ENIGMA

HISTORICAL ENCRYPTION FOR CONTEXT, PLAINTEX AND LATEX DOCUMENTS

Philipp Gesang

Contact *phg42.2a@gmail.com*
License 2-clause bsd
Release date 2012-06-07 22:32:13+0200
Version hg tip

Part 1: Manual

Part 2: Implementation

1

Manual

INTRODUCTION

This module implements an *Enigma* cipher that is equivalent to the most widely distributed model: the Enigma I (subtypes M1 M2 and M3).[1] Machines of this type had three rotors, a non-moving reflector and, as a novelty at the time of their introduction, a plugboard. The simulation accepts arbitrary configurations of these components, as well as the starting position of the rotors, and then processes text accordingly. Depending on the input, this yields the plaintext or ciphertext, as encryption and decryption are the same.

The code is provided as a module (interface for ConTEXt) as well as a package (PlainTEX, LATEX). It is subject to the BSD license, see below, *page 11*, for details.

[1] See *http://www.cryptomuseum.com/crypto/enigma/i/index.htm* for a showcase.

USAGE

LOADING THE MODULE / PACKAGE

The intention is for the Enigma codebase to integrate with the three most popular (as of 2012) TEX formats: ConTEXt, PlainTEX, and LATEX. If the user interface does not fully conform with the common practice of the latter two, please be lenient toward the author whose intuitions are for the most part informed by ConTEXt. For this reason, a couple words concerning the interfaces will be necessary. The examples in this manual will be cycling through all three formats, but once you get the general idea of how it works, you will have no problem translating between coding styles. Those familiar with ConTEXt might, therefore, skip the following paragraphs and continue directly with the next section on *page 6*.

The package is loaded as usual. For PlainTEX, issue a `\input{enigma}`. LATEX-users need to place `\usepackage{enigma}` somewhere inside the preamble. (There are no package options.) From this point on, instructions for both formats are the same.

The interface provides two basic macros from which all functionality will be derived: **\defineenigma** and **\setupenigma**. Both are a kind of *meta-macros*, meaning that they generate other macros which may then be employed to access the functionality of the package. As such they naturally belong inside the preamble (if you chose to use Enigma with LATEX, that is). The correct order is to **\defineenigma** an enigma machine first and then **\setupenigma** it. The definition takes a single, setups a double mandatory argument. Thus, `\defineenigma{encrypt}` creates a new environment consisting of the macros **\beginencrypt** and **\endencrypt**.[2] These correspond to **\startencrypt**/**\stopencrypt** in the ConTEXt interface. The ConTEXt-examples below are easily translated to Plain/LATEX-syntax by replacing curly brackets (groups) with square brackets and substituting environment prefixes: `\start<foo>` becomes `\begin<foo>` and `\stop<foo>` becomes `\end<foo>`. Except for those conventions the syntax, even in key-value assignments, is identical.

[2] ConTEXt-users will have noticed that there is no direct macro `\encrypt{foo}`. The reason for this is that the callback which the module relies on operates on node-level. This means that for the Enigma encryption to have an effect it will have to process entire paragraphs. As encrypted passages are supposed to stand on their own, this small limitation should not have a severe impact on functionality. If you should, however, need a macro that works for smaller portions of text, please send a feature request to the maintainer (phg42.2a@gmail.com).

However, the environment is not usable right away, as we still have to set the initial state of the machine. This is achieved by the second meta-macro, `\setupenigma{encrypt}{<args>}`, where `<args>` is a placeholder for a list of *assignments*, i. e. pairs of *key=value* statements by means of which all further parameters are specified. The possible parameters are listed in the next section, examples of their effects will be given further below in the section on functionality (see *page 8*).[3]

OPTIONS EXPLAINED

At the moment, the **\setupenigma** macro in any format accepts the following parameters.

- *other_chars* <*boolean*> How to handle non-encodable characters, i. e. glyphs outside the bare set of Latin letters; see below on *page 7*.
- *day_key* <*string*> Encryption key, i. e. a description of the initial setup of an Enigma machine: the reflector used, the choice and positions of the three rotors, the ring settings, and the plugboard wiring.
- *rotor_setting* <*string*> The initial rotor advancement.
- *spacing* <*boolean*> Auto-space output?
- *verbose* <*integer*> Controls overall verbosity level (*global*!).

```
\input {enigma}
%% Definition ·········· %%
\defineenigma {encryption}
%% Setup ··············· %%
\setupenigma  {encryption} {
  other_chars = no,
  day_key = I II III
            01 01 01,
  rotor_setting = aaa,
  spacing = yes,
  verbose = 1,
}
%% Usage ··············· %%
\beginencryption
  aaaaa aaaaa aaaaa
  aaaaa aaaaa aaaaa
\endencryption
\beginencryption
  Nobody in Poland is going
  to be able to read this,
  har har!
\endencryption
\bye
```

Figure 1 Usage example for the PlainTEX format.

To state the obvious, the value of *day_key* serves as the *day key* for encryption. An Enigma day key ordinarily consists of (1) a list of the the rotor configuration, (2) the ring settings, and (3) the plugboard connections.[4] Together these have the denotation *day key*, because they are meant to be supplied in special code books by central authority, one for each day.[5] In the Enigma setup, the day key starts with a triple of

[3] If you grasp the concept of paired `\define<foo>` – `\setup<foo>` macros, then congratulations are in order: you qualify for migration from your current macro package to ConTEXt.

[4] For a description of the initialization process see *http://wtp.com/enigma/mewirg.htm*.

[5] Read about the historical directives for daily key renewal at *http://users.telenet.be/d.rijmenants/en/enigmaproc.htm*. there are some PDFs with images of *Kenngruppenbücher* at *http://cryptocellar.org/enigma/*, section *Enigma Messages and Keys*. Also, while you're at it, don't miss the explanation on Wikipedia: *http://en.wikipedia.org/wiki/Cryptanalysis_of_the_Enigma#Key_setting*.

Roman numerals (I to V) describing which of the five rotors is located in which of the three slots. (e. g. `I VI II`).[6] Its next part is the ring setting, a triple of two-digit integers that are the amount by which the internal wiring of each rotor has been shifted (`03 23 11`). As the Enigma encrypts only the letters of the Latin alphabet, sane values range from one (first position: no shift) to twenty six.[7] The third section specifies which pairs of letters are substituted by each other by means of plugboard connections (`NI CE GO LD ...`). There can be zero to thirteen of these redirections, thus the presence of this section is entirely optional. Also part of the *day_key*, but not mentioned yet, is the choice of the *reflector*. It may be specified as one of the three letters `A`, `B` and `C` as the first item. If no reflector is requested explicitly, the machine defaults to `B`, which is actually the only one of the three models that had been in widespread use (see below on *page 24* for the wirings).

Initialization is not complete without a *rotor_setting*. This is a triple of letters, each representing the initial state of one rotor (e. g. `fkd`). Historically this was not part of the day key but supposed to be chosen at random by the operating signal officer.

The output can be automatically grouped into sequences of five characters, delimited by spaces (option *spacing*). This does not only conform with traditional crypto-style, but also allows for the resulting text to be sanely broken into lines by TEX.

Most documents don't naturally adhere to the machine-imposed restriction to the 26 letters of the Latin alphabet. The original encipherment directives comprised substitution tables to compensate for a set of intrinsic peculiarities of the German language, like umlauts and common digraphs. The Enigma simulation module strives to apply these automatically but there is no guarantee of completeness.

However, the Enigma lacks means of handling languages other than German. When the substitution lookup fails, there are two ways of proceeding: either to ignore the current character or to pass it on to the output as if nothing happened. The default behaviour is to drop alien letters and move on. If the user intends to keep these foreign characters instead, E can achieve this by setting the *other_chars* key in the Enigma setup to the value *true*. An example of how the result of both methods may look, other things being equal, is given in below listing (example for ConTEXt; see the file `enigma-example-context.tex` in the `doc/` subtree of your installation path).

```
\usemodule [enigma]
\defineenigma [secretmessage]
\setupenigma  [secretmessage] [
  other_chars = yes,
  day_key = B V III II 12 03 01 GI JV KZ WM PU QY AD CN ET FL,
  rotor_setting = ben,
```

[6] For the individual wirings of the five rotors see *http://www.ellsbury.com/ultraenigmawirings.htm*, as well as the implementation below at *page 24*.

[7] Consult *http://en.wikipedia.org/wiki/Enigma_rotor_details#The_ring_setting* for an introduction into the ring mechanics.

```
]

\defineenigma [othermessage] [secretmessage]
\setupenigma  [othermessage] [other_chars=wrong]

\starttext

\startsecretmessage
  føo bąr baž
\stopsecretmessage
\startothermessage
  føo bąr baž
\stopothermessage

\stoptext
```

Both methods have their disadvantages: if the user chooses to have the unknown characters removed it might distort the decrypted text to becoming illegible. Far more serious, however, are the consequences of keeping them. As artefacts in the ciphertext they would convey information about the structure of the plain text.

BASIC FUNCTIONALITY

Encrypt the text of your document using the script interface. For a start try out the settings as given in below listing.

```
mtxrun --script mtx-t-enigma                    \
       --setup="day_key = B I II III 01 01 01,  \
                rotor_setting = xyz,            \
                verbose=0"                      \
       --text="Gentlemen don't read each other's mail, Mr. Turing\!"
```

This will result in the thoroughly scrambled string `omribshpwfrfjovkntgqgi abbkhjpxmhdztapkatwrvf`. Then, use the same settings you encrypted the text with in your document.

```
\usemodule[enigma]
\defineenigma [secretmessage]
\setupenigma  [secretmessage] [
  day_key = B I II III 01 01 01,
  rotor_setting = xyz,
  verbose=3,
]

\starttext

\startsecretmessage
  omribshpwfrfjovkntgqgiabbkhjpxmhdztapkatwrvf
\stopsecretmessage

\stoptext
```

If you compile this document with ConTEXt, the plain text will reappear. Notice that punctuation is substituted with the letter “x” before encryption and that spaces are omitted.

Now it’s certainly not wise to carry around the key to encrypted documents as plain text within those documents. The keys will have to be distributed via an independent channel, e. g. a *code book*. Keys in general don’t have to be supplied inside the document. If there is none specified, the module will interrupt the TEX run and *ask* for user input. Suppose Alice wanted to send an encrypted file to Bob and already generated the cipher text as follows:

```
mtxrun --script mtx-t-enigma                                          \
       --setup="day_key =B I IV V 22 07 10 AZ DG IE YJ QM CW,         \
                rotor_setting = bar,                                  \
                verbose=0"                                            \
       --text="I have nothing to hide. From the NSA, that is."
```

Alice would then include the result of this line in her LATEX document as follows:

```
\documentclass{scrartcl}
\usepackage{enigma}
\defineenigma{decryption}
%% Encryption key not given in the setup.
\setupenigma{decryption}{
  rotor_setting = bar,
  verbose       = 3,
}
\begin{document}

\begindecryption
usbatbwcaajhzgeyzkqskupzbmdhbdepccgeh
\enddecryption

\end{document}
```

She subsequently mails this file to Bob and conveys the key through a secure channel. They only thing that will be left for Bob to do now, is to enter the key at the prompt when compiling the document with LuaLATEX.

USES AND ABUSES

In LuaTEX, *callback*s may stack. This allows filtering the input through many enigma machines successively. For instance, in the following listing, two instances of the same machine are generated and applied.

```
\usemodule [enigma]              %% load the module
\defineenigma [secretmessage]    %% generate and
\setupenigma  [secretmessage] [ %% configure a machine
  day_key = B IV V II o1 o1 o1 AD CN ET FL GI JV KZ PU QY WX,
  rotor_setting = foo,
  verbose=3,
]
```

```
%% now, copy the first machine's settings
\defineenigma [othermessage] [secretmessage]

%% here we go!
\starttext

\startothermessage  %% enable machine 1
\startsecretmessage %% enable machine 2 while no 1 is active
Encryption equals decryption.
\stopothermessage
\stopsecretmessage

\stoptext \endinput
```

METADATA

LICENSE

ACKNOWLEDGEMENTS

— The idea to implement the *Enigma* cipher for TEX came up while I was reading *The Code Book* by Simon Singh. This work contains an excellent portrayal of the history of German military cryptography and Allied cryptanalysis before and during the Second World War.

http://simonsingh.net/books/the-code-book/

– A more detailed account from a historical-cryptological perspective is provided by Ulrich Heinz in his Dissertation (in German), which is freely available. Includes an interesting albeit speculative note on the effectiveness of the Soviet code breaking efforts (chapter seven).
http://rzblo4.biblio.etc.tu-bs.de:8080/docportal/receive/DocPortal_document_00001705

– Also, the Enigma module drew lots of inspiration from Arno Trautmann's Chickenize package, which remains the unsurpassed hands-on introduction to callback trickery.
https://github.com/alt/chickenize

– Finally, without LuaTEX, encryption on node-level would not have been possible.
http://www.luatex.org/

2

Implementation

PLAINTEX MACROS

enigma.tex

```
\newif\ifenigmaisrunningplain
\ifcsname ver@enigma.sty\endcsname\else
  \enigmaisrunningplaintrue
  \input{luatexbase.sty}
  \catcode`\@=11
% \else latex
\fi
\catcode`\_=11 % There's no reason why this shouldn't be the case.
\catcode`\!=11
```

Nice tool from luat-ini.mkiv. This really helps with those annoying string separators of Lua's that clutter the source.

```
% this permits \typefile{self} otherwise nested b/e sep problems
\def\luastringsep{===}
\edef\!!bs{[\luastringsep[}
\edef\!!es{]\luastringsep]}
```

PREREQUISITES

Package loading and the namespacing issue are commented on in *enigma.lua*.

```
\directlua{
  packagedata = packagedata or { }
  dofile(kpse.find_file\!!bs enigma.lua\!!es)
}
```

First, create somthing like ConTEXt's asciimode. We found **\newluatexcatcodetable** in *luacode.sty* and it seems to get the job done.

```
\newluatexcatcodetable \enigmasetupcatcodes
\bgroup
  \def\escapecatcode      {0}
  \def\begingroupcatcode  {1}
  \def\endgroupcatcode    {2}
  \def\spacecatcode      {10}
  \def\lettercatcode     {11}
  \setluatexcatcodetable\enigmasetupcatcodes {
      \catcode`\^^I = \spacecatcode % tab
      \catcode`\    = \spacecatcode
      \catcode`\{   = \begingroupcatcode
      \catcode`\}   = \endgroupcatcode
      \catcode`\^^L = \lettercatcode    % form feed
      \catcode`\^^M = \lettercatcode    % eol
```

```
  }
\egroup
```

SETUPS

Once the proper catcodes are in place, the setup macro \do_setup_enigma doesn't to anything besides passing stuff through to Lua.

```
\def\do_setup_enigma#1{%
    \directlua{
      local enigma = packagedata.enigma
      local current_args = enigma.parse_args(\!!bs\detokenize{#1}\!!es)
      enigma.save_raw_args(current_args, \!!bs\current_enigma_id\!!es)
      enigma.new_callback(
        enigma.new_machine(\!!bs\current_enigma_id\!!es),
        \!!bs\current_enigma_id\!!es)
    }%
  \egroup%
}
```

The module setup \setupenigma expects key=value, notation. All the logic is at the Lua end, not much to see here ...

```
\def\setupenigma#1{%
  \bgroup
    \edef\current_enigma_id{#1}
    \luatexcatcodetable \enigmasetupcatcodes
    \do_setup_enigma%
}
```

ENCODING MACROS

The environment of \begin<enigmaid> and \end<enigmaid> toggles Enigma encoding. (Regarding environment delimiters we adhere to Knuth's practice of prefixing with `begin`/`end`.)

```
\def\e!start{begin} %{start}
\def \e!stop{end}   %{stop}
\edef\c!pre_linebreak_filter{pre_linebreak_filter}
\def\do_define_enigma#1{%
  \@EA\gdef\csname \e!start\current_enigma_id\endcsname{%
    \endgraf
    \bgroup%
    \directlua{%
      if packagedata.enigma                          and
         packagedata.enigma.machines[ \!!bs#1\!!es] then
        luatexbase.add_to_callback(
          \!!bs\c!pre_linebreak_filter\!!es,
          packagedata.enigma.callbacks[ \!!bs#1\!!es],
          \!!bs#1\!!es)
      else
        print\!!bs ENIGMA: No machine of that name: #1!\!!es
```

```
        os.exit()
      end
    }%
  }%
  \@EA\gdef\csname \e!stop\current_enigma_id\endcsname{%
    \endgraf
    \directlua{
      luatexbase.remove_from_callback(
        \!!bs\c!pre_linebreak_filter\!!es,
        \!!bs#1\!!es)
      packagedata.enigma.machines[ \!!bs#1\!!es]:processed_chars()
    }%
    \egroup%
  }%
}

\def\defineenigma#1{%
  \begingroup
  \let\@EA\expandafter
  \edef\current_enigma_id{#1}%
  \@EA\do_define_enigma\@EA{\current_enigma_id}%
  \endgroup%
}


\catcode`\_=8  % \popcatcodes
\catcode`\!=12 % reserved according to source2e
\ifenigmaisrunningplain\catcode`\@=12\fi
% vim:ft=plaintex:sw=2:ts=2:expandtab:tw=71
```

CONTEXT MACROS

t-enigma.mkvi

```
\unprotect

\writestatus{loading} {ConTeXt module / Enigma Document Encryption}

\startinterface all
  \setinterfacevariable {enigma} {enigma}
\stopinterface

\definenamespace [\v!enigma] [
 \v!command=\v!no,
    comment=Enigma Document Encryption,
    \s!name=\v!enigma,
  \s!parent=\v!enigma,
      % setup=\v!list,
      setup=\v!no,
      style=\v!no,
       type=module,
    version=hg-tip,
]
```

Loading the Lua conversion routines.

```
\startluacode
thirddata = thirddata or { }
\stopluacode
\registerctxluafile{enigma}
```

MACRO GENERATOR

The main setup. The **\defineenigma** macro does not adhere to the reommended practis of automatical macro derivation. Rather, we have our own parser do the job of setting globals. This is a consequence of the intention to offer the same behavior in any of the three main formats, PlainTEX, ConTEXtand LATEX. Hence, we don't rely on the internal mechanisms but implement our own macro generator.

```
\def\enigma_define_indeed#id{%
```

```
  \edef\enigmaid{#id}%
  \expandafter\gdef\csname \e!start\enigmaid\endcsname{%
    \endgraf
    \bgroup
    \ctxlua{
      if thirddata.enigma.machines["#id"] then
        nodes.tasks.enableaction("processors",
                                 "thirddata.enigma.callbacks.#id")
      else
        print([[ENIGMA: No machine of that name: #id!]])
      end
    }%
  }%
  %
  \expandafter\gdef\csname \e!stop\enigmaid\endcsname{%
    \endgraf%% We need to force a paragraph here for the callback to be
            %% applied.
    \ctxlua{
      nodes.tasks.disableaction("processors",
                                "thirddata.enigma.callbacks.#id")
      thirddata.enigma.machines["#id"]:processed_chars()
    }%
    \egroup%
  }%
}
```

The **\enigma_inherit** is called as an intermediate step when deriving one machine from an already existing one. It gets the stored configuration of its ancestor, relying on the **retrieve_raw_args** function (see *page 46*.

```
\def\enigma_inherit#to#from{%
  \ctxlua{%
    local enigma       = thirddata.enigma
    local current_args = enigma.retrieve_raw_args(\!!bs#from\!!es)
    enigma.save_raw_args(current_args, \!!bs#to\!!es)
    enigma.new_callback(enigma.new_machine(\!!bs#to\!!es),
                        \!!bs#to\!!es)
  }%
  \enigma_define_indeed{#to}%
}

\def\enigma_define[#id][#secondid]{%
  \ifsecondargument %% Copy an existing machine and callback.
    \enigma_inherit{#id}{#secondid}%
  \else             %% Create a new machine.
    \iffirstargument
      \enigma_define_indeed{#id}%
    \else
      \donothing
    \fi
  \fi
  \endgroup%
}

\def\defineenigma{%
  \begingroup
```

```
  \dodoubleempty\enigma_define
}
```

SETUP

```
\def\enigma_setup_indeed#args{%
    \ctxlua{
      local enigma = thirddata.enigma
      local current_args =
        enigma.parse_args(\!!bs\detokenize{#args}\!!es)
      enigma.save_raw_args(current_args, \!!bs\currentenigmaid\!!es)
      enigma.new_callback(
        enigma.new_machine(\!!bs\currentenigmaid\!!es),
        \!!bs\currentenigmaid\!!es)
    }%
}
```

The module setup **\setupenigma** expects key=value, notation. All the logic is at the Lua end, not much to see here ...

```
\def\enigma_setup[#id][#args]{%
  \ifsecondargument
    \edef\currentenigmaid{#id}
    \pushcatcodetable
    \catcodetable \txtcatcodes
      \enigma_setup_indeed{#args}%
  \else
    \donothing
  \fi
  \popcatcodetable
  \egroup%
}

\def\setupenigma{%
  \bgroup
  \dodoubleempty\enigma_setup%
}


\protect
% vim:ft=context:sw=2:ts=2:tw=71
```

LaTeX WRAPPER

enigma.sty

```
\ProvidesPackage
  {enigma}
  [2013/03/28 Enigma Document Encryption]
\RequirePackage{luatexbase}
\input{enigma}
\endinput
% vim:ft=tex:sw=2:ts=2:expandtab:tw=71
```

FUNCTIONALITY

enigma.lua

```
#!/usr/bin/env texlua
-----------------------------------------------------------------------
--         FILE:  enigma.lua
--        USAGE:  Call via interface from within a TeX session.
--  DESCRIPTION:  Enigma logic.
-- REQUIREMENTS:  LuaTeX capable format (Luaplain, ConTeXt).
--       AUTHOR:  Philipp Gesang (Phg), <phg42 dot 2a at gmail dot com>
--      VERSION:  release
--      CREATED:  2013-03-28 02:12:03+0100
-----------------------------------------------------------------------
--
```

FORMAT DEPENDENT CODE

Exported functionality will be collected in the table *enigma*.

```
local enigma = { machines = { }, callbacks = { } }
local format_is_context = false
```

Afaict, LaTeX for LuaTeX still lacks a globally accepted namespacing convention. This is more than bad, but we'll have to cope with that. For this reason we brazenly introduce *packagedata* (in analogy to ConTeXt's *thirddata*) table as a package namespace proposal. If this module is called from a LaTeX or plain session, the table *packagedata* will already have been created so we will identify the format according to its presence or absence, respectively.

```
if packagedata then            -- latex or plain
  packagedata.enigma = enigma
elseif thirddata then          -- context
  format_is_context = true
  thirddata.enigma  = enigma
else                           -- external call, mtx-script or whatever
  _ENV.enigma = enigma
end
```

PREREQUISITES

First of all, we generate local copies of all those library functions that are expected to be referenced frequently. The format-independent stuff comes first; it consists of functions from the *io*, *lpeg*, *math*, *string*, *table*, and *unicode* libraries.

```
local get_debug_info               = debug.getinfo
local ioread                       = io.read
local iowrite                      = io.write
local mathfloor                    = math.floor
local mathrandom                   = math.random
local next                         = next
local nodecopy                     = node and node.copy
local nodeid                       = node and node.id
local nodeinsert_before            = node and node.insert_before
local nodeinsert_after             = node and node.insert_after
local nodelength                   = node and node.length
local nodenew                      = node and node.new
local noderemove                   = node and node.remove
local nodeslide                    = node and node.slide
local nodetraverse                 = node and node.traverse
local nodetraverse_id              = node and node.traverse_id
local nodesinstallattributehandler
local nodestasksappendaction
local nodestasksdisableaction
if format_is_context then
  nodesinstallattributehandler = nodes.installattributehandler
  nodestasksappendaction       = nodes.tasks.appendaction
  nodestasksdisableaction      = nodes.tasks.disableaction
end
local stringfind                   = string.find
local stringformat                 = string.format
local stringlower                  = string.lower
local stringsub                    = string.sub
local stringupper                  = string.upper
local tableconcat                  = table.concat
local tonumber                     = tonumber
local type                         = type
local utf8byte                     = unicode.utf8.byte
local utf8char                     = unicode.utf8.char
local utf8len                      = unicode.utf8.len
local utf8lower                    = unicode.utf8.lower
local utf8sub                      = unicode.utf8.sub
local utfcharacters                = string.utfcharacters

--- debugging tool (careful, this *will* break context!)
--dofile(kpse.find_file("lualibs-table.lua")) -- archaic version :(
--table.print = function (...) print(table.serialize(...)) end

local tablecopy
if format_is_context then
  tablecopy = table.copy
else -- could use lualibs instead but not worth the overhead
  tablecopy = function (t) -- ignores tables as keys
    local result = { }
    for k, v in next, t do
      if type(v) == table then
        result[k] = tablecopy(v)
      else
        result[k] = v
      end
    end
```

```
    return result
  end
end

local GLYPH_NODE                  = node and nodeid"glyph"
local GLUE_NODE                   = node and nodeid"glue"
local GLUE_SPEC_NODE              = node and nodeid"glue_spec"
local KERN_NODE                   = node and nodeid"kern"
local DISC_NODE                   = node and nodeid"disc"
local HLIST_NODE                  = node and nodeid"hlist"
local VLIST_NODE                  = node and nodeid"vlist"

local IGNORE_NODES = node and {
--[GLUE_NODE] = true,
  [KERN_NODE] = true,
--[DISC_NODE] = true,
} or { }
```

The initialization of the module relies heavily on parsers generated by LPEG.

```
local lpeg = require "lpeg"

local C,   Cb, Cc, Cf, Cg,
      Cmt, Cp, Cs, Ct
  = lpeg.C,   lpeg.Cb, lpeg.Cc, lpeg.Cf, lpeg.Cg,
    lpeg.Cmt, lpeg.Cp, lpeg.Cs, lpeg.Ct

local P, R, S, V, lpegmatch
    = lpeg.P, lpeg.R, lpeg.S, lpeg.V, lpeg.match

--local B = lpeg.version() == "0.10" and lpeg.B or nil
```

By default the output to `stdout` will be zero. The verbosity level can be adjusted in order to alleviate debugging.

```
--local verbose_level = 42
local verbose_level = 0
```

Historically, Enigma-encoded messages were restricted to a size of 250 characters. With sufficient verbosity we will indicate whether this limit has been exceeded during the TEX run.

```
local max_msg_length = 250
```

GLOBALS

The following mappings are used all over the place as we convert back and forth between the characters (unicode) and their numerical representation.

```
local value_to_letter   -- { [int] -> chr }
local letter_to_value   -- { [chr] -> int }
local alpha_sorted      -- string, length 26
local raw_rotor_wiring  -- { string0, .. string5, }
local notches           -- { [int] -> int } // rotor num -> notch pos
local reflector_wiring  -- { { [int] -> int }, ... } // symmetrical
```

```
do
  value_to_letter = {
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z",
  }

  letter_to_value = {
    a = 01, b = 02, c = 03, d = 04, e = 05, f = 06, g = 07, h = 08,
    i = 09, j = 10, k = 11, l = 12, m = 13, n = 14, o = 15, p = 16,
    q = 17, r = 18, s = 19, t = 20, u = 21, v = 22, w = 23, x = 24,
    y = 25, z = 26,
  }
```

The five rotors to simulate. Their wirings are created from strings at runtime, see below the function **get_rotors**.

```
  --[[
    Nice: http://www.ellsbury.com/ultraenigmawirings.htm
  ]]--
  alpha_sorted = "abcdefghijklmnopqrstuvwxyz"
  raw_rotor_wiring = {
    [0] = alpha_sorted,
          "ekmflgdqvzntowyhxuspaibrcj",
          "ajdksiruxblhwtmcqgznpyfvoe",
          "bdfhjlcprtxvznyeiwgakmusqo",
          "esovpzjayquirhxlnftgkdcmwb",
          "vzbrgityupsdnhlxawmjqofeck",
  }
```

Notches are assigned to rotors according to the Royal Army mnemonic.

```
  notches = { }
  do
    local raw_notches = "rfwkannnn"
    --local raw_notches = "qevjz"
    local n = 1
    for chr in utfcharacters(raw_notches) do
      local pos = stringfind(alpha_sorted, chr)
      notches[n] = pos - 1
      n = n + 1
    end
  end
```

UKW a	AE BJ CM DZ FL GY HX IV KW NR OQ PU ST
UKW b	AY BR CU DH EQ FS GL IP JX KN MO TZ VW
UKW c	AF BV CP DJ EI GO HY KR LZ MX NW QT SU

Table 1 The three reflectors and their substitution rules.

```
  reflector_wiring = { }
  local raw_ukw = {
    { a = "e", b = "j", c = "m", d = "z", f = "l", g = "y", h = "x",
      i = "v", k = "w", n = "r", o = "q", p = "u", s = "t", },
    { a = "y", b = "r", c = "u", d = "h", e = "q", f = "s", g = "l",
      i = "p", j = "x", k = "n", m = "o", t = "z", v = "w", },
    { a = "f", b = "v", c = "p", d = "j", e = "i", g = "o", h = "y",
      k = "r", l = "z", m = "x", n = "w", q = "t", s = "u", },
```

```
  }
  for i=1, #raw_ukw do
    local new_wiring = { }
    local current_ukw = raw_ukw[i]
    for from, to in next, current_ukw do
      from = letter_to_value[from]
      to   = letter_to_value[to]
      new_wiring[from] = to
      new_wiring[to]   = from
    end
    reflector_wiring[i] = new_wiring
  end
end
```

PRETTY PRINTING FOR DEBUG PURPOSES

The functions below allow for formatting of the terminal output; they have no effect on the workings of the enigma simulator.

```
local emit
local pprint_ciphertext
local pprint_encoding
local pprint_encoding_scheme
local pprint_init
local pprint_machine_step
local pprint_new_machine
local pprint_rotor
local pprint_rotor_scheme
local pprint_step
local polite_key_request
local key_invalid
do
  local eol = "\n"

  local colorstring_template = "\027[%d;1m%s\027[0m"
  local colorize = function (s, color)
    color = color and color < 38 and color > 29 and color or 31
    return stringformat(colorstring_template,
                        color,
                        s)
  end

  local underline = function (s)
    return stringformat("\027[4;37m%s\027[0m", s)
  end

  local s_steps     = [[Total characters encoded with machine "]]
  local f_warnsteps = [[ (%d over permitted maximum)]]
  pprint_machine_step = function (n, name)
    local sn
    name = colorize(name, 36)
    if n > max_msg_length then
      sn = colorize(n, 31) .. stringformat(f_warnsteps,
```

```
                                        n - max_msg_length)
  else
    sn = colorize(n, 37)
  end
  return s_steps .. name .. "": " .. sn .. "."
end
local rotorstate = "[s \027[1;37m%s\027[0m n\027[1;37m%2d\027[0m]> "
pprint_rotor = function (rotor)
  local visible = rotor.state % 26 + 1
  local w, n    = rotor.wiring, (rotor.notch - visible) % 26 + 1
  local tmp = { }
  for i=1, 26 do
    local which = (i + rotor.state - 1) % 26 + 1
    local chr   = value_to_letter[rotor.wiring.from[which]]
    if i == n then -- highlight positions of notches
      tmp[i] = colorize(stringupper(chr), 32)
    --elseif chr == value_to_letter[visible] then
    ---- highlight the character in window
    --  tmp[i] = colorize(chr, 33)
    else
      tmp[i] = chr
    end
  end
  return stringformat(rotorstate,
                      stringupper(value_to_letter[visible]),
                      n)
      .. tableconcat(tmp)
end

local rotor_scheme = underline"[rot not]"
                  .. "  "
                  .. underline(alpha_sorted)
pprint_rotor_scheme = function ()
  return rotor_scheme
end

local s_encoding_scheme = eol
                       .. [[in > 1 => 2 => 3 > UKW > 3 => 2 => 1]]
pprint_encoding_scheme = function ()
  return underline(s_encoding_scheme)
end
local s_step     = " => "
local stepcolor  = 36
local finalcolor = 32
pprint_encoding = function (steps)
  local nsteps, result = #steps, { }
  for i=0, nsteps-1 do
    result[i+1] = colorize(value_to_letter[steps[i]], stepcolor)
                .. s_step
  end
  result[nsteps+1] = colorize(value_to_letter[steps[nsteps]],
                              finalcolor)
  return tableconcat(result)
end

local init_announcement
```

```
    = colorize("\n" .. [[Initial position of rotors: ]],
                                  37)
pprint_init = function (init)
  local result = ""
  result = value_to_letter[init[1]] .. " "
        .. value_to_letter[init[2]] .. " "
        .. value_to_letter[init[3]]
  return init_announcement .. colorize(stringupper(result), 34)
end

local machine_announcement =
  [[Enigma machine initialized with the following settings.]] .. eol
local s_ukw  = colorize("        Reflector:", 37)
local s_pb   = colorize("Plugboard setting:", 37)
local s_ring = colorize("   Ring positions:", 37)
local empty_plugboard = colorize(" --", 34)
pprint_new_machine = function (m)
  local result = { "" }
  result[#result+1] = underline(machine_announcement)
  result[#result+1] = s_ukw
                   .. " "
                   .. colorize(
                        stringupper(value_to_letter[m.reflector]),
                        34
                      )
  local rings = ""
  for i=1, 3 do
    local this = m.ring[i]
    rings = rings
         .. " "
         .. colorize(stringupper(value_to_letter[this + 1]), 34)
  end
  result[#result+1] = s_ring .. rings
  if m.__raw.plugboard then
    local tpb, pb = m.__raw.plugboard, ""
    for i=1, #tpb do
      pb = pb .. " " .. colorize(tpb[i], 34)
    end
    result[#result+1] = s_pb .. pb
  else
    result[#result+1] = s_pb .. empty_plugboard
  end
  result[#result+1] = ""
  result[#result+1] = pprint_rotor_scheme()
  for i=1, 3 do
    result[#result+1] = pprint_rotor(m.rotors[i])
  end
  return tableconcat(result, eol) .. eol
end

local step_template  = colorize([[Step № ]], 37)
local chr_template   = colorize([[  --  Input ]], 37)
local pbchr_template = colorize([[ → ]], 37)
pprint_step = function (n, chr, pb_chr)
  return eol
      .. step_template
```

```
      .. colorize(n, 34)
      .. chr_template
      .. colorize(stringupper(value_to_letter[chr]), 34)
      .. pbchr_template
      .. colorize(stringupper(value_to_letter[pb_chr]), 34)
      .. eol
end

-- Split the strings into lines, group them in bunches of five etc.
local tw = 30
local pprint_textblock = function (s)
  local len = utf8len(s)
  local position = 1    -- position in string
  local nline    = 5    -- width of current line
  local out      = utf8sub(s, position, position+4)
  repeat
    position = position + 5
    nline    = nline + 6
    if nline > tw then
      out = out .. eol .. utf8sub(s, position, position+4)
      nline = 1
    else
      out = out .. " " .. utf8sub(s, position, position+4)
    end
  until position > len
  return out
end

local intext  = colorize([[Input text:]], 37)
local outtext = colorize([[Output text:]], 37)
pprint_ciphertext = function (input, output, upper_p)
  if upper_p then
    input  = stringupper(input)
    output = stringupper(output)
  end
  return eol
      .. intext
      .. eol
      .. pprint_textblock(input)
      .. eol .. eol
      .. outtext
      .. eol
      .. pprint_textblock(output)
end
```

emit is the main wrapper function for *stdout*. Checks if the global verbosity setting exceeds the specified threshold, and only then pushes the output.

```
emit = function (v, f, ...)
  if f and v and verbose_level >= v then
    iowrite(f(...) .. eol)
  end
  return o
end
```

The **polite_key_request** will be called in case the *day_key* field of the machine setup is empty at the time of initialization.

```
  local s_request = "\n\n                      "
                 .. underline"This is an encrypted document." .. [[


           Please enter the document key for enigma machine
                             "%s".

                             Key Format:

Ref R1 R2 R3 I1 I2 I3 [P1 ..]   Ref: reflector A/B/C
                                Rn:  rotor, I through V
                                In:  ring position, 01 through 26
                                Pn:  optional plugboard wiring, upto 32

>]]
  polite_key_request = function (name)
    return stringformat(s_request, colorize(name, 33))
  end

  local s_invalid_key = colorize"Warning!"
                     .. " The specified key is invalid."
  key_invalid = function ()
    return s_invalid_key
  end
end
```

The functions **new** and **ask_for_day_key** are used outside their scope, so we declare them beforehand.

```
local new
local ask_for_day_key
do
```

ROTATION

The following function **do_rotate** increments the rotational state of a single rotor. There are two tests for notches:

1. whether it's at the current character, and
2. whether it's at the next character.

The latter is an essential prerequisite for double-stepping.

```
  local do_rotate = function (rotor)
    rotor.state = rotor.state % 26 + 1
    return rotor,
           rotor.state     == rotor.notch,
           rotor.state + 1 == rotor.notch
  end
```

The **rotate** function takes care of rotor (*Walze*) movement. This entails incrementing the next rotor whenever the notch has been reached and covers the corner case *double stepping*.

```
local rotate = function (machine)
  local rotors     = machine.rotors
  local rc, rb, ra = rotors[1], rotors[2], rotors[3]

  ra, nxt = do_rotate(ra)
  if nxt or machine.double_step then
    rb, nxt, nxxt = do_rotate(rb)
    if nxt then
      rc = do_rotate(rc)
    end
    if nxxt then
      --- weird: home.comcast.net/~dhhamer/downloads/rotors1.pdf
      machine.double_step = true
    else
      machine.double_step = false
    end
  end
  machine.rotors = { rc, rb, ra }
end
```

INPUT PREPROCESSING

Internally, we will use lowercase strings as they are a lot more readable than uppercase. Lowercasing happens prior to any further dealings with input. After the encoding or decoding has been accomplished, there will be an optional (re-)uppercasing.

Substitutions are applied onto the lowercased input. You might want to avoid some of these, above all the rules for numbers, because they translate single digits only. The solution is to write out numbers above ten.

```
local pp_substitutions = {
  -- Umlauts are resolved.
  ["ö"]  = "oe",
  ["ä"]  = "ae",
  ["ü"]  = "ue",
  ["ß"]  = "ss",
  -- WTF?
  ["ch"] = "q",
  ["ck"] = "q",
  -- Punctuation -> "x"
  [","]  = "x",
  ["."]  = "x",
  [";"]  = "x",
  [":"]  = "x",
  ["/"]  = "x",
  ["’"]  = "x",
  ["‘"]  = "x",
  ["„"]  = "x",
  ["“"]  = "x",
  ["”"]  = "x",
  ["-"]  = "x",
  ["–"]  = "x",
  ["—"]  = "x",
```

```
  ["!"]  = "x",
  ["?"]  = "x",
  ["‽"]  = "x",
  ["("]  = "x",
  [")"]  = "x",
  ["["]  = "x",
  ["]"]  = "x",
  ["<"]  = "x",
  [">"]  = "x",
  -- Spaces are omitted.
  [" "]  = "",
  ["\n"] = "",
  ["\t"] = "",
  ["\v"] = "",
  ["\\"] = "",
  -- Numbers are resolved.
  ["0"]  = "null",
  ["1"]  = "eins",
  ["2"]  = "zwei",
  ["3"]  = "drei",
  ["4"]  = "vier",
  ["5"]  = "fuenf",
  ["6"]  = "sechs",
  ["7"]  = "sieben",
  ["8"]  = "acht",
  ["9"]  = "neun",
}
```

MAIN FUNCTION CHAIN TO BE APPLIED TO SINGLE CHARACTERS

As far as the Enigma is concerned, there is no difference between encoding and decoding. Thus, we need only one function (**encode_char**) to achieve the complete functionality. However, within every encoding step, characters will be wired differently in at least one of the rotors according to its rotational state. Rotation is simulated by adding the *state* field of each rotor to the letter value (its position on the ingoing end).

boolean	direction	meaning
true	“from”	right to left
false	“to”	left to right

Table 2 Directional terminology

The function **do_do_encode_char** returns the character substitution for one rotor. As a letter passes through each rotor twice, the argument *direction* determines which way the substitution is applied.

```
local do_do_encode_char = function (char, rotor, direction)
  local rw     = rotor.wiring
  local rs     = rotor.state
```

```
  local result = char
  if direction then -- from
    result = (result + rs - 1) % 26 + 1
    result = rw.from[result]
    result = (result - rs - 1) % 26 + 1
  else -- to
    result = (result + rs - 1) % 26 + 1
    result = rw.to[result]
    result = (result - rs - 1) % 26 + 1
  end
  return result
end
```

Behind the plugboard, every character undergoes seven substitutions: two for each rotor plus the central one through the reflector. The function **do_encode_char**, although it returns the final result only, keeps every intermediary step inside a table for debugging purposes. This may look inefficient but is actually a great advantage whenever something goes wrong.

```
--- ra -> rb -> rc -> ukw -> rc -> rb -> ra
local do_encode_char = function (rotors, reflector, char)
  local rc, rb, ra = rotors[1], rotors[2], rotors[3]
  local steps = { [0] = char }
  --
  steps[1] = do_do_encode_char(steps[0], ra,  true)
  steps[2] = do_do_encode_char(steps[1], rb,  true)
  steps[3] = do_do_encode_char(steps[2], rc,  true)
  steps[4] = reflector_wiring[reflector][steps[3]]
  steps[5] = do_do_encode_char(steps[4], rc, false)
  steps[6] = do_do_encode_char(steps[5], rb, false)
  steps[7] = do_do_encode_char(steps[6], ra, false)
  emit(2, pprint_encoding_scheme)
  emit(2, pprint_encoding, steps)
  return steps[7]
end
```

Before an input character is passed on to the actual encoding routing, the function **encode_char** matches it agains the latin alphabet. Characters failing this test are either passed through or ignored, depending on the machine option *other_chars*. Also, the counter of encoded characters is incremented at this stage and some pretty printer hooks reside here.

encode_char contributes only one element of the encoding procedure: the plugboard (*Steckerbrett*). Like the rotors described above, a character passed through this device twice; the plugboard marks the beginning and end of every step. For debugging purposes, the first substitution is stored in a separate local variable, *pb_char*.

```
local encode_char = function (machine, char)
  machine.step = machine.step + 1
  machine:rotate()
  local pb = machine.plugboard
  char = letter_to_value[char]
  local pb_char = pb[char]           -- first plugboard substitution
  emit(2, pprint_step, machine.step, char, pb_char)
```

```
  emit(3, pprint_rotor_scheme)
  emit(3, pprint_rotor, machine.rotors[1])
  emit(3, pprint_rotor, machine.rotors[2])
  emit(3, pprint_rotor, machine.rotors[3])
  char = do_encode_char(machine.rotors,
                        machine.reflector,
                        pb_char)
  return value_to_letter[pb[char]]   -- second plugboard substitution
end

local get_random_pattern = function ()
  local a, b, c
      = mathrandom(1,26), mathrandom(1,26), mathrandom(1,26)
  return value_to_letter[a]
      .. value_to_letter[b]
      .. value_to_letter[c]
end

local pattern_to_state = function (pat)
  return {
    letter_to_value[stringsub(pat, 1, 1)],
    letter_to_value[stringsub(pat, 2, 2)],
    letter_to_value[stringsub(pat, 3, 3)],
  }
end

local set_state = function (machine, state)
  local rotors = machine.rotors
  for i=1, 3 do
    rotors[i].state = state[i] - 1
  end
end
```

When Enigma is called from TEX, the encoding proceeds character by character as we iterate one node at a time. **encode_string** is a wrapper for use with strings, e. g. in the mtx-script (*page 8*). It handles iteration and extraction of successive characters from the sequence.

```
local encode_string = function (machine, str) --, pattern)
  local result = { }
  for char in utfcharacters(str) do
    local tmp = machine:encode(char)
    if tmp ~= false then
      if type(tmp) == "table" then
        for i=1, #tmp do
          result[#result+1] = tmp[i]
        end
      else
        result[#result+1] = tmp
      end
    end
  end
  machine:processed_chars()
  return tableconcat(result)
```

```
end
```

INITIALIZATION STRING PARSER

Reflector in slot	Rotor setting			Initial rotor			Plugboard wiring									
	1	2	3	1	2	3	1	2	3	4	5	6	7	8	9	10
B	I	IV	III	16	26	08	AD	CN	ET	FL	GI	JV	KZ	PU	QY	WX

Table 3 Initialization strings

```
local roman_digits = {
  i   = 1, I   = 1,
  ii  = 2, II  = 2,
  iii = 3, III = 3,
  iv  = 4, IV  = 4,
  v   = 5, V   = 5,
}

local p_init = P{
  "init",
  init               = V"whitespace"^-1 * Ct(V"do_init"),
  do_init            = (V"reflector" * V"whitespace")^-1
                     * V"rotors"     * V"whitespace"
                     * V"ring"
                     * (V"whitespace" * V"plugboard")^-1
                     ,
  reflector          = Cg(C(R("ac","AC")) / stringlower, "reflector")
                     ,

  rotors             = Cg(Ct(V"rotor" * V"whitespace"
                          * V"rotor" * V"whitespace"
                          * V"rotor"),
                          "rotors")
                     ,
  rotor              = Cs(V"roman_five"  / roman_digits
                        + V"roman_four"  / roman_digits
                        + V"roman_three" / roman_digits
                        + V"roman_two"   / roman_digits
                        + V"roman_one"   / roman_digits)
                     ,
  roman_one          = P"I"   + P"i",
  roman_two          = P"II"  + P"ii",
  roman_three        = P"III" + P"iii",
  roman_four         = P"IV"  + P"iv",
  roman_five         = P"V"   + P"v",

  ring               = Cg(Ct(V"double_digit" * V"whitespace"
                           * V"double_digit" * V"whitespace"
                           * V"double_digit"),
                          "ring")
```

```
                    ,
  double_digit      = C(R"02" * R"09"),

  plugboard         = Cg(V"do_plugboard", "plugboard"),
  --- no need to enforce exactly ten substitutions
  --do_plugboard      = Ct(V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination" * V"whitespace"
  --                     * V"letter_combination")
  do_plugboard      = Ct(V"letter_combination"
                       * (V"whitespace" * V"letter_combination")^0)
                    ,
  letter_combination = C(R("az", "AZ") * R("az", "AZ")),

  whitespace        = S" \n\t\v"^1,
}
```

INITIALIZATION ROUTINES

The plugboard is implemented as a pair of hash tables.

```
local get_plugboard_substitution = function (p)
  --- Plugboard wirings are symmetrical, thus we have one table for
  --- each direction.
  local tmp, result = { }, { }
  for _, str in next, p do
    local one, two = stringlower(stringsub(str, 1, 1)),
                     stringlower(stringsub(str, 2))
    tmp[one] = two
    tmp[two] = one
  end
  local n_letters = 26

  local lv = letter_to_value
  for n=1, n_letters do
    local letter  = value_to_letter[n]
    local sub = tmp[letter] or letter
    -- Map each char either to the plugboard substitution or itself.
    result[lv[letter]] = lv[sub or letter]
  end
  return result
end
```

Initialization of the rotors requires some precautions to be taken. The most obvious of which is adjusting the displacement of its wiring by the ring setting.

Another important task is to store the notch position in order for it to be retrievable by the rotation subroutine at a later point. The actual bidirectional mapping is implemented as a pair of tables. The initial order of letters, before the ring shift is applied, is alphabetical on the input (right, "from") side and, on the output (left, "to") side taken by the hard wired correspondence as specified in the rotor wirings above. NB the descriptions in terms of "output" and "input" directions is misleading in so far as during any encoding step the electricity will pass through every rotor in both ways. Hence, the "input" (right, from) direction literally applies only to the first half of the encoding process between plugboard and reflector. The function **do_get_rotor** creates a single rotor instance and populates it with character mappings. The *from* and *to* subfields of its *wiring* field represent the wiring in the respective directions. This initital wiring was specified in the corresponding *raw_rotor_wiring* table; the ringshift is added modulo the alphabet size in order to get the correctly initialized rotor.

```
local do_get_rotor = function (raw, notch, ringshift)
  local rotor = {
    wiring = {
      from  = { },
      to    = { },
    },
    state = 0,
    notch = notch,
  }
  local w = rotor.wiring
  for from=1, 26 do
    local to   = letter_to_value[stringsub(raw, from, from)]
    --- The shift needs to be added in both directions.
    to   = (to   + ringshift - 1) % 26 + 1
    from = (from + ringshift - 1) % 26 + 1
    rotor.wiring.from[from] = to
    rotor.wiring.to  [to  ] = from
  end
  --table.print(rotor, "rotor")
  return rotor
end
```

Rotors are initialized sequentially accordings to the initialization request. The function **get_rotors** walks over the list of initialization instructions and calls **do_get_rotor** for the actual generation of the rotor table. Each rotor generation request consists of three elements:

1. the choice of rotor (one of five),
2. the notch position of said rotor, and
3. the ring shift.

```
local get_rotors = function (rotors, ring)
  local s, r = { }, { }
  for n=1, 3 do
    local nr = tonumber(rotors[n])
```

```
    local ni = tonumber(ring[n]) - 1 -- "1" means shift of zero
    r[n] = do_get_rotor(raw_rotor_wiring[nr], notches[nr], ni)
    s[n] = ni
  end
  return r, s
end

local decode_char = encode_char -- hooray for involutory ciphers
```

The function **encode_general** is an intermediate step for the actual single-character encoding / decoding routine **enchode_char**. Its purpose is to ensure encodability of a given character before passing it along. Characters are first checked against the replacement table *pp_substitutions* (see *page 30*). For single-character replacements the function returns the encoded character (string). However, should the replacement turn out to consist of more than one character, each one will be encoded successively, yielding a list.

```
local encode_general = function (machine, chr)
  local chr = utf8lower(chr)
  local replacement
      = pp_substitutions[chr] or letter_to_value[chr] and chr
  if not replacement then
    if machine.other_chars then
      return chr
    else
      return false
    end
  end

  if utf8len(replacement) == 1 then
    return encode_char(machine, replacement)
  end
  local result = { }
  for new_chr in utfcharacters(replacement) do
    result[#result+1] = encode_char(machine, new_chr)
  end
  return result
end

local process_message_key
local alpha        = R"az"
local alpha_dec    = alpha / letter_to_value
local whitespace   = S" \n\t\v"
local mkeypattern  = Ct(alpha_dec  * alpha_dec * alpha_dec)
                   * whitespace^0
                   * C(alpha * alpha *alpha)
process_message_key = function (machine, message_key)
  message_key = stringlower(message_key)
  local init, three = lpegmatch(mkeypattern, message_key)
  -- to be implemented
end

local decode_string = function (machine, str, message_key)
  machine.kenngruppe, str = stringsub(str, 3, 5), stringsub(str, 6)
  machine:process_message_key(message_key)
```

```
  local decoded = encode_string(machine, str)
  return decoded
end

local testoptions = {
  size = 42,

}
local generate_header = function (options)
end

local processed_chars = function (machine)
  emit(1, pprint_machine_step, machine.step, machine.name)
end
```

The day key is entrusted to the function **handle_day_key**. If the day key is the empty string or `nil`, it will ask for a key on the terminal. (Cf. below, *page 41*.) Lesson: don't forget providing day keys in your setups when running in batch mode.

```
local handle_day_key handle_day_key = function (dk, name, old)
  local result
  if not dk or dk == "" then
    dk = ask_for_day_key(name, old)
  end
  result = lpegmatch(p_init, dk)
  result.reflector = result.reflector or "b"
  -- If we don't like the key we're going to ask again. And again....
  return result or handle_day_key(nil, name, dk)
end
```

The enigma encoding is restricted to an input – and, naturally, output – alphabet of exactly twenty-seven characters. Obviously, this would severely limit the set of encryptable documents. For this reason the plain text would be *preprocessed* prior to encoding, removing spaces and substituting a range of characters, e. g. punctuation, with placeholders ("X") from the encodable spectrum. See above *page 30* for a comprehensive list of substitutions.

The above mentioned preprocessing, however, does not even nearly extend to the whole unicode range that modern day typesetting is expected to handle. Thus, sooner or later an Enigma machine will encounter non-preprocessable characters and it will have to decide what to do with them. The Enigma module offers two ways to handle this kind of situation: *drop* those characters, possibly distorting the deciphered plain text, or to leave them in, leaving hints behind as to the structure of the encrypted text. None of these is optional, so it is nevertheless advisable to not include non-latin characters in the plain text in the first place. The settings key *other_chars* (type boolean) determines whether we will keep or drop offending characters.

```
new = function (name, args)
  local setup_string, pattern = args.day_key, args.rotor_setting
  local raw_settings = handle_day_key(setup_string, name)
  local rotors, ring =
    get_rotors(raw_settings.rotors, raw_settings.ring)
```

```
    local plugboard
        = raw_settings.plugboard
          and get_plugboard_substitution(raw_settings.plugboard)
          or  get_plugboard_substitution{ }
    local machine = {
      name                = name,
      step                = o, -- n characters encoded
      init                = {
        rotors = raw_settings.rotors,
        ring   = raw_settings.ring
      },
      rotors              = rotors,
      ring                = ring,
      state               = init_state,
      other_chars         = args.other_chars,
      spacing             = args.spacing,
      ---> a>1, b>2, c>3
      reflector           = letter_to_value[raw_settings.reflector],
      plugboard           = plugboard,
      --- functionality
      rotate              = rotate,
      --process_message_key = process_message_key,
      encode_string       = encode_string,
      encode_char         = encode_char,
      encode              = encode_general,
      decode_string       = decode_string,
      decode_char         = decode_char,
      set_state           = set_state,
      processed_chars     = processed_chars,
      --- <badcodingstyle> -- hackish but occasionally useful
      __raw               = raw_settings
      --- </badcodingstyle>
    } --- machine
    local init_state
      = pattern_to_state(pattern or get_random_pattern())
    emit(1, pprint_init, init_state)
    machine:set_state(init_state)

    --table.print(machine.rotors)
    emit(1, pprint_new_machine, machine)
    return machine
  end

end
```

SETUP ARGUMENT HANDLING

```
do
```

As the module is intended to work both with the Plain and LaTeX formats as well as ConTeXt, we can't rely on format dependent setups. Hence the need for an argument parser. Should be more efficient anyways as all the functionality resides in Lua.

```
  local p_args = P{
```

```
  "args",
  args           = Cf(Ct"" * (V"kv_pair" + V"emptyline")^0, rawset),
  kv_pair        = Cg(V"key"
                    * V"separator"
                    * (V"value" * V"final"
                    + V"empty"))
                 * V"rest_of_line"^-1
                 ,
  key             = V"whitespace"^0 * C(V"key_char"^1),
  key_char        = (1 - V"whitespace" - V"eol" - V"equals")^1,
  separator       = V"whitespace"^0 * V"equals" * V"whitespace"^0,
  empty           = V"whitespace"^0 * V"comma" * V"rest_of_line"^-1
                  * Cc(false)
                  ,
  value           = C((V"balanced" + (1 - V"final"))^1),
  final           = V"whitespace"^0 * V"comma" + V"rest_of_string",
  rest_of_string = V"whitespace"^0
                  * V"eol_comment"^-1
                  * V"eol"^0
                  * V"eof"
                  ,
  rest_of_line    = V"whitespace"^0 * V"eol_comment"^-1 * V"eol",
  eol_comment     = V"comment_string" * (1 - (V"eol" + V"eof"))^0,
  comment_string = V"lua_comment" + V"TeX_comment",
  TeX_comment     = V"percent",
  lua_comment     = V"double_dash",
  emptyline       = V"rest_of_line",

  balanced        = V"balanced_brk" + V"balanced_brc",
  balanced_brk    = V"lbrk"
                  * (V"balanced" + (1 - V"rbrk"))^0
                  * V"rbrk"
                  ,
  balanced_brc    = V"lbrc"
                  * (V"balanced" + (1 - V"rbrc"))^0
                  * V"rbrc"
                  ,
  -- Terminals
  eol             = P"\n\r" + P"\r\n" + P"\n" + P"\r",
  eof             = -P(1),
  whitespace      = S" \t\v",
  equals          = P"=",
  dot             = P".",
  comma           = P",",
  dash            = P"-",    double_dash  = V"dash" * V"dash",
  percent         = P"%",
  lbrk            = P"[",    rbrk         = P"]",
  lbrc            = P"{",    rbrc         = P"}",
}
```

In the next step we process the arguments, check the input for sanity etc. The function **parse_args** will test whether a value has a sanitizer routine and, if so, apply it to its value.

```
local boolean_synonyms = {
```

```
  ["1"]    = true,
  doit     = true,
  indeed   = true,
  ok       = true,
  [""]    = true,
  ["true"] = true,
  yes      = true,
}
local toboolean
    = function (value) return boolean_synonyms[value] or false end
local alpha = R("az", "AZ")
local digit = R"09"
local space = S" \t\v"
local ans   = alpha + digit + space
local p_ans = Cs((ans + (1 - ans / ""))^1)
local alphanum_or_space  = function (str)
  if type(str) ~= "string" then return nil end
  return lpegmatch(p_ans, str)
end
local ensure_int = function (n)
  n = tonumber(n)
  if not n then return 0 end
  return mathfloor(n + 0.5)
end
p_alpha = Cs((alpha + (1 - alpha / ""))^1)
local ensure_alpha = function (s)
  s = tostring(s)
  return lpegmatch(p_alpha, s)
end

local sanitizers = {
  other_chars   = toboolean,          -- true = keep, false = drop
  spacing       = toboolean,
  day_key       = alphanum_or_space,
  rotor_setting = ensure_alpha,
  verbose       = ensure_int,
}
enigma.parse_args = function (raw)
  local args = lpegmatch(p_args, raw)
  for k, v in next, args do
    local f = sanitizers[k]
    if f then
      args[k] = f(v)
    else
      -- OPTIONAL be fascist and permit only predefined args
      args[k] = v
    end
  end
  return args
end
```

If the machine setting lacks key settings then we'll go ahead and ask the user directly, hence the function **ask_for_day_key**. We abort after three misses lest we annoy the user ...

```
local max_tries = 3
ask_for_day_key = function (name, old, try)
```

```
    if try == max_tries then
      iowrite[[
Aborting. Entered invalid key three times.
]]
      os.exit()
    end
    if old then
      emit(o, key_invalid)
    end
    emit(o, polite_key_request, name)
    local result = ioread()
    iowrite("\n")
    return alphanum_or_space(result) or
           ask_for_day_key(name, (try and try + 1 or 1))
  end
end
```

CALLBACK

This is the interface to TEX. We generate a new callback handler for each defined Enigma machine. ConTEXt delivers the head as third argument of a callback only (...?), so we'll have to do some variable shuffling on the function side.

When grouping output into the traditional blocks of five letters we insert space nodes. As their properties depend on the font we need to recreate the space item for every paragraph. Also, as ConTEXt does not preload a font we lack access to font metrics before `\starttext`. Thus creating the space earlier will result in an error. The function **generate_space** will be called inside the callback in order to get an appropriate space glue.

```
local generate_space = function ( )
  local current_fontparms = font.getfont(font.current()).parameters
  local space_node        = nodenew(GLUE_NODE)
  space_node.spec         = nodenew(GLUE_SPEC_NODE)
  space_node.spec.width   = current_fontparms.space
  space_node.spec.shrink  = current_fontparms.space_shrink
  space_node.spec.stretch = current_fontparms.space_stretch
  return space_node
end
```

Registering a callback ("node attribute"?, "node task"?, "task action"?) in ConTEXt is not straightforward, let alone documented. The trick is to create, install and register a handler first in order to use it later on ... many thanks to Khaled Hosny, who posted an answer to *tex.sx*.

```
local new_callback = function (machine, name)
  enigma.machines [name] = machine
  local format_is_context = format_is_context
  local current_space_node
  local mod_5 = 0

  --- First we need to choose an insertion method. If autospacing is
  --- requested, a space will have to be inserted every five
```

```
--- characters.  The rationale behind using differend functions to
--- implement each method is that it should be faster than branching
--- for each character.
local insert_encoded

if machine.spacing then -- auto-group output
  insert_encoded = function (head, n, replacement)
    local insert_glyph = nodecopy(n)
    if replacement then -- inefficient but bulletproof
      insert_glyph.char = utf8byte(replacement)
      --print(utf8char(n.char), "=>", utf8char(insertion.char))
    end
    --- if we insert a space we need to return the
    --- glyph node in order to track positions when
    --- replacing multiple nodes at once (e.g. ligatures)
    local insertion  = insert_glyph
    mod_5 = mod_5 + 1
    if mod_5 > 5 then
      mod_5 = 1
      insertion = nodecopy(current_space_node)
      insertion.next, insert_glyph.prev = insert_glyph, insertion
    end
    if head == n then --> replace head
      local succ = head.next
      if succ then
        insert_glyph.next, succ.prev = succ, insert_glyph
      end
      head = insertion
    else --> replace n
      local pred, succ = n.prev, n.next
      pred.next, insertion.prev = insertion, pred
      if succ then
        insert_glyph.next, succ.prev = succ, insert_glyph
      end
    end

    --- insertion becomes the new head
    return head, insert_glyph -- so we know where to insert
  end
else

  insert_encoded = function (head, n, replacement)
    local insertion = nodecopy(n)
    if replacement then
      insertion.char = utf8byte(replacement)
    end
    if head == n then
      local succ = head.next
      if succ then
        insertion.next, succ.prev = succ, insertion
      end
      head = insertion
    else
      nodeinsert_before(head, n, insertion)
      noderemove(head, n)
    end
```

```
    return head, insertion
  end
end

--- The callback proper starts here.
local aux aux = function (head, recurse)
  if recurse == nil then recurse = 0 end
  for n in nodetraverse(head) do
    local nid = n.id
    --print(utf8char(n.char), n)
    if nid == GLYPH_NODE then
      local chr         = utf8char(n.char)
      --print(chr, n)
      local replacement  = machine:encode(chr)
      --print(chr, replacement, n)
      local treplacement = replacement and type(replacement)
      --if replacement == false then
      if not replacement then
        noderemove(head, n)
      elseif treplacement == "string" then
        --print(head, n, replacement)
        head, _ = insert_encoded(head, n, replacement)
      elseif treplacement == "table" then
        local current = n
        for i=1, #replacement do
          head, current = insert_encoded(head, current, replacement[i])
        end
      end
    elseif nid == GLUE_NODE then
      if n.subtype ~= 15 then -- keeping the parfillskip
        noderemove(head, n)
      end
    elseif IGNORE_NODES[nid] then
      -- drop spaces and kerns
      noderemove(head, n)
    elseif nid == DISC_NODE then
      --- ligatures need to be resolved if they are characters
      local npre, npost = n.pre, n.post
      if nodeid(npre)  == GLYPH_NODE and
         nodeid(npost) == GLYPH_NODE then
        if npre.char and npost.char then -- ligature between glyphs
          local replacement_pre  = machine:encode(utf8char(npre.char))
          local replacement_post = machine:encode(utf8char(npost.char))
          insert_encoded(head,  npre, replacement_pre)
          insert_encoded(head, npost, replacement_post)
        else -- hlists or whatever
          -- pass
          --noderemove(head, npre)
          --noderemove(head, npost)
        end
      end
      noderemove(head, n)
    elseif nid == HLIST_NODE or nid == VLIST_NODE then
      if nodelength(n.list) > 0 then
        n.list = aux(n.list, recurse + 1)
      end
```

```
--      else
--        -- TODO other node types
--        print(n)
      end
    end
    nodeslide(head)
    return head
  end -- callback auxiliary

  --- Context requires
  ---  × argument shuffling; a properly registered "action" gets the
  ---    head passed as its third argument
  ---  × hacking our way around the coupling of pre_linebreak_filter
  ---    and hpack_filter; background:
  ---    http://www.ntg.nl/pipermail/ntg-context/2012/067779.html
  local cbk = function (a, _, c)
    local head
    current_space_node = generate_space ()
    mod_5              = 0
    if format_is_context == true then
      head = c
      local cbk_env = get_debug_info(4) -- no getenv in lua 5.2
      --inspect(cbk_env)
      if cbk_env.func == nodes.processors.pre_linebreak_filter then
        -- how weird is that?
        return aux(head)
      end
      return head
    end
    head = a
    return aux(head)
  end

  if format_is_context then
    local cbk_id = "enigma_" .. name
    enigma.callbacks[name] = nodesinstallattributehandler{
      name      = cbk_id,
      namespace = thirddata.enigma,
      processor = cbk,
    }
    local cbk_location = "thirddata.enigma.callbacks." .. name
    nodestasksappendaction("processors",
                           --"characters",
                           --"finalizers",
                           --- this one is tagged "for users"
                           --- (cf. node-tsk.lua)
                           "before",
                           cbk_location)
    nodestasksdisableaction("processors", cbk_location)
  else
    enigma.callbacks[name] = cbk
  end
end
```

Enigma machines can be copied and derived from one another at will, cf. the \defineenigma on *page 17*. Two helper functions residing inside the *third-data.enigma* namespace take care of these actions: save_raw_args and retrieve_raw_args. As soon as a machine is defined, we store its parsed options inside the table *configurations* for later reference. For further details on the machine derivation mechanism see *page 18*.

```
local configurations = { }

local save_raw_args = function (conf, name)
  local current = configurations[name] or { }
  for k, v in next, conf do
    current[k] = v
  end
  configurations[name] = current
end

local retrieve_raw_args = function (name)
  local cn = configurations[name]
  return cn and tablecopy(cn) or { }
end

enigma.save_raw_args     = save_raw_args
enigma.retrieve_raw_args = retrieve_raw_args
```

The function new_machine instantiates a table containing the complete specification of a workable *Enigma* machine and other metadata. The result is intended to be handed over to the callback creation mechanism (new_callback). However, the arguments table is usally stored away in the *thirddata.enigma* namespace anyway (save_raw_args), so that the specification of any machine can be inherited by some new setup later on.

```
local new_machine = function (name)
  local args = configurations[name]
  --table.print(configurations)
  verbose_level = args and args.verbose or verbose_level
  local machine = new(name, args)
  return machine
end

enigma.new_machine  = new_machine
enigma.new_callback = new_callback

-- vim:ft=lua:sw=2:ts=2:tw=71:expandtab
```

SCRIPTING

mtx-t-enigma.lua

```
--
-------------------------------------------------------------------------------
--         FILE:  mtx-t-enigma.lua
--        USAGE:  mtxrun --script enigma --setup="s" --text="t"
--  DESCRIPTION:  context script interface for the Enigma module
-- REQUIREMENTS:  latest ConTeXt MkIV
--       AUTHOR:  Philipp Gesang (Phg), <gesang@stud.uni-heidelberg.de>
--      CREATED:  2013-03-28 02:14:05+0100
-------------------------------------------------------------------------------
--

environment.loadluafile("enigma")

local iowrite = io.write

local helpinfo = [[
==============================================================
    The Enigma module, command line interface.
    © 2012--2013 Philipp Gesang. License: 2-clause BSD.
    Home: <https://bitbucket.org/phg/enigma/>
==============================================================

USAGE:

    mtxrun --script enigma --setup="settings" --text="text"
           --verbose=int

    where the settings are to be specified as a comma-delimited
    conjunction of "key=value" statements, and "text" refers to
    the text to be encoded. Note that due to the involutory
    design of the enigma cipher, the text can be both plaintext
    and ciphertext.

==============================================================
]]

local application = logs.application {
    name     = "mtx-t-enigma",
    banner   = "The Enigma for ConTeXt, hg-rev 37+",
    helpinfo = helpinfo,
}

local ea = environment.argument
```

```
local setup, text = ea"setup" or ea"s",  ea"text" or ea"t"
local verbose     = ea"verbose" or ea"v"

local out = function (str)
  iowrite(str)
end

local machine_id = "external"
if setup and text then
  local args    = enigma.parse_args(setup)
  if not args then
    application.help()
    iowrite"\n\n[Error] Could not process enigma setup!\n\n"
  end
  enigma.save_raw_args(args, machine_id)
  --local machine = enigma.new_machine(enigma.parse_args(setup))
  local machine = enigma.new_machine(machine_id)
  --machine.name  = machine_id
  local result  = machine:encode_string(text)
  if result then
    out(result)
  else
    application.help()
  end
else
    application.help()
end
```

REGISTER

COLOPHON

Typeset using *ConTEXt* and *LuaTEX*.

Source code pretty printing by the *t-vim* module.

This documentation was generated from
sources by the *I Can Haz Docz* script.