any effect, except when you are porting your program to Forth.

New reserved words

If you used the any of the new reserved words in your program as a name, you should replace those names by another. The new reserved words are 'ARRAY', 'TABLE', '.(', 'ABORT', 'S>D', '"', 'RECURSE', '[IF]', '[THEN]', 'ARGS' and 'ARGN'.

Using 'VALUE' with 'ALLOT'

If you ALLOTted any space to a VALUE, you should rewrite your code. Note that this is bad practice anyway. Example:

```
10 value room 10 cells allot \ allotting space to a VALUE
20 to room                   \ changing ROOM
5 ' room first + 4 th !      \ accessing allotted space
```

Change this to:

```
11 array room                \ define an ARRAY
10 room 0 th !               \ init 1st element of ROOM
20 room 0 th !               \ change 1st element
 5 room 4 th !               \ accessing allotted space
```

Using 'VARIABLE' with 'ALLOT'

This should be common practice to define cell arrays. However, as Wil Baden pointed out, this is not a common practice in ANS-Forth. Therefore, the word 'ARRAY' has been added, which can easily be implemented in both Forth-79 and ANS-Forth. All the programs using the old syntax have to be modified, though. Example:

```
variable room 15 cells allot
```

Change this to:

```
 16 array room
```

Special care must be taken of arrays that are sized using a constant, e.g. when the same constant is used to check the range. Example:

```
15 constant size
variable room size cells allot

: room?                      \ is it a valid variable?
  dup                        ( n n)
  size not and               ( n f)
  if                         \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
```

Change this to:

```
16 constant size
size array room

: room?                        \ is it a valid variable?
  dup                          ( n n)
  size 1- not and              ( n f)
  if                           \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
```

ANS-Forth compatibility

Sometimes it proved to be impossible to port a program to ANS-Forth since some constructions could not be implemented. There is no such thing as a 'state' in 4tH, which means that compilation- and interpretation semantics are completely the same, e.g.

```
c" This is a string" value addr
." String address has been stored in ADDR" cr
```

This is perfectly valid in 4tH, but cannot be ported in any way to ANS-Forth. With the new version, you can write:

```
" This is a string" value addr
.( String address has been stored in ADDR) cr
```

So if you have a 4tH program which you wanted to port to ANS-Forth, but couldn't, study the Porting Guide and try again. Note that no change is required if you do not intend to port your program to Forth. Apart from the modifications already mentioned you do not have to change a single line.

Our apologies for any inconvenience caused. It is certainly not our policy to change the syntax with every single version, but we found the arguments in favor of this change so strong that we didn't see any other way.

In order to prepare your programs for other changes, we strongly advise you not to use any names which are also mentioned in the ANS-Forth standard, except for porting purposes.

Part I

Getting Started

Chapter 2

Overview

2.1 Introduction

Like Forth, 4tH is a compiler and a interpreter. Unlike Forth you cannot switch between the two. Like Forth, 4tH runs Forth-programs. Not all of them but some. But in a quite different way.

Most things have already been written. There have been Forths written in a high level language. There have been portable Forths. There have been Forths that could interface with C. Different architectures have been used to implement Forth. There have been Forths that were 16 kB or even less.

Well, all of that has been done. But here is a compiler/interpreter that's all of the above. And none of them either. It sounds like an ancient Greek riddle, but it isn't. It's 4tH.

2.2 History

To understand 4tH you have to know how it came to be. As most things in life, 4tH developed slowly. Its predecessor is a C-function called strcalc(). This function is an implementation of a RPN calculator in one very compact function (about 6 kB source). It works with signed 32 bits integers and has about 20 commands and 20 variables. The C-programmer can add additional variables.

Using it in a C-program is very easy too. Just pass the source as a string and add any variables you need. It will return the result of that calculation.

Well, although primitive it can still be very useful. You can implement an interactive RPN calculator in less than 5 lines of C. It can also be used to make calculations from sources stored elsewhere, like in a file or an environment-variable. If you can store a string there, you can store strcalc() source.

But we were not satisfied. We wanted to create some successor to strcalc() that could be used to create applets, small applications that can be embedded in an application. Like strcalc() it had to be fast and compact and easy to use. All these requirements and 'Reverse Polish Notation'. What language comes to mind first? Forth.

There were a few advantages and disadvantages to that approach. First, if it looked like Forth, it had to be compatible with Forth up to a certain point. Second, if it looked like Forth, we wouldn't have to write thick manuals and explain how to use the language. Third, if it looked like Forth, could we make it crash-proof?

A user can easily crash a Forth-system. Store something at a wrong address and your system hangs. We don't like that, even when the user is at fault. So we had to make a few concessions somewhere, since adding checks means the program will be less compact and slower.

For a very long time we just didn't get the right idea. Then on a dark night in October 1994, it happened. The baby was called 4tH and could do everything strcalc() did.

It took quite a while before 4tH had successfully got away from its strcalc() roots. The very first version was very buggy and little more than an RPN calculator with (incompatible) flowcontrol and some string facilities. It required two passes to compile a source and the resulting bytecode could not be saved. The I/O was C-based and very primitive. There was no Character Segment.

The second version got string and file facilities. The I/O and flowcontrol was completely rewritten, so they now were fully Forth-compatible. The second pass was discarded and H-code could finally be saved. The first move to ANS-Forth was made.

The third version came to be when the H-code eXecutable was created. This fileformat made it possible to port bytecode across platforms. At the same time, 4tH moved more and more toward ANS-Forth. Exception-handling and assertions were introduced. And in the spring of 1997, version 3.1c was released to the general public.

Of course, 4tH didn't stop there. Since then, conditional compilation, enumerations, structures, forward declarations, inline-macros, pipes, source file inclusion, threads, private declarations and a small IDE have been added. The compatibility with ANS-Forth has been significantly improved. Neither the compactness nor the speed of 4tH have been compromised. It uses less memory than previous versions and is 50% faster.

2.3 Applications

4tH is an excellent platform to learn Forth. It looks and behaves like a conventional compiler, but essentially is Forth. A Forth that detects virtually every error and reports what was wrong and where it went wrong, but still is quite fast and compact.

But like any good teacher 4tH is quite strict. Forth allows constructions that should be avoided. 4tH on the other hand, either does not implement these words or restricts their usage.

Other Forth concepts are hard to handle, like the different wordsets for different kinds of numbers. 4tH only uses signed 32 bit integers, which enables the programmer to make a wide range of applications without being bothered by overflow. Pointers, integers and characters are transparently converted.

That doesn't mean that 4tH cannot be used as a scripting language anymore. There are still excellent facilities in 4tH to do just that. They are just modified in order to allow programmers to use 4tH as a stand-alone language. If you wonder how we did all that, here is the answer.

2.4 Architecture

4tH is a segmented Forth. There are different segments for constant strings, characters, cells and tokens. This shows you where each data-type is located:

- Return stack (Integer Segment)

- Data stack (Integer Segment)
- Variables & values (Integer Segment)
- String variables (Character Segment)
- Temporary storage (Character Segment)
- Compiled code (Code Segment)
- Compiled constants (Code Segment)
- String constants (String Segment)

The return-stack, data-stack and variables are allocated in one large array of signed 32 bit integers. On top of that 4tHs primitives check all parameters. This makes 4tH a very safe environment.

4tH also propagates clean programming. E.g. storing and fetching of the data-stack is not allowed. You can only store and fetch in the Variable Area.

In effect, as far as we know 4tH cannot be crashed by a user-program. The memory layout of the Integer Segment looks like figure 2.1.

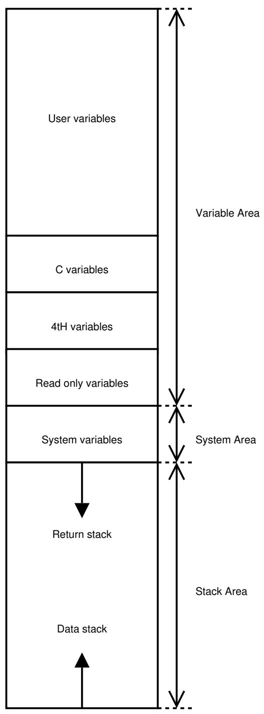

Figure 2.1: Integer segment layout

The allocation of variables is totally transparent to the C-programmer. He can also transfer C-variables to the user-program (application variables). These variables can be used like any other variable.

Combining return- and data-stack means the C-programmer only has to worry about the size of the stack and not the sizes of both stacks, thus allowing a wider range of user-applications with different requirements.

The Code Segment contains words. A word is a structure that contains a unsigned byte (the token) and a signed long integer (the argument). Only the argument can be accessed by the 4tH programmer. He cannot change the program in memory, since we never really liked self-modifying code.

True, this scheme has some redundancy, but a more elaborate scheme means a more code to encode and decode the tokens and arguments. That means the memory-space we saved by compacting the program-code will make the compiler and interpreter less compact. And it certainly won't run any faster!

The String Segment contains all string constants. The words which use strings contain an offset to the ASCIIZ strings in the String Segment. The 4tH programmer can copy strings from this segment, but cannot write any. Constants are constants.

Finally there is a chunk of memory the user can manipulate at will. It contains the TIB, the PAD and all string variables (if any). The memory layout of the Character Segment looks like figure 2.2.

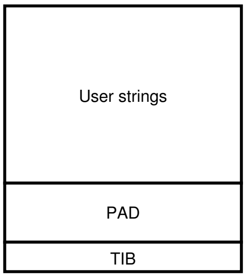

Figure 2.2: Character segment layout

The 4tH programmer can store and fetch anything here. Since 4tH uses some C-functions ASCIIZ strings are used. The words that act on counted strings take the same parameters and deliver functionally the same results.

File I/O is supported too in a more Forth-like way than Forth itself. You can have six concurrently open files and/or pipes. 4tH has threads too. A thread can be saved to disk and reloaded. The only restriction is that all files are closed when the execution of a thread is suspended.

2.4.1 The 4tH language

Most Forths use four different datatypes: signed 16 bit numbers, unsigned 16 bit numbers, signed 32 bit numbers and unsigned 32 bit numbers. The latter two are usually called "double numbers". Unlike C they all have their own operators. On top of that there are mixed operators too. Highly confusing!

We never liked that in the first place. Application programmers want to make an application. They don't want to worry whether any intermediate result could possibly be larger than 32767. So 4tH gets rid of most data- types and operators. It uses signed 32 bit numbers. That's it. No mixed, double or unsigned operators.

Second, a Forth programmer has to know how much address-units a cell takes. Since every data-type in 4tH has its own segment, the address-unit of a segment is always one, regardless the data-type. Consequently, ANS- Forth words like 'CELLS' and 'CHARS' are 'NOOP's. Which fits 4tH nicely.

Although 4tH has different words for storing and fetching different data- types, most of its vocabulary is still compatible with Forth. E.g. the word "C!" takes an address in the Character Segment and "!" takes an address in the Integer Segment. Since the Code Segment and String Segment do not allow any writing, there is no need for such operators.

Each segment has its own allocation operators too. 'VARIABLE', 'ARRAY' and 'VALUE' allocate space in the Integer Area. 'STRING' allocates space in the Character Area. Other words like "' and 'CREATE' have restricted functionality and compatibility with Forth.

4tH was originally loosely based on the Forth-79 standard, but now it supports most of the CORE wordset of ANS-Forth. Note that compatibility never had the highest priority. 4tH was designed to write applets, not to be the next "fully ANS-Forth compatible compiler with a little difference". If that is what you want, 4tH is not for you.

2.4.2 H-code

Long before the dawn of the original IBM-XT there was a language called UCSD Pascal. Like Forth, it was a compiler and an interpreter. In fact, it didn't compile source into object-code for some silicon-based processor. Instead it made P-code. So if you wanted to execute it, you needed a P-code interpreter for your system.

Such an interpreter can run faster than an ordinary interpreter since it doesn't interpret source-statements with all of its symbolic labels intact, but optimized P-code. It seems to have been discovered again, since Java and previous versions of Visual Basic work the same way. Visual Basic hides the interpreter in a DLL, but basically it doesn't work any different.

The 4tH uses the same basic architecture. First the source is compiled into H-code. Then the H-code interpreter is run. A token is a very simple structure. It's got a single byte instruction and an argument. Here's a sample of disassembled H-code:

```
[62] CR (0)
[63] VARIABLE (2)
[64] @ (0)
[65] 1- (0)
[66] DUP (0)
[67] VARIABLE (2)
[68] ! (0)
[69] 0BRANCH (62)
```

BTW, building a decompiler for tokenized code is quite simple. There is one for Visual Basic and it seems like one emerged for Java too. The H- code was the result after compiling this little piece of source code:

```
cr begin times @ 1- dup times ! until
```

You can clearly see that everything is actually compiled. Flow-statements are compiled into BRANCH and 0BRANCH instructions pointing to addresses in the Code Segment.

Compiled H-code can be used on its own. It can be kept in memory, loaded, saved, decompiled and executed. H-code is a combination of the String Segment, the Code Segment and a header (figure 2.3). The header contains all the information to set up the runtime environment and some information on the String- and the Code Segments. The Integer Segment and the Character Segment are created at runtime.

Although speed was an issue when 4tH was designed, it is beaten by some other Forths. There are several possible explanations.

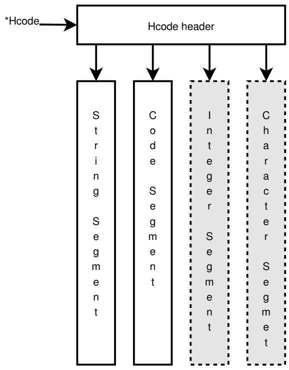

Figure 2.3: Hcode structure

- 4tH uses 32 bit numbers, while most other Forths use only 16 bit numbers
- 4tH checks all parameters, while other Forths depend on signals or don't do any checking at all
- 4tH is written in C, while some other Forths are written in assembler

When 4tH is compiled with a 32-bit compiler it outruns Python, Perl and most other C-based Forths (upto 4 times) or has a comparable performance (with the possible exception of GCC optimized Forth compilers). In real life applications the difference is barely noticeable.

To make compiled H-code portable, a separate scheme was developed: the Hcode-eXecutable. Or HX-file for short. It contains all the information in the header, a compacted Code Segment, the String Segment and some additional information on compatibility and integrity. Numbers are stored in an architecture-independant way.

2.4.3 H-code compiler

The H-code compiler looks a lot like any conventional compiler or assembler. Basically it is a simple one-pass compiler. In order to understand the workings of 4tH you have to know that not all H-code instructions are equal:

- Immediate words (flow control, declarations, etc.)
- Predefined constants (addresses, aliases, etc.)
- Simple words (do not require an argument)
- Symboltable entries (user-definitions)

To determine the initial size of both the Code Segment and the symbol- table the source is parsed first and the actual number of words counted. This determines the initial size of the Code Segment with a high degree of accuracy, so extending the Code Segment is never necessairy. After compilation the Code Segment will be shrunk to its actual size.

The parser can distinguish between directives and string constants. The size of the symbol-table is determined by simply counting all definitions. Every definition needs one symbol-table entry. That makes determining the size of the symbol-table very easy.

During compilation all simple words are compiled into tokens without a valid argument. When a definition is encountered, like a colon-definition or a variable-declaration, a symbol is added to the symbol-table.

There are four compiler directives which determine how a number is interpreted. '[BINARY]' interprets numbers as binary numbers, '[HEX]' interprets them as hexadecimal numbers. '[DECIMAL]' and '[OCTAL]' are available too. The "simple words" 'HEX', 'DECIMAL' and 'OCTAL' only act during execution and do not determine how a number is interpreted during compilation.

During compilation the compiler also resolves all flow words. It simply matches the correct instruction and enters the jump-address into the argument of the 'BRANCH', '?DO', 'LOOP', '+LOOP', 'CALL' or '0BRANCH' word. The way 4tH handles flow control is almost completely identical to Forth.

It may sound strange, but colon-definitions are also treated like flow-words. The colon simply compiles into a 'BRANCH' instruction that skips the colon definition.

When the user calls a colon definition, it simply compiles into a 'CALL' instruction that puts the current address on the return-stack and jumps inside the colon definition, after the 'BRANCH'. The semi-colon works like a RETURN instruction that pops the return address from the return-stack. Yes, like a subroutine in BASIC or assembler!

2.4.4 Error handling

When 4tH finds an error during compilation or execution it stops and sets the H-code member ErrNo. It works like "errno" in C. You can optionally link in an array of error-messages. ErrNo is an index to this array, which makes issuing the correct error message very simple. The instruction pointer is frozen at the point where the error occured, so it is very easy to find out where the error occured.

2.4.5 Interfacing with C

A minimal compiler would take only a few lines of C-code. The C-programmer can send C-variables to the interpreter, just like strcalc().

E.g. a compile takes a string-pointer as argument and returns a pointer to H-code:

```
object = comp_4th (source);
```

Executing H-code is easy too:

```
ReturnVal = exec_4th (object, argc, argv, 3, Var1, Var2, Var3);
```

Which would preload variables Var1, Var2 and Var3. You must specify how many variables are preloaded. Also 'argc' and '**argv' are available from the 4tH program.

The value returned by exec_4th() and stored into ReturnVal is the value of the 4tH variable 'OUT', which initially contains CELL_MIN. If an error occurs exec_4th() will *always* return CELL_MIN, regardless the value stored in 'OUT'.

Chapter 3

Installation Guide

3.1 About this package

4tH will compile ordinary text-files (MS-DOS and Unix) as well as block-files produced by the 4tH editor. The user-interface of this line-editor is highly compatible with conventional Forth block-editors.

4tHs special architecture almost forces you to write "clean" code, so you will learn Forth the proper way. This does not mean that you can't write portable code with 4tH. In fact, because Forth is so flexible you can usually write a small interface to your well-written 4tH-code in a matter of minutes.

You can use 4tH in virtually every environment, from Linux to MS-Windows. You don't even have to recompile your applications since 4tH uses a special executable format, that is interpreted by the 4tH virtual machine.

3.1.1 Example code

There are a lot of example programs, written in 4tH. From line-editors and calculators to adventure-games. Not all have been especially written for 4tH. There are quite a few programs from the hand of people like Professor C.H. Ting and Leo Brodie that started their existence as Forth-programs.

Most are available in source. That means they have the extension *'.4th'*. You can examine or edit them like any other source-file. Source-files written with the 4tH editor get the extension *'.scr'*. They can only be edited with the 4tH editor or other Forth blockfile editors. Executables have the extension *'.hx'* (Hcode eXecutable).

3.1.2 Main program

You will find a binary program within this package called 4tH. You can copy this binary to any directory. 4tH is a small development system by itself. When you start it, it will automatically enter interactive mode and show you a menu not unlike early versions of Turbo Pascal. You can edit, compile, run and debug programs from the 4tH prompt. Please read chapter 4 for more details.

You can also use 4tH from the commandline:

```
4th <commands> <file> [file | argument .. argument]
```

It takes most combinations of these ten commands:

m enter interactive mode

e edit a 4tH screenfile

c load a sourcefile (.4th) and compile it

l load an objectfile (.hx)

d decompile a 4tH program

g generate a C sourcefile (default: out.c)

s save a 4tH program (default: out.hx)

x execute a 4tH program

v enter verbose mode

q suppress copyright message

A few examples:

- To compile a 4tH program and save the object code: `4th csv <source.4th> [object.hx]`
- To compile a 4tH program and execute it: `4th cx <source.4th>`
- To decompile object code: `4th ld <object.hx>`
- To convert object code to C source: `4th lg <object.hx> [source.c]`
- To load and execute object code: `4th lx <object.hx> [arguments]`
- To load and execute object code without arguments: `4th <object.hx>`
- To edit a 4tH screenfile: `4th e <source.scr>`
- To enter interactive mode: `4th m <source.scr>`
- To enter interactive mode (without loading a screenfile): `4th`

Note: don't include the "[]" and "<>" in your commandline. They are just there to show whether an argument is optional ([arg]) or mandatory (<arg>).

3.1.3 Unix package

It is not possible for us to provide Unix binaries for all possible platforms, not now and not in the future, simply because we don't have access to them all. Here is a list of the Unix (like) platforms that are known to compile 4tH:

- Intel - FreeBSD
- Intel - Coherent
- Intel - Linux

- Intel - BeOS
- RS/6000 - AIX
- Zaurus - Linux
- Sun - Solaris
- ARM - RISC/OS
- Apple - Linux
- Apple - OS/X

If your platform is not listed, give it a try anyway. The chances are it will compile flawlessly, since we've never had a report of a Unix platform that refused to compile or run 4tH. Please send us an email with your results, so we can add it (or remove it) from our list.

You have to compile 4tH yourself, which is not difficult if you read the 'Developers Guide'. Usually this will do the trick:

```
make
make install
```

If you have any special needs, feel free to edit the makefile.

3.1.4 Linux package

You will find Linux binaries in this package. They will run under *most modern Linux distributions for Intel*. If the Linux binary doesn't run, you can easily recompile it. Just enter:

```
make
make install
```

You don't have to run './configure'. If you have any special needs, feel free to edit the makefile, e.g. compiling for the Zaurus means you have to add the '-DZAURUS' option.

You'll also find some icons for KDE or GNOME and a 'man' page. However, you have to install them manually. If you want to embed 4tH in KDE or GNOME you have to do that manually as well. Please consult your KDE or GNOME documentation.

3.1.4.1 /etc/magic

If you want Linux to recognize your 4tH files, you have to add the following lines to your /etc/magic file:

```
# From hansoft@bigfoot.com
# These are the magic numbers for 4tH HX files

0       belong          0x01020400      4tH eXecutable
>9      leshort x       \b, version %x
```

E.g. if you enter:

```
file editor.hx
```

It will respond:

```
editor.hx: 4tH eXecutable, version 35c
```

3.1.4.2 Using binfmt_misc

There is a module in Linux that will allow you to execute 4tH programs from the prompt without explicitly calling the 4tH interpreter. It is called 'binfmt_misc'. 4tH has built-in support for this module. Just add the following lines to your 'boot.local'[1] file:

```
insmod binfmt_misc
cd /proc/sys/fs/binfmt_misc
echo ':HX:M::\x01\x02\x04\x00\xff\xff\xff\x7f\x04\x5c\x03\x08::/usr/local/bin/4thx:' >register
```

If you use a kernel version later than 2.4.13 you have to add these lines:

```
insmod binfmt_misc
mount -t binfmt_misc none /proc/sys/fs/binfmt_misc
cd /proc/sys/fs/binfmt_misc
echo ':HX:M::\x01\x02\x04\x00\xff\xff\xff\x7f\x04\x5c\x03\x08::/usr/local/bin/4thx:' >register
```

You can find out whether 4tH support has been properly installed by issuing:

```
cd /proc/sys/fs/binfmt_misc
cat HX
```

And Linux should answer:

```
enabled
interpreter /usr/local/bin/4thx
offset 0
magic 01020400ffffff7f045c0308
```

Finally, you should go to the directory where 4tH has been installed (usually /usr/local/bin) and enter:

```
ln -s 4th 4thx
```

Now, after you've compiled a program you should make it executable and it will run like it is a native executable, e.g.:

```
4th cs asc2html.4th asc2html
chmod 755 asc2html
asc2html ascii7.4th ascii7.html
```

Note you have to be *root* in order to run some of these commands!

[1] On SuSE 'boot.local' is located in the /sbin/init.d directory.

3.1.4.3 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, 4tH will try to get it here. You can set this environment variable in your `.profile` or `.bashrc` file. Simply login into your default user account and type:

```
cd
vi .profile
```

or:

```
cd
vi .bashrc
```

This will launch the editor and allow you to edit the appropriate file. In this example your default 4tH directory is `/home/joe/4th`:

```
export DIR4TH=/home/joe/4th/
```

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `/home/joe/4th/lib/anscore.4th`. *Do not forget to add the trailing slash.* If you do, it will not work properly.

3.1.5 MS-DOS package

The "4th.exe" that is included in the MS-DOS package is a 32-bit MS-DOS version of the main Unix utility. It will only run on 80386 class machines and up. It allows you to compile and run very large 4tH programs. It requires CWSDPMI.EXE somewhere in your path. It is also available as "4th86.exe", which will run on any IBM-PC with 256 KB memory. This version is a bit slower and you may experience some memory restrictions.

3.1.5.1 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, 4tH will try to get it here. You can set this environment variable in your `autoexec.bat` file. In this example your default 4tH directory is `C:\4th`:

```
set DIR4TH=C:\4th\
```

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `C:\4th\lib\anscore.4th`. *Do not forget to add the trailing backslash.* If you do, it will not work properly.

3.1.6 MS-Windows package

Run "setup.exe" to install the package. It runs with Windows 95 OSR2 and up, Windows NT 4.0, Windows 2000, Windows XP and Windows Vista.

You can launch Explorer and double-click an HX-file. Windows will complain it doesn't recognize the file and tell you what to do. Browse to "4th.exe" and select it. After that you can click on an HX-file and it will be executed. You can even add HX-files to your desktop where they will start and run like ordinary Windows applications.

This is a true 32-bit version, so it does take long filenames, but you can't run it with Windows V3.x and early versions of Windows 95. It is a console application, so you'll need an MS-DOS box to run and use it. Note that it will exit immediately once a program has halted. We recommend you run 4tH from the MS-DOS prompt when you're using 4tH as a development environment.

3.1.6.1 DIR4TH environment variable

This variable is used to indicate where 4tHs default directory is. If a sourcefile cannot be found in the current directory, 4tH will try to get it here. In this example your default 4tH directory is `C:\4th`:

```
set DIR4TH=C:\4th\
```

If 4tH is unable to find a sourcefile, e.g. `lib/anscore.4th`, it will try to load `C:\4th\lib\anscore.4th`. *Do not forget to add the trailing backslash*. If you do, it will not work properly.

MS-Windows 9x While it is possible to set environment variables in the same way as for MS-DOS by editing `autoexec.bat`, it is easier to use `msconfig`. First run `msconfig` from the task bar by selecting "Run ".

Select the "Autoexec.bat" pane, then go to the bottom of the window, select the last entry and click the "New" button. A small input window appears below the last entry, and in this you should type a new entry with the exact syntax as shown in the example above. Then click "OK" and a small pen appears against the entry, indicating that `autoexec.bat` will be modified. You may have to reboot afterwards.

MS-Windows NT Click on the "My computer" icon or the "Start" menu, then click on the "Control panel". Click on the "System" icon to get the "System Properties" dialog box. For Windows NT use the "Environment" tab instead of the "Advanced" tab. Click on the "Environment Variables" button and select "New". Enter the `DIR4TH` and its value in the boxes and then click "OK".

If there are several users on the PC, it is probably better to set the variables as "System variables", rather than "User variables" since they will then automatically be accessible for all users. You will need to have Administrator rights to do this.

3.2 Now what?

After you've installed and played around with the utilities, we suggest you either click the 4tH icon on your desktop or start an interactive session by entering:

```
4th m session1.scr
```

And start reading the Primer. When you've thoroughly read and understood the very first section you're ready to go on. Start up your favourite editor (or use the built-in editor if you don't have one) and make your own very first 4tH program. If you don't know how to use the built-in editor, read chapter 4.

If you encounter an error during compilation or execution, refer to the 'Errors Guide' for a detailled description what it means, what probable causes are and how you can fix it.

3.3 Pedigree

4tH is basically an original work. However, some concepts have been derived from the work of other, much smarter people.

- The interpreter is taken from strcalc() and modified.
- The pictured numeric output and flow-control routines are based on Abersoft Forth.
- The exception handler is based on the dpANS-6 implementation.
- The enumerations are based on the Swift-Forth implementation.
- The structures are based on the GForth implementation.
- The 'ASSERT(' and ')' words are based on an idea implemented in GForth.
- The implementation of '[DECIMAL]', '[HEX]', '[OCTAL]' and '[BINARY]' was suggested by William Tanksley.
- The HX-format was suggested by Mikael Cardell.

4tH was discussed in Volume XVIII, Number 3 of Forth Dimensions. Thank you, Marlin Ouverson for giving me that opportunity.

3.4 Questions

We tried to provide you with all the documentation you'll probably ever need. That doesn't mean that you'll never have any questions. NEVER EMAIL THE PEOPLE WHOSE SITE YOU GOT THIS FROM! THEY DON'T KNOW EITHER! INSTEAD, MAIL TO:

```
hansoft@bigfoot.com
```

You'll usually get fast answers, although when your question is very complex ("how do I add floating point to 4tH?") we'll probably give you just some general directions. We have to stress that *any* comment is welcome, always.

3.4.1 4tH Website

You can visit our website, which is dedicated to 4tH. You will find all the latest information there, including additions and bugfixes (service packs):

```
http://hansoft.come.to
```

3.4.2 4tH Google group

We've got a Google group for discussions about 4tH. If you want to interact with other 4tH users, we recommend you subscribe to this group. You will also have to become a Google member if you are not already, e.g. when you already have a *gmail* account:

```
http://groups.google.com/group/4th-compiler
```

Important! Your posts will not be accepted by the server if you don't subscribe first!

3.4.2.1 Conditions of use

This group has been created as a service to, and in support of, the 4tH (and Forth) community. As in most discussion groups, there are a few rules to ensure the survivability of the group for the future.

1. This group is for discussions of 4tH problems, 4tH questions and answers. It is not to be used for non-4tH discussions.
2. This is not an 4tH advocacy group. Stick to 4tH questions and problem-solving or move your discussion to an appropriate channel. i.e. alternative site or private e-mail.
3. Flames, insults, foul language will not be tolerated. You will be unsubscribed and barred from re-subscribing under your present e-mail address.

3.4.2.2 What to discuss?

Well, Problems, wishes, needs, solutions (how you did something) basically anything 4tH related.

3.4.3 Newsgroup

There is no special newsgroup for 4tH. However, comp.lang.forth will prove to be able to answer most of your questions.

Chapter 4

A guided tour

4.1 4tH interactive

4tH's interactive mode was introduced with version 3.3c, but it is still fully compatible with previous versions, so you can still use all your external IDE's and script files. The interactive mode is especially useful when you are using an environment where other tools are not available or impossible to use. This document shows you how to use interactive mode and get the most out of it.

4.2 Starting up 4tH

You can enter 4tH's interactive mode by just clicking the icon (when you are using MS-Windows) or by issuing this command on the Unix or MS-DOS commandline:

```
4th
```

4tH will respond by showing you this screen:

```
(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>_
```

This is the main menu. It is slightly reminiscent to the earlier versions of Turbo Pascal. At the bottom is the prompt. Just press the appropriate key and hit enter, e.g. "S", which stands for the name of the screenfile. 4tH will now prompt you for the name of the screenfile. Note that 4tH is not case sensitive, so both "s" and "S" will do.

4.3 Running a program

We assume you've installed 4tH according to the instructions. If not, this might not work. Now press "S" and hit enter. 4tH will prompt you for the name of a screenfile:

```
Screen file name:
```

Answer by typing "examples/romans.scr"[1] and hit answer. 4tH will return to the menu:

```
Screen file name: examples/romans.scr

(S)creen file: examples/romans.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments
(Q)uit  (G)enerate  (B)uild  (D)ecompile

>
```

Now hit "R" and press enter. What now appears is your program that is actually running:

```
>r
Enter number: 2005
Roman number: MMV
```

After the program has ended, you will return to the menu. Well, that wasn't too hard, was it?

4.4 Starting an editing session

We start by entering the editor mode. Just type "e" and hit enter. Ignore any file opening errors. The "OK" prompt shows you you're now in the editor. Now type:

```
0 clear
```

This will erase the first screen and select it for editing. 4tH's editor is a typical Forth editor. Forth organizes its mass storage into "screens" of 1024 characters. Forth may have one screen in memory at a time for storing text. The screens are numbered, starting with screen 0.

Each screen is organized as 16 lines with 64 characters. The Forth screens are merely an arrangement of virtual memory and do not correspond to the screen format of the target machine.

Depending on memory model and operating system, you have either 28, 32 or 64 screens available. This will be sufficient in most situations. These screens correspond to a region in memory, which acts like a RAM drive.

The actual editing is done in an area that is called the 'workspace'. With the word 'clear' you wipe all information in the workspace. With the word 'list' you can select a certain screen for editing and load its information from the RAM disk into the workspace. The

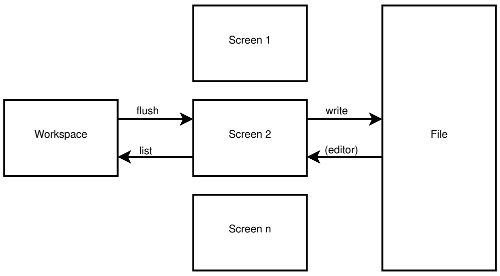

Figure 4.1: Editor architecture

figure below shows you how to transfer information between the screenfile, the RAM disk and the workspace (figure 4.1).

When you enter the editor the file is automatically loaded into the RAM disk. With 'list' you transfer the source from a screen in the RAM disk into the workspace. Since we started a new file (that's why you got the error message) all screens are empty. That why we cleared screen 0 and selected it for editing. You can quit the editor without changes by pressing "q" and hitting the enter key.

4.5 Writing your first 4tH program

We start our program by giving it a name. Press "s" and enter "hello.scr". Now we're going to enter the source text, so we start up the editor by pressing "e" (you know by now you have to press the enter key afterwards). Then we select screen 0 for editing by entering:

```
0 clear
```

If you want to know what you've entered so far you can list the editing screen by entering:

```
l
```

The editor will now show you a full listing:

```
Scr # 0
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
```

[1]This works for both Windows and Unix type Operating Systems.

```
 12
 13
 14
 15

 ^                                                                  0 OK
```

The first line will tell you which screen you're working on, which is screen 0. Then all sixteen lines are listed, all blank of course. Finally it will show you the current line, which is line 0. The "^" is the cursor, which is at the beginning of the line. You can move the cursor around with the "m" command. Try:

```
10 m
```

The editor will respond with:

```

^                                                       0 OK
```

And shows you this way that the cursor has moved 10 positions. If you want to move the cursor backwards, you can do that too. Just enter a negative value, like:

```
-5 m
```

And the cursor will move back five positions:

```

^                                                            0 OK
```

If you enter a larger value, that is perfectly acceptable too:

```
128 m
```

Note that every line is 64 characters long, so the editor will tell you you've just moved to line 2:

```

^                                                            2 OK
```

Don't be afraid that you'll do something wrong and lose your source. Note that this is 4tH, not Forth. If you try something funny like entering a very large value, the editor will just issue an error message:

```
1024 m
Off screen OK
```

You just tried to go beyond the workspace and the editor won't allow you to do that. Okay, we've moved around enough. How about writing that program? You can enter text with the "p" command, which stands for "PUT". Just provide the editor with the appropriate linenumber and the text:

```
0 p ." Hello world!" cr
```

Let's list our screen:

```
l

Scr # 0
  0 ." Hello world!" cr
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^." Hello world!" cr                                          0 OK
```

That's it. That's it? What about all that red tape like "Program Hello" or "int main()", opening parenthesis or closing braces? Hey, this is Forth[2], not C or something. You've just told the compiler it has to print the text "Hello world!" and write a newline. Isn't that what you wanted?

According to the figure in section 3, we first have to save the workspace in the RAM disk by entering "flush", then save it to disk by entering "write" and subsequently leave the editor by entering "q". Although perfectly correct, it is a lot of typing for just saving and exiting. You can do that a lot faster by just entering "wq", which stands for "Write and Quit".

Now we're back in the main menu and we want to see our program run. Just hit "R" and press enter. Don't we have to compile it first? Sure, but 4tH will notice your program hasn't been compiled yet and will compile it automatically for you. If you get an error message like this:

```
Compiling;      Word 0: Undefined name
```

Then you know you've just made a classical beginners error: there is a space between ." and the text. You'll have to go back to the editor to correct it. Reload screen 0 by entering:

```
0 list

Scr # 0
  0 ."Hello world!" cr
  1
```

[2]If you are not familiar with Forth and want to learn it, please read the primer. Everything you want to know is explained there in detail.

```
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
OK
```

Now let's see where our cursor is:

```
0 m

 ^."Hello world!" cr                                  0 OK
```

Now we know we have to move our cursor two positions and enter a space. Entering text at the cursor position is done by the "c" command, which stands for "COPY". Note that you have to add a space after each command, so adding a space at the cursor position is done by entering a "c" with <u>two</u> spaces:

```
2 m

 ."^Hello world!" cr                                  0 OK
c

 ." ^Hello world!" cr                                 0 OK
```

Now we can exit the editor again and rerun our program. Yes, 4tH will know you've changed the text and recompile your program automatically:

```
wq

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>r
Hello world!
```

That's it! You've just successfully entered, compiled and ran your very first 4tH program!

4.6 A more complex program

Note that this is not a tutorial on Forth. If you do not know the language you'll probably won't understand the statements we're going to enter. You don't have to, but if you need to please refer to our highly acclaimed 4tH primer.

Okay, let's presume you're looking at your 4tH prompt. We want to write a program which converts Unix ASCII files to DOS ASCII files. Unix ASCII files use a single linefeed to signify the end of a line while DOS ASCII files use an carriage return/linefeed pair for that purpose.

First, we need to name our program, so we press "s" to enter the name of the screen file. We'll call it "convert.scr". Then we enter the editor by pressing "e" and are greeted by the "OK" prompt. First we'll define a word (that's what a subroutine is called in Forth) that converts a file:

```
0 clear
0 p : ProcessFile
1 p   begin
2 p     refill
3 p   while
4 p     0 parse-word
5 p     type 13 emit 10 emit
6 p   repeat
7 p ;
```

Note that 4tH confirms you after each line that everything is "OK", but we left those messages out. When we list our program it looks like this:

```
l

Scr # 0
  0 : ProcessFile
  1   begin
  2     refill
  3   while
  4     0 parse-word
  5     type 13 emit 10 emit
  6   repeat
  7 ;
  8
  9
 10
 11
 12
 13
 14
 15

 ^: ProcessFile                                                   0 OK
```

It is a good custom to start each screen with a comment line, so others will know what we've been doing. However, line 0 is already taken. To insert a blank line we use the "s" command, which stands for "SPREAD". All lines following it will move down. If you happen to use line 15 you're in trouble since that one will be lost:

```
0 s 0 s l

Scr # 0
  0
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12
 13
 14
 15

 ^                                                                  0 OK
```

Yes, as long as you're not entering a command with a trailing text parameter, you can enter multiple commands on a single line. So this one tells the editor "spread at line 0, spread at line 0, list". Now we're going to enter our comment line:

```
0 p ( Conversion from UNIX ASCII files to DOS ASCII files - I)
OK
l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

That will do nicely. Although this word will do the job, we still have to open the input- and the output file. Since we want to test our program quickly we make a quick and dirty word that will do the job:

```
11 p : test s" code.txt" inpud open s" out.txt" outpud open
12 p over 0= over 0= or abort" Error!" use use;
13 p ttest ProcessFile
wq
```

When we try to compile this program by entering ”c”, it doesn’t work:

```
Compiling;      Word 17: Undefined name
```

Oops, we’ve obviously made an error, but where? Word 17? Where is word 17? We can find that out by decompiling the program and see where it went wrong. Just press ”d”:

```
Object size: 17 words
String size: 9 chars
Variables  : 0 cells
Strings    : 0 chars
Reliable   : No

[   8] type     (0)
[   9] literal  (13)
[  10] emit     (0)
[  11] literal  (10)
[  12] emit     (0)
[  13] branch   (0)
[  14] exit     (0)
[  15] branch   (0)
[  16] s"       (0)    code.txt
```

The last thing it compiled was the start of the ’TEST’ definition. It must have gone wrong right after that one. So we go back to the editor and find out. Sure, ”inpud” must be ”input”. We can even find out it we made more errors like this:

```
f pud

 : test s" code.txt" inpud^ open s" out.txt" outpud open          11 OK
n

 : test s" code.txt" inpud open s" out.txt" outpud^ open          11 OK
n
Not found OK
```

And yes, we did. On lines eleven and twelve to be exact. With the ”f” command (which stand for ”FIND”) we can find a string. By entering ”n” (which stands for ”NEXT”) we can find the same text again. Now we have to correct it. We’ll get back to the top of the screen and find the offending word:

```
top f pud

 : test s" code.txt" inpud^ open s" out.txt" outpud open          11 OK
```

Note that the cursor is positioned at the end of ”input”. We only have to wipe one character and insert the correct one:

```
1 w c t

 : test s" code.txt" inpu^ open s" out.txt" outpud open           11
 : test s" code.txt" input^ open s" out.txt" outpud open          11 OK
```

With the command ”1 w” we destructively backup the cursor by one position. Then we enter the ’t’ at the cursor position by using the ”c” command. However, there is a quicker way to do this:

```
x pud

 : test s" code.txt" input open s" out.txt" out^ open             11 OK
c put

 : test s" code.txt" input open s" out.txt" output^ open          11 OK
```

The ”x” command works very much like ”f”, but it does not only *find* the string, it also *deletes* it. Still, there are other errors left in the source:

```
f test

 ttest^ ProcessFile                                               13 OK
b

 t^test ProcessFile                                               13 OK
1 w
```

Yes, ”test” has an extra ”t”. So we find the next occurrance of ”test”. Note that a search is always performed from the cursor position, so the definition of ”test” is not found. The ”b” command will move the cursor backwards up to the point where ”test” begins and we can delete the superfluous ”t” with the command ”1 w”. The final typo we have to correct is a lacking space between ”then” and the semicolon. That can be fixed pretty quickly:

```
top f use

 over 0= over 0= or abort" Error!" use^ use;                      12 OK
n

 over 0= over 0= or abort" Error!" use use^;                      12 OK
till ;

 over 0= over 0= or abort" Error!" use use^                       12 OK
c  ;

 over 0= over 0= or abort" Error!" use use ;^                     12 OK
```

Now the cursor is positioned right after then. The ”till” command deletes everything from the current cursor position (indicated by the caret, remember?) to the end of the following string. In this case the semicolon but you can use any string. Finally, we copy the correct string into the text, which is a space followed by a semicolon. Four errors corrected. Let’s write the screen back to RAM disk and see what we have got:

```
flush l

Scr # 0
( Conversion from UNIX ASCII files to DOS ASCII files - I)
: ProcessFile
  begin
    refill
  while
    0 parse-word
    type 13 emit 10 emit
  repeat
;
: test s" code.txt" input open s" out.txt" output open
over 0= over 0= or abort" Error!" use use ;
test ProcessFile

^test ProcessFile                                              13 OK
```

Seems to be okay. Let's go back to the main 4tH screen by issuing the "wq" command. We recompile the source by pressing "c" and presto: we got a program! Simply hit "r" to run it. After we've run the program we find a file named "out.txt" in our working directory and examine it with a hex editor:

```
4261 6445 7865 6375 7465 2028 202d 2d20
2920 2d34 2045 5845 4355 5445 203b 0d0a
3a20 4261 6441 6464 7265 7373 2028 202d
2d20 2920 2d34 2040 2044 524f 5020 3b0d
0a3a 2042 6164 416c 6967 6e20 2820 2d2d
2029 2031 2040 2044 524f 5020 3b0d 0a
```

It seems our program is working perfectly. However, it doesn't seem very practical to copy a textfile to your working directory, rename it, start up 4tH, load your program and finally run it. That can be fixed. How? Well, you'll read that in the next section.

4.7 Advanced features

What we actually want is a program we can run from the prompt, something like:

```
convert in.txt out.txt
```

And if you do not provide the required parameters it has to issue an error message:

```
Usage: convert infile outfile
```

We will get there, but we still have some coding to do. First of all, we have to structure our program. We already have a working word[3] called "ProcessFile". It seems like a good idea to define two others, one that opens the files and one that closes the files. And we have to get rid of our "test" word. So let's fire up the editor and take care of that right now:

[3] A subroutine in Forth is called a "word", remember?

```
OK
0 list

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 : test s" code.txt" input open s" out.txt" output open
 12 over 0= over 0= or abort" Error!" use use ;
 13 test ProcessFile
 14
 15
OK
11 d l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 over 0= over 0= or abort" Error!" use use ;
 12 test ProcessFile
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

You can remove lines with the ”d” command, which stands for ”DELETE”. This will remove the line and move all remaining lines up. Line 15 becomes blank. But there is another way to get rid of unwanted lines:

```
11 e l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
```

```
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11
 12 test ProcessFile
 13
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

The "e" command, which stands for "ERASE", will leave every line at exactly the same position. It just *blanks* that line. Let's finish this:

```
11 p : Convert OpenFiles ProcessFile ;
OK
12 e
OK
13 p Convert
OK
l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2 : ProcessFile
  3   begin
  4     refill
  5   while
  6     0 parse-word
  7     type 13 emit 10 emit
  8   repeat
  9 ;
 10
 11 : Convert OpenFiles ProcessFile ;
 12
 13 Convert
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

Seems neat enough, but we still haven't got a "OpenFiles" word. This has to be defined before "Convert", but do we have still have room for that on screen 0? No, we haven't. Fortunately, you can insert screens with the 4tH editor[4]. Don't forget to flush. That is not only a good practice when you've visited the bathroom, but also when you're working with a Forth editor:

[4]Note that this command is usually *not* available in other Forth editors!

```
flush 0 insert
OK
```

We start our screen with a comment of course. We'll use the same comment as in our previous screen, so why not copy it?

```
1 list 0 h 0 list 0 r l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              0 OK
```

What did we do here? First, we switched to screen 1, which is our previous screen 0. Then we used the "h"[5] command, which copied line 0 into PAD. PAD is a buffer, which is able to hold the contents of a single line. Note that line 0 of screen 1 remains intact. It is only copied.

Then we switched back to screen 0 and issued the "r" command, which stands for "REPLACE". It replaces whatever is there with the contents of the PAD. Finally, we listed the screen. Let's play around a little with this PAD thing:

```
1 r 1 t

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              1 OK
```

Yes, the line we copied was still in PAD! We also used the command "t" to "TYPE" line 1. This command is very similar to "h", since it copies line 1 to PAD. But is also moves the cursor to the beginning of the line and types it. Let's see if you can explain this one:

```
1 d 2 r 2 t

 ^( Convert UNIX ASCII files to DOS ASCII files - I)              2 OK
```

Sure, the "d" command not only *deletes* the line, it also *copies* it to PAD. So when the "r" command is issued, it replaces line 2 with the contents of the line we deleted. Let's do one final test:

[5] In case you wondered, "h" stands for "HOLD".

```
0 i l

Scr # 0
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1 ( Convert UNIX ASCII files to DOS ASCII files - I)
  2
  3 ( Convert UNIX ASCII files to DOS ASCII files - I)
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^                                                               2 OK
```

Here we used the "i" command, which stands for "INSERT". It inserted the contents of PAD at line 0 and moved all the remaining lines down. Note that the cursor didn't move a bit. That's enough play for one day, let's get back to work:

```
1 e 3 e 2 p : OpenFile
OK
3 p   args 2dup 2>r rot open dup
OK
4 p   0= if
OK
5 p      drop ." Cannot open " 2r> type cr abort
OK
6 p   else
OK
7 p      dup use 2r> 2drop
OK
8 p   then
OK
9 p ;
OK
l

Scr # 0
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - I)
  1
  2 : OpenFile
  3   args 2dup 2>r rot open dup
  4   0= if
  5      drop ." Cannot open " 2r> type cr abort
  6   else
  7      dup use 2r> 2drop
```

```
  8   then
  9 ;
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - I)      0 OK
```

Hmm, it seems like we're going to need another screen. It is always wise to leave some room for future extensions, so this screen is full enough. But don't forget the commentline. We don't want to enter that one again, so let's store it in PAD:

```
0 h flush 1 insert 0 r l

Scr # 1
  0 ( Convert UNIX ASCII files to DOS ASCII files - I)
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15

^                                                               2 OK
```

Hold the line in PAD, flush the screen, insert screen 1 and replace line 0 with the contents in PAD. But the commentline is not entirely correct, so let's fix it:

```
top x I)

 ( Convert UNIX ASCII files to DOS ASCII files - ^                0 OK
c II)

 ( Convert UNIX ASCII files to DOS ASCII files - II)^             0 OK
```

The cursor is still on line 2, so we move it to the top again. Then we find and delete "I)". Finally we copy in "II)". We can do that since the cursor is at the right position. Now let's enter our final word:

```
2 p : OpenFiles
```

```
OK
3 p   argn 3 < abort" Usage: convert infile outfile"
OK
4 p   input  1 OpenFile
OK
5 p   output 2 Openfile
OK
6 p ;
OK
l

Scr # 1
  0 ( Conversion from UNIX ASCII files to DOS ASCII files - II)
  1
  2 : OpenFiles
  3   argn 3 < abort" Usage: convert infile outfile"
  4   input  1 OpenFile
  5   output 2 Openfile
  6 ;
  7
  8
  9
 10
 11
 12
 13
 14
 15

 ^( Conversion from UNIX ASCII files to DOS ASCII files - II)     0 OK
```

Almost there! We just have to fix the commentline in screen 2:

```
flush 2 list
OK
top x I)

 ( Convert UNIX ASCII files to DOS ASCII files - ^                0 OK
c III)

 ( Convert UNIX ASCII files to DOS ASCII files - III)^            0 OK
```

The current screen is flushed, then screen 2 is listed. We position the cursor at the top, find and delete "I)" and copy "III)" in at the cursor position. Done! Let's leave the editor and see what we have got. It compiles cleanly and when we run it it answers:

```
Usage: convert infile outfile
```

Sure, but what we actually want is to convert a file. Well, you can do that too without leaving 4tH. Just press "a" and enter the filenames, just like you would do at the prompt:

```
(S)creen file: convert.scr
```

```
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>a
Arguments: code.txt out.txt
```

When you press ”r” now, the arguments entered will be passed to your 4tH program, just like they would at the prompt. To clear the arguments, press ”a” again and just hit enter when prompted for arguments.

But how *do* we run it from the prompt? Easy, just press ”o” and enter ”convert.hx” at the prompt. Now press ”b”:

```
(S)creen file: convert.scr
(O)bject file: convert.hx

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>b
```

If 4tH has nothing to complain about, it doesn’t complain, so you can safely assume that everything is okay. Now we can go to the prompt[6] and run it:

```
user@linux:~ > 4th lxq convert.hx code out.txt
Cannot open code
user@linux:~ >
```

That was to be expected. Our file was called ”code.txt”, not ”code”. It is always a good idea to test all exceptions as well. There could be a bug in that code too.

```
user@linux:~ > 4th lxq convert.hx code.txt out.txt
user@linux:~ >
```

Well, it seems to work.. But what we really want is a standalone program. One that can be run without invoking 4tH and shared with our friends and families. Why this ”.hx” thing? HX-files do have their merits. First of all, it is very small, less than 200 bytes. But most importantly, you can take this file and run it on a Windows NT, MS-DOS or other Unix machine without modification or recompilation, provided a 4tH is available for that platform.

If you still want a standalone program, startup 4tH and reload ”convert.scr”. Then press ”o” and enter ”convert.c”. Isn’t that the extension of a C-program? Yes, it is. 4tH is able to generate C code. Just press ”g” and you’ve created a C program. You don’t even have to know C. If you know how to compile a C program that’s more than enough[7]. We assume you’ve installed the 4tH library and header files, since those are needed to compile ”convert.c”[8]:

[6] Windows users can do this by starting an MS-DOS session.

[7] Windows users need to consult the documentation that came with their C compiler. Some Windows compilers may not be able to compile standard C programs. MS-DOS users are encouraged to use the ’DJGPP’ compiler, which is free.

[8] Read the ”Developers Guide” if you are not sure how to do this.

```
user@linux:~ > cc -o convert convert.c -l4th
user@linux:~ > convert
Usage: convert infile outfile
user@linux:~ >
```

Is that all? No that's not all 4tH can do. We have a few surprises left.

4.8 Suspending a program

We've entered this program:

```
Scr # 0
." Is everybody in? The ceremony is about to begin.." cr
44596 36 base !
pause
." Wake up! Do you remember where it was?" cr
." Has this dream stopped? " . cr
```

The first line is simply a string we print to screen. The next line, we push a number on the stack and we change the radix. Then we go to sleep. After that, we wake up again, print a few lines and retrieve the number on the stack. Let's run it:

```
(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>r
Is everybody in? The ceremony is about to begin..

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>
```

At first, it seems like 'PAUSE' is nothing more than an alias for 'ABORT', but that is not entirely true. Let's save the executable and enter "r" one more time:

```
>b

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>r
Wake up! Do you remember where it was?
Has this dream stopped? YES

(S)creen file: new.scr
(O)bject file: out

(E)dit  (C)ompile   (R)un    (A)rguments

(Q)uit  (G)enerate  (B)uild  (D)ecompile

>
```

Now the *second* part of the program is run, that is the part after 'PAUSE'. Note that both the stack *and* the radix have remained intact. Every time 'PAUSE' is invoked, it will return you to the prompt. When you enter "r" again, it will continue where it left off, until it meets one of the following three conditions:

1. It encounters another 'PAUSE'; entering "r" will continue where it left off.
2. It encounters 'ABORT', 'QUIT' or 'ABORT"'; entering "r" will restart the program.
3. There are no more instructions to execute; entering "r" will restart the program.

But why did we save an executable? We'll have to go back to the shell to show you:

```
user@linux:~> 4th lxq out
Wake up! Do you remember where it was?
Has this dream stopped? YES
user@linux:~>
```

Entering "b" during suspension will save the program in its suspended state. When you run the resulting executable, it will behave like you've entered "r". That's neat, isn't it?

4.9 Calculator mode

Startup 4tH and enter the editor. We're going to show you this baby can do a lot more than just editing:

```
OK
.( Hello world!) cr
Hello world!
OK
```

Hey, that is a lot like the very first program we ran! Yes, it is. You can enter a subset of the 4tH language at the editor prompt, so you can test simple programs like this without getting into the ”edit-compile-run” cycle. You can even make some simple calculations:

```
OK
23 45 + .
68 OK
```

Simple? Aren’t the operators and operands entered in the wrong order? No, they aren’t. 4tH uses Reverse Polish Notation, which is also used by HP calculators and the Unix ”dc” command. 4tH has even eight built-in variables in which you can store numbers:

```
23 45 + A. !
OK
A. ?
68 OK
```

It even understands binary, hexadecimal and octal numbers:

```
23 45 + binary .
1000100 OK
1000011111 hex FACE octal 765 + + decimal .
65250 OK
```

This is called the ”calculator mode” and you don’t have to do anything if you want to use it. It is part of the editor command set. You can mix editor commands and calculations[9] as you like. Nice extra, isn’t it?

4.10 Epilogue

This concludes our tour of the 4tH interactive mode. We hope we’ve shown you what you can do with it and how to use it. Of course, you don’t have to use 4tH’s interactive mode. It will happily reside and cooperate with existing external IDE’s, editors and the like. But if memory is tight and you have nothing else, 4tH will prove to be a completely selfcontained environment.

If you’re still wondering what you can do with Forth and 4tH in particular, let me tell you this: if you worked your way through this tour, you’ve been working with Forth all the time. The entire editor is a 4tH program, embedded in the 4tH executable, taking up less than 3 KB. It is run by the very same interpreter as your initial ”Hello world!” program. Have fun!

[9]The full calculator command set is listed in the ”Editor reference guide”.

Part II

Primer

Chapter 5

Introduction

Don't you hate it? You've just got a new programming language and you're trying to write your first program. You want to use a certain feature (you know it's got to be there) and you can't find it in the manual.

I've had that experience many times. So when I wrote 4tH I promised myself, that would not happen to 4tH-users. In this manual you will find many short features on all kind of topics. How to input a number from the keyboard, what a cell is, etc.

I hope this will enable you to get quickly on your way. If it didn't, email me at 'hansoft@bigfoot.com'. You will not only get an answer, but you will help future 4tH users as well.

You can use this manual two ways. You can either just get what you need or work your way through. Every section builds on the knowledge you obtained in the previous sections. All sections are grouped into levels. We advise you to use what you've learned after you've worked your way through a level.

There are five levels. First, 4tH fundamentals. It assumes a working knowledge of programming and covers the basics. Second, 4tH arrays. We'll try to explain to you what an address is and teach you basic string handling.

Third, 4tHs Character Segment. We'll explain you how it is laid out and what you can do with it. Fourth, 4tHs Integer Segment and Code Segment. We'll explain you how it is laid out and what you can do with it.

Finally, advanced programming techniques. We'll teach you how to program multilevel exits, write interpreters, use jump-tables, emulate floating point calculation and a lot more!!

I don't think it is enough to teach you Forth, from which 4tH was derived, but you can always get a good textbook on Forth, like "Starting Forth" by Leo Brodie. Have fun!

Chapter 6

4tH fundamentals

6.1 Making calculations without parenthesis

To use 4tH you must understand Reverse Polish Notation. This is a way to write arithmetic expressions. The form is a bit tricky for people to understand, since it is geared towards making it easy for the computer to perform calculations; however, most people can get used to the notation with a bit of practice.

Reverse Polish Notation stores values in a stack. A stack of values is just like a stack of books: one value is placed on top of another. When you want to perform a calculation, the calculation uses the top numbers on the stack. For example, here's a typical addition operation:

```
1 2 +
```

When 4tH reads a number, it just puts the value onto the stack. Thus 1 goes on the stack, then 2 goes on the stack. When you put a value onto the stack, we say that you push it onto the stack. When 4tH reads the operator '+', it takes the top two values off the stack, adds them, then pushes the result back onto the stack. This means that the stack contains:

```
3
```

after the above addition. As another example, consider:

```
2 3 4 + *
```

(The '*' stands for multiplication.) 4tH begins by pushing the three numbers onto the stack. When it finds the '+', it takes the top two numbers off the stack and adds them. (Taking a value off the stack is called popping the stack.) 4tH then pushes the result of the addition back onto the stack in place of the two numbers. Thus the stack contains:

```
2 7
```

When 4tH finds the '*' operator, it again pops the top two values off the stack. It multiplies them, then pushes the result back onto the stack, leaving:

```
14
```

The following list gives a few more examples of Reverse Polish expressions. After each, we show the contents of the stack, in parentheses.

```
7 2 -          (5)
2 7 -          (-5)
12 3 /         (4)
-12 3 /        (-4)
4 5 + 2 *      (18)
4 5 2 + *      (28)
4 5 2 * -      (6)
```

6.2 Manipulating the stack

You will often find that the items on the stack are not in the right order or that you need a copy. There are stack-manipulators which can take care of that.

To display a number you use '.', pronounced "dot". It takes a number from the stack and displays it. 'SWAP' reverses the order of two items on the stack. If we enter

```
2 3 . . cr
```

4tH answers:

```
3 2
```

If you want to display the numbers in the same order as you entered them, you have to enter:

```
2 3 swap . . cr
```

In that case 4tH will answer:

```
2 3
```

You can duplicate a number using 'DUP'. If you enter:

```
2 . . cr
```

4tH will complain that the stack is empty. However, if you enter:

```
2 dup . . cr
```

4tH will display:

```
2 2
```

Another way to duplicate a number is using 'OVER'. In that case not the topmost number of the stack is duplicated, but the number beneath. E.g.

```
2 3 dup . . . cr
```

will give you the following result:

```
3 3 2
```

But this one:

```
2 3 over . . . cr
```

will give you:

```
2 3 2
```

Sometimes you want to discard a number, e.g. you duplicated it to check a condition, but since the test failed, you don't need it anymore. 'DROP' is the word we use to discard numbers. So this:

```
2 3 drop .
```

will give you "2" instead of "3", since we dropped the "3".

The final one I want to introduce is 'ROT'. Most users find 'ROT' the most complex one since it has its effects deep in the stack. The thirdmost item to be exact. This item is taken from its place and put on top of the stack. It is 'rotated', as this small program will show you:

```
1 2 3               \ 1 is the thirdmost item
. . . cr            \ display all numbers
                    ( This will display '3 2 1' as expected)
1 2 3               \ same numbers stacked
rot                 \ performs a 'ROT'
. . . cr            \ same operation
                    ( This will display '1 3 2'!)
```

6.3 Deep stack manipulators

No, there are no manipulators that can dig deeper into the stack. A stack is NOT an array! So if there are some Forth-83 users out there, I can only tell you: learn Forth the proper way. Programs that have so many items on the stack are just badly written. Leo Brodie agrees with me.

If you are in 'deep' trouble you can always use the returnstack manipulators. Check out that section.

6.4 Passing arguments to functions

There is no easier way to pass arguments to functions as in 4tH. Functions have another name in 4tH. We call them "words". Words take their "arguments" from the stack and leave the "result" on the stack.

Other languages, like C, do exactly the same. But they hide the process from you. Because passing data to the stack is made explicit in 4tH it has powerful capabilities. In other languages, you can get back only one result. In 4tH you can get back several!

All words in 4tH have a stack-effect-diagram. It describes what data is passed to the stack in what order and what is returned. The word '*' for instance takes numbers from the stack, multiplies them and leaves the result on the stack. It's stack-effect-diagram is:

```
n1 n2 -- n3
```

Meaning it takes number n1 and n2 from the stack, multiplies them and leaves the product (number n3) on the stack. The rightmost number is always on top of the stack, which means it is the first number which will be taken from the stack. The word '.' is described like this:

```
n --
```

Which means it takes a number from the stack and leaves nothing. Now we get to the most powerful feature of it all. Take this program:

```
2     ( leaves a number on the stack)
3     ( leaves a number on the stack on top of the 2)
*     ( takes both from the stack and leaves the result)
.     ( takes the result from the stack and displays it)
```

Note that all data between the words '*' and '.' is passed implicitly! Like putting LEGO stones on top of another. Isn't it great?

6.5 Making your own words

Of course, every serious language has to have a capability to extend it. So has 4tH. The only thing you have to do is to determine what name you want to give it. Let's say you want to make a word which multiplies two numbers and displays the result.

Well, that's easy. We've already seen how you have to code it. The only words you need are '*' and '.'. You can't name it '*' because that name is already taken. You could name it 'multiply', but is that a word you want to type in forever? No, far too long.

Let's call it '*.'. Is that a valid name? If you've programmed in other languages, you'll probably say it isn't. But it is! The only characters you can't use in a name are whitespace characters (<CR>, <LF>, <space>, <TAB>). Note that 4tH is not case-sensitive!

So '*.' is okay. Now how do we turn it into a self-defined word. Just add a colon at the beginning and a semi-colon at the end:

```
: *. * . ;
```

That's it. Your word is ready for use. So instead of:

```
2 3 * .
```

We can type:

```
: *. * . ;
2 3 *.
```

And we can use our '*.' over and over again. Hurray, you've just defined your first word in 4tH!

6.6 Adding comment

Adding comment is very simple. In fact, there are two ways to add comment in 4tH. That is because we like programs with a lot of comments.

You've already encountered the first form. Let's say we want to add comment to this little program:

```
: *. * . ;
2 3 *.
```

So we add our comment:

```
: *. * . ;          This will multiply and print two numbers
2 3 *.
```

4tH will not understand this. It will desperately look for the words 'this', 'will', etc. However the word '\' will mark everything up to the end of the line as comment. So this will work:

```
: *. * . ;        \ This will multiply and print two numbers
2 3 *.
```

There is another word called '(' which will mark everything up to the next ')' as comment. Yes, even multiple lines. Of course, these lines may not contain a ')' or you'll make 4tH very confused. So this comment will be recognized too:

```
: *. * . ;        ( This will multiply and print two numbers)
2 3 *.
```

Note that there is a whitespace-character after both '\' and '('. This is mandatory! However the closing paren) does not have to have a leading blank space. It is optional.

6.7 Text-format of 4tH source

4tH source is a simple ASCII-file. And you can use any layout as long a this rule is followed:

> All words are separated by at least one whitespace character!

Well, in 4tH everything is a word or becoming a word. Yes, even '\' and '(' are words! And you can add all the empty lines or spaces or tabs you like, 4tH won't care and your harddisk supplier either.

6.8 Displaying string literals

Displaying a string is as easy as adding a comment. Let's say you want to make the ultimate program, one that is displaying "Hello world!". Well, that's almost the entire program. The famous 'hello world' program is simply this in 4tH:

```
." Hello world!"
```

Compile this and it works. Yes, that's it! No declaration that this is the main function and it is beginning here and ending there. May be you think it looks funny on the display. Well, you can add a carriage return by adding the word 'CR'. So now it looks like:

```
." Hello world!" cr
```

Still pretty simple, huh?

6.9 Declaring variables

One time or another you're going to need variables. Declaring a variable is easy.

```
variable one
```

The same rules for declaring words apply for variables. You can't use a name that already has been taken. A variable is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive!

6.10 Using variables

Of course variables are of little use when you could not assign values to them. This assigns the number 6 to variable 'ONE':

```
6 one !
```

We don't call '!' bang or something like that, we call it 'store'. Of course you don't have to put a number on the stack to use it, you can use a number that is already on the stack. To retrieve the value stored in 'ONE' we use:

```
one @
```

The word '@' is called 'fetch' and it puts the number stored in 'one' on the stack. To display it you use '.':

```
one @ .
```

There is a shortcut for that, the word '?', which will fetch the number stored in 'ONE' and displays it:

```
one ?
```

6.11 Built-in variables

4tH has only three built-in variables. They are called 'BASE', '>IN' and 'OUT'. 'BASE' controls the radix at run-time, '>IN' is used by 'WORD' and 'OUT' returns a value to the host program.

6.12 What is a cell?

A cell is simply the space a number takes up. So the size of a variable is one cell. The size of a cell is important since it determines the range 4tH can handle. It also helps make code portable across machines with different cell sized, for example 16 bit and 32 big systems. We'll come to that further on.

6.13 What is a literal expression?

A literal expression is simply anything that *compiles* to a literal. All numbers, all defined constants and some expressions are compiled to a literal. In the glossary you can find what compiles to a literal, but we list them here too:

```
'      <name>
[']    <name>
CHAR   <char>
[CHAR] <char>
<literal> [NOT]
<literal> <literal> [*]
<literal> <literal> [+]
<literal> <literal> [=]
[DEFINED]   <name>
[UNDEFINED] <name>
```

6.14 Declaring arrays of numbers

You can make arrays of numbers very easily. It is very much like making a variable. Let's say we want an array of 16 numbers:

```
16 array sixteen
```

That's it, we're done! You must omit the word 'CELLS', since 'ARRAY' implicates that you want an array of numbers, not characters. The size is a literal expression. You can't take it from the stack or calculate it, so this is invalid:

```
3 5 * 1+ array sixteen
```

4tH will let you know that this is not a valid construction, but in case you wonder.. By the way, 4tH allows you size an array just like that as we will learn later on.

6.15 Using arrays of numbers

You can use arrays of numbers just like variables. The array cells are numbered from 0 to N, N being the size of the array minus one. Storing a value in the 0th cell is easy. It works just like a simple variable:

```
5 sixteen 0 th !
```

Which will store '5' in the 0th cell. So storing '7' in the 8th cell is done like this:

```
7 sixteen 8 th !
```

Of course when you want to store a value in the first, second or third cell you have to use 'TH' too, since it is a word. If you don't like that try defining 'ST', 'ND' and 'RD' yourself:

```
: st th ;
: nd th ;
: rd th ;
4 sixteen 1 st !
5 sixteen 2 nd !
6 sixteen 3 rd !
```

Isn't 4tH wonderful? Fetching is done the same of course:

```
sixteen 0 th @
sixteen 4 th @
```

Plain and easy.

6.16 Declaring and using constants

Declaring a simple constant is easy too. Let's say we want to make a constant called 'FIVE':

```
5 constant five
```

Now you can use 'FIVE' like you would '5'. E.g. this will print five spaces:

```
five spaces
```

The same rules for declaring words apply for constants. You can't use a name that already has been taken. A constant is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive. By the way, '5' is a literal expression. You can't take it from the stack or calculate it.

6.17 Built-in constants

There are several built-in constants. Of course, they are all literals in case you wonder. Here's a list. Refer to the glossary for a more detailed description:

```
/PAD
/TIB
/HOLD
/CELL
/CHAR
MAX-N
MAX-RAND
(ERROR)
BL
FALSE
LO
APP
PAD
STACK-CELLS
TIB
TRUE
VARS
WIDTH
INPUT
OUTPUT
STDOUT
STDIN
TEXT
APPEND
PIPE
FILES
4TH#
```

6.18 Using booleans

Booleans are expressions or values that are either true or false. They are used to conditionally execute parts of your program. In 4tH a value is false when it is zero and true when it is non-zero. Most booleans come into existence when you do comparisons. This example will determine whether the value in variable 'VAR' is greater than 5. Try to predict whether it will evaluate to true or false:

```
variable var
4 var !
var @ 5 > .
```

No, it wasn't! But hey, you can print booleans as numbers. Well, they are numbers. But with a special meaning as we will see in the next section.

6.19 IF-ELSE constructs

Like most other languages you can use IF-ELSE constructs. Let's enhance our previous example:

variable var

```
4 var !
var @ 5 >
if ." Greater" cr
else ." Less or equal" cr
then
```

So now our program tells you when it's greater and when not. Note that contrary to other languages the condition comes before the 'IF' and 'THEN' ends the IF-clause. In other words, whatever path the program takes, it always continues after the 'THEN'. A tip: think of 'THEN' as 'ENDIF'..

6.20 FOR-NEXT constructs

4tH has FOR-NEXT constructs as well. The number of iterations is known in this construct. E.g. let's print the numbers from 1 to 10:

```
11 1 do i . cr loop
```

The first number represents the limit. When the limit is reached or exceeded the loop terminates. The second number presents the initial value of the index. That's where it starts off. So remember, this loop iterates at least once! You can use '?DO' instead of 'DO'. That will not enter the loop if the limit and the index are the same to begin with:

```
0 0 ?do i . cr loop
```

'i' represents the index. It is not a variable or a constant, it is a predefined word, which puts the index on the stack, so '.' can get it from the stack and print it.

But what if I want to increase the index by two? Or want to count downwards? Is that possible. Sure. There is another construct to do just that. Okay, let's take the first question:

```
11 1 do i . cr 2 +loop
```

This one will produce exactly what you asked for. An increment by two. This one will produce all negative numbers from -1 to -10:

```
-11 -1 do i . cr -1 +loop
```

Note that the step is not a literal expression. You can change the step if you want to, e.g.:

```
32767 1 do i . i +loop
```

This will print: 1, 2, 4, 8, all up to 16384. Pretty flexible, I guess. You can break out of a loop by using 'LEAVE'. Note that 'LEAVE' only sets the index to the value of the limit: it doesn't branch or anything. Make sure that there is no code left between 'LEAVE' and 'LOOP' that you don't want to execute. So this is okay:

```
10 0 do i dup 5 = if drop leave else . cr then loop
```

And this is not:

```
10 0 do i dup 5 = if drop leave then . cr loop
```

Since it will still get past the '.' before leaving. In this case you will catch the error quickly, because the stack is empty.

6.21 WHILE-DO constructs

A WHILE-DO construction is a construction that will perform zero or more iterations. First a condition is checked, then the body is executed. Then it will branch back to the condition. In 4tH it looks like this:

BEGIN <condition> WHILE <body> REPEAT

The condition will have to evaluate to TRUE in order to execute the body. If it evaluates to FALSE it branches to just after the REPEAT. This example does a Fibbonaci test.

```
: fib 0 1
  begin
    dup >r rot dup r> >     \ condition
  while
    rot rot dup rot + dup . \ body
  repeat
  drop drop drop ;          \ after loop executed
```

You might not understand all of the commands, but we'll get to that. If you enter "20 fib" you will get:

```
1 2 3 5 8 13 21
```

This construct is particularly handy if you are not sure that all data will pass the condition.

6.22 REPEAT-UNTIL constructs

The counterpart of WHILE-DO constructs is the REPEAT-UNTIL construct. This executes the body, then checks a condition at 'UNTIL'. If the expression evaluates to FALSE, it branches back to the top of the body (marked by 'BEGIN') again. It executes at least once. This program calculates the largest common divisor.

```
: lcd
  begin
    swap over mod         \ body
    dup 0=                \ condition
  until drop . ;
```

If you enter "27 21 lcd" the programs will answer "3".

6.23 Infinite loops

In order to make an infinite loop one could write:

```
begin ." Diamonds are forever" cr 0 until
```

But there is a nicer way to do just that:

```
begin ." Diamonds are forever" cr again
```

This will execute until the end of times, unless you exit the program another way.

6.24 Including source files

4tH has a vocabulary of over 200 words. If you use them in one of your 4tH programs 4tH will recognize them instantly. These words are *internal*.

But if you take a look at the glossary, you'll find that there are a lot of other words too. Words that 4tH will not recognize; they have to be *included* first. These words are *external*.

These words are defined in an *include file*. An include file is just an ordinary ASCII file with 4tH source. You can read them if you want. In order to use these words, you have to tell 4tH where it can find the include file.

This is done by the '[NEEDS' directive, which is equivalent to the COMUS word 'INCLUDE' (which 4tH also supports). Everything up to the next "]" is considered to be a filename, so the path may contain embedded spaces. You can use absolute paths or relative paths, *just make sure that you're starting 4tH from the proper directory*. E.g. this one includes additional ANS-Forth CORE-words from the directory just above 'lib'[1]:

```
[needs lib/anscore.4th]
```

Or:

```
include lib/anscore.4th
```

4tH comes with a rich library of words, which covers a large part of ANS-Forth and COMUS[2] standard words and beyond. They are all located in the 'lib' directory. In the next level we're going to need a lot of these words, so you'd better know how to include them.

6.25 Getting a number from the keyboard

The word to enter a number from the keyboard can be found in the 'lib' directory and is defined in the 'enter.4th' file. To include it you have to tell 4tH. We *assume* your working directory is just above the 'lib' directory[3]:

```
[needs lib/enter.4th]
```

That's all! Now you can use 'ENTER' just like any 4tH word. This will allow you to enter a number and print it:

```
[needs lib/enter.4th]
enter . cr
```

By the way, this is the end of the first level. Take our advise and give it a try!

[1] If you're not sure where that is, enter the 'lib' directory and execute "cd ..".

[2] In case you wonder, COMUS stands for COMmon USage.

[3] As a matter of fact, we will *always* assume that! If you don't know what we mean, execute "cd <path to lib directory>" and then "cd ..". Now you're there for sure!

Chapter 7

4tH arrays

7.1 Aligning numbers

You may find that printing numbers in columns (I prefer "right-aligned") can be pretty hard. That is because the standard word to print numbers ('.') prints the number and then a trailing space. That is why '.R' was added.

The word '.R' works just like '.' but instead of just printing the number with a trailing space '.R' will print the number right-aligned in a field of N characters wide. Try this and you will see the difference:

```
140 . cr
150 5 .r cr
```

In this example the field is five characters wide, so '150' will be printed with two leading spaces.

7.2 Creating arrays of constants

Making an array of constants is quite easy. First you have to define the name of the array by using the word 'TABLE' or 'CREATE' (which is ANS-Forth). Then you specify all its elements. Note that every element is a literal expression. All elements (even the last) are terminated by the word ','. An example:

```
create sizes 18 , 21 , 24 , 27 , 30 , 255 ,
```

Please note that ',' is a word! It has to be separated by spaces on both ends.

7.3 Using arrays of constants

Accessing an array of constants is very much like accessing an array of numbers. In an array of numbers you access the 0th element like this:

```
sixteen 0 th @
```

When you access the first element of an array of constants you use this construction:

```
sizes 0 th @c
```

The only difference is the word '@C', which is exclusively used to access arrays of constants.

7.4 Using values

A value is a cross-over between a variable and a constant. May be this example will give you an idea:

declaration:

```
variable a              ( No initial value)
1 constant b            ( Literal expression assigned at compiletime)
2 b + value c           ( Expression assigned at runtime)
```

fetching:

```
a @                     ( Variable throws address on stack)
b                       ( Constant throws value on stack)
c                       ( Value throws value on stack)
```

storing:

```
2 b + a !               ( Expression can be stored at runtime)
                        ( Constant cannot be reassigned)
2 b + to c              ( Expression can be stored at runtime)
```

In many aspects, values behave like variables and can replace variables. The only thing you cannot do is make arrays of values.

A value is not a literal expression either, so you can't use them to size arrays. In fact, a value is a variable that behaves in certain aspects like a constant.

Why use a value at all? Well, there are situations where a value can help:

- When converting Forth programs (replacing constants)
- When a constant *can* change during execution

Note that although 'VALUE' and 'TO' are aliases, it is more portable and more readable to use 'VALUE' for declaration and 'TO' for reassignment. Note that each 'TO' or 'VALUE' consumes a little memory when compiling, so reassignments have to be rare. It is certainly not a good idea to replace all variables by values.

7.5 Creating strings

In 4tH you have to define the maximum length of the string, like Pascal:

```
10 string name
```

You cannot add the 'CHARS' keyword, since 'STRING' already implies that you are creating an array of characters. Note that the string variable includes the terminator. That is a special character that tells 4tH where the string ends (see section 7.13). You usually don't have to add that yourself because 4tH will do that for you. But you will have to reserve space for it.

That means that the string "name" we just declared can contain up to nine characters *AND* the terminator. These kind of strings are usually referred to as ASCIIZ strings.

E.g. when you want to define a string that has to contain "Hello!" (without the quotes) you have to define a string that is at least 7 characters long:

```
7 string hello
```

7.6 What is an address?

An address is a location in memory. Usually, you don't need to know addresses, because 4tH will take care of that. But if you want it, you can retrieve them as we will show you later. Think of memory like a city. It has roads and houses and inhabitants. There are three roads in 4tH city:

1. INTEGER SEGMENT, that is where the cells live;
2. CHARACTER SEGMENT, that is where the strings live;
3. CODE SEGMENT, that is where the instructions that form your program live.

If you want to visit a certain person, you go to the city where he lives, find the right street and knock on the door. If you want to retrieve a certain string or integer, you do the same.

When you define a string, you actually create a constant with the address of that string. When you later refer to the string you just defined its address is thrown on the stack. An address is simply a number that refers to its location. As you will see you can work with string-addresses without ever knowing what that number is. But *because* it is a number you can manipulate it like any other number. E.g. this is perfectly valid:

```
16 string hello

hello                      \ address of string on stack
dup                        \ duplicate it
drop drop                  \ drop them both
```

Later, we will tell you how to get "Hello!" into the string.

7.7 String literals

In 4tH a string literal is created by the word 'S"'. The word 'S"' is very much like '."', but instead of printing it to the screen you will just be defining a string literal.

```
s" This is a string"
```

4tH is a stack oriented language, so what does 'S"' leave on the stack? In 4tH, a string is usually represented by on the stack by its address and its count. So in order to get its length, you only have to get the first value on the stack. In order to get its address you have to get the second value on the stack, which is demonstrated by this small program:

```
s" This is a string"       \ create a temporary string
." Length : " . cr         \ show the length
." Address: " . cr         \ show the address
```

And what about string literals with quotes. Easy, there is an equivalent to 'S"' that does the same thing:

```
s| "This is a string with quotes"|
." Length : " . cr         \ show the length
." Address: " . cr         \ show the address
```

Instead of a quote, the string is delimited by a bar. And about a string literal that includes them both? Sorry pal, in that case you're out of luck!

7.8 String constants

String constants work the same way as numeric constants:

```
10 constant ten            \ define a string constant
ten . cr                   \ equivalent to: 10 . cr
```

In fact, you give a name to a literal value. After that, you can refer to that literal throughout your program by using its name. String constants do the same thing. Take a look at this little piece of code:

```
s" This is a string"       \ create a temporary string
." Length : " . cr         \ show the length
." Address: " . cr         \ show the address
```

Now we do the same thing, but this time we define a string constant by using 'SCONSTANT':

```
s" This is a string" sconstant mystring
                           \ define a string constant
mystring                   \ now we use the string constant
." Length : " . cr         \ show the length
." Address: " . cr         \ show the address
```

Why use string constants? Well, first of all, if you use a string constant throughout your program, it will save you some editing when you have to change your program for one reason or another. Second, it will make your program a little smaller.

7.9 Initializing string variables

You can initialize a string with the 'S"' word. If you want the string to contain your first name use this construction:

```
s" Hello!" name place
```

The word 'PLACE' copies the contents of a string literal into a string-variable.

If you still don't understand it yet, don't worry. As long as you use this construction, you'll get what you want. Just remember that assigning a string literal to a string that is too short will result in an error or even worse, corrupt other strings.

7.10 Getting the length of a string variable

You get the length of a string variable by using the word 'COUNT'. It will not only return the length of the string variable, but also the string address. It is illustrated by this short program:

```
32 string greeting         \ define string greeting
s" Hello!" greeting place  \ set string to 'Hello!'
greeting count             \ get string length
." String length: " . cr   \ print the length
drop                       \ discard the address
```

Most string handling words return or take an address/count pair. One of the exceptions is the string variable itself (see section 7.9). To copy the contents of an address/count pair represented string into a string variable, we use 'PLACE'. In order to convert a string variable back to an address/count pair represented string, we use 'COUNT':

```
32 string my-string        \ create a string variable
                           \ create an address/count
s" This is a string"       \ pair represented string
my-string place            \ copy it into the variable
my-string count            \ convert it into an address/count pair
." Length : " . cr         \ show the length
." Address: " . cr         \ show the address
```

Note that the contents of the string variable do not change by a 'COUNT' conversion!

7.11 Printing a string variable

Printing a string variable is pretty straight forward. The word that is required to print a string variable is 'TYPE'. It requires an address/count pair. Yes, that are the values that are left on the stack by 'COUNT'! So printing a string means issuing both 'COUNT' and 'TYPE':

```
32 string greeting           \ define string greeting
s" Hello!" greeting place    \ set string to 'Hello!'
greeting count type cr       \ print the string
```

If you don't like this you can always define a word like 'PRINT$':

```
: print$ count type ;
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting print$ cr          \ print the string
```

7.12 Copying a string variable

You might want to copy one string variable to another. Let's take a look at this example:

```
32 string one               \ define the first string
32 string two               \ define the second string

s" Greetings!" one place    \ initialize string one
one count                   \ get the length of string one
two place                   \ and copy it into string two
two count type cr           \ print string two
```

First we place the string "Greetings!" into a string variable. 'S"' will put an address/count pair on the stack, that is consumed by 'PLACE'. Variable "ONE" only puts its address on the stack, that is converted into an address/count pair by 'COUNT'. After it has been consumed again by 'PLACE' we need 'COUNT' again to provide 'TYPE' with an address/count pair.

7.13 The string terminator

In order for 'COUNT' to work, it has to know where the string stops. So a special character at the end of the string, the string terminator, is used to indicate the end of an ASCIIZ string. It has nothing to do with Arnold Schwarzenegger obliterating innocent strings! It is simply a character, having the ASCII value zero. It may also be referred to as the NULL-character. Although most strings in 4tH will be terminated automatically it is considered bad style to rely on that.

7.14 Slicing strings

Slicing strings is just like copying strings. We just don't copy all of it and we don't always start copying at the beginning of a string. We'll show you what we mean:

```
[needs lib/anstring.4th]

32 string one               \ define string one
s" Hans Bezemer" one place  \ initialize string one
one count 2dup type cr      \ duplicate and print it
1 /string                   \ move one character forward
2dup type cr                \ duplicate and print it again
1 /string                   \ move one character forward
2dup type cr                \ duplicate and print it again
1 /string                   \ move one character forward
type cr                     \ print it for the last time
```

First it will print "Hans Bezemer", then "ans Bezemer", then "ns Bezemer" and finally "s Bezemer". The word '/STRING' adjusts the address/count pair by a given number of characters, in this case one character. It is part of the library member 'anstring.4th', so we have to include that one. The word '2DUP' is much like 'DUP', but it copies the top *two* values on the stack. It is functionally equivalent to:

```
over over
```

If we want to discard the first name at all we could even write:

```
[needs lib/anstring.4th]

32 string one                \ define string one
s" Hans Bezemer" one place   \ initialize string one
one count 5 /string type cr  \ print sliced string
```

The five characters we want to skip are the first name (which is four characters) and a space (which adds up to five). There is a special word for slicing strings in the library member 'slice.4th'. You call it with:

```
address count position-to-start position-to-end
```

Both positions start counting at zero. So this will copy the first name to string "two" and print it:

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 0 3 slice          \ slice the first name
two place                    \ copy it to string two
two count type cr            \ print string two
```

This will slice the last name off and store it in string "two":

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 5 11 slice         \ slice the last name
two place                    \ copy it to string two
two count type cr            \ print string two
```

Since the last name is seven characters long and starts at position five (start counting with zero!).

7.15 Appending strings

The word '+PLACE[1]' appends two strings. In this example string "one" holds the first name. The second string literal is appended to string "one" to form the full name. Finally string "one" is printed.

```
32 string one              \ define string one

s" Hans " one place        \ initialize first string
s" Bezemer" one +place     \ append 'Bezemer' to string
one count type cr          \ print first string
```

7.16 Comparing strings

If you ever sorted strings you know how indispensable comparing strings is. As we mentioned before, there are very few words in Forth that act on strings. Here is a word that can compare two strings. It is located in the library member 'compare.4th'.

```
[needs lib/compare.4th]
                           \ compare two chars
32 string one              \ define string one
s" Hans Bezemer" one place  \ initialize string one
32 string two              \ define string two
s" HANS BEZEMER" two place  \ initialize string two

one count two count compare \ compare two strings
if
  ." Strings differ"        \ message: strings ok
else
  ." Strings are the same"  \ message: strings not ok
then
cr                          \ send CR
```

Simply pass two strings (represented by their address/count pairs) to 'COMPARE' and it will return a TRUE flag when the strings are different. This might seem a bit odd, but strcmp() does exactly the same. If you don't like that you can always add '0=' to the end of 'COMPARE' to reverse the flag.

You'll soon find out that ANS-Forth's 'COMPARE' is case sensitive. Lucky for you, you can modify the behaviour of 4tH's 'COMPARE'. Just define this *before* the '[NEEDS' directive:

```
false constant ignorecase
[needs lib/compare.4th]
                           \ compare two chars
32 string one              \ define string one
s" Hans Bezemer" one place  \ initialize string one
32 string two              \ define string two
s" HANS BEZEMER" two place  \ initialize string two
```

[1]There is a COMUS word called 'APPEND' which works exactly the same.

```
one count two count compare  \ compare two strings
if
  ." Strings differ"         \ message: strings ok
else
  ." Strings are the same"   \ message: strings not ok
then
cr                           \ send CR
```

Now 'COMPARE' will do a case sensitive comparison.

7.17 Finding a substring

Sometimes you need to find a string within a string. ANS-Forth has defined a word for that too. It is called 'SEARCH'. You need to include 'search.4th' in order to use it. Now lets find "the" in this string:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search               \ search for 'the'
0= if ." not " then ." found: "
type                         \ print the result
```

'SEARCH' always returns a flag and a address/count pair. If it returns true, the substring was found; if it returns false, the substring was *not* found. Now that's pretty straightforward, isn't it? That means that the small program above will print:

```
found:
```

When the substring was found and:

```
not found:
```

When the substring was not found. But what kind of string does it return when the substring was not found? Well, the entire string you fed it, so this *would have been* its output if we had been looking for the substring "now" instead of "the":

```
not found: How the cow catches the hare
```

But in this specific example we are looking for "the". When found, 'SEARCH' returns the string after the first occurrence of the substring we were looking for:

```
found: the cow catches the hare
```

Why that? Why not a position? Well, first of all, you can look for the same substring again:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search drop          \ drop the flag
2dup type                    \ print the string
s" the" search drop          \ now search again
type                         \ print the string
```

This will print:

```
the cow catches the hare
the hare
```

But if you still want to see a position instead of a string, you can simply define this:

```
[needs lib/search.4th]

: position
  2>r over swap 2r> search 0= >r drop swap - r> if 1- then
;

s" How the cow catches the hare"
s" the" position . cr
```

That will take care of your problems. If the substring was found, "POSITION" will return a positive number. If it wasn't found, it will return a negative number. Note that 'SEARCH' can be persuaded to do a case-sensitive comparison, just like 'COMPARE':

```
false constant ignorecase
[needs lib/search.4th]
```

Now 'SEARCH' will do a case sensitive comparison, just like 'COMPARE'.

7.18 Replacing substrings

Sometimes finding is not enough. You have replace it by something else. You can do that very easily with 4tH. Just include "replace.4th". It contains a word that will do all that. Take this example:

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the" s" a"
replaceall type cr
```

It will print:

```
How a cow catches a hare
```

Yes, this one replaces all occurrences of "the" by "a". Note that like 'COMPARE' and 'SEARCH' this one can be made case sensitive too:

```
false constant ignorecase
[needs lib/replaceall.4th]
```

7.19 Deleting substrings

Yes, we even got a word for 'search-and-destroy' missions. You only have to include "replace.4th":

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the"
deleteall type cr
```

This will print:

```
How cow catches hare
```

Yes, it deletes all occurrences of "the". Note that like 'COMPARE', 'SEARCH' and 'REPLACEALL' this one can be made case sensitive too:

```
false constant ignorecase
[needs lib/replaceall.4th]
```

7.20 Removing trailing spaces

You probably know the problem. The user of your well-made program types his name and hits the spacebar before hitting the enter-key. There you go. His name will be stored in your datafile with a space and nobody will ever find it.

In 4tH there is a special word called '-TRAILING' that removes the extra spaces at the end with very little effort. Just paste it after 'COUNT'. Like we did in this example:

```
32 string one              \ define a string
s" Hans Bezemer    "       \ string with trailing spaces
one place                  \ now copy it to string one

one dup                    \ save the address

." ["                      \ print a bracket
count type                 \ old method of printing
." ]" cr                   \ print bracket and newline

." ["                      \ print a bracket
count -trailing type       \ new method of printing
." ]" cr                   \ print a bracket and newline
```

You will see that the string is printed twice. First with the trailing spaces, second without trailing spaces.

7.21 Removing leading spaces

And what about leading spaces? Patience, old chap. You've got a lot of ground to cover. There is no built-in word for that, but we can use a library member like we did in this example:

```
[needs lib/scanskip.4th]

32 string one                \ define a string
s"     Hans Bezemer"         \ string with leading spaces
one place                    \ now copy it to string one

one dup                      \ save the address

." ["                        \ print a bracket
count type                   \ old method of printing
." ]" cr                     \ print bracket and newline

." ["                        \ print a bracket
count -leading type          \ new method of printing
." ]" cr                     \ print a bracket and newline
```

You will see that the string is printed twice. First with the leading spaces, second without leading spaces. Happy?

7.22 Upper and lower case

Sometimes you will have to convert a string to upper or lower case. 4tH has a library member for that too. Just include:

```
[needs lib/ulcase.4th]
```

This will define several easy to use conversion words. E.g. in order to convert a string to upper case, just enter:

```
s" Convert this!" s>upper    \ convert addr/count string to uppercase
type cr                      \ type the string
```

Its lower case counterpart is:

```
s" Convert this!" s>lower    \ convert addr/count string to lowercase
type cr                      \ type the string
```

Like most string words it takes and returns an address/count pair. Note that the string in question is *modified*, so if you still need the original, copy it first. You can also convert an individual character:

```
char A char>lower emit       \ convert a character and show it
```

And consequently, its counterpart is:

```
char a char>upper emit       \ convert a character and show it
```

These words take an ASCII value from the stack, convert it and put the converted ASCII value back on the stack. If the value does not represent a alphabetic character, it is left unchanged.

7.23 String literals and string variables

Most computer languages allow you to mix string literals and string variables. Not in 4tH. In 4tH they are two distinct datatypes. To print a string literal you use the word '."'. To print a string variable you use the 'COUNT TYPE' construction.

There are only three different actions you can do with a string literal. First, you can define one using 'S"'. Second, you can print one using '."' Finally, you can compile a string into your program using ',"'.

This may seem a bit mind-boggling to you now, but we'll elaborate a bit further on this subject later.

7.24 Printing individual characters

"I already know that!"

Sure you do. If you want to print "G" you simply write:

```
." G"
```

Don't you? But what if you want to use a TAB character (ASCII 9)? You can't type in that one so easily, huh? You may even find it doesn't work at all!

Don't ever use characters outside the ASCII range 32 to 127 decimal. It may or may not work, but it won't be portable anyway. the word 'EMIT' may be of some help. If you want to use the TAB-character simply write:

```
9 emit
```

That works!

7.25 Distinguishing characters

Like in a novel, not all characters are created equal. There are upper case characters, lower case characters, control characters, whitespace, etc. Sometimes it is necessary to find out what kind of character we are dealing with. Of course, 4tH can help you there. You need to include 'istype.4th' in order to use it:

```
char a is-lower . cr
char a is-upper . cr
```

4tH will first print a TRUE value (because 'a' is a lower case character) and then a FALSE value. This table tells you what words 4tH offers and the ranges of valid characters:

WORD	RANGE (ASCII)	DESCRIPTION
IS-ASCII	0 - 127	All 7-bit ASCII characters
IS-PRINT	32 - 127	As above, without control characters
IS-WHITE	0 - 32	All control characters plus space
IS-DIGIT	'0' - '9'	All digits
IS-LOWER	'a' - 'z'	All lower case characters
IS-UPPER	'A' - 'Z'	All upper case characters
IS-ALPHA	'a'-'z', 'A' - 'Z'	All alphabetic characters
IS-ALNUM	'0' - '9', 'a' - 'z', 'A' - 'Z'	All alphanumeric characters

Table 7.1: Character typing words

7.26 Getting ASCII values

Ok, 'EMIT' is a nice addition, but it has its drawbacks. What if you want to emit the character "G". Do you have to look up the ASCII value in a table? No. 4tH has another word that can help you with that. It is called 'CHAR'. This will emit a "G":

```
char G emit
```

The word 'CHAR' looks up the ASCII-value of "G" and leave it on the stack. You can also use '[CHAR]'. It does exactly the same thing. It is included for compatibility with ANS-Forth versions. Note that 'CHAR' only works with printable characters (ASCII 33 to 127 decimal).

7.27 Printing spaces

If you try to print a space by using this construction:

```
char   emit
```

You will notice it won't work. Sure, you can also use:

```
."   "
```

But that isn't too elegant. You can use the built-in constant 'BL' which holds the ASCII-value of a space:

```
bl emit
```

That is much better. But you can achieve the same thing by simply writing:

```
space
```

Which means that if you want to write two spaces you have to write:

```
space space
```

If you want to write ten spaces you either have to repeat the command 'SPACE' ten times or use a DO-LOOP construction, which is a bit cumbersome. Of course, 4tH has a more elegant solution for that:

```
10 spaces
```

Which will output ten spaces. Need I say more?

7.28 Fetching individual characters

Take a look at this small program:

```
32 string one                \ define string one
s" Hans" one place           \ initialize string one
```

What is the second character of string "one"? Sure, its an "a". But how can you let your program determine that? You can't use '@' because that word can only access variables.

Sure, you can do that in 4tH, but it requires a new word, called 'C@'. Think of a string as an array of characters and you will find it much easier to picture the idea. Arrays in 4tH always start with zero instead of one. So accessing the first character might be done with:

```
one 0 th c@
```

We do not recommend using this construction, although it will work perfectly. If you never want to convert your program to Forth you might even choose to keep it that way. We recommend the construction:

```
one 0 chars + c@
```

Which is slightly more wordy. 4tH will compile both constructions in exactly the same way. Anyway, accessing the second character is easy now:

```
one 1 chars + c@
```

This is the complete program:

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
one 1 chars + c@              \ get the second character
emit cr                       \ print it
```

7.29 Storing individual characters

Storing individual characters works just the same. Keep that array of characters in mind. When we want to fetch a variable we write:

```
my_var @
```

When we want to store a value in a variable we write:

```
5 my_var !
```

Fetching only requires the address of the variable. Storing requires both the address of the variable *and* the value we want to store. On top of the stack is the address of the variable, below that is value we want to store. Keep that in mind, this is very important.

Let's say we have this program:

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
```

Now we want to change "Hans" to "Hand". If we want to find out what the 4th character of string "one" is we write:

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
one 3 chars + c@              \ get the fourth character
```

Remember, we start counting from zero! If we want to store the character "d" in the fourth character, we have to use a new word, and (yes, you guessed it right!) it is called 'C!':

```
32 string one                 \ define string one
s" Hans" one place            \ initialize string one
one 3 chars +                 \ address of the fourth char
char d                        \ we want to store 'd'
swap                          \ get the order right
c!                            \ now store 'd'
```

If we throw the character "d" on the stack before we calculate the address, we can even remove the 'SWAP':

```
32 string one                 \ define string one
char d                        \ we want to store 'd'
s" Hans" one place            \ initialize string one
one 3 chars +                 \ address of the fourth char
c!                            \ now store 'd'
```

We will present the very same programs, but now with stack-effect-diagrams in order to explain how this works. We will call the index 'i', the character we want to store 'c' and the address of the string 'a'. By convention, stack-effect-diagrams are enclosed by parenthesis.

If you create complex programs this technique can help you to understand more clearly how your program actually works. It might even save you a lot of debugging. This is the first version:

```
32 string one               ( --)
s" Hans" one place          ( --)
one 3 chars                 ( a i)
+                           ( a+i)
char d                      ( a+i c)
swap                        ( c a+i)
c!                          ( --)
```

Now the second, optimized version:

```
32 string one               ( --)
char d                      ( c)
s" Hans" one place          ( c)
one 3 chars                 ( c a i)
+                           ( c a+i)
c!                          ( --)
```

7.30 Getting a string from the keyboard

Of course, you don't want to initialize strings all your life. Real applications get their input from the keyboard. We've already shown you how to get a number from the keyboard. Now we turn to strings.

When programming in BASIC, strings usually have an undefined length. Some BASICs move strings around in memory, others have to perform some kind of "garbage-collection". Whatever method they use, it takes up memory and processor-time.

4tH forces you to think about your application. E.g. when you want to store somebodies name in a string variable, 16 characters will be too few and 512 characters too many. But 64 characters will probably do.

But that poses a problem when you want to get a string from the keyboard. How can you prevent that somebody types a string that is just too long? And how do you terminate it?

The word 'ACCEPT' takes two arguments. First, the string variable where you want to save the input and second, the maximum number of characters it can take. It automatically terminates the string when reading from the keyboard. But there is a catch. This program can get you into trouble:

```
64 constant #name           \ length of string
#name string name           \ define string 'name'

name #name accept           \ input string
name swap type cr           \ swap count and print
```

Since 64 characters *plus* the terminator add up to 65 characters. The word 'ACCEPT' always returns the number of characters it received. You will find that you won't need that information most of the time.

This is the end of the second level. Now you should be able to understand most of the example programs and write simple ones. I suggest you do just that. Experience is the best teacher after all.

Chapter 8

Character Segment

8.1 The Character Segment

Wonder where all these strings are created? I bet you do. Well, when you define a string, memory is allocated in the Character Segment. When you define another one, space is allocated after the first string. That means that if you go beyond the boundaries of the first string, you'll end up in the space allocated to the second string.

After the second string there is a void. If you end up there your program will end with an error-message. And what about the space before the first string? Well, take a look at figure 8.1.

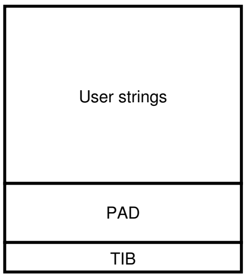

Figure 8.1: Character segment

The lower memory is at the bottom. Yes, before your strings there are two other areas, the TIB and the PAD. We'll elaborate on that in the next section.

The Character Segment is created at run-time. That means that it isn't there when you compile a program. The compiler just keeps track of how much memory would be needed to create such a Character Segment and stores that information in the header.

When you run the program the header is read first. Then the Character Segment is created, so it is already there when your program starts executing. When you exit the program, the Character Segment is destroyed and all information stored there is lost (unless you save it first).

8.2 What is the TIB?

The TIB stands for "Terminal Input Buffer" and is used by one single, but very important word called 'REFILL'. In essence, 'REFILL' does the same thing as 'ACCEPT', except that it has a dedicated area to store its data and sets up everything for parsing. Whatever you type when you call 'REFILL', it is stored in the TIB.

8.3 What is the PAD?

The PAD is short for "scratch-pad". It is a temporary storage area for strings. It is heavily used by 4tH itself, e.g. when you print a number the string is formed in the PAD. Yes, that's right: when you print a number it is first converted to a string. Then that string is 'COUNT'ed and 'TYPE'd. You can even program that subsystem yourself as we will see when we encounter formatted numbers (see section 8.8).

Furthermore, string constants (compiled by 'S"' or ',"') are temporarily stored in the PAD. Finally, 'NUMBER' and 'ARGS' also use the PAD. The PAD is actually a circular buffer. That means that strings are stored in the PAD until it runs out of space. Then it starts to overwrite the oldest strings. Usually, they have turned into garbage that is no longer used, but sometimes they still have some significance to your program. In that case, you'll have to save the string that was overwritten into a variable. Don't rely on the PAD to keep your strings alive!

8.4 How do I use TIB and PAD?

In general, you don't. The TIB is a system-related area and it is considered bad practice when you manipulate it yourself. The PAD can be used for temporary storage, but beware! Temporary really means temporary. A few words at the most, provided you don't generate any output or do any parsing.

Think of both these areas as predefined strings. You can refer to them as 'TIB' and 'PAD'. You don't have to declare them in any way. This program is perfectly alright:

```
s" Hello world" pad place      \ store a string in pad
pad count type cr              \ print contents of the pad
```

If you want to know how big TIB and PAD are, you can use the predefined constants '/TIB' and '/PAD':

```
." Size of TIB: " /TIB . cr    \ print sizeof TIB
." Size of PAD: " /PAD . cr    \ print sizeof PAD
```

Note, this does *not* print the length of a string stored in the area, but the maximum size of the string that can be stored there. Some space of the PAD is reserved for number generation (see section 8.3). You can get the size of this area by the predefined constant '/HOLD'. This will print the size of this area and the size of PADs circular buffer:

```
." Size of HOLD  : " /HOLD . cr \ print sizeof HOLD
." Size of buffer: " /PAD /HOLD - . cr
```

If that area did not exist even printing a number could corrupt the circular buffer. In some unusual circumstances, the PAD can get corrupted. If so, identify the temporary string that gets corrupted and store it explitly into a string variable.

8.5 Simple parsing

We have already discussed 'REFILL' a bit. We've seen that it is closely related to 'ACCEPT'. 'REFILL' returns a true flag if all is well. When you use the keyboard it usually is, so we can safely drop it, but we will encounter a situation where this flag comes in handy.

If you want to get a string from the keyboard, you only have to type:

```
refill drop                    \ get string from keyboard
```

Every next call to 'REFILL' will overwrite any previously entered string. So if you want to do something with that string you've got to get it out of there, usually to one of your own strings.

But if accessing the TIB directly is not the proper way, what is? The use of 'REFILL' is closely linked to the word 'PARSE-WORD', which is a parser. 'PARSE-WORD' looks for the delimiter, whose ASCII code is on the stack.

If the string starts with the delimiter, it will skip this and all subsequent occurrences until it finds a string. Then it will look for the delimiter again and slice the string right there. It then returns its address and count.

This extremely handy when you want to obtain filtered input. E.g. when you want to split somebodies name into first name, initials and lastname:

```
Hans L. Bezemer
```

Just use this program:

```
." Give first name, initials, lastname: "
refill drop                    \ get string from keyboard
bl parse-word                  \ parse first name
." First name: "               \ write message
type cr                        \ type first name
bl parse-word                  \ parse initials
." Initials  : "               \ write message
type cr                        \ type initials
bl parse-word                  \ parse last name
." Last name : "               \ write message
type cr                        \ write last name
```

You don't have to parse the entire string with the same character. This program will split up an MS-DOS filename into its components:

```
." DOS filename: " refill      \ input a DOS filename
drop cr                        \ get rid of the flag

char : parse-word              \ parse drive
." Drive: " type ." :" cr
                               \ print drive
begin
  char \ parse-word            \ parse path
  dup 0<>                      \ if not a NULL string
while                          \ print path
  ." Path : " type cr
repeat                         \ parse again
drop drop                      \ discard string
```

If 'PARSE-WORD' reaches the end of the string and the delimiter is still not found, it returns the remainder of that string. If you try to parse beyond the end of the string, it returns a NULL string. That is an empty string or, in other words, a string with length zero.

Therefore, we checked whether the string had zero length. If it had, we had reached the end of the string and further parsing was deemed useless.

8.6 Converting a string to a number

We now learned how to parse strings and retrieve components from them. But what if these components are numbers? Well, there is a way in 4tH to convert a string to a number, but like every number-conversion routine it has to act on invalid strings. That is, strings that cannot be converted to a valid number.

4tH uses an internal error-value, called '(ERROR)'. The constant '(ERROR)' is a strange number. You can't negate it, you can't subtract any number from it and you can't print it. If 4tHs number-conversion word 'NUMBER' can't convert a string it returns that constant. Let's take a look at this program:

```
." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
number dup                     \ convert to a number
(error) =                      \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  ." The number was: " . cr
then
```

You first enter a string, then it is parsed and 'PARSE-WORD' returns the address and count. 'NUMBER' tries to convert it. If 'NUMBER' returns '(ERROR)' it wasn't a valid string. Otherwise, the number is right on the stack, waiting to be printed. That wasn't so hard, was it?

8.7 Controlling the radix

If you are a programmer, you know how important this subject is to you. Sometimes, you want to print numbers in octal, binary or hex. 4tH can do that too. Let's take the previous program and alter it a bit:

```
." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
number dup                     \ convert to a number
(error) =                      \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  hex
  ." The number was: " . cr
then
```

We added the word 'HEX' just before printing the number. Now the number will be printed in hexadecimal. 4tH has a number of words that can change the radix, like 'DECIMAL' and 'OCTAL'. They work in the same way as 'HEX'.

4tH always starts in decimal. After that you are responsible. Note that all radix control follows the flow of the program. If you call a self-defined word that alters the radix all subsequent conversion is done too in that radix:

```
: .hex hex . ;                 \ print a number in hex

." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
number dup                     \ convert to a number
(error) =                      \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  ." The number was: " .hex cr
then
```

In this example not only that single number is printed in hex, but also all subsequent numbers will be printed in hex! A better version of the ".HEX" definition would be:

```
: .hex hex . decimal ;
```

Since that one resets the radix back to decimal. Words like 'HEX' do not only control the output of a number, but the input of numbers is also affected:

```
." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
hex                            \ convert hexadecimal
number dup                     \ convert to a number
(error) =                      \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  dup
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'NUMBER' will now also accept hexadecimal numbers. If the number is not a valid hexadecimal number, it will return '(ERROR)'. You probably know there is more to radix control than 'OCTAL', 'HEX' and 'DECIMAL'. No, we have not forgotten them. In fact, you can choose any radix between 2 and 36. This slightly modified program will only accept binary numbers:

```
: binary 2 base ! ;

." Enter a number: "           \ write prompt
```

```
refill drop                      \ enter string
bl parse-word                    \ parse string
binary                           \ convert hexadecimal
number dup                       \ convert to a number
(error) =                        \ test for valid number
if                               \ if not valid
  ." You didn't enter a valid number!" drop cr
else                             \ print if valid
  dup                            \ both decimal and hex
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'BASE' is a predefined variable that enables you to select any radix between 2 and 36. This makes 4tH very flexible. However, this won't work:

```
hex 02B decimal . cr
```

4tH will try to compile "02B", but since it isn't a word or a valid decimal number, it will fail. Words like 'HEX' and the 'BASE' variable work only at run-time, not at compile-time! Isn't there a way to compile non-decimal numbers?

Sure, there is, although it is not that flexible. There are four words that control the interpretation of numbers at compile-time:

1. [BINARY]
2. [OCTAL]
3. [DECIMAL]
4. [HEX]

They work fundamentally different than their run-time equivalents. First, they only work at compile-time. Second, they are interpreted sequentially and do not follow the flow of the program. Let's take a look at these two programs:

```
[binary] 101 . cr
[octal] 101 . cr
[decimal] 101 . cr
[hex] 101 . cr
```

This will print the decimal numbers "5", "65", "101" and "257", since each one of them is compiled with a specific radix.

```
: binary 2 base ! ;
binary 101 . cr
octal 101 . cr
decimal 101 . cr
hex 101 . cr
```

Now the decimal number "101" is printed in four different radixes, since at compile-time the radix was set to decimal (which is the default). Now take a look at this program:

```
: do_binary [binary] ;
: do_decimal [decimal] ;
do_binary 101 decimal . cr
do_decimal 101 decimal . cr
```

The program will print "101" two times! Haven't we selected binary at compile-time? No, both '[BINARY]' and '[DECIMAL]' are interpreted sequentially!

When '[BINARY]' is encountered at the first time, it will set the radix at compile-time to binary. When '[DECIMAL]' is encountered in the second line, it will set the radix to decimal. When the third line it compiled, the radix is still set to decimal. If you want to make this program work, try this:

```
[binary]
101 decimal . cr
[decimal]
101 decimal . cr
```

When the first line is encountered, it sets the radix (at compile-time) to binary. So the number "101" at line two is compiled as a binary number. 'DECIMAL' will just be compiled. It will only influence the radix at run-time. The third line sets the radix at compile-time to decimal. So the number "101" at line four is compiled as a decimal number.

Since the run-time of 4tH starts up in decimal, both occurrences of 'DECIMAL' have little value. We can even eliminate 'DECIMAL' from the program altogether without affecting the result:

```
[binary] 101 . cr
[decimal] 101 . cr
```

Note that both the compile-time radix control words and the run-time radix control words stay in effect until they are superseded by others:

```
[binary]                        \ compile-time binary
101                             \ first binary number
1011                            \ second binary number
[decimal]                       \ compile-time decimal
5                               \ decimal 5
do                              \ set run-time radix
  i base !                      \ to loop-index
  dup . cr                      \ print number
loop
drop                            \ clean stack
```

8.8 Pictured numeric output

You probably have used this before, like when writing Basic. Never heard of "PRINT USING.."? Well, it is a way to print numbers in a certain format. Like telephone-numbers, time, dates, etc. Of course 4tH can do this too. In fact, you've probably used it before. Both '.' and '.R' use the same internal routines. They are called just before a number is printed.

This numeric string is created in the PAD and overwritten with each new call. But we'll go into that a bit later on.

What you have to remember is that you define the format reverse. What is printed first, is defined last in the format. So if you want to print:

```
060-5556916
```

You have to define it this way:

```
6196555-060
```

Formatting begins with the word '<#' and ends with the word '#>'. A single number is printed using '#' and the remainder of the number is printed using '#s' (which is always at least one digit). Let's go a bit further into that:

```
: print# <# #s #> type cr ;
256 print#
```

This simply prints a single number (since only '#S' is between the '<#' and the '#>' and goes to a new line. There is hardly any difference with '.'. You can try any (positive) number. Note that the values that '#>' leaves on the stack can directly be used by 'TYPE'.

This is a slightly different format:

```
: print3# <# # # # #> type cr ;
256 print3#
1 print3#
1000 print3#
```

This one will print "256", "001" and "000". Always the last three positions. The '#' simply stands for 'print a single digit'. So if you want to print a number with at least three digits, the format would be:

```
#s # #
```

That is: print the remainder of the number (at least one digit) and then two more. Now reverse it:

```
# # #s
```

Enclose it by '<#' and '#>' and add 'TYPE CR':

```
<# # # #s #> type cr
```

And that's it! Is it? Not quite. So far we've only printed *positive* numbers. If you try a negative number, you will find it prints garbage. This behavior can be fixed with the word 'SIGN'.

'SIGN' simply takes the number from the stack and prints a "-" when it is negative. The problem is that all other formatting words can only handle positive numbers. So we need the same number twice. One with the sign and one without. A typical signed number formatting word looks like:

```
: signed# dup abs <# #s sign #> type ;
```

Note the 'DUP ABS' sequence. First the number is duplicated (for 'SIGN') and then the absolute value is taken (for the other formatting words). So we got the number on the stack twice. First with sign (for 'SIGN'), second without sign (for the other formatting words). Does that make sense to you?

We can place 'SIGN' wherever we want. If we want to place the sign after the number (like some accountants do) we would write:

```
: account# dup abs <# sign #s #> type ;
```

But that is still not enough to write "$2000.15" is it? Well, in order to do that there is another very handy word called 'HOLD'. The word 'HOLD' just copies any character into the formatted number. Let's give it a try:

```
$2000.16
```

Let's reverse that:

```
61.0002$
```

So we first want to print two numbers, even when they are zero:

```
# # .0002$
```

Then we want to print a dot. This is where 'HOLD' comes in. 'HOLD' takes an ASCII code and places the equivalent character in the formatting string. We don't have to look up the ASCII code for a dot of course. We can use 'CHAR':

```
# # char . hold 0002$
```

Then we want to print the rest of the number (which is at least one digit):

```
# # char . hold #s $
```

Finally we want to print the character "$". Another job for 'HOLD':

```
# # char . hold #s char $ hold
```

So this is our formatting word:

```
: currency <# # # char . hold #s char $ hold #> type cr ;
```

And we call it like this:

```
200016 currency
```

You can do some pretty complex stuff with these formatting words. Try to figure out this one from the master himself, Leo Brodie:

```
: sextal 6 base ! ;
: :00 # sextal # decimal 58 hold ;
: time# <# :00 :00 #S #> type cr ;
3615 time#
```

Yeah, it prints the time! Pretty neat, huh? Now try the telephone-number we discussed in the beginning. That shouldn't be too hard.

8.9 Converting a number to a string

Since there is no special word in 4tH which will convert a number to a string, we'll have to create it ourselves. In the previous section we have seen how a numeric string is created in the PAD. We can use this to create a word that converts a number to a string.

Because the PAD is highly volatile, we have to save the string immediately after its creation. So we'll create a word that not only creates the string, but places it directly in its proper location:

```
: >string >r dup abs <# #s sign #> r> place ;
( n a -- )
```

It takes a number, the address of a string and returns nothing. Example:

```
16 string num$
-1024 num$ >string
num$ count type cr
```

8.10 Aborting a program

Some conditions are so grave you can consider them to be fatal errors. In such cases the only thing you can do is abort the program as soon as possible. Of course, there is a way in 4tH to do just that. You can use either 'ABORT' or 'QUIT'. Same thing. Both will terminate your program immediately. This small program prints nothing:

```
abort
." This will never be printed." cr
```

But there is more. Let's say you only want to exit a program when a certain condition is met, e.g. a word left a non-zero value on the stack. In that case you would have to write something like this:

```
if
    ." We have an error condition!" cr quit
then
```

You can write that much shorter by using the word 'ABORT"':

```
abort" We have an error condition!"
```

'ABORT"' will print the message following it and abort, but *only* when there is a non-zero value on the stack. So this program does *not* abort:

```
false abort" This will not be printed!"
." This will be printed!"
```

You will find that 'ABORT"' is a very handy tool when processing error conditions.

8.11 Opening a file

You probably don't want to write programs that only write to the screen and read from the keyboard. So 4tH has a few words that allow you to work with files. Since 4tH is a scripting language, its capabilities are limited. But you will find that you can perform most common operations.

One of the limitations is that you can have a limited number of open files, but it will do in most situations.

Opening a output-file is pretty simple. Just throw the address and length of a filename and a read/write-mode flag on the stack and execute the word 'OPEN'. The value 'OPEN' returns is a simple number which bears little significance. However, you have to save it to a variable or value, for you will need it later. We'd like to use values for storing file pointers, so we created the word 'FILE'. 'FILE' simply creates a value and initializes it, so if you use it prematurely 4tH will issue an error message.

```
file myfile
s" outfile.dat" output open dup as myfile   ( a1 n1 f1 -- f2)
0= abort" File could not be opened"
```

'OUTPUT' is a write-flag and will open a file for writing. 'OPEN' leaves a value on the stack. If it is a non-zero value, the file was successfully opened. If not, it wasn't. 'AS' is simply an alias for 'TO', since 'FILE' is nothing but an initialized value. You may use 'TO' if you want to, it's just a little syntactic icing on the cake.

The syntax for opening an input file is the same, except for the read-flag 'INPUT' of course:

```
file myotherfile
s" infile.dat" input open dup as myotherfile
0= abort" File could not be opened"
```

A second input-file closes the first one, just like an output-file.

8.12 Reading and writing from/to a file

There are no special words to read from or write to a file. You can use all the words you used for keyboard-input and screen-output.

But if you open a file and do some I/O you will notice nothing has changed. Of course not. You should be able to determine whether you write to a file or to the screen. There are special words to do just that:

```
file OutFile                  \ file variable
s" outfile.dat" output open dup as OutFile
0= abort" File could not be opened"
                              \ open the file
OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
```

After you've opened the file, the program will still write to the screen. When 'USE' executes, all output will be redirected to the file. When 'USE' executes again, but this time with the 'OUTPUT' flag, all output will go to the screen again, but the output-file will not be closed! Both words take the same read/write-flags as 'OPEN'.

You can call 'USE' again and again, without closing or opening any files. Here is an example using an input-file:

```
file OutFile
s" outfile.dat" output open dup as OutFile
0= abort" File could not be opened"
                              \ open output file
OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
OutFile close                 \ close file

s" outfile.dat" input open dup as OutFile
0= abort" File could not be opened"

OutFile use                   \ read from disk
pad dup 32 accept             \ read 32 characters
type                          \ write string to screen
stdin use                     \ read from keyboard
OutFile close                 \ close file
```

The output of this program is:

```
This is written to screen
This is written to disk
```

Microsoft-users, note that files are opened in binary mode (no CR/LF translations). If you issue 'CR' the line will be terminated by a linefeed. Don't worry. You can fix that as we will see later on.

8.13 Closing a file

There is usually no need to close any files. When you quit the program all files are closed. It seems like there is no need at all to close files manually, but that is a mistake.

If you want to open a file for reading to which you've just written, you will find it doesn't work. Of course, you can open a file only once.

No, there is a word which closes either the input- or the output-file, using the same read/write-flags. You've already seen it, it is called 'CLOSE'. When you close an active file, the input (or output) is redirected to the keyboard (or screen).

8.14 Writing text-files

Writing text to a file is just as easy as writing text to screen. Open the file, redirect the output, and write like you would write to the screen:

```
file OutFile                  \ value for file
s" outfile.dat"               \ put the filename on the stack
output                        \ add the modifier
open dup as OutFile           \ open the file
0= abort" File could not be opened"

OutFile use                   \ write to file
." This is written to disk" cr
```

That's all! Note that if you execute your program on a Microsoft Operating System, it will write a Microsoft text file. If you do so on a Unix Operating System, it will write a Unix text file. If you want to override that you'll have to issue the end-of-line sequences yourself using 'EMIT'.

8.15 Reading text-files

Reading text-files is pretty straightforward. You don't even have to open a file in text-mode contrary to other languages. Just open the file and call 'REFILL' until it signals end-of-file (EOF):

```
\ Example program. It reads a file line by line
\ and prints it to the screen.

file InFile
s" readln.4th" input open dup as Infile
0= abort" Could not open file" \ open file
InFile use                     \ read from file

begin
  refill                       \ read a line
while                          \ while EOF not found
  0 parse-word                 \ parse the entire line
  type                         \ print it
  cr                           \ terminate line
repeat                         \ read next line
```

You will find that if you run this program, it will print itself to the screen.

'REFILL' will return a non-zero value if EOF was *not* detected. By using the word '0=' you can invert this value. Finally, it will read Unix ASCII-files as well as DOS ASCII-files, no matter where your program is executed.

8.16 Reading long lines

The TIB is only /TIB characters long. If you read a line that is longer than that, only /TIB - 1 characters are read. The rest of the line is read when you invoke 'REFILL' again. Although you don't lose any information that way, it might not be what you want. Fortunately, you can define your own TIB:

```
2048 constant /mytib            \ length of your TIB
/mytib string mytib             \ define your own TIB

mytib /mytib source!            \ tell the system about your TIB
```

The next time you invoke 'REFILL', it will use your TIB instead of the system TIB, so it will now read lines up to 2047 characters. 'SOURCE!' takes an address/count pair and makes it the current TIB. So if you want to use the system TIB again you issue:

```
tib /tib source!
```

And if you have forgotten which TIB you're using try this:

```
source . . cr
```

'SOURCE' will return the address/count pair of the TIB you're currently using. In fact, this definition does absolutely *nothing*:

```
: doesnothing source source! ;
```

For the simple reason that it reassigns the TIB it is already using.

8.17 Reading binary files

If you process binary files, you won't get far reading it line by line. You want to read chunks of data. 4tH can do that too by using 'ACCEPT'. You feel there must be a catch, since 'ACCEPT' terminates strings automatically. Well, there isn't. When 'ACCEPT' does not read from the keyboard, it won't add that extra byte.

Reading blocks of data usually means defining buffers. If maintainability is an issue, define a constant for the sizes of these buffers. You cannot only use this constants when defining buffers, but also when calling 'ACCEPT'.

Furthermore, 'ACCEPT' returns the number of characters actually read. If this value is compared to the number of characters we actually wanted to read, we can determine whether a reading error or EOF occurred:

```
1024 constant bufsize          \ actual buffersize
bufsize string buffer          \ define buffer
file InFile                    \ value for file
                               \ open input file
s" infile.dat" input open dup as InFile
0= abort" File could not be opened"

InFile use                     \ redirect input

begin                          \ using bufsize
  bufsize                      ( n1)
  buffer over                  ( n1 a n1)
  accept                       ( n1 n2)
  <>                           ( f)  \ make EOF flag
until                          \ until EOF
```

Note that "BUFFER" is actually not a string, but a chunk of memory. But since a character in 4tH takes up a single address-unit (=byte), raw chunks of memory are allocated in the Character Segment. This is not an uncommon practice in both Forth and C.

8.18 Writing binary files

Writing binary files is very easy. Of course you need a buffer, like we discussed in the previous section. The program is not much different than the previous one:

```
1024 constant bufsize         \ actual buffersize
bufsize string buffer         \ define buffer
file OutFile                  \ value for file

buffer bufsize char H fill    \ fill the buffer
                              \ open output file
s" infile.dat" output open dup as Outfile
0= abort" File could not be opened"

OutFile use                   \ redirect input
buffer bufsize type           \ write to file
```

This will write 1024 "H"s to "infile.dat". The actual command that does all writing is 'TYPE'. The word 'TYPE' does not return anything. You can be assured that everything was alright, since if it wasn't, 4tH would have caught the error itself.

8.19 Reading and writing block files

Block files are a special kind of files used by Forth compilers. In the old days Forth controlled the *entire* computer and directly communicated with all peripherals, including disks. To Forth, a disk is just a bunch of numbered blocks. Each block is divided into 16 lines of *exactly* 64 characters. A block file simply mimics that layout.

Before we can begin, you need to create a block file. Well, that's easy, an empty file will do:

```
s" blocks.scr" output open close
```

Then we have to load the ANS Forth wordset and tell 4tH which file to use:

```
include lib/ansblock.4th
s" blocks.scr" use-block
```

Note that apart from creating the file, we haven't performed any I/O yet. First, we have to request a block. When a block is requested, its contents are transferred to a memory buffer. You can manipulate this buffer any way you want with the standard words. If you request another block its contents are transferred to the buffer too, overwriting whatever is there. All changes you have made are lost, *unless* you have flagged the block as dirty, which means its contents are different from the block on disk. If a block is dirty, it is written to disk *before* the next block is read. 'CLEAR' is a special word, assigning an empty buffer to a block without reading it first. The buffer is BLANKed. 'UPDATE' will flag the buffer as dirty. 'FLUSH' writes the dirty buffer to disk and unassigns the buffer. So, first we clear block 0 and write it to disk:

```
0 clear update flush
```

Then we clear block 1, copy a string to it and flag it as dirty:

```
1 clear
s" Hello world!" >r 1 block r@ cmove update
```

'BLOCK' returns the address of the buffer assigned to that block[1]. If the block is not present, it is read into the buffer. If the buffer is dirty, it is FLUSHed first. We can also write the dirty buffer to disk, without unassigning it:

```
save-buffers
```

Note that the buffer is not dirty anymore, since it has been synchronized. Let's write something to another block:

```
s" Goodbye cruel world!" 0 block swap cmove update
```

It is always a nice game to figure out what will happen now. The current block is block 1. Since we haven't UPDATEd it since 'SAVE-BUFFERS', it is clean. That means that 4tH won't perform a write. Since block 0 isn't current, it is read into the buffer. The 'UPDATE' will flag the buffer as dirty.

```
1 block r> type cr
```

This is fun! The current block is block 0. It is dirty, so it is written to disk. Since block 1 isn't current, it is read into the buffer. You catch my drift? If you want to print the contents of a block, you can use 'LIST'. Of course, 'LIST' uses 'BLOCK' and applies to the same rules:

```
0 list
." This block has been listed: " scr ? cr
```

'SCR' is a variable containing the last screen LISTed. Note that is not the same thing as the current block! Finally, we can discard all our changes:

```
empty-buffers
```

'EMPTY-BUFFERS' does not perform any I/O nor does it change the contents of the buffer. It just unassigns the buffer and flags it as clean. Note that you don't have to close a block file since all I/O is block-oriented. You can use different block files within the same program, but you'll lose the changes in any dirty buffers.

8.20 Parsing textfiles

As we've already seen, it is very easy to enter a line using 'REFILL' and parse it. You can also use 'REFILL' to read lines from a text-file. It is quite similar to reading lines from the keyboard, except that you have to open a file first. This little program prints all the words of a textfile on a new line:

[1]That is particularly handy if your implementation can handle multiple buffers. In this implementation we have only one buffer, so we always return the same address.

```
file InFile                   \ value for file
s" file.txt" input open dup as InFile
0= abort" File could not be opened"
                              \ open the file
InFile use                    \ redirect input to file

begin
  refill                      \ get a line from file
while                         \ check if EOF
  begin
    bl parse-word             \ if not, parse line
    dup 0<>                   \ check if zero length
  while
    type cr                   \ if not, print word
  repeat                      \ parse next word
  drop drop                   \ drop address/count
repeat                        \ get next line
```

Now that flag left by 'REFILL' makes sense! If it is zero, we have reached the end of the line. Note that you don't have to open a file in text-mode and both Microsoft ASCII and Unix ASCII files are supported.

8.21 Parsing binary files

And what about binary files, like classic Forth blockfiles? Well, you could use 'REFILL' in that context too, but it would probably break up words since it can't find an end-of-line marker and its buffer is smaller than 1024 characters. Does that mean it can't be done? No! But 'REFILL' makes it easier for you, because it handles a few tasks automatically.

First, it has its own buffer (TIB). When you're not using 'REFILL' you have to define one yourself. Second, it terminates the string for you. You don't want 'PARSE-WORD' to wander into new territory, do you? Third, it sets '>IN' for you every time its receives new input. You have to take care of that one too.

Never heard of '>IN'? Well, the only way for 'PARSE-WORD' to know on what position the previous scan ended is to store that information into a variable. This variable is called '>IN'.

Not all internal 4tH variables are accessible, mostly because we can't imagine what use they could have to you. Some variables are just better left alone. But '>IN' is available for some very obvious reason: you can reset it and make 'PARSE-WORD' work for you. Note that for '>IN' to work, you have to make the buffer the parsing area by using 'SOURCE!'

The following program will read the first screen of a block-file for you and print out all the words. You will see that all spaces are eliminated and every word is printed on a new line, just the behavior you would expect from 'PARSE-WORD'.

```
1025 constant /buffer         \ screensize + terminator
/buffer -1 [+] constant c/scr \ size of the block
file InFile                   \ value for file

/buffer string buffer         \ 1: our own buffer

: openfile                    \ open the block file
```

```
  s" romans.blk" input open dup as InFile
  0= abort" Cannot open file"
  InFile use                  \ read from file
;

: readfile                    \ fill the buffer
  buffer c/scr 2dup           \ address and count
  bl fill                     \ clear the buffer
  accept drop                 \ fill the buffer
  input close                 \ close the file
;

: initparse                   \ configures parsing
  0 buffer c/scr chars + c!   \ 2: terminate screen
  buffer /buffer source!      \ 3: make buffer the parse area
  0 >in !                     \ 4: reset >IN
;

: parseblock
  begin
    bl parse-word             \ get word
    dup 0<>                   \ length zero?
  while
    type cr                   \ if so, print it
  repeat
  2drop                       \ else drop addr/cnt
  ." End of block" cr         \ signal "End of block"
;

: parsefile                   \ do it all
  openfile                    \ open the file
  readfile                    \ read it
  initparse                   \ set up parsing
  parseblock                  \ parse it
;

parsefile
```

Note there is no need to reset '>IN' if you use 'REFILL', since it will be reset *automatically*. In this case, if you want to parse another block, you will have to reset '>IN' again.

8.22 Parsing comma-delimited files

'PARSE-WORD' is a powerful and very useful word, but it is less than useful when parsing comma-delimited files. Why? Well, because 'PARSE-WORD' skips leading delimiters. So when you have a file like this it doesn't work:

```
FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX
Hans,Bezemer,hansoft@bigfoot.com„http://hansoft.come.to,
```

Again, 'PARSE-WORD' skips leading delimiters, so instead of an empty string we get the homepage when we're trying to read the (non-existant) telephone number. Fortunately, we got a word like 'PARSE'. 'PARSE' also takes a delimiter from the stack, just like 'PARSE-WORD', but it acts on leading delimiters. Take a look at this program:

```
file OutFile                  \ value for output file
file InFile                   \ value for input file

: WriteCommaFile              ( --)
  s" address.csv" output open dup as OutFile
  0= abort" Could not write CSV file"
  OutFile use                 \ redirect output to file
  ." FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX" cr
  ." Hans,Bezemer„,http://hansoft.come.to," cr
  OutFile close               \ close file
  stdout use                  \ redirect output to screen
;

: ProcessLine                 ( --)
  refill                      \ get line
  0= abort" Read error"
  [char] , parse type cr      \ parse first name
  [char] , parse type cr      \ parse name
  [char] , parse type cr      \ parse email
  [char] , parse type cr      \ parse telephone
  [char] , parse type cr      \ parse homepage
  [char] , parse type cr cr   \ parse fax
;

: ReadCommaFile               ( --)
  s" address.csv" input open dup as InFile
  0= abort" Could not read CSV file"
  InFile use                  \ redirect input to file
  ." _Headerline_" cr         \ this is the headerline
  ProcessLine                 \ now process headerline
  ." _First record_" cr       \ this is the first record
  ProcessLine                 \ now process first record
  InFile close                \ close file
;

WriteCommaFile                \ write the CSV file
ReadCommaFile                 \ read the CSV file
```

With "WriteCommaFile" we write a simple comma-delimited file to disk. We got to read something, don't we? Then we read the file we've just written with "ReadCommaFile". "ProcessLine" does the actual job. Since we have six fields we use 'PARSE' six times. We cannot do this with a loop. Why not? 'PARSE-WORD' can do it that way.

Well, 'PARSE' not only returns a NULL-string when we've reached the end of a line, but also when a field is empty. So we've got to know how many fields we actually want to read. Of course, you could parse the headerline with 'PARSE-WORD' to find that out, but you already know how to do this.

8.23 Advanced parsing

Let's think of something difficult to parse, e.g.:

```
;;;;;;;;.........Can you parse me.
;;;;,,,,And me too, huh?
```

If we would use 'PARSE' we would have to know how many semicolons to skip and there is a different number of them on each line. If we would use 'PARSE-WORD' we'd lose all the semicolons, but the parsed string would have all these nasty leading dots.

Even worse, if we were able to skip the semicolons and use 'PARSE-WORD' with the leading delimiter we'd get "Can you parse me" and "And me too" instead of "Can you parse me." and "And me too, huh?". What can we do?

Fortunately, 4tH doesn't really know about 'PARSE-WORD' but translates it into a sequence of words[2]. We can also use them directly. 'OMIT' is very handy. It doesn't actually do anything, it just skips leading delimiters and sets '>IN' accordingly. It takes an ASCII value from the stack as its delimiter. This will correctly parse the first line:

```
char ; omit                   \ omit the semicolon
char . omit                   \ omit the dot
0 parse                       \ parse the remainder of the line
```

This will correctly parse the second line:

```
char ; omit                   \ omit the semicolon
char , omit                   \ omit the comma
0 parse                       \ parse the remainder of the string
```

Please note that this are special 4tH words! Unfortunately you cannot port this to ANS-Forth, where only a limited version of 'PARSE-WORD' and 'PARSE' are available.

8.24 Appending to existing files

You can use a so-called modifier to signal 4tH, it shouldn't overwrite the file it opens, but append to it:

```
file OutFile                  \ value for output file

s" outfile.dat" output open dup as OutFile
0= abort" Cannot open file"   \ open the file
OutFile use                   \ now write to disk
10 0 do i . loop              \ write 0 to 9
OutFile close                 \ close the file

s" outfile.dat" output append + open dup as OutFile
0= abort" Cannot open file"   \ reopen in append mode
OutFile use                   \ now write to disk
20 10 do i . loop             \ write 10 to 19
OutFile close                 \ close the file
```

Take a look at the contents of this file after you've run the program and you'll find it contains the number 0 to 19.

[2]If you're really curious, 'PARSE-WORD' is equivalent to 'DUP OMIT PARSE'.

8.25 Using pipes

If you're using Windows 95 OSR2 (and up), Linux or another Unix system you're in for a treat! With Unix you can do neat tricks like this:

```
ls | mail root
```

Which means you can redirect the output of 'ls' to 'mail', so in effect you send an email to root with the contents of your current working directory. Yes, 4tH can do this too, but you can do even more. You can start 'ls' and read its output line by line as if it were a file. You can also start 'mail' and write the output of a 4tH program to it. We do that by opening a *pipe* to a program.

If you've ever written a program using C, you know this is a bit cumbersome, since you've got to use special functions to use pipes. In 4tH you don't. Just let 4tH know it's a pipe that you're opening and not a file:

```
file InFile                    \ value for input file
file OutFile                   \ value for output file

s" ls" input pipe + open dup as InFile 0<>
s" mail hans" output pipe + open dup as OutFile 0<> and
0= abort" Cannot open pipe"
```

The only thing you have to do to signal 4tH that you're using a pipe is add the word 'PIPE', just like 'APPEND'. The filename is replaced by the command you want to execute. That's all. If one of the pipes in this program fails, the program aborts.

```
InFile use OutFile use
." These are the contents of my current working directory:"
cr cr
```

Now we can treat our pipes just as if they are ordinary files. We redirect input and output and write a nice header to our email. Now we can start to process the output of 'ls':

```
begin
  refill
while
  0 parse-word type cr
repeat
```

Note that we don't have to signal that we're reading the pipe to 'ls' as a text file. We just read it line by line until 'REFILL' returns zero. Then we can parse the line and 'TYPE' it to 'mail'.

```
InFile close OutFile close
```

Of course you don't *have to* close the pipe, but it won't harm when you don't. 4tH knows what to do. After executing this program, Hans will receive this email:

```
From: Hans Bezemer <hansoft@bigfoot.com>
Message-Id: <200202252017.VAA00712@bigfoot.com>
To: hans@localhost.org
Status: RO

These are the contents of my current working directory:

4th.c
4thc.c
4thd.c
4thg.c
4thx.c
```

Well, that wasn't too hard, was it?

8.26 Opening a file in read/write mode

For special purposes you might want to open a file in read/write mode. That's quite easy:

```
file InOutFile                \ value for output file

s" outfile.dat" output input + open dup as InOutFile
0= abort" Cannot open file"   \ open the file
```

Just add both together like you adding a modifier. Note that once you 'USE' this file, you're both reading *and* writing to this file. Furthermore, the file has to exist otherwise you get an error. If you want to write to a new file, you first have to open it in write mode:

```
file InOutFile                \ value for output file
                              \ create a new file
s" outfile.dat" output open close
s" outfile.dat" output input + open dup as InOutFile
0= abort" Cannot open file"   \ open the file
```

8.27 Using random access files

Upto now we've always accessed a file sequentially, but it is also possible to use random access files. Two words are crucial here, 'SEEK' and 'TELL'. 'SEEK' will seek for the desired file position and 'TELL' will tell you that you're there. It is as simple as that..!

Let's take a look at this example. We've got a block-file called "Messages.scr" with the following contents:

```
Scr # 0
  0 (0) No errors
  1 (1) Out of memory
  2 (2) Bad object
  3 (3) Stack overflow
  4 (4) Stack empty
  5 (5) Return stack overflow
```

```
 6 (6) Return stack empty
 7 (7) Bad string
 8 (8) Bad variable
 9 (9) Bad address
10 (A) Divide by zero
11 (B) Bad token
12 (C) Bad radix
13 (D) Undefined name
14 (E) I/O error
15 (F) Assertion failed
```

First, let's define a word that reads a message and then displays it:

```
: next-msg pad dup 64 accept -trailing type cr ;
```

Since this is a simple program we can safely use the 'PAD' to store our messages. Every message has the length of a standard block-file line, which is 64 characters. Trailing spaces are stripped by '-TRAILING'. Now we need a word that tells us what our file position is:

```
: tell-msg cr ." Current position: " dup tell . cr ;
```

'TELL' needs a file pointer and leaves the current position of that file pointer on the stack. This word assumes that the top of the stack contains a valid file pointer. Finally we need a word that sets the file position:

```
: seek-msg over seek abort" Seek failed" ;
```

'SEEK' needs a file position and a file pointer. If it returns false, it was successful; if it returns true there was an error. This word assumes that the top of the stack contains a valid file pointer. We're ready now, let's play. First we open the file and use it:

```
s" Messages.scr" input open dup use
```

This leaves a file pointer on the top of the stack, assuming everything went OK. Now let's read some messages:

```
next-msg next-msg next-msg tell-msg
```

You'll see these messages appear on the screen:

```
(0) No errors
(1) Out of memory
(2) Bad object

Current position: 192
```

After reading three messages we've obviously reached position 192 in the file. That makes sense, since 3 lines of 64 characters makes 192 characters in total. Let's see what 'SEEK' does:

```
0 seek-msg tell-msg next-msg
```

This should take us back to the very beginning of the file, as if we've freshly opened it. And yes, it does:

```
Current position: 0
(0) No errors
```

After executing 'SEEK', 'TELL' confirms that we've actually returned to the very beginning of the file. Reading the next message reconfirms that again. When you feed 'SEEK' positive values, it always starts seeking from the *beginning* of the file. When you feed 'SEEK' negative values, it seeks from the *end* of the file. So this one takes you to the last line:

```
-64 seek-msg tell-msg next-msg
```

On screen it looks like this:

```
Current position: 960
(F) Assertion failed
```

Finally, we clean up the mess we made:

```
close
```

This will consume the file pointer we left on the stack and close the file. Note that 'SEEK' and 'TELL' come with a few restrictions. Pipes are out of the question and so are the standard streams 'STDIN' and 'STDOUT'. Apart from that you can pretty much do with them what you want.

8.28 The layout of the I/O system

You're probably quite confident manipulating files now, so I guess it is time to offer you a view under the hood. 4tH has two channels, an input channel and an output channel. All words read from the input channel or write to the output channel. At startup, the input *channel* is connected a *stream* that reads from the keyboard ('STDIN') and the output *channel* is connected to a *stream* that writes to the screen ('STDOUT').

With 'OPEN' you can open *additional* streams, which are connected to a file or a pipe. The return value of 'OPEN' points to the stream that was opened. There are few words that directly handle streams, 'USE', 'CLOSE', 'TELL' and 'SEEK' being the exceptions. 'USE' attaches a stream to one or both channels, which results in redirecting all in- and/or output to that stream. E.g. if a file is opened in read/write mode using 'OPEN', a stream is returned. If we 'USE' that stream, both the input and the output channel are connected to that stream. If it had only been opened in read mode, only the input channel would have been connected to the stream.

'CLOSE' closes a stream, even if it is still attached to a channel. If that is the case, the appropriate default streams ('STDIN', 'STDOUT') are reattached. We can find out which streams are currently used by 'CIN' and 'COUT'. 'CIN' returns the stream that is currently attached to the input channel, 'COUT' returns the stream that is currently attached to the output channel.

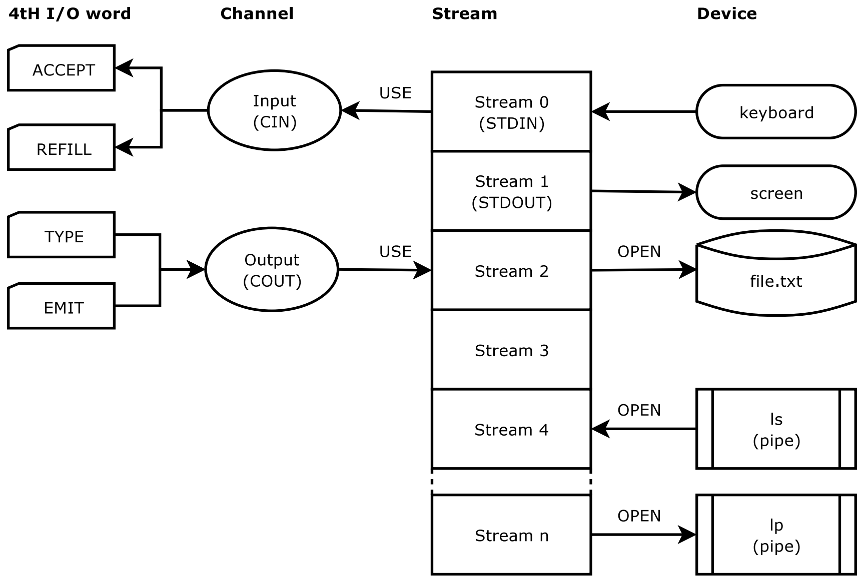

Figure 8.2: The 4tH I/O system

8.29 Speech synthesis

If you're using Unix[3], you're in for a treat. 4tH can talk! All you need is the "Festival" speech synthesis package[4] and a small 4tH interface. If you want to imitate old Arnold, this will do:

```
include lib/say.4th
s" I'll be back!" say abort" Festival not available"
```

Well, that's cool, isn't it?

8.30 Using a printer

How you access a printer depends on the operating system you're working on. That is not a flaw of 4tH, you will encounter this problem with every programming language. If you're working with MS-DOS or MS-Windows it is quite basic:

```
file printer                  \ value for printer

s" lpt1" output open dup as printer
if
  printer use
  ." This will be printed." cr
  stdout use
then
```

[3]There might still be an MS-Windows package available at *http://flame.cs.dal.ca/~lalita/festival/festivalXP.htm*. This has not been tested.

[4]Homepage: *http://www.cstr.ed.ac.uk/projects/festival/*.

Just open the port as a file and print to it. Unix isn't that different, but instead of opening a file, you open a pipe:

```
file printer                  \ value for printer

s" lp" output pipe + open dup as printer
if
  printer use
  ." This will be printed." cr
  stdout use
then
```

If you're using a different Operating System, you may have to check your manual.

8.31 The layout of the Character Segment

The final topic of this chapter again. You already know that 4tH checks whether an operation is still within the Character Segment. However, sometimes you want to check this yourself.

You already know how you can obtain the size of TIB and PAD. Yes, you can using '/TIB' and '/PAD'. But TIB and PAD have their addresses too. And when you query them, you will find that PAD comes after TIB:

```
." Address of TIB: " tib . cr
." Size of TIB : " /tib  . cr
." Address of PAD: " pad . cr
." Size of PAD : " /pad  . cr
```

And beyond PAD, what is there? Well, allocated memory of course. Things you defined using 'STRING'. There are two words which can give you information about allocated memory. First, 'LO'. 'LO' gives you the lowest address of allocated memory. Second, 'HI'. 'HI' gives you the highest valid address of the Character Segment. That means that:

```
0 hi c!
```

Is always valid and:

```
0 hi char+ c!
```

Is always invalid. It you try it, 4tH will stop executing the program with an error-message. 'LO' and 'HI' are addresses. Addresses are just numbers, so you can print and compare them. E.g.

```
hi char+ lo - . cr
```

Will print how much memory as allocated to your strings. And:

```
lo hi >
```

Will indicate whether you allocated any memory at all. If 'LO' is greater than 'HI', you didn't. If 'HI' is greater or equal to 'LO' you did. Experiment a bit with the knowledge you obtained in this chapter and continue with the next one where we will go much deeper into the secrets of the Integer Segment and Code Segment.

Chapter 9

Integer Segment and Code Segment

9.1 The Code Segment

It is know by designers of microprocessors that a processor can run much faster when every instruction has the same length. In fact, 4tH has his own virtual microprocessor. The compiler is nothing more than an assembler and the interpreter nothing more than an emulator on top of the real microprocessor.

In order to speed up 4tH, all instructions have the same length. They consist of a token (which is the real instruction) and an argument. The argument is a value that gives meaning to the instruction, e.g. the 'LITERAL' token means that a number is compiled here. The argument is the actual number.

Some instructions wouldn't need an argument, but for speeds sake, they have: it is always zero. Isn't that a lot overhead? Not really. Half the instructions in an actual program need an argument. Decoding a more elaborate scheme would need more processor time and more programming. So in the end, it would make hardly any difference. Except for the speed.

A token with its argument is called a word. And the Code Segment is one large array of words. Each of these words has an address and can be accessed by the word '@C'. In fact, '@C' throws the argument on the stack. Where have we seen '@C' before?

Yes, when fetching from an array of constants. These arrays are compiled into the Code Segment. How come that 4tH isn't confused by these arrays? Because they have the token 'NOOP', which does absolutely nothing.

9.2 The address of a colon-definition

You can get the address of a colon definition by using the word ''' (tick):

```
: add + ;                    \ a colon definition
' add . cr                   \ display address
```

Very nice, but what good is it for? Well, first of all the construction "' ADD" throws the address of "ADD" on the stack. In fact, it is a literal expression. You can assign it to a variable, define a constant for it, or compile it into an array of constants:

```
' add constant add-address

variable addr
' add addr !

create addresses ' add ,
```

Are you with us so far? If we would simply write "ADD", "ADD" would be executed right away and no value would be left on the stack. Tick forces 4tH to throw the address of "ADD" on the stack instead of executing "ADD".

Note this only works for your own colon-definitions. It doesn't work for 4tHs built-in words. If you try to, you'll get an error-message. What you can actually do with it, we will show you in the next section.

9.3 Vectored execution

This is a thing that can be terribly difficult in other languages, but is extremely easy in Forth. Maybe you've ever seen a BASIC program like this:

```
10 LET A=40
20 GOSUB A
30 END
40 PRINT "Hello"
50 RETURN
60 PRINT "Goodbye"
70 RETURN
```

If you execute this program, it will print "Hello". If you change variable "A" to "60", it will print "Goodbye". In fact, the mere expression "GOSUB A" can do two different things. In 4tH you can do this much more comfortable:

```
: goodbye ." Goodbye" cr ;
: hello ." Hello" cr ;

variable a

: greet a @ execute ;

' hello a !
greet

' goodbye a !
greet
```

What are we doing here? First, we define a few colon-definitions, called "HELLO" and "GOODBYE". Second, we define a variable called "A". Third, we define another colon-definition which fetches the value of "A" and executes it by calling 'EXECUTE'. Then, we get the address of "HELLO" (by using "' HELLO") and assign it to "A" (by using "A !"). Finally, we execute "GREET" and it says "Hello".

It seems as if "GREET" is simply an alias for "HELLO", but if it were it would print "Hello" throughout the program. However, the second time we execute "GREET", it prints "Goodbye". That is because we assigned the address of "GOODBYE" to "A".

The trick behind this all is 'EXECUTE'. 'EXECUTE' takes the address of e.g. "HELLO" from the stack and calls it. In fact, the expression:

```
hello
```

Is equivalent to:

```
' hello execute
```

This can be extremely useful as we will see in the next chapter when we build a full-fledged interpreter. We'll give you a little hint:

```
create subs ' hello , ' goodbye ,
```

Does this give you any ideas?

9.4 The Integer Segment

Wonder where all these variables are created? Or where that infamous stack really is? I bet you do. Well, when you define a variable, memory is allocated in the Integer Segment. When you define another one, space is allocated after the first variable. That means that if you go beyond the boundaries of the first variable, you'll end up in the space allocated to the second variable.

After the second variable there is a void. If you end up there your program will end with an error-message. However, if you define an 'ARRAY', single variable is created with a number of additional cells. You can only access these additional by referring to the array itself.

And what about the space before the first variable? There are other variables and they are not defined by you. Well, take a look at figure 9.1.

Lower memory is at the bottom. The user variables are the variables you defined yourself. The application variables can differ from host program to host program. Refer to your documentation on that subject. You have already seen the 4tH variables, which are 'BASE' and '>IN'. There are also variables you cannot access. These variables are hidden and only used by the system. All these variables are located in the Variable Area.

There is also a Stack Area, which contain the datastack and the returnstack. If you enter a number like "5", it is thrown on the datastack. Most words in 4tH take or put numbers on the datastack. It is very heavily used. We'll come to the returnstack later on.

The datastack and the returnstack share the same memory space. The datastack grows upward and the returnstack downward. If they clash the stack is full and 4tH will issue an error-message.

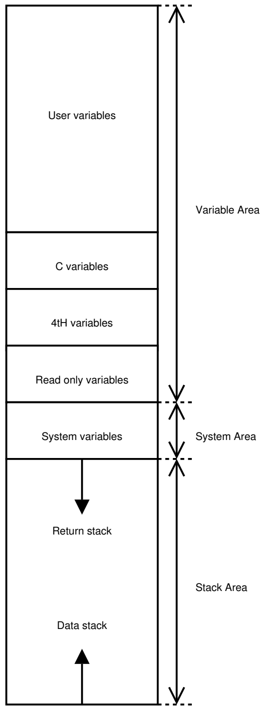

Figure 9.1: Integer segment

9.5 A portable way to access application variables

A host program can add special variables to the 4tH environment. If 4tH is used as a scripting language in e.g. a printer program, the programmer can "send" variables to 4tH. These variables are called "application variables". Do not confuse them with 4tH variables, like 'BASE' or '>IN' which are used internally by 4tH. 4tH doesn't do anything with application variables.

If the creator of the host program provided special names for each of these variables, he will probably have documented them. However, even if he didn't there is another way to access these variables.

They are stored in a predefined array called 'APP' and its values can be fetched like any other array, e.g.:

```
app 1 th @
```

Which fetches the value of the second element in the array. This also enables you to write programs that can be compiled and run under all "standard" versions of 4tH.

9.6 Returning a result to the host program

The 'APP' array can feed values from the host program to yours, but it can't return any. For that you need 'RESULT', the third 4tH variable. Returning a value is very easy. Just store it in 'RESULT'. Let's assume the host program has send two values to the 'APP' array and you want to return the sum. All you have to do is add them and store the result in 'RESULT':

```
app 0 th @ app 1 th @ + result !
```

Nothing to it..

9.7 Using commandline arguments

A host program can also transfer an array of strings to the 4tH environment. Usually, commandline arguments will be transferred this way, although any string array with the correct format can be used. If so, you will probably find it in the documentation of the host program.

If you are familiar with C, the concept is probably quite easy to understand. There are two words, 'ARGN' and 'ARGS'. 'ARGN' will leave the number of commandline arguments on the stack. The commandline arguments itself are numbered from 0 to (ARGN - 1), e.g.

```
argn 0>                         / test if there are
if                              / any arguments
  argn 0 do                     / loop through them
    i args type cr              / print them
  loop
then
```

First, we test if there are any commandline arguments. Second, if that is the case we loop through them with 'ARGN' as upper limit. Why? Since "ARGN 1- ARGS" is always the last valid commandline argument!

Third, when 'ARGS' executes, it takes a number from the stack as index. Then it leaves the address of the Character Segment (where it is temporarily stored, usually PAD) and its count on the stack.

Using the expression "TYPE CR" we can print that string. Because it is already stored in the Character Segment we can treat it like any other string. Remember, that if you don't save it anywhere else it won't last long!

9.8 The layout of the Variable Area

There are special words that allow you to get information about the layout of the Variable Area. They are called 'VARS', 'APP', 'FIRST' and 'LAST'.

'VARS' is the address of the very first variable. Before that is the Stack Area and other variables you are not allowed to touch. 'APP' is the address of the first application variable. All variables before that are 4tHs own built-in variables. 'FIRST' is the address of the first user-variable, a variable you defined yourself in your 4tH program. 'LAST' is the address of the last accessible variable, so

```
last ?
```

will *never* fail. The first question that will pop in your mind is, what can I do with them. Well, you can use it to see how many variables there are of a certain kind, so you can prevent runtime errors:

```
." number of 4tH variables: "         app vars - . cr
." number of application variables: " first app - . cr
." number of user variables: "        last first - 1+ . cr
```

These tests are possible too:

```
app vars - 0= if ." No 4tH variables" cr then
first app - 0= if ." No application variables" cr then
last first - 1+ 0= if ." No user variables" cr then
```

This is a general test to see whether the address of any variable is within range:

```
dup 0<
dup last >
or
if ." Out of range" cr then
```

You can use this check on numeric arrays too, of course.

9.9 The stacks

The Stack Area contains two stacks. So far we've talked about one stack, which is the Data Stack. The Data Stack is heavily used, e.g. when you execute this code:

```
2 3 + .
```

Only the Data Stack is used. First, "2" is thrown on it. Second, "3" is thrown on it. Third, '+' takes both values from the stack and returns the sum. Fourth, this value is taken from the stack by '.' and displayed. So where do we need the other stack for?

Well, we need it when we want to call a colon-definition. Before execution continues at the colon-definition, it saves the address of the currently executed token in the Code Segment on the other stack, which is called the Return Stack for obvious reasons.

Then execution continues at the colon-definition. Every colon-definition is terminated by ';', which compiles into 'EXIT'. When 'EXIT' is encountered, the address on top of the Return Stack is popped. Execution then continues at that address, which in fact is the place where we came from.

If we would store that address on the Data Stack, things would go wrong, because we can never be sure how many values were on that stack when we called the colon-definition, nor would be know how many there are on that stack when we encounter 'EXIT'. A separate stack takes care of that.

Try and figure out how this algorithm works when we call a colon-definition from a colon-definition and you will see that it works (4tH is proof of that).

It now becomes clear how 'EXECUTE' works. When 'EXECUTE' is called, the address of the colon-definition is on the Data Stack. All 'EXECUTE' does is copy its address on the Return Stack, take the address from the Data Stack and call it. 'EXIT' never knows the difference..

But the Return Stack is used by other words too. Like 'DO' and 'LOOP'. 'DO' takes the limit and the counter from the Data Stack and puts them on the Return Stack. 'LOOP' takes both of them from the Return Stack and compares them. If they don't match, it continues execution after 'DO'. That is one of the reasons that you cannot split a 'DO..'LOOP'.

However, if you call a colon-definition from within a 'DO'..'LOOP' you will see it works: the return address is put on top of the limit and the counter. As long as you keep the Return Stack balanced (which isn't too hard) you can get away with quite a few things as we will see in the following section.

9.10 Saving temporary values

We haven't shown you how the Return Stack works just for the fun of it. Although it is an area that is almost exclusively used by the system you can use it too.

We know we can manipulate the Data Stack only three items deep (using 'ROT'). Most of the time that is more than enough, but sometimes it isn't.

In 4tH there are special words to manipulate stack items in pairs, e.g. '2DUP' (n1 n2 – n1 n2 n1 n2) or '2DROP' (n1 n2 –). Although they are already part of 4tH, we could easily define those two ourselves:

```
: 2dup over over ;
: 2drop drop drop ;
```

You will notice that '2SWAP' (n1 n2 n3 n4 – n3 n4 n1 n2) becomes a lot harder. How can we get this deep? You can use the Return Stack for that..

The word '>R' takes an item from the Data Stack and puts it on the Return Stack. The word 'R>' does it the other way around. It takes the topmost item from the Return Stack and puts it on the Data Stack. Let's try it out:

```
: 2swap ( n1 n2 n3 n4)        \ four items on the stack
  rot   ( n1 n3 n4 n2)        \ rotate the topmost three
  >r    ( n1 n3 n4)           \ n2 is now on the Return Stack
  rot   ( n3 n4 n1)           \ rotate other items
  r>    ( n3 n4 n1 n2)        \ get n2 from the Return Stack
;
```

And why does it work in this colon-definition? Why doesn't the program go haywire? Because the Return Stack is and was perfectly balanced. The only thing we had to do was to get off "n2" before the semi-colon was encountered. Remember, the semi-colon compiles into 'EXIT' and 'EXIT' pops a return-address from the Return Stack. Okay, let me show you the Return Stack effects:

```
: 2swap ( r1)
  rot   ( r1)
  >r    ( r1 n2)
  rot   ( r1 n2)
  r>    ( r1)
;       ( --)
```

Note, these are the Return Stack effects! "R1" is the return-address. And it is there on top on the Return Stack when 'EXIT' is encountered. The general rule is:

"Clean up your mess inside a colon-definition"

If you save two values on the Return Stack, get them off there before you attempt to leave. If you save three, get three off. And so on. This means you have to be very careful with looping and branching. Otherwise you have a program that works perfectly in one situation and not in another:

```
: this-wont-work ( n1 n2 -- n1 n2)
  >r             ( n1)
  0= if          ( --)
    r>           ( n2)
    dup          ( n2 n2)
  else
    1 2          ( 1 2)
  then
;
```

This program will work perfectly if n1 equals zero. Why? Let's look at the Return Stack effects:

```
: this-wont-work ( r1)
  >r             ( r1 n2)
  0= if          ( r1 n2)
    r>           ( r1)
    dup          ( r1)
  else           ( r1 n2)
    1 2          ( r1 n2)
  then
;
```

You see when it enters the 'ELSE' clause the Return Stack is never cleaned up, so 4tH attempts to return to the wrong address. Avoid this, since this can be very hard bugs to fix.

9.11 The Return Stack and the DO..LOOP

We've already told you that the limit and the counter of a DO..LOOP (or DO..+LOOP) are stored on the Return Stack. But how does this affect saving values in the middle of a loop? Well, this example will make that quite clear:

```
1         ( n)
10 0 do   ( n)
  >r      ( --)
  i .     ( --)
  r>      ( n)
loop      ( n)
cr        ( n)
drop      ( --)
```

You might expect that it will show you the value of the counter ten times. In fact, it doesn't. Let's take a look at the Return Stack:

```
1       ( --)
10 0 do ( 1 c)
  >r    ( 1 c n)
  i .   ( 1 c n)
  r>    ( 1 c)
loop    ( --)
cr      ( --)
drop    ( --)
```

You might have noticed that it prints ten times the number "1". Where does it come from? Usually 'I' prints the value of the counter, which is on top of the Return Stack.

This time it isn't: the number "1" is there. So 'I' thinks that "1" is actually the counter and displays it. Since that value is removed from the Return Stack when 'LOOP' is encountered, it doesn't do much harm.

We see that we can safely store temporary values on the Return Stack inside a DO..LOOP, but we have to clean up the mess, before we encounter 'LOOP'. So, this rule applies here too:

> "Clean up your mess inside a DO..LOOP"

But we still have to be prepared that the word 'I' will not provide the expected result (which is the current value of the counter). In fact, 'I' does simply copy the topmost value on the Return Stack. Which is usually correct, unless you've manipulated the Return Stack yourself.

Note that there are other words beside 'I', which do exactly the same thing: copy the top of the Return Stack. But they are intended to be used outside a DO..LOOP. We'll see an example of that in the following section.

9.12 Other Return Stack manipulations

The Return Stack can avoid some complex stack acrobatics. Stack acrobatics? Well, you know it by now. Sometimes all these values and addresses are just not in proper sequence, so you have to 'SWAP' and 'ROT' a lot until they are.

You can avoid some of these constructions by just moving a single value on the Return Stack. You can return it to the Data Stack when the time is there. Or you can use the top of the Return Stack as a kind of local variable.

No, you don't have to move it around between both stacks all the time and you don't have to use 'I' out of its context. There is a well-established word, which does the same thing: 'R@'. This is an example of the use of 'R@':

```
: delete      ( n )
  >r #lag +   ( a1)
  r@ #lag     ( a1 a2 n2)
  r@ negate   ( a1 a2 n2 n3)
  r# +!       ( a1 a2 n2)
  #lead +     ( a1 a2 n2 a3)
  swap cmove  ( a1)
  r> blanks   ( --)
;
```

'R@' copies the top of the Return Stack to the Data Stack. This example is taken from the 4tH-editor. It deletes "n" characters left of the cursor. By putting the number of characters on the Return Stack right away, its value can be fetched by 'R@' without using 'DUP' or 'OVER'. Since it can be fetched at any time, no 'SWAP' or 'ROT' has to come in.

9.13 Altering the flow with the Return Stack

The mere fact that return addresses are kept on the stack means that you can alter the flow of a program. This is hardly ever necessary, but if you're a real hacker you'll try this anyway, so we'd better give you some pointers on how it is done. Let's take a look at this program. Note that we comment on the Return Stack effects:

```
: soup ." soup " ;              ( r1 r2)
: dessert ." dessert " ;        ( r1 r6)
: chicken ." chicken " ;        ( r1 r3 r4)
: rice ." rice " ;              ( r1 r3 r5)
: entree chicken rice ;         ( r1 r3)
: dinner soup entree dessert ;  ( r1)
dinner cr                       ( --)
```

And this is the output:

```
soup chicken rice dessert
```

Before we execute "DINNER" the Return Stack is empty. When we enter "DINNER" the return address to the main program is on the Return Stack (r1).

"DINNER" calls "SOUP". When we enter "SOUP" the return address to "DINNER" is on the Return Stack (r2). When we are done with "SOUP", its return address disappears from the Return Stack and execution continues within "DINNER".

Then "ENTREE" is called, putting another return address on the Return Stack (r3). "ENTREE" on its turn, calls "CHICKEN". Another return address (r4) is put on the Return Stack. Let's take a look on what currently lies on the Return Stack:

- Top Of Return Stack (TORS)

r4 returns to ENTREE

r3 returns to DINNER

r1 returns to main program

As we already know, ';' compiles an 'EXIT', which takes the TORS and jumps to that address. What if we lose the current TORS? Will the system crash?

Apart from other stack effects (e.g. too few or the wrong data are left on the Data Stack) nothing will go wrong. Unless the colon-definition was called from inside a DO..LOOP, of course. But what DOES happen? The solution is provided by the table: it will jump back to "DINNER" and continue execution from there.

```
: soup ." soup " ;                ( r1 r2)
: dessert ." dessert " ;          ( r1 r6)
: chicken ." chicken " r> drop ;  ( r1 r3 - r4 gets lost!)
: rice ." rice " ;                ( r1 r3 r5)
: entree chicken rice ;           ( r1 r3)
: dinner soup entree dessert ;    ( r1)
dinner cr                         ( --)
```

Since "CHICKEN" gets rid of the return address to "ENTREE", "RICE" is never called. Instead, a jump is made to "DINNER" that assumes that "ENTREE" is done, so it continues with "DESSERT". This is the output:

```
soup chicken dessert
```

Note that this is *not* common practice and we do not encourage its use. However, it gives you a pretty good idea how the Return Stack is used by the system.

9.14 Leaving a colon-definition

You can sometimes achieve the very same effect by using the word 'EXIT' on a strategic place. We've already encountered 'EXIT'. It is the actual word that is compiled by ';'.

What you didn't know is that you can compile an 'EXIT' without using a ';'. And it does the very same thing: it pops the return address from the Return Stack and jumps to it. Let's take a look at our slightly modified previous example:

```
: soup ." soup " ;              ( r1 r2)
: dessert ." dessert " ;        ( r1 r6)
: chicken ." chicken " ;        ( r1 r3 r4)
: rice ." rice " ;              ( is never reached)
: entree chicken exit rice ;    ( r1 r3)
: dinner soup entree dessert ;  ( r1)
dinner cr                       ( --)
```

After "CHICKEN" has been executed by "ENTREE", an 'EXIT' is encountered. 'EXIT' works just like ';', so 4tH thinks the colon-definition has come to an end and jumps back to "DINNER". It never comes to calling "RICE", so the output is:

```
soup chicken dessert
```

'EXIT' is mostly used in combination with some kind of branching like IF..ELSE..THEN. Compare it with 'LEAVE' that leaves a DO..LOOP early.

But now for the big question: what is the difference between 'EXIT' and ';'? Both compile an 'EXIT', but they are not aliases. 4tH will try to match every ';' with a ':'. If it doesn't succeed, it will issue an error message. This matching is not performed by 'EXIT'.

9.15 The layout of the Stack Area

Before we tell you how to obtain information on the Stack Area, we first have to explain you how it is laid out. We've already seen that there are two stacks: the Data Stack and the Return Stack. We also know what they are used for.

The next question is what part of the Stack Area is used by the Data Stack and what part is used by the Return Stack. In fact, both stacks share the very same Stack Area.

The Data Stack grows upward from the bottom and the Return Stack grows downward from the top. When they meet, you're in trouble. If the Return Stack causes the overflow, 4tH will report that the Return Stack overflowed. If it was the Data Stack, it will report that the Data Stack overflowed.

If an overflow happens, you can't say which stack actually overflowed. If the Data Stack filled up the Stack Area and a colon-definition tries to put a return address on the Return Stack, the Return Stack will get the blame.

Now for the good news. Because of this shared stack space, programs with different requirements can run without having to modify stack sizes (you can't do that; only the programmer of your application can). It can be a program that heavily uses the Return Stack (recursive colon-definitions) or a program that needs lots of data on the Data Stack.

What you can check is how big the Stack Area actually is. It is a constant named 'STACK'. It will report the size in cells. Every value on any stack (address or value) takes up a single cell.

You can also ask 4tH how many values are on the Data Stack using 'DEPTH'. It will report the number of values, before you executed 'DEPTH'. Let's elaborate on that a little more:

```
." Begin" cr                  \ no values on the stack
10                            \ 1 value on the stack
5                             \ 2 values on the stack
9                             \ 3 values on the stack
depth                         \ 4 values on the stack
. cr                          \ 4tH reports "3"
```

If you want to know what values the actual stack pointers have, you have to use 'SP@' and 'RP@'. By subtracting 'SP@' from 'RP@' you can see how much space is left in the Stack Area:

```
rp@ sp@ -
." Space left: " . ." cells" cr
```

9.16 Booleans and numbers

You might have expected we had discussed this subject much earlier. But we haven't and for one very good reason. We've told you a few chapters ago that 'IF' branches if the top of the stack is non-zero. Any number will do. So you would expect that this program will print "I'm here":

```
1 2 and
if
  ." I'm here"
then
```

In fact, it doesn't! Why? Well, 'AND' is a BINARY operator, not a LOGICAL operator. That means it reacts on bit-patterns. Given two numbers, it will evaluate bits at the same position.

The number "1" is "01" in binary. The number "2" is "10" in binary. 'AND' will evaluate the first bit (binary digit, now you know where that came from!). The first bit is the rightmost bit, so "0" for the number "2" and "1" for the number "1".

'AND' works on a simple rule, if both bits are "1" the result will be "1" on that position. Otherwise it will be "0". So "1" and "0" are "0". The evaluation of the second bit has the same result: "0". We're stuck with a number that is "0". False. So 'IF' concludes that the expression is not true:

```
2 base ! [binary]              \ set radix to binary
10                             \ binary number "2"
01 AND                         \ binary number "1"
. cr                           \ binary result after AND
```

It will print "0". However, "3" and "2" would work just fine:

```
2 base ! [binary]              \ set radix to binary
10                             \ binary number "2"
11 AND                         \ binary number "3"
. cr                           \ binary result after AND
```

It will print "10". The same applies to other binary operators as 'OR' and 'INVERT'. 'OR' works just like 'AND' but works the other way around. If both bits are "0" the result will be "0" on that position. Otherwise it will be "1":

```
2 base ! [binary]              \ set radix to binary
10                             \ binary number "2"
01 OR                          \ binary number "1"
. cr                           \ binary result after OR
```

It will print "11". We do not encourage the use of 'INVERT' for logical operations. You should use '0=' instead.

'0=' takes the top of the stack and leave a true-flag if it is zero. Otherwise it will leave a false-flag. That means that if a condition is true (non-zero), it will leave a false-flag. Which is exactly what a logical NOT should do.

Take a look at his brother '0<>'. '0<>' takes the top of the stack and leaves a true-flag if it is non-zero. Otherwise it will leave a false-flag.

The funny thing is 'AND' and 'OR' work perfectly with flags and behave as expected. '0<>' will convert a value to a flag for you. So this works:

```
1 0<>
2 0<>
and if
  ." I'm here" cr
then
```

Of course, you don't have to use '0<>' when a word returns a flag. You should check the glossary for details on that.

9.17 Using ' with other names

So far we've only used "' (tick) with colon-definitions, but you can also use it with all constants, variables, values, strings, vectors (see section 10.11) and constant arrays. However, the information it provides is not always useful. E.g. the expression:

```
10 constant ten
' ten
```

Does not compile differently from:

```
10 constant ten
ten
```

The same applies to constant arrays and strings. It will give you possibly information on the address of variables, vectors, arrays and values, e.g.:

```
variable ten
' variable ." Relative address of ten: " . cr
```

Yes, *relative* address! What does that mean? When a 4tH program is compiled it has no idea how many application variables a host program will provide. So it stores a *relative* address. This address is relative to the address returned by 'FIRST'. You might call it an offset if you want to. 4tH provides a word which will convert the relative address of vectors, variables, values and numeric arrays to an absolute address, called '>BODY'. So this piece of code does exactly the same thing:

```
variable ten
ten                           \ throw address of 'ten' on stack
dup                           \ duplicate address
10 swap !                     \ store 10 at address
? cr                          \ show value stored at address
```

As this piece of code:

```
variable ten
' ten >body                   \ calculate address
dup                           \ duplicate address
10 swap !                     \ store 10 at address
? cr                          \ show value stored at address
```

There are not too many occasions where this is useful, but it let's take a look at this one:

```
0 value ten                   \ define a value
' ten >body                   \ calculate address of value
dup                           \ duplicate address
10 swap !                     \ store 10 at address
? cr                          \ show value stored at address
```

We already know that values, numeric arrays and vectors are stored in the very same area of the Integer Segment. This construction makes it possible to access them as variables.

You can access string constants or arrays of string constants with tick, but they will return a value which only has a meaning to 4tH itself. You won't be able to do anything useful with those values.

You should avoid these kind of constructions, but there might be some situations out there where it might come in handy. Note that you can only tick your *own* names. All of 4tHs built-in variables, strings, words, etc. cannot be accessed by tick.

9.18 Assertions

You have probably seen this before: you've made a program, compiled it and it doesn't work. Then you start putting code at strategic places, trying to pinpoint the error. And when you're finally done, you've got to revisit all of these places to remove that code. And you probably forget a few..

4tH has a built-in facility which allows to put that code there, debug your program and remove the debugging code from your program by changing a single line.

It is called "assertion" and those of you who have ever worked with C probably know what we're talking about.

An assertion is a line of code that will evaluate an expression. If the expression evaluates to false, it will exit the program with an error message. Let's take a look at this simple colon-definition:

```
: add                          \ expects two numbers on the stack
  +
;
```

If we call add by writing:

```
1 add
```

it will fail. Now we add this assertion:

```
assert( depth 2 >= )
```

It will evaluate to false when there are less than two items on the stack. The program will be terminated and the appropriate error message will be issued. You may think that this is nice, but you still have to remove all assertion manually.

Not true! If you tried this out already you will see that you won't find an assertion anywhere. It's gone! True, if you want to use assertions you have to enable them. You do that with the word '[ASSERT]':

```
[assert]
: add
  assert( depth 2 >= )
  +
;

1 add
```

Now assertions will compile and work. If you remove the word '[ASSERT]' all assertions will disappear like they were comment. '[ASSERT]' works just like '[DECIMAL]', '[HEX]', etc. They work linear and do not follow the program flow. If you put '[ASSERT]' halfway your source-file you will notice that assertions work from that point:

```
: add
  assert( depth 2 >= )        \ assertions disabled
  +
;

[assert]                      \ enable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions enabled
  . cr r> base !
;
```

Assertions are only enabled in "PRINT-HEX". The assertion inside "ADD" will be removed and thus be disabled. But there is more to '[ASSERT]' than the eye meets. It doesn't enable assertions, it toggles them. When the 4tH compiler starts, assertions are disabled. The first '[ASSERT]' enables them. A second '[ASSERT]' will disable them again:

```
[assert]                      \ enable assertions

: add
  assert( depth 2 >= )        \ assertions enabled
  +
;

[assert]                      \ disable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions disabled
  . cr r> base !
;
```

There are many possibilities:

- You can start testing low level colon-definitions and move your way up to the high level definitions by moving '[ASSERT]' down.
- You can enable assertions on certain parts of your code by enclosing them with an '[ASSERT]' pair.
- You can switch the entire context of '[ASSERT]'s by adding a single '[ASSERT]' to the top of your source.

You are not limited to range-checking when using 'ASSERT('. Any expression that evaluates to TRUE is allowed:

```
[assert]
: add
  assert( ." ADD starts at " here . cr true )
  assert( depth 2 >= )
  assert( ." Values: " over over . . cr true )
  +
;
```

We're sure you can come up with more useful ideas. We did too.

9.19 Breakpoints

4tH also offers you the possibility to set breakpoints. It's quite easy to enable this facility. Just add this to the very beginning of your source:

```
[needs lib/debug.4th]
```

Setting a breakpoint is quite easy too, e.g. this piece of code malfunctions:

```
32 string argument
1 args argument place
```

Change it to:

```
32 string argument
1 args argument ~~ place
```

Now the breakpoint is enabled. It will enter a Forth-like shell just before 'PLACE' is executed. Now a host of words are at your disposal. You can examine any region of the Character Segment with "DUMP" or print any string variable with "TYPE". 4tHs internal variables and regions are known by name, like "PAD", "TIB", ">IN", "BASE" and "OUT". You can examine them or any other variable by using "?", "@" and ".".

You have a small calculator, that you can use to multiply, substract, add. You can change 'BASE' by executing "OCTAL", "HEX", "BINARY" or "DECIMAL". It also has a host of binary operators like "OR", "AND", "XOR", "INVERT", "LSHIFT" and "RSHIFT". It also has stack operators like "DUP", "DROP", "OVER" and "SWAP". "CLEAR" will clear the stack for you.

You can examine both stacks. ".S" will show you the data stack (including any rubbish you put there yourself during the debugging session) and "R.S" will show you the return stack. "DEPTH" and "RDEPTH" will tell you how many items there are on the stack. When you're done, you may leave the debugger by typing "BYE". Your program will continue as usual.

A word of caution: since the debugger is a 4tH program itself, it doesn't actually freeze the virtual machine. It just *seems* like it is frozen. The contents of PAD may be slightly different than you expected. If you *really* need to examine the PAD as it was, don't examine it directly, but use "SPAD" to examine the "shadow PAD". "SPAD" leaves the address for the "shadow PAD" on the stack. The same goes for ">IN", "BASE" and "OUT": never examine these by address, but always by name. Although every effort has been made to catch any errors, some extreme stress tests might fail. It is not recommended to use the debugger when stack space is very tight.

9.20 Random numbers

If you want to program a game or a simulation, you'll probably need random number generation. Of course, you can do that too with 4tH. It generates a number between 0 and 'MAX-RAND', but we'll teach you how to generate a number for virtually any range below that.

There are two things important when you want to do that: the range limit and the lower limit. Say, we want to simulate a dice with numbers in a range from 1 to 6. The lower limit is 1. We subtract that from the upper limit to get the range limit.

So: upper limit minus lower limit gives a range limit of five (6 - 1 = 5). The general formula looks like this:

<range-limit> 1+ random * max-rand 1+ / <lower limit> +

When we apply this to the dice-example, the complete formula is:

```
5 1+ random * max-rand 1+ / 1 +
```

This will give you a dice-simulation, that produces random numbers between 1 and 6. Happy? Then thank Wil Baden for the algorithm!

9.21 Timers

There is a very low level word in 4tH that keeps track of time. It has several uses. Like a timer that measures how long certain operation takes, like the execution of a colon-definition ("DO-SOME-WORD" in this case):

```
time do-some-word time
swap -
." Do-Some-Word took " . ." seconds." cr
```

There is a somewhat more elaborate library member that does it all for you:

```
[needs lib/timer.4th]

timer-reset
do-some-word
.elapsed
```

This always prints the number of seconds that have elapsed. If you want to create your own display, you can define one easily:

```
[needs lib/timer.4th]

:noname <# # 6 base ! # decimal 58 hold # #> type ."  mins" ;
is timer-stop
```

You define "TIMER-STOP" *after* inclusion of "time.4th", but *before* the first usage of ".ELAPSED".

9.22 Time & Date

There is also a word in 4tH that will tell you what time and what date it is. With a little trouble ;). The word is called 'TIME' (again) and it will tell you how many seconds have gone since January 1st, 1970. You can also find out quickly how late it is:

```
[needs lib/time.4th]

now ." hours:" . ." minutes:" . ." seconds:" . cr
```

Note that it doesn't know about daylight-saving! It does know about timezones, which may be neccesary on *some* systems. You can determine your timezone by looking at an email message from a local friend. It will probably say somewhere:

```
Date: Mon, 25 Feb 2002 22:28:59 +0100 (CET)
```

The '+0100' means that you're in timezone CET, which is one hour later than GMT. If it said:

```
Date: Sun, 16 Dec 2001 02:19:40 -0800 (PST)
```

This indicates that you're in timezone PST, which is eight hours earlier than GMT. In that case 'tz' would be:

```
-8 3600 [*] constant tz        \ Pacific Standard Time
```

If you need it, define it accordingly *before* the inclusion of "time.4th". The day of the week is another thing you can easily calculate:

```
[needs lib/time.4th]

: Weekdays
  dup 0 = if drop s" Monday"    exit then
  dup 1 = if drop s" Tuesday"   exit then
  dup 2 = if drop s" Wednesday" exit then
  dup 3 = if drop s" Thursday"  exit then
  dup 4 = if drop s" Friday"    exit then
  dup 5 = if drop s" Saturday"  exit then
  dup 6 = if drop s" Sunday"    exit then
;

today weekday Weekdays type cr
```

By the way, didn't you hate the way we had to define "Weekdays"? Ugly, isn't it? Well, there is a better way to do it. You'll learn that in the next chapter (see section 10.13)! You can also print the full date:

```
[needs lib/time.4th]

today ." year:" . ." month:" . ." day:" . cr
```

Just don't ask me how this thing works, Everett F. Carter figured this one out. "TODAY" does the easy work. "JDATE" converts the Julian day to the Gregorian date. There is also a way to convert a Gregorian date to a Julian day, called "JDAY":

```
[needs lib/time.4th]

26 02 2002 jday 64 - jdate
today ." year:" . ." month:" . ." day:" . cr
```

This can be quite handy if you want to calculate which date it was 64 days ago. ANS-Forth also defines a word that does it all called "TIME&DATE". This word throws seconds, minutes, hours, day, month and year (TOS) on the stack, but always returns GMT:

```
[needs lib/ansfacil.4th]

time&date . . . . . . cr
```

And finally, we even got a word that returns the date of easter:

```
[needs lib/easter.4th]

2005 easterSunday ." year:" . ." month:" . ." day:" . cr
```

Well, tell me, isn't that kind of neat?

9.23 What is not implemented

When writing a product like 4tH that is modelled after an existing programming language like Forth one has to cut a few corners somewhere.

Forth has a fundamentally different architecture, which allows you to extend the compiler with ease. 4tH is much more like conventional programming languages and many still wonder how we got this far.

When you're learning 4tH to learn Forth you will find there are things you can't do in 4tH. This section sums up most of the restrictions 4tH has in comparison to Forth and other languages.

Datatypes There are no words that allow you to define your own datatypes, although you can change the behaviour of individual variables.

Floating point 4tH does not support floating point, but there are several ways to implement fixed point calculation which can do nearly the same job as you will learn in the next chapter.

Interpreter Since 4tH is a conventional compiler, you won't find a built-in interpreter. There is a library-source, which will enable you to make an interpreter for specific applications with ease. Next chapter we will show you how to use it.

If you have more questions concerning the functionality of 4tH, please read the ANS-Forth document. This describes the compliance of 4tH to the ANS-Forth standard. Further information can be obtained by studying the glossary.

9.24 Known bugs and limitations

Like every software product, 4tH has bugs. Because a work-around is available, fixing these bugs has no high priority.

- When you use '\' without any actual comment in a Unix ASCII file the complete next line will be marked as comment. With MS-DOS ASCII files 4tH will correctly detect a null string and terminate with an error. Use "\ ." just to be safe.
- There can be only *one space* between '[DEFINED]', '[UNDEFINED]', '[CHAR]', 'CHAR' and the string following it. If you don't comply, 4tH will complain about empty string constants.
- You cannot comment out '[THEN]' or ')' This is bad practice anyway.

Chapter 10

Advanced programming

10.1 Compiletime calculations

When you've reached this chapter, you must have quite some experience with 4tH. This chapter will help you to use 4tH to its full capacity. You'll be able to use software exceptions, conditional compilation, compiletime calculation, lookup tables, fixed point calculation and much, much more.

We'll start with something wich may seem difficult at first, but is extremely handy in some circumstances. We've already explained that when you define a string, it has to be preceded by a literal expression. So you cannot define something like this:

```
64 constant name
16 constant #names

name #names * string name_space
```

When you want to do this in 4tH you first have to calculate the size of "name_space" by hand and then insert it into your program:

```
64 constant name
16 constant #names

1024 string name_space
```

But this has a serious drawback when maintaining your code, because when you change either "name" or "#names", you have to remember that you have to recalculate "name_space"!

Is there no solution to this problem? Of course, although you had to wait for version 3.3a to get it:

```
64 constant name
16 constant #names

name #names [*] string name_space
```

The word '[*]' takes two subsequent literal expressions and multiplies them to a new single literal expression as if you'd written "1024" yourself! The difference between:

```
10 10 *
```

and:

```
10 10 [*]
```

is essentially that the first expression will compile to two literals and the word '*' and the second expression will compile to just a literal.

In other words: the first expression will just compile and quietly wait until it is evaluated at runtime, while the second is already evaluated at compiletime. Which means that everything that is evaluated, must already be known at compiletime, thus a literal expression.

There are other compiletime calculations possible too. The first one is '[+]', which adds two literal expressions. This will compile to the literal "12":

```
5 7 [+]
```

You can even mix and chain compiletime calculations. This will compile to the literal "500":

```
5 25 75 [+] [*]
```

Just as if you'd just written "500" in the sourcecode yourself. You can also write:

```
25 75 [+] 5 [*]
```

Because it's just simple postfix notation. Note that there must be two subsequent literal expressions available at any time, so this doesn't work:

```
5 5 [*] dup [*]
```

Since 'DUP' isn't a literal expression, but a word which is simply compiled. But don't worry: 4tH will notify you when you make an error like this. Another useful word is '[NEGATE]', e.g. when you need to assign a negative value to a constant:

```
              16 constant +range
+range [negate] constant -range
```

In this example the value of "-RANGE" is -16. The final word, we'd like to present you is '[NOT]', which logically inverts a flag at compiletime, just like '0=' at runtime. This expression will compile to a true flag:

```
false [not]
```

You might wonder why we included this one, but that will become clear when you read the next section.

10.2 Conditional compilation

This is something which can be very handy when you're designing a 4tH program for different environments or even different Forth compilers. Let's say you've written a general ledger program in 4tH that is so good, you can sell it. Your customers want a demo, of course. You're willing to give one to them, but you're afraid they're going to use the demo without ever paying for it.

One thing you can do is limit the number of entries they can make. So, you copy the source and make a special demo version. But you have to do that for every new release. Wouldn't it just be easier to have one version of the program and just change one single constant? You can with conditional compilation:

```
true constant DEMO

DEMO [if]
256 constant #Entries
[then]

variable CurrentEntry

DEMO [not] [if]
limit constant #Entries
[then]

#Entries array Entries
```

We defined a constant, called "DEMO", which is true. So, when the compiler reaches the "DEMO [if]" line, it knows that it has to compile "256 constant Entries", since "DEMO" is true. When it comes to "DEMO [not] [if]", it knows it has to skip everything up to "[then]" since "[not] DEMO" is evaluated at compiletime to false. So, in this case the compiler behaves like you've written:

```
256 constant #Entries
variable CurrentEntry
#Entries array Entries
```

Would you change "DEMO" to false, the compiler would behave as if you wrote:

```
variable CurrentEntry
limit constant #Entries
#Entries array Entries
```

The word '[IF]' only works at compile time and is *never* compiled into the object. '[IF]' takes a literal expression. If this expression is true, the code following the '[IF]' is compiled, just as '[IF]' wasn't there. Is this expression false, everything up to '[THEN]' is discarded as if it wasn't there.

That also means you can discard any code that is superfluous in the program. E.g. when you're making a colon-definition to check whether you can make any more entries. If you didn't use conditional compilation, you might have written it like this:

```
: CheckIfFull                ( n -- n)
  dup #Entries =             ( n f)
  if                         ( n)
    drop                     ( --)

    DEMO                     ( f)
    if                       ( --)
      ." Buy the full version"
    else                     \ give message and exit program
      ." No more entries"
    then                     ( --)

    cr quit
  then                       ( n)
;
```

But his one is nicer and will take up less code:

```
: CheckIfFull                ( n -- n)
  dup #Entries =             ( n f)
  if                         ( n)
    drop                     ( --)

DEMO [if]                    ( n f)
    ." Buy the full version"
[then]

DEMO [not] [if]
    ." No more entries"
[then]

    cr quit
  then                       ( n)
;
```

You can also use conditional compilation to discard large chunks of code. This is a much better way than to comment all the lines out, e.g. this won't work anyway:

```
(
: room?                      \ is it a valid variable?
  dup                        ( n n)
  size 1- invert and         ( n f)
  if                         \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
)
```

This is pretty cumbersome and prone to error:

```
\ : room?                    \ is it a valid variable?
\   dup                      ( n n)
```

```
\   size 1- invert and        ( n f)
\   if                        \ exit program
\     drop ." Not an element of ROOM" cr quit
\   then
\ ;
```

But this is something that can easily be handled:

```
false [if]
: room?                       \ is it a valid variable?
  dup                         ( n n)
  size 1- invert and          ( n f)
  if                          \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
[then]
```

Just change "false" to "true" and the colon-definition is part of the program again. Note that '[IF] .. [THEN]' can be nested! Conditional compilation is very powerful and one of the easiest features a language can have. And it's ANS-Forth compatible!

10.3 Checking the environment at compiletime

Let's say you've written something which works perfectly on your own machine and you want to use it on the mainframe at work. It turns out to be it doesn't work. Why? Because your program assumed that a cell was four address units wide. And it didn't turn out to be that way.

You could have prevented that if you had used a check at compiletime. You can do that this way:

```
/cell 4 [=] [NOT] [IF]
( do something)
[THEN]
```

'/CELL' is a constant which holds the number of address units in a cell. '/CELL' has got a little brother called '/CHAR', which will tell you how many address units there are in a character. '[=]' will check whether a cell has four address units and '[NOT]' will reverse that flag. Neat huh?

But then again, what do we do if it doesn't turn out to be that way. Any action will first be executed at runtime so a message or 'ABORT' won't do. Further compilation will be useless, so we actually want to stop. You're in luck, since we have a special word that will stop the compiler regardless. It's called '[ABORT]'. So this is our complete snippet:

```
/cell 4 [=] [NOT] [IF]
[ABORT]
[THEN]
```

10.4 Checking a definition at compiletime

We've already encountered 'COMPARE' in section 7.16. 'COMPARE' is word that compares two strings. It can do that both case sensitive and case insensitive. If you define a constant called "IGNORECASE" before the '[NEEDS' directive and set it to FALSE, it will perform a case sensitive comparison. If you don't, it will do a case insensitive comparison by default.

Most approaches would require the definition of "IGNORECASE", regardless which mode you select. This one doesn't:

```
[DEFINED] ignorecase [NOT] [IF]
true constant ignorecase             \ default ignore case
[THEN]

: compare                            ( a1 n1 a2 n2 -- f )
  rot over over swap - >r            ( a1 a2 n2 n1)
  min 0 tuck                         ( a1 a2 0 n 0)
  ?do                                ( a1 a2 f)
    drop                             ( a1 a2)
    over i + c@                      ( a1 a2 c1)
ignorecase [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    over i + c@                      ( a1 a2 c1 c2)
ignorecase [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    - dup                            ( a1 a2 f f)
    if leave then                    ( a1 a2 f)
  loop
  >r drop drop r> r> swap dup        ( f1 f2 f2)
  if swap then drop                  ( f)
;
```

'[DEFINED]' checks whether the word following it has been defined and leaves a TRUE flag if it was. It doesn't matter whether the word is built-in, included or defined in your program. It can be a variable, a word, a constant, anything you like.

In this case it checks whether "IGNORECASE" has been defined. If it wasn't it will define it for you. Later, it checks whether "IGNORECASE" is TRUE. If it is, a line of code is compiled. If it is FALSE, it will compile that line.

If an 'IF' had been used, the code would always be compiled with the added overhead of testing a constant at runtime. This construction allows for tighter and faster code.

Note that '[DEFINED]' has also a counterpart called '[UNDEFINED]'. It is equivalent to "[DEFINED] [NOT]" and leaves a true flag when the word following it has *not* been defined.

10.5 Exceptions

You know when you violate the integrity of 4tH, it will exit and report the cause and location of the error. Wouldn't it be nice if you could catch these errors within the program?

It would save a lot of error-checking anyway. It is quite possible to check every value within 4tH, but it takes code and performance, which makes your program less compact and slower.

Well, you can do that too in 4tH. And not even that, you can trigger your own errors as well. This simple program triggers an error and exits 4tH when you enter a "0":

```
[needs lib/enter.4th]         \ get a number
                              \ if non-zero, return it
                              \ if zero, throw exception
: could-fail                  ( -- n)
  enter dup 0=
  if 1 throw then
;
                              \ drop numbers and
                              \ call COULD-FAIL
: do-it                       ( -- )
  drop drop could-fail
;
                              \ put 2 nums on stack and
                              \ execute DO-IT
: try-it                      ( -- )
  1 2 ['] do-it execute
  ." The number was" . cr
;
                              \ call TRY-IT
try-it
```

"TRY-IT" puts two numbers on the stack, gets the execution token of "DO-IT" and executes it. "DO-IT" drops both numbers and calls "COULDFAIL". "COULD-FAIL" gets a number and compares it against "0". If zero, it calls an exception. If not, it returns the number.

The expression "1 THROW" has the same effect as calling 'QUIT'. The program exits, but with the error message "Unhandled exception". You can use any positive number for 'THROW', but "0 THROW" has no effect. This is called a "user exception", which means you defined and triggered the error.

There are also system exceptions. These are triggered by the system, e.g. when you want to access an undefined variable or print a number when the stack is empty. These exceptions have a negative number, so:

```
throw -4
```

Will trigger the "Stack empty" error. You can use these if you want but we don't recommend it, since it will confuse the users of your program.

You're probably not interested in an alternative for 'QUIT'. Well, 'THROW' isn't. It just enables you to "throw" an exception and exceptions can be caught by your program. That means that 4tH won't exit, but transfers control back to some routine. Let's do just that:

```
[needs lib/enter.4th]

: could-fail                  ( -- n)
  enter dup 0=
```

```
  if 1 throw then
;

: do-it                         ( -- )
  drop drop couldfail
;

: try-it                        ( -- )
  1 2 ['] do-it catch
  if drop drop ." There was an exception" cr
  else ." The number was" . cr
  then
;

try-it
```

The only things we changed is a somewhat more elaborate "TRY-IT" definition and we replaced 'EXECUTE' by 'CATCH'.

'CATCH' works just like 'EXECUTE', except it returns a result-code. If the result-code is zero, everything is okay. If it isn't, it returns the value of 'THROW'. In this case it would be "1", since we execute "1 THROW". That is why "0 THROW" doesn't have any effect.

If you enter a nonzero value at the prompt, you won't see any difference with the previous version. However, if we enter "0", we'll get the message "There was an exception", before the program exits.

But hey, if we got that message, that means 4tH was still in control! In fact, it was. When "1 THROW" was executed, the stack-pointers were restored and we were directly returned to "TRY-IT". As if "1 THROW" performed an 'EXIT' to the token following 'CATCH'.

Since the stack-pointers were returned to their original state, the two values we discarded in "DO-IT" are still on the stack. But the possibility exists they have been altered by previous definitions. The best thing we can do is discard them.

So, the first version exited when you didn't enter a nonzero value. The second version did too, but not after giving us a message. Can't we make a version in which we can have another try? Yes we can:

```
[needs lib/enter.4th]

: could-fail                    ( -- n)
  enter dup 0=
  if 1 throw then
;

: do-it                         ( -- )
  drop drop could-fail
;

: retry-it                      ( -- )
  begin
    1 2 ['] do-it catch
  while
    drop drop ." Exception, keep trying" cr
  repeat
```

```
  ." The number was " . cr
;

retry-it
```

This version will not only catch the error, but it allows us to have another go! We can keep on entering "0", until we enter a nonzero value. Isn't that great? But it gets even better! We can exhaust the stack, trigger a system exception and still keep on going. But let's take it one step at the time. First we change "COULD-FAIL" into:

```
: could-fail                  ( -- n)
  enter dup 0=
  if drop ." Stack: " depth . cr 1 throw then
;
```

This will tell us that the stack is exhausted at his point. Let's exhaust is a little further by redefining "COULD-FAIL" again:

```
: could-fail                  ( -- n)
  enter dup 0=
  if drop drop then
;
```

Another 'DROP'? But wouldn't that trigger an "Stack empty" error? Yeah, it does. But instead of exiting, the program will react as if we wrote "-4 THROW" instead of "DROP DROP". The program will correctly report an exception when we enter "0" and act accordingly.

This will work with virtually every runtime error. Which means we won't have to protect our program against every possible user-error, but let 4tH do the checking.

We won't even have to set flags in every possible colon-definition, since 4tH will automatically skip every level between 'THROW' and 'CATCH'. Even better, the stacks will be restored to the same depth as they were before 'CATCH' was called.

You can handle the error in any way you want. You can display an error message, call some kind of error-handler, or just ignore the error. Is that enough flexibility for you?

10.6 Mixing character and number data

Sometimes you have to mix character and number data, e.g. when you're porting a Forth program or when the need complex datastructures arises. Since 4tH gives each datatype its own segment this is not easy. However, there is a library that can help you. Let's have a look at this program:

```
16 constant /my               \ size of array
/my array my                  \ define array
0                             \ set up counter
begin
  dup dup                     \ duplicate counter
  cells my + !                \ store counter in array
  1+                          \ increment counter
```

```
  dup /my =                   \ limit reached?
until drop                    \ drop the counter

my                            \ set up index
begin
  dup @ . cr                  \ print the value
  cell+                       \ next element
  dup my /my cells + =        \ limit reached
until drop                    \ drop the index
```

This simple program defines a small array, fills and displays it. Now, this little thing does the same thing, but is located in the Character Segment:

```
include lib/ncoding.4th
                              \ load the library
16 constant /my               \ size of array
/my nell [*] string my        \ define array
0                             \ set up counter
begin
  dup dup                     \ duplicate counter
  nells my + n!               \ store counter in array
  1+                          \ increment counter
  dup /my =                   \ limit reached?
until drop                    \ drop the counter

my                            \ set up index
begin
  dup n@ . cr                 \ print the value
  nell+                       \ next element
  dup my /my nells + =        \ limit reached
until drop                    \ drop the index
```

You see that the code is *very* similar. The 'STRING' declaration clearly indicates that the array is allocated in the Character Segment. But as you can see it is not an array of *cells*, but an array of *nells*. 'NELL' holds the size of a single nell, so we multiply it by the number of nells we want to get the proper size of the array. After that, it is just replacing the Integer Segment words with nell equivalents:

NELL	CELL
/nell	/cell
nells	cells
n@	@
n!	!
nell+	cell+
nell-	cell-

Table 10.1: NELL equivalents

Note that although you can replace every cell with a nell, you *do* pay a penalty in execution speed, so use with caution.

10.7 Enumerations

Sometimes you need a lot of constants:

```
0 constant Monday
1 constant Tuesday
2 constant Wednesday
3 constant Thursday
4 constant Friday
5 constant Saturday
6 constant Sunday
```

A little error here may ruin your program. This does the very same thing, except it is easier to maintain:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  enum Sunday   drop
```

'ENUM' is much like a 'CONSTANT', but increments and leaves a value after the constant has been created. That is why we need to add 'DROP' after the final enumeration. To show you that 'ENUM' and 'CONSTANT' are much alike, you could also write the declaration above as:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  constant Sunday
```

Since 'CONSTANT' just consumes the value, you don't need the final 'DROP'.

10.8 Dynamic memory allocation

If you don't know what this is, you probably shouldn't bother. Sometimes you don't know how much memory you will actually need, sometimes you know how much you need, but you won't need it during the entire execution of the program. In these cases, you can temporarily allocate a chunk of memory and release it when you no longer need it.

4tH has similar facilities. E.g. if you want to allocate 600 bytes, you simply include "ansmem.4th" and allocate it:

```
[needs lib/ansmem.4th]
600 allocate
```

'ALLOCATE' leaves two items on the stack. The first one is a flag. If it is true, memory allocation has failed, so we can easily add some error checking to our little program:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
```

It it returns false, memory has been allocated. Its address is the second item on the stack. You can pretty much do what you want with it, but remember that memory is always allocated in the Character Segment, so if you want to store numbers over there, read section 10.6 again. Anyway, this is completely valid:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
s" Hello temporary world!" rot place
```

Let's change that one a little bit to prove we've actually stored anything:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
>r s" Hello temporary world!" \ Let's save the address
r@ place                      \ Now store the string
r> count type cr              \ Let's print the string
```

Let's allocate another 100 bytes and free all memory afterwards:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

100 allocate
abort" Out of memory" >r      \ Second allocation
s" I'm a little crammed!"
r@ place                      \ Store another string
r@ count type cr              \ Let's print the string

r> free
abort" Cannot free memory"    \ Now free the first block
r> free
abort" Cannot free memory"    \ Now free the second block
```

Yes, that's right: you feed 'FREE' the address that 'ALLOCATE' returned and it returns a flag. If it is a true flag, an error occurred; if not, everything is hunky dory. Let's try to free it twice:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r@ free
abort" First attempt"         \ Now let's free the block
r> free
abort" Second attempt"        \ And try to free it again..
```

Yes, now 4tH terminated with the error message ”Second attempt”. You can not free a block twice..! But you can reallocate it if you happen to change your mind. You can increase or decrease its size, without losing any data. When the new block is too small to hold all the data, the data is truncated. Let’s see it in action:

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r> 100 resize
abort" Out of memory" >r      \ Now resize the block
r@ count type cr              \ Here is your string again

r> free
abort" Cannot free memory"    \ Now free it
```

You’ll see that your precious string is still alright. Apart from a flag, ’RESIZE’ also returns the address of the reallocated block. If ’RESIZE’ fails, *your original data is still alright*, so in some circumstances you might want to save the old address. Sometimes ’RESIZE’ fails, even when you’re decreasing the size of a block. Well, ’RESIZE’ *always allocates a new block*, so when memory is low or fragmented it may not succeed.

10.9 Tweaking dynamic memory

You might find that 4tH doesn’t reserve much memory for dynamic allocation. Dynamic memory is allocated at the heap, which is 16 kB. You can increase it, but first you have to know how dynamic memory works. You can determine how much memory has been allocated by using the word ’ALLOCATED’:

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

And it will print something like:

```
768 allocates 64 bytes.
```

64 bytes? I thought we allocated 50 bytes! Let’s try another one:

```
[needs lib/ansmem.4th]
500 allocate
```

```
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

This time it prints something like:

```
768 allocates 512 bytes.
```

As a matter of fact, 'ALLOCATED' will *always* return multiples of 64 bytes. That is a consequence of how 4tH handles dynamic memory. 4tH divides dynamic memory into *fragments*. When you allocate memory, 4tH allocates as much fragments as it needs to provide you with the memory you requested. Then these fragments are marked as 'taken'. This marking is done in the Heap Allocation Table, which is located in the Integer Segment. Every fragment is represented by a cell in the HAT.

You can fine-tune this mechanism by defining some constants before including "ansmem.4th". This will create a heap with 512 fragments of 256 bytes, which is 128 kB:

```
512 constant #heap            \ 512 fragments
256 constant /heap            \ each fragment is 256 bytes
[needs lib/ansmem.4th]
500 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

Try to keep the number of fragments *low*. 1024 seems like a nice upper limit. If you need that much memory, it is *much* better to handle it in larger chunks. This avoids fragmentation and keeps the time to search the HAT within acceptable limits.

10.10 Application stacks

Did you ever feel like a second return stack would be nice? Well, you can. As a matter of fact you can have several dedicated stacks. It's quite easy to use:

```
[needs lib/stack.4th]

16 array mystack              \ allocate some space
mystack stack                 \ convert it into a stack

234 mystack >a                \ push 234 on the stack
456 mystack >a                \ push 456 on the stack
mystack a@ . cr               \ examine top of stack
mystack a>                    \ pop 456 from the stack
mystack a>                    \ pop 234 from the stack
. . cr                        \ show the values
```

Wouldn't it be nice to have a string stack too? Yes, 4tH provides that one too! It works the same way:

```
[needs lib/stsstack.4th]

1024 string mystack           \ allocate some space
mystack string-stack          \ convert it to a string stack

s" Hello" mystack >s          \ push string 'Hello' on stack
s" World" mystack >s          \ push string 'World' on stack
mystack s@ type cr            \ examine top of stack
mystack s>                    \ pop 'World' from the stack
mystack s>                    \ pop 'Hello' from the stack
type cr type cr               \ show the values
```

Note there is a catch: when you've popped a string from the string stack, the string itself is untouched, so the address-count pair is still valid. However, if you push another string onto the same stack, the popped string is clobbered. There is another way to create a string stack without these disadvantages, but it is slightly larger and slower. It is initialized by:

```
[needs lib/strstack.4th]

1024 constant /mystack        \ allocate some space
/mystack string mystack
mystack /mystack string-stack \ convert it to a string stack
```

All other words work the same way.

10.11 Forward declarations

It doesn't happen very often, but sometimes you have a program where two colon-definitions call each other. When you look at 4tHs source you find several examples. The throw() function calls the rpop() function, because 'THROW' takes items from the Return Stack. On the other hand, when the Return Stack underflows, it has to call 'THROW'.

There is a special instruction in 4tH to do this, called 'DEFER'. 'DEFER' doesn't create an executable word, but a vector containing an execution token, which is executed when called. You might want to consult section 9.3 first to see how this works. But for all purposes you might consider it to be an executable word, because it behaves the same way.

```
defer Step2
```

Now we can create "STEP1" without a problem:

```
: Step1 1+ dup . cr Step2 ;
```

But "STEP2" does not have a body yet. Of course, you could create a new colon-definition, tick it and assign the execution token to "STEP2" manually, but it is much neater to use ':NONAME'. ':NONAME' can be used like a normal ':', but it doesn't require a name. Instead, it pushes the execution token of the colon-definition it created on the stack. No, ':NONAME' does *not* create a literal expression, but it is just what we need:

```
:noname 1+ dup . cr Step1 ; is Step2
```

Now we are ready! We can simply execute the program by calling "STEP1":

```
1 Step1
```

Note that if you run this program, you'll get stack errors! Sorry, but the example has been taken from a Turbo Pascal manual ;-). If you have forgotten what a deferred word actually executes, you can retrieve the execution token by using 'DEFER@':

```
defer thisword              \ create a vector

: plus + ;                  \ define a word

' plus is thisword          \ assign the word to the vector
' thisword defer@           \ retrieve the execution token
2 3 rot execute             \ execute the deferred word
. cr                        \ display the result
```

As a matter of fact, this expression:

```
' thisword defer@ execute
```

Is equivalent to this one:

```
thisword
```

You can also reassign a vector without using 'IS'. 'IS' is a parsing version. That means the actual vector to which a certain behaviour is assigned is determined at *compiletime*. 'DEFER!' can be used to assign a certain behaviour at *runtime*. 'DEFER!' takes two execution tokens:

```
defer thisword              \ create a vector

: plus + ;                  \ define a word

' plus ' thisword defer!    \ assign it to a vector
```

This is equivalent to this:

```
defer thisword              \ create a vector

: plus + ;                  \ define a word

' plus is thisword          \ assign it to a vector
```

I guess you'll agree with me that this creates countless possibilities.

10.12 Recursion

Yes, but can she do recursion? Of course she can! It is even very natural and easy. Everybody knows how to calculate a factorial. In 4tH you can do this by:

```
: factorial                   ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    factorial *
  then
;

10 factorial . cr
```

Which is exactly as one would expect. Unfortunately, this is not the way it is done in ANS-Forth. In order to let a colon-definition call itself, you have to use the word 'RECURSE'. 4tH supports this word too:

```
: factorial                   ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    recurse *
  then
;

10 factorial . cr
```

It will even compile to the same code. If you use the word 'RECURSE' outside a colon-definition, the results are undefined. Note that recursion lays a heavy burden on the return stack. Sometimes it is wiser to implement such a routine differently:

```
: factorial
  dup
  begin
    dup 2 >
  while
    1- swap over * swap
  repeat

  drop
;

10 factorial . cr
```

So if you ever run into stack errors when you use recursion, keep this in mind.

10.13 Lookup tables with integer keys

No CASE construct, huh? Now how are we supposed to make those complex decisions? Well, do it the proper way. Leo Brodie wrote: "I consider the case statement an elegant solution to a misguided problem: attempting an algorithmic expression of what is more aptly described in a decision table". And that is exactly what we are going to teach you.

Let's say we want a routine that takes a number and then prints the appropriate month. In ANS-Forth, you could do that this way:

```
: Get-Month
  case
     1 of ."  January " endof
     2 of ." February " endof
     3 of ."   March  " endof
     4 of ."   April  " endof
     5 of ."    May   " endof
     6 of ."   June   " endof
     7 of ."   July   " endof
     8 of ."  August  " endof
     9 of ." September" endof
    10 of ."  October " endof
    11 of ." November " endof
    12 of ." December " endof
  endcase
  cr
;
```

This takes a lot of code and a lot of comparing. In this case (little wordplay) you would be better of with an indexed table, like this:

```
create MonthTable
  ,"  January "
  ," February "
  ,"   March  "
  ,"   April  "
  ,"    May   "
  ,"   June   "
  ,"   July   "
  ,"  August  "
  ," September"
  ,"  October "
  ," November "
  ," December "

: Get-Month                   ( n -- )
  12 min 1- MonthTable @c count type cr
;
```

Which does the very same thing and will certainly work faster. True, you can't do that this easily in ANS-Forth, but in 4tH you can, so use it! The word ',"' compiles a string, whose address can be retrieved by '@C' as if it were a numeric constant. Note that '@C' just returns the *address* of the string, so you have to use 'COUNT' to obtain an address/count pair. Of course, there is also an equivalent to ',"' called ',|'. The latter is delimited by a bar instead of a quote, but essentially works the same way.

But can you use the same method when you're working with a random set of values like "2, 1, 3, 12, 5, 6, 4, 7, 11, 8, 10, 9". Yes, you can. But you need a special routine to access such a table. Of course we designed one for you. It is called ”ROW” and you can use it by adding this directive:

```
[needs lib/row.4th]
```

This routine takes three values. The first one is the value you want to search. The second is the address of the table you want to search. And on top of the stack you’ll find the number of fields this table has. The first field must be the "index" field. It contains the values which have to be compared. That field has number zero.

This routine can search zero-terminated tables. That means the last value in the index field must be zero. Finally, it can only lookup positive values. It returns the value you searched, the address of the row where it was found and a flag. If the flag is false, the value was not found.

Now, how do we apply this to our month table? First, we have to redefine it:

```
create MonthTable
  1 ,  ," January "
  2 ,  ," February "
  3 ,  ,"   March  "
  4 ,  ,"   April  "
  5 ,  ,"    May   "
  6 ,  ,"   June   "
  7 ,  ,"   July   "
  8 ,  ,"  August  "
  9 ,  ," September"
  10 , ,"  October "
  11 , ," November "
  12 , ," December "
  NULL ,
```

Note that this table is sorted, but that doesn’t matter. It would work just as well when it was unsorted. Let’s get our stuff together: the address of the table is "MonthTable", it has two fields and we want to return the address of the string, which is located in field 1. Field 0 contains the values we want to compare. We can now define a routine which searches our table:

```
: Search-Month                ( n1 -- n2 f)
  [’] nkey= is key=           \ set ROW datatype
  MonthTable 2 row            \ search the table
  dup >r                      \ save flag
  if nip cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

Because ”ROW” is able to search integer tables and string tables, you have to define which one it is[1]. This expression tells ”ROW” we’re searching for an integer value:

```
[’] nkey= is key=
```

[1] Note that some 4tH libraries, like ’environ.4th’ and ’interprt.4th’ use ”ROW” as well. If you experience any unexpected results you should reassign ”KEY=” before invoking ”ROW”.

Now, we define a new "Get-Month" routine:

```
: Get-Month                    ( n --)
  Search-Month                 \ search table

  if                           \ if month is found
    count type                 \ print its name
  else                         \ if month is not found
    drop ." Not found"         \ drop value
  then                         \ and show message
  cr
;
```

Is this flexible? Oh, you bet! We can extend the table with ease:

```
3 Constant #MonthFields

create MonthTable
  1 ,  ,"  January " 31 ,
  2 ,  ," February " 28 ,
  3 ,  ,"   March  " 31 ,
  4 ,  ,"   April  " 30 ,
  5 ,  ,"    May   " 31 ,
  6 ,  ,"   June   " 30 ,
  7 ,  ,"   July   " 31 ,
  8 ,  ,"  August  " 31 ,
  9 ,  ," September" 30 ,
  10 , ,"  October " 31 ,
  11 , ," November " 30 ,
  12 , ," December " 31 ,
  NULL ,
```

Now we make a slight modification to "Search-Month":

```
: Search-Month                 ( n1 -- n2 f)
  ['] nkey= is key=            \ set ROW datatype
  MonthTable #MonthFields row \ search the table
  dup >r                       \ save flag
  if nip cell+ @c else drop then
  r>                           \ if found get value
;                              \ if not drop address
```

This enables us to add more fields without ever having to modify "SearchMonth" again. If we add another field, we just have to modify "#MonthFields". We can now even add another routine, which enables us to retrieve the number of days in a month:

```
: Search-#Days                 ( n1 -- n2 f)
  ['] nkey= is key=            \ set ROW datatype
  MonthTable #MonthFields row \ search the table
  dup >r                       \ save flag
  if nip cell+ cell+ @c else drop then
  r>                           \ if found get value
;                              \ if not drop address
```

Of course, there is room for even more optimization, but for now we leave it at that. Do you now understand why 4tH doesn't have a CASE construct?

10.14 Lookup tables with string keys

But what if the table we're using looks like this:

```
create MonthTable
  ," January" 31 ,
  ," February" 28 ,
  ," March" 31 ,
  ," April" 30 ,
  ," May" 31 ,
  ," June" 30 ,
  ," July" 31 ,
  ," August" 31 ,
  ," September" 30 ,
  ," October" 31 ,
  ," November" 30 ,
  ," December" 31 ,
  NULL ,
```

Sure, 4tH compiled some kind of integer value there, but an address to a string is less than helpful. We have to compare *strings* in order to find the correct entry, not *addresses*. So, we need a word that searches the table and returns the contents of the field that follows the appropriate string. Well, of course there is such a word. It is "ROW" again. You can use it by entering:

```
[needs lib/row.4th]
```

At the *beginning* of your program. "ROW" takes an address/count pair of the string that has to be found, the address of the table it has to search for that string and the number of fields the table has. It returns the original address/count pair of the string, the address of the row where the search stopped and a flag. That makes it quite a useful word, e.g. how many days has June:

```
: GetDays                    ( a n --)
  ['] skey= is key=          \ set ROW datatype
  MonthTable 2 row           \ search the table
  if
    cell+ @c . drop drop     \ if found, display the number of days
  else                       \ else an error message
    drop type ."  is not a month!"
  then
  cr
;

s" June" GetDays
```

If "ROW" returns true, the value was found. If it returns false, it wasn't. Note you have to indicate which datatype "ROW" has to deal with. "ROW" is quite versatile, but that is not the only merit of "ROW" as we will see in the next sections.

10.15 Lookup tables with multiple keys

Some tables have multiple keys to search them, e.g. by name or by number. So far all tables we've seen dealt with a single key in the first column. It would be a shame if you had to split a table into two tables, simply because you had two different ways to access it. Fortunately, "ROW" can handle this kind of tables as well as long as you put the key columns up front and add a NULL at the end of the table for every key, e.g.

```
create mytable
  ," Monday"      1 ,
  ," Tuesday"     2 ,
  ," Wednesday"   3 ,
  ," Thursday"    4 ,
  ," Friday"      5 ,
  ," Saturday"    6 ,
  ," Sunday"      7 ,
  NULL , NULL ,
```

This table consists only of key fields. You can search for the name and get a number or search for the number and get the equivalent name. The trick is to keep in mind what the key field is and the relative position of the datafield. In this case we want to search on number, so the corresponding name is the field *before* the key field. When you start the search the pointer you pass to "ROW" has to point to the key field you want to search. In this case that is equivalent to:

```
mytable cell+
```

Let's assume we want to search this table both ways:

```
: day>num                    ( a1 n1 -- a1 n1 -f | n2 f)
  ['] skey= is key=
  mytable 2 row dup >r
  if nip nip cell+ @c else drop then r>
;

: num>day                    ( n1 -- n1 -f | a1 n2 f)
  ['] nkey= is key=
  mytable cell+ 2 row dup >r
  if nip cell- @c count else drop then r>
;
```

The first word doesn't hold any surprises. It is a vanilla search word. The second one passes a slightly modified pointer to "ROW" and decrements the address it returns, so it now points to the name field. We can use both words quite easily and transparently:

```
s" Friday" day>num if . else type ."  not found" then cr
s" New yearsday" day>num if . else type ."  not found" then cr

5 num>day if type else . ." not found" then cr
8 num>day if type else . ." not found" then cr
```

You will see they work as expected.

10.16 Lookup tables with duplicate keys

Although most tables come with unique keys you may find yourself in a situation where you have to resume a search. ”ROW” can handle that situation as well. Let’s examine this table:

```
create people
  ," Ritchie"  ," Lionel"
  ," Dijkstra" ," Edsger"
  ," Moore"    ," Henri"
  ," Ritchie"  ," Dennis"
  ," Wirth"    ," Nick"
  ," Hopper"   ," Grace"
  ," Moore"    ," Chuck"
  ," Hopper"   ," Dennis"
  NULL ,
```

There is one key field, since the table is terminated with only one NULL. We also find multiple Hoppers and Moores, so we can’t be sure we’ve found the right one right away. In order to get that one we might have to continue our search. That is exactly what this program does:

```
: >surname 2 row ;            ( a n x1 -- a n x2 f)
: first? rot cell+ @c count compare 0= ;
: >next cell+ cell+ >surname ;
: .name type space type ;     ( a1 n1 a2 n2 --)

: >first                      ( a1 n1 x a2 n2 --)
  2>r                         ( a n x)
  if                          ( a n x)
    dup 2r@ first?            ( a n x f)
    if                        ( a n x)
      drop ." Found " 2r> .name cr
    else                      ( a n x)
      >next 2r> recurse       ( a n x a n)
    then                      ( --)
  else                        ( a n x)
    drop 2r> .name ."  not found" cr
  then                        ( --)
;

: >name -rot 2>r >surname 2r> >first ;

: demo
  s" Ritchie" s" Dennis" people >name
  s" Moore" s" Chuck" people >name
  s" Lovelace" s" Ada" people >name
;

['] skey= is key=             \ set ROW datatype
demo                          \ run the demo
```

”>NAME” is a wrapper around this programs most important words ”>SURNAME” and ”>FIRST”. ”>SURNAME” simply searches for a given surname in the table and returns its address. ">FIRST" takes over and compares the first name. If it checks out we're done, if not it calls ">NEXT". ">NEXT" increments the pointer, so it now points to the next row. Then it calls ">SURNAME" again, effectively continuing the search. Finally ">FIRST" calls itself to check the first name again. Depending on the contents of the table, this process can be repeated several times.

10.17 Interpreters

Those of you who know Forth will be very surprised to see that 4tH doesn't have a Forth prompt. Some will be even more surprised to see that 4tH does have an interpreter. It is a library routine, written in 4tH, that can easily be adapted and expanded. If you can write 4tH and maintain a table, you can use it. The next question you have to ask yourself, is do you want your interpreter to be case sensitive or not? If it is, "id" will work, but "Id" or "ID" will not. If you want it to be case sensitive, change the constant "ignorecase" to "false". Example:

```
false constant ignorecase    \ don't ignore case
[needs lib/interpret.4th]

: _+ + ;
: _. . ;
: id ." This is 4tH" cr ;
```

Well, that isn't very hard, is it. Now we add a table to all that:

```
create wordlist
  ," +"  ' _+ ,
  ," ."  ' _. ,
  ," id" ' id ,
  NULL ,
```

Remember to terminate your table with "NULL"! Every entry consists of a string and an address to your routine. What will happen is that your user enters the string and the appropriate routine will be called. In this case, your interpreter has three commands: "+", "." and "id". We're a hair away from a real interpreter. We just have to assign our table to the dictionary. These lines do the job:

```
wordlist to dictionary
refill drop interpret
```

Now you can compile your application and run it. Enter:

```
45 12 + .
```

And it will print:

```
57
```

Yes, it's just as easy as that! If you enter something the interpreter doesn't recognize it will try to convert it to a number and throw it on the stack. But you will also see that it exits after you've entered that single line. That is because the interpreter is called just once. If you change that to:

```
begin refill drop interpret again
```

It will return with an new prompt. In that case it is wise to add a routine like:

```
: _quit quit ;
```

And add it to your interpreter, because otherwise your user will not be able to leave the application. Note that you have to do all the error-checking. E.g., if your user calls "_+" without putting sufficient items on the stack, 4tH will exit with an error. Of course, you can catch any exceptions. ”INTERPRET” has a builtin word, ”NotFound”, that deals with any unrecognized strings. You can define your own if you want to. The only thing you have to do is to write a word which takes an address/count string and returns nothing, e.g.:

```
:noname 2drop ." I don't understand this!" cr ; is NotFound
```

Or more elaborate:

```
:noname ." I don't what '" type ." ' means!" cr ; is NotFound
```

You could even integrate it with the exception trapping, if you defined one:

```
1 constant #UndefName

:noname #UndefName throw ; is NotFound
```

When a word is not found, a user exception is thrown. This example is taken from "dc.4th":

```
: dc
  begin                        \ main interpretation loop
    ." OK" cr                  \ print prompt
    refill drop                \ get input from use
    ['] interpret              \ interpret it
    catch dup                  \ catch any errors
    if                         \ if one occurred
      ShowMessage              \ show a message
    else                       \ otherwise
      drop                     \ drop the throw code
    then
  again                        \ loop back
;
```

You can still see the basic structure, but this one is much more advanced. You can also remove the code from the interpreter that decodes numbers. In that case, if a word is not found in the "dictionary" table it will exit immediately and report an error:

```
true constant ignorenumbers
```

Don't let anybody ever tell you you can't make interactive applications with 4tH. As you have seen, you can with very little effort.

10.18 Adding your own library

This is a lot easier than you might think! As a matter of fact, almost any program can be turned into a specialized library. A well-written program contains a lot of definitions and only *one* executable word. Take that word away and you've got a library!

A library may contain word defintions, variables, constants, almost anything you like. And a program that includes that library will have all these definitions at its disposal. As a matter of fact, the resulting program will behave like you entered the contents of the entire library file at the position of the '[NEEDS' directive, e.g. these are the contents of "null.4th":

```
-1 constant NULL
```

When it is included in this file:

```
\ This is a sample table using NULL

[needs lib/null.4th]

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

It will compile to the same code as this:

```
\ This is a sample table using NULL

-1 constant NULL

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

So it is *not* a good idea to make your library files too big, since there will be a lot of superfluous code included in the compilant which 4tH will *not* dispose of automatically.

You can nest '[NEEDS' directives, so one library file may include other library files. This helps to prevent duplicate code, which can be a serious maintenance problem. You can nest them as deep as you want, available memory being the only restriction.

However, when nesting inclusions you always have the problem of multiple inclusions. Don't think that all 4tH users know by heart which library files calls which. Multiple inclusions will lead to errors, unless you take precautions. We have '[DEFINED]' and '[UNDEFINED]' to prevent that:

```
[UNDEFINED] 2drop [IF]
: 2drop drop drop ;
: 2dup over over ;
: 2swap rot >r rot r> ;
[THEN]
```

If you have included this file before, '2DROP' is already defined, so in fact all definitions are skipped when the file is included for the second time. Of course, it will take up some extra memory, but at least it won't generate any errors.

If you want to port your library file, it might be a good idea to hide specific 4tH constructions, e.g.:

```
[DEFINED] 4TH# [IF]             ( a n --)
: string! chars + 0 swap c! ;
[THEN]                          \ make an ASCIIZ string

[UNDEFINED] 4TH# [IF]
: string! swap 1- c! ;          ( a n --)
[THEN]                          \ make a counted string
```

Since '4TH#' is a 4tH specific constant, it will not be defined in other Forth compilers. This way the compiler will automatically select the correct definition.

Where you place your library files is up to you. You can add them to the library files that come with 4tH, you can put them in another directory, whatever pleases you.

10.19 Adding templates

When you include a library file you add some words to your program. When you include a template you add some words to an existing program. That is the major difference between a library file and a template file. We've included a template with 4tH which allows you to create conversion program pretty quickly. The template is called "convert.4th" and it allows you to create a conversion program by defining just three words.

A standard conversion program takes an input file and creates an output file in a different format. When it can't open a file it will issue an error message, e.g.

```
Cant open input.txt
```

When you don't supply an input file and an output file, it will issue an error message e.g.:

```
Usage: myconversion input output
```

And of course, it will read and process the input file. And that's all you have to tell 4tH:

- The usage message
- How to read the file
- How to process the file

So, let's create a program that will convert a block file to a regular text file. How do we do that? First of all we've got to issue a usage message, like:

```
Usage: blk2txt blockfile textfile
```

Well, that is easy. If it comes to that we've got to abort the program, so this will do:

```
: Usage abort" Usage: blk2txt blockfile textfile" ;
```

Then we've got to read the file. A block file contains lines of 64 characters, always. So, we've got to create a buffer and read 64 characters. This will do:

```
64 string buffer
: Read-file buffer 64 accept ;
```

Finally, we've got to write the output file. Adding a 'CR' after typing the line will do, but we don't want any trailing spaces, so we need to strip those trailing spaces:

```
: Process buffer 64 -trailing type cr ;
```

Now we need to include the template and we're done:

```
[needs lib/convert.4th]
```

Wow! Do you know how much coding we need to do when we try to do this in C? This source code takes less than 256 bytes! Compile it and we're done! So how does it work? Well, the template expects us to define "Usage", "Read-file" and "Process". If you don't it will abort compilation:

```
\ Has Usage been defined? If not, abort!
[DEFINED] Usage [NOT] [IF]
[ABORT] [THEN]

\ Has Read-File been defined?
[DEFINED] Read-File [NOT] [IF]
[ABORT] [THEN]

\ Has Process been defined?
[DEFINED] Process [NOT] [IF]
[ABORT] [THEN]
```

Furthermore, you can optionally define "PreProcess" and "PostProcess" if you need anything at the top of the file or the bottom of the file:

```
: ProcessFile                       \ process the input file line by line
[DEFINED] PreProcess [IF]
  PreProcess                        \ do any preprocessing
[THEN]
  begin
    Read-file                       \ read the file
  while
    Process                         \ process the line or buffer
  repeat
[DEFINED] PostProcess [IF]
  PostProcess                       \ do any postprocessing
[THEN]
;
```

If you don't define it, it won't include it. You can use such templates for many different programs, e.g. this will convert a Unix text file to an MS-DOS text file:

```
: Usage abort" Usage: udc infile outfile " ;
: Read-file refill ;
: Process 0 parse-word type 13 emit 10 emit ;
```

You can make them as sophisticated or as simple as you like. You can create other words as well, as long as those three words have been defined. Templates can be handled like any other library file. You can place them where you want, they can hold anything you want. Amaze your collegues by writing programs in a fraction of the time they should need!

10.20 Private declarations

Sometimes you want to hide some definitions from other programmers. This is especially true when you're writing libraries or templates. The Application Programmers Interface must be public of course, but you don't want anyone else to tinker with the internals of your library. And there is the problem of cluttering your name space.

Relax, 4tH has a way to get rid of these internal words. It's easy, just tell 4tH to hide them:

```
VARIABLE #emits                       \ private

: SHOW emit 1 #emits +! ;             \ public
: NL CR 0 #emits ! ;                  \ public

hide #emits
```

After that the name "#EMITS" is no longer recognized and can be reused if you want to, e.g. this is completely valid:

```
: dummy ;
hide dummy
: dummy ." I am no longer a dummy!" cr ;
```

As a matter of fact, the previous declaration of "DUMMY" has been turned into a ':NONAME' declaration by the use of 'HIDE'.

10.21 Aliases

Sometimes you want to make an alias for a word. Of course you can embed the word you want to alias in a new definition:

```
: noop ;
: nop noop ;
```

Although this approach works perfectly under all circumstances it has its disadvantages, because calling a word is relatively slow. Unless you're trying to make an alias for an internal word, you'd better use an 'ALIAS ':

```
: noop ;
' noop alias nop
```

This is completely equivalent to:

```
defer nop
: noop ;
' noop is nop
```

Although the vector takes up a little space, it will save you from most of the calling overhead. Since you can only alias self-defined executable words, 'ALIAS' is quite limited. 'AKA' does not have that disadvantage:

```
: noop ;
aka noop nop
```

Both words are now completely equivalent and compile to exactly the same code. Even better, you can use 'AKA' with *every* self-defined word, including variables, vectors, files, values, fields and constants. 'AKA' is also known as "also known as".

10.22 Changing behaviour of data

One of the most ingenious things Forth can do, is change the behaviour of data at runtime. With 4tH, you cannot do this for an entire datatype, but you can do it for individual 'VARIABLE's, 'CREATE's, 'STRING's, 'ARRAY's and 'CONSTANT's. Just use:

```
:THIS <name> DOES> <definition> ;
```

Where <name> is a previously defined 'VARIABLE', 'STRING', etc. The 'DOES>' word is optional. The definition will behave as if the 'VARIABLE', etc. has just been thrown on the stack, e.g. to make a 'VARIABLE' behave as a 'CONSTANT 'you define:

```
variable me

10 me !
:this me does> @ ;
```

The body of the definition will behave as if it said:

```
me @
```

Which boils down to a (rather slow) constant. You cannot change the contents of the variable anymore if you haven't taken precautions, because there is no way to address it. Here is another, more elaborate example:

```
create life                    \ create an array of string constants
  ," This is my life!"
  ," This is your life!"
```

```
0 constant my                \ create two constants
1 constant your              \ to address the elements
                             \ now change the behaviour of LIFE
:this life does> swap th @c count type cr ;

my life                      \ use it!
your life
```

At runtime, this will print:

```
This is my life!
This is your life!
```

Wording has always been very important to Forth. Using this technique, you can make your programs even more readable.

10.23 Multidimensional arrays

We've seen two dimensional arrays with 'ARRAY' and 'STRING', but what about multi-dimensional arrays. Well, it's the same thing all over again. C doesn't actually have multi-dimensional arrays either. When you define one, just a chunk of memory is allocated.

In 4tH you can do the same thing, but now you have to do it yourself. E.g. when you want to define a matrix of cells of 4 rows by 5 elements, you have to multiply those and allocate an array of that size:

```
4 5 [*] array my_array
```

But what if you want to reference the fourth element of the third row? You *cannot* write something like:

```
2 3 my_array @
```

That's right. But you can change the behaviour of "MY_ARRAY" accordingly:

```
:this my_array               ( n1 n2 -- a)
  does>
  rot 5 *                    \ calculate row offset
  rot +                      \ calculate element offset
  cells                      \ calculate number of cells
  +                          \ add to address of my_array
;
```

This word calculates the correct offset for you. Note that the third row is row number two (since we start counting from 0) and the fourth element is element number three:

```
2 3 my_array @
```

You can also use "MY_ARRAY" to initialize an array, since it simply calculates the correct address for you:

```
5 2 3 my_array !            \ sets 3rd row 4th element to 5
```

You can add more dimensions if you want. This works basically the same way: create an array of a size that equals the products of its dimensions and design a word that calculates the correct address.

10.24 Binary string constants

A binary string constant is an unterminated string that doesn't necessarily contain characters. Creating binary string constants is easy. Just compile them by their ASCII value into the String Segment with 'C,':

```
char H c, char i c, char ! c, 0 c,
```

The fun of it all is that 4tH doesn't allow you to access the String Segment directly, so you can *never* retrieve them. You need 'OFFSET' to define a word which does all the hard work for you. At runtime it takes an index and leaves the ASCII value of the character in question on the stack. 'OFFSET' is used just before you compile the ASCII values:

```
offset greet char H c, char i c, char ! c, 0 c,
```

Note that you have to terminate a binary string constant *manually* if you need to, although it is perfectly legal to create binary string constants with no termination at all. Retrieving characters is easy. This will print "Hi!":

```
0 greet emit
1 greet emit
2 greet emit cr
```

And so will this:

```
0 begin                     \ setup index
  dup greet dup             \ retrieve character
while                       \ if not terminated
  emit 1+                   \ emit and increase index
repeat drop drop            \ clear stack
```

You can use binary string constants for compact tables, bitstrings or any other raw data as long as each element doesn't exceed the size of a single character.

10.25 Records and structures

The easiest way is to allocate a structure in the Character Segment. Just define the structure like this:

```
struct
  32 +field Name
  64 +field Address
  32 +field City
  12 +field Age
end-struct /Person
```

This might be a familiar example to you. We'll store information on a single person in this structure. Now we got the fields, the length of the fields and the length of the entire structure, stored in "/Person". Both the fields and the entire structure are nothing more than a set of constants, e.g. he offset of the field "Name" is stored in a CONSTANT named "Name". However, we still haven't allocated any memory. We can allocate room for the structure we've just defined by just using the word 'STRING'. Note that you can also create a cell-based structure. Then you need the word 'ARRAY' to allocate the memory required.

```
/Person string Person
```

Now we can define a word which initializes the fields:

```
: InitRecord                  \ initialize fields
  s" Hans Bezemer" Person -> Name    place
  s" Lagendijk 79" Person -> Address place
  s" Den Helder"   Person -> City    place
  s" 44"           Person -> Age     place
;
```

Of course, you can also use 'ACCEPT' to enter the contents of the fields. Fields act like ordinary strings. Note that numbers are stored as strings as well. This is not too much of a problem since 'NUMBER' can convert them back to numbers anyway.

This is a very simple use of structures. You can also use structures within structures:

```
struct
  64 +field Address
  32 +field City
end-struct /Location

struct
  32 +field Name
  /Location field Location
end-struct /Person

/Person string Person

s" Delft" Person -> Location -> City place
```

If you want to make an array of structures, that can be done as well:

```
struct                      \ create structure
  32 +field Name
  64 +field Address
  32 +field City
  12 +field Age
end-struct /Person

32 constant #Person         \ size of array of structs
                            \ now allocate the room
#Person /Person [*] string Persons
                            \ make it behave properly
```

```
:this Persons does> swap /Person * + ;
                            \ initialize the first record
s" Hans Bezemer" 0 Persons -> Name    place
s" Lagendijk 79" 0 Persons -> Address place
s" Den Helder"   0 Persons -> City    place
s" 44"           0 Persons -> Age     place
```

You can also extend an already existing structure:

```
struct                      \ create structure
  32 +field Name
  64 +field Address
  32 +field City
  12 +field Age
end-struct /Person
                            \ now extend the structure
/Person
  32 +field Job
  16 +field Emp-number
end-struct /Employee
```

You now got two different structures, "/Person" and "/Employee", that share the first four fields. Well, if that isn't a complete implementation, I don't know what is..

10.26 Fixed point calculation

We already learned that if we can't calculate it in dollars, we can calculate it in cents. And still present the result in dollars using pictured numeric output:

```
: currency <# # # [char] . hold #s [char] $ hold #> type cr ;
```

In this case, this:

```
200012 currency
```

Will print this:

```
$2000.12
```

Well, that may be a relief for the bookkeepers, but what about us scientists? You can do the very same trick. We have converted some Forth code for you that gives you very accurate results. You can use routines like SIN, COS and SQRT. A small example:

```
[needs lib/math.4th]

45 sin . cr
```

You will get "7071", because the result is multiplied by 10000. You can correct this the same way you did with the dollars: just print the number in the right format. You can also use a delightful little library created by Leo Brodie called "fraction.4th". This library allows you to do arithmetic in the range of -13.1072 and 13.1071 with a precision of 0.0001! It seamlessly integrates with the "math.4th" library:

```
[needs lib/math.4th]
[needs lib/fraction.4th]

45 sin s>v v. cr
```

This one will actually print:

```
0.7071
```

Of course, you can also use it with other math words, as long as they are properly scaled. The scaling constant is called '10K' and is available from "constant.4th". The nice thing about fractions is that you can do all basic math (within its range, of course) without bothering about the decimal point. To convert a fixed point number to a fraction, you have to multiply it by '10K', which is 10,000 and call 'S>V', e.g. in order to convert '0.7071' to a fraction you need to do this:

```
7071 s>v
```

You can also define a fraction yourself, e.g. this converts two-thirds to a fraction:

```
2 3 v/
```

You can add or subtract fractions, e.g.:

```
7071 s>v 2 3 v/ - v. cr
```

This is equivalent to:

```
PRINT 0.7071 - 0.6666
```

You can multiply or divide fractions, e.g.:

```
7071 s>v 2 3 v/ v* v. cr
```

This is equivalent to:

```
PRINT 0.7071 * 0.6666
```

Since a fraction is still a single-cell number, you can manipulate the stack in the usual way. In some cases you can even mix cells and fractions, as the following table will show you:

With the word 'V>S' you can convert a fraction back to a '10K' scaled fixed point number. You can print a fraction using the word 'V.'.

Another example is SQRT from "math.4th". If you enter a number of which the root is an integer, you will get a correct answer. You don't even need a special formatting routine. If you enter any other number, it will return only the integer part. You can fix this by scaling the number.

However, scaling it by 10 will get you nowhere, since "3" is the square root of "9", but "30" is not the square root of "90". In that case, we have to square the scale itself; we need 100, 10,000 or even 1,000,000 to get a correct answer. In order to retrieve the next digit of the square root of "650", we have to multiply it by 100, which is the square of 10:

2OS	TOS	Word	Result
fraction	fraction	V*	fraction
fraction	cell	V*	cell
cell	fraction	V*	cell
fraction	fraction	V/	fraction
cell	cell	V/	fraction
cell	fraction	V/	cell
fraction	fraction	+	fraction
fraction	fraction	-	fraction

Table 10.2: Fraction words

```
[needs lib/math.4th]

: .fp <# # [char] . hold #S #> type cr ;
650 100 * sqrt .fp
```

Which will print:

```
25.4
```

To acquire greater precision we have to scale it up even further, like 10,000. This will show us, that "25.49" brings us even closer to the correct answer. If you want to use the ”fraction.4th” library, you will have to scale it to the square of ’10K’, which means you’re restricted to numbers upto ”21.475”:

```
include lib/math.4th
include lib/fraction.4th

10K 10K [*] constant 100M

21 100M * sqrt s>v v. cr
```

All these words and more are included in the library file ”math.4th”. You might not find the word you actually need, because we are not much of a mathematician. When we encounter new routines they will be added to 4tH. We would appreciate your input!

10.27 Double numbers

C’mon, indulge me, run this program:

```
max-n . cr
```

You will probably see some fairly large number displayed on your screen. What is it? Well, it is the largest number that can fit in a cell. Larger numbers and 4tH will start to behave erratically:

```
max-n 1+ . cr
```

Still, it is large enough to do the accounting for a reasonably sized enterprise. But it hasn't been always like that. Early Forths could barely handle the accounting of an average schoolboy. In order to get some real work done they had to expand the range somehow. And if one cell isn't enough you simply take two cells. That is what double numbers are all about: they are numbers that are composed of two cells.

The problem is that Forths operators aren't overloaded. If you try to add up two double numbers with '+' you will end up with one double number and the addition of the two parts of the first double number. So in order to add up two double numbers, a separate word had to be defined. If you need a special word for addition, you will also need one for multiplication, subtraction, division and negation.

It is no secret that Charles Moore, the inventor of Forth, thought that double numbers had become superfluous after the introduction of modern processors and modern Forth compilers. That is one of the reasons that 4tH doesn't have a native double word implementation. But should you need this vastly expanded range for one reason or another 4tH allows you to enter the murky world of double, unsigned and mixed numbers.

So, how does it work. First of all, if you need a full implementation you have to include these three libraries:

```
include lib/anscore.4th       \ double storage words
include lib/todbl.4th         \ double number input
include lib/dbldot.4th        \ double number output
```

Then you probably need some variables. But hey, if double numbers take up two cells, you can't use ordinary variables. That's true. You will need small arrays:

```
2 array dVar1                 \ double variable one
2 array dVar2                 \ double variable two
```

And here comes the next problem. How do you enter double numbers? Depending on the size of the number, you can use two approaches. The easiest one is to convert a single number to a double number with "U>D". The only catch is this only works for *positive* numbers. If you want to enter a *negative* number, you have to negate it afterwards:

```
500 u>d 60000 u>d d+          \ add 500 and 60000
2dup d. cr                    \ print the double number
dVar1 2!                      \ store it in variable one
```

Yes, every single operation has a *double* counterpart:

SINGLE	DOUBLE
+	d+
negate	dnegate
.	d.
2/	d2/
max	dmax
min	dmin
dup	2dup
@	2@
!	2!

Table 10.3: Examples of single and double number counterparts

But what if you want to enter a very large number right away? In that case you will have to convert a string to a double number with ">NUMBER[2]". This one is no better than "U>D" where negative numbers are concerned since it only takes *positive* numbers. Second, ">NUMBER" adds the converted number to a double number already on the stack. "0." puts a double number zero for you on the stack.

```
0. s" 5000000000" >number     \ convert a string to double
2drop 2dup d. cr              \ print the double number
dVar1 2@ dmax dVar2 2!        \ save the largest in variable two
```

Finally, when you have done all you needed to do and you're left with a number that is small enough you can convert it back to a single number with "D>U". Note this will only work for positive numbers:

```
dVar2 2@ dVar1 2@ d-          \ subtract both double variables
2dup d. cr                    \ print the double number
d2/ d2/ d>u . cr              \ divide by 4 and convert to unsigned
```

Rule of the thumb is: stay away from double numbers if you can. It is slow, cumbersome and error-prone. If you can't, goodnight and good luck!

10.28 Complex control structures

Sometimes, the normal control structures of 4tH are not enough. Take this implementation of '-TRAILING':

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup                       \ quit if length is zero
  while
    2dup 1- chars + c@ bl <>  \ is it still a space?
    if exit else 1- then      \ if not, quit
  repeat                      \ if so, decrement length
;
```

No one will tell you that this is elegant. You have to perform a test and quit the word. And this is still palatable. Imagine you have to test several conditions like this! It will become horrible pretty soon! Therefore, 4tH supports extended control structures. We've seen the basic control structures in sections 6.21, 6.22 and 6.23. Now we're expanding those into:

```
BEGIN .. WHILE .. WHILE .. AGAIN | REPEAT
BEGIN .. WHILE .. WHILE .. UNTIL
```

Yes, that's right: 'REPEAT' and 'AGAIN' are actually aliases. But what can we do with them? Well, take a look at our modified '-TRAILING' word:

[2]There is also a single number ">NUMBER" word. In that case you can select the double number version by using its alias ">DOUBLE".

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup
  while                       \ quit if length is zero
    2dup 1- chars + c@ bl =
  while                       \ quit if it is not a space
    1-                        \ decrement length
  repeat
;
```

You have to admit that the latter version is much more elegant and readable.

10.29 Sorting

Yes, 4tH can do that too. You just have to include 'qsort.4th' to make it all possible. It works pretty much like the sort routines you've seen in C, which means you have to devise a word to compare two values. Note that 'qsort.4th' can only sort *integer arrays*. Setting it up is pretty simple. First you have to include it:

```
include lib/qsort.4th
```

Then you have to create a word that returns a true flag when the second value on the stack is smaller than the top of the stack. In this example we will just compare two integers, so that is pretty easy:

```
: MyPrecedes < ;
```

'qsort.4th' creates a deferred word[3] called "PRECEDES". Now we have to assign our word to "PRECEDES", so that it is executed when "PRECEDES" is called:

```
' MyPrecedes is Precedes
```

That's it! We're ready to rock 'n roll now. Let's set up a simple testing environment:

```
10 constant #elements
#elements array elements

: InitElements #elements 0 do random elements i th ! loop ;
: ShowElements #elements 0 do elements i th @ . loop cr ;
```

This creates an array of ten elements, which is filled with random values by "InitElements". "ShowElements" will show on screen what is stored there. The actual sort is straightforward: tell "SORT" which array and how many elements there are to sort and you're done:

```
: SortElements elements #elements sort ;
```

Now let's put it all together:

[3] See section 10.11.

```
InitElements
ShowElements
SortElements
ShowElements
```

It will initialize the array, show its contents, sort it and show it again. It will output something like this:

```
12717 6028 1389 31870 14234 15884 31062 14788 18186 149
149 1389 6028 12717 14234 14788 15884 18186 31062 31870
```

And what if these were string addresses? Well, "SORT" would have sorted them too, from the lowest addresses up to the highest addresses, but that's probably not what you meant. You wanted to sort the actual strings, not just their addresses. Can 4tH do that too? Sure, you just got to create another "PRECEDES" word. Something like this:

```
: SPrecedes >R COUNT R> COUNT COMPARE 0< ;
```

This will take the two values and treat them as strings. Now the actual strings are sorted, not just the addresses itself. Note that the strings themselves will not move in memory. The *pointers* move, the strings themselves don't.

10.30 Tokenizing strings

Sometimes you want to split up a string in several different parts. This is called "tokenizing". Doing it with 4tH is (as usual ;-) quite easy. Just include 'tokenize.4th'. Now you got several words to get what you want. 'tokenize.4th' creates a deferred word[4] called "IS-TYPE". It decides whether a character is of a certain type. In this example, we just want to know whether it is a lowercase 'a':

```
include lib/tokenize.4th
:noname [char] a = ; is is-type
```

Now it's time to play ball:

```
s" 01234aBcDe01234" scan type cr
```

"SCAN" will now skip all characters unless it is an 'a'. When it is found, it stops and returns the remainder of the string:

```
aBcDe01234
```

Yes, "SCAN" starts at the beginning of the string. But there is also a word that starts *at the end* of the string:

```
s" 01234aBcDe01234" -scan type cr
```

[4] See section 10.11.

It returns a different result too:

```
01234a
```

And what about the rest of the string? Well, that is discarded. But if you need it, there is also a word that just splits up the string:

```
s" 01234aBcDe01234" split type cr type cr
```

So, ”SPLIT” returns *two* strings:

```
01234
aBcDe01234
```

And of course, he’s got a little brother that works the other way around:

```
s" 01234aBcDe01234" -split type cr type cr
```

”-SPLIT” returns two strings too:

```
01234a
BcDe01234
```

That was quite easy. But what if you want to find the *first non-digit*? That’s what we got ”SKIP” for! ”SKIP” skips all characters of a certain kind. We’ve already seen how we can distinguish characters in section 7.25. So in this case we just got to include ’istype.4th’ and assign ”IS-DIGIT” to ”IS-TYPE”:

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" skip type cr
```

”SKIP” returns a single string:

```
aBcDe01234
```

And he has a counterpart too:

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" -skip type cr
```

May be this result will surprise you, although it is completely correct:

```
01234aBcDe
```

What exactly did we ask for? We wanted the *first non-digit*, starting from *the end*. ’e’ is the first non-digit, so ”-SKIP” is completely correct.

10.31 Regular expressions

Well, we don't offer full regular expressions (yet), but you can use wildcards for basic pattern matching. First you have to include it:

```
include lib/wildcard.4th
```

You've probably used wildcards before. It is very easy. A "*" stands for *zero or more characters* and a "?" stands for *a single character*. E.g. if you're looking for a line that begins with a date in the 21st century, then the word "INVOICE" and finally ends with a name, you could try this:

```
s" 20??-??-??  == INVOICE ==  *" mystring count wild-match
```

"WILD-MATCH" returns true if the string matches and false if the string doesn't match, which is different from "COMPARE". Both strings are consumed, so save their address/count pair if you need them later on. Note that "WILD-MATCH" is *case sensitive*, so if you need a case insensitive comparison you will have to convert them first.

Finally, "WILD-MATCH" is faster than regular expressions, but also less precise. E.g. "gr?y" will not only return "grey" and "gray", but also "groy"; "reg*exp*" will not only return "regular expressions", but also "registered express mail".

10.32 Running 4tH programs from the Unix shell

If you're using Unix (which we highly recommend), you can run 4tH programs right from the Unix shell. All you have to do is to add one single line at the top:

```
#! /usr/lbin/4th cxq
." Hello world!" cr
```

It indicates the way you normally compile and run a 4tH program, but without the filename, e.g.:

```
/usr/lbin/4th cxq hello.4th
```

In this case, you're using the classic 4tH compiler, which is located in the /usr/lbin directory. Note that you can add options if you want. The 'cxq' options tell the compiler to silently compile and execute a program.

Note this trick only works with 4tH sources, not compiled programs. You also have to flag the 4tH source as 'executable'. You can do that by issuing this command:

```
chmod 555 hello.4th
```

Now you can simply enter:

```
hello.4th
```

at the Unix prompt and your program will be compiled and executed. Don't worry about compromising the portability of your program. It will still compile and run happily under other Operating Systems, since '#!' is an alias for '\'. It only has a special meaning to the Unix shell.

10.33 Embedding 4tH programs in a batch file

If you're running a Microsoft Operating System like Windows or DOS[5], you can embed 4tH source code in an ordinary batch file[6]. All you have to do is to make the shell ignore the 4tH code, e.g.:

```
@goto exec
." Hello world!" cr
(
:exec
@4th cxq %0.bat %1 %2 %3 %4 %5 %6 %7 %8 %9
@rem )
```

Now save your file as "EXAMPLE.BAT" in the current working directory[7] and run it:

```
example
```

Don't add the ".BAT" extension or the whole thing won't work. 4tH will now automatically pick up the batch file and execute it. Well, how does it work?

It's simple: the shell silently jumps to the "EXEC" label and executes 4tH. 4tH will compile the batch file. It ignores the line that starts with '@GOTO', since '@GOTO' is an alias for '\'. It compiles anything up to the opening parenthesis, since that is the start of a multiline comment. The shell on its turn ignores the closing parenthesis, since that has been commented out by "@REM".

10.34 This is the end

This is the end of it. If you mastered all we have written about 4tH, you may be just as proficient as we are. Or even better. In the meanwhile you may even have acquired a taste for this strange, but elegant language. If you do, it may be time to step up to Forth, since 4tH does have it limitations. This is in no way an obligation. If you feel comfortable with 4tH, please do stick with it!

If you need any help, you can contact us by sending an email to:

```
hansoft@bigfoot.com
```

Note that we do appreciate *any* input, so if you've written a state of the art application in 4tH, used 4tH in some special way or do have any comments or suggestions on 4tH, we'd like to hear from you! We do also have a web-site:

```
http://hansoft.come.to
```

You will find there lots of documentation and news on 4tH. We'd like to thank you for putting so much effort in 4tH. We tried to be of assistance and we hope we did it well!

[5]DOS version 3.3 or higher.

[6]This method was taken from CSL, the "C Scripting Language". You can learn more about CSL at "http://csl.sourceforge.net".

[7]If you want to store it permanently in another directory, you may have to add additional path information.

Part III

Reference guide

Chapter 11

Glossary

This glossary contains all of the word definitions used in version 3.5c of 4tH. The definitions are presented in order of their ASCII sort. Availability of the word in the appropriate ANS-Forth wordset is listed. This does not mean any conformance to the ANS-Forth definition.

PRONUNCIATION: Natural-language pronunciation if it differs from English.

INCLUDE: Following library file provides this word.

COMPILES TO: Describes the transformation of the word to tokens through all passes. Compiler directives will lack this section.

SYNTAX: Describes definition characteristics if non-conformance should lead to a compilation error.

<char>	Character
<string>	String constant, delimited by spaces
<literal>	Expression which compiles to LITERAL (n)
<name>	String of characters, stored in the symboltable
<space>	Space character
<word>	Any valid 4tH word.

COMPILER: Describes special actions the compiler takes when compiling this word.

STACK EFFECTS: Describes the action of the tokens on the parameter stack at runtime. The symbols indicate the order in which input parameters have been placed on the stack. Two dashes indicate the execution point. Any parameters left on the stack are listed. In this notation, the top of the stack is to the right.

n	32 bits signed number	
c	8 bits character	
f	boolean flag	
fam	file access method	
h	file handle (stream)	
d	double number (2 cells)	
sp	stack pointer	Stack Area
x	address of a cell	Variable Area
addr	address of a character	Character Segment
xt	execution token	Code Segment

FORTH: Describes the deviation of 4tH from ANS-Forth and gives suggestions for porting Forth programs.

! CORE

PRONUNCIATION: store

COMPILES TO: ! (0)

STACK EFFECTS: n x —

Stores n in the variable at address x.

CORE

PRONUNCIATION: number-sign

COMPILES TO: # (0)

STACK EFFECTS: n1 — n2

FORTH: In Forth a double number is required.

Generate from n1 the next ASCII character which is placed in an output string, stored in PAD. Result n2 is the quotient after the division by BASE, and is remained for further processing. Used between <# and #>.

#! 4TH

SYNTAX: #!<space><string>

The remainder of the line is discarded. This word is used to start a 4tH source program from a Unix type shell. An alias for \.

#> CORE

PRONUNCIATION: number-sign-greater

COMPILES TO: #> (0)

STACK EFFECTS: n1 — addr n2

FORTH: In Forth a double number is required.

Terminates numeric output conversion by dropping n1, leaving the address in PAD and character count n2 suitable for TYPE.

#S CORE

PRONUNCIATION: number-sign-s

COMPILES TO: #S (0)

STACK EFFECTS: n1 — n2

FORTH: In Forth a double number is required.

Generates ASCII text in PAD by the use of # until a zero number n2 results. Used between <# and #>.

#TIB CORE EXT

PRONUNCIATION: number-t-i-b

INCLUDE: obsolete.4th

STACK EFFECTS: — x

X is the address of a cell containing the number of characters in the terminal input buffer (see */TIB*).

' CORE

PRONUNCIATION: tick

COMPILES TO: LITERAL (<argument of symbol>)

SYNTAX: '<space><name>

STACK EFFECTS: — x | xt | n

FORTH: In Forth you can determine the address of variables, constants, etc. In 4tH the contents of the symboltable entry is returned. Of course the token addresses of built-in primitives cannot be determined either. E.g. use

```
: _+ + ; ' _+
```

instead of

```
' +
```

Compile the value contents of the symboltable entry identified as symbol <name> as a literal.

(CORE FILE

PRONUNCIATION: paren

SYNTAX: (<space><string>)

Ignore a comment that will be delimited by a right parenthesis. May occur inside or outside a colon-definition. A blank after the leading parenthesis is required.

(ERROR) 4TH

COMPILES TO: LITERAL (<largest negative integer>)

STACK EFFECTS: — n

Returns 4tHs internal error-flag. This number cannot be printed. Usually -2^31.

) **4TH**

COMPILES TO: EQ0 (0)

0BRANCH (<address of THROW>)

LITERAL (<M4ASSERT>)

THROW (0)

STACK EFFECTS: f —

FORTH: Similar constructions are available in GForth and Win32Forth.

If flag f is FALSE, the program will terminate with an error. Its compilation is dependant on the presence of [ASSERT] (see: *[ASSERT]* and *ASSERT(*).

* **CORE**

PRONUNCIATION: star

COMPILES TO: * (0)

STACK EFFECTS: n1 n2 — n3

Leave the product n3 of two numbers n1 and n2.

*/ **CORE**

PRONUNCIATION: star-slash

COMPILES TO: */ (0)

STACK EFFECTS: n1 n2 n3 — n4

Leave the ratio n4 = n1*n2/n3.

*/MOD **CORE**

PRONUNCIATION: star-slash-mod

COMPILES TO: >R (0)

* (0)

R> (0)

/MOD (0)

STACK EFFECTS: n1 n2 n3 — n4 n5

Leave the quotient n5 and remainder n4 of the operation n1*n2/n3.

+ CORE

PRONUNCIATION: plus

COMPILES TO: + (0)

STACK EFFECTS: n1 n2 — n3

Leave the sum n3 of n1+n2.

+! CORE

PRONUNCIATION: plus-store

COMPILES TO: +! (0)

STACK EFFECTS: n x —

Add n to the value in variable at address x.

+CONSTANT 4TH

SYNTAX: <literal><space>+CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for +CONSTANT. The instruction pointer is decremented, actually deleting the literal.

A defining word used to create word <name>. When <name> is later executed, it will add the value of <literal> on the top of the stack.

+FIELD FACILITY EXT

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>END-STRUCT<space><name>

COMPILER: Take two previous compiled literals. The last literal is added to the first and recompiled. The first literal is the value of a named +CONSTANT. The instruction pointer does not change.

Create a field for STRUCTURE implementations. The created fieldname is an +CONSTANT that memorizes the current offset (see: *+FIELD, STRUCT*, *END-STRUCT*).

+LOOP CORE

PRONUNCIATION: plus-loop

COMPILES TO: +LOOP (<address of matching DO token>)

SYNTAX: DO<space>..<space>+LOOP

STACK EFFECTS: n —

Used in the form DO .. n1 +LOOP. At runtime, +LOOP selectively controls branching back to the corresponding DO based on n1, the loop index and the loop limit. The increment n1 is added to the index and the total compared to the limit. The branch back to DO occurs until the new index is equal to or greater than the limit (n > 0), or until the new index is less than the limit (n < 0). Upon exiting the loop, the parameters are discarded and execution continues ahead.

+PLACE COMUS

COMPILES TO: COUNT (0)

+ (0)

PLACE (0)

STACK EFFECTS: addr1 n addr2 —

Copies the string at address addr1 with count n to address addr2.

, CORE

PRONUNCIATION: comma

COMPILES TO: , (<literal>)

SYNTAX: <literal><space>,

COMPILER: The previously compiled literal is changed into a NOOP instruction. The instruction pointer is not incremented.

FORTH: Forth pops a value from the stack. This is not possible in 4tH. Instead the previously compiled literal has its codefield changed to NOOP.

Store the literal into the next available location.

," COMUS

COMPILES TO: ," (<address of string constant>)

SYNTAX: ,"<space><string>"

FORTH: Compilation characteristics are quite different. 4tH compiles only the address, Forth compiles the entire string.

Compile the string, delimited by " in the String Segment and leave the offset as the address of a string constant (see: *@C*).

,| 4TH

COMPILES TO: ,'' (<address of string constant>)

SYNTAX: ,|<space><string>|

Compile the string, delimited by | in the String Segment and leave the offset as the address of a string constant (see: *@C*).

- CORE

PRONUNCIATION: minus

COMPILES TO: - (0)

STACK EFFECTS: n1 n2 — n3

Leave the difference of n1 - n2 in n3.

-> 4TH

COMPILER: The instruction pointer is not incremented. In fact, -> is a dummy.

SYNTAX: <name><space>-><space><name>

Separation between a structure and its member.

-ROT COMUS

COMPILES TO: ROT (0)

ROT (0)

STACK EFFECTS: n1 n2 n3 — n3 n1 n2

Rotate top stack item below the next two items.

-TRAILING STRING

PRONUNCIATION: dash-trailing

COMPILES TO: -TRAILING (0)

STACK EFFECTS: addr n1 — addr n2

Adjusts the character count n1 of a string beginning address to suppress the output of trailing blanks, i.e. the characters from addr+n1 to addr+n2 are blanks.

. CORE

PRONUNCIATION: dot

COMPILES TO: . (0)

STACK EFFECTS: n —

Print a number to the current output device, converted according to the numeric BASE. A trailing blank follows.

." CORE

PRONUNCIATION: dot-quote

COMPILES TO: ." (<address of string constant>)

SYNTAX: ."<space><string>"

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device.

.(CORE EXT

PRONUNCIATION: dot-paren

COMPILES TO: ." (<address of string constant>)

SYNTAX: .(<space><string>)

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device. An alias for .".

.R CORE EXT

PRONUNCIATION: dot-r

COMPILES TO: .R (0)

STACK EFFECTS: n1 n2 —

Print the number n1 right aligned in a field whose width is n2 to the current output device. No following blank is printed.

.S TOOLS

PRONUNCIATION: dot-s

INCLUDE: anstools.4th

STACK EFFECTS: —

Copy and display the values currently on the data stack.

.| 4TH

COMPILES TO: ." (<address of string constant>)

SYNTAX: .|<space><string>|

Compiles string in the String Segment with an execution procedure to transmit the string to the selected output device.

/ CORE

PRONUNCIATION: slash

COMPILES TO: / (0)

STACK EFFECTS: n1 n2 — n3

Leaves the quotient n3 of n1/n2.

/CELL COMUS

COMPILES TO: LITERAL (<size of a cell>)

STACK EFFECTS: — n

Returns the size of a cell in address units.

/CHAR COMUS

COMPILES TO: LITERAL (<size of char>)

STACK EFFECTS: — n

Returns the size of a character in address units.

/MOD CORE

PRONUNCIATION: slash-mod

COMPILES TO: /MOD (0)

STACK EFFECTS: n1 n2 — n3 n4

Leave the remainder n3 and quotient n4 of n1/n2.

/PAD 4TH

COMPILES TO: LITERAL (<size of PAD>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: /PAD S" /PAD" ENVIRONMENT? DROP ;
```

Returns the size of PAD.

/STRING STRING

PRONUNCIATION: slash-string

COMPILES TO: SWAP (0)

OVER (0)

- (0)

>R (0)

+ (0)

R> (0)

STACK EFFECTS: addr1 n1 n2 — addr2 n3

Adjust the character string at addr1 by n2 characters. The resulting character string, specified by addr2 n3 , begins at addr1 plus n2 characters and is n1 minus n characters long.

/TIB 4TH

COMPILES TO: LITERAL (<size of TIB>)

STACK EFFECTS: — n

Returns the size of the terminal input buffer.

0< CORE

PRONUNCIATION: zero-less

COMPILES TO: 0< (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is less than zero (negative), otherwise leave a FALSE flag in f.

0<> CORE EXT

PRONUNCIATION: zero-not-equals

COMPILES TO: 0<> (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is not equal to zero, otherwise leave a FALSE flag in f.

0= CORE

PRONUNCIATION: zero-equals

COMPILES TO: 0= (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is equal to zero, otherwise leave a FALSE flag in f.

0> CORE EXT

PRONUNCIATION: zero-greater

COMPILES TO: 0> (0)

STACK EFFECTS: n — f

Leave a TRUE flag if number n is greater than zero (positive), otherwise leave a FALSE flag in f.

1+ CORE

PRONUNCIATION: one-plus

COMPILES TO: 1+ (0)

STACK EFFECTS: n — n+1

Increment n by 1.

1- CORE

PRONUNCIATION: one-minus

COMPILES TO: 1+ (0)

STACK EFFECTS: n — n+1

Decrement n by 1.

2! CORE

PRONUNCIATION: two-store

INCLUDE: anscore.4th

STACK EFFECTS: n1 n2 x —

Store the cell pair n1 n2 at x, with n2 at x and n2 at the next consecutive cell.

2* CORE

PRONUNCIATION: two-star

COMPILES TO: 2* (0)

STACK EFFECTS: n — n*2

Multiply n by 2. Performs a left shift.

2/ CORE

PRONUNCIATION: two-slash

COMPILES TO: 2/ (0)

STACK EFFECTS: n — n/2

Divide n by 2. Performs a right shift.

2>R CORE EXT

PRONUNCIATION: two-to-r

COMPILES TO: >R (0)

>R (0)

STACK EFFECTS: n1 n2 —

FORTH: Forth swaps both values before transfering them to the return stack.

Transfer cell pair n1 n2 to the return stack.

2@ CORE

PRONUNCIATION: two-fetch

INCLUDE: anscore.4th

STACK EFFECTS: x — n1 n2

Fetch the cell pair n1 n2 stored at x. n2 is stored at x and n1 at the next consecutive cell.

2DROP CORE

PRONUNCIATION: two-drop

COMPILES TO: DROP (0)

	DROP (0)
STACK EFFECTS:	n1 n2 —

Drop cell pair n1 n2 from the stack.

2DUP CORE

PRONUNCIATION:	two-dupe
COMPILES TO:	OVER (0)
	OVER (0)
STACK EFFECTS:	n1 n2 — n1 n2 n1 n2

Duplicate cell pair n1 n2.

2OVER CORE

PRONUNCIATION:	two-over
INCLUDE:	2rotover.4th
STACK EFFECTS:	n1 n2 n3 n4 — n1 n2 n3 n4 n1 n2

Copy cell pair n1 n2 to the top of the stack.

2R> CORE EXT

PRONUNCIATION:	two-r-from
COMPILES TO:	R> (0)
	R> (0)
STACK EFFECTS:	— n1 n2
FORTH:	Forth swaps both values after transfering them from the return stack.

Transfer cell pair n1 n2 from the return stack.

2R@ CORE EXT

PRONUNCIATION:	two-r-fetch
COMPILES TO:	R> (0)
	I (0)
	OVER (0)
	>R (0)

STACK EFFECTS: — n1 n2

FORTH: Forth swaps both values after transfering them from the return stack.

Copy cell pair n1 n2 from the return stack.

2ROT DOUBLE EXT

PRONUNCIATION: two-rote

INCLUDE: 2rotover.4th

STACK EFFECTS: n1 n2 n3 n4 n5 n6 — n3 n4 n5 n6 n1 n2

Rotate the top three cell pairs on the stack bringing cell pair n1 n2 to the top of the stack.

2SWAP CORE

PRONUNCIATION: two-swap

COMPILES TO: ROT (0)

>R (0)

ROT (0)

R> (0)

STACK EFFECTS: n1 n2 n3 n4 — n3 n4 n1 n2

Exchange the top two cell pairs.

4TH# 4TH

COMPILES TO: LITERAL (<4tH version in hexadecimal>)

STACK EFFECTS: — n

Constant containing the 4tH version in hexadecimal.

: CORE

PRONUNCIATION: colon

COMPILES TO: BRANCH (<address of matching ; token>)

SYNTAX: :<space><name>..<space>;

Creates a subroutine defining <name> as equivalent to the following sequence of 4tH word definitions until the next ;.

:NONAME CORE EXT

PRONUNCIATION: colon-no-name

COMPILES TO: LITERAL (<address of next BRANCH>)

BRANCH (<address of matching ; token>)

SYNTAX: :NONAME<space>..<space>;

STACK EFFECTS: — xt

Create an execution token xt and compile the current definition. The execution semantics of xt will be determined by the words compiled into the body of the definition. This definition can be executed later by using xt EXECUTE.

:THIS 4TH

COMPILES TO: BRANCH (<address of matching ; token>)

LITERAL (<original value>) | VARIABLE (<original value>)

SYNTAX: :THIS<space><name><space>DOES><space>..<space>;

Create an subroutine <name> that first pushes the original value of <name> on the stack. The words after DOES> determine what the actual execution behaviour will be (see: *DOES>*).

; CORE

PRONUNCIATION: semi-colon

COMPILES TO: EXIT (0)

SYNTAX: See *:*

Terminate a colon definition. At runtime, return to the calling word by popping a token-address from the return stack.

< CORE

PRONUNCIATION: less-than

COMPILES TO: < (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is less than n2; otherwise leave a FALSE flag in f.

<# CORE

PRONUNCIATION: less-number-sign

COMPILES TO: <# (0)

FORTH: In Forth a double number is required.

Setup for pictured numeric output formatting in PAD using the words <#, #, #S, SIGN, HOLD, #>.

<> CORE EXT

PRONUNCIATION: not-equals

COMPILES TO: <> (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 does not equal n2; otherwise leave a FALSE flag in f.

<= 4TH

COMPILES TO: > (0)

0= (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is less or equal than n2; otherwise leave a FALSE flag in f.

= CORE

PRONUNCIATION: equals

COMPILES TO: = (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 equals n2; otherwise leave a FALSE flag in f.

> CORE

PRONUNCIATION: greater-than

COMPILES TO: > (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is greater than n2; otherwise leave a FALSE flag in f.

>= 4TH

COMPILES TO: < (0)

0= (0)

STACK EFFECTS: n1 n2 — f

Leave a TRUE flag if n1 is greater or equal than n2; otherwise leave a FALSE flag in f.

>BODY CORE

PRONUNCIATION: to-body

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

STACK EFFECTS: n — x

FORTH: In Forth, >BODY works with every CREATEd datatype.

n is the ticked value of a VARIABLE, VALUE, DEFER or FILE. >BODY returns its address in the Variable Area.

>IN CORE

PRONUNCIATION: to-in

COMPILES TO: LITERAL (<address of >IN>)

STACK EFFECTS: — x

A variable containing the address within the Character Segment from which the next text will be parsed. PARSE uses and moves the value of >IN.

>NUMBER CORE

PRONUNCIATION: to-number

INCLUDE: tonumber.4th

STACK EFFECTS: n1 a1 n2 — n3 a2 n4

n3 is the unsigned result of converting the characters within the string specified by a1 n2 into digits, using the number in BASE, and adding each into n1 after multiplying n1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any + or -, is encountered or the string is entirely converted. a2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. n4 is the number of unconverted characters in the string. An ambiguous condition exists if n3 overflows during the conversion.

>NUMBER CORE

PRONUNCIATION: to-number

INCLUDE: todbl.4th

STACK EFFECTS: d1 a1 n1 — d2 a2 n2

d2 is the unsigned result of converting the characters within the string specified by a1 n2 into digits, using the number in BASE, and adding each into d1 after multiplying d1 by the number in BASE. Conversion continues left-to-right until a character that is not convertible, including any + or -, is encountered or the string is entirely converted. a2 is the location of the first unconverted character or the first character past the end of the string if the string was entirely converted. n2 is the number of unconverted characters in the string. An ambiguous condition exists if d2 overflows during the conversion.

>R CORE

PRONUNCIATION: to-r

COMPILES TO: >R (0)

STACK EFFECTS: n —

Remove n from the stack and place it on the return stack. Use should be balanced with R> in the same definition.

? TOOLS

PRONUNCIATION: question

COMPILES TO: @ (0)

. (0)

STACK EFFECTS: x —

Print the value contained in the variable at address x in free format according to the current BASE.

?DO CORE EXT

PRONUNCIATION: question-do

COMPILES TO: ?DO (0)

SYNTAX: ?DO<space>..<space>+LOOP

?DO<space>..<space>LOOP

STACK EFFECTS: n1 n2 —

If n1 is equal to n2, continue execution at LOOP or +LOOP. Otherwise set up loop control parameters with index n2 and limit n1 and continue executing immediately following ?DO. Anything already on the return stack becomes unavailable until the loop control parameters are discarded.

?DUP CORE

PRONUNCIATION: question-dupe

INCLUDE: anscore.4th

STACK EFFECTS: n — 0 | n n

Duplicate n if it is non-zero.

@ CORE

PRONUNCIATION: fetch

COMPILES TO: @ (0)

STACK EFFECTS: x — n

Leave the contents n of the variable at address x on the stack.

@C CROSS EXT

COMPILES TO: @C (0)

STACK EFFECTS: xt — n | addr

FORTH: In Forth the word @ can also be used to fetch values from the dictionary. Due to 4tHs internal structure this is not possible.

Leave the contents n of the parameter field of token address xt on the stack. If n contains an string constant compiled by ," it is copied to the PAD. Its address is returned as addr.

@GOTO 4TH

SYNTAX: @GOTO<space><string>

The remainder of the line is discarded. This word is used to start a 4tH source program from a MS type shell. An alias for \.

ABORT CORE

FORTH: In Forth the behaviour of ABORT is different from QUIT (`-1 THROW`). In 4tH it doesn't really matter which one you use.

An alias for QUIT.

ABORT" CORE

PRONUNCIATION: abort-quote

COMPILES TO: 0BRANCH (<address of QUIT>)

LITERAL (stdout)

USE (0)

." (<address of string constant>)

CR (0)

QUIT (0)

SYNTAX: ABORT"<space><string>"

STACK EFFECTS: n —

FORTH: In Forth the behaviour of ABORT" is different from QUIT (`-2 THROW`).

Remove n from the stack. If any bit of n is not zero, display the string and set the program counter to the end of the program. Effectively quits execution.

ABS CORE

PRONUNCIATION: abs

COMPILES TO: ABS (0)

STACK EFFECTS: n1 — n2

Leave the absolute value of n1 as n2.

ACCEPT CORE

COMPILES TO: ACCEPT (0)

STACK EFFECTS: addr n1 — n2

FORTH: In Forth no null character is appended.

Read n1 characters from the current input device to address addr. If input is read from the terminal CR will terminate the input stream. All other devices will terminate reading when an EOF occurs. In all cases input will end when n1 characters have been read. A null character is added to the end of the input when reading from the keyboard. The number n2 represents the number of characters actually read.

AGAIN CORE EXT

COMPILES TO: BRANCH (<address of the token following BEGIN>)

SYNTAX: BEGIN<space>..<space>AGAIN

At runtime, AGAIN forces execution to return to the corresponding BEGIN. Execution cannot leave this loop. AGAIN is an alias for REPEAT.

AKA 4TH

SYNTAX: AKA<space><word name><space><name>

Create a word <name> with the same compilation and execution semantics as the existing word <word name>. The word <word name> has to be user defined.

ALIAS 4TH

COMPILES TO: TO (<variable address>)

STACK EFFECTS: xt —

SYNTAX: ALIAS<space><name>

Store xt in the value identified by name. ALIAS is an alias for IS, but does not require a previously defined DEFER.

ALIGN CORE

COMPILER: The instruction pointer is not incremented. In fact, ALIGN is a dummy.

If the dataspace pointer is not aligned, reserve enough space to align it.

ALIGNED CORE

COMPILER: The instruction pointer is not incremented. In fact, ALIGNED is a dummy.

STACK EFFECTS: n — n

n is the first aligned address greater than or equal to n.

ALLOCATE MEMORY

INCLUDE: ansmem.4th

STACK EFFECTS: n — addr f

Allocate n address units of contiguous data space. The initial content of the allocated space is undefined. If the allocation succeeds, addr is the aligned starting address of the allocated space and f is false. If the operation fails, addr does not represent a valid address and f is true.

AND CORE

COMPILES TO: AND (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical AND of n1 AND n2 as n3.

APP 4TH

COMPILES TO: LITERAL (<application variable>)

STACK EFFECTS: — x

This word returns the variable address x in the Variable Area to an array of application specific variables. If APP equals FIRST no application specific variables have been defined.

APPEND 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method modifier on the stack, signalling that output will be appended. Must be added to another file access modifier. Used in combination with OUTPUT.

ARGN 4TH

COMPILES TO: ARGN (0)

STACK EFFECTS: — n

Returns the number of arguments that have been passed to 4tH (see: *ARGS*).

ARGS 4TH

COMPILES TO: ARGS (0)

STACK EFFECTS: n1 — addr n2

Copies argument n1 to the PAD and leaves address addr and length n2 on the stack (see: *ARGN*).

ARRAY 4TH

SYNTAX: <literal><space>ARRAY<space><name>

COMPILER: The previously compiled literal is taken as an argument for ARRAY. The instruction pointer is decremented, actually deleting the literal.

FORTH: Roughly equivalent to:

```
: ARRAY CREATE CELLS ALLOT ;
```

Allocate <literal> cells of contiguous data space beginning at <name> in the Integer Segment. The initial content of the allocated space is undefined.

AS 4TH

COMPILES TO: TO (<variable address>)

STACK EFFECTS: h —

SYNTAX: AS<space><name>

Store filehandle h in the value identified by name, previously defined by FILE (see: *FILE*). An alias for TO.

ASSERT(4TH

SYNTAX: ASSERT(<space><word>..<word><space>)

FORTH: Similar constructions are available in GForth and Win32For.

Mark the beginning of an assertion. If assertions are disabled all words following upto) are commented out (see: *[ASSERT]* and *)*).

BASE CORE

COMPILES TO: LITERAL (<address of BASE>)

STACK EFFECTS: — x

A variable containing the current number BASE used for input and output.

BEGIN CORE

SYNTAX: BEGIN<space>..<space>AGAIN

BEGIN<space>..<space>WHILE<space>..<space>UNTIL

BEGIN<space>..<space>WHILE<space>..<space>REPEAT

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime begin marks the start of a sequence that may be repetitively executed. It serves as a return point from the corresponding UNTIL, AGAIN or REPEAT. When executing UNTIL, a return to BEGIN will occur if the top of the stack is false; for AGAIN and REPEAT a return to BEGIN always occurs. Multiple WHILEs may be used.

BIN FILE

INCLUDE: ansfile.4th

STACK EFFECTS: fam1 — fam2

Modify file access method fam1 to additionally select a binary, i.e., not line oriented, file access method, giving access method fam2. Since 4tH does this automatically, BIN is a dummy.

BL CORE

PRONUNCIATION: b-l

COMPILES TO: LITERAL (<ASCII value of space>)

STACK EFFECTS: — c

A constant that leaves the ASCII value for "blank".

BLANK STRING

COMPILES TO: LITERAL (<ASCII value of space)

FILL (0)

STACK EFFECTS: n addr —

If n is greater than zero, store the character value for space in n consecutive character positions beginning at addr.

BLK BLOCK

PRONUNCIATION: b-l-k

INCLUDE: ansblock.4th

STACK EFFECTS: — x

FORTH: In Forth, a block cannot have the number zero. BLK contains the number of the block being *interpreted*.

x is the address of a cell containing the number of the mass-storage block currently cached. Altering the contents of BLK will have no lasting effects.

BLOCK BLOCK

INCLUDE: ansblock.4th

STACK EFFECTS: n — addr

Addr is the address of the first character of the block buffer assigned to mass-storage block n. An ambiguous condition exists if u is not an available block number. If block n is already in a block buffer, addr is the address of that block buffer. If block n is not already in memory, unassign the block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage and transfer block n from mass storage into that buffer. a-addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to n.

BOUNDS COMUS

COMPILES TO: OVER (0)

+ (0)

SWAP (0)

STACK EFFECTS: addr n — addr addr+n

Convert a starting value and count into the form required for a DO or ?DO loop.

BUFFER BLOCK

INCLUDE: ansblock.4th

STACK EFFECTS: n — addr

Addr is the address of the first character of the block buffer assigned to mass-storage block n. An ambiguous condition exists if n is not an available block number. If block n is already in a block buffer, addr is the address of that block buffer. If block n is not already in memory, unassign the block buffer. If the block in that buffer has been UPDATEd, transfer the block to mass storage and transfer block n from mass storage into that buffer. a-addr is the address of that block buffer. At the conclusion of the operation, the block buffer pointed to by addr is the current block buffer and is assigned to n.

C! CORE

PRONUNCIATION: c-store

COMPILES TO: C! (0)

STACK EFFECTS: c addr —

Store 8 bits of c at address addr in the Character Segment.

C, CORE

PRONUNCIATION: c-comma

SYNTAX: <literal><space>C,

COMPILER: The previously compiled literal is added as a character to the String Segment. The instruction pointer is decremented, actually deleting the literal.

FORTH: Forth pops a value from the stack. This is not possible in 4tH.

Reserve space for one character in the String Segment and store char in the space.

C@ CORE

PRONUNCIATION: c-fetch

COMPILES TO: C@ (0)

STACK EFFECTS: addr — c

Leave the 8 bits contents of Character Segment address addr as c.

CATCH EXCEPTION

COMPILES TO: CATCH (0)

(CATCH) (0)

STACK EFFECTS: xt — n

Push an exception frame on the return stack and execute the execution token xt in such a way that control can be transferred to a point just after CATCH if THROW is executed during the execution of xt (see: *THROW*).

CELL+ CORE

PRONUNCIATION: cell-plus

COMPILES TO: 1+ (0)

STACK EFFECTS: x1 — x2

Add the the size of a cell in cells to x1 giving x2.

CELL- COMUS

COMPILES TO: 1- (0)

STACK EFFECTS: x1 — x2

Subtract the the size of a cell in cells to x1 giving x2.

CELLS CORE

COMPILER: The instruction pointer is not incremented. In fact, CELLS is a dummy.

STACK EFFECTS: n — n

n is the size in cells of n cells.

CHAR CORE

PRONUNCIATION: char

COMPILES TO: LITERAL (<ASCII-value of character>)

SYNTAX: CHAR<space><char>

STACK EFFECTS: — c

Compiles the ASCII-value of <char> as a literal. At runtime the value is thrown on the stack.

CHAR+ CORE

PRONUNCIATION: char-plus

COMPILES TO: 1+ (0)

STACK EFFECTS: addr1 — addr2

Add the the size of a character in characters to addr1 giving addr2.

CHAR- 4TH

COMPILES TO: 1- (0)

STACK EFFECTS: addr1 — addr2

Subtract the the size of a character in characters to addr1 giving addr2.

CHARS CORE

PRONUNCIATION: chars

COMPILER: The instruction pointer is not incremented. In fact, CHARS is a dummy.

STACK EFFECTS: n — n

FORTH: In 4tH CHARS is a dummy, but it can be used to make a program ANS-compatible.

n is the size in characters of n characters.

CHOP 4TH

COMPILES TO: 1- (0)

SWAP (0)

1+ (0)

SWAP (0)

STACK EFFECTS: a n — a+1 n-1

Deletes the first character from the string defined by address a and length n.

CIN 4TH

COMPILES TO: ENVIRON (<address of CIN>)

STACK EFFECTS: — n

Identifies the input source.

CLOSE 4TH

COMPILES TO: CLOSE (0)

STACK EFFECTS: h —

CLOSE will close a file or pipe, previously opened by OPEN and release the stream. Depending on the file access method, the terminal will be made the current input-device, otherwise the screen will be made the current output-device.

CLOSE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — f

Close the file identified by handle h. Flag f is the implementation-defined I/O result code.

CMOVE STRING

PRONUNCIATION: c-move

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

FORTH: In Forth there are two words for this operation, CMOVE and CMOVE>. Usage depends on the direction of the move. In 4tH CMOVE is smart, like MOVE.

Move the specified quantity of bytes (n) beginning at address addr1 to addr2.

CMOVE> STRING

PRONUNCIATION: c-move-up

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

An alias for CMOVE (see: *CMOVE*).

COMPARE STRING

INCLUDE: anstring.4th

STACK EFFECTS: addr1 n1 addr2 n2 — n3

Compare the string specified by addr1 n1 to the string specified by addr2 n2 . The strings are compared, beginning at the given addresses, character by character, up to the length of the shorter string or until a difference is found. If the two strings are identical, n3 is zero. If the two strings are identical up to the length of the shorter string, n3 is -1 if n1 is less than n2 and 1 otherwise. If the two strings are not identical up to the length of the shorter string, n3 is -1 if the first non-matching character in the string specified by addr1 n1 has a lesser numeric value than the corresponding character in the string specified by addr2 n2 and 1 otherwise.

CONSTANT CORE

SYNTAX: <literal><space>CONSTANT<space><name>

COMPILER: The previously compiled literal is taken as an argument for CONSTANT. The instruction pointer is decremented, actually deleting the literal.

FORTH: In Forth, the literal value is popped from the stack. This cannot be done in 4tH.

A defining word used to create word <name>. When <name> is later executed, it will push the value of <literal> on the stack.

COUNT CORE

COMPILES TO: COUNT (0)

STACK EFFECTS: addr1 — addr2 n

FORTH: Programs assuming that the string is a so-called counted string will *not* work. Well-written programs only assume the correct input- and output-parameters.

Leave the Character Segment address addr2 and count n of an ASCIIZ string beginning at Character Segment address addr1. Typically COUNT is followed by TYPE.

COUT 4TH

COMPILES TO: ENVIRON (<address of COUT>)

STACK EFFECTS: — n

Identifies the output source.

CR CORE

PRONUNCIATION: c-r

COMPILES TO: CR (0)

Transmit a carriage return to the selected output-device. The actual sequence sent is OS- and stream-dependant.

CREATE CORE

SYNTAX: CREATE<space><name>

FORTH: In Forth this will create a dictionary header.

Leaves <name> in the symboltable and replace further occurences with LITERAL <xt>. <xt> represents the address in the Code Segment where CREATE was compiled.

CREATE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n fam — h f

Create the file named in the character string specified by addr and n, and open it with file access method fam. The meaning of values of fam is implementation defined. If a file with the same name already exists, recreate it as an empty file. If the file was successfully created and opened, f is zero, handle h is its identifier, and the file has been positioned to the start of the file. Otherwise, f is the implementation-defined I/O result code and h is undefined.

D+ DOUBLE

PRONUNCIATION: d-plus

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

Add d1 to d2, giving the sum d3.

D- DOUBLE

PRONUNCIATION: d-minus

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

Subtract d2 from d1, giving the difference d3.

D. DOUBLE

PRONUNCIATION: d-dot

INCLUDE: dbldot.4th

STACK EFFECTS: d —

Display d in free field format.

D.R DOUBLE

PRONUNCIATION: d-dot-r

INCLUDE: dbldot.4th

STACK EFFECTS: d n —

Display d right aligned in a field n characters wide. If the number of characters required to display d is greater than n, all digits are displayed with no leading spaces in a field as wide as necessary.

D0< DOUBLE

PRONUNCIATION: d-zero-less

INCLUDE: ansdbl.4th

STACK EFFECTS: d — f

Flag f is true if and only if d is less than zero.

D0= DOUBLE

PRONUNCIATION: d-zero-equals

INCLUDE: ansdbl.4th

STACK EFFECTS: d — f

Flag f is true if and only if d is equal to zero.

D2* DOUBLE

PRONUNCIATION: d-two-star

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the result of shifting d1 one bit toward the most-significant bit, filling the vacated least-significant bit with zero.

D2/ DOUBLE

PRONUNCIATION: d-two-slash

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the result of shifting d1 one bit toward the least-significant bit, leaving the most-significant bit unchanged.

D< DOUBLE

PRONUNCIATION: d-less-than

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — f

Flag f is true if and only if d1 is less than d2.

D= DOUBLE

PRONUNCIATION: d-equals

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — f

Flag f is true if and only if d1 is equal to d2.

D>S DOUBLE

PRONUNCIATION: d-to-s

COMPILER: The instruction pointer is not incremented. In fact, D>S is a dummy.

STACK EFFECTS: n — n

Convert the number n to number n with the same numerical value.

DABS DOUBLE

PRONUNCIATION: d-abs

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the absolute value of d1.

DMAX DOUBLE

PRONUNCIATION: d-max

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

D3 is the greater of d1 and d2.

DMIN DOUBLE

PRONUNCIATION: d-min

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 d2 — d3

D3 is the lesser of d1 and d2.

DNEGATE DOUBLE

PRONUNCIATION: d-negate

INCLUDE: ansdbl.4th

STACK EFFECTS: d1 — d2

D2 is the negation of d1.

DECIMAL CORE

COMPILES TO: RADIX (10)

FORTH: See *HEX*.

Set the numeric conversion BASE for decimal output at runtime.

DEFER CORE EXT

COMPILES TO: LITERAL ((ERROR))

TO (<variable address>)

SYNTAX: DEFER<space><name>

STACK EFFECTS: —

Create a value name which will hold an execution token for a word whose behavior will be determined later and may be varied. The initial value will trigger an error if used before proper assignment.

DEFER! CORE EXT

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

! (0)

STACK EFFECTS: xt x —

Set the vector x to execute xt.

DEFER@ CORE EXT

COMPILES TO: ENVIRON (<address of FIRST>)

+ (0)

@ (0)

STACK EFFECTS: x — xt

xt is the xt associated with the deferred word corresponding to x.

DEPTH CORE

COMPILES TO: SP@ (0)

STACK EFFECTS: — n

Returns the number of items on the stack in n, before DEPTH was executed. An alias for SP@.

DO CORE

COMPILES TO: DO (0)

SYNTAX: DO<space>..<space>+LOOP

DO<space>..<space>LOOP

STACK EFFECTS: n1 n2 —

At runtime DO begins a sequence with repetitive execution controlled by a loop limit n1 and an index with initial value n2. DO removes these from the stack. Upon reaching LOOP or +LOOP the index is altered. Until the new index equals or exceeds the limit, execution loops back to just after DO; otherwise the loop parameters are discarded and execution continues ahead. Both n1 and n2 are determined at runtime and may be the result of other operations. Within a loop I will copy the current value of the index on the stack.

DOES> CORE

PRONUNCIATION: does

COMPILER: The instruction pointer is not incremented. In fact, DOES> is a dummy.

SYNTAX: :THIS<space><name><space>DOES><space>..<space>;

FORTH: In Forth, DOES> is usually combined with CREATE, changing the behaviour of an entire datatype. In 4tH it is used with :THIS, changing only the referenced definition.

Append the run-time semantics to the referenced definition. Execute the portion of the definition that begins with the initiation semantics appended by the DOES> which modified name.

DROP CORE

COMPILES TO: DROP (0)

STACK EFFECTS: n —

Drop the number from the stack.

DUMP TOOLS

INCLUDE: dump.4th

STACK EFFECTS: addr n —

Display the contents of n consecutive addresses starting at addr.

DUP CORE

PRONUNCIATION: dupe

COMPILES TO: DUP (0)

STACK EFFECTS: n — n n

Duplicate the value on the stack.

ELSE CORE

COMPILES TO: BRANCH (<address of matching THEN token>)

SYNTAX: IF<space>..<space>ELSE<space>..<space>THEN

At runtime ELSE executes after the true following IF. ELSE forces execution to skip over the following false part and resumes execution after the THEN.

EMIT CORE

COMPILES TO: EMIT (0)

STACK EFFECTS: c —

Transmit the ASCII character with code n to the selected output device.

EMPTY-BUFFERS BLOCK EXT

INCLUDE: ansblock.4th

STACK EFFECTS: —

Unassign all block buffers. Do not transfer the contents of any UPDATEd block buffer to mass storage.

END-STRUCT 4TH

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>END-STRUCT<space><name>

COMPILER: The previously compiled literal is taken as an argument for END-STRUCT, creating a constant that holds the length of the STRUCT. The instruction pointer is decremented, actually deleting the literal.

FORTH: Similar constructions are available in gForth. +FIELD is part of the Forth 200x draft.

Terminate the definition of a STRUCT. The created structure is an constant that memorizes the size of the structure (see: *+FIELD, STRUCT*).

ENUM 4TH

SYNTAX: <literal><space>ENUM<space><name>

COMPILER: The previously compiled literal is taken as an argument for ENUM and incremented afterwards. The instruction pointer is left unchanged.

FORTH: This word is available in some Forths.

A defining word used to create word <name>. When <name> is later executed, it will push the value of <literal> on the stack.

ENVIRONMENT? CORE

PRONUNCIATION: environment-query

INCLUDE: environ.4th

STACK EFFECTS: addr n — -f

addr is the address of a character string and n is the string's character count. The character string should contain a keyword from ANS-Forth environmental queries or the optional word sets to be checked for correspondence with an attribute of the present environment. The system treats the attribute as unknown, the returned flag is false.

ERASE CORE EXT

COMPILES TO: LITERAL (0)

FILL (0)

STACK EFFECTS: addr n —

If n is greater than zero, clear all bits in each of n consecutive address units of memory beginning at addr.

EVALUATE CORE

INCLUDE: evaluate.4th

STACK EFFECTS: addr n —

FORTH: In Forth, the entire dictionary is available. In 4tH, the only words available are explicitly defined by the program.

Make the string described by addr and n the input buffer and interpret. Other stack effects are due to the words EVALUATEd.

EXECUTE CORE

COMPILES TO: EXECUTE (0)

STACK EFFECTS: xt —

Execute the colon definition whose token-address xt is on the stack. The current token-address is pushed on the returnstack.

EXIT CORE

COMPILES TO: EXIT (0)

When compiled within a colon-definition, terminates execution of that definition at that point. At runtime functionally equivalent to ;.

EXPECT CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: addr n —

Receive a string of at most n-1 characters. The editing functions, if any, that the system performs in order to construct the string of characters are implementation-defined. Input terminates when an implementation-defined line terminator is received or when the string is n-1 characters long. When input terminates the display is maintained in an implementation-defined way. Store the string at addr and its length in SPAN (see *SPAN*).

FALSE CORE EXT

COMPILES TO: LITERAL (<false>)

STACK EFFECTS: — -f

Returns a FALSE flag on the stack.

FILE 4TH

COMPILES TO: LITERAL ((ERROR))

TO (<variable address>)

SYNTAX: FILE<space><name>

STACK EFFECTS: —

Create a value name which will hold a filehandle. The initial value will trigger an error if used before proper assignment.

FILE-POSITION FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — n f

n is the current file position for the file identified by handle h. Flag f is the implementation-defined I/O result code. n is undefined if f is non-zero.

FILE-SIZE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: h — n f

n is the size, in characters, of the file identified by handle h. Flag f is the implementation-defined I/O result code. This operation does not affect the value returned by FILE-POSITION. n is undefined if f is true.

FILE-STATUS FILE EXT

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 — n2 f

Return the status of the file identified by the character string addr n1. If the file exists, flag f is zero; otherwise flag f is the implementation-defined I/O result code. n2 contains implementation defined information about the file.

FILES 4TH

COMPILES TO: LITERAL (<number of open files>)

STACK_EFFECTS: — n

Returns the maximum number of open streams 4tH can handle. Two of these streams are predefined, STDIN and STDOUT.

FILL CORE

COMPILES TO: FILL (0)

STACK EFFECTS: addr n c —

Fills n bytes in the Character Segment, beginning at address addr, with character c.

FIRST 4TH

COMPILES TO: ENVIRON (<address of FIRST>)

STACK EFFECTS: — x

Leaves the variable address x of the first user-variable. If FIRST is greater than LAST, no user-variables have been defined.

FLUSH BLOCK

INCLUDE: ansblock.4th

STACK EFFECTS: —

Perform the function of SAVE-BUFFERS, then unassign the block buffer.

FLUSH-FILE FILE EXT

INCLUDE: ansfile.4th

STACK EFFECTS: h — f

Attempt to force any buffered information written to the file referred to by handle h to be written to mass storage, and the size information for the file to be recorded in the storage directory if changed. If the operation is successful, f is zero. Otherwise, it is an implementation-defined I/O result code.

FM/MOD CORE

PRONUNCIATION: f-m-slash-mod

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 — n2 n3

Divide d1 by n1, giving the floored quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

FREE MEMORY

INCLUDE: ansmem.4th

STACK EFFECTS: addr — f

Return the contiguous region of data space indicated by addr to the system for later allocation. addr shall indicate a region of data space that was previously obtained by ALLOCATE or RESIZE. If the operation succeeds, f is false. If the operation fails, f is true.

HERE CORE

COMPILES TO: LITERAL (<token address>)

STACK EFFECTS: — xt

FORTH: Leaves the address of the next available dictionary location. Since 4tH doesn't have a dictionary location, its use is very different.

At runtime, HERE leaves the address xt in the Code Segment where it was compiled.

HEX CORE EXT

COMPILES TO: RADIX (16)

FORTH: In Forth this construction

```
HEX : SOMETIN 16 ;
```

will compile 16 as a hexadecimal number. In 4tH it will simply be compiled and 16 will be compiled as a decimal number. To emulate this construction use

```
[HEX] : SOMETIN 16 ;
```

instead.

Set the numeric conversion BASE for hexadecimal output at runtime.

HI 4TH

COMPILES TO: ENVIRON (<address of HI>)

STACK EFFECTS: — addr

Leaves the address of the last character in the Character Segment.

HIDE 4TH

SYNTAX: HIDE<space><name>

Find <name>, then delete name from the symbol table. Used to create private definitions.

HOLD CORE

COMPILES TO: HOLD (0)

STACK EFFECTS: c —

Used between <# and #> to insert an ASCII character into a pictured numeric output string, e.g. `[HEX] 2E HOLD` will place a decimal point.

I CORE

COMPILES TO: I (0)

STACK EFFECTS: — n

Used with a DO .. LOOP to copy the loop index to the stack. An alias for R.

IF CORE

COMPILES TO: 0BRANCH (<address of matching ELSE|THEN token>)

STACK EFFECTS: f —

SYNTAX: See *ELSE, THEN*

At runtime, IF selects execution based on f. If f is non-zero, execution continues ahead through the true part. If f is zero execution skips till just after ELSE to execute the false part. After each part, execution resumes after THEN.

IMMEDIATE CORE

COMPILER: The instruction pointer is not incremented. In fact, IMMEDIATE is a dummy.

STACK EFFECTS: —

Make the most recent definition an immediate word.

INCLUDE COMUS

SYNTAX: INCLUDE<space><string><space>

COMPILER: The contents of the file are inserted at this position.

An alias for [NEEDS (see: *[NEEDS*).

INPUT 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method on the stack, signalling an operation on an input-device.

INVERT CORE

COMPILES TO: INVERT (0)

STACK EFFECTS: n1 — n2

Leave n1's binary complement as n2. This word is *not* equivalent to 0=.

IS CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: xt —

SYNTAX: IS<space><name>

Store xt in the value identified by name, previously defined by DEFER (see: *DEFER*).

J CORE

COMPILES TO: J (0)

STACK EFFECTS: — n

Used with an embedded DO .. +LOOP to copy the outer loop index to the stack. Copies in fact the third item of the returnstack.

LAST 4TH

COMPILES TO: ENVIRON (<address of LAST>)

STACK EFFECTS: — x

Leaves the variable address x of the last variable in the Variable Area.

LEAVE CORE

COMPILES TO: LEAVE (0)

Force termination of a DO .. +LOOP at the next opportunity by setting the loop index equal to the loop limit. The limit itself remains unchanged, and execution proceeds normally until +LOOP is encountered.

LIST BLOCK EXT

INCLUDE: ansblock.4th

STACK EFFECTS: n —

Display block n in an implementation-defined format. Store n in SCR.

LO 4TH

COMPILES TO: LITERAL (<TIB+PAD>)

STACK EFFECTS: — addr

Leaves the offset of the first character of the Allocation Area in the Character Segment. If LO is greater than HI, no memory has been allocated.

LOAD BLOCK

INCLUDE: ansblock.4th

BEFORE: evaluate.4th

STACK EFFECTS: n —

FORTH: In Forth, the entire dictionary is available. In 4tH, the only words available are explicitly defined by the program.

Save the current input-source specification. Store n in BLK (thus making block n the input source and setting the input buffer to encompass its contents), set >IN to zero, and execute EVALUATE. When the parse area is exhausted, restore the prior input source specification.

LOOP CORE

COMPILES TO: LOOP (<address of matching DO token>)

SYNTAX: DO<space>..<space>+LOOP

Used in the form DO .. LOOP. At runtime, LOOP selectively controls branching back to the corresponding DO based on the loop index and the loop limit. The index is incrementex and compared to the limit. The branch back to DO occurs until the new index is equal to the limit. Upon exiting the loop, the parameters are discarded and execution continues ahead.

LSHIFT CORE

PRONUNCIATION: l-shift

COMPILES TO: SHIFT (0)

STACK EFFECTS: n1 n2 — n3

Performs a logical bit shift on n1. Specifically, SHIFT shifts a number a number of bits, specified in n2, using a logical register shift. An alias for SHIFT.

M* CORE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: n1 n2 — d

d is the signed product of n1 times n2.

M*/ DOUBLE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 n2 — d2

Multiply d1 by n1 producing the triple-cell intermediate result t. Divide t by n2 giving the double-cell quotient d2. An ambiguous condition exists if n2 is zero or negative, or the quotient lies outside of the range of a double-precision signed integer.

M+ DOUBLE

PRONUNCIATION: m-star

INCLUDE: mixed.4th

STACK EFFECTS: d1 n1 — d2

Add n1 to d1, giving the sum d2.

MAX CORE

COMPILES TO: MAX (0)

STACK EFFECTS: n1 n2 — n3

Leave n3 as the greater of the two numbers n1 and n2.

MAX-N COMUS

COMPILES TO: LITERAL (<largest positive integer>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: MAX-N S" MAX-N" ENVIRONMENT? DROP ;
```

Returns the largest positive integer that 4tH can handle. Usually 2^31.

MAX-RAND 4TH

COMPILES TO: LITERAL (<largest integer returned by RANDOM>)

STACK EFFECT: — n

Returns the largest positive integer that RANDOM can return.

MIN CORE

COMPILES TO: MIN (0)

STACK EFFECTS: n1 n2 — n3

Leave n3 as the smaller of the two numbers n1 and n2.

MOD CORE

COMPILES TO: MOD (0)

STACK EFFECTS: n1 n2 — n3

Leave the remainder of n1/n2 with the same sign as n1 in n3.

MOVE CORE

COMPILES TO: CMOVE (0)

STACK EFFECTS: addr1 addr2 n —

Move the specified quantity of bytes (n) beginning at address addr1 to addr2 in the Character Segment.

MS FACILITY EXT

INCLUDE: ansfacil.4th

STACK EFFECTS: n —

FORTH: In Forth, the resolution is significantly higher than between +0 and +1999 ms.

Wait at least u milliseconds.

NEGATE CORE

COMPILES TO: NEGATE (0)

STACK EFFECTS: n1 — -n1

Leave n1 negated (two's complement).

NIP CORE EXT

COMPILES TO: SWAP (0)

DROP (0)

STACK EFFECTS: n1 n2 — n2

Drop the first item below the top of stack.

NOT COMUS

COMPILES TO: 0= (0)

STACK EFFECTS: n — f

An alias for 0= (see: *0=*).

NUMBER 4TH

COMPILES TO: NUMBER (0)

STACK EFFECTS: addr n1 — n2

FORTH: Some Forths support this word too, but issue a message on error.

Convert an string at offset addr with length n1 in the Character Segment to number n2. If numeric conversion is not possible (ERROR) is left on the stack.

OCTAL 4TH

COMPILES TO: RADIX (8)

FORTH: See *HEX*.

Set the numeric conversion BASE for octal output at runtime.

OFFSET 4TH

SYNTAX: OFFSET<space><name>

FORTH: Equivalent to:

```
: OFFSET CREATE DOES> SWAP CHARS + C@ ;
```

Leaves <name> in the symboltable and replaces further occurences of <name> with an execution procedure which takes an index from the stack and leaves the character concerned on the stack.

OMIT 4TH

COMPILES TO: OMIT (0)

STACK EFFECTS: c —

Skips all leading delimiters in the Character Segment, using character c as a delimiter.

OPEN 4TH

COMPILES TO: OPEN (0)

STACK EFFECTS: addr n fam — h | -f

OPEN will open the file, which name has been specified by an ASCIIZ string, starting at offset addr in the Character Segment and having length n. Depending on the file access method, the file or pipe will be opened for reading, otherwise for writing. If the file or pipe was succesfully opened it will be connected to a stream and a filehandle will be left on the stack. If not, a FALSE flag will be left on the stack. Note that OPEN does not connect a stream to a channel (see: *USE*).

OPEN-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n fam — h f

Open the file named in the character string specified by addr n, with file access method indicated by fam. The meaning of values of fam is implementation defined. If the file is successfully opened, flag f is zero, handle h is its identifier, and the file has been positioned to the start of the file. Otherwise, f is the implementation-defined I/O result code and h is undefined.

OR CORE

COMPILES TO: OR (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical OR in n3 of the numbers n1 and n2.

OUT COMUS

COMPILES TO: LITERAL (<address of OUT>)

STACK EFFECTS: — x

A variable containing the the value that will be returned to the host program.

OUTPUT 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method on the stack, signalling an operation on an output-device.

OVER CORE

COMPILES TO: OVER (0)

STACK EFFECTS: n1 n2 — n1 n2 n1

Copy the second stack value to the top of the stack.

PAD CORE EXT

COMPILES TO: LITERAL (<address of PAD>)

STACK EFFECTS: — addr

Leave the address of the text output buffer.

PARSE CORE EXT

COMPILES TO: PARSE (0)

STACK EFFECTS: c — addr n

Reads a string from the Character Segment, using character c as a delimiter. Leaves the addr/count pair addr n. The resulting string is *not* zero-terminated. If the parse area was empty, the resulting string has a zero length.

PARSE-WORD 4TH

COMPILES TO: DUP (0)

OMIT (0)

PARSE (0)

STACK EFFECTS: c — addr n

Reads a string from the Character Segment, using character c as a delimiter and skipping all leading delimiters. Leaves the addr/count pair addr n. The resulting string is *not* zero-terminated.

PAUSE 4TH

COMPILES TO: PAUSE (0)

STACK_EFFECTS: —

Saves a stackframe, closes all files and quits execution. Leaves the virtual machine in a state where it can resume execution.

PICK CORE EXT

INCLUDE: anscore.4th

STACK EFFECTS: nu .. n1 n2 u — nu .. n1 n2 nu

Remove u. Copy the nu to the top of the stack.

PIPE 4TH

COMPILES TO: LITERAL (<fam>)

STACK EFFECTS: — fam

This will leave a file access method modifier on the stack, signalling an operation on a pipe. Must be added to another file access modifier. Used in combination with INPUT and OUTPUT. If an OS does not support pipes, opening a pipe will always fail.

PLACE COMUS

COMPILES TO: PLACE (0)

STACK EFFECTS: addr1 n addr2 —

Copies the string at address addr1 with count n to address addr2.

QUERY CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: —

Make the user input device the input source. Receive input into the terminal input buffer, replacing any previous contents. Make the result, whose address is returned by TIB, the input buffer. Set >IN to zero.

QUIT CORE

COMPILES TO: QUIT (0)

FORTH: This word has quite another meaning in Forth.

Sets the program counter to the end of the program. Effectively quits execution.

R> CORE

PRONUNCIATION: r-from

COMPILES TO: R> (0)

STACK EFFECTS: — n

Remove the top value from the return stack and leave it on the stack.

R'@ TOOLBELT

COMPILES TO: R> (0)

I (0)

SWAP (0)

>R (0)

STACK EFFECTS: — n

Copy the second return stack item to the stack.

R/O FILE

PRONUNCIATION: r-o

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the read only file access method.

R/W FILE

PRONUNCIATION: r-w

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the read write file access method.

R@ CORE

PRONUNCIATION: r-fetch

COMPILES TO: I (0)

STACK EFFECTS: — n

Copy the top of the return stack to the stack.

RANDOM COMUS

COMPILES TO: RANDOM (0)

STACK EFFECTS: — n

Returns a pseudo-random number in n, between 0 and MAX-RAND. Seed is automatically set.

READ-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 h — n2 f

Read n1 consecutive characters to addr from the current position of the file identified by handle h. If n1 characters are read without an exception, flag f is zero and n2 is equal to n1. If the end of the file is reached before n1 characters are read, flag f is zero and n2 is the number of characters actually read. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

READ-LINE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n1 h — n2 f1 f2

Read the next line from the file specified by handle h into memory at the address addr. At most n1 characters are read. Up to two implementation-defined line terminating characters may be read into memory at the end of the line, but are not included in the count n2. The line buffer provided by addr should be at least n1+2 characters long. If the operation succeeded, flag f1 is true and flag f2 is zero. If a line terminator was received before n1 characters were read, then n2 is the number of characters, not including the line terminator, actually read (0 <= n2 <= n1). When n1 = n2 the line terminator has yet to be reached. If the operation is initiated when the value returned by FILE-POSITION is equal to the value returned by FILE-SIZE for the file identified by handle h, flag f1 is false, flag f2 is zero, and n2 is zero. If flag f2 is non-zero, an exception occurred during the operation and f2 is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character read.

RECURSE CORE

COMPILES TO: CALL (<last defined word>)

Compile a call to the current colon-definition inside the current colon-definition. If this word is used outside a colon definition it is undefined.

REFILL CORE EXT FILE EXT

COMPILES TO: REFILL (0)

STACK EFFECTS: — f

Attempt to fill the input buffer from the input source, returning a true flag if successful. When the input source is the user input device, attempt to receive input into the terminal input buffer. When the input source is a text file, attempt to read the next line from the text-input file. If successful, make the result the input buffer, set >IN to zero, and return true. Receipt of a line containing no characters is considered successful. If there is no input available from the current input source, return false.

REPEAT CORE

COMPILES TO: BRANCH (<address of matching BEGIN>)

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>REPEAT

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime, REPEAT forces an unconditional branch back to just after the corresponding BEGIN. Multiple WHILEs may be used.

REPOSITION-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: n h — f

Reposition the file identified by handle h to n. Flag f is the implementation-defined I/O result code. An ambiguous condition exists if the file is positioned outside the file boundaries. At the conclusion of the operation, FILE-POSITION returns the value n.

RESIZE MEMORY

INCLUDE: ansmem.4th

STACK EFFECTS: addr1 n — addr2 f

Change the allocation of the contiguous data space starting at the address addr1, previously allocated by ALLOCATE or RESIZE, to n address units. n may be either larger or smaller than the current size of the region. If the operation succeeds, addr2 is the aligned starting address of n address units of allocated memory and f is false. The values contained in the region at addr1 are copied to addr2, up to the minimum size of either of the two regions. If they are the same, the values contained in the region are preserved to the minimum of n or the original size. If addr2 is not the same as addr1, the region of memory at addr1 is returned to the system according to the operation of FREE. If the operation fails, addr2 equals addr1, the region of memory at addr1 is unaffected, and f is true.

RESTORE-INPUT CORE EXT

INCLUDE: evaluate.4th

STACK EFFECTS: n1 n2 a1 n3 h n4 — f

Attempt to restore the input source specification to the state described by n1 through h. Flag is true if the input source specification cannot be so restored.

ROLL CORE EXT

INCLUDE: anscore.4th

STACK EFFECTS: nu n1 .. n2 u — n1 .. n2 nu

Remove u. Rotate u+1 items on the top of the stack.

ROT CORE

PRONUNCIATION: rote

COMPILES TO: ROT (0)

STACK EFFECTS: n1 n2 n3 — n2 n3 n1

Rotate the top three values on the stack, bringing the third to the top.

RP@ 4TH

COMPILES TO: RP@ (0)

STACK EFFECTS: — sp

Return the address sp of the stack position of the top of the return stack as it was before RP@ was executed.

RSHIFT CORE

PRONUNCIATION: r-shift

COMPILES TO: NEGATE (0)

SHIFT (0)

STACK EFFECTS: n1 n2 — n3)

Perform a logical right shift of n2 bit-places on n1, giving n2. Put zeroes into the most significant bits vacated by the shift.

S" CORE FILE

PRONUNCIATION: s-quote

COMPILES TO: S" (<address of string constant>)

SYNTAX: S"<space><string>"

STACK EFFECTS: — addr n

Compiles string delimited by " in the String Segment with an execution procedure to move the string to PAD. Leaves the address and the length of the string on the stack.

S>D CORE

PRONUNCIATION: s-to-d

COMPILER: The instruction pointer is not incremented. In fact, S>D is a dummy.

STACK EFFECTS: n — n

Convert the number n to double number n with the same numerical value.

S| 4TH

COMPILES TO: S" (<address of string constant>)

SYNTAX: S|<space><string>|

STACK EFFECTS: — addr n

Compiles string delimited by | in the String Segment with an execution procedure to move the string to PAD. Leaves the address and the length of the string on the stack.

SAVE-BUFFERS BLOCK

INCLUDE: ansblock.4th

STACK EFFECTS: —

Transfer the contents of each UPDATEd block buffer to mass storage. Mark the buffer as unmodified.

SAVE-INPUT CORE EXT

INCLUDE: evaluate.4th

STACK EFFECTS: — n1 n2 a1 n3 h n4

n1 through h describe the current state of the input source specification for later use by RESTORE-INPUT.

SCONSTANT 4TH

SYNTAX: S"<space><string>"<space>SCONSTANT<space><name>

COMPILER: The previously compiled string address is taken as an argument for SCONSTANT. The instruction pointer is decremented, actually deleting the string address.

A defining word used to create word <name>. When <name> is later executed, it will push the current address and the length of the string constant on the stack.

SCR BLOCK EXT

PRONUNCIATION: s-c-r

INCLUDE: ansblock.4th

STACK EFFECTS: — x

x is the address of a cell containing the block number of the block most recently LISTed.

SEARCH STRING

INCLUDE: search.4th

STACK EFFECTS: addr1 n1 addr2 n2 — addr3 n3 f

Search the string specified by addr1 n1 for the string specified by addr2 n2 . If flag is true, a match was found at addr3 with n3 characters remaining. If flag is false there was no match and addr3 is addr1 and n3 is n1.

SEEK 4TH

COMPILES TO: SEEK (0)

STACK EFFECTS: n h — f

Reposition the file identified by handle h to n. If n is positive, TELL returns the value n. If n is negative, the file is repositioned relative to the end of the file. If the operation is successful, FALSE is returned, otherwise TRUE.

SIGN CORE

COMPILES TO: SIGN (0)

STACK EFFECTS: n1 n2 — n2

Stores an ASCII '-' sign just before the converted numeric output string in PAD when n1 is negative. n1 is discarded, but n2 is maintained. Must be used between <# and #>.

SM/REM CORE

PRONUNCIATION: s-m-slash-rem

INCLUDE: mixed.4th

STACK EFFECTS: d n1 — n2 n3

Divide d by n1, giving the symmetric quotient n3 and the remainder n2. Input and output stack arguments are signed. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell signed integer.

SOURCE CORE

COMPILES TO: LITERAL (<address of TIB variable>)

@ (0)

LITERAL (<address of TIB-size variable>)

@ (0)

STACK EFFECTS: — addr n

addr is the address of, and n is the number of characters in, the currently used TIB.

SOURCE! SOURCEFORGE

COMPILES TO: LITERAL (<address of TIB-size variable>)

! (0)

LITERAL (<address of TIB variable>)

! (0)

STACK EFFECTS: addr n —

Make the string described by c-addr and u the current input buffer. Set >IN to zero. A program is allowed to refill the input buffer without restoring the original input source; upon a refill, the system shall accept the new portion of text to the current refill buffer and make it the input buffer.

SOURCE-ID CORE EXT FILE

PRONUNCIATION: source-i-d

COMPILES TO: ENVIRON (<address of CIN>)

STACK EFFECTS: — n

Identifies the input source.

SP@ 4TH

COMPILES TO: SP@ (0)

STACK EFFECTS: — sp

Return the address sp of the stack position of the top of the stack as it was before SP@ was executed.

SPACE CORE

COMPILES TO: LITERAL (<ASCII value of space>)

EMIT (0)

Transmit an ASCII blank to the current output device.

SPACES CORE

COMPILES TO: SPACES (0)

STACK EFFECTS: n —

Transmit n ASCII blanks to the current output device.

SPAN CORE EXT

INCLUDE: obsolete.4th

STACK EFFECTS: — x

X is the address of a cell containing the count of characters stored by the last execution of EXPECT (see *EXPECT*).

STACK-CELLS 4TH

COMPILES TO: LITERAL (<number of integers>)

STACK EFFECTS: — n

FORTH: Equivalent to:

```
: STACK-CELLS S" STACK-CELLS" ENVIRONMENT? DROP ;
```

Returns the number of integers that the Stack Area can contain. Both stacks share the Stack Area.

STDIN 4TH

COMPILES TO: LITERAL (<address of stream>)

STACK EFFECTS: — h

Leaves a filehandle on the stack associated with the standard keyboard input device. This stream cannot be closed.

STDOUT 4TH

COMPILES TO: LITERAL (<address of stream>)

STACK EFFECTS: — h

Leaves a filehandle on the stack associated with the standard screen output device. This stream cannot be closed.

STRING 4TH

SYNTAX: <literal><space>STRING<space><name>

COMPILER: The previously compiled literal is taken as an argument for STRING. The instruction pointer is decremented, actually deleting the literal.

FORTH: This word is 4tH specific. Roughly equivalent to:

```
: STRING CREATE CHARS ALLOT ;
```

Allocate <literal> characters of contiguous data space beginning at <name> in the Character Segment. The initial content of the allocated space is undefined.

STRUCT 4TH

COMPILES TO: LITERAL (0)

SYNTAX: STRUCT<space><literal><space>+FIELD<space><name><space>END-STRUCT<space><name>

STACK EFFECTS: — n

FORTH: Similar constructions are available in GForth. +FIELD is part of the Forth 200x draft.

A constant, which initiates a STRUCT definition (see: *+FIELD, END-STRUCT*).

SWAP CORE

COMPILES TO: SWAP (0)

STACK EFFECTS: n1 n2 — n2 n1

Exchange the top two values on the stack.

SYNC 4TH

COMPILES TO: SYNC (0)

STACK EFFECTS: —

Attempt to force any buffered information written to the device referred to by the output channel to be written.

TABLE 4TH

SYNTAX: TABLE<space><name>

FORTH: Available in some Forths.

Leaves <name> in the symboltable and replace further occurences with LITERAL <xt>. <xt> represents the address in the Code Segment where TABLE was compiled. An alias for CREATE.

TELL 4TH

COMPILES TO: TELL (0)

STACK EFFECTS: h — n

n is the current file position for the file identified by handle h.

TH COMUS

COMPILES TO: + (0)

STACK EFFECTS: x1 n — x2

FORTH: This word is not part of ANS-Forth or Forth-79, but can be found in other Forths. It can be very handy when porting 4tH programs. Just define TH as:

```
:  TH CELLS + ;
```

When you're using a construction like:

```
VAR 2 TH
```

In both 4tH and Forth the third element will be referenced. The use of TH to reference an element of a string in the Character Segment is allowed in 4tH, but the resulting source cannot be ported to Forth.

Used to reference an element in an array of integers. Will return the address of the n-th element in array x1 as x2. An alias for +.

THEN CORE

SYNTAX: IF<space>..<space>ELSE<space>..<space>THEN

At runtime THEN serves only as the destination of a forward branch from IF or ELSE. It marks the conclusion of the conditional structure.

THROW EXCEPTION

COMPILES TO: THROW (0)

STACK EFFECTS: n —

FORTH: The values of THROW are not conforming the ANS-Forth standard.

If n is non-zero, pop the topmost exception frame from the return stack, along with everything beyond that frame. Then adjust the return- and datastacks so they are the same as the depths saved in the exception frame, put n on top of the data stack, and transfer control to a point just after the CATCH that pushed that exception frame (see: *CATCH*).

TIB CORE EXT

PRONUNCIATION: t-i-b

COMPILES TO: LITERAL (<address of Terminal Input Buffer>)

STACK EFFECTS: — addr

FORTH: In Forth this is a variable. However, it is unlikely you'll ever find a program which assigns another value to it.

A constant which leaves the address of the Terminal Input Buffer on the stack.

TIME 4TH

COMPILES TO: TIME (0)

STACK EFFECTS: — n

Returns the number of seconds since January 1st, 1970.

TIME&DATE FACILITY

PRONUNCIATION: time-and-date

INCLUDE: ansfacil.4th

STACK EFFECTS: — n1 n2 n3 n4 n5 n6

Return the current time and date. n1 is the second {0...59}, n2 is the minute {0...59}, n3 is the hour {0...23}, n4 is the day {1...31}, n5 is the month {1...12}, and n6 is the year (e.g., 1991).

TO CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: n —

SYNTAX: TO<space><name>

Store n in the value identified by name.

TRUE CORE EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

FORTH: In ANS-Forth TRUE is represented by a -1 value.

Returns a true flag on the stack.

TUCK CORE EXT

COMPILES TO: SWAP (0)

OVER (0)

STACK EFFECTS: n1 n2 — n2 n1 n2)

Copy the first (top) stack item below the second stack item.

TYPE CORE

COMPILES TO: TYPE (0)

STACK EFFECTS: addr n —

Transmit n characters from addr to the selected output device.

U. CORE

PRONUNCIATION: u-dot

INCLUDE: dbldot.4th

STACK EFFECTS: n —

Display n in free field format as an unsigned number.

U.R CORE EXT

PRONUNCIATION: u-dot-r

INCLUDE: dbldot.4th

STACK EFFECTS: n1 n2 —

Display unsigned number n1 right aligned in a field n2 characters wide. If the number of characters required to display n1 is greater than n2, all digits are displayed with no leading spaces in a field as wide as necessary.

U< CORE

PRONUNCIATION: u-less-than

INCLUDE: ansdbl.4th

STACK EFFECTS: n1 n2 — f

Flag f is true if and only if unsigned number n1 is less than unsigned number n2.

U> CORE EXT

PRONUNCIATION: u-greater-than

INCLUDE: ansdbl.4th

STACK EFFECTS: n1 n2 — f

Flag f is true if and only if unsigned number n1 is greater than unsigned number n2.

UM* CORE

PRONUNCIATION: u-m-star

INCLUDE: mixed.4th

STACK EFFECTS: n1 n2 — d

Multiply n1 by n2, giving the unsigned double-cell product d. All values and arithmetic are unsigned.

UM/MOD CORE

PRONUNCIATION: u-m-slash-mod

INCLUDE: mixed.4th

STACK EFFECTS: d n1 — n2 n3

Divide d by n1, giving the quotient n3 and the remainder n2. All values and arithmetic are unsigned. An ambiguous condition exists if n1 is zero or if the quotient lies outside the range of a single-cell unsigned integer.

UNLOOP CORE

COMPILES TO: R> (0)

R> (0)

DROP (0)

DROP (0)

STACK EFFECTS: —

Discard the loop-control parameters for the current nesting level. An UNLOOP is required for each nesting level before the definition may be EXITed.

UNTIL CORE

COMPILES TO: 0BRANCH (<address of matching BEGIN>)

STACK EFFECTS: f —

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>UNTIL

FORTH: The optional WHILE word is not supported.

At runtime UNTIL controls the conditional branch back to the corresponding BEGIN. If f is FALSE execution returns to just after BEGIN; if f is TRUE execution continues ahead.

UPDATE BLOCK

INCLUDE: ansblock.4th

STACK EFFECTS: —

Mark the current block buffer as modified. An ambiguous condition exists if there is no current block buffer. UPDATE does not immediately cause I/O.

USE 4TH

COMPILES TO: USE (0)

STACK EFFECTS: h —

USE will associate the stream identified by filehandle h with the appropriate input- or output-channel, depending on the file access method used when opening the stream (see: *OPEN*). No streams are closed.

VALUE CORE EXT

COMPILES TO: TO (<variable address>)

STACK EFFECTS: n —

SYNTAX: <literal><space>VALUE<space><name>

Create a symboltable entry for the value name with an initial value n. At runtime, n will be placed on the stack. In 4tH it is an alias for TO.

VARIABLE CORE

SYNTAX: VARIABLE<space><name>

A defining word used to create variable <name>. When <name> is later executed, it will push the address <var> on the stack, so that a fetch or store may access this location.

VARS 4TH

COMPILES TO: VARS (0)

STACK EFFECTS: — x

This word returns the begin of the variables area.

W/O FILE

PRONUNCIATION: w-o

INCLUDE: ansfile.4th

STACK EFFECTS: — fam

fam is the implementation-defined value for selecting the write only file access method.

WHILE CORE

COMPILES TO: 0BRANCH (<address of matching REPEAT token>)

STACK EFFECTS: f —

SYNTAX: BEGIN<space>..<space>WHILE<space>..<space>REPEAT

BEGIN<space>..<space>WHILE<space>..<space>UNTIL

FORTH: Within a BEGIN .. REPEAT construct, multiple WHILEs may be used as well, but additional words are necessary to complete the construct.

At runtime, WHILE selects conditional execution based on number n. If f is TRUE, WHILE continues execution of the code thru to REPEAT, which branches back to BEGIN. If f is FALSE, execution skips to just after REPEAT, exiting the structure. Multiple WHILEs may be used.

WIDTH 4TH

COMPILES TO: LITERAL (<number of characters>)

STACK EFFECTS: — n

A constant which leaves the maximum number of characters, allowed in a <name> label.

WITHIN CORE EXT

INCLUDE: range.4th

STACK EFFECTS: n1 n2 n3 — f

Perform a comparison of a test value n1 with a lower limit n2 and an upper limit n3 , returning true if either (n2 < n3 and (n2 <= n1 and n1 < n3)) or (n2 > n3 and (n2 <= n1 or n1 < n3)) is true, returning false otherwise.

WORD CORE

INCLUDE: word.4th

STACK EFFECTS: c — addr

Skip leading delimiters and parse characters delimited by c. Addr is the address of the parsed word. If the parse area was empty or contained no characters other than the delimiter, the resulting string has a zero length.

WRITE-FILE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n h — f

Write n characters from addr to the file identified by handle h starting at its current position. Flag f is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character written to the file, and FILE-SIZE returns a value greater than or equal to the value returned by FILE-POSITION.

WRITE-LINE FILE

INCLUDE: ansfile.4th

STACK EFFECTS: addr n h — f

Write n characters from addr followed by the implementation-dependent line terminator to the file identified by handle h starting at its current position. Flag f is the implementation-defined I/O result code. At the conclusion of the operation, FILE-POSITION returns the next file position after the last character written to the file, and FILE-SIZE returns a value greater than or equal to the value returned by FILE-POSITION.

XOR CORE

PRONUNCIATION: x-or

COMPILES TO: XOR (0)

STACK EFFECTS: n1 n2 — n3

Leave the bitwise logical XOR of n1 XOR n2 as n3.

['] CORE

PRONUNCIATION: bracket-tick

COMPILES TO: LITERAL (<tok>)

SYNTAX: [’]<space><name>

STACK EFFECTS: — n | x | xt

FORTH: See ’.

Compile the value contents of the symboltable entry identified as symbol <name> as a literal. An alias for ’.

[*] 4TH

COMPILES TO: LITERAL (<product>)

SYNTAX: <literal><space><literal><space>[*]

COMPILER: Two previously compiled literals are taken as arguments, multiplied and their product recompiled. The instruction pointer is decremented, actually deleting the literals.

STACK EFFECTS: — n

FORTH: Equivalent to *.

[+] 4TH

COMPILES TO: LITERAL (<sum>)

SYNTAX: <literal><space><literal><space>[+]

COMPILER: Two previously compiled literals are taken as arguments, added and their sum recompiled. The instruction pointer is decremented, actually deleting the literals.

STACK EFFECTS: — n

FORTH: Equivalent to +.

[=] 4TH

COMPILES TO: LITERAL (<flag>)

SYNTAX: <literal><space><literal><space>[=]

COMPILER: Two previously compiled literals are taken as arguments and a true flag is recompiled when they are equal. The instruction pointer is decremented, actually deleting the literals.

STACK EFFECTS: — f

FORTH: Equivalent to =.

[ABORT] 4TH

FORTH: Roughly equivalent to:

```
[ ABORT ]
```

Compilation is aborted immediately.

[ASSERT] 4TH

Toggles assertions. Assertions are disabled by default (see: *ASSERT(* and *)*).

[BINARY] 4TH

FORTH: Roughly equivalent to:

```
[ 2 BASE ! ]
```

When encountered during compilation it will set the radix to binary. All subsequent literals will be interpreted as binary numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[CHAR] CORE

PRONUNCIATION: bracket-char

COMPILES TO: LITERAL (<ASCII-value of character>)

SYNTAX: [CHAR]<space><char>

STACK EFFECTS: — c

Compiles the ASCII-value of <char> as a literal. At runtime the value is thrown on the stack. An alias for CHAR.

[DECIMAL] 4TH

FORTH: Roughly equivalent to:

```
[ DECIMAL ]
```

When encountered during compilation it will set the radix to decimal. All subsequent literals will be interpreted as decimal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[DEFINED] SEARCH EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

SYNTAX: [DEFINED]<space><name>

If the name is defined, return TRUE, else return FALSE.

[HEX] 4TH

FORTH: Roughly equivalent to:

```
[ HEX ]
```

When encountered during compilation it will set the radix to hexadecimal. All subsequent literals will be interpreted as hexadecimal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[IF] TOOLS EXT

PRONUNCIATION: bracket-if

SYNTAX: <literal><space>[IF]<space><word>..<word><space>[THEN]

COMPILER: The previously compiled literal is taken as an argument for [IF]. The instruction pointer is decremented, actually deleting the literal.

FORTH: Forth pops a value from the stack. This is not possible in 4tH.

If the literal is nonzero, do nothing. Otherwise, skipping leading spaces, parse and discard space-delimited words from the source, including nested occurences of [IF] .. [THEN], until the word [THEN] has been parsed and discarded.

[NEEDS 4TH

SYNTAX: [NEEDS<space><string>]

COMPILER: The contents of the file are inserted at this position.

Open the file specified by <string> and include its contents at the current position. When the end of the file is reached, close the file and continue compilation. An error condition exists if the named file can not be opened, if an I/O exception occurs reading the file, or if an I/O exception occurs while closing the file.

[NEGATE] 4TH

COMPILES TO: LITERAL (<negation>)

SYNTAX: <literal><space>[NEGATE]

COMPILER: A previously compiled literal is taken as an argument, negated and recompiled. The instruction pointer is decremented, actually deleting the literal.

STACK EFFECTS: — -n

FORTH: Equivalent to NEGATE.

[NOT] 4TH

COMPILES TO: LITERAL (<flag>)

SYNTAX: <literal><space>[NOT]

COMPILER: A previously compiled literal is taken as an argument and a true flag is recompiled when it is equal to zero. The instruction pointer is decremented, actually deleting the literal.

STACK EFFECTS: — f

FORTH: Equivalent to 0=.

[OCTAL] 4TH

FORTH: Roughly equivalent to:

```
[ 8 BASE ! ]
```

When encountered during compilation it will set the radix to octal. All subsequent literals will be interpreted as octal numbers. Runtime behaviour will be controlled by HEX, OCTAL and DECIMAL.

[THEN] TOOLS EXT

PRONUNCIATION: bracket-then

Does nothing. Acts as a marker for [IF] (see: *[IF]*).

[UNDEFINED] SEARCH EXT

COMPILES TO: LITERAL (<flag>)

STACK EFFECTS: — f

SYNTAX: [UNDEFINED]<space><name>

If the name is defined, return FALSE, else return TRUE.

\ CORE EXT

PRONUNCIATION: backslash

SYNTAX: \<space><string>

The remainder of the line is discarded. Used for comment.

Chapter 12

Editor manual

12.1 Introduction

Forth organises its mass storage into "screens" of 1024 characters. Forth may have one screen in memory at a time for storing text. The screens are numbered, starting with screen 0.

Each screen is organised as 16 lines with 64 characters. The Forth screens are merely an arrangement of virtual memory and do not correspond to the screen format of the target machine. Due to this format, the use of the comment word '\' is not allowed. Use '(' instead.

12.2 Selecting a screen and input of text

After you've started an editing session, you need to select a screen to edit. The screen is given a number and selected by using:

```
n CLEAR (clear screen n and select for editing).
```

To input new text to screen after CLEAR, the P (put) command is used. Example:

```
0 P THIS IS HOW
1 P TO INPUT TEXT
2 P TO LINES 0, 1, 2 OF SELECTED SCREEN.
```

12.3 Line editing

During this description of the editor, reference is made to PAD. This is a text buffer which may hold a line of text to be found or deleted by a string editing command. Do not confuse this PAD with 4tHs PAD. It is only called that way by convention.

12.4 Line editing commands

n D Delete line n but hold it in PAD. Line 15 becomes free as all statements move up 1 line.

n E Erase line n with blanks.

n I Insert the text from PAD at line n, moving the old line n and following lines down. Line 15 is lost.

n H Hold line n at PAD (used by system more often than by user).

n R Replace line n with the text in PAD.

n S Spread at line n. Line n and following lines move down 1 line. Line n becomes blank. Line 15 is lost.

n T Display line n and copy it to PAD.

n P text Put 'text' at line n, overwriting its previous contents.

12.5 Screen editing commands

n LIST List screen n and select it for editing: if screen n is not the current screen, it will request to load from memory.

n CLEAR Clear screen n with blanks and select it for editing.

n INSERT Insert screen n. The current screen n and all screens following it are moved down. The last screen is lost.

n m COPY Copy the contents of screen n to screen m. The original contents of screen m are lost.

FLUSH Used at the end of an editing session to save the current screen to memory.

UNDO Used to reload the current screen again, thus undoing all changes since the last flush (triggered by CLEAR, FLUSH or LIST).

L List the current screen. The cursor line is relisted after the screen listing to show the cursor position.

12.6 Cursor control and string editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided from the user to position the cursor either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

12.7 Commands to position the cursor

n M Move the cursor by n characters and the cursor line. The position of the cursor on its line is shown by a ^ (caret).

n W Wipe n characters to the left of the cursor.

TOP Position the cursor at the start of the screen.

12.8 String editing commands

B Used after F to back up the cursor by the length of the most recent text.

C text Copy in text to the cursor line at the cursor position.

F text Search forward from the current cursor position until string 'text' is found. The cursor is left at the end of the string and the cursor line printed. If the string is not found an error message is given and the cursor repositioned to the top of the screen.

N Find the next occurrence of the string found by an F command

TILL text Delete on the cursor line from the cursor till the end of string text.

X text Find and delete the next occurrence of the string 'text'.

12.9 Saving and exiting

WRITE Saves the current contents of all screens to the block-file. No flushing is done.

WQ Flushes the current screen and saves the current contents of all screens to the block-file.

Q Quits the editor without saving.

12.10 Calculator mode

The calculator mode is a simulation of what is known as the "Forth calculator mode". You can use it to try out a host of 4tH words in interactive mode. It also serves nicely as a deskcalculator. You can freely mix editor and calculator commands.

We tried to include as many 4tH words as possible, although we had to modify some due to the limitations imposed by the system. There are eight pre-defined user-variables called "A." though "H.". You can use these variables like any other user-variable.

You cannot declare new variables or make any colon-definitions in interactive mode. If you are unclear how to use the built-in calculator please refer to the Primer and the Glossary. By convention, calculator mode uses "OK" as the prompt. The following table shows you which commands are available:

Editor	4th equivalent	Editor	4th equivalent
+	+	@	@
th	th	?	?
-	-	base!	base !
*	*	decimal	decimal
/	/	octal	octal
q	quit	binary	2 base !
quit	quit	.(<*string*>)	.(<*string*>)
bye	quit	mod	mod
.	.	abs	abs
.r	.r	negate	negate
drop	drop	invert	invert
dup	dup	min	min
rot	rot	max	max
swap	swap	or	or
over	over	and	and
A.	variable a. a.	xor	xor
B.	variable b. b.	lshift	lshift
C.	variable c. c.	rshift	rshift
D.	variable d. d.	depth	depth
E.	variable e. e.	cells	cells
F.	variable f. f.	1+	1+
G.	variable g. g.	cell+	cell+
H.	variable h. h.	1-	1-
!	!	cell-	cell-
+!	+!	space	space
random	random	spaces	spaces
wait	wait	2*	2*
time	time	2/	2/
char	char	/mod	/mod
[char]	[char]	*/	*/
emit	emit	*/mod	*/mod
cr	cr	(<*string*>)	(<*string*>)

Table 12.2: DC commands

Chapter 13

Shell manual

13.1 Introduction

The *4tsh* shell is a multitasking environment for 4tH. 4tH features cooperative multitasking, which means programs have to relinquish control to the shell using 'PAUSE', otherwise the program will keep in control. The best place to add 'PAUSE' is usually somewhere in a loop. 4tH comes with several example multitasking programs for you to try out. *4tsh* can be used as a command line replacement for *4th*, since you can enable multitasking in the editor.

4tsh is scriptable. Scripts are stored in blockfiles, because block I/O is completed within a single context. If you prefer to use your own editor, you need to convert your script to a blockfile. 4tH comes with a conversion program, called `txt2blk.4th`. Every twelve lines are converted to a block, leaving four additional lines for future modifications. Your lines should be limited to 63 characters or less.

When a script it loaded, the first block is executed automatically. By convention, the first block is block 0. When the execution of a block has completed, the script stops. You can call other blocks by using "LOAD". When the execution of a called block has completed, the execution of the previous block will resume at the point where execution was transferred to the called block. It is recommended to use the first block as an *application load screen*[1], e.g.

```
( 4tsh application load screen)
1 load  ( initialization)
2 load  ( checking conditions)
3 load  ( error handling)
```

An *application load screen* is simply a block that consecutively loads all the blocks that make up your script. You can run an arbitrary number of scripts at startup by issuing them on the command line, e.g.

```
4tsh boot.scr startup.scr tasks.scr
```

When all scripts have finished execution, control will automatically be transferred to the monitor.

[1] See "Thinking Forth", chapter 5.

13.2 Loading and saving

load" s"	Loads HX file **s** from disk and leaves the task number on the stack.
compile" s"	Loads and compiles source file **s** and leaves the task number on the stack.
n save" s"	Saves task number **n** to HX file **s**.
n write" s"	Generates C source file **s** from task number **n**.
n1 n2 see	Decompiles task number **n1** from opcode **n2** on.

13.3 Task management

mon	Leaves the task number of the monitor on the stack.
pause	Deactivates the monitor for one cycle.
n pauses	Deactivates the monitor for **n** cycles.
n run	Awakes and switches to task number **n**.
n awake	Awakes task number **n**.
n sleep	Deactivates task number **n**, but leaves it in memory.
n kill	Deactivates task number **n** and removes it from memory.
tasks	Lists all tasks.
halt	Kills all tasks and shuts down *4tsh*.

13.4 Scripting

script" s"	Run script **s**. Does only work in interactive mode.
n load	Load and interpret block **n**. Does not work in interactive mode.
:: s	Define label **s**.
goto s	Goto label **s**. Works in interactive mode, but only if **s** resides on the same line.
n if	Execute the words between *if* and the corresponding *then*, but only if **n** is non-zero. Works in interactive mode. If you fail to provide a corresponding *then* you will be prompted to provide it manually.
then	Marker for *if*.
n not	Leaves a non-zero value on the stack if **n** is zero, otherwise zero.
n status	Leaves the status of task number **n** on the stack.
done	Constant holding the termination status returned by *status*.
running	Constant holding the active status returned by *status*.
sleeping	Constant holding the inactive status returned by *status*.

13.5 Stack, I/O and arithmetic

4tsh	4TH EQUIVALENT	*4tsh*	4TH EQUIVALENT
+	+	.	.
-	-	dup	dup
*	*	rot	rot
/	/	over	over
cr	cr	swap	swap
.(	.(	drop	drop
(	(	=	=

Table 13.1: 4tsh commands

Chapter 14

ANS Forth statement

Forth, like BASIC, has always suffered from a lack - or may be an abundance of standards. Both languages had many dialects, which were highly incompatible. However, although there was never a generally accepted BASIC standard, a simple BASIC program can be easily converted to almost any existing implementation of the language.

The Forth community had a different approach to the problem. They kept changing the core every few years, so even now it's very hard to find a program which can run on any Forth with little modification. Calling those very different versions a standard didn't really help.

So when the ANSI-standard committee began its work they had a few very though nuts to crack. In our view the ANS-Forth standard is big step forward, but not perfect. It has not fully regained the simplicity we found in the Forth-79 and still has some serious flaws, although most are an inheritance from Forth-83.

We do feel the need for a real Forth standard, so we tried to make 4tH as ANS-Forth compatible as possible without sacrificing the ease of use that we had in mind when we designed it. About 95% of the CORE wordset is supported.

4tH was built according to the ANS-Forth standard, but with a tiny Forth-79 flavor. Full compliance to the ANS-Forth standard was never an objective. According to the ANS-Forth standard 4tH cannot be an "ANS-Forth System", since the standard does not cover this kind of implementation.

14.1 ANS-Forth Label

According to the ANS-Forth standard, section 5.2.2, this system is is capable of compiling:

ANS Forth Programs

Requiring:

- *the Exception word set*
- *the Memory word set*

Requiring selected words from:

- *the Core Extensions word set*

- *the Block word set*
- *the Block Extensions word set*
- *the Double number word set*
- *the Double number Extensions word set*
- *the Facility Extensions word set*
- *the File-Access word set*
- *the File-Access Extensions word set*
- *the Programming-Tools word set*
- *the Programming-Tools Extensions word set*
- *the String word set.*

End of label. Although the ANS-Forth standard (section 4.1) requires documentation to be presented in a prescribed format, 4tH does not comply for the simple reason that due to its architecture it is not considered to be a "ANS Forth System" (sections 3.3, 3.4, 5.1).

Note that due to this special architecture some words are missing from the CORE wordset or behave slightly different, so some "ANS Forth Programs" with the requirements mentioned above may not compile or compile only with modifications.

14.2 Unsupported CORE words

These words are not available in 4tH. Some CORE words are only available in source (ANS-Forth, section 3). You can find them in the 4tH glossary. The behaviour of some 4tH words may differ from the ANS-Forth definition.

ALLOT
FIND
KEY
LITERAL
POSTPONE
STATE
[
]

14.3 Supported ANS Forth word sets

The words in the following sections are supported by 4tH; *external words are in italics*. Please note that due to 4tHs special architecture some words may behave slightly different, so some "ANS Forth Programs" using these words may need modifications in order to run properly. More words are available in source and can be loaded when required.

14.3.1 Core Extensions word set

#TIB

.(

.R

0<>

0>

2>R

2R>

2R@

:NONAME

<>

?DO

AGAIN

ERASE

EXPECT

FALSE

HEX

NIP

PAD

PARSE

PICK

QUERY

REFILL

RESTORE-INPUT

ROLL

SAVE-INPUT

SOURCE-ID

TIB

TO

TRUE

TUCK

U.R

U>

VALUE

WITHIN

****

14.3.2 Block word set

BLK

BLOCK

BUFFER

FLUSH

LOAD

SAVE-BUFFERS

UPDATE

14.3.3 Block Extensions word set

EMPTY-BUFFERS

LIST

SCR

14.3.4 Double number word set

D+

D-

D.

D.R

D0<

D0=

D2*

D2/

D<

D=

D>S

DABS

DMAX

DMIN

DNEGATE

M+

M*/

14.3.5 Double number Extensions word set

2ROT

14.3.6 Facility Extensions word set

+FIELD
MS
TIME&DATE

14.3.7 File-Access word set

(
BIN
CLOSE-FILE
CREATE-FILE
FILE-POSITION
FILE-SIZE
OPEN-FILE
R/O
R/W
READ-FILE
READ-LINE
REPOSITION-FILE
S"
SOURCE-ID
W/O
WRITE-FILE
WRITE-LINE

14.3.8 File-Access Extensions word set

FILE-STATUS
FLUSH-FILE
REFILL

14.3.9 Programming-Tools word set

.S
?
DUMP

14.3.10 Programming-Tools Extensions word set

[IF]
[THEN]

14.3.11 String word set

-TRAILING

/STRING

BLANK

CMOVE

CMOVE>

COMPARE

SEARCH

Chapter 15

Errors guide

15.1 How to use this manual

This manual contains all the error messages 4tH can possibly issue. It is organized like this:

MESSAGE: This features the message from "errs_4th.c", the error-code returned in ErrNo and the C-mnemonic.

WORDS: Words that can trigger this error.

EXAMPLE: This features a 4tH one-liner that will trigger the error.

CAUSE: This lists all possible causes of the error.

HINTS: This will give you some directions on how to fix the error.

15.2 Interpreter (exec_4th)

When exiting this function ErrLine will contain the address of the word in the Code Segment where the error occured.

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully executed.

HINTS: Make an error ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the Character Segment or the Integer Segment.

HINTS:

1. Reduce the amount of memory your program allocates and recompile.
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another operating system (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: An unknown token was encountered in the H-code.

HINTS: Contact us, this should never happen.

MESSAGE: **Stack overflow** (*#3 M4SOVFLW*)

WORDS: Any word that pushes items on the Data Stack.

EXAMPLE: `STACK 1+ 0 DO I LOOP`

CAUSE: The Data Stack collided with the Return Stack.

HINTS:

1. Don't push too many elements on the Data Stack.
2. Merge colon-definitions. Reduce the number of nested DO..LOOPs.
3. If you are using recursion, try if you can achieve the same result with a loop.
4. Make sure that your stacks are still balanced when returning from a colon-definition. Don't leave any unused data on the Data Stack. Flow-control words can have unexpected stack effects!

MESSAGE: **Stack empty** (*#4 M4SEMPTY*)

WORDS: Any word that pops items from the Data Stack.

EXAMPLE: `0 SWAP`

CAUSE: The Data Stack did not contain the required number of items to complete the operation.

HINTS:

1. Make sure that your stack is still balanced when returning from a colon-definition.
2. Make sure that the required number of items are on the stack when performing the operation.

3. If the problem occurs within an interpreter driven application, make sure that you check the number of elements are on the stack before allowing the operation.

MESSAGE: **Return stack overflow** (*#5 M4ROVFLW*)

WORDS: Any word that pushes items on the Return Stack; calling a user defined word

EXAMPLE: `: DUMMY DUMMY ; DUMMY`

CAUSE: The Return Stack collided with the Data Stack.

HINTS:

1. Don't push too many elements on the Data Stack.
2. Merge colon-definitions. Reduce the number of nested DO..LOOPs.
3. If you are using recursion, try if you can achieve the same result with a loop.
4. Make sure that your stacks are still balanced when returning from a colon-definition. Don't leave any unused data on the Data Stack. Flow-control words can have unexpected stack effects!

MESSAGE: **Return stack empty** (*#6 M4REMPTY*)

WORDS: Any word that pops items from the Return Stack; returning from a user defined word

EXAMPLE: `R>`

CAUSE: The Return Stack did not contain the required number of items to complete the operation.

HINTS:

1. Balance R> and >R inside your colon-definition. Flow-control words can have unexpected stack effects!
2. Be careful when using R> and >R inside a DO..LOOP.

MESSAGE: **Bad string** (*#7 M4BADSTR*)

WORDS: `ARGS OFFSET`

EXAMPLE: -1 ARGS

CAUSE: There was either no argument on the command line or no binary string constant with this index.

HINTS:

1. Use a valid index for ARGS.
2. Use a valid index for the offset.

MESSAGE: **Bad variable** (*#8 M4BADVAR*)

WORDS: `! @ +! ?`

EXAMPLE: `6 ARRAY NAME NAME -5 TH @`

CAUSE: You tried to access a variable or array element, but its address in the Integer Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or stores are made.
2. Use a valid array index or address.

MESSAGE: **Bad address** (*#9 M4BADADR*)

WORDS: All string handling words

EXAMPLE: `10 STRING BUFFER TIB CHAR- BUFFER /TIB CMOVE`

CAUSE: You tried to access a character, but its address in the Character Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or stores are made.
2. Make sure that the number of elements is correct when you use words like CMOVE, COUNT, FILL.
3. Terminate strings.
4. You exceeded the maximum length of PAD when you defined a string constant using S".
5. You exceeded the maximum length of PAD when you fetched a commandline argument using ARGS.

MESSAGE: **Divide by zero** (*#10 M4DIVBY0*)

WORDS: `/ MOD /MOD */ */MOD`

EXAMPLE: `1 0 / . CR`

CAUSE: You tried to divide by zero.

HINTS: Check the divisor before you use it.

MESSAGE: **Bad token** (*#11 M4BADTOK*)

WORDS: `@C EXECUTE EXIT CATCH`

EXAMPLE: `: DUMMY ; ' DUMMY 5 - DUP @C SWAP EXECUTE`

CAUSE: You tried to jump to a token or access the argument of a token, but its address in the Code Segment is invalid.

HINTS:

1. Be sure that all stack-items are in the right order when address calculations, fetches or jumps are made.
2. Make sure the address you're using is within the Code Segment.
3. Be sure that the name after ' is that of a colon-definition.

MESSAGE: **Bad radix** (*#12 M4BADRDX*)

WORDS: `.R . # NUMBER`

EXAMPLE: `1 BASE ! 5 . CR`

CAUSE: The 4tH variable BASE contained a value outside the 2 to 36 range during a conversion.

HINTS: Take care that BASE stays within the 2 to 36 range.

MESSAGE: **Bad pointer** (*#13 M4BADPTR*)

WORDS: `CATCH THROW PAUSE`

EXAMPLE: `: ME R> R> R> DROP -5 >R >R >R 1 THROW ; ' ME CATCH`

CAUSE: The stack pointer THROW or PAUSE tried to use was invalid.

HINTS:

1. Be careful when you manipulate the Return Stack.
2. Contact us, this should never happen.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: All words performing I/O

EXAMPLE: `OUTPUT FILE 5 . CR`

CAUSE:

1. You tried to read from or write to an unopened file.
2. You tried to USE, SEEK or TELL an unused stream.
3. There was an I/O error when you tried to read from or write to a file.
4. There was an error when you tried to close an open file with CLOSE.
5. There was an error when 4tH tried to close a file after the program terminated.

HINTS:

1. Open a file before you try to read or write to it. Check the value OPEN returns.
2. Make sure the values on the stack are correct when you perform I/O.
3. Make sure the values on the stack are correct when addressing streams.

4. Make sure that there is enough space left on the device you try to write to. Make sure it functions correctly.

MESSAGE: **Assertion failed** (*#15 M4ASSERT*)

WORDS: `)`

EXAMPLE: `[ASSERT] ASSERT( FALSE )`

CAUSE: The top of the stack was FALSE when) executed.

HINTS: Correct the condition) acted upon.

MESSAGE: **Unhandled exception** (*#16 M4THROW*)

WORDS: `THROW`

EXAMPLE: `1 THROW`

CAUSE: A THROW was encountered without a previous call from CATCH. The top of stack contained an error number outside the range of system errors.

HINTS: Make sure that a THROW can only be reached from a previous CATCH.

MESSAGE: **Bad stream** (*#17 M4BADDEV*)

WORDS: `USE SEEK TELL CLOSE`

EXAMPLE: `-1 CLOSE`

CAUSE:

1. The filehandle you tried to use was out of range.
2. You may not SEEK, TELL or CLOSE the streams STDIN and STDOUT.
3. You may not SEEK or TELL a pipe.

HINTS:

1. Make sure you use a proper stream when using USE, SEEK, TELL or CLOSE.
2. Check stack manipulations or use a variable or value.

15.3 Compiler (comp_4th)

When exiting this function ErrLine will contain the address in the Code Segment where the next word would have been compiled if the error hadn't occured. This is logical, since 4tH always reports where the error occured. And all previous words have been succesfully compiled.

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A source was succesfully compiled.

HINTS: Make an error ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the H-code header, the Code Segment, the symbol-table or the control-stack.

HINTS:

1. Compact your source by removing all comment and whitespace.
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another compiler (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

WORDS: All defining words

EXAMPLE: Not applicable

CAUSE:

1. A word could not be compiled due to lack of space in the Code Segment.
2. A definition could not be compiled due to lack of space in the symbol-table.

HINTS: Contact us, this should never happen with normal source-code.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: `[NEEDS INCLUDE`

EXAMPLE: `[NEEDS nosuchfile.4th]`

CAUSE:

1. The source file you tried to read doesn't exist.
2. There was an error reading the source file.
3. There was an error when 4tH tried to close the source file.

HINTS:

1. Make sure the file you try to open exists and is in the path. Change your working directory if necessary. Check the *DIR4TH* environment variable.

2. Make sure the device functions correctly.

MESSAGE: **Bad literal** (*#18 M4BADLIT*)

WORDS: All words requiring a literal expression

EXAMPLE: `10 5 * ARRAY NAME`

CAUSE: The expression, which was compiled right before the word which caused the error, did not compile to a literal.

HINTS: Use a literal expression.

MESSAGE: **Undefined name** (*#19 M4NONAME*)

WORDS: <*name*> `' [']  RECURSE :THIS AKA HIDE`

EXAMPLE: `' HELLO ( "hello" is not defined)`

CAUSE:

1. The name which caused the error is not present in the symbol-table.
2. The name is not defined at all.
3. You tried to create a :THIS definition for an invalid datatype.
4. It is not a valid number in the current radix.
5. RECURSE is used outside a colon definition.
6. The name you used is longer than WIDTH characters.

HINTS:

1. Note that the words above only work with names defined inside the program and not with built-in names.
2. Usually a typo; correct spelling.
3. Use a proper datatype when creating a :THIS definition.
4. Set the appropriate radix by using [BINARY], [OCTAL], [DECIMAL] or [HEX].
5. Remove the offending RECURSE.
6. Use a shorter name.

MESSAGE: **Nesting too deep** (*#20 M4NONEST*)

WORDS: All flow control words and colon definitions

EXAMPLE: `10 0 DO 10 0 DO` <*more flow-control structures*> `LOOP LOOP`

CAUSE: The control-stack, that holds all references to addresses of flow-control structures in the Code Segment, overflowed.

HINTS: Make separate colon-definitions of the flow-control structures that caused the error.

MESSAGE: **No program** (*#21 M4NOPROG*)

WORDS: All words that do *not* compile any tokens

EXAMPLE: `10 ARRAY NAME ( Won't compile)`

CAUSE:

1. The source didn't contain any compilable words.
2. The source was corrupt.
3. A runaway comment or conditional compilation clause.
4. In rare cases use of reserved words as names.

HINTS:

1. Make a program that does something.
2. Make sure that the source actually contains 4tH source- code.
3. Terminate your comments and conditional compilations properly.
4. Don't use any reserved words as names.

MESSAGE: **Incomplete declaration** (*#22 M4NODECL*)

WORDS: All defining words and compiler directives

EXAMPLE: `10 CONSTANT CONSTANT NAME`

CAUSE:

1. Syntax errors; usually a missing name or a literal expression.
2. Incomplete compiler directives or expressions, like a leading comma or a trailing CHAR.
3. An assertion, beginning with ASSERT(, is missing a right parenthesis. Assertions are not enabled at that point.
4. An [IF] is not balanced by a [THEN].

HINTS:

1. Use an appropriate expression or name.
2. Complete compiler directives and expressions.
3. Add a right parenthesis at the end of the expression.
4. Add a [THEN] for each [IF] statement.

MESSAGE: **Unmatched conditional** (*#23 M4NOJUMP*)

WORDS: All flow control words and colon definitions

EXAMPLE: `: WRONG IF DROP BEGIN FALSE LOOP ;`

CAUSE: The flow-control word that caused the error didn't match with the previous flow-control word (BEGIN after IF) or was missing.

HINTS: Use the appropriate flow-control word to terminate a flow-control structure.

MESSAGE: **Unterminated string** (*#24 M4NOSTR*)

WORDS: `." \ ( .( ," S" ABORT" S| ,| [CHAR] CHAR @GOTO [NEEDS INCLUDE [DEFINED] [UNDEFINED]`

EXAMPLE: `." Hello world`

CAUSE:

1. A required delimiter is missing at the end of a string.
2. An internal error occured at the very end of the source.

HINTS:

1. Add the required delimiter at the end of the string.
2. Contact us, this should never happen.

MESSAGE: **Null string** (*#25 M4NULSTR*)

WORDS: *See error #24*

EXAMPLE: ." "

CAUSE:

1. The string between the word and its delimiter did not contain any characters.
2. There was more than one whitespace character between a [DEFINED], [UNDEFINED], [CHAR], CHAR or INCLUDE and the string following it.

HINTS:

1. Use a string that contains at least one single character.
2. Delete all superfluous whitespace characters between [DEFINED], [UNDEFINED], [CHAR], CHAR or INCLUDE and the string following it.

MESSAGE: **Duplicate name** (*#26 M4DUPNAM*)

WORDS: All defining words

EXAMPLE: `: TH CELLS + ;`

CAUSE: The name you used for a definition is already in use by 4tH or your own program.

HINTS: Use a different name.

MESSAGE: **Compilation aborted** (*#27 M4CABORT*)

WORDS: `[ABORT]`

EXAMPLE: `[ABORT]`

CAUSE: An [ABORT] directive was encountered during compilation.

HINTS: The original programmer must have had a reason to abort compilation in this particular circumstance. See the program for additional information.

15.4 Loader (load_4th)

Since the loader works with complete segments, the words don't have to do much with fixing an error. Therefore, it reports that nothing has been loaded (word 0) or everything has been loaded (the last word).

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully loaded.

HINTS: Keep up the good work. ;)

MESSAGE: **Out of memory** (*#1 M4NOMEM*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: There was not enough free memory to allocate the header, the Code Segment or the String Segment.

HINTS:

1. Reduce the amount of memory your program allocates and recompile.
2. Add more physical memory or increase swap space.
3. Recompile 4tH under another compiler (flat memory space) or another memory model.

MESSAGE: **Bad object** (*#2 M4BADOBJ*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. You tried to load a file, that was not an HX-file.
2. You tried to load an HX-file from a previous version of 4tH.
3. You tried to load an HX-file for a different application.
4. You tried to load an inconsistent HX-file.

HINTS:

1. Use a proper HX-file.
2. Recompile the source, using the current 4tH compiler.
3. If the source is compatible, you might recompile the source, using your own 4tH compiler.
4. Recompile the source, using your own 4tH compiler.

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. The file could not be opened.
2. There was an I/O error while the file was read.
3. The file could not be closed.

HINTS:

1. Use a valid filename.
2. Make sure the device functions correctly.
3. Make sure the device functions correctly.

15.5 Saver (save_4th)

Since the saver works with complete segments, the words don't have to do much with fixing an error. Therefore, it reports that nothing has been saved (word 0) or everything has been saved (the last word).

MESSAGE: **No errors** (*#0 M4NOERRS*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE: A program was succesfully saved.

HINTS: Keep up the good work. ;)

MESSAGE: **I/O error** (*#14 M4IOERR*)

WORDS: Not applicable

EXAMPLE: Not applicable

CAUSE:

1. The file could not be opened.
2. There was an I/O error while the file was written.
3. The file could not be closed.

HINTS:

1. Make sure you got enough inodes or directory-entries left on the device you want to write to. Use a valid filename.
2. Make sure that there is enough space left on the device you try to write to. Make sure it functions correctly.
3. Make sure the device functions correctly.

Chapter 16

Porting guide

16.1 Introduction

4tH is ANS-Forth compatible. That means that 4tH and ANS-Forth share a common word-set, so you can write programs that run on both systems. This guide will show you how you can write portable programs or convert eligible ANS-Forth programs to 4tH with as little effort as possible.

16.2 General guidelines

We have already stated that 4tH and ANS-Forth have much in common, but it is unlikely that you can write a non-trivial program that runs unmodified on both platforms without resorting to conditional compilation, which allows you to "hide" implementation specific code. The word '4TH#' not only holds 4tH's version number, but is also an effective way to differentiate between 4tH and other compilers:

```
[DEFINED] 4TH# [IF]
variable span
: expect 1- accept span ! ;
[THEN]
```

Of course, the opposite works too:

```
[UNDEFINED] 4TH# [IF]
s" easy.4th" included
[THEN]
```

If you have an interactive program you might want to disable the 4tH autostart:

```
[DEFINED] 4TH# [IF] start-program [THEN]
```

Otherwise 'REFILL' will try to get its input from the file instead of the keyboard.

16.3 Differences between 4tH and ANS-Forth

Like any software, 4tH is a compromise. We have to address the requirements of both newbies and power users, which means we have to make choices[1] concerning ANS-Forth compliancy. There are several reasons why 4tH is not completely ANS-Forth compliant:

1. 4tH uses a different architecture which makes it impossible to be ANS-Forth compliant, so some constructions are simply not feasible;
2. Some constructions in ANS-Forth are considered to be illogical, unelegant, bloated, not intuitive, error prone, inefficient or otherwise not acceptable;
3. 4tH maintains a close relationship with C, so it is more logical and efficient to use C-conventions instead of ANS-Forth conventions.

Where possible, we try to minimize the consequences for our users by hiding the differences behind abstractions or other transparent solutions. But sometimes, we simply can't. In this section we will show you which differences there are between 4tH and ANS-Forth and how you can either avoid or resolve them.

16.3.1 Strings

In 4tH, strings are stored in an ASCIIZ format. ANS-Forth uses counted strings. In 4tH there is no such thing as a countbyte, since it uses a terminator. If you limit the use of 'COUNT' only to string variables and constants, and exclusively use 'PLACE' or '+PLACE' you should be fine, since the address/count convention of ANS-Forth is fully supported. Should you resort to low level operations which require a terminator, you might have to define an equivalent word in ANS-Forth to make your program portable.

'S"' does have interpretation semantics, but the string stored at the address 'S"' returns might have a very short lifespan, depending on your ANS-Forth compiler. 4tH has a transparent, circular buffer that protects the string from overwriting, but when you port your program you might not be that lucky. Note that ANS-Forth does not require compilers to provide these facilities.

16.3.2 Double numbers

4tH uses only signed 32 bit cells, but some words in ANS-Forth, like '<#', '#>', 'FILE-SIZE', 'FILE-POSITION' and 'REPOSITION-FILE' require the use of double numbers. You can easily fix this by adding 'S>D', which converts a number to a double number. Its counterpart, 'D>S', is available too. In 4tH these words have no effect.

16.3.3 Booleans

Another nice topic for a flame war is the value of truth. In ANS-Forth the 'TRUE' has the value "-1", which means all bits are set. Which is very clever. You can 'XOR', 'OR', 'AND' and 'INVERT' it with any other value and it will behave as logical value. But "the all bits set" flag has its drawbacks too. Let's see what the ANS-Forth standard says about flags:

[1] You may or may not agree with the choices we made, but you can rest assured we have given them considerable thought.

> "A FALSE flag is a single-cell datum with all bits unset, and a TRUE flag is a single-cell datum with all bits set. While Forth words which test flags accept any non-null bit pattern as true, there exists the concept of the well-formed flag. If an operation whose result is to be used as a flag may produce any bit-mask other than TRUE or FALSE, the recommended discipline is to convert the result to a well-formed flag by means of the Forth word 0<> so that the result of any subsequent logical operations on the flag will be predictable. In addition to the words which move, fetch and store single-cell items, the following words are valid for operations on one or more flag data residing on the data stack: AND OR XOR INVERT"

We highly recommend the discipline of converting a non-zero value to a well-formed flag. But we don't understand why 'INVERT' is a valid way to manipulate a flag. We'll try to explain you why.

Forth traditionally has no specific logical operators. Instead, binary operators were used. This put 'INVERT' (or 'NOT' as it was called in Forth-79) in a difficult position. 'INVERT'ing any non-zero value will result in a non-zero value, except when all bits are set.

That is why '0=' was introduced, a full-fledged logical operator. So why use 'INVERT' when you want to perform a logical operation? Another quote:

> "Since a "char" can store small positive numbers and since the character data type is a sub-range of the unsigned integer data type, C! must store the n least-significant bits of a cell (8 <= n <= bits/cell). Given the enumeration of allowed number representations and their known encodings, "TRUE xx C! xx C@" must leave a stack item with some number of bits set, which will thus will be accepted as non-zero by IF."

This is another problem of using "all bits set" as a true flag: you store a well formed flag in an address unit that should easily be able to handle it and you'll never get it back. A flag is a boolean and can have two values: either true or false. The smallest unit that can hold a boolean is a bit. ANS-Forth programmers are denied that privilege.

But why are some Forth programmers so keen on their "all bits set" flag? Well, you can do neat things with it.

```
: >CHAR DUP 9 > 7 AND + ASCII 0 + ;
```

This will convert a digit to its ASCII representation. True, it is a clever piece of programming, but in our opinion it is bad style. Why? Because you are using a flag as a bitmask, which is a completely different datatype. Although there is no such thing as "data typing" in Forth, this way of programming makes it difficult to understand and maintain a program, which the ANS-Forth standard acknowledges:

> "The discipline of circumscribing meaning which a program may assign to various combinations of bit patterns is sometimes called data typing. Many computer languages impose explicit data typing and have compilers that prevent ill-defined operations. Forth rarely explicitly imposes data-type restrictions. Still, data types implicitly do exist, and discipline is required, particularly if portability of programs is a goal. In Forth, it is incumbent upon the programmer (rather than the compiler) to determine that data are accurately typed."

That is why 4tH uses "1" as a true flag. Usually, it won't make much difference. Except when you use 'INVERT' to invert a flag or intend to make obfuscated programs. If you use '0=' instead, you won't run in any trouble, not even when you port your program to ANS-Forth. Clarity may introduce a little overhead, but in this age of multi-gigaherz machines, who is counting? E.g. you could program ">CHAR" like this:

```
\ convert a flag to a bit mask
: >MASK 0 SWAP IF INVERT THEN ;           ( f -- mask)
                                          \ convert a digit to ASCII
: >CHAR DUP 9 > >MASK 7 AND + ASCII 0 + ; ( n -- c)
```

If you still want to change the true flag, you can by simply changing a #define in "cmds_4th.h":

```
#define F_T ~(0L)
```

But we doubt whether it will be a great benefit to your programming style.

16.3.4 CREATE..DOES>

In both 4tH and ANS-Forth it is possible to change the runtime behaviour of variables. E.g. in ANS-Forth, 'CONSTANT' is usually defined as:

```
: CONSTANT CREATE , DOES> @ ;
10 CONSTANT MY_CONST
MY_CONST . CR
```

Of course there is a predefined word in 4tH that does this, but if you wanted to mimic this behaviour you would have to define it like this:

```
CREATE MY_CONST 10 ,                      \ CREATE part
:THIS MY_CONST DOES> @C ;                 \ DOES> part
MY_CONST . CR                             \ Works the same way
```

The point is that the ANS-Forth "CREATE DOES>" construct cannot be ported to 4tH, although all words seem to be supported. A rule of the thumb is that *defining words cannot be used to define new defining words*, like in ANS-Forth. Most errors will be trapped by 4tH's compiler, though.

Just remember that a ':THIS' definition can easily be ported to ANS-Forth. If you want to write a portable program, ':THIS' is the way to go.

16.3.5 HERE

Be careful with 'HERE'. 'HERE' looks and acts a lot like the ANS-Forth 'HERE', but since the architecture is different it serves quite another function. When 'HERE' is used for address arithmetic with definitions or arrays of constants, it works right out of the box. If not, it usually doesn't.

16.3.6 Interpretation and compilation mode

There are several words, which act differently in interpretation and compilation mode. In Forth-79, some of them were "state-smart", which means they adjusted their behaviour depending on the mode the system was in. In Forth-83 and subsequently ANS-Forth, they became "dumb" words and counterparts were designed for each mode. Other words lacked interpretation semantics all together.

4tH has got neither a true interpretation mode nor a state. But if you want to port 4tH code to ANS-Forth, this has to be dealt with. In 4tH this porting issue is resolved by several aliases. Some words have an alias since they do not have interpretation semantics in the ANS-Forth standard, but are often used outside colon-definitons in 4tH. This will enable you to make a word that mimics these interpretation semantics.

This table lists all "dumb" words with their counterparts. "Interpretation" means it has to be used outside colon-definitions. "Compilation" means it has to be used inside colon-definitions.

INTERPRETATION	COMPILATION
'	[']
.(	."
CHAR	[CHAR]

Table 16.1: Dumb words

Finally, in ANS-Forth all flowcontrol words (like IF, THEN, BEGIN, WHILE, DO, LOOP) may only be used inside colon-definitions.

16.3.7 BEGIN..WHILE..REPEAT

4tH allows you to use multiple WHILE's in a BEGIN..WHILE..REPEAT construct. ANS-Forth allows that too, but requires an extra 'THEN' for each additional 'WHILE'. In short, this is the 4tH version:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat
```

And this is the ANS-Forth version:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat then
```

To make this work, we have to resort to conditional compilation:

```
0 begin dup 10 < while 1+ dup 5 mod while 1+ repeat
[undefined] 4th# [if] then [then]
```

It's not beautiful, but it works. The same applies when 'UNTIL' is used instead of 'REPEAT'. BEGIN..WHILE..AGAIN constructs are not supported by ANS-Forth, so be careful when considering 'AGAIN' too much of an alias of 'REPEAT'.

16.3.8 DO..LOOP

It is well-known in the Forth community that DO..LOOP is flawed. There have been several attempts to correct this, but they never got it right. On many occasions it even got worse. But why is DO..LOOP flawed?

'DO' puts the limit and the index on the Return Stack, but it doesn't decide whether the loop is actually entered. So, every loop is executed at least once. After each iteration 'LOOP' decides whether it iterates once more.

In our opinion it would have been better when 'DO' had made that decision (like any other language), but we can still live with that. The real trouble came with DO..+LOOP.

'+LOOP' is a logical extension. Every single language allows you to change the step. But contrary to what one might expect, '+LOOP' doesn't terminate when the loop limit is reached or exceeded, but when the loop index crosses "the boundary between the loop limit minus one and the loop limit".

What does that mean? Well, consider these three loops and try to predict what will be printed. Note: every loop is executed at least once:

```
( 1) 5 0 do i . 1 +loop cr
( 2) 5 0 do i . -1 +loop cr
( 3) -5 0 do i . -1 +loop cr
```

You would probably expect to see:

```
0 1 2 3 4
0
0 -1 -2 -3 -4
```

And that is what you get when you use 4tH. But this is *not* what you will get with ANS-Forth:

```
0 1 2 3 4
0 -1 -2 -3 .. 6 5
0 -1 -2 -3 -4 -5
```

The behaviour of the second loop is caused, because '+LOOP' doesn't take into account that it is counting down. So it iterates until the loop index reaches the loop limit by wrap-around arithmetic.

The behaviour of the third loop is caused by the ANS-Forth definition: the loop index must "cross the boundary between the loop limit minus one and the loop limit". In this case, the boundary is between -5 and -6.

DO..+LOOP didn't behave like this since the beginning of Forth: it was introduced in Forth-83. We preserved the Forth-79 definition as closely as possible, because it is much more intuitive.

Some claim that ?DO..+LOOP will save it. As a matter of fact, it does. But only when the loop index and the loop limit are the same:

```
0 0 ?do i . loop
```

In that case the loop won't be entered. But it still won't save us for loops like this:

```
5 0 do i . -1 +loop cr
```

The authors of gForth claim that a whole host of new DO..LOOP words are the solution. We don't think so:

```
100 -100 do i . i 1+ 2/ negate +loop cr
```

The bottomline is: you can't let two words make the same decision. 4tHs '+LOOP' checks which direction it is going (up or down) and evaluates the loop arguments accordingly. We feel it is the best we can do for you.

Is there no way we can circumvent these problems? Yes, there is. It may not be too elegant or even fast, but it solves the problem. We just emulate C's for():

```
12 >r begin                          \ set up loop index
  r@ 10 <                            \ check loop limit
while
  r@ .                               \ access loop index
  r> 2+ >r                           \ increment loop index
repeat                               \ next iteration
r> drop cr                           \ drop loop index
```

Which is "equivalent" to:

```
10 12 ?do i . 2 +loop cr
```

Except that it works as expected. And as an extra bonus it is portable to ANS-Forth. Are these differences between the ANS-Forth and 4tH implementation of DO..LOOP really that important? Not in practice. Nobody really wants a loop that depends on wrap-around arithmetic, and you'll hardly ever see a '+LOOP' with a negative subscript. Everybody wants their programs to be understandable and maintainable, so the DO..LOOPs you'll encounter will usually be well-behaved.

16.3.9 I/O

It is trivial to define the ANS-Forth **FILE** wordset in 4tH, but almost impossible to do the opposite. So if you want to make a portable program use the ANS-Forth **FILE** wordset by including the *'ansfile.4th'* library file. The reason why 4tH uses a different I/O subsystem is twofold:

1. 4tH's I/O subsystem is far more powerful and elegant. Instead of defining a whole new wordset, 4tH reuses most of the available I/O words, like 'TYPE', 'EMIT', 'ACCEPT' and 'REFILL'[2], which is very Forth-like.
2. 4tH's I/O was initially quite primitive and this was the only way to extend the system without breaking too much code.

This example is taken from gForth, but runs identically on both 4tH and gForth:

[2] As a matter of fact, a new set of words have been proposed that allow redirection of these words. 4tH already has that functionality.

```
[defined] 4th# [if]                     \ if this is 4tH, include
include lib/ansfile.4th                 \ ANS Forth FILE wordset
include lib/compare.4th                 \ and the word COMPARE
[then]

[undefined] 4th# [if]                   \ if this is not 4tH,
s" lib/easy.4th" included               \ include 4tH compatibility
[then]

0 Value fd-in                           \ input file handle
0 Value fd-out                          \ output file handle
: open-input ( addr u -- )  r/o open-file throw to fd-in ;
: open-output ( addr u -- )  w/o create-file throw to fd-out ;

s" foo.in" open-input                   \ open input file
s" foo.out" open-output                 \ open output file

: show 2dup type cr ;                   \ show the line

256 Constant max-line                   \ size of basic buffer
max-line 2 [+] string  line-buffer      \ extend by two bytes:
                                        \ ANS Forth requirement!
: scan-file ( addr u -- )
  begin                                 \ read a line
      line-buffer max-line fd-in read-line throw
  while                                 \ is it identical?
      >r 2dup line-buffer r> show compare dup
  while                                 \ if so, exit loop
      drop                              \ clean up
  repeat                                \ ANS requires an extra
[undefined] 4th# [if] then [then]       \ then after each WHILE
  drop 2drop
;
                                        \ now scan the file
s" The text I search is here" scan-file

fd-in close-file throw                  \ close input file
fd-out close-file throw                 \ close output file
```

Since section 11.3.12 of the ANS-Forth standard clearly states that an I/O exception shall not cause a 'THROW', we have to mention that 'FILE-SIZE', 'FILE-STATUS' and 'REPOSITION-FILE' are not entirely ANS-Forth compliant.

16.4 Easy 4tH

4tH programs won't run on ANS-Forth all by itself. You'll usually need several definitions to make them work. In collaboration with Wil Baden we have developed an interface between ANS-Forth and 4tH. It consists of two files, "easy.4th" and "ezneeds.4th". These library files enable you to run most 4tH programs under ANS-Forth. In order to succesfully compile and run a 4tH program under ANS-Forth it must have been written with ANS-Forth in mind. The rest is simple: just add a couple of lines at the beginning of your 4tH program:

```
[UNDEFINED] 4TH# [IF]
s" easy4th.4th" included
[THEN]
```

That's all! Most of the 4tH words are now known to your very own ANS-Forth compiler. If your compiler already supports 'INCLUDE', you might be tempted to use:

```
[UNDEFINED] 4TH# [IF]
include easy4th.4th
[THEN]
```

This will actually work but since 4tH recognizes and acts on the 'INCLUDE' directive, it will load the interface. A slight memory and CPU penalty is the result.

16.4.1 Disabling DOES>

Easy 4tH will effectively disable 'DOES>', since 'DOES>' is nothing more than some syntactic sugar in 4tH. If you want to use the standard Forth 'DOES>', you have to define '(_KEEP_DOES_)' somewhere, e.g.

```
[UNDEFINED] 4TH# [IF]
0 constant (_keep_does_)
s" easy4th.4th" included
[THEN]
```

Easy 4tH will now leave 'DOES>' alone. Note that you have to refrain from using 'DOES>' in your ':THIS' definitions (see section 16.3.4).

16.4.2 Enabling the String Space

Optionally, you can define a 'CONSTANT' before including Easy 4tH, which enables support for *arrays of string constants*:

```
<size> constant /STRING-SPACE
```

The parameter "SIZE" represents the size of the String Segment. When you decompile a 4tH program, it will show you exactly how much space is allocated to the String Segment. In order to port a single 4tH program, this is all the information you need!

```
4tH Message : No errors at word 1105
Object size : 1106 words
String size : 2539 chars
Variables   : 19 cells
Strings     : 262 chars
Reliable    : Yes
```

In this case the "/STRING-SPACE" must be at least 2539 bytes. So, give or take a few changes, let's say 3072 bytes. We advise you to allocate a little more memory than is strictly necessary. You can also use Easy 4tH to make ANS-Forth understand 4tH. Just type:

```
16384 constant /STRING-SPACE
s" easy4th.4th" included
```

Now you can play around with ANS-Forth using the 4tH language. If you use a lot of string constants you might run out of space, but your ANS-Forth compiler will give you a message when that happens.

Note that - depending on the ANS-Forth compiler you're using - Easy 4tH *may* redefine some words, although it will try to minimize these redefinitions as much as possible.

16.4.3 The structure of Easy 4tH

Easy 4tH may look like a large program, but it isn't. It basically tries to figure out what your compiler supports and what is still left to define. It always prefers the native definition to its own. E.g. if your compiler already supports 'PLACE', Easy 4tH will leave that definition intact and *assume* it has been defined correctly.

- Easy 4tH will start by defining several defining words like 'STRING', 'STRUCT' and 'ARRAY'. Since there is no standard definition for these words, it will overwrite any existing definition.
- After that, Easy 4tH will query the environment and define a 'CONSTANT' when successful. When not, a warning is issued.
- Several 4tH specific compiling words are defined.
- Easy 4tH checks for the presence of several ANS-Forth and COMUS words. If they are not there, they are defined. Warnings are issued where applicable.
- In the next stage the PARSING, CONVERSION, TIME and RANDOM NUMBER subsystems of 4tH are defined. Warnings are issued where applicable.
- All 4tH words that *cannot* be defined in ANS-Forth are marked as *unsupported*. When used, an error message is issued and compilation aborted.
- The standard ANS-Forth 'DOES>' is disabled (unless you override it).
- "ezneeds.4th" is loaded and '[NEEDS' and 'INCLUDE' are defined if needed.

Note that your own compiler may issue error messages or warnings too, e.g. about redefinitions.

16.5 Converting ANS-Forth programs to 4tH

4tH is a subset of ANS-Forth, so it might be difficult to find a program that will run on 4tH without at least *some* rewriting. And there is no guarantee that it will work, because most ANS-Forth programs weren't written with 4tH in mind. We'll list the major pitfalls:

- Programs requiring unsupported words or most words from the **FACILITY**, **FACILITY EXT**, **FLOATING**, **FLOATING EXT**, **SEARCH** and **SEARCH EXT** wordsets are generally impossible to port.
- Definitions manipulating the dictionary or the stacks. But 4tH has no dictionary and does not allow direct access to the stacks.

- Definitions that switch between interpretation and compilation mode. 4tH either interprets or compiles; you cannot switch between the two on the fly. User-defined 'IMMEDIATE' words generally don't work.

- Definitions using 'CREATE' and 'DOES>' can be difficult to port. The only way is to do the 'CREATE' part manually and wrap the 'DOES>' part into a ':THIS' definition.

- Definitions requiring the **LOCAL** and **LOCAL EXT** wordsets are difficult to port. You'll need to rewrite them extensively by using the *'locals.4th'* library file.

- Definitions using ANS-Forth enhanced flow control require some rewriting and conditional compilation.

- Programs that assume they may store cells and characters in the same dataspace require some rewriting. Use the *'ncoding.4th'* library file.

Part IV

Development guide

Chapter 17

Compiling the source

17.1 Introduction

4tH is primarily designed as a powerful and easy to use toolkit for developers. You can use it "as is" and you've got a very flexible calculation-engine. You can tailor it to a specific application and push it even further. Or you just might want to use one of the safest and easiest Forth-alike environments ever created. These are all valid reasons and I will try to address them all.

First, I will show you how to compile the example applications. They are far from useless. In fact, you will have created a complete programming environment. Second, I will show you how to create the 4tH library and how to use it. Third, I will explain to you how the compiler works and how you can make simple additions to 4tH.

If you find any errors in this document please contact me by sending email to "hansoft@bigfoot.com". You would be helping a lot of future 4tH users.

17.2 Recommended and preferred compilers

4tH is written in ANSI-C and K&R C and should be portable to any platform that supports such a compiler. All memory-models are supported, although the usual restrictions apply. Of course, it is impossible to test every single compiler on the market, but there are a number of compilers that are known to work. Preferred compilers are Open Source and are available for a number of platforms. When properly installed, the entire compilation can be performed in two simple steps:

```
make
```

Then login as root and enter:

```
make install
```

That's all. Any documentation, library files or example programs must be installed manually. Recommended compilers are free (as in beer) and are known to work. It's up to you to figure out the correct installation procedure. Unlisted compilers may or may not work.

There is no such list for OS/X or Linux. Those platforms usually already come with an Open Source GCC compiler.

COMPILER	URL	PLATFORM	LABEL
GCC 2.95 msvcrt	http://downloads.activestate.com/pub/staff/gsar/gcc-2.95.2-msvcrt.zip	Win32	Preferred
Cygwin	http://www.cygwin.com/	Win32	Preferred
MinGW	http://sourceforge.net/projects/mingw/	Win32	Preferred
Pelles C	http://www.smorgasbordet.com/pellesc/	Win32	Recommended
LCC	http://www.cs.virginia.edu/~lcc-win32/	Win32	Recommended
TCC	http://fabrice.bellard.free.fr/tcc/	Win32	Recommended
Turbo C V2.01	http://bdn.borland.com/article/images/20841/tc201.zip	DOS	Recommended
DJGPP	http://www.delorie.com/djgpp/	DOS	Preferred

Table 17.1: List of compilers

17.3 Compiling 4th

First copy all files to any directory you like. Now make the latter directory your current directory. The following commands are applicable to Linux, OS/X and other unices.

If you aren't using an ANSI-C compiler add the "-DARCHAIC" switch. If you are working on a Unix platform add the "-DUNIX" switch. If you add the include-files to your own /usr/include directory, use the "-DUSRLIB4TH" switch. Note that this isn't a recommended practice.

Finally, if your compiler features a `stricmp()` function you want to use instead of 4tHs builtin `MatchName()`, use the "-DSTRICMP" switch. *These optional switches will be referred to in the following examples as "$(CFLAGS)".*

Unfortunately, not all C-compilers are created equal, so you have to check the documentation that came with your compiler. You have to check for three things:

1. First, the 4tH-toolkit assumes that all chars are signed. I know there are a few compilers out there, that assume that chars are unsigned (like the RS/6000 and GNU compilers). In most cases switches are available to correct that. Some K&R compilers do not support the type "void". If so, you might have to add "#define void" to 4th.h.
2. Second, compilers on non-Unix platforms might use different or additional switches, like those that determine the memory-model.
3. Third, this documentation assumes you call your compiler with "cc". Borland compilers are called with "bcc" or "tcc". Watcom compilers are called with "wcc". GNU compilers are called with "gcc". Microsoft compilers are called with "cl" for some obscure reason (C Language?).

The 4th program is a do-all compiler. It compiles the source, executes it and if you made a programming error it will decompile whatever it could compile. You can also load existing objects or save new objects. It takes at least two arguments, which are the command string and the 4tH-program. In Linux, OS/X or other unices, entering:

```
make
su
make install
```

should do the trick. Most MS-DOS compilers contain a version of 'make', but you probably have to recreate the makefile. Otherwise, compile it with:

```
cc $(CFLAGS) -O -o 4th 4th.c open_4th.c comp_4th.c exec_4th.c
dump_4th.c load_4th.c save_4th.c errs_4th.c name_4th.c free_4th.c
cgen_4th.c
```

If your compiler does not support a command line interface, you have to include these files:

```
free_4th.c
errs_4th.c
name_4th.c
dump_4th.c
exec_4th.c
load_4th.c
save_4th.c
comp_4th.c
open_4th.c
cgen_4th.c
4th.c
```

17.4 Compiling the library

If you want to add the 4tH compiler to your own programs, I strongly advise you to create a library. As a matter of fact, I will assume that you have created the library later on. The library uses only a handful of functions, so it is feasible to create the library manually. However, we advise you to use the makefile. Carefully check all macros. In Linux, OS/X or other unices, entering:

```
make
su
make install
```

should do the trick. Most MS-DOS compilers contain a version of 'make', but you probably have to recreate the makefile. The library consists of 10 functions:

```
comp_4th()
exec_4th()
dump_4th()
free_4th()
save_4th()
load_4th()
errs_4th[]
name_4th[]
open_4th()
cgen_4th()
```

You can compile each function manually by issuing the command:

```
cc $(CFLAGS) -O -c <function>.c
```

You'll end up with 11 objectfiles. You can add these functions to a library by issuing:

```
ar r lib4th.a <function>.o
```

The resulting library must be moved to the /usr/lib directory. When using MS-DOS we strongly advise you to create a library for each memory-model, like "4ths.lib" for a small memory-model library, "4thl.lib" for a large memory-model library, etc. When you compile a function for a particular memory-model you'll have to add the memorymodel switch. Then create a library by issuing this command for each function:

```
lib 4th<model>.lib + <function>.obj
```

Whether you use a makefile or create the library manually, you should end up with a working 4tH library. If you are unable to make a working makefile, write a script- or batchfile. If you have to recreate the library, it will save you a lot of work.

17.5 Using the library

Before we dive into the depths of the API, we will tell you how to compile a program that uses the 4tH library. If you are using an advanced MS-DOS or MS-Windows compiler this may or may not apply to you. In that case we advise you to check your documentation. If you are using a plain vanilla compiler this will usually work. Compile the program with:

```
cc -c <program>.c
```

When you use a non-Unix system, link the objectfile with the startup code and libraries to create an executable program. The <model>.obj is the startup code for C-programs and <model>.lib is the runtime-library.

```
link <model>.obj <program.obj>, <program>„ 4th<model>.lib <model>.lib
```

Unix developers just give the command:

```
cc <program>.c -o <program> -l4th
```

If you happen to use the GNU Compiler Collection (gcc), just issue:

```
gcc -s -Wall -fsigned-char <program>.c -o <program> -l4th
```

After you've built the library you can also issue these commands to (re)compile 4th.

17.6 Optimizations

It really depends on your compiler. Some compilers allow optimizations up to level 3 (`-O3`), others won't even produce a usable compilant with any optimization enabled. It can even depend on the platform you're compiling for. It is hard to give a general recommendation, but try compiling 4tH without any optimization first, then crank optimization gradually up until the compilant doesn't work properly anymore or optimization doesn't give you any more speed or size advantages. You may have to do some benchmarking to find this out[1]. There is also a program that allows you to test the virtual machine. All listed preferred compilers allow the highest optimization level. The optimization level can be set manually or can be adjusted in the Makefile. Consult your compiler documentation for details.

[1] 4tH comes with a wide selection of benchmarking programs.

Chapter 18

Using the 4tH API

18.1 Introduction

One of the design requirements of the 4tH library was that it had to be very easy to use. We've seen many APIs that were impossible to use and put most of the burden on the developer.

4tH takes a different direction. We've designed an API that almost exactly matches the tasks you want to perform. Want to compile? Compile. Want to decompile? Decompile. Want to save? Save. Just like that. No difficult to understand datatypes, no initialization, no garbage collection, no checks.

The only error, you, the developer can make is fill up memory. Virtually all other errors are caught by the API. E.g. 4tH will refuse to save or execute a 4tH-program when compilation failed. Of course, if you manipulate 4tHs datastructures directly you can still bring it to its knees, but I assume that is not what you want.

18.2 A sample program

There are ten API-functions.

API	FUNCTION
comp_4th()	Loads and compiles a 4tH source to an H-code object
exec_4th()	Executes an H-code object
dump_4th()	Decompiles an H-code object
free_4th()	Frees an H-code object from memory
save_4th()	Saves an H-code object to an HX-file on disk
store_4th()	Saves an H-code object to an HX-file in memory
load_4th()	Loads an HX-file from disk and installs H-code
fetch_4th()	Loads an HX-file from memory and installs H-code
open_4th()	Creates a small 4tH program that loads a 4tH source
cgen_4th()	Generates a standalone C source with embedded H-code

Table 18.1: API functions

That's really all! So if you want to compile a 4tH source, save it to disk, execute it and finally discard it, you've virtually written the program.

H-code is nothing but a pointer to a structure. Even if you thought you never worked with structures before, it's as easy as working with files. We'll show you.

When you want to use files you first have to include "stdio.h", like:

```
#include <stdio.h>
```

If you want to use a single file for output, you've got to declare a file-pointer like:

```
FILE* Outfile;
```

Before you can use a file, you have to open it:

```
Outfile = fopen ("filename.ext", "w");
```

If the file cannot be opened, `fopen()` returns a NULL-pointer. You have to check that before you can safely use the file:

```
if (Outfile == NULL) printf ("Unable to open file");
else fprintf (Outfile, "This is written to disk");
```

Finally, when you're done, you have to close the file:

```
fclose (Outfile);
```

Working with the 4tH library is very similar. When you want to use the 4tH library you write:

```
#include "4th.h"
```

When you use an H-code object (which is a compiled 4tH program), you declare a H-code pointer:

```
Hcode* Program;
```

Before you can use the H-code pointer, we first got to compile a 4tH source. 4tH source is simply a `malloc()`ed ASCIIZ string. That means that anything you can convert to a string stored in dynamic memory can be used as 4tH source. That includes constant strings, environment variables, lines read from a file, etc.

In this example we use the `strdup()` function to convert a constant string. We know that not all runtime-libraries contain such a function, so if you want give it a try, here is the source:

```
char *strdup (char* str)
{
      char *p;
      p = calloc (strlen (str) + 1, sizeof (char));
      if (p) strcpy (p, str);
      return (p);
}
```

Now we can all create a 4tH source and compile it. This one will create the famous "Hello world" program:

```
Program = comp_4th (strdup (".\" Hello world\" cr"));
```

Without any checking you can try to execute it:

```
exec_4th (Program, 0, NULL, 0);
```

The "0" means we do not want to transfer any variables or constants to the execution environment, but we'll get to that later on. You could make a second call to `exec_4th()` and see the program execute twice. The H-code is still in memory. In order to free it from memory, we have to end our program with:

```
free_4th (Program);
```

We are finished now. The full program looks like this:

```
#include "4th.h"
#include <stdlib.h>

int main(int argc, char** argv)
{
      Hcode* Program;
      Program = comp_4th (strdup (".\" Hello world\" cr"));
      exec_4th (Program, 0, NULL, 0);
      free_4th (Program);
      return (EXIT_SUCCESS);
}
```

Now compile the C-program and execute it. You should see that it prints "Hello world" on the screen. Now, that wasn't too hard, was it?

18.3 A first look at open_4th()

You probably don't want to compile constant strings. Most certainly you want to create a source-file and compile it. If you come to think of it, that could be the hardest part when you want to make your own 4tH-compiler.

Don't worry. We've created a function that handles just that: `open_4th()`. The function creates a tiny 4tH program that tells `comp_4th()` which file to load. `open_4th()` just wants to know which file to load.

If an error occurs (which is very rare), `open_4th()` returns a NULL pointer. If `open_4th()` is successful it returns a char-pointer to the program in memory. You can feed the return-value of `open_4th()` directly to `comp_4th()`. So, we've got to declare a char-pointer for the return-value of `open_4th()` and call the function:

```
char* source;

source = open_4th ("venture.4th");
```

No need to open or close files, `comp_4th()` will take care of that. You're one step closer to the creation of your own compiler.

18.4 A closer look at H-code

H-code is not just a simple pointer to some simple structure. In fact, it is more complex than a file-pointer. It is comprised of several parts.

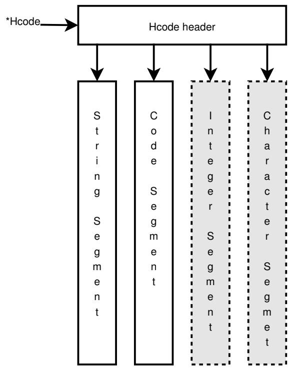

Figure 18.1: Hcode structure

1. First, the header. The header contains all information about the actual program, e.g. the number of variables, the size of allocated space, its status. The mere existence of an H-code pointer doesn't mean you actually got a program you can execute.
2. Second, the Code Segment. This contains the actual program. If you don't have a Code Segment, there is nothing to execute since you have no program. The Code Segment is an array of words. A word consists of a token and an argument. Every token matches a piece of compiled C in the interpreter. We'll get to that later on.
3. Third, the String Segment. This segment only contains constant strings, defined by e.g. S" , ," , .(and .".
4. Fourth, the Integer Segment. This segment contains the stacks and all writable integer data. It is only present when a program is sleeping (or hibernating, if you prefer).
5. Fifth, the Character Segment. This segment contains all writable character data. It is only present when a program is sleeping.

The first three parts are read-only to the 4tH-programmer. If you are smart, you consider them to be read-only too. There is no need whatsoever to change anything here. The API knows best.

18.5 A first look at comp_4th()

The function `comp_4th()` is one of the most complex and most important functions of the API. It takes a source and compiles that to H-code. If you want to save the source, copy it or reload it, since `comp_4th()` consumes it entirely. That is why source has to be allocated in dynamic memory. On the other hand, it explains why compilation needs very little memory.

If `comp_4th()` can't compile anything, it returns just the header, containing all the information on what went wrong and where. If there is not enough memory to allocate even the header it returns a NULL-pointer. If it could compile part of the code it returns what it could compile. But that might not be everything and you need it all to get a program you can execute.

Lucky for you, but the API can determine whether `comp_4th()` returned an executable H-code or not. If `exec_4th()` doesn't execute the program, it just couldn't.

The `comp_4th()` function is just as smart. It can survive the NULL pointer you feed it. When you read on, you'll find out that the `comp_4th()` function is a toolkit by itself with lots of tools to create your own 4tH words.

If we extend the example we started with `open_4th()`, we could continue like this. In order to make an executable H-code we feed source to `comp_4th()` and get H-code in return. We just need a pointer to store its address:

```
Hcode *Program;
```

So the entire 4tH-compiler now looks like:

```
char *source;
Hcode *Program;

source = open_4th ("venture.4th");
Program = comp_4th (source);
```

We assume you want to execute the program you've just compiled, so we'll continue with the interpreter-function called `exec_4th()`.

18.6 A first look at exec_4th()

The `exec_4th()` function is essentially very simple. It contains small pieces of C that can be matched with the tokens in the Code Segment. The `exec_4th()` function executes these small pieces of C until there are no more words left to execute or an error occurs.

Additionally, it creates two new segments, which are discarded when `exec_4th()` terminates without hibernation. Each segment is essentially an array of a specific datatype. The sizes of these segments are specified in the header of the H-code.

The Character Segment contains characters. First, the Terminal Input Buffer, abbreviated to TIB. When you execute REFILL, this is the place where the string you typed is stored. Second, the PAD. This is the place where `exec_4th()` stores temporary strings. Finally the Allocation Area where strings, defined by STRING are allocated.

The Integer Segment contains signed 32 bit integers. First, the Stack Area. It contains both the return stack and the data stack. The return stack grows downward and the data stack grows upward. Second, the System Variable Area. The System Variable Area contains three variables, HANDLER, HERE and HLD. They can only be accessed by 4tH itself. Third, the Variable Area. The Variable Area contains four basic types of variables.

First, the environment variables. In standard 4tH there are five environment variables, HI, FIRST, LAST, CIN and COUT. They are all read-only. Second, the predefined variables. In standard 4tH there are five predefined variables, BASE, >IN, OUT and the variable pair SOURCE. Later on, we'll teach you how to add your own. Third, the application variables. These are copies of C-variables or constants. Fourth, the user-defined variables. These are defined by the application-programmer with VARIABLE, VALUE, DEFER, FILE or ARRAY.

The `exec_4th()` function takes an H-code pointer and returns the current value of the variable OUT when it exits. When an error occurs, `exec_4th()` returns always the largest negative 32 bit integer, which is also the default value of OUT. In order to give you maximum control, you can also transfer string-arrays and C-variables or constants to `exec_4th()`.

C-variables have to be of type 'cell'. This type is predefined in the '4th.h' headerfile. So you can just write:

```
#include "4th.h"
cell february = 29;
```

You can mix constants or variables as you like. Just add the appropriate cast. You also have to declare the number of variables or constants you transfer (in this case 12). Let's say you transfer the number of days of each month to `exec_4th()`:

```
#include "4th.h"
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell february = 29;
      cell Result;
      Hcode *Program;
      char *source;

      source = open_4th ("months.4th");
      Program = comp_4th (source);
      Result = exec_4th (Program, 0, NULL, 12, (cell) 31,
      february, (cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell)
      31, (cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell) 31);
      return (EXIT_SUCCESS);
}
```

The application-programmer can use the 'APP' variable to access all months:

```
 12 0 do app i th ." days: " ? cr loop
```

Later, we'll learn you how to assign names to those application-dependant variables. Note that although 'Result' contains the return-value of the 'months.4th' program, the C-program does absolutely nothing with it. In the next example we'll show you how you can use this value.

Take this C-program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      Hcode *Program;
      char *source;
```

```
        source = open_4th ("calc.4th");
        Program = comp_4th (source);
        Result = exec_4th (Program, 0, NULL, 2, (cell) 5, (cell) 7);
        printf ("Result: %ld\n", (long) Result);
        return (EXIT_SUCCESS);
}
```

This program compiles 'calc.4th' and transfers '5' and '7' to `exec_4th()`. What is returned in 'Result' depends on what 'calc.4th' does. Let's take a look at 'calc.4th':

```
app 0 th @ app 1 th @ + out !
```

It fetches both variables, adds them and assigns the sum to OUT. Thus, when `exec_4th()` terminates it returns the value of OUT. This value is assigned to 'Result'. Then, 'Result' is cast to 'long' and displayed:

```
        Result: 12
```

We haven't used the other two arguments of `exec_4th()` yet. These are used to pass string constants to your 4tH application. Most of you will use it to pass commandline arguments to 4tH.

Let's say we want to pass these arguments to 4tH in C-style. So "0 ARGS" is the name of our 4tH-program. That means we have to skip the name of the C-program itself. The name of our 4tH-program in in argv [1]. That also means we have to decrement argc, before passing it to `exec_4th()`:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
        Hcode *Program;
        char *source;

        if (argc > 0) {
                source = open_4th (argv [1]);
                Program = comp_4th (source);
                (void) exec_4th (Program, argc - 1, argv + 1, 0);
                return (EXIT_SUCCESS);
        }
        return (EXIT_FAILURE);
}
```

Note that argv [1] is not the same as argv + 1! When we call our C-program with:

```
main args.4th hello here I am
```

argc will be 6. That means `exec_4th()` will get a stringarray with five strings (since we discarded one). We can access these strings by:

```
argn 0> if argn 0 do i args count type cr loop then
```

Which would print:

```
args.4th
hello
here
I
am
```

You do not *have* to pass argc and argv; other string arrays of the same type ('**char' or '*char[]') are okay too. Just don't forget to pass the correct number of elements in the string array.

So now you know how your H-code program can communicate with your C-program. You can transfer any number of variables to the 4tH-environment and retrieve the result. You can even pass commandline arguments to the 4tH-environment and use them inside your application. In the next section you will be introduced to some other interesting properties of the 4tH-environment.

18.7 A first look at free_4th()

Let's take a look at this program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      Hcode *Program;
      char *source;

      source = open_4th ("calc.4th");
      Program = comp_4th (source);
      Result = exec_4th (Program, 0, NULL, 2, (cell) 5, (cell) 7);
      printf ("Result: %ld\n", (long) Result);
      Result = exec_4th (Program, 0, NULL, 2, (cell) 12, (cell) 9);
      printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCESS);
}
```

Essentially, it is the very same program we've seen in the previous section. But here the 'calc.4th' program is executed twice with different arguments. Can that be done without recompiling the program? Yes! The pointer 'Program' is still valid and points to the very same 4tH program in memory. You can execute it any number of times without recompiling. You can go even further:

```
#include "4th.h"
#include <stdio.h>
```

```
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      char *source;
      Hcode *Multiply;
      Hcode *Subtract;

      source = open_4th ("multiply.4th");
      Multiply = comp_4th (source);
      source = open_4th ("subtract.4th");
      Subtract = comp_4th (source);

      Result = exec_4th (Multiply, 0, NULL, 2, (cell) 7, (cell) 5);
      printf ("Result: %ld\n", (long) Result);
      Result = exec_4th (Subtract, 0, NULL, 2, (cell) 7, (cell) 5);
      printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCESS);
}
```

The file 'multiply.4th' contains:

```
app 0 th @ app 1 th @ * out !
```

The file 'subtract.4th' contains:

```
app 0 th @ app 1 th @ - out !
```

So executing this C-program will give the following result:

```
Result: 35
Result: 2
```

And yes, both programs can be re-executed any number of times. But what if some 4tH program has served his purpose? Does it have to remain in memory all the time? No. Since it is located in dynamic memory it can be freed. Not with `free()`, since H-code is too complex to be served with a simple function like `free()`. But a special function is included in the API, which serves the same purpose:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      cell Result;
      char *source;
      Hcode *Multiply;
      Hcode *Subtract;

      source = open_4th ("multiply.4th");
```

```
        Multiply = comp_4th (source);
        source = open_4th ("subtract.4th");
        Subtract = comp_4th (source);

        Result = exec_4th (Multiply, 0, NULL, 2, (cell) 7, (cell) 5);
        printf ("Result: %ld\n", (long) Result);

        free_4th (Multiply);

        Result = exec_4th (Subtract, 0, NULL, 2, (cell) 7, (cell) 5);
        printf ("Result: %ld\n", (long) Result);
        free_4th (Subtract);
        return (EXIT_SUCCESS);
}
```

The function `free_4th()` takes an Hcode-pointer and frees all resources. There is really nothing more to it. Remember that `free_4th()` doesn't alter the pointer itself. It may still contain a value, but of course using that value is asking for trouble. The API checks quite a few things by itself, but that doesn't mean you can start to write sloppy programs!

18.8 A first look at save_4th()

We've already seen that we can compile a 4tH-program and keep it in memory for as long as we want. We can also discard it if we don't need it anymore. But what if we want to reuse the compilant later? Or if we want to distribute 4tH programs without revealing our source-code?

You can do that easily. 4tH uses another main format which not only enables you to load compiled programs, but also run them on a multitude of platforms. It is called the 'Hcode eXecutable' (HX-file) and it is fully portable across all platforms 4tH supports.

Saving a program is very easy too. You don't even have to open or close files. Here is a very simple compiler:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
        char *source;
        Hcode *Program;

        if (argc == 3) {
                source = open_4th (argv [1]);
                Program = comp_4th (source);
                save_4th (Program, argv [2]);

                free_4th (Program);
                return (EXIT_SUCCESS);
        }
        return (EXIT_FAILURE);
}
```

You just declare the input and the output file on the commandline and when no errors occur an HX-file is saved to disk. The `save_4th()` function takes the Hcode pointer and the filename you want to save it to. Note `save_4th()` supports hibernation too; just feed it a sleeping virtual machine. That's all!

18.9 A first look at load_4th()

But you don't just save compiled programs. You want to be able to reuse them too. There is a special function that reads an HX-file and restores the H-code to its original form. This API-function is easy to use too. Just feed it the name of the file and it returns a pointer to the H-code. This is the listing of a simple HX-execute:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      Hcode *Program;
      cell Result;

      if (argc == 2) {
            Program = load_4th (argv [1]);
            Result = exec_4th (Program, 0, NULL, 0);
            printf ("Result: %ld\n", (long) Result);
            free_4th (Program);
            return (EXIT_SUCCESS);
      }
      return (EXIT_FAILURE);
}
```

You just declare the HX-file on the commandline and when no errors occur it is executed. Finally, it displays the result of the program.

18.10 A first look at error-trapping

If you are a professional programmer you might appreciate the ease of use of the 4tH toolkit, but you have the feeling you don't have any control. If that were the case, we would feel the same way. In fact, you have all the control you'll ever need. In the background, 4tH keeps track of everything that is happening.

We've already discussed the header of the H-code. All status-information is stored here. And it's all available. May be you'll find it a little more complex and intimidating, but you can easily master it. Let's take a look at this piece of code:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));
```

This piece of code tries to compile the classic "Hello world" program. But did it compile? If `comp_4th()` returned a NULL-pointer, you know there was not enough memory. But like any other compiler, there are a million other things that can go wrong. Although other API functions will refuse unreliable H-code, sometimes we want to check it ourselves and take alternative action if necessary.

All information regarding the status is saved in the header. But if `comp_4th()` returns a NULL-pointer there is no header. So we have to check that first:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL) printf ("Not enough memory\n");
```

If the program enters the 'else' clause we know that a header exists. Now we need to check the status. Did an error occur? There are two members in the header we can check. First, 'ErrNo' which contains an error-code. If 'ErrNo' contains '0', there were no errors:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      if (Program->ErrNo == 0)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("There were errors\n");
}
```

Note that the member 'ErrNo' is closely linked to the H-code. That is hardly surprising since it is part of the H-code! But we still don't know which error occurred.

Fortunately, there is a predefined array of error-messages we can use. It is called `errs_4th[]` and you can use it without declaring it explicitly, since '4th.h' takes care of that. If you have correctly built the library it will automatically be linked in:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      if (Program->ErrNo == 0)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("Error: %s\n", errs_4th [Program->ErrNo]);
}
```

Of course, checking error-codes by the number is not ideal from a maintenance point of view. In '4th.h' you'll find a lot of #define-s describing these errors. The mnemonic for 'no errors' is 'M4NOERRS', so we can slightly alter our program to:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
     printf ("Not enough memory\n");
else {
      if (Program->ErrNo == M4NOERRS)
            (void) exec_4th (Program, 0, NULL, 0);
      else
            printf ("Error: %s\n", errs_4th [Program->ErrNo]);
}
```

In the next section we'll show how you can help the 4tH-programmer to pinpoint his errors even more precisely.

18.11 A first look at dump_4th()

Although 4tH can tell you what error you made and where you made it, you may find it pretty hard to locate it anyway. That is because 4tH makes a reference to the compilant instead of the source.

That is because 4tH preprocesses the source and never looks further ahead than one single word, so a reference to the source wouldn't help you much anyway. That is the bad news.

The good news is that the instructions 4tH uses internally are virtually identical to the ones you used in your source. If you decompile the program you should still be able to recognize your source. The function `dump_4th()` is essentially a decompiler. Let us show you a small part of a program by Leo Brodie we converted to 4tH:

```
VARIABLE SPAN
: EXPECT ACCEPT SPAN ! ;

16 CONSTANT #NAME
 8 CONSTANT #EYES
16 CONSTANT #ME               ( length of fields )

#NAME STRING NAME
#EYES STRING EYES
#ME   STRING ME               ( calculate values )

...
```

If you decompile the entire program you will get a listing, which consists of two parts. First the header:

```
4tH Message        : No errors at word 80
Object size        : 81 words
String size        : 208 chars
Variables          : 1 cells
Strings            : 40 chars
Reliable           : Yes
```

First it will present the current status of this Hcode program. The words are numbered and we begin counting at zero. This means this program is okay, since word 80 is the very last word. We can derive that information from the second field that lists that there are 81 words, numbered from 0 to 80. The third field tells us there are 208 characters stored in the String Segment.

The next two fields tell us something about the runtime-environment. The total number of strings we defined take up 40 bytes and we defined one single variable. Finally, 4tH tells us this piece of Hcode is reliable. That means it can be saved to disk or executed. If it had told us the Hcode was *not* reliable, we could still have decompiled it. Otherwise it could get very hard to pinpoint an error. Next is the decompiled program itself:

```
[ 0] branch       (4)
[ 1] accept       (0)
[ 2] variable     (0)
[ 3] !            (0)
[ 4] exit         (0)
...
```

As you will see, you can still tell what this program is all about. Since 4tH has no dictionary, but uses a symbol-table, all lexical references are gone. There is no indication that the first word was ever called 'EXPECT' or that variable #0 was named 'SPAN'. In fact, if you would name them differently, it would still compile to the very same Hcode.

The bracketed numbers are the 'addresses' of the words. Then it prints the name of the compiled token. Finally the argument part of the word is printed within parentheses.

Not all tokens have arguments. 'ACCEPT' and 'EXIT' don't need one. They either take their arguments from the stack or don't have any. But 'LITERAL' and 'BRANCH' do need them. 'LITERAL' needs the value of the number it has to throw on the stack and 'BRANCH' needs the address it has to branch to (in fact, it branches to the next token after the indicated address). The interpreter "knows" which token has a valid argument and which ones it can ignore.

But you surely want to know how you can integrate this decompiler into your own programs. Like all other functions, it needs an Hcode-pointer. It also needs a device where you can send the report to.

To give you maximum flexibility we used an open stream, so you can use the screen, a printer or a file. A disadvantage is you have to open and close the file yourself when applicable.

We also gave you the opportunity to do a partial listing. You can tell `dump_4th()` what range you want to decompile. These parameters are protected too. If you feed `dump_4th()` an invalid range it will try to figure out what range is most applicable. This allows you to do a full listing with minimum effort by issuing a range from word 0 to word -1.

A sample application may look like this:

```
FILE *ErrFile;
Hcode *Program;

/* other code */

if ((ErrFile = fopen ("error.lst", "w")) == NULL)
      printf ("Cannot open file\n");
else {
```

```
        dump_4th (Program, ErrFile, 0, -1);
        if (fclose (ErrFile)
            printf ("Error closing file\n");
}
```

If you want to print it to screen, you can use either 'stdout' or 'stderr'. Note that 'stderr' cannot be redirected easily under MS-DOS, so we'll use 'stdout' here:

```
Hcode *Program;

/* other code */

dump_4th (Program, stdout, 0, -1);
```

You can always provide a report when compiling or you can use error-checking to decide whether you execute or save Hcode or print a report. This is the listing of a complete compiler:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      char *source;
      Hcode *Program;

      if (argc == 2) {
            source = open_4th (argv [1]);
            Program = comp_4th (source);

            if (Program == NULL)
                printf ("Not enough memory\n");
            else {
                  if (Program->ErrNo == M4NOERRS)
                     (void) exec_4th (Program, argc - 1, argv + 1, 0);
                  else dump_4th (Program, stdout, 0, -1);

                  free_4th (Program);
                  return (EXIT_SUCCESS);
            }
      }
      return (EXIT_FAILURE);
}
```

This program loads a source, compiles it and executes the resulting Hcode if no errors occurred. It tells you when there is not enough memory and provides a decompiler-listing on screen when a programming error was made. Pretty neat, huh?

18.12 A first look at cgen_4th()

`cgen_4th()` allows you to create native standalone programs with minimal effort. It is a lot like `save_4th()`. All you need is a Hcode object in memory and an open stream. The stream will allow you to send the C program `cgen_4th()` generates to screen, file or a printer. A sample application could look like this:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      Hcode *Program;
      cell Result;
      FILE *CFile;

      if (argc == 2) {
            if ((CFile = fopen ("myfile.c", "w")) != NULL) {
                  Program = load_4th (argv [1]);
                  cgen_4th (Program, CFile);
                  free_4th (Program);
                  fclose (CFile);
                  return (EXIT_SUCCESS);
            }
      }
      return (EXIT_FAILURE);
}
```

This program will load an HX file and send a complete C source to "myfile.c". You can compile it by issuing:

```
cc myfile.c exec_4th.c errs_4th.c -o myfile
```

or, if you have the 4tH library installed by:

```
cc myfile.c -o myfile -l4th
```

Just like any other 4tH related C program. You will have a native executable for your platform and nobody will ever know it's actually a 4tH program. You can make it even easier on yourself by creating a small `Makefile` in your current working directory[1]:

```
# GNU Make - implicit rules for 4tH
# Copyright 2006, Hans Bezemer
%.c : %.4th
        4th cgq $< $@
%.c : %.hx
        4th lgq $< $@
CC=gcc
CFLAGS=-fsigned-char -Wall -O3 -s
LDLIBS=-l4th
LDFLAGS=-s
```

Note this is only an example, so you may have to change it for your system. Now copy `4th.h` to the directory where the 4tH source is located and you're done:

[1] That is the directory where you are when you call 4tH.

```
$ make examples/eliza
4th cgq examples/eliza.4th examples/eliza.c
gcc -fsigned-char -Wall -O3 -s -c examples/eliza.c -o examples/eliza.o
gcc -s examples/eliza.o -l4th -o examples/eliza
rm examples/eliza.c examples/eliza.o
$ examples/eliza
HI! I'M ELIZA. WHAT'S YOUR PROBLEM?
>
```

Isn't that easy? But there are even more ways to use embedded 4tH as we will see.

18.13 Converting HX-files

With the 4tH program `bin2h.4th` you can convert HX-files to portable C-source. This opens a whole new range of applications. `cgen_4th()` is a quick way to create standalone programs, but `bin2h.4th` allows you to embed highly compacted HX code into your C program.

This is particularly useful when memory is tight, because you can load the HX code when it is actually needed and discard it afterwards. Furthermore, you can have several HX code snippets inside your C program, which is not possible when using `cgen_4th()`.

On the downside, HX code is a little more difficult to handle. `bin2h.4th` just creates the embedded code, not an entire program. You will have to write that yourself. Furthermore, in order to use `bin2h.4th` you need to create an HX file first. Still, using `bin2h.4th` is dead easy:

```
4th cxq bin2h.4th HelloWorld myprog.hx myprog.h
```

You might have noticed that the only thing you have to provide is the name of the HX file (`myprog.hx`), the name of the include file (`myprog.h`) and the name of the variable in which the embedded HX code is stored (`HelloWorld`).

18.14 A first look at fetch_4th()

A typical `bin2h.4th` generated includefile looks like this:

```
static unit HelloWorld [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x5c', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xfd',
};
```

Since it contains compiled code there is no need for functions like `comp_4th()`. However, HX code can not be fed to `exec_4th()` directly. It has to be loaded first. The function `load_4th()` does this automatically. There is a function in the 4tH API to load bytecode from memory, called `fetch_4th()`. Just pass the HX code pointer to it:

```
Hcode* Program;
Program = fetch_4th (HelloWorld);
```

Now the bytecode is installed and can be executed by `exec_4th()` in the usual way:

```
(void) exec_4th (Program, 0, NULL, 0);
```

In this case, it will simply print "Hello world!" to your screen.

18.15 A first look at store_4th()

And what if you don't want to store bytecode on disk, but in memory? Well, you can do that too. 4tH provides the function `store_4th()`, which takes a Hcode pointer, a pointer to a buffer and the size of that buffer. It is very easy to use. Just create a sufficiently large buffer, either dynamic or static, and load the Hcode. No need to worry about buffer overflow, when properly used `store_4th()` will prevent such mishaps. It will even return the amount of bytes it has written:

```
#include "4th.h"
#include <stdlib.h>

int main (int argc, char **argv)
{
  Hcode *Object;                       /* Hcode object */
  unit   Buffer [1024];                /* memory allocated to bytecode */
  char   *Program;                     /* 4tH sourcecode */
  size_t MySize;                       /* number of bytes written to memory */

  Program = open_4th ("hello.4th");    /* create sourcecode */
  Object  = comp_4th (Program);        /* compile and save the sourcecode */
  MySize  = store_4th (Object, Buffer, sizeof (Buffer));
  printf ("%s, %ld bytes used\n", errs_4th [Object->ErrNo], MySize);
  free_4th (Object);                   /* destroy the Hcode object */

  Object  = fetch_4th (Buffer);        /* read the bytecode from 'Buffer' */
  exec_4th (Object, 0, NULL, 0);       /* execute the Hcode object */
  free_4th (Object);                   /* destroy the Hcode object again */
  return (EXIT_SUCCESS);               /* signal 'everything ok' */
}
```

This program compiles some 4tH source, saves the HX code in a buffer and then discards the original Hcode. Finally, the HX code is reloaded, run and discarded again. Note that `store_4th()` supports hibernation, like its close brother `save_4th()`. When this program is run it will display:

```
No errors, 36 bytes used
Hello world!
```

Needless to say that you can do very neat things with this function, like paging programs in and out very quickly using *very* little memory, storing multiple programs in a single buffer, etc. Use your imagination!

18.16 Examples of embedded HX code

The include-files `bin2h.4th` generates contain global variables. You can either integrate them in your sourcecode or include them, e.g:

```
#include <stdlib.h>
#include "4th.h"

#include "hello.h"
```

or:

```
#include <stdlib.h>
#include "4th.h"

static unit HelloWorld [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x5c', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xfd',
};
```

It's really up to you. You can install and uninstall HX code as often as you want. You can also have multiple instances of the HX code in memory if you need to. E.g. this is perfectly valid:

```
#include <stdlib.h>
#include "4th.h"

static unit HelloWorld [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x5c', '\x03', '\x08', '\x02', '\x02', '\x02', '\x0d', '\x08', '\x08',
  '\x08', '\x05', '\x08', '\x02', '\x48', '\x65', '\x6c', '\x6c', '\x6f',
  '\x20', '\x77', '\x6f', '\x72', '\x6c', '\x64', '\x21', '\x00', '\xfd',
};

int main (int argc, char** argv)
{
      Hcode*      Instance1;
      Hcode*      Instance2;
                       /* load two instances of HX code */
      Instance1 = fetch_4th (HelloWorld);
      Instance2 = fetch_4th (HelloWorld);
                       /* execute both instances */
      (void) exec_4th (Instance1, 0, NULL, 0);
      (void) exec_4th (Instance2, 0, NULL, 0);
                       /* free first instance */
      free_4th (Instance1);
                       /* execute and free second instance */
      (void) exec_4th (Instance2, 0, NULL, 0);
      free_4th (Instance2);
                       /* reinstall first instance and execute */
```

```
        Instance1 = fetch_4th (HelloWorld);
        (void) exec_4th (Instance1, 0, NULL, 0);
        free_4th (Instance1);

        return (EXIT_SUCCESS);
}
```

The combination of different pieces of HX code is possible too. This code contains two pieces of HX code. The first one adds up two numbers, the second one divides two numbers. Both return the result of the calculation to the variable "Result":

```
#include <stdlib.h>
#include <stdio.h>
#include "4th.h"

/* app dup @ swap cell+ @ + out ! */

static unit Addition [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x5c', '\x03', '\x08', '\x02', '\x09', '\x08', '\x08', '\x08', '\x08',
  '\x39', '\x02', '\x0a', '\x11', '\x07', '\x10', '\x33', '\x07', '\x0b',
  '\x39', '\x02', '\x07', '\x08', '\xe3'
};

/* app dup @ swap cell+ @ / out ! */

static unit Division [] = {
  '\x01', '\x02', '\x04', '\x00', '\xff', '\xff', '\xff', '\x7f', '\x04',
  '\x5c', '\x03', '\x08', '\x02', '\x09', '\x08', '\x08', '\x08', '\x08',
  '\x39', '\x02', '\x0a', '\x11', '\x07', '\x10', '\x33', '\x07', '\x0e',
  '\x39', '\x02', '\x07', '\x08', '\xe6'
};

int main (int argc, char** argv)
{
      Hcode*      Instance;
      cell        Result;
                        /* load addition HX code */
      Instance = fetch_4th (Addition);
                        /* execute: add 5 to 7 */
      Result = exec_4th (Instance, 0, NULL, 2, (cell) 5, (cell) 7);
                        /* free instance */
      free_4th (Instance);

                        /* load division HX code */
      Instance = fetch_4th (Division);
                        /* execute: div Result by 6 */
      Result = exec_4th (Instance, 0, NULL, 2, Result, (cell) 6);
                        /* free instance */
      free_4th (Instance);
                        /* print Result and exit */
      printf ("Result: %ld\n", (long) Result);
      return (EXIT_SUCCESS);
}
```

There are no restrictions whatsoever to the use of the rest of the 4tH API, since `fetch_4th()` returns an ordinary Hcode pointer. For instance, you can still use `load_4th()` to load additional HX-files. Happy embedding!

18.17 Suspended execution

People are wondering how they can enable hibernation[2]. Well, you can't. Only a 4tH programmer can do that by using the word 'PAUSE'. Normally, 4tH closes all files, releases the runtime environment and exits. When 'PAUSE' is encountered, 4tH creates a stackframe, closes all files and exits. *All* API functions recognize a dormant VM and act accordingly, so there is not much you can do. You can recognize a dormant VM by examining the "Offset" member of the Hcode structure. If is non-zero, you got a dormant VM at your hands:

```
Object->Offset
```

Still, there is a lot you can do with a dormant VM as long as you have created special provisions in your 4tH program. Take this very simple interpreter:

```
include lib/null.4th
include lib/throw.4th

\ The words supported by the interpreter
: bye ." ZZzzzz.." cr pause ." Waky, waky!" cr ;
: test ." Test successfully executed!" cr ;
: _+ + ;
: _- - ;
: _* * ;
: _/ / ;
: _.( [char] ) parse type ;
: _cr cr ;
: _. . ;

\ The dictionary of the interpreter
create dictionary
  ," bye"       ' bye ,
  ," test"      ' test ,
  ," +"         ' _+ ,
  ," -"         ' _- ,
  ," *"         ' _* ,
  ," /"         ' _/ ,
  ," .("        ' _.( ,
  ," cr"        ' _cr ,
  ," ."         ' _. ,
  NULL ,

: notfound e.user throw ;             \ unknown command: throw exception

include lib/interprt.4th

\ The interpreter itself
```

[2] Also referred to as 'hibernation', 'sleeping VMs' or 'dormant VMs'.

```
: go ['] interpret catch if ." Oops!" cr then ;
: prompt ." OK" cr refill drop go ;
: script 1 args tib place 0 >in ! go bye ;
: run begin argn 2 = if script else prompt then again ;

run
```

Note you can only *suspend* this program. When you provide a commandline argument, it will interpret it as a script and execute it. That is neat! Now take a look at this C program:

```
#include "4th.h"
#include <stdio.h>
#include <stdlib.h>

#define HX "tiny.hx"

/*
This function starts an interactive session
*/

void Prompt (Hcode *Program)
{
  puts ("Keyboard control enabled..\n");
  (void) exec_4th (Program, 0, NULL, 0);
  puts ("\nHost control enabled..");
}


/*
This function builds an argument list and starts a script
*/

void Script (Hcode *Program, char *script)
{
  char *(Args [3]);                    /* mimics argv[][] */

  Args [0] = HX;                       /* the program name */
  Args [1] = script;                   /* the script itself */
  Args [2] = NULL;                     /* the list terminator */
  puts ("Script control enabled..\n");
  (void) exec_4th (Program, 2, Args, 0);
  puts ("\nHost control enabled..");
}


/*
Host program, which calls the shots
*/

int main (int argc, char** argv)
{
  Hcode *Program;
  Program = load_4th (HX);
```

```
  if (Program)                          /* if loading was successful */
     {
       Prompt (Program);                /* interactive session */
       Script (Program, ".( This is a test) cr test test test");
       Prompt (Program);                /* interactive session */
       Script (Program, ".( Leaving an item on the stack) cr 23 45 +");
       Prompt (Program);                /* interactive session */

       puts ("Host shutting down..");
       free_4th (Program);              /* free resources */
     }

  return (EXIT_SUCCESS);
}
```

There is a function that supplies arguments and executes a script and a function that does not supply arguments and enters interactive mode. In `main()` we call these functions alternately. It is a pretty mean program! You can allow a user to do what he wants and when he relinguishes control, you can execute whatever you want from your C program. It is so mean that even when the user enters something like:

```
11 13 * 4 + bye .( I still gotta do this!) cr
```

The part after 'BYE' will *still* be executed before the interpreter starts executing a script. Whatever is on the stack *stays* on the stack, no matter if it was left there by a script or an interactive session. Now *that* is powerful! But there are more neat things you can do with suspended execution. You can also use it to read or write 4tH data. That may seem a bit tricky at first, but as a matter of fact it is very easy. Take a look at this example:

```
10 constant NUMCELLS                  \ array size
NUMCELLS array  Xarray                \ array to be exported
32 string Xstring                     \ string to be exported

: export out ! pause ;                \ save literal and sleep

: run
  Xarray export                       \ export Xarray
  Xarray 10 bounds do i ? loop cr     \ show array contents
  s" This comes straight from 4tH!" Xstring place
  Xstring export                      \ export Xstring
  ." Famous last words.." cr          \ final message
;

run
```

You probably remember that the contents of OUT are returned by `exec_4th()`, so what 'EXPORT' actually does is saving an address of a variable before returning control to the C program. 'EXPORT' is called almost immediately in this example. Obviously something has been done, since the contents of 'XARRAY' are dumped. Then a string variable is initialized and 'EXPORT' is called again. Finally a message is printed. Doesn't that make you wonder what the C program does?

```
#include "4th.h"
#include "cmds_4th.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MK_CP(a,b) ((a)->CellSeg + STACKSIZ + SYS4TH + (b))
#define MK_UP(a,b) ((a)->UnitSeg + (b))

#define NUMCELLS 10
#define HX "bulk.hx"

int main (int argc, char** argv)
{
  cell   Pointer;
  cell   Carray [] = { 0L, 10L, 20L, 30L, 40L, 50L, 60L, 70L, 80L, 90L };
  Hcode *Program;
  cell  *p;
  int    x;

  Program = load_4th (HX);
  if (Program)                          /* if loading was successful */
     {                                  /* fill the 4tH array */
        Pointer = exec_4th (Program, 0, NULL, 0);
        p = MK_CP (Program, Pointer);
        for (x = 0; x < NUMCELLS; x++, p++) *p = Carray [x];
                                        /* show array and setup string */
        Pointer = exec_4th (Program, 0, NULL, 0);
        puts (MK_UP (Program, Pointer));
                                        /* show famous last words */
        Pointer = exec_4th (Program, 0, NULL, 0);
        free_4th (Program);             /* free hcode */
     }

  return (EXIT_SUCCESS);
}
```

The first thing you notice are the two macros, `MK_CP()` and `MK_UP()`. They have been defined to create pointers to the Integer Segment and the Character Segment. It is very easy: just call 4tH from C and export your variable of choice with 'EXPORT'. In this program, the value is stored in 'Pointer'. Note that it is *essential* that you know *exactly* what has been returned: a CELL or a UNIT.

In this example, `exec_4th()` first returns a cell, so we have to call `MK_CP()` to convert it to a pointer to a cell. After that we can transfer the contents of a C array to 4tH. The second time `exec_4th()` returns a unit, so we'll have to call `MK_UP()` to convert it to a pointer to an unit. After that we can use the pointer to print the string. Then `exec_4tH()` is called for the last time and a string is printed. Note that you don't have to let 4tH finish. You can call `free_4th()` any moment you want to.

That is still not enough for you? You'd like to see something even fancier? Well, what do you think about this tiny cooperative multitasker:

```
#include "4th.h"
```

```
#include <stdlib.h>

#define MAX_TASK 16

Hcode *Processes [MAX_TASK];           /* process slots */

/*
This routine adds a task to the process space. It returns the PID
if successful, CELL_MIN if not.
*/

cell task_4th (Hcode **Process, Hcode *Task)
{
  cell x;

  if (Task)                            /* if there is a task */
    for (x = 0; x < MAX_TASK; x++)     /* search all process slots */
      if (Process [x] == NULL)         /* if empty slot found */
        {
          Process [x] = Task;          /* add the task */
          return (x);                  /* and return success */
        }

  return (CELL_MIN);                   /* if not, return failure */
}


/*
This routine searches the process space for a given task. If found, it is
executed and the return value returned to the calling process. If not found,
it signals termination.
*/

cell wake_4th (Hcode **Process, cell task)
{
  cell x, y;

  for (x = task; x < MAX_TASK; x++)    /* search slots beginning with task */
    if (Process [x])                   /* if it is an active process */
      {                                /* execute it */
        y = exec_4th (Process [x], 0, NULL, 1, x);
                                       /* if the process has terminated */
        if (Process [x]->Offset == 0)
          {                            /* free the process */
            free_4th (Process [x]);    /* and set the pointer to NULL */
            Process [x] = NULL;
          }
                                       /* return next process number */
        return (y == CELL_MIN ? ++x : y);
      }
                                       /* signal no active processes found */
  return (task == 0 ? CELL_MIN : MAX_TASK);
}
```

```
/*
This is the true multitasker. It keeps on looping through the processes until
it receives a kill signal from wake_4th().
*/

void multi_4th (Hcode **Process)
{
  cell pid = 0L;                        /* process id */

  while (pid >= 0L)                     /* seach all process slots */
    {
      pid = wake_4th (Process, pid);    /* now wake this process */
      if (pid >= MAX_TASK) pid = 0;     /* and loop around */
    }
}


/*
Host program, which calls the shots
*/

int main (int argc, char** argv)
{
  cell x;
                                       /* set all slots to NULL */
  for (x = 0; x < MAX_TASK; x++) Processes [x] = NULL;
                                       /* now load two processes */
  printf ("Process %ld installed\n", task_4th (Processes, load_4th ("1.hx")));
  printf ("Process %ld installed\n", task_4th (Processes, load_4th ("2.hx")));

  multi_4th (Processes);               /* start the multitasker */
  return (EXIT_SUCCESS);               /* return success */
}
```

Since `load_4th()` returns a NULL pointer when it doesn't succeed, it is safe to pass it to `task_4th()`. Even if programs aren't suited to do any multitasking, you can use this program: they will just be executed consecutively. `task_4th()` will add a program to the process list. `wake_4th()` will try to wake up a program beginning with the PID which is passed to it. Note that the PID is passed to the 4tH program, so it can be queried. If 'OUT' contains a valid PID, it will be the next process that the program will try to awaken. When a process terminates, it is taken from the process list. When all processes have terminated, `wake_4th()` returns CELL_MIN. Note that a program can terminate the multitasker by returning a non-zero value. That's as fancy as it gets, folks!

The String Segment and the Code Segment are *read-only* for a reason. Although you *can* access them, we advise you *not to attempt it*. The same goes for the Stack Area and the System Area. The interfaces we've provided here are a relatively safe way to exchange information between 4tH and C, but if you make any errors your program might crash. Therefore, it is a good idea to add exception handlers to critical sections of your 4tH program.

18.18 Useful variables

We've already seen that `dump_4th()` can provide you with a lot of information about Hcode. If you need this information, you don't have to call `dump_4th()`. The `dump_4th()` function simply uses the information that is already available. This small program shows you how to obtain it:

```
Hcode *Program;

Program = comp_4th (strdup (".\" Hello world\" cr"));

if (Program == NULL)
      printf ("Not enough memory\n");
else {
      printf ("Error#       : %u\n", Program->ErrNo);
      printf ("Error at word: %d\n", Program->ErrLine);
      printf ("Object size  : %d\n", Program->CodeSiz);
      printf ("String size  : %u\n", Program->StringSiz);
      printf ("Var. offset  : %u\n", Program->Offset);
      printf ("Variables    : %u\n", Program->Variables);
      printf ("Strings      : %u\n", Program->Strings);
      printf ("Reliable     : %s\n", Program->Reliable ? "Yes" : "No");
}
```

The labels are kept the same as in `dump_4th()`, so if you need more information, read that section again.

Chapter 19

Modifying 4tH

19.1 Introduction

A good scripting language must be easy extendible. We will cover the most common extensions. You will acquire indepth knowledge of the inner workings of 4tH. All of 4tHs functions are a toolkit in itself and can be put to your own use (especially `comp_4th()`, which is 4tHs most complex function).

As we proceed, there will be more files to edit and the modifications will get more complex. Be sure you mastered the previous extensions, before you get on with the more elaborate ones. A good knowledge of C is required for most operations.

19.2 A closer look at comp_4th()

As we already know, `comp_4th()` compiles source to H-code. First of all, we need to have a source. This is a simple character array, which is pointed to by "Source". Then an H-code header is created.

The header is initialized by `InitObject()`, which calls `ParseText()` to get the initial size of the Code Segment. The initial size of the Code Segment is stored in the header-member "CodeSiz".

`ParseText()` calls two other functions: `ParseDirectives()`, which picks out all directives and calls `ParseStrings()` if need be, which parses the source for string-arguments.

If `ParseStrings()` encounters a '[NEEDS' or 'INCLUDE' directive, it will call `DoNeeds()`, which will create enough room for the file to be included and read the actual file. If `DoNeeds()` fails, it will set the "ErrNo" member of the header accordingly and exit. `MakeRoom()` just moves the last part of the source to the end of the reallocated space. Since all variables are adjusted accordingly, `ParseText()` will pick up parsing after the '[NEEDS' or 'INCLUDE' directive. It never knows the difference.

After parsing `ParseText()` returns the initial size of the Code Segment (the number of words). The function `ParseText()` sets three important variables:

VARIABLE	CONTENT
SourceStrings	The number of source-words
SourceWords	The size of the Code Segment
SourceSymbols	The size of the symbol-table

Table 19.1: comp_4th() variables

It is imperative to know these numbers since the 4tH-environment has to size its resources. No resizing is required until all compilation is done. All allocated resources should be large enough to contain the resulting compilant, so extending these resources should never be necessary. In the end, they are only shrunk to their actual sizes. Now we can start to allocate the compiler-resources:

- Code Segment
- Symboltable
- Controlstack

This is done by simply calling `AllocResource()`. Note that it is not necessary to allocate the String Segment. Strings remain in memory already allocated to the source and are just shifted to the front.

Then compilation can begin. First, the variable "Cursor" is set to the beginning of the source. Then every call to `GetNextWord()` sets the variable "CurrentWord" to point to the next source-word. If there are no more source-words, "CurrentWord" is set to NULL and compilation is terminated.

Now compilation really gets off. It is important to know that not all words are created equal. There are five kinds of words:

- Immediate words
- Words
- Constants
- Symbols
- Numbers

So, there are five distinct functions which handle these words:

- Immediate words are compiled by `GetImmediate()`
- Words are compiled by `GetWord()`
- Constants are compiled by `GetConstant()`
- Symbols are compiled by `GetSymbol()`
- Numbers are compiled by `comp_4th()`

If the first four functions fail there is one more chance that this is a valid source-word: it might be a number. So, the source-word is converted to a number in the current radix. If this works, the number is compiled. If it doesn't, it isn't a valid source-word and the member "ErrNo" is set.

Compiling is done by a single function called `CompileWord()`. You just provide the token and its argument and `CompileWord()` takes care of the rest.

When all compiling is done, we can discard the symboltable and the controlstack. This is done by calling the function `FreeResource()`. It is called by `ReallocSegs()`, which shrinks the Code Segment and the String Segment to their actual sizes, and `AbortCompile()`, which shuts the compiler down in case of an error.

If an error occurred before any compiling took place `AbortCompile()` also discards the Code Segment and the String Segment, thus returning a bare H-code header. As you already know, the error-code is stored in the member "ErrNo" of the H-code header.

If the error occurred after words have been compiled, this partial compilant is not discarded. Instead the member "Reliable" is set to FALSE, indicating that the compilant cannot be run or saved. It can be decompiled, thus enabling the user to track the error.

In the next sections we will take a closer look at the three main tables in `comp_4th()`, which contain all of 4tHs built-in words.

19.3 Adding a constant

Adding a constant is very easy. You only have to update a single table in `comp_4th()`. The table with constants, which is embedded in `GetConstant()` has four members:

1. Length byte
2. Name
3. Type
4. Value

The length-byte is used to quickly scan the table. All words are skipped until it reaches the constants with the same length. Then it starts to compare the names.

What happens then depends on the argument "mode". When it equals W_EXEC, the constant is compiled (a literal with the value as argument). Otherwise, only the index in the table is returned, pointing to the constant we just found. This enables mere searches in the table. When a name isn't found at all, it returns MISSING.

Say, we want to add a constant called "TWENTY". At least, we know its name: "TWENTY". The name "TWENTY" is six characters long. And of course, we want to compile the number "20" each time it is referenced in the source. To make `comp_4th()` compile a number, we need the token LITERAL. The four members are:

1. Length byte = 6
2. Name = "TWENTY"
3. Type = LITERAL
4. Value = 20

Since every constant is a signed 32 bits number, we add the modifier 'L' to the "20". So the complete line we have to add to the table reads:

```
{ 6, "TWENTY", LITERAL, 20L },
```

Now we have to insert this line into the table. Note that constants with the same length have to be grouped together:

```
{ 5, "INPUT", LITERAL, F4_READ },
{ 6, "OUTPUT", LITERAL, F4_WRITE },
{ 6, "APPEND", LITERAL, F4_APPND },
{ 7, "(ERROR)", LITERAL, CELL_MIN },
```

We decide to put our constant behind "APPEND". Of course, we could have put it behind "INPUT" or "OUTPUT" as well:

```
{ 5, "INPUT", LITERAL, F4_READ },
{ 6, "OUTPUT", LITERAL, F4_WRITE },
{ 6, "APPEND", LITERAL, F4_APPND },
{ 6, "TWENTY", LITERAL, 20L },
{ 7, "(ERROR)", LITERAL, CELL_MIN },
```

Now recompile 4tH and run this simple program:

```
twenty . cr
```

It should compile without errors and print "20". That's all there is to it! Not to difficult to begin with, huh?

19.4 Adding a word

Now for something a little more difficult. Let's say we want to implement 'NIP'. Of course, 'NIP' is already available, but if you compile this program:

```
1 2 nip
```

You will see that it actually compiles to:

```
[0]   literal     (1)
[1]   literal     (2)
[2]   swap        (0)
[3]   drop        (0)
```

So, 'NIP' is actually expanded to 'SWAP' and 'DROP'. That is because 'NIP' can be defined as:

```
: nip swap drop ;       ( n1 n2 -- n2)
```

It removes the number under the top of stack. We call this an inline macro, which we will discuss later on. If you really want to try out 'NIP' in the following example, you have to remove a line from `ImmedList[]`:

```
{ 3, 0, 1, "NIP", "", DoNip },
```

Your compiler might complain about unused functions, but it will work.

Inline macros are not the only way to add 'NIP' to 4tH. We can also implement 'NIP' as a word. Most words can be found in the function `GetWord()`. There is a single table, which is laid out like the table of constants we encountered in the previous section, so:

1. Length byte = 3
2. Name = "NIP"

But instead of the value of the constant, we have to add something else. That something is called the token. The tokens are defined in "cmds_4th.h". Let's have a look:

```
#define PAUSE    100
#define VECTOR   101
#define ENVIRON  102
#define PLITERAL 103
#define FSEEK    104
#define FTELL    105

/* ranges */
#define LastWord4th  FTELL
#define LastMsg4th   M4CABORT
```

Well, in fact you can place a new token anywhere, but then you have to renumber all other tokens. The easiest way is to place it after the last token you defined, which in this case is "FTELL". The token "FTELL" has the number "105". Well, the next number is "106" and that is the number "NIP" is going to get. So we add:

```
#define PAUSE    100
#define VECTOR   101
#define ENVIRON  102
#define PLITERAL 103
#define FSEEK    104
#define FTELL    105
#define NIP      106

/* ranges */
#define LastWord4th  FTELL
#define LastMsg4th   M4CABORT
```

What part don't you understand? But we're not ready yet. If you look at the #define below the entry we just made, you will see that it says that the last word in 4tH is "FTELL". That is incorrect now. We've just added "NIP". Fixing it is very easy. Just change the line to:

```
#define LastWord4tH     NIP
```

We're done now with "cmds_4th.h". Now we have change "comp_4th.c", so that the compiler can recognize and compile "NIP". Now we can complete the entry for `GetWord()`:

1. Length byte = 3
2. Name = "NIP"
3. Token = NIP

So the complete entry reads:

```
{ 3, "NIP", NIP },
```

And like we did with our constant "TWENTY", we have to give it a place inside the table between all other 3-letter words. We decided that pasting "NIP" between "USE" and "SEEK" would be a good idea (but there are plenty of other places too):

```
{ 3, "HEX", HEX },
{ 3, "USE", USE },
{ 3, "NIP", NIP },
{ 4, "SEEK", FSEEK },
{ 4, "TELL", FTELL },
{ 3, "HEX", HEX },
```

Are we done now? No. The compiler will recognize and compile "NIP", but what does it do? That behavior will have to be defined in "exec_4th.c", but we'll discuss that in the next section.

19.5 A closer look at exec_4th()

Although it is a perfectly acceptable way to create a threaded virtual machine, some C-programmers will shake their head when they take a look at `exec_4th()`. Their comments are usually concentrated around these items, so let's get them out of the way as quickly as possible:

- **QUESTION:** Why this enormous `switch()` statement? Why wasn't a structure used with pointers to functions?
- **ANSWER:** That one was built too, but it proved to be upto four times slower on all platforms. Which is perfectly understandable, because every time you evaluate a token, you have to take the overhead of calling a function into account.
- **QUESTION:** Why are the tokens in a random order?
- **ANSWER:** We have statistically analyzed which tokens are used more often. They are up front. Some C-compilers generate a jumptable. Others generate a repeated "if .. elif .. endif" construction. Compilants produced by the latter perform better when tokens are ordered this way.

But if you think you can build a better one, please do. If it works, please inform us! Now let's get to business..

Since 4tH has a segmented structure, there are special words for each segment, e.g. "C!" for the Character Segment and "!" for the Variable Area. But when one wants a Virtual Machine that checks every access, the parameters of these words need to be checked.

There are very few words that access the Variable Area, so these are checked within the code for the token itself. Others, like the ones accessing the data stack, are used more often. So, special functions were created that allow or check access to those areas. There are thirteen functions you should know about. They form the basic API.

FUNCTION/MACRO	DESCRIPTION
DPOP	Gets an item from the data stack
DPUSH (cell)	Puts an item on the data stack
DFREE (cell)	Checks amount of free space on the data stack
DSIZE (cell)	Checks the number of items on the data stack
DS (cell)	Random acces item on data stack
RPOP	Gets an item from the return stack
RPUSH (cell)	Puts an item on the return stack
RFREE (cell)	Checks amount of free space on the return stack
RSIZE (cell)	Checks the number of items on the return stack
RS (cell)	Random acces item on return stack
unit fetch (cell)	Gets a character from the Character Segment
void store (cell, unit)	Puts a character in the Character Segment
cell toPAD (char*)	Puts a string in the PAD
char* toCstring (cell, cell)	Puts a addr/count string in the PAD

Table 19.2: exec_4th() basic API

We strongly recommend you use these functions when accessing any of these segments. But you'll probably need more than functions to create your code. What about variables?

Well, of course there is a host of variables you can use, but there are three variables that are used more frequently. They are called "a", "b" and "c" and are of type cell. Now, what did NIP do?

```
NIP   ( n1 n2 -- n2)
```

That means the first cell is taken from the data stack and saved, then the second cell is taken from the data stack and dropped and finally the first cell is replaced on the data stack. Note there need to be at least two items on the stack to make it work. Since there will be one item less on the stack after the operation, we don't need to check whether there is enough space. So this will do the trick:

```
DSIZE (2);
a = DPOP;
DDROP;
DPUSH (a);
```

There is even a faster way to do it. This implementation uses the `DS()` macro which allows direct access to the datastack:

```
DSIZE (2);
DS (2) = DS (1);
DDROP;
```

`DS(1)` equals ”Top of stack” and `DS(2)` equals ”Second of stack”. Note that the `DS()` macro *doesn’t adjust the stack pointer*. That is where DDROP comes in. It discards the superfluous item on the stack. Now we add a label and a "break" (which is necessary in a `switch()`):

```
case (NIP): DSIZE (2);
            DS (2) = DS (1);
            DDROP;
            break;
```

Although this is slightly more difficult than the previous API, this implementation is much, much faster. So, now we’re finally done! We can compile the whole thing and test our new command:

```
     2 3 NIP . cr
```

Can we? No, there is one more thing we have to do. Your source will compile OK and execute OK, but it is nowhere to be found when we decompile it. Does that mean we have to edit `dump_4th()`? Wrong again, but we will see that in the following section.

Before we leave, we’ll take a look at another word, OVER. This once differs from NIP, since it leaves more items than it consumes. If we wouldn’t take any precautions, we might run into serious trouble. So, what does OVER do?

```
OVER   ( n1 n2 -- n1 n2 n1)
```

’OVER’ requires two items and leaves three items on the stack. That means we need 3 - 2 = 1 extra item on the stack:

```
DSIZE (2);
DFREE (1);
a = DS (2);
DPUSH (a);
```

Note we *must* use `DPUSH()` to adjust the stack pointer. Of course, there are macros that manipulate the return stack in a similar way. One tip: *be careful with nesting macros*, since you might not get what you want.

Strings are quite another ballgame. In 4tH these are usually address/count strings and consequently incompatible with C, since they are not terminated. Strings in the String Segment are not accessible by a 4tH programmer at all. In order to solve these problems, the PAD was created. The PAD is essentially a circular string buffer where temporary strings are stored. The address and count are usually returned to the 4tH programmer enabling him to manipulate these strings. At the same time you can be assured that they are terminated and C-compatible.

There are two API functions which allow you to access the PAD, `toPAD()` and `toCstring()`. The first one allows you to copy a C string to the PAD. It puts the address on the stack and returns the count. The second one allows you to copy an address/count string to the PAD. It returns a pointer to a C string. In `exec_4th()` there are two C string pointers which you can use, `p` and `q`. A little example: suppose there is an address/count string on the stack which you need to access:

```
DSIZE (2);
a = DPOP;
b = DPOP;
p = toCstring (b, a);
```

Note that the count resides on the top of the stack and is popped first. The address comes after that. You can use `toPAD()` to copy any C string to the PAD, e.g.

```
DFREE (2);
DPUSH (toPAD ("Hello world!"));
```

Note that `toPAD()` already left the address on the stack, so you only need to push the count it returns.

19.6 A first look at name_4th()

May be "name_4th.c" looks like a function, but it definitely isn't. It is a global array that is as global as globals can get. You can refer to it in any program that uses the 4tH library without defining it first. It contains the names of all the tokens, so when we decompile, the tokens get a readable name:

```
#include "4th.h"
#include "cmds_4th.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char** argv)
{
      puts (name_4th [SWAP]);
      return (EXIT_SUCCESS);
}
```

This program will print "swap". Yes, you can use the token-code as an index to this array. We can't make it any easier than that! Now you understand that we can't do the same thing for "NIP", since we didn't add that one. In fact, any program trying it will either print garbage or crash. Let's take a look at the last few lines of `name_4th[]`:

```
  "environ", "+literal",  "seek",    "tell"
};
```

Yeah, we've seen that before. All entries are ordered in the same way as in "cmds_4th.h". Since "NIP" was added at the end of "cmds_4th.h", we have to do the very same thing here. Don't forget the comma after "TELL"

```
  "environ", "+literal",  "seek",    "tell",      "nip"
};
```

Now we are finally done. We can recompile the library (and the compiler) and we can use "NIP" like any other 4tH word.

19.7 Extending the compiler

So far we've only added words that are directly compiled into a token/argument pair. If that was the way all 4tH words worked, we would never have branches, variables or other things that make up a language. In fact, you would never construct this complex architecture, since there are easier ways to achieve the same functionality.

The secret lies in the last major function of `comp_4th()` we have discussed, which is called `GetImmediate()`. When you look at it for the first time, it looks quite like `GetConstant()` and `GetWord()`. The associated table `ImmedList[]` has a length-byte and a name but instead of a token or a value the last field is a pointer to a function.

You'll find these functions above `GetImmediate()` and they all start with `Do..()`. In fact, they are the icing on the cake. They make 4tH a compiler, since they allow non-linear compiling. That includes:

- Branching
- Comments
- Allocation of variables
- String handling
- Assertions
- Constants

If there is something you want to do that you cannot define in a single token/argument, this is the place where you have to be. But before you are starting to make new functions, note there are many functions that you can use.

E.g. when you want to compile a word, you don't have to bother yourself with error-checking or other 4tH-internals. Just make a call to `CompileWord()`:

```
CompileWord (NIP, 0);
```

Extending the compiler is quite easy. We will illustrate that by using a very simple example. We have some compiler-words that handle the radix at compile time. They are:

```
[BINARY]
[OCTAL]
[DECIMAL]
[HEX]
```

Now we want to add a new one called "[SEXTAL]", which sets the radix to six. The radix at compile-time is handled by a single variable called "Base". First we have to make a function, which sets "Base". We call it `DoSextal()`. Note that this function cannot receive or return any values, like all `Do..()` functions:

```
#ifndef ARCHAIC
      static void DoSextal (void)
#else
      static void DoSextal ()
#endif
```

Branching	
MarkLink()	Adds a link to the controlstack
MakeLink()	Makes a back-link
PairLink()	Retrieves a link from the controlstack
CompileMark()	Compiles a token and adds a link to the controlstack
Symboltable	
AddSymbol()	Adds a symbol to the symboltable
MakeSymbol()	Adds the current word as a symbol to the symboltable
GetSymbol()	Retrieves an entry from the symboltable
Parsing	
GetNextWord()	Gets next word in the source
DecodeSymbol()	Decodes a symbol from source
DecodeLiteral()	Gets a compiled literal expression
DecodeOperand()	Gets a compiled expression
DecodeWord()	Gets a name from source
DecodeName()	Gets a previously delared name from source
SkipSource()	Discards all source between two labels
Compiling	
CompileWord()	Compiles a token and its argument
InlineWords()	Compiles a sequence of tokens without arguments
Strings	
MoveString()	Moves a string inside the String Segment
CompileString()	Compiles a token and its associated string

Table 19.3: comp_4th() basic API

```
{
      Base = 6;
}
```

Just place it anywhere before `ImmedList[]` and you're safe. Now we have to make it work. Like the other `Get..()` functions, there is a table called `ImmedList[]` which drives this behaviour, so let's get started:

1. Length byte = 8
2. Symboltable entries = 0[1]
3. Additional tokens = -1[2]
4. Delimiter = ""[3]
5. Name = "[SEXTAL]"
6. Function = DoSextal

Now let's update that table:

[1] See section 19.20
[2] See section 19.14
[3] See section 19.16

```
{ 7,  0, -1, "ALIGNED",     "",    DoDummy },
{ 7,  0,  0, "RECURSE",     "",    DoRecurse },
{ 8,  0, -1, "[SEXTAL]",    "",    DoSextal },
{ 8,  1, -3, "CONSTANT",    "",    DoConstant },
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
```

You're done now! Recompile the program and "[SEXTAL]" has become a part of 4tH. Note that not all words within `ImmedList[]` can be defined that easily. If special tokens are required, you might have to edit other files as well.

19.8 Making aliases

Sometimes you need two different names that do the same thing. Well-known examples are "CHAR" and "[CHAR]" or "I" and "R@". How can this be done?

In fact, it is very simple. Think about it. 4tHs vocabulary is stored in tables. These tables link a name with some kind of behavior. So we have to make two different names that are linked to the same thing. Take a look at this excerpt of `ImmedList[]`:

```
{ 5,  0, -1, "ALIGN",       "",    DoDummy },
{ 5,  0, -1, "CELLS",       "",    DoDummy },
{ 5,  0, -1, "DOES>",       "",    DoDummy },
{ 5,  0, -1, "CHARS",       "",    DoDummy },
{ 5,  1, -2, "TABLE",       "",    DoCreate },
{ 6,  1, -2, "CREATE",      "",    DoCreate },
{ 6,  0, -1, "[THEN]",      "",    DoDummy },
```

You might have noticed a few aliases.

WORD	COMPILED BY
[THEN], ALIGN, CELLS, DOES>, CHARS	DoDummy()
TABLE, CREATE	DoCreate()

Table 19.4: Examples of aliases

That means that if you write your 4tH program you can choose between "TABLE" and "CREATE". It doesn't matter, it will compile to the same thing. But we have to add to that some aliases are created because in Forth they *do* have different meanings, like "[CHAR]" and "CHAR". Read the glossary for details.

We will show you another example. This one comes from `GetWord()`:

```
{ 5, "CHAR+", INC },
{ 5, "CHAR-", DEC },
{ 5, "CELL+", INC },
{ 5, "CELL-", DEC },
```

You see that "CHAR+" and "CELL+" do two very different things. At least in Forth. Within 4tH the smallest addressunit is always an element of that particular segment, thus one. So in 4tH these words are aliases and will compile to the very same code.

19.9 Giving a name to an application variable

We already learned that you can transfer variables to 4tH:

```
Result = exec_4th (Program, 0, NULL, 12, (cell) 31, february,
(cell) 31, (cell) 30, (cell) 31, (cell) 30, (cell) 31, (cell) 31,
(cell) 30, (cell) 31, (cell) 30, (cell) 31);
```

These kind of variables are called "application variables". Of course, you don't *have* to use the same variables every time you call `exec_4th()`, but if you do it may be a good idea to give them a significant name. That makes it a lot easier for a 4tH application programmer to reference your variables. Like everything in 4tH, that is very easy too.

If we take a look at "cmds_4th.h" you will see a C-constant named "VAR4TH". This constant has two functions. First, it shows how many internal 4tH variables there are. Second, it is an index to the first application variable, so 'APP' is defined as "VAR4TH". That means that:

```
app 0 th
```

Is the very first application variable and:

```
app 1 th
```

Is the second application variable. You can do the same. Let's say you have three application variables, which contain the document-number, the page-number and the line-number. You'd like to call them "&DOC", "&PAGE" and "&LINE". The ampersands are not really necessary, but we add them in order to identify the application specific words. To make it work you have to call `exec_4th()` by:

```
Result = exec_4th (Program, 0, NULL, 3, (cell) Doc,(cell) Page,
(cell) Line);
```

Now these are the mappings.

C VARIABLE	4TH EXPRESSION
Doc	app 0 th
Page	app 1 th
Line	app 2 th

Table 19.5: Mapping between 4tH and C variables

Now all we have to do is add constants that are equivalent to these addresses. As we've seen before, we can do that by modifying `comp_4th()`. That is `GetConstant()` to be exact:

```
{ 4, "&DOC",  LITERAL, VAR4TH+0 },
{ 5, "&PAGE", LITERAL, VAR4TH+1 },
{ 5, "&LINE", LITERAL, VAR4TH+2 },
```

That's all! You can now refer to these variables with their proper names.

Note that if you use this technique you are bound to calling `exec_4th()` with these arguments in this order! Failure to do so may cause unpredictable results (but no crashes of course).

C VARIABLE	4TH EXPRESSION	4TH VARIABLE
Doc	app 0 th	&doc
Page	app 1 th	&page
Line	app 2 th	&line

Table 19.6: Mapping between 4tH and C variable names

19.10 Adding new variables

In standard 4tH there are five environment variables, HI, FIRST, LAST, CIN and COUT. There are also five predefined variables, '>IN', 'BASE', 'OUT and the variable pair 'SOURCE'. These variables are initialized by `exec_4th()`, so their initial value should be known by then.

Adding new variables is not difficult. We're going to make a variable that contains 4tHs release number, called "VERSION". First take a look at "cmds_4th.h". It contains a #define called "VAR4TH":

```
/* variables and environs */
#define SYS4TH   3
#define VAR4TH  10
#define ENV4TH   5
```

Now remember that number behind "VAR4TH". You will need it later. Then increment it:

```
/* variables and environs */
#define SYS4TH   3
#define VAR4TH  11
#define ENV4TH   5
```

If you would have preferred to make 'VERSION' an environment, you should also have incremented "ENV4TH". But we assume you'll allow the variable to be overwritten. Now add a symbolic value for the variable. Just append it to the list and increment the number:

```
#define VBASE    5
#define VIN      6
#define VOUT     7
#define VTIB     8
#define VTIBS    9
#define VVERS    10
```

Or if you prefer it to make an environment variable, add it to the environment variable list:

```
#define VHI      0
#define VFIRST   1
#define VLAST    2
#define VCIN     3
#define VCOUT    4
#define VVERS    5

#define VBASE    6
```

```
#define VIN      7
#define VOUT     8
#define VTIB     9
#define VTIBS   10
```

That's all. Now save "cmds_4th.h" and load `comp_4th()` in your editor. This stage is very much like adding a name to an application variable. We simply define a constant that contains the address of our new internal variable. You will remember how we add a constant. Right, we add an entry to the `GetConstant()` table:

```
{ 7, "VERSION", LITERAL, VVERS },
```

Making it an environment variable is very easy too: just replace the 'LITERAL' token by an 'ENVIRON' token:

```
{ 7, "VERSION", ENVIRON, VVERS },
```

All we need to do now is to initialize the variable in `exec_4th()`. Since it is a variable, it resides in the Variable Area of the Integer Segment. The Integer Segment is just a large array of unsigned longs.

The pointer "Stack" points to the beginning of the Integer Segment, which is also the beginning of the Stack Area. The pointer "Vars" points to the area that is assigned to 4tHs variables. Our constant "VERSION" is an index to that array, so the expression "Vars [VVERS]" is a valid reference to our "VERSION" variable.

However, this indexed way of referencing is slower than a pointer. Therefore, we have created pointers that reference these frequently used variables:

```
cell        *In;               /* equivalent of forth >IN */
cell        *Result;           /* return value for apps */

Base        = &(Vars [VBASE]);  /* assign pointer to BASE */
In          = &(Vars [VIN]);    /* assign pointer to >IN */
```

You might have noticed the absence of "Base". Well, since it is referenced elsewhere as well, this is a global variable. But don't worry, there is no need to reference "VERSION" globally. So, we need to define a pointer to a cell, assign it to "Vars [3]" and initialize it:

```
Vars [VVERS] = Version4th;      /* initialize it */
```

That is all! Any questions? Where does "Version4th" come from? It is defined in "cmds_4th.h". Anybody else? Next subject, please.

19.11 Resizing the 4tH environment

You might come up with a situation that the stack isn't big enough. Or that you want to give your programmers deeper nesting. Or that 512 characters isn't just good enough for temporary storage.

Relax! All these things can be changed with very little effort. And after that, you just need to recompile 4tH like we've done before.

There is a single file you need to edit, "cmds_4th.h". You will find several easy to change #defines there.

```
/* compiler */
#define LINKSIZ      64
#define SYMLEN       16

/* interpreter */
#define STACKSIZ     512
#define TIBSIZ       256
#define PADSIZ       512
#define DOTSIZ        64
#define RNDMASK    32767
#define MAXDEVS        8
#define PIPEWAIT 102400L
```

You already know "VAR4TH", since we discussed that one earlier in this document. Right, it determines the number of internal variables! "LINKSIZ" determines the nesting depth. Nesting depth has to do with the number of nested branches, e.g.

```
IF
      IF
            IF
            THEN
      THEN
THEN
```

Each 'IF' puts its address and a reference (I'm an IF) on the flow control stack. Each 'THEN' takes an entry off the flow control stack and takes the appropriate action. So, in the current version of 4tH you can nest upto 64 consecutive conditionals, before you get an error. You may increase or decrease that number.

"SYMLEN" is the maximum length of any name you define, e.g. a colon-definition, a constant, a variable. The default is 16, which is enough to define a name like "multiplications". You can define a longer name, but only the first fifteen characters will be significant. You can increase the maximum number of significant characters, but beware: this can take up a lot of memory!

"STACKSIZ" is the combined size of both data and return stack. This size will do for most applications, since it allows you a combination of high usage of the data stack and low usage of the return stack or vice versa. You might encounter a situation where recursion forces you to resize the Stack Area. Decreasing is possible too, of course, but at your own risk.

"TIBSIZ" is the size of the Terminal Input Buffer used by 'REFILL'. If you need 'REFILL' to accept longer lines than 256 characters and you don't want to allocate your own buffer, resize it.

"PADSIZ" is the size of the scratch PAD, used to store temporary strings. A part of the PAD is reserved to numbers. The size of this area is determined by "DOTSIZ". The rest of PAD (PADSIZ - DOTSIZ) is a circular string buffer. A bigger PAD will allow you to store longer temporary strings that survive longer before getting overwritten.

"RNDMASK" is used to truncate the value returned by `rand()`. Some compilers return a 32 bit number and others a 16 bit number. In order to maintain maximal compatibility across all platforms, 4tH always returns a 16 bit value. You can fiddle around with it, but you will compromise the portability of your 4tH programs.

"MAXDEVS" is the maximum number of I/O devices that 4tH can manage. Note that two of them (STDIN, STDOUT) are already in use, so you can open up to six additional devices concurrently. Finally, " PIPEWAIT" is discussed in detail in the next section.

You will find there are other defines here too. Please, do *not* change them. That just doesn't work. In fact, 4tH just won't work properly anymore.

19.12 Tuning pipe failure detection

Pipes in 4tH are opened by the `popen()` function. This has one big disadvantage. Although `popen()` is able to detect a failed `fork()`, it is unable to detect whether the program was successfully started or not. E.g. if the program cannot be found in the path the pipe fails, although `popen()` has already reported it was successful. In some cases this can have serious consequences.

After careful study we decided to monitor the process for a while and then report success or failure. The default value works very well on most modern systems, but with some systems it may be neccesary to adjust it. This is the case when you experience one of the following symptoms:

- Opening a pipe is slow; there is a long delay before 4tH reports the pipe is successfully opened.
- 4tH reports that the pipe was successfully opened, but most of the time this was not the case.

In that case, you have to adjust a #define in "cmds_4th.h". That is a lot easier than you might think. We've developed a small program to do that. It should be portable across most Unixes:

```
#include <stdio.h>
#include <limits.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>

long TimeBadPipe (void)
{
  FILE *p;                             /* filepointer to pipe */
  long  x;                             /* simple counter */
  int   s = 0;                         /* status of child */

  p = popen ("nosuchprogram", "r");    /* perform a normal popen() */
  for (x = 0; x < INT_MAX && s == 0; x++) waitpid (-1, &s, WNOHANG);
  pclose (p);                          /* close the pipe */
  return (x);                          /* return the count */
}


int main (int argc, char **argv)
{
  int x;                               /* simple counter */
  int now;                             /* return of TimeBadPipe() */
```

```
  int sofar = 0;                        /* highest count */
  int total = 0;                        /* total of all counts */
                                        /* warn the user */
  puts ("Doing 1000 iterations, wait..");
  puts ("(This is going to be messy..)");
  sleep (5);                            /* allow him to read the message */

  for (x = 0; x < 1000; x++) {
    now = TimeBadPipe ();               /* time a bad pipe */
    if (now > sofar) sofar = now;       /* adjust sofar */
    total += now;                       /* add to total */
  }
                                        /* show the results */
  printf ("\nAverage : %d\n", total / 1000);
  printf ("Maximum : %d\n", sofar);
  return (EXIT_SUCCESS);
}
```

If you run it, it will print something like this:

```
Doing 1000 iterations, wait..
(This is going to be messy..)
sh: nosuchprogram: command not found
sh: nosuchprogram: command not found
...
sh: nosuchprogram: command not found
sh: nosuchprogram: command not found
Average : 8685
Maximum : 41235
```

This means that on average the process had to be checked 8685 times, but at no occasion a process died after it had been checked 41235 times. Run it several times, so you will get a good impression of how your particular system behaves. Ignore extremely high and extremely low values. Then take the highest value that pops up several times and change "cmds_4th.h" appropriately:

```
#define PIPEWAIT 49152L
```

Note we rounded the value a little (we're a binary kind of guy) and it doesn't have to be exact. Now you can safely use pipes on your system and the result returned will assure you that the pipe was actually successful opened and ready for use.

If this still doesn't work, you may have to adjust this #define manually: decrease it if opening a pipe is slow, increase it if 4tH incorrectly reports a successfully opened pipe. If you use MS-DOS, just forget all this. We don't provide any pipes there.

19.13 Adding new error messages

Adding new messages is quite simple. It requires not much more than adding a #define and adding a string. You might have noticed that every 4tH message has a mnemonic. Although this is not required, it makes it much easier to read and thus maintain your code.

This mnemonic is no longer than eight characters, all uppercase and begins with "M4" (which stands for Message 4tH). Let's say you've added the ANS-Forth floating point wordset and you want to add the error message "Floating point exception". We'll do it the easy way and just append the message at the end of the table.

First we have to come up with a mnemonic. We decide to use "M4FLOATE". Now we start up our favorite editor and load "4th.h". Then we look for the table with error mnemonics:

```
#define M4NOSTR  24
#define M4NULSTR 25
#define M4DUPNAM 26
#define M4CABORT 27
```

Now we simply add "M4FLOATE" to the end of the table. Since the last message had code 27, we give our message code 28:

```
#define M4NOSTR  24
#define M4NULSTR 25
#define M4DUPNAM 26
#define M4CABORT 27
#define M4FLOATE 28
```

We can now save "4th.h". Now we have to add the message itself. That is done by adding it to "errs_4th.c". That file just contains an array of messages. Note that the messages are listed in order of their codes:

```
  "Unterminated string",
  "Null string",
  "Duplicate name",
  "Compilation aborted"
};
```

If you change that order, your compiler might display the right errorcode, but the wrong error message. Since our mnemonic comes last, our message comes last:

```
  "Unterminated string",
  "Null string",
  "Duplicate name",
  "Compilation aborted"
  "Floating point exception"
};
```

Don't forget adding a comma after the last message! If you don't your compiler will certainly complain about that. Are we done now. No, not quite yet. The 'THROW' routine wants to know which codes are exceptions generated by the system and which one are generated by the user. Why? Because user exceptions do not have messages attached to them! We can change that in "cmds_4th.h".

```
/* ranges */
#define LastWord4th      FTELL
#define LastMsg4th       M4CABORT
```

Now it still points to the "duplicate name" error. We simply change "LastMsg4th" to our mnemonic:

```
/* ranges */
#define LastWord4th      FTELL
#define LastMsg4th       M4FLOATE
```

We're done now! Note that this final step is not necessary when you *insert* messages. Instead, you will have to renumber the table in "4th.h". No two mnemonics may ever share the same error code, remember that! If you don't keep the mnemonics, the errorcodes and the messages properly synchronized you may get some pretty strange error messages. Which is less than helpful.

19.14 Sizing the Code Segment

By default 4tH assumes a 1:1 relationship between a word in source and a compiled word (in the Code Segment). When `ParseText()` is called it will count the number of words in the source. This number is later used to size the initial Code Segment. This 1:1 relationship is not so strange as it may seem at first, e.g.:

```
BL DROP
```

Will compile to:

```
[0] literal      (32)
[1] drop         (0)
```

Two words in source, two compiled words. But there are exceptions too,

e.g.:

```
BL ,
```

Will compile to:

```
[0] ,            (32)
```

That is because ',' does not compile to anything, but changes the previously compiled literal to an constant array element. Note that ',' is an immediate word. In fact, all exceptions to this 1:1 relationship rule are immediate words! The vast majority of 4tH words obey this 'one on one' rule:

- All numbers and constants compile to literals
- All ordinary words compile to a word without argument
- All symbols compile to a word with argument

In a previous chapter we've created an immediate word called "[SEXTAL]". When you take a closer look, you will see that it just changes the base; it doesn't compile to anything. Still, without the proper argument 4tH assumes it will compile a token and reserves space in the Code Segment.

Can you prevent this? Yes, you can. There is a member in the table of `ImmedList[]` which allows you to signal 4tH that it shouldn't reserve space in the Code Segment for "[SEXTAL]". The first field indicates the length of the keyword, the third indicates the correction 4tH should make to the sizing of the Code Segment when this keyword is encountered, the fourth is the keyword itself and the last one is the C function that compiles the word. The second and the fifth field will be discussed later.

```
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
{ 8,  0, -1, "[ASSERT]",    "",    DoAssert },
{ 8,  0, -1, "[BINARY]",    "",    DoBinary },
{ 9,  0, -1, "[DECIMAL]",   "",    DoDecimal },
```

Now, the correction we want to make is that 4tH should allocate one word *less* in the Code Segment, since "[SEXTAL]" does not compile to anything. One less means "-1". We can now change the table accordingly:

```
{ 8,  1, -2, "VARIABLE",    "",    DoVariable },
{ 8,  0, -1, "[ASSERT]",    "",    DoAssert },
{ 8,  0, -1, "[BINARY]",    "",    DoBinary },
{ 8,  0, -1, "[SEXTAL]",    "",    DoSextal },
{ 9,  0, -1, "[DECIMAL]",   "",    DoDecimal },
```

That's all. We'll give to a few more examples.

E.g. 'VARIABLE' does not compile to anything either; it just reserves space in the Variable Area. But 'VARIABLE' always comes with a name, which doesn't compile to anything either. So we should decrease the the number of words in the Code Segment by two!

'CONSTANT' not only requires a name, but consumes a previously compiled literal as well. *Initially* this literal allocates space in the Code Segment, but it is gone after 'CONSTANT' has been compiled. So we decrease the number of words in the Code Segment by three!

'VALUE' is even more complicated. You can write something like:

```
10 value ten
```

But this will compile to:

```
[0]   literal     (10)
[0]   to          (0)
```

'VALUE' does *not* consume the previously compiled literal! But the name does *not* compile to anything. 'VALUE' takes a value from the Data Stack at run time, while 'CONSTANT', 'STRING' and 'ARRAY' take a previously compiled literal at compile time. If you don't believe us, check the glossary.

'VALUE' itself compiles to something, the literal is undisturbed, only the name vanishes. That means only one word less, thus "-1".

19.15 Adding inline macros

We've already seen how we can add new words to 4tH. We add a token and write the runtime. But this approach has a few disadvantages. First, the number of tokens is limited. You can use them, but once you run out of them, that's it. Second, writing a runtime is a little complex if you only have limited knowledge of C.

You can add as many inline macros as you want. From a user point of view there is not much difference between an ordinary 4tH word and an inline macro. The word is recognized by the compiler and works as expected.

Inline macros are simply sequences of existing tokens. As we've seen before, 'NIP' is implemented as an inline macro, so this source:

```
1 2 nip drop
```

Will compile to:

```
[0]   literal     (1)
[1]   literal     (2)
[2]   swap        (0)
[3]   drop        (0)
[4]   drop        (0)
```

There are disadvantages to inline macros as well. Every time you use 'NIP' it will expand to two words, so your Hcode will become a little bigger. We recommend to limit inline macros to three words. Second, an implementation using inline macros will make the compiler less compact compared to an implementation using tokens.

On the other hand, you only need to change the compiler when you use an inline macro. No changes to the interpreter or the decompiler will be necessary. Existing HX files will still run, although some 4tH sources will need modification.

Let's go to business. How can we implement an inline macro. Let's take 'NIP'. First we have to make an entry in the `ImmedList[]` table.

```
{ 2,  1, -1, "TO",          "",    DoValue },
{ 2,  1, -1, "IS",          "",    DoIs },
{ 2,  1, -1, "AS",          "",    DoValue },
{ 2,  0,  0, "IF",          "",    DoIf },
{ 2,  0,  0, "DO",          "",    DoDo },
{ 2,  0,  0, "->",          "",    DoDummy },
{ 3,  0,  1, "2R>",         "",    DoTwoRGet },
{ 3,  0,  1, "2>R",         "",    DoTwoRPut },
{ 3,  0,  3, "2R@",         "",    DoTwoRCopy },
```

Since 'NIP' is defined by:

```
: nip swap drop ;
```

It will compile to:

```
swap  (0)
drop  (0)
```

Which is one token *more* than the parser would expect. So the value of the fourth field is "1". But this will only reserve space for 'NIP'. The word won't be recognized by the compiler yet. In order to do that we have to make a word that compiles the tokens for 'NIP'. You can only do that with an "immediate" word:

```
#ifndef ARCHAIC
      static void DoNip (void)
#else
      static void DoNip ()
#endif

{
      CompileWord (SWAP, 0L);
      CompileWord (DROP, 0L);
}
```

This will compile 'SWAP' and 'DROP' into the compilant. Now we have to make an entry in `ImmedList[]` to link this function to the name "NIP":

```
{ 2,  1, -1, "TO",          "",    DoValue },
{ 2,  1, -1, "IS",          "",    DoIs },
{ 2,  1, -1, "AS",          "",    DoValue },
{ 2,  0,  0, "IF",          "",    DoIf },
{ 2,  0,  0, "DO",          "",    DoDo },
{ 2,  0,  0, "->",          "",    DoDummy },
{ 3,  0,  1, "2R>",         "",    DoTwoRGet },
{ 3,  0,  1, "2>R",         "",    DoTwoRPut },
{ 3,  0,  3, "2R@",         "",    DoTwoRCopy },
{ 3,  0,  1, "NIP",         "",    DoNip },
```

That's all! Since 'NIP' uses existing tokens, the compiler can handle it all by itself. There is no need to write runtime code or define tokens. All the burden is put on the compiler.

19.16 Adding string words

We've already seen that some words in 4tH have a name attached to them, like 'STRING', 'CREATE' or 'VARIABLE'. Since all these names are delimited by whitespace (like any other 4tH word), there is no need for special code.

Some words have special strings attached to them, like '."', '(' or '\'. These string are not delimited by whitespace, so they need special treatment. It's even more complex: each word has a different delimiter. '."' is delimited by '"', '(' is delimited by ')', and '\' is delimited by an end-of-line marker.

In this section we're going to explain how we added '."', since it's the most complex string word. Other string words like '(' are handled by the compiler only.

The first step to adding a string word is letting the compiler know, what delimiter is used. We do that by modifying `ImmedList[]`:

```
{ 2,  0, -2, "#!",          EOL,   DoComment },
{ 2,  0, -1, ",\"",         "\"",  DoCommaQuote },
```

```
{ 2,  0, -1, ",|",          "|",   DoCommaQuote },
{ 2,  0, -1, ".\"",         "\"",  DoDotQuote },
{ 2,  0, -1, ".(",          ")",   DoDotQuote },
{ 2,  0,  1, ">=",          "",    DoGreaterEqual },
{ 2,  0,  1, "<=",          "",    DoLessEqual },
{ 2,  0, -1, "S\"",         "\"",  DoSQuote },
{ 2,  0, -1, "S|",          "|",   DoSQuote },
```

The fifth field tells the parser whether this word requires a special delimiter. Yes, '."' is delimited by '"', so we enter a quote in the fifth field. If you enter an empty string, the parser assumes the word isn't a string word at all.

Now, what will the parser do when it encounters '."'? It will find an entry in `ImmedList[]` for '."'. It will see that this is a string word. Then it will make a call to `ParseString()` to find the delimiter and flag everything in between as a word. Which means that if you call `GetNextWord()`, you will get the entire string and not just the next word, e.g.:

```
: hello ." Hello world" cr ;
```

Will be parsed as:

```
GetNextWord ():   :
GetNextWord ():   hello
GetNextWord ():   ."
GetNextWord ():   Hello world
GetNextWord ():   cr
GetNextWord ():   ;
```

You will notice that there are some words that are delimited by whitespace, e.g. 'CHAR'. Why is that? Isn't every word delimited by whitespace? Yes, it is. But note that every word, which is parsed by `ParseText()` is also checked by `ParseDirectives()`. This expression would cause problems:

```
CHAR (
```

After 'CHAR' is parsed by `ParseText()`, '(' follows and is recognized by `ParseDirectives()` to be the start of a comment. To prevent this, we let the string following 'CHAR' be parsed by `ParseString()`. By the time `ParseText()` regains control, the character following 'CHAR' is already parsed and can cause no more problems. In short, if an expression like:

word (

or

*word *

is valid, let it be parsed by `ParseString()` by making an entry in the delimiter field of the `ImmedList[]` table. If not, don't.

Next, we have to develop a word that compiles '."'. Now, how can we compile '."'? First, we have to get the string and move it to the String Segment. We can do that by calling

GetNextWord() manually, but then we have to check for NULL-pointers. It is much easier to call DecodeWord() which sets the ErrNo member automatically when an error occurs.

DecodeWord() takes one argument, which is the error code it should set the ErrNo member to. It returns TRUE if GetNextWord() was called successfully. "CurrentWord" now points to the string after '."'.

Then we have to move the string to the String Segment. MoveString() does just that. It expects "CurrentWord" to point to the string that has to be moved. It returns a number. We'll need that when we design the runtime code.

There is no token or combination of tokens for printing strings. So this one will need a token of its own. We'll call it "PRINT" for the time being. Now, we got all components.

- The string can be parsed
- We can move it to the String Segment
- We can compile a token and an argument

This is the code for DoDotQuote():

```
#ifndef ARCHAIC
  static void DoDotQuote (void)
#else
  static void DoDotQuote ()
#endif
{
  CompileString (PRINT);
}
```

We aren't done yet. We still have to link the string '."' to this routine by adding an entry to ImmedList[]:

```
{ 2,  0, -1, ",\"",         "\"",  DoCommaQuote },
{ 2,  0, -1, ",|",          "|",   DoCommaQuote },
{ 2,  0, -1, ".\"",         "\"",  DoDotQuote },
{ 2,  0, -1, ".(",          ")",   DoDotQuote },
```

Now we can save comp_4th() and get on with the next file. Remember, we still got to add a token. As you will probably know, we do that in "cmds_4th.h":

```
#define NOOP       0
#define CELLD      0
#define EXECUTE    1
#define CR         2
#define SPACES     3
#define EMIT       4
#define PRINT      5
#define DOT        6
#define FETCH      7
```

And of course, we have to add a name to "name_4th.c", so it can be decompiled properly:

```
char *name_4th [] = {
  ",",       "execute",   "cr",       "spaces",   "emit",    ".\"",
```

Are we done yet? Not by a long shot. We have created a word with an argument, which is the offset of the string in the String Segment. That requires some special techniques. But we'll go into that in the next section.

19.17 Adding words with arguments

The very first thing you have to do is to make sure that your code can be saved and loaded again. Words that only consist of a token are saved without the argument. That reduces the size of the HX file. If you want to save the argument you have to add a line to both `load_4th()` and `save_4th()`:

```
case (LITERAL):
case (PRINT):
case (BRANCH):
case (BRANCH0):
```

Now we have to add code to `exec_4th()` in order to execute '."'. The first problem we encounter is: how do we access the argument? Accessing an argument is quite an expression:

```
Object->CodeSeg [Object->ErrLine].Value
```

In which:

OBJECT = Hcode pointer

CODESEG = Member of Hcode, pointing to the Code Segment

ERRLINE = Member of Hcode, pointing to the current word

VALUE = Member of word, holding the argument

In plain English it means: give me the argument of the currently executed word in the Code Segment. But we can also make our lives a lot easier by using this macro:

```
OPERAND
```

But this is only half the problem. How can we access the String Segment where the string-constant is stored? We made a pretty table on that subject.

DATATYPE	EXPRESSION	TYPE
String	Object->StringSeg [*{cell}*]	char
String	Object->StringSeg + *{cell}*	char*
Character	Object->UnitSeg [*{cell}*]	char
Character	Object->UnitSeg + *{cell}*	char*
Variable	Vars [Object->Offset + *{cell}*]	cell
Code	Object->CodeSeg [*{cell}*]	dict

Table 19.7: Accessing 4tH data from C

Most of the time it is more convenient to use functions to access those segments instead of addressing them directly:

AREA	FETCH	STORE
Data Stack	DPOP	DPUSH (value)
Return Stack	RPOP	RPUSH (value)
Character Segment	fetch (location)	store (location, char)

Table 19.8: exec_4th() data access API

You might consider using other functions too if certain datatypes are accessed more frequently.

Back to '."'. We have to access the String Segment for this one. Since all output is channelled through `emit()`, we have to convert the string to "units", which are unsigned characters. We could use the expression "Object->StringSeg [{arg}]", but that would be slower than pointer access on most systems. We decide to use "p", which is a temporary string-pointer:

```
case (PRINT):   for (p = Object->StringSeg + (unsigned) OPERAND; *p; p++)
                     emit ((unit) *p);
                break;
```

We assign "p" to a pointer to the string `(Object->StringSeg + {cell})`. We check for null-characters `(*p)`. If it is not a null-character, we 'EMIT' it `(emit (*p))` and advance the pointer `(p++)` before entering the loop again.

Now we are done. Let's do something more complicated now, like adding conditionals.

19.18 Adding conditionals

Basically, there are nine branch-instructions in 4tH:

1. **BRANCH**, which unconditionally branches to an address in the Code Segment.
2. **0BRANCH**, which branches to an address if the Code Segment if the top of the Data Stack is zero.
3. **CALL**, which unconditionally branches to an address in the Code Segment, throwing its origin on the Return Stack.
4. **EXIT**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Return Stack.
5. **VECTOR**, which unconditionally branches to an address in the Code Segment, which is taken from the contents of a variable, throwing its origin on the Return Stack.
6. **EXECUTE**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Data Stack, throwing its origin on the Return Stack.

7. **CATCH**, which unconditionally branches to an address in the Code Segment, which is taken from the top of the Data Stack, throwing the data stack pointer, the previous handler and its origin on the Return Stack.
8. **LOOP**, which branches to an address in the Code Segment if the top of the Return Stack is less than value below it.
9. **+LOOP**, which branches to an address in the Code Segment if the top of the Return Stack plus the top of the Data Stack is not equal to the value below the top of the Return Stack.
10. **?DO**, which branches to an address in the Code Segment if the top of the Return Stack plus the top of the Data Stack is equal to the value below the top of the Return Stack.

The first four are the most common and the most useful ones. Together, they control your entire program. But how do they know where to branch to? There is no instruction like 'BRANCH'. And where is 'DO'?

Well, 'DO' doesn't do any branching. It just puts the loop parameters on the Return stack. And as for 'BRANCH', this is how it works:

> IF something ELSE other thing THEN

This is an expression we are very familiar with. We pronounce it as:

> "If TOS is non-zero then something is executed."

This is not entirely true. In fact, it is:

> "If TOS is zero then branch after 'ELSE'"

Which in effect results in the execution of "something". But when "something" has executed, it has to branch after the "other thing". Unconditionally, that is. 'THEN' does nothing, except serve as a marker for the branch. It doesn't have to compile to anything.

So this little piece to code will compile to:

```
[0]   0branch     (2)
[1]   ...
[2]   branch      (3)
[3]   ...
[4]   ...
```

You see that 'IF' compiles to a '0BRANCH' instruction, 'ELSE' to a 'BRANCH' instruction and 'THEN' to nothing! If you have a closer look you might assume that 4tH will branch to instruction [2] and then branch directly to instruction [3]. This is not quite what was intended.

What a 'BRANCH' instruction actually does is setting the instruction counter to a specific value. Then, like after every instruction, the instruction counter is incremented. Why make exceptions? That only slows the interpreter down. Let's take a look at this piece of code:

```
10 dup if 1+ else . then cr
```

This will compile into:

```
[0]   literal     (10)
[1]   dup         (0)
[2]   0branch     (4)
[3]   1+          (0)
[4]   branch      (5)
[5]   .           (0)
[6]   cr          (0)
```

Now how does this execute. We will show you by giving the value of the instruction pointer before execution, after execution and after the automatic increment.

INSTRUCTION#	INSTRUCTION	BEFORE	AFTER	INCREMENT
[0]	literal	[0]	[0]	[1]
[1]	dup	[1]	[1]	[2]
[2]	0branch	[2]	[2]	[3]
[3]	1+	[3]	[3]	[4]
[4]	branch	[4]	[5]	[6]
[6]	cr	[6]	[6]	[7]

Table 19.9: Example execution plan

You see that ’0BRANCH’ has no effect when the top of the Data Stack is non-zero. And while ’BRANCH’ sets the instruction pointer to "5", it will resume execution at location "6".

If you compile this little piece of code by hand and start compiling from the beginning, you will also see that you can’t fill in the destination until it has been compiled. So how does 4tH do that?

4tH has a small stack (Control Stack) where it stores these addresses. So when it encounters an ’IF’ or ’ELSE’ or ’THEN’ instruction, it stores its current address there. What would have happened during the compile of the previous program:

- ’IF’ is encountered. It compiles a ’0BRANCH’ instruction and stores ’2’ on the Control Stack.
- ’ELSE’ is encountered. It compiles a ’BRANCH’ instruction, takes ’2’ from the Control Stack and changes the argument of the ’0BRANCH’ word to its own address, which is ’4’. It stores ’4’ on the Control Stack again.
- ’THEN’ is encountered. It takes ’4’ from the Control Stack and changes the argument of the ’BRANCH’ word to address of the last compiled word, which is ’5’.

But this example was correct. What would have happened if we had written something like:

```
5 0 : test begin 1 dup if ; while . then dup dup repeat
```

To prevent the compiler from accepting these kind of constructions, a reference is added. This reference tells the compiler what conditional was put on the stack. If the reference isn't correct, the compiler will throw an exception. There are five predefined references, but you may add your own.

There are three functions which handle conditionals:

FUNCTION	DESCRIPTION
MarkLink	Throws an address on the stack
PairLink	Gets an address from the stack
MakeLink	Makes a "backlink"

Table 19.10: Branch resolving API

They all take a reference as argument. All address calculation and errorchecking is done by these functions. Let's get to business and retrace our steps when we added 'BEGIN.. WHILE.. REPEAT'.

All conditionals are "immediate" words. So they have to be added to `ImmedList[]`. That also means, that each word has its own function. Let's design the one for 'BEGIN'. 'BEGIN' is just a marker, which means we have little more to do than to save the address on the stack:

```
#ifndef ARCHAIC
      static void DoBegin (void)
#else
      static void DoBegin ()
#endif

{
      MarkLink (R_BEGIN);
}
```

Yes, that's all. Just call `MakeLink()` with the proper reference! Just make sure, you've compiled everything you wanted to compile. Jumps resolved by `MarkLink()` will in effect always continue from the word that will be compiled next, although it *seems* they jump to the word compiled last.

Now we have to make 'WHILE'. 'WHILE' executes a piece of code when the top of the DataStack is non-zero. Which means it jumps to 'REPEAT' when the top of the DataStack is zero. Sounds like a '0BRANCH' instruction to us. Note, that 'BEGIN' doesn't play any part whatsoever here!

Because the address it has to jump to isn't known yet (we haven't encountered 'REPEAT', we can only compile the '0BRANCH' instruction with an arbitrary address. But 'REPEAT' will have to know the address in order to make a backlink, so we have to throw it on the stack:

```
#ifndef ARCHAIC
      static void DoWhile (void)
#else
      static void DoWhile ()
#endif

{
  CompileMark (BRANCH0, R_WHILE);
}
```

Note that `CompileMark()` is equivalent to:

```
CompileWord (BRANCH0, 0L);
MarkLink (R_WHILE);
```

Now we come to 'REPEAT'. It has a lot of things to do. First, it has to compile a 'BRANCH' instruction in order to get back to 'BEGIN'. Second, it will have to resolve the backlink from 'WHILE'.

In a way, 'REPEAT' has an advantage over 'WHILE'. It doesn't have to compile an arbitrary address, since it is already on the control stack. It has been provided by 'BEGIN'. It can retrieve that address by calling `PairLink()` with the proper reference:

```
CompileWord (BRANCH, PairLink (R_BEGIN));
```

But there is a problem. `MakeLink()` should always make a link to the last compiled word. And we can't compile the 'BRANCH' instruction first, because of the "WHILE" reference on the top of the controlstack!

So we have to resolve the 'WHILE' backlink first. For that purpose, `MakeLink()` has an extra argument. Usually, `MakeLink()` is called with "LASTW", which means it will jump to the word compiled last. Then the instruction counter will be incremented and the interpreter will continue from there.

In order to compile 'REPEAT', we have to make a backlink that points to the word compiled *next*. So, this statement is inserted before `CompileWord()`:

```
MakeLink (R_WHILE, NEXTW);
CompileWord (BRANCH, PairLink (R_BEGIN));
```

And what if we had made a "BEGIN..AGAIN" or a "BEGIN..UNTIL" loop? Well, in any case we would have to branch back to 'BEGIN'. 'UNTIL' conditionally and 'AGAIN' unconditionally. The address of 'BEGIN' would have already been on the control stack, so a single statement could have taken care of it:

```
CompileWord (0BRANCH, PairLink (R_BEGIN));
CompileWord (BRANCH, PairLink (R_BEGIN));
```

Actually, since 4tH supports multiple 'WHILE's the problem is a little more complex. 'REPEAT' must resolve all 'WHILE's on the control stack before it can even think of compiling a 'BRANCH' instruction. We've already seen that the only difference between an 'AGAIN' and an 'UNTIL' is the branch instruction which is compiled. So in 4tH 'REPEAT', 'UNTIL' and 'AGAIN' are handled in a similar way:

```
#ifndef ARCHAIC
  static void CompileAgain (unit AgainToken)
#else
  static void CompileAgain (AgainToken) unit AgainToken;
#endif
{
  while ((ToCS > 0) && (ControlStack [ToCS - 1].Mark == R_WHILE))
        MakeLink (R_WHILE, NEXTW);
  CompileWord (AgainToken, PairLink (R_BEGIN));
}
```

As long as the control stack is not empty and there is a 'WHILE' reference on top of the control stack, backlinks are made. Finally, the branch instruction is compiled, which jumps back to 'BEGIN'. 'REPEAT' can now be reduced to:

```
CompileAgain (BRANCH);
```

Note that a colon definition also uses the control stack. This reference is resolved by ';', which compiles 'EXIT' and creates a backlink. The 'BRANCH' instruction will prevent the interpreter from entering the definition. At the same time, ':' creates a symbol. We'll go into that in the next section.

If you want to create your own branch instructions, you'll have to define their behaviour in `exec_4th()`. If the argument of the token contains the address you want to jump to in the end, you'll have to define it like this:

```
JUMP (OPERAND);
```

That is pretty easy. This macro changes the Program Counter, which is part of the Hcode header:

```
Object->ErrLine
```

Of course, we've defined a macro for that:

```
PROGCOUNT
```

If the address you want to jump to is issued by the user, you probably want to check whether it is a valid execution token. Just use the macro `XT()`:

```
DSIZE (1);
a = DPOP;
XT (a);
JUMP (a);
```

Consequently, leaving the current Program Counter value on the return stack is pretty easy too:

```
RFREE (1);
RPUSH (PROGCOUNT);
```

I think that covers it all, don't you?

19.19 Extending the I/O subsystem

The 4tH I/O system is entirely built upon the buffered C streams[4] concept. That means every device that can be assigned to a `FILE*` and accessed through `fgetc()` and `fputc()` can be integrated into the 4tH I/O system. If can open a device with `fopen()` and close it with `fclose()`, you're done, it's already supported. If not, you have to design two functions that take the same parameters and return the same values as `fopen()` and `fclose()`. If you can't you can probably still use those devices within 4tH, but you can't integrate them into the I/O system.

If you have defined these functions, you'll have to make changes to `OpenStream()` in `exec_4th()`. 'Mode' is the sum of all file access methods, e.g.

[4] See: http://www.aquaphoenix.com/ref/gnu_c_library/libc_118.html

```
s" ls" input pipe + open
```

This definition contains two file access methods, INPUT and PIPE. You can find these values in cmds_4th.h. INPUT equals 1 and PIPE equals 8. That makes 9 and that is what ends up in the 'Mode' parameter of `OpenStream()`. However, if one would allow all possible combinations of all file access methods some would surely make little sense. That is why 'Mode' is filtered by `Mapping[]`. You will find that element 9 of `Mapping[]`[5] contains the value 5. That number corresponds to element 5 in `Modelist[]`, which lists the correct file access method (which is still 9, of course) and the mode parameter for `fopen()`. `OpenStream()` continues by initializing the members of the `Stream[]` structure.

MEMBER	FUNCTION
Mode	Uniform file access method
Device	FILE pointer to opened device
Connect	Function pointer to `fopen()` like function
Disconnect	Function pointer to `fclose()` like function

Table 19.11: Members of Stream[] structure

When this is done, it uses the `Connect()` member to open the device. After that, everything is completely transparent to the programmer.

If you want to add a new device, you probably want to signal which type of device you're using. In order to do that, you must first add a `#define` to the 'file modes' section in cmds_4th.h. Each new file access method has exactly twice the value of the previous one, which means that the first one you define would have to be 16. Then you have to figure out which modes are actually supported. Can your device be opened in read-write mode? Does appending make sense? You add those 'ideal' states to `ModeList[]`. Then map all possible combinations of all file access methods to the 'ideal' states in `ModeList[]` using `Mapping[]` conversion table. Every additional file access method *doubles* the size of `Mapping[]`, so beware!

MACRO	FUNCTION	THROW
`DEV(n)`	Aborts if *n* is not a device	`M4BADDEV`
`UDEV(n)`	Aborts if *n* is not opened by the user	`M4BADDEV`
`ODEV(n)`	Aborts if *n* is not opened	`M4IOERR`
`SDEV(n)`	Aborts if *n* is a pipe	`M4BADDEV`

Table 19.12: Device status macros

If you want to check a device in `exec_4th()`, there are several macros you can use. They all take the value returned by `OpenStream()` as parameter.

19.20 Using the symbol table

The symboltable is a way to dynamically add words to the vocabulary of 4tH. All other words are hard-coded into the compiler. If you want to add any, you have change the entire compiler and make a new executable. We've seen that before.

[5] We start counting at 0.

Without the symboltable there wouldn't be any strings, tables, variables or even colon-definitions. May be you think that such a powerful feature must be hard to work with. No, it isn't!

The only thing you have to tell the symboltable is "hey, if that word comes along, compile this token and this argument into the object". That's all. There are three functions that control the symbol-table:

FUNCTION	DESCRIPTION
AddSymbol()	Adds a symbol to the symboltable
MakeSymbol()	Makes a symbol of the current word
GetSymbol()	Searches the symboltable
SearchDictionary()	Searches the entire dictionary

Table 19.13: Symboltable API

You can forget about the `GetSymbol()`. You will hardly ever need it. Let's see how it works. We'll continue with ':':

```
if (DecodeWord (M4NODECL))
   {
     AddSymbol (CALL, Object->ErrLine, CurrentWord);
     CompileMark (BRANCH, R_COLON);
   }
```

First it uses `DecodeWord()` to set "CurrentWord" to the next word in the source, which is the name of the definition. Then it adds a symbol to the symboltable. Hey, shouldn't we compile a word first?

No. The member "ErrLine" of an Hcode header always points to the next available word in the Code Segment. That is the place where we will compile our 'BRANCH' instruction. When the instruction pointer is set to that location, it will be automatically incremented and get *inside* the definition. So that is okay.

The token we'll use to branch inside that definition is not 'BRANCH' or '0BRANCH', but 'CALL'. 'CALL' throws the address of its own location on the Return Stack. When the definition is done, 'EXIT' takes that address off the Return Stack and jumps backs.

`AddSymbol()` adds an entry to the symboltable. No, you don't need to check the symboltable when you add a symbol. `AddSymbol()` does that for you and sets the member "ErrNo" when needed.

Finally, a 'BRANCH' token is compiled with a dummy argument. It will be solved later with a backlink, marked by `MarkLink()`. Now, how does it actually work? We'll give you an example. Take this small program:

```
: hello's 0 do ." Hello " loop cr ;
20 hello's 10 hello's
```

When the ':' is reached by the compiler, it hasn't compiled a thing, so "ErrLine" still points to the first word in the Code Segment (0). `DecodeWord()` is called, so "CurrentWord" points to "HELLO'S". Then a symbol is added to the symboltable by calling `AddSymbol()`. The entry looks like this:

HELLO’S -> CALL (0)

That means that every time the name "HELLO’S" is found in the source, the word "CALL (0)" will be compiled. See for yourself:

```
[0]   branch      (6)
[1]   literal     (0)
[2]   do          (0)
[3]   ."          (0)
[4]   loop        (2)
[5]   cr          (0)
[6]   exit        (0)
[7]   literal     (20)
[8]   call        (0)
[9]   literal     (10)
[10]  call        (0)
```

Basically, that is all you need to know about the symboltable. Yes, you can search it yourself, but why should you? It is done automatically for you. But if you really want to know: you do it by calling `GetSymbol()`.

All you need is the name of the symbol you’re looking for and what you want the compiler to do when i finds it. E.g. if you were looking for "HELLO’S" and didn’t want to compile it, you’d have to write:

```
int x = GetSymbol ("HELLO\'S", W_SEARCH);
```

`GetSymbol()` returns the index of "HELLO’S" in the symboltable. You can use this index to access the symboltable, called "SymTable":

```
printf ("%d, %ld, %s\n", (int) SymTable [x].Token,
       (long) SymTable [x].Value, SymTable [x].Name);
```

Which prints the token, the argument and the name of the symbol. If the name isn’t listed `GetSymbol()` returns "MISSING":

```
if ((x = GetSymbol ("HELLO\'S", W_SEARCH)) == MISSING)
     printf ("Not found\n");
```

If you search the symboltable in order to compile a word, you only have to tell:

```
int x = GetSymbol ("HELLO\'S", W_EXEC);
```

This will not only return the index, but compile the word as well. Note that this function can only search the symboltable. It cannot look for other words. These words have their own function, but basically work the same:

CLASS	FUNCTION
Immediate words	GetImmediate()
Simple words	GetWord()
Constants	GetConstant()

Table 19.14: Table search API

They will return an index as well, but that will be of little use since the tables they search are private and cannot be accessed outside the function. `SearchDictionary()` combines all these functions (including `GetSymbol()`) but will only return a boolean to indicate that that the word was found. It is the most common way to access these lower level functions.

If you decide to add your own words that use the symboltable, you have to make an entry in `ImmedList[]`. Let's say you want to add a word, which defines a floating point number, e.g.:

```
float fp_number
```

Now we have to let 4tH know that for each "FLOAT" a symboltable entry has to be reserved:

```
{ 5,  1,  -2, "FLOAT",       "",    DoFloat },
```

Yes, that is where that famous second field is for! It tells 4tH how many symboltable entries it has to reserve for a specific immediate word.

19.21 Using variables and datatypes

We're slowly entering the area where extensions are becoming projects on its own. You should be able to make the most common extensions yourself now. What we have ahead is just for the interested reader of someone who want to add a completely new wordset.

We're going to explain you how 4tH handles strings and other datatypes. Variables (any variable!) are not created during compilation. That means no space is reserved. 4tH only monitors how much space has been allocated to each datatype. This information is saved in the header.

At the moment there are only two basic datatypes: characters (Character Segment) and 32 bit signed integers (Integer Segment). You'll find the size of these segments in the Hcode members "Variables" and "Strings". The Character Segment and Integer Segment are created when a Hcode program is executed and discarded when the Hcode program is terminated.

So the only thing the compiler has to do is keep track of the sizes of the segments and assign pointers to variables. This is quite easy. When an Hcode header is initialized by `InitObject()`, both "Variables" and "Strings" are set to zero. Then it parses this declaration:

```
variable one
```

As a consequence, `DoVariable()` is called:

```
if (DecodeWord (M4NODECL))
    AddSymbol (VARIABLE, Object->Variables++, CurrentWord);
```

It calls `DecodeWord()`, so "CurrentWord" now points to "ONE". "Variables" is still zero. If `DecodeWord()` was called successfully, it just adds a symbol by calling `AddSymbol()`, which looks like this:

ONE -> VARIABLE (0)

After that "Variables" is incremented, so it now holds the value "1". It doesn't matter, what comes next: "ONE" will always compile to "VARIABLE(0)". The next variable will compile to "VARIABLE(1)". Unless it is an array:

```
10 array list
```

The "10" is compiled as a literal. Then the compiler encounters "ARRAY", so `DoArray()` is called:

```
cell val = DecodeSymbol ();

if (! Object->ErrNo) {
      AddSymbol (VARIABLE, (cell) Object->Variables, CurrentWord);
      Object->Variables += (unsigned) val;
}
```

First it calls `DecodeSymbol()`, which does two things:

1. It calls `DecodeWord()`, so "CurrentWord" now points to "LIST".
2. It removes the previously compiled literal (by decrementing the member "ErrLine") and returns it.

Now "val" holds the value "10". If no error occurred, `DoArray()` will call `AddSymbol()`. There an entry is created that looks like this:

LIST -> VARIABLE (1)

So every time the name "LIST" is encountered a 'VARIABLE' token will be compiled with argument "1". Finally, the number of variables is incremented by "val", so the member "Variables" now holds "11". This means that 10 variables have been added, which is correct.

It works about the same for 'STRING', only we compile a literal value here. Why? Because the system areas in the Character Segment are fixed. In the Variable Area there are also the application variables and 4tH cannot know at compile time how many there will be at runtime.

We could have placed the variables right after the system variables, but that would have made it much more difficult to add names to your application variables. But now we have to resolve what 'VARIABLE' has to do at runtime. So we have to edit `exec_4th()`.

Well, the only thing it has to do is calculate its address in the Variable Area. There is a member in the header that holds the offset of the user variables inside the Variable Area. The only thing we have to do is to add the operand to it and push the result:

```
DPUSH (Object->Offset + OPERAND);
```

Since the next word takes the address of the Data Stack there is no real difference with a literal. The changes you want to use the address of a variable as a literal expression are quite remote.

There are two macros that check the status of a variable. `VAR(n)` checks whether *n* is a variable at all. `UVAR (n)` checks whether *n* is a writable variable. When any of these macros fail, `M4BADVAR` will be thrown. *n* is the value that VARIABLE leaves on the stack.

Of course, if you want to add an entirely new datatype, you have lots of work to do, but you can use the same tools as we have used. We have to stress that you use a separate segment for each datatype. That keeps 4tH simple and it won't take more memory than other implementations.

Note that if you want to create constants for a certain datatype you have to work out a scheme to load and save them. If this scheme depends on a certain, non-portable encoding, you won't be able to use the resulting .HX files on different platforms.

19.22 Other tools

We have known assertions since version 3.1c and conditional compilation since version 3.1d. Both conditionally skip source between to markers. And they can be nested. That sounds like quite a challenge, but it isn't. In fact, there is only one simple routine that handles it.

If we encounter a situation where source has to be skipped, we just call `SkipSource()`. In case of conditional compilation, the source that we have to skip is between the markers '[IF]' and '[THEN]'.

First, we call `DecodeLiteral()`. This function gets the argument part of the previously compiled literal and removes that literal from the compilant (actually, the member "ErrLine" is decremented, so it will be overwritten):

```
cell val = DecodeLiteral ();
if (! Object->ErrNo) if (! val) SkipSource ("[IF]", "[THEN]");
```

Then `SkipSource()` is called with the argument "[IF]" and "[THEN]". It will handle everything, including any nested markers. Note that "CurrentWord" still has to point to the "[IF]" that triggered the action.

19.23 Patching 4tH

We're getting at the end of the story here. There is one topic left we want to discuss with you.

It is a drag when you have made some nice extensions to 4tH and you have to reapply them each time a new version of 4tH is released. However, there is a solution. 4tH comes with a program called `patch4th.4th` which can help you. The only thing you have to do is to create a 4tH patch file. It consists of six parts, which *all have to appear in the order presented* to you here. If a section is not applicable, leave it blank.

19.23.1 Tokens

The first part are the tokens or the instructions of the virtual machine, if you prefer. Every entry consists of three fields, delimited by a tilde[6]:

1. The first field is the C constant of the token, as it appears in `cmds_4th.h` (*see* 19.4);

[6] A tilde is rarely used by 4tH, so that seemed a good choice. If you prefer another delimiter, you have to change the source of `patch4th.4th`.

2. The second field indicates whether the token needs a parameter (*see* 19.17);

3. The third field is the mnemonic, as used in `name_4th.c` (*see* 19.6).

So a sample entry might look like:

```
[tokens]
NIP~no~"nip"
```

To terminate this section, add an empty line.

19.23.2 Words

The second part are the words you actually use in a 4tH program. As you will know by now, a word can compile to zero or more tokens. Every entry consists of eight fields, delimited by a tilde:

1. The first field is the name of the word as you will use it in a 4tH program;

2. The second field is the token it will compile to;

3. The third field contains the type of word, *constant*, *immediate* or *word*;

4. The fourth field is the fixed parameter of a *constant*;

5. The fifth field is the number of symbol entries it will need;

6. The sixth field is the source correction that will be applied;

7. The seventh field is the delimiter it uses;

8. The eighth field is the C function which handles the *immediate* word.

A simple *word* requires fields 1, 2 and 3. A *constant* requires fields 1, 2, 3 and 4. An *immediate* word requires 1, 3, 5, 6, 7 and 8. If a field is not applicable for a certain type it will not matter what you enter there. See section 19.2 for more information. A sample entry might look like:

```
[words]
BIRTHDAY~LITERAL~constant~19600902L~0~0~~
BINARY~RADIX~constant~2L~0~0~~
NIP~NIP~word~~0~0~~
[SEXTAL]~~immediate~~0~-1~""~DoSextal
```

To terminate this section, add an empty line.

19.23.3 The virtual machine

These *sections are copied verbatim* - including idententation - into `exec_4th.c`. The first section are the additional `#include` directives you might need. These are located in the `[vm.include]` section:

```
[vm.include]
#include <sys/stat.h>
```

You terminate this section by directly continuing with the next section, `[vm.support]`. This contains any support functions for the virtual machine, e.g. `OpenStream()`. They will appear right before the main `exec_4th()` function:

```
[vm.include]
#include <sys/stat.h>
[vm.support]
/*
Custom support functions
*/
```

You terminate this section by directly continuing with the next section, `[vm.extension]`. This contains the actual C code which will be copied into the main loop of `exec_4th()`:

```
[vm.include]
#include <sys/stat.h>
[vm.support]
/*
Custom support functions
*/
[vm.extension]
    case (NIP): DSIZE (2);
                DS (2) = DS (1);
                DDROP;
                break;
```

You terminate this section by directly continuing with the next section.

19.23.4 Immediate words

This section contains the C functions that are executed when immediate words are compiled (*see* 19.2 and 19.7). They will be inserted verbatim just before `ImmedList[]`:

```
[vm.include]
#include <sys/stat.h>
[vm.support]
/*
Custom support functions
*/
[vm.extension]
    case (NIP): DSIZE (2);
                DS (2) = DS (1);
```

```
                DDROP;
                break;
[immediate.words]
#ifndef ARCHAIC
      static void DoSextal (void)
#else
      static void DoSextal ()
#endif

{
      Base = 6;
}
```

You don't have to explicitly terminate this section.

19.23.5 Applying the patches

Make a subdirectory, copy the original `cmds_4th.h`, `comp_4th.c`, `exec_4th.c`, `name_4th.c`, `save_4th.c` and `load_4th.c` sources into it and rename them to `.txt`. They will serve as templates for your custom 4tH sources. In this example we will assume your custom patchfile and the compiled `patch4th.4th` are also located there, but that is not required. When you make a directory listing you will see the following files:

```
cmds_4th.txt
comp_4th.txt
exec_4th.txt
name_4th.txt
save_4th.txt
load_4th.txt
mypatch.txt
patch4th.hx
```

Now run it:

```
4th lxq patch4th.hx mypatch.txt
```

When everything is alright, you will see the following messages:

```
Opening mypatch.txt
.. 1 tokens read
.. 4 words read
Processing cmds_4th.txt
.. done
Processing save_4th.txt
.. done
Processing load_4th.txt
.. done
Processing name_4th.txt
.. done
Processing exec_4th.txt
.. done
```

```
Processing comp_4th.txt
.. done
Closing mypatch.txt
.. done
```

When you list the directory again, you will see that new `cmds_4th.h`, `comp_4th.c`, `exec_4th.c`, `name_4th.c`, `save_4th.c` and `load_4th.c` sources have been created.

19.23.6 Error messages

Usage: patch4th patch-file	Issue a patchfile on the commandline
Bad boolean	"yes" or "no" was expected in this field
Bad datatype	"word", "constant" or "immediate" was expected in this field
Bad number	A number was expected in this field
Cannot find [tokens]	A `[tokens]` section was expected in the patchfile
Cannot find [words]	A `[words]` section was expected in the patchfile
Cannot find [vm.include]	A `[vm.include]` section was expected in the patchfile
Too many tokens	Too many tokens were defined in the patchfile
Cannot find /* ranges */	Corrupted `cmds_4th.txt` file
Cannot find NOOP token	Corrupted `cmds_4th.txt` file
Cannot find LastWord4th	Corrupted `cmds_4th.txt` file
Cannot open <file>.h|c	Could not create a source file
Cannot open <file>.txt	Could not find a template file

DOCUMENT ENDS HERE