Das

# Optical-Formula-Recognition (OFR) System

der

# Universität des Saarlandes

Dokumentation

# Inhaltsverzeichnis


| | |
|---|---|
| Systemstruktur (Module, Verbindungen) | 1 |
| Verzeichnisstruktur | 2 |
| Konfiguration des Systems | 3 |
| Übersetzen des Laufzeitsystems | 4 |
| Starten des Laufzeitsystems | 4 |
| InterProzeß-Kommunikation (IPC) | 5 |
| Schnittstellen | 6 |
| Datentypen & Dateiformate | 7 |

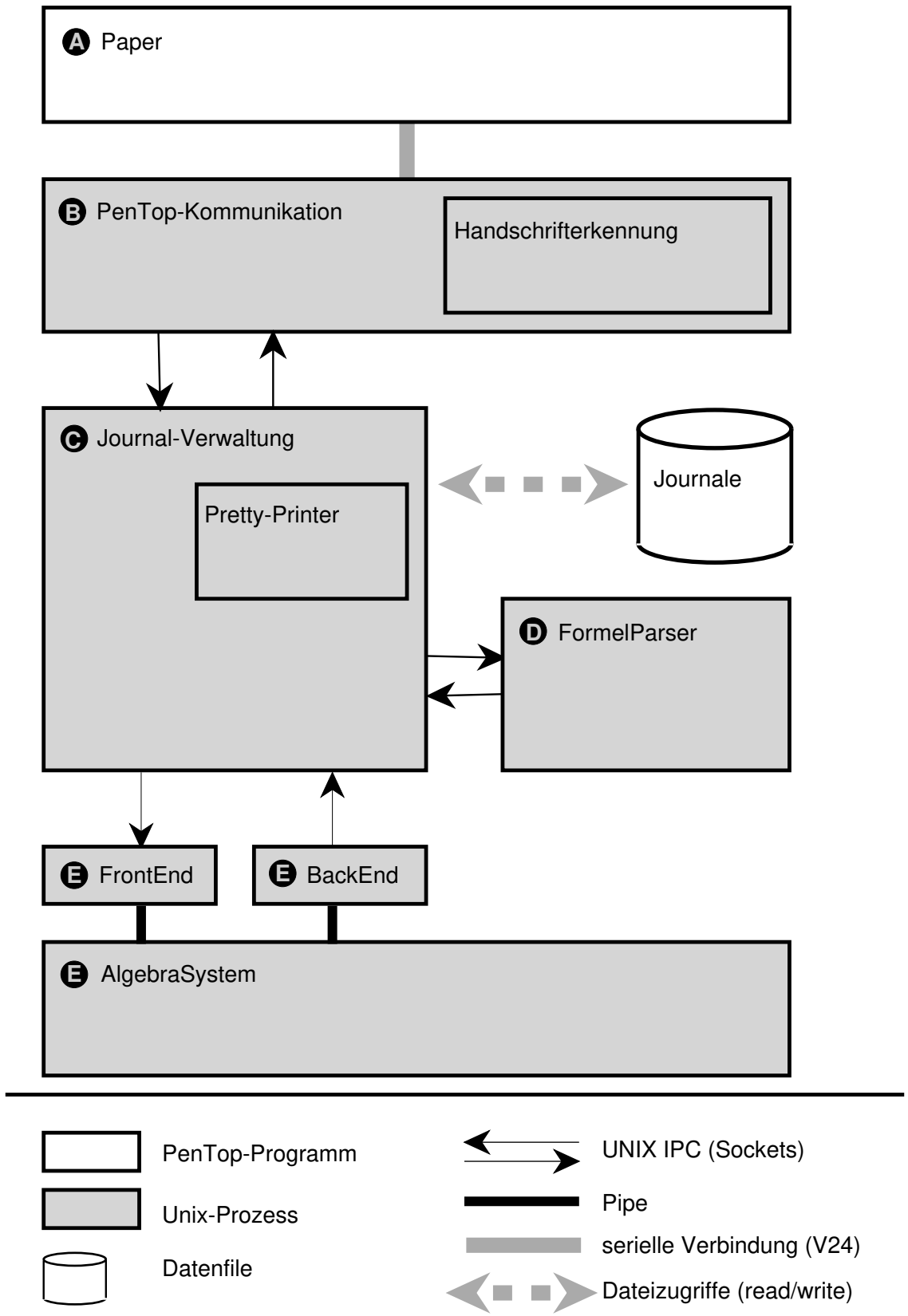



Bild 1 Überblick über die Module

# Systemstruktur (Module, Verbindungen)

Das OFR-System der Universität des Saarlandes besteht aus mehreren Software-Komponenten, über die Bild 1 einen Überblick liefert. Sie sollen im folgenden benannt und ihre Funktion kurz vorgestellt werden.

**PAPER.EXE**

ist die Software-Komponente auf PenTop-Seite. **PAPER** sammelt die vom Stift gelieferten Rohdaten (Koordinatenfolgen), führt bereits eine erste Interpretation durch und liefert die Daten an die Workstation.

**Erkennen**

erfüllt zwei Aufgaben: es initiiert und unterhält die (serielle) Verbindung zum PenTop, verteilt die eingehenden Daten an die Erkennung (Vektordaten) und ans Journal (Befehle) und leitet die Ausgaben des Journals an den PenTop weiter. Zum zweiten enthält Erkennen das Neuronale Netz, das die Mustererkennung durchführt. Es sammelt die eingehenden Vektordaten und 'übersetzt' sie in Einzelzeichen

**Journal**

koordiniert die Arbeit von Erkennen/Parser/PrettyPrinter und Algebrasystem. Es sammelt die erkannten Zeichen von Erkennen, übergibt dem Parser alle Zeichen einer Formel zur syntaktischen Auswertung, nimmt die Ergebnisse des Parsers entgegen, gibt sie an das Algebrasystem weiter und mit Hilfe des PrettyPrinters aus. außerdem empfängt es die Antworten des Algebrasystems und gibt auch diese mit dem PrettyPrinter aus.

**Parser**

analysiert die zweidimensionale Struktur einer Formel und erzeugt daraus einen Syntaxbaum

**FrontEnd | Algebrasystem | BackEnd**

sind genau in dieser Form zu einer UNIX-pipe verknüpft. Da die bislang verwendeten AlgebraSysteme nicht über eine Schnittstelle zur IPC verfügen, sondern lediglich über stdin/stdout bedient werden können, sorgen FrontEnd und BackEnd für die Umwandlung IPC->stdin...stdout->IPC

In Bild 1 nicht erwähnt werden die folgenden beiden Prozesse:

**Butler**

ist eine Art grafischer Oberfläche des OFR-Systems und übernimmt folgende Aufgaben:

- grafische Konfiguration des OFR-Systems (nur teilweise implementiert)
- Start aller übrigen Prozesse des OFR-Systems
- Vermittlerrolle beim Aufbau der Verbindungen zwischen den Prozessen (siehe IPC)
- Speichern der Prozessnummern, um ein gezieltes Abbrechen des Systems zu erlauben

**xpli** (XWindows-PaperLike-Interface)
ist eine (optische) Kopie der Arbeitsoberfläche auf dem PenTop und macht die dortigen Vorgänge auch auf der Workstation sichtbar.

# Verzeichnisstruktur

Alle Dateien, die zum OCR-System gehören, also sowohl die zur Erzeugung des Laufzeitsystems benötigten Quellen als auch die Binaries des Laufzeitsystems selbst, befinden sich in einem gesonderten Teilbaum des Dateisystems, der im weiteren als **Systemverzeichnis** bezeichnet wird. Außerdem werden im Folgenden alle Pfade so angegeben, als ob das Systemverzeichnis das aktuelle Verzeichnis (also ./ ) wäre.

## Prozesse

Die Aufteilung des Systems in Prozesse spiegelt sich auch in der Struktur des Systemverzeichnisses wieder. Jeder Prozess bildet ein eigenes Unterverzeichnis, in dem sich Sourcen, Makefiles, übersetzte Sourcen, sowie die Binaries der unterschiedlichen Versionen befinden.

Solche Prozeß-Unterverzeichnisse sind (vgl. mit Bild 1)

| | |
|---|---|
| ./Erkennen/ | PenTop-Kommunikation & Handschrifterkennung |
| ./Journal/ | die JournalVerwaltung |
| ./Parser/ | der FormelParser |
| ./Editor/ | ein Editor für die Grammatiken des Parsers |
| ./Butler/ | der Butler |
| ./AlgebraSysteme/ | Anbindung an die unterstützten AlgebraSysteme, u.a. |
| .../maple/ | an Maple |
| .../reduce/ | an Reduce |
| ./Xpli/ | X-Windows Oberfläche |

## Module

Desweiteren gibt es globale Module, die von mehreren Prozessen genutzt werden (z.B. die InterProzess-Kommunikation, IPC) und bei denen sichergestellt werden muß, daß alle Prozesse die gleiche Version benutzen. Solche Module sind in dem Verzeichnis

./module

zusammengefaßt, wobei wiederum jedes Modul ein eigenes Unterzeichnis belegt. Es sind dies im Einzelnen:

| | |
|---|---|
| ./module/chartab/ | Information zu den im System benutzten Zeichen |

| | |
|---|---|
| ./module/debug/ | Routinen zur einheitlichen Gestaltung von Debug-Ausgaben |
| ./module/rusage/ | Ermitteln der benutzten Systemresourcen |
| ./module/sockets/ | InterProzess-Kommunikation |
| ./module/ttyctrl/ | Handhabung der seriellen Schnittstelle |
| ./module/vfont | frei skalierbarer Vektorfont unter XWindows |

Prozessen und Modulen gemeinsam ist die Struktur der zugehörigen Unterverzeichnisse. Sie enthalten die folgenden Komponenten:

| | |
|---|---|
| include/ | enthält alle Header-Files die nur von dieser Teilkomponente benötigt werden. |
| *.c | die Quelltexte der (lokalen) Module, aus denen die Teilkomponente sich zusammensetzt. |
| *.l, *.y | bei Verwendung von lex & yacc sind auch Quellen mit diesen Endungen vorhanden |
| [mM]akefile | beschreibt die Übersetzung der Teilkomponente, siehe unten |
| obj/ | enthält die Übersetzungsprodukte (*.o-Files), die bei `make` anfallen |
| debug_obj/ | enthält die Übersetzungsprodukte (*.o-Files), die bei `make debug` anfallen |

Besteht ein Prozess oder Modul aus weiteren, komplexen Modulen, die nicht in einem quelltext zusammengefaßt werden, so existieren entsprechende Unterverzeichnisse. Bislang ist dies nur bei Journal der Fall:

| | |
|---|---|
| ./Journal/pretty/ | enthält den Pretty Printer |
| ./Journal/scanner/ | eine (veraltete) Komponente von Journal |

Auch solche UnterModule haben die gleiche Struktur wie Prozeß- bzw. Modul-Verzeichnisse.

## Globale Header

Ebenfalls im Systemverzeichnis befindet sich

./include

ein Verzeichnis, in dem die globalen Header-Dateien zusammengefaßt sind, also solche, die von mehr als einer Teilkomponente benötigt werden. Es sind dies z.B. die zu den globalen Modulen gehörigen Deklarationen sowie solche, die systemweit benötigte Parameter/Datentypen zur Verfügung stellen.

## Daten

Eine weitere Gruppe von Dateien stellen die externen Daten der einzelnen Prozesse dar. Diese sind unter dem Verzeichnis

./System/

zusammengefaßt und beinhalten:

| | |
|---|---|
| ./System/Fonts/ | Vektorzeichensätze, wie xpli sie verwendet |
| ./System/Images/ | Pixel-Bilder (butler/xpli) |
| ./System/Journale/ | Arbeitssitzungen, wie Journal sie speichern kann |
| ./System/LogFiles/ | Protokolle des letzten Programmlaufs, nach Prozessen getrennt |
| ./System/Ports/ | Informationen des butlers zu den einzelnen Verbindungen |
| ./System/Sounds/ | Sound-Dateien im .au-Format, |
| ./Erkennen/wt/ | Neuronale Netze der Erkennung (*sollten eigentlich unter ./System/Netze/ liegen).* |

### Diverses

Außerdem gibt es noch einige Dateien/Verzeichnisse, die keinem bestimmten Schema entsprechen:

| | |
|---|---|
| ./Dokumentation/ | die Dokumentation des Systems |
| ./config | Zusammenfassung der wichtigsten Systemparameter |
| ./config.* | rechnerabhängige Versionen von ./config |
| ./.make/ | enthält Include-Makefiles (siehe make) |
| ./Makefile | d a s Makefile zur Übersetzung des Systems |
| ./OFR* | das Shell-Script zum Start des OFR-Systems |
| ./tools/ | diverse Hilfsprogramme |

# Konfiguration des Systems

## Die Datei ./config

In dem csh-Script

./config (bzw. ./config.<*hostname*>)

sind alle bedeutenden Parameter des OCR-Systems in Form von Environment–Variablen zusammengefaßt. Je nach dem Zeitpunkt zu dem ein Parameter ausgewertet wird, lassen sich die 4 folgenden Kategorien unterscheiden:

ConfigZeit

Parameter dieser Kategorie werden nur im ./config-File benutzt, also bei dessen Ausführung ausgewertet. Um Änderungen wirksam zu machen, muß mit `source config` das config-File erneut eingelesen werden, d.h. ein Setzen des neuen Wertes mit setenv genügt **nicht**, da andere Variablenwerte sich auf diesen Wert beziehen.

wichtige ConfigZeit-Parameter sind MONTAGEDIR und KUNDE (s.u.)

CompileZeit
die entsprechende Variable wird mit der Direktive –D$(VARIABLE) an den C–Präprozessor weitergegeben. Dadurch können mittels bedingter Compilierung Programmteile je nach Parameterwert weggelassen/eingefügt werden. Um der Syntax des C-Präprozessors zu genügen, muß der Wert einer solchen Variable stets ein gültiger C-Bezeichner sein.
CompileZeit-Parameter sind vorallem MACHINE, INPUT und ALGSYS (s.u.)

StartZeit
solche Variablen werden bei jedem Start des Systems abgefragt. Änderungen werden erst bei einem erneuten Start wirksam.

LaufZeit
diese Kategorie ist leider nur hypothetischer Art. LaufZeit-Parameter sollen zur Laufzeit ständig neu ausgewertet. werden. Da jedoch Änderungen der Environment-Variablen nicht an Child-Prozesse weitergegeben werden, werden Änderungen nicht wirksam

**Allgemein zur Installation**

```
setenv MONTAGEDIR          /usr/users/montage
```

$(MONTAGE) gibt das Systemverzeichnis an, also den oben schon bezeichneten Unterbaum des Dateisystems, in dem sich das komplette OFR-System befindet.
Kategorie: CompileZeit, ConfigZeit, StartZeit

```
setenv MACHINE             SUN
```

$(MACHINE) beschreibt den Typ von Hardware/BS-Konstellation, auf dem das OFR-System installiert wurde. Definierte Werte sind bislang nur SUN, NeXT und LINUX
Kategorie: CompileZeit, ConfigZeit(?)

```
setenv VERSION             MTEST1_
```

(vormals $(KUNDE)) Bestimmt zur CompileZeit, welche Version übersetzt wird und zur ConfigZeit, welche Version gestartet wird.
Kategorie: CompileZeit, ConfigZeit, StartZeit(?)

**Eingabegerät**

```
setenv INPUT               NCR
```

$(INPUT) beschreibt das zur Eingabe benutzte Gerät. Möglich sind bisher folgende Werte:

EIGENBAU, ARISTO, GRID, MOMENTA und NCR.
Da sich die letzten drei aus Sicht des OFR-Systems nicht unterschieden, wird für alle drei NCR angegeben.
Kategorie: CompileZeit, ConfigZeit(?)

```
setenv COM                  /dev/ttyb
```

(Serielle) Schnittstelle, unter der das Eingabegerät angesprochen wird.
Kategorie: ConfigZeit, StartZeit(?)

```
setenv HANDSHAKE            -
```

Ein Parameter für Erkennen. Legt fest, ob auf der seriellen Schnittstelle der RTS/CTS-Handshake benutzt wird(+) oder nicht (-)
Kategorie: ConfigZeit, StartZeit(?)

**System-Version**

```
setenv ALGSYS               MAPLE
```

$(ALGSYS) legt fest, welches Algebrasystem verwendet wird.
Definierte Werte sind bislang MAPLE und REDUCE
Kategorie: CompileZeit( von Journal und Parser), ConfigZeit, StartZeit(?)

```
setenv BENUTZER             joha
```

$(BENUTZER) spezifiziert das neuronale Netz, das Verwendung findet. Eingetragen werden kann hier jeder Benutzer, für den ein Netz vorliegt oder das MultiUserNetz (MULTI). Gültige Namen sind je nach System unterschiedlich.
Kategorie: ConfigZeit, StartZeit

**Interne Parameter**

Gemeint sind damit all die Parameter, deren Sinn und Zweck dem Benutzer des OFR-Systems nicht direkt ersichtlich ist (um die er sich i.A. auch nicht kümmern muß)

```
setenv MAKE_DIR             $MONTAGEDIR/.make
```

Die Übersetzung der OFR-Sourcen erfordert spezielle Makefiles, die sich in einem eigenen Verzeichnis befinden (siehe Make und Makefiles). Normalerweise befindet sich dieses Verzeichnis innerhalb von $(MONTAGE).
Kategorie: CompileZeit

```
setenv JOURNALE             $MONTAGEDIR/System/Journale
```

Unter diesem Verzeichnis werden die Journale von Journal abgelegt bzw. geladen.

Natürlich kann man diesen Pfad den Wünschen eines Benutzers anpassen.
Kategorie: LaufZeit(?)

```
setenv DIGIT_PORT_NUM_1          dispatcher_port1
setenv DIGIT_PORT_NUM_2          dispatcher_port_2
setenv NEURONET_PORT_NUM         server_port_num2
setenv SERVER_PORT_NUM1          dispatcher_port1
setenv SERVER_PORT_NUM2          server_port_num2
setenv DISPATCHER_PORT_2         dispatcher_port_2
setenv PARSER_JOURNAL_PORT       parser_journal_port
setenv JOURNAL_TO_ALGSYS_PORT    journal_algsys_port
setenv ALGSYS_TO_JOURNAL_PORT    algsys_journal_port
setenv XPLI_PORT                 xpli_port
```

diese Parameter, die alle das Infix PORT enthalten, geben den Verbindungen zwischen den Modulen Namen. Butler legt ggf. zur Laufzeit im Verzeichnis ./System/Ports eine Datei gleichen Namens an, die die Telefonnummer der betreffenden Verbindung enthält (siehe IPC und Verbindungsaufbau).
Kategorie: LaufZeit(?)

```
setenv ERKENNEN_START     "$MONTAGEDIR/Erkennen/${KUNDE}Er
                          kennen -g -l ${COM} -
                          b${BAUDRATE} -h-"
setenv JOURNAL_START      "$MONTAGEDIR/Journal/Journal-
                          ${KUNDE}Version.debug -r -aon"
setenv PARSER_START       "$MONTAGEDIR/Parser/Parser-
                          ${KUNDE}Version.debug -d+ahqpo"
setenv ALGEBRA_START      "/usr/cmd/maple -q"
setenv FRONT_START        "$MONTAGEDIR/AlgebraSystem/maple
                          /${KUNDE}mf"
setenv BACK_START         "$MONTAGEDIR/AlgebraSystem/maple
                          /${KUNDE}mb"
setenv XPLI_START         "$MONTAGEDIR/Xpli/Xpli"
setenv KILLXDVI           ${MONTAGEDIR}/tools/killxdvi
```

Die Parameter mit dem Suffix _START definieren die Kommandozeile, mit der Butler die entsprechenden Teilprozesse startet. Auch KILLXDVI definiert eine Kommandozeile, die jedoch nicht von Butler sondern von Journal aufgerufen wird und den XDVI-Treiber beendet.
Kategorie: LaufZeit(?)

Achtung: *die Parameter ALGEBRA_START, FRONT_START und BACK_START sind in config abhängig von $(ALGSYS) definiert !!*

```
setenv reduce             /vol/reduce3.4/sun4
```

```
setenv reduce_init        /vol/Schwarz/reduce
```

Zwei Pfade für den Gebrauch von Reduce. $(reduce) gibt das System-Verzeichnis von Reduce an. reduce selbst erwartet diese Variable. $(reduce_init) spezifiziert den Namen von recude selbst.

# Übersetzen des Laufzeitsystems

## Make und Makefiles

Genauso wie durch den Einsatz von globalen Modulen den Parallelen zwischen den Laufzeitaufgaben der Prozesse Rechnung getragen wird, werden Analogien zwischen den Übersetzungen der Teilkomponenten berücksichtigt. So ist es z.B. wünschenswert, beim Erzeugen aller Module global einheitliche Compiler-Optionen verwenden zu können. Man erreicht dies, indem man auch die Makefiles in Module zerlegt. UNIX-make erlaubt die Einbindung solcher "Makefile-Module" mit der Direktive include ( vgl. make (1) )
Im System werden drei solche "Makefile-Module" verwendet, die sich im Verzeichnis . /.make
befinden:

config.make
definiert einige Makros, die globale Optionen festlegen (verwendeter Compiler, CompilerOptionen, Pfade unter denen sich die globalen Module finden etc.).

common.make
beinhaltet alle Übersetzungsregeln und sorgt insbes. für

- das Sammeln der übersetzten C-Quellen in einem eigenen Verzeichnis .../obj/,
- Einfügen eines weiteren Pass zwischen Präprozessor und Compiler (undbg, siehe dort)
- Definition weiterer Aktionen neben der Übersetzung eines Prozesses. So es z.B. möglich mit make clean alle bei der Übersetzung erzeugten Files zu entfernen.

module.make
sorgt fuer die Übersetzung der globalen Module

Neben diesen drei Makefiles besitzt jede Teilkomponente des Systems ein eigenes Makefile, das die zur Erzeugung des Moduls erforderlichen Aktionen beschreibt. Durch die Verwendung der obigen Includes weicht der Inhalt eines solchen Makefiles vom allg. bekannten Schema ab. Deswegen hier ein kommentiertes Beispiel zum (hoffentlich) besseren Verständnis der angewendeten Prinzipien (bedeutende Variablen sind **FETT** gedruckt)

```
# @(#) Makefile fuer den Parser
#

NAME = Parser-$(KUNDE)Version
```

Ein Name für den Prozess wird definiert. $(KUNDE) ist die (weiter oben erklärte) Environment-Variable, die die zu erzeugende Version beschreibt

```
include $(MAKE_DIR)/config.make
```

Mit dieser Anweisung wird das erwähnte Unter-Makefile config.make eingebunden. Es enthält die im System bedeutenden Pfade.

Der weitaus größte Teil der Übersetzungsregeln wird im Unter-Makefile common.make zusammengefaßt. Um dieses File von allen Makefiles aus nutzen zu können, wird es mit Hilfe zahlreicher Variablen parametrisiert, denen natürlich vor dem Einbinden von common.make entsprechende Werte zugewiesen werden müssen:

```
CFILES   = \
   token.c journal.c tools.c setop.c grammar.c \
   neighbor.c parser.c main.c consistency.c \
   draw.c decode.c treeview.c debugFlags.c
```

Die Variable CFILES z.B. enthält alle C-Module, aus denen der beschriebene Prozess besteht und die nur zu diesem Prozess gehören (also nicht etwa die oben erwähnten globalen Module)

```
CFLAGS    = \
   -I$(INCLUDE) \
   -D$(ALGSYS)
```

CFLAGS beschreibt die Compileroptionen, die bei der Übersetzung eines jeden in CFILES angegebenen Moduls gesetzt werden sollen

```
OTHER_OFILES = \
   $(SOCKET_OBJ) \
   $(PLITALK_OBJ) \
   $(RUSAGE_OBJ) \
   $(DEBUG_OBJ) \
   $(MODULE)/chartab/obj/chartab.o
```

OTHER_OFILES spezifiziert die globalen Module, die vom Prozess benötigt werden. Die hier benutzten Variablen $(SOCKET_OBJ) wurden z.B. in config.make definiert.

```
LIBS = -lm
```

Sollen Bibliotheken zum Prozess hinzugelinkt werden, können diese in der Variablen

LIBS zusammengefaßt werden. Dabei muß für jede einzelne Bibliothek die vom Linker ld vorgegebene Syntax -l*libraryname* verwendet werden.

```
include $(MAKE_DIR)/common.make
```

Hiermit wird endlich common.make eingebunden. common.make definiert im wesentlichen folgende Zusammenhänge:

- alle *.l-Files werden mit lex in *.c-Files übersetzt
- alle *.y-Files werden mit yacc in *.c-Files übersetzt
- alle *.c-Files (also die aus CFILES und die bei der Übersetzung der *.l/*.y-Files erzeugten) werden zunächst mit undbg zwischenübersetzt und dann mit $(CC) in *.o-Files übersetzt und in ./obj/ abgelegt
- alle *.o-Files (also die in ./obj/ sowie die in OTHER_OFILES werden zusammen mit den LIBS zu $(NAME) zusammengelinkt

```
# Abhaengigkeiten

grammar.o:      include/grammar/defs.h

decode.o:       include/grammar/defs.h
```

Alle Abhängigkeiten, die zwischen den Quellen des Prozesses bestehen, können hier angegeben werden und werden dann von make wie gewohnt automatisch überprüft.

```
# globale Module

include $(MAKE_DIR)/module.make
```

Wenn module.make ebenfalls eingebunden wird, so wird bei jeder Übersetzung des Prozesses dafür gesorgt, daß auch die globalen Module übersetzt werden, falls nötig. Das spart u.U. einen weiteren Aufruf von make.

## Das Komplettsystem

Soll das ganze System (z.B. nach der Installation einer neuen Version oder Änderungen der CompileZeit-Parameter) übersetzt werden, so empfiehlt sich folgendes Vorgehen:

```
ocr%beispielhost> cd <SystemVerzeichnis>
```

wechselt in das SystemVerzeichnis.

```
ocr%beispielhost> source ./config
```

stellt sicher, daß die korrekten Parameterwerte gesetzt sind.

```
ocr%beispielhost> make clean
```

beseitigt die Resultate vorheriger Übersetzungsvorgänge (dies ist insbesondere dann nötig, wenn CompileZeit-Parameter geändert wurden, da UNIX-make nur auf Dateiänderungen, nicht aber auf Änderungen der Environement-Variablen reagiert)

```
ocr%beispielhost> make
```

startet den eigentlichen Übersetzungsvorgang. Es werden automatisch alle Komponenten des Systems übersetzt.

# Starten des Systems

startet den eigentlichen Übersetzungsvorgang. Es werden automatisch alle

# InterProzeß-Kommunikation (IPC)

## Sockets

Sockets sind eine von Unix bereitgestellte Möglichkeit zur Interprozess-Kommunikation. Die Programmierung von Sockets ist ein breites und oft besprochenes Thema. Für erste Informationen empfiehlt sich ein Blick ins Unterverzeichnis

```
./Dokumentation/ipctut
```

in dem sich ein Tutorial zur Interprozess-Kommunikation mit Beispielprogrammen befindet. Weitere Details finden sich natürlich in den entsprechenden UNIX-manpages (z.B. mit `apropos socket`).

Im folgenden werden nur die Besonderheiten beschrieben, die sich bei der Verwendung von Sockets innerhalb des OFR-System ergeben.

## Verbindungsaufbau

Soll eine Verbindung über Sockets aufgebaut werden, so sind die Rollen klar verteilt: Es gibt einen Server, der einen Anschluß zur Verfügung stellt und einen Client, der dieses Kommunikationsangebot nutzt. Sobald der Server seinen Anschluß einrichtet, muß er diesem eine im System (Rechner) eindeutige (Telefon-)Nummer geben oder sich vom Betriebssystem eine solche eindeutige Nummer zuweisen lassen. Dem Client muß diese Nummer bekannt sein, da er mit dieser Nummer dem Betriebssystem mitteilt, mit welchem Server er kommunizieren möchte.
Vergibt ein Server seinem Anschluß eine feste (von ihm bestimmte) Nummer, so läuft er Gefahr, daß diese Nummer bei einer NeuInstallation auf einem ihm unbekannten System bereits vergeben ist, er seinen Anschluß also nicht einrichten kann. Läßt er sich

dagegen vom Betriebssytem eine eindeutige Nummer zuweisen, vermeidet er zwar solche Kollisionen, steht aber vor dem neuen Problem, diese Nummer dem Client mitteilen zu müssen.
Auf fast allen Unix-Systemen gibt es deshalb ein einfaches Telefonbuch, /etc/services, das einem Server (vielmehr seiner eindeutigen Socket-Nummer) einen Namen zuordnet. Leider unterstützt Unix nur eine Abfrage dieses Telefonbuchs, nicht aber eine Erweiterung/Änderung der Einträge zur Laufzeit eines Prozesses. Vielmehr obliegt die Verwaltung der Datei /etc/services dem SystemAdministrator.

## Butler

Darum gibt es im OFR-System ein eigenes, erweiterbares Telefonbuch, den Butler. Auch der Butler ist nur über IPC erreichbar. Deswegen wird ihm bei der Installation über /etc/services eine Nummer unter dem Namen 'butler' vergeben.

Jeder Prozess, der die Rolle eines Servers übernimmt, richtet zunächst seinen Anschluß ein und läßt sich vom Betriebssystem eine eindeutige Nummer zuweisen. Diesen Anschluß melden er beim Butler an. Durch Abfrage der /etc/services kann er dazu die Nummer des Butlers erfragen und (als Client) eine Verbindung zu diesem aufbauen. In einem definierten Protokoll teilt er dem Bulter Identität (einen namensgebenden String), Prozess-ID und die für seinen Anschluß vergebene Nummer mit.

Jeder Prozess, der die Rolle eines Client übernimmt, kann sich nun ebenfalls durch Abfrage von /etc/services und Aufbau einer entsprechenden Verbindung an den Butler wenden. Der Client kann nun durch Angabe der Identität(des Namens) seines Servers dessen Anschlußnummer beim Butler nachfragen.

Diese Vorgehensweise wurde gewählt, da
- sie die Vorteile der dynamischen Vergabe der Anschlußnummern zur Laufzeit nutzt
- sie nur einen Eintrag in /etc/services verlangt, was die Zahl der fest vergebenenen Nummern und somit die Zahl der denkbaren Kollisionen verringert
- dem Butler weitere Aufgaben zugeordnet werden können

Da alle Prozesse, die IPC benutzen, die verwendeten Protokolle kennen müssen, wurden die entsprechenden Funktionen in das **IPC-Modul** (proc_com.o) integriert.

## Datentransport

Um die Übertragung typisierter Daten zwischen Prozessen zu vereinfachen, werden vom Modul proc_com.o zwei Funktionen bereitgestellt, die mit einem Aufruf eine variable Anzahl von Parametern versenden/empfangen können (ähnlich den printf()/scanf()-Fkt. der C-Library, siehe auch vararg(3)) und dabei auf eine typgerechte .Übertragung achten.

```
int sendf(int socket, char *typinfo ...);
int receivef(int socket, char *typinfo ...);
```

sendf() schreibt Daten in den angegebenen Socket socket, während receivef() Daten daraus liest. Der 2. Parameter typinfo beschreibt Zahl und Typ der weiteren Parameter. sendf() erwartet für jede Typbeschreibung in typinfo einen weiteren Parameter, der Daten dieses Typs liefert (bei skalaren Typen einen Wert, bei Vektoren einen Zeiger auf das erste Element), wohingegen receivef() für jede Typbeschreibung die Adresse eines Speicherbereichs erwartet, der Daten dieses Typs aufnehmen kann.
Bislang können folgende Datentypen übertragen werden:

| Typ | Bezeichn. | Bedeutung | sendf() | receivef() |
|---|---|---|---|---|
| Char | c | ein ASCII-Zeichen | char | char * |
| Word | w | eine 16-Bit Ganzzahl | short | short * |
| Int | i | eine 32-Bit Ganzzahl | int | int * |
| Long | l | eine 32-Bit-Ganzzahl | long | long * |
| String | s | ein C-String (\0-terminierte Zeichenfolge) | char * | char * |

Das Konzept zur Kodierung der TypInformation der variablen Parameter ist dem Formatbezeichner der printf()-Fkt. abgeschaut. Zunächst besteht typinfo aus einer Folge von ASCII-Zeichen, wobei jedes Zeichen außer '%' für sich selbst (und damit für den Datentyp **Char)** steht. Ein solches Zeichen wird ohne weitere Behandlung verschickt (sendf()) bzw. erwartet und ignoriert (receivef()). Ein '%' dagegen leitet die Beschreibung eines weiteren Parameters ein, die dem Muster

```
%[count]typechar
```

folgt. Dabei ist typechar einer der TypBezeichner aus Liste xxx. Wird der optionale Parameter count angegeben, so erwarten die Funktionen einen count Elemente umfassenden Vektor vom beschriebenen Typ. count kann entweder als fester Wert (pos. ganze Zahl in ASCII-Dartstellung) angegeben werden oder als '*'. Letzteres bedeutet, daß der Vektor von dynamischer Größe ist und ein weiterer Parameter (noch vor dem Vektor) die zu verwendende Vektorgröße angibt.

Einige Beispiele:

```
sendf(port, "A%i%s", (int) nummer, (char *) name);
```

verschickt ein 'A', eine ganze Zahl und einen String über port.

```
receivef(port, "%c%i%s", &zeichen, &nummer, buffer);
```

kann ebendiese Daten lesen (zeichen == 'A' ...)

```
int liste_out[20];
sendf(port, "%10i%*c", liste_out, 5, name);
```

verschickt die ersten 10 ints aus dem Array liste[] und anschließend die ersten 5 Zeichen des Strings name.

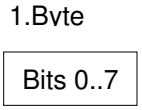
```
int liste_in[10];
receivef(port, "%*i%5c", 10, liste_in, buffer);
```

liest diese Daten.

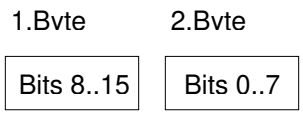
## Externe Darstellung

Ein weiterer, oben bereits erwähnter Zweck der Funktionen sendf()/receivef() ist es, die (hardwareabhängigen) Details der binären Darstellung von Daten zu verstecken und eine typsichere Übertragung auch zwischen verschiedenen Rechnerfamilien zu garantieren. sendf() und receivef() definieren zu diesem Zweck eine eigene binäre Darstellung der gebräuchlichsten C-Standardtypen, die im weiteren **externe Darstellung** genannt wird:

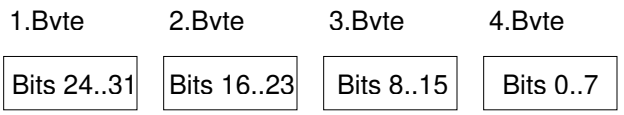
ASCII-Zeichen



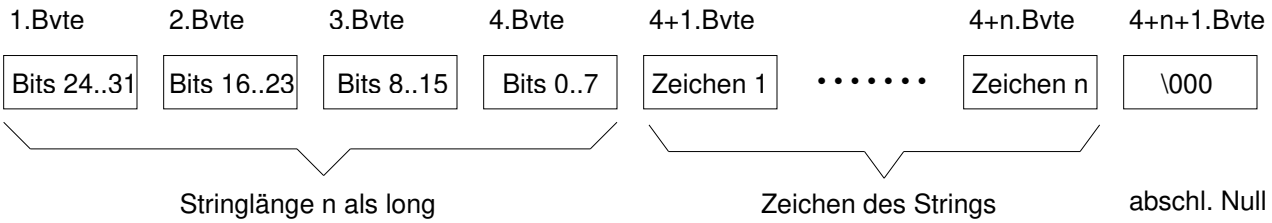

16-Bit Ganzzahl




32-Bit Ganzzahl




'\0'-terminierter C-String




Diese Darstellung entspricht gerade der internen Darstellung der SUN-SPARC-Familie bzw. der der Motorola-CPUs (68000..68040).

sendf() wandelt seine Parameter von der internen Darstellung des Wirtsrechners um in diese externe Darstellung während receivef() umgekehrt Daten in externer Darstellung umwandelt in die interne Darstellung seines Wirtsrechners.

***Bug:*** *in der (nicht freigegebenen) Linux-Version des OFR-Systems wird stillschweigend eine andere externe Darstellung verwendet, die der internen Darstellung der INTEL-(iX86–)Prozessoren entspricht. Solange keine Kommunikation zwischen INTEL- und Motorola-/Sparc-CPUs nötig ist, bereitet dies keine Probleme.*

# Schnittstellen

## PenTop-Software (Paper -> Erkennen)

Das Datenformat der Verbindung zwischen PenTop und Workstation, also zwischen **PAPER** und **Erkennen** wurde bestimmt von dem anfänglichen Wunsch, statt eines PenTop auch ein Digitalisiertablett anschließen zu können. Digitalisiertabletts verschicken normalerweise kleine DatenPakete fixer Länge, von denen jedes die Koordinaten eines Punktes in der Ebene und evtl. einige Flagbits enthält. Dabei kann man zuweilen wählen, wie die Koordinaten dargestellt werden, ob als ASCII-Ziffernfolge, in BCD-Codierung oder binär.

Sowohl die PenTop-Software als auch das empfangende Modul auf Workstationseite unterstützen zwei Formate, ein (umfangreicheres) ASCII-Format, das dem Standard-format des anfänglich benutzten Digitalisiertabletts (ARISTO) entspricht (und in den Geräteunterlagen des Herstellers beschrieben wird), sowie ein kompakteres binäres Format. Welches Format benutzt wird, hängt vom Parameter INPUT ab (siehe Konfi-guration des Systems) Für INPUT == ARISTO verwenden beide Module die ASCII-Darstellung. Für jeden anderen Wert von INPUT dagegen findet das kompaktere binäre Format Verwendung, das im folgeden beschrieben wird.

Die Daten werden in Paketen verschickt, die aus 6 Byte bestehen und folgenden Aufbau haben:

1. Byte

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | x | FB_Formelende |
| | | | | | | x | | FB_VorlFormelende |
| | | | | | x | | | FB_Radiergummi |
| | | | | x | | | | FB_EraserOff |
| | | | x | | | | | FB_PenUp |
| | | x | | | | | | FB_Command |
| | x | | | | | | | FB_Default |

1 Header-Sync-Bit

FB_formelende wird gesetzt, wenn die formelende-Taste am PenTop gedrückt wird.
FB_VorlFormelende wird gesetzt, sobald der Stift eine bestimmte Zeit abgehoben bleibt (also eine Schreibpause entsteht). **Erkennen** verwendet diese Information, um die Segmentierung der Einzelzeichen (siehe **Erkennen**) bei Schreibpausen nicht unnötig zu verzögern.
FB_Radiergummi gibt das verwendete Schreibwerkzeug an. Sobald der Benutzer den Radiergummi aufnimmt, bleibt dieses Bit gesetzt, bis er ihn wieder ablegt. D.h. alle Koordinaten, die während eines Radiervorgangs gesendet werden, haben FB_Radiergummi gesetzt. Eine gesonderte Mitteilung, daß das Schreibwerkzeug gewechselt wurde, erfolgt nicht.
FB_EraserOff ohne Bedeutung
FB_PenUp zeigt an, daß der Stift abgehoben ist (die Koordinaten im Paket sind dann je nach PenTop ungültig)
Ist FB_Command gesetzt, so enthält dieses Paket keine Koordinaten, sondern kodiert einen Befehl (also z.B. einen in PAPER gedrückten Button)

Byte 1 ist das einzige mit gesetztem 7. Bit. Dadurch ist es theoretisch möglich, bei Leitungsproblemen (verlorene Bytes) den Empfänger neu zu synchronisieren. Die aktuelle Software macht davon allerdings keinen Gebrauch.

2. Byte, das Tasten-, bzw. Befehlsbyte

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | k | k | k | k | k | k |

Bit 0-6 bezeichnen einen Befehl nach folgender Kodierung:

| | |
|---|---|
| ZURUECKBLAETTERN | 0x0041 |
| VORBLAETTERN | 0x0042 |
| SPRUNG_ANFANG | 0x0040 |
| SPRUNG_ENDE | 0x0043 |
| NEUES_BLATT | 0x0044 |
| SITZUNG_LADEN | 0x0045 |
| SITZUNG_SICHERN | 0x0046 |
| LINKS_SCHIEBEN | 0x004b |
| RECHTS_SCHIEBEN | 0x0048 |
| HOCH_SCHIEBEN | 0x0049 |
| RUNTER_SCHIEBEN | 0x004a |
| AUSSCHNEIDEN | 0x004c |
| EINKLEBEN | 0x004d |
| PROGRAMM_BEENDEN | 0x004f |
| ZOOM_IN | 0x0047 |
| ZOOM_OUT | 0x004e |

dieses 2. Byte ist nur von Bedeutung, wenn im Header-Byte das FB_Command-Bit gesetzt ist.

3. und 4. Byte
Diese beiden Bytes kodieren die X-Koordinate als positive Ganzzahl mit 14 Bits nach folgendem Schema:

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| 0 | x13 | x12 | x11 | x10 | x9 | x8 | x7 | 3. Byte |
| 0 | x6 | x5 | x4 | x3 | x2 | x1 | x0 | 4. Byte |

Mit den zur Verfügung stehenden 14 bits lassen sich Werte im Bereich 0..+16384 darstellen.

5. und 6. Byte kodieren die Y-Koordinate analog den Bytes 3 & 4.

Wenn ein AUSSCHNEIDEN oder EINKLEBEN Befehl gesendet werden soll, so werden zwei Pakete geschickt. Im ersten wird FB_Command gesetzt und die linke Obere Ecke des Rechtecks (also minimale X,Y-Koordinaten) gesendet, im 2. Paket dann die max. Koordinaten.

Einen nicht sehr eleganten Sonderfall stellen die nachträglich eingefügten Befehle SITZUNG_SICHERN und SITZUNG_LADEN dar. Der Benutzer sollte die Möglichkeit haben, den Dateinamen der zu ladenden Sitzung am PenTop einzugeben. Das machte aber den Versand von Zeichenketten nötig. Um die Änderungen an der Kommunikation zu minimieren, wurde für ein kleines Alphabet von 39 Zeichen (a-z, 0-9, ., __, ⊥ ) eine Gödel-Nummerierung eingeführt, die für Strings der Länge 5 Zahlen im Bereich $0..39^5-1 = 90224198$ liefert. Mit dem Wertebereich der beiden Koordinaten lassen sich aber solche max. 8-stelligen Zahlen leicht kodieren in der Form X*10000+Y. Mit diesem Verfahren ist es demnach möglich in einem Paket 5 Zeichen zu verschicken. SITZUNG_LADEN und SITZUNG_SICHERN verwenden 2 solche Pakete, um den Namen einer Datei zu kodieren. SITZUNG_LADEN verwendet zwei weitere Pakete, um auch noch einen Benutzernamen zu verschicken, den z.B. **Erkennen** benutzt, um das passende neuronale Netz zu laden

*siehe auch Quelltexte zu*

| | |
|---|---|
| digikoord_get() | in ./Erkennen/tty.c |
| buildname(), | |
| make_segment() | in ./Erkennen/digit.c |

## Erkennen -> PAPER

**PAPER** verschickt nicht nur Koordinaten, sondern versteht auch einige wenige Befehle, die zur Darstellung von Informationen auf dem PenTop dienen. Kodiert werden diese Befehle durch ein einleitendes ASCII-Zeichen, auf das ggf. weitere Parameter folgen:

| Zeichen | Bedeutung und evtl. weitere Parameter |
|---|---|
| R | Reset.<br>führt eine Initialisierung von PAPER durch (alle evtl. noch gepufferten Koordinaten sowie der sichtbare Zeichenbereich werden gelöscht, ein evtl. benutzter Radiergummi abgelegt)<br>keine Parameter |
| Q | Koordinaten-ReQuest<br>fordert PAPER auf, ein neues Datenpaket zu senden.<br>keine Parameter |
| D | skaliertes Zeichen Darstellen.<br>Zusätzliche Parameter sind Code des Zeichens und umfassendes Rechteck. Das angegebene Zeichen wird als Vektorgrafik so skaliert und ausgegeben, daß es das angegebene Rechteck genau ausfüllt.Die Parameter beanspruchen 9 weitere Bytes und werden folgendermaßen kodiert:<br><br>2. Byte<br><br>das darzustellende Zeichen (Bytewert == Zeichencode)<br><br>3. bis 10. Byte<br><br>umfassendes Rechteck im Format xmin, ymin, xmax, ymax. Jede Koordinate wird als positive 14-Bit-Ganzzahl übertragen, wozu pro Koordinate 2 Bytes nach folgendem Schema (wie oben) belegt werden:<br><br>7 6 5 4 3 2 1 0<br><br>0 x13 x12 x11 x10 x9 x8 x7 1. Byte<br>0 x6 x5 x4 x3 x2 x1 x0 2. Byte<br><br>Um sicherzustellen, daß nur positive Werte übertragen werden, wird jeder Koordinatenwert um 8192 erhöht, d.h. aus -300 wird 8192 - 300 = 7892. |
| H | Kopf(Header-)zeile einblenden.<br>Weitere Parameter sind die Koordinaten, an denen die Ausgabe erfolgen soll sowie der Text der Ausgabe. Das Format der Parameter |

ergibt sich aus folgendem printf()-FormatString:

"HX%04dY%04d%.80s"

wobei die beiden Integerwerte die X- und Y-Koordinate angeben (linke obere Ecke == (0,0)) und der Stringparameter den auszugebenden Text.

*siehe auch Quelltexte zu*
PostVonBifi() in ./Erkennen/comm.c

## Erkennen -> Journal

In dieser Richtung werden Befehle, die Erkennen von der PenTop-Software empfängt, an das Journal weitergeleitet

Allen Befehlen ist gemeinsam, daß zunächst eine den Befehl beschreibende ganze Zahl (int) gesendet wird. Folgende Befehls-Codes sind in

./include/codierung.h

definiert und (hoffentlich) selbsterklärend:

| | |
|---|---|
| AUSSCHNEIDEN | (*) |
| EINKLEBEN | (*) |
| FORMELENDE | |
| GLEICHUNG_LOESEN | |
| HOCH_SCHIEBEN | |
| LINKS_SCHIEBEN | |
| NEUES_BLATT | |
| PROGRAMM_BEENDEN | |
| RECHTS_SCHIEBEN | |
| RUNTER_SCHIEBEN | |
| SITZUNG_LADEN | (*) |
| SITZUNG_SICHERN | (*) |
| SPRUNG_ANFANG | |
| SPRUNG_ENDE | |
| VORBLAETTERN | |
| ZOOM_IN | |
| ZOOM_OUT | |
| ZURUECKBLAETTERN | |

Alle Befehle außer den mit (*) gekennzeichneten erfordern keine weiteren Parameter.

AUSSCHNEIDEN und EINKLEBEN werden gefolgt von der Definition eines Rechtecks bestehend aus den 4 Koordinaten xmin, ymin, xmax, ymax. Die Angabe der

Koordinaten erfolgt als ganze Zahl (int) im Koordinatensystem des PenTop, d.h. Translation und Skalierung wie Journal sie zur Verfügung stellt, werden nicht berücksichtigt.

TypSpezifizierer (siehe IPC/Datentransport):

```
sendf("%i%i%i%i", xmin, ymin, xmax, ymax);
```

SITZUNG_SICHERN hat als einzigen Parameter einen String, der einen Dateinamen repräsentiert.:

```
sendf("%s", sitzungs_name);
```

SITZUNG_LADEN wird von zwei String-Parametern gefolgt. Der erste String gibt den Namen des Benutzers, der zweite den Dateinamen der zu ladenden Arbeitssitzung an:

```
sendf("%s%s", benutzer_name, sitzungs_name);
```

*siehe auch Quelltexte zu*

| | |
|---|---|
| make_segment() | in ./Erkennen/digit.c |
| handleCommand() | in ./Journal/main.c |

*Format der erkannten Zeichen*

## Journal -> Erkennen

sowie alle Ausgaben, die auf dem PenTop erscheinen sollen, entgegengenommen, ins entsprechende Format konvertiert und an die PenTop-Software weitergeleitet.
Auch hier entscheidet zunächst eine ganze Zahl (Definitionen siehe ./include/codierung.h) über Anzahl und Bedeutung der weiteren Parameter:

MESSAGE leitet eine Meldung ein, die in der Statuszeile der PenTop-Software erscheinen soll. Auf den Befehlscode folgt der auszugebende String

Alle übrigen Codes werden als auszugebendes Vektorzeichen (gemäß der in ./include/codierung.h definierten Kodierung) interpretiert. Auf ein solches Zeichen muß das umfassende Rechteck in der bereits oben angegebenen Form xmin, ymin, xmax, ymax folgen. Auch hier gilt: alle Koordinaten sind ganzzahlige PenTop-Koordinaten (ohne Translation und ohne Skalierung), müssen also bereits von Journal umgerechnet werden.

*siehe auch Quelltexte zu*

| | |
|---|---|
| PostVonBifi() | in ./Erkennen/comm.c |
| SendePCSymbol(), | |
| SendePCMessage(), | |
| clearPadDisplay() | in ./Journal/communication.c |

## Journal -> Parser

Sobald Journal von Erkennen den Befehl FORMELENDE empfängt, sendet es alle Zeichen, die sich auf der aktuellen Seite (dem gerade sichtbaren Blatt) befinden, an den Parser. Dabei wird folgendes Protokoll eingehalten (aus der Sicht von Journal):

```
sendf("%i%i%i", FORMELANFANG, jobID, ZahlDerZeichen);
```

d.h. der in ./include/codierung.h definierte Code FORMELANFANG leitet einen neuen Parsingauftrag ein. Er wird gefolgt von jobID, einer von Journal vergebenen, eindeutigen Kennung (ganze Zahl), die vom Parser bei allen Antworten, die sich auf die nachfolgende Formel beziehen, angegeben wird. Da der Parser asynchron zum Journal arbeitet, ist dieses Vorgehen nötig, um die Parsingantworten den entsprechenden Anfragen zuordnen zu können. Dritter Parameter ist die Zahl der Zeichen, aus denen die Formel besteht. Daran schließen sich die Zeichen an, für die jeweils folgende Daten gesendet werden:

```
sendf("%i%i%i%i%i%i%i", code, xmin, ymin, xmax, ymax,
                                          dummy1, dummy2);
```

`code` ist der Zeichencode gemäß OFR-Codierung, `xmin` bis `ymax` beschreiben das umfassende Rechteck in Journal-Koordinaten und dummy1 und dummy2 sind undefiniert und nur historisch begründet)

Abgeschlossen wird die Formel durch

```
sendf("%i", FORMELENDE);
```

FORMELENDE wird ebenfalls in ./include/codierung.h definiert.

*siehe auch Quelltexte zu*

| | |
|---|---|
| handleFormelende() | in ./Journal/main.c |
| mainloop(), | |
| takeAndParse() | in ./Parser/main.c |

## Parser -> Journal

Wird dem Parser eine Formel zur Analyse vorgelegt, bearbeitet er diese im Hintergrund, bis alle möglichen Kombinationen abgearbeitet sind. Sobald er eine gültige Interpretation für die Formel gefunden hat, informiert er Journal

folgendermaßen über das Ergebnis:

```
sendf("%i%i%d%s", msgParsingResult, jobID, p,
formelstring);
```

`msgParsingResult` ist ein in ./include/messages.h definierter Code, der Journal anzeigt, daß eine gültige Interpretation für die Formel gefunden wurde. `jobID` ist die bereits erwähnte eindeutige kennung für den vorliegenden Parsingauftrag (*siehe vorheriger Abschnitt)*. Die gefundene Interpreation der Formel wird durch die Parameter `p` (Wahrscheinlichkeit, die für die Interpretation berechnet wurde) und `formelstring` (Klartext-Version der Interpretation) beschrieben.
Durchläuft der Parser dagegen alle Kombinationen, ohne eine Interpreation für die vorliegende Formel zu finden, so wird Journal der Mißerfolg mitgeteilt durch

```
sendf("%i%i", msgNoParsingResult, jobID);
```

Entscheidend ist hier nur die ebenfalls in ./include/messages.h definierte Kennung `msgNoParsingResult`, die den Mißerfolg anzeigt. Mittels `jobID` kann Journal die Antwort der Anfrage zuordnen und entsprechend reagieren.

*siehe auch Quelltexte zu*

| | |
|---|---|
| handleMessageFromParser() | in ./Journal/main.c |
| sendResult() | |
| sendNoResult() | in ./Parser/main.c |

***Bug:*** *kein echter Bug, aber eine Unsauberkeit. Die verschiedenen Nachrichten, die zwischen den Prozessen Erkennen, Journal & Parser verschickt werden, sollten wohl entweder in jeweils einem verbindungsspezifischen Header oder gar alle in einem einzigen Header (z.B. ./include/messages.h) zusammengefaßt werden. Stattdessen sind sie verstreut.*

## Journal -> AlgebraSystem
## AlgebraSystem -> Journal

Die Kommunikation mit den Algebrasysteme erfolgt bislang nur über stdin/stdout. Daher werden Eingaben an das Algebrasystem wie auch dessen Ausgaben als reiner ASCII-Text ohne jedes Protokoll verschickt.

## ? -> Butler

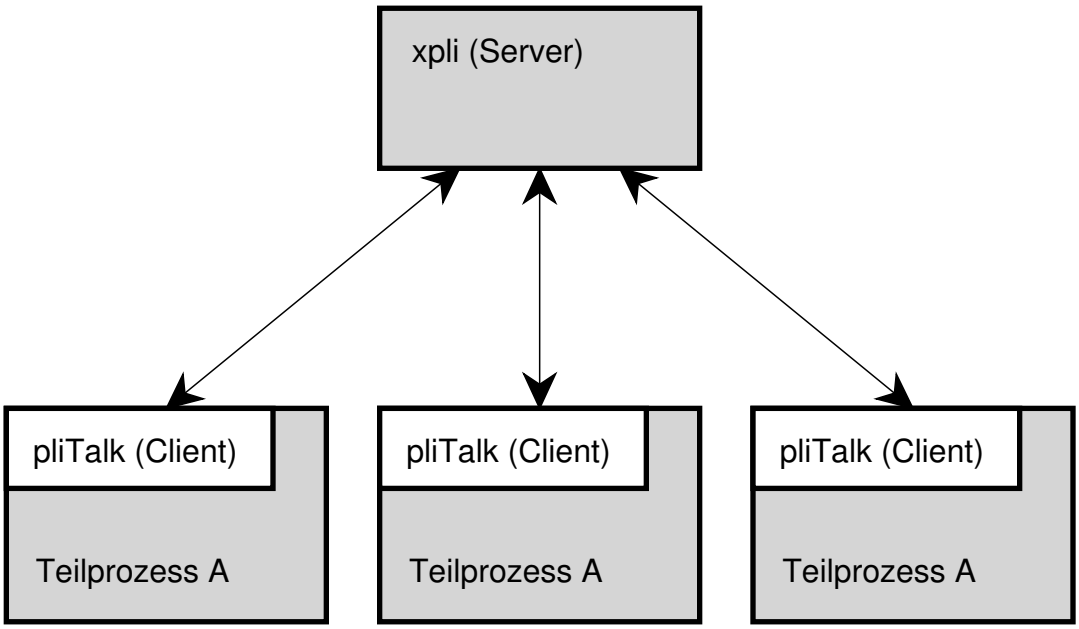
*Zwar ist der Butler z.T. in der IPC schon beschrieben, aber das Format seiner Schnittstelle könnte hier noch beschrieben werden. andererseits steht dafür mit proc_com.c ein fertiges Interface zur Verfügung.*

## ? -> Xpli

Die Hauptaufgaben von Xpli sind:

- (zeitliche) Koordination der graphischen Ausgaben der OFR-Teilprozesse
- Zusammenfassung dieser Ausgaben in einem Fenster
- Entlastung der Teilprozesse (XWindows-Ballast für Init., Redraw etc.)

Xpli ist daher als IPC-Server konzipiert, der mit mehreren Prozessen gleichzeitig kommunizieren kann. Die nötige Client-Funktionalität ist im Modul ./Xpli/pliTalk.c zusammengefaßt, das eine einfache prozedurale Schnittstelle anbietet.




Folgende Funktionen stehen auf Client-Seite zur Verfügung:

```
int pliOpen()
```

baut eine Verbindung zu Xpli auf und gibt eine Kennung zurück, die die (IPC–)Verbindung zu Xpli identifiziert und die bei allen übrigen Funktionen als 1.Parameter ( int cNo) übergeben werden muß. Mit diesem Mechanismus ist es prinzipiell möglich, mehr als eine Verbindung zu Xpli aufzubauen, von denen dann jede eine eigene Kennung und ein eigenes Set globaler Variablen (z.B. aktuelle Cursorposition) benutzt. Ein solches Vorgehen empfiehlt sich z.B. bei der Verwendung von child-Prozessen, die ebenfalls mit xpli kommunizieren.

```
   void pliClose( int cNo )
```

schließt die angegebene Verbindung. Die Kennung `cNo` ist nicht länger gültig.

```
void pliClear( int cNo );
```

löscht den gesamten Fensterinhalt. Vorzugsweise sollte nur ein Client diese Funktion nutzen.

```
void pliMoveTo( int cNo, int x, int y );
void pliLineTo( int cNo, int x, int y );
void pliLine(int cNo, int x0, int y0, int x1, int y1 );
void pliBox( int cNo, int x0, int y0, int x1, int y1 );
```

erlauben einfache Zeichenoperationen. `pliMoveTo()` bewegt einen Cursor auf die angegebene Position, `pliLineTo()` zieht eine Linie vom Cursor zur angegebenen Position und bewegt den Cursor anschließend an die neue Position, `pliLine()` zieht eine Linie zwischen den angegebenen Punkten ohne den Cursor zu verändern und `pliBox()` zeichnet ein Rechteck mit den angegebenen Koordinaten, ebenfalls ohne den Cursor zu verändern..

```
void pliChar( int cNo,  int c, int x0, int y0, int x1,
int y1 );
```

gibt ein Zeichen c als Vektorgraphik aus und skaliert es dabei so, daß das angegebene Rechteck (möglichst) genau ausgefüllt wird.

```
void pliEraseBox( int cNo, int x0, int y0, int x1, int
y1 );
```

löscht alle Darstellungselemente in dem angegebenen Bereich. Dabei werden (mit `pliLineTo()`, `pliLine()` gezeichnete) Linien gelöscht, sofern sie in dem angegebenen Rechteck enthalten sind, Zeichen (mit `pliChar()` gezeichnet) dagegen nur, wenn ihr umfassendes Rechteck mit dem angegebenen identisch ist.

```
void pliMessage( int cNo, char *msg )
```

gibt den Text msg in der Kopfzeile des xpli-Fensters aus (entspricht der Kopfzeile auf dem Notepad).

Eine genauere Beschreibung der Funktionsweise von xpli & pliTalk.c würde zu sehr ins Detail führen. Der Quelltext sowie die Beschreibung der typisierten IPC sollten aber genügen, um die Funktionsweise von xpli & pliTalk.c zu verstehen, da die Zeichenfunktionen nur dazu dienen, die Parameter mit TypInformation zu versehen und zu versenden.

xpli ist objektorientiert (also nicht pixelorientiert), was bedeutet, daß jede Zeichenoperation ein Darstellungselement erzeugt, das von xpli in einer (Display-)Liste gespeichert wird. Soll also ein solches Darstellungselement gelöscht werden, so kann

nicht einfach mit einer weiteren Zeichenoperation der entsprechende bildbereich gelöscht werden, sondern das Darstellungselement muß aus der erwähnten Liste geloescht werden.

# Datentypen & Dateiformate

## Systemeigener Zeichencode.

Da in mathematischen Anwendungen viele Symbole und Sonderzeichen benötigt werden, die der normale ASCII-Zeichensatz nicht enthält, wurde im OFR- System eine eigene Kodierung aller darstellbaren Zeichen gewählt:

```
DRUCK_a ... DRUCK_z = 0 ... 25
ZAHL_0 ... ZAHL_9   = 26 ... 35

MAL_ZEICHEN         =  36     PLUS_ZEICHEN       =  37
STRICH_WAAGERECHT   =  38     INTEGRALZEICHEN    =  39
SUMMENZEICHEN       =  40     PRODUKTZEICHEN     =  41

SLASH               =  42

KLEINER_ZEICHEN     =  43     GRÖSSER_ZEICHEN    =  44
RUNDE KLAMMER AUF   =  45     RUNDE KLAMMER ZU   =  46

WURZELZEICHEN       =  47
ALPHA               =  48
BETA                =  49
DELTA               =  50
MU                  =  51

DRUCK_A ... DRUCK_Z = 55 .. 80

ECKIGE KLAMMER AUF  =  81     ECKIGE KLAMMER ZU  =  82
G_KLAMMER_AUF       =  83     G_KLAMMER_ZU       =  84
FRAGEZEICHEN        = 100     BACKSLASH          = 101
PFEIL HOCH          = 102     SEMIKOLON          = 103
DOPPELPUNKT         = 104     MINUS              = 105
GLEICHHEITSZEICHEN  = 106     UNDERSCORE         = 107
STRICH SENKRECHT    = 108     BRUCHSTRICH        = 109
PUNKT               = 110     AUSRUFEZEICHEN     = 111
ZUWEISUNGSZEICHEN   = 112     BLANK              = 113
KOMMA               = 114     PIZEICHEN          = 115
```

```
DRUCK_ä             = 130
DRUCK_ö             = 131
DRUCK_ü             = 132
DRUCK_Ä             = 133
DRUCK_Ö             = 134
DRUCK_Ü             = 135

SCHREIB_a ... SCHREIB_z= 189 ... 214
SCHREIB_A ... SCHREIB_Z= 215 ... 240

SCHREIB_ä           = 241
SCHREIB_ö           = 242
SCHREIB_ü           = 243
SCHREIB_Ä           = 244
SCHREIB_Ö           = 245
SCHREIB_Ü           = 246
```

*siehe auch*
./include/alphabet.h
./include/codierung.h

**Erweiterung des Zeichencodes**

***Achtung!:*** *Dieser Abschnitt beschreibt* ***n i c h t*** *, wie Mustererkennung oder Parser um neue Zeichen erweitert werden.*

Soll ein neues Zeichen in die Kodierung aufgenommen werden, so müssen folgende Informationen vorliegen:

- Code des Zeichens (positiver, ganzzahliger Wert)
- symbolischer Name zur Verwendung im Quelltext (optional)
- symbolischer Name als String (zur Ausgabe als Text)
- Proportionen gemäß Proportionen-Tabelle (siehe dort)
- skalierbare Vektordarstellung (oder Referenz auf einen Font)

Nur wenn alle diese Informationen vorliegen, wird das Zeichen korrekt behandelt

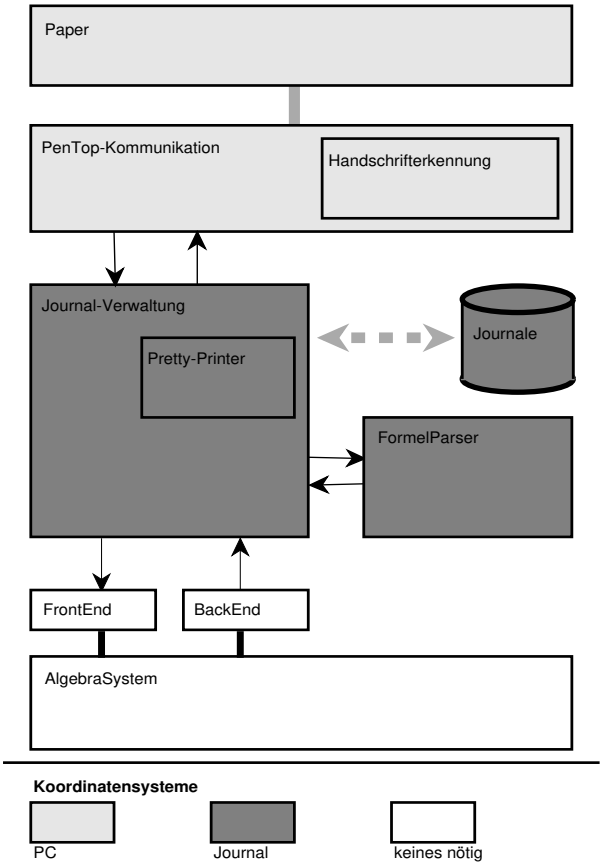
**Wo überall wird diese Kodierung benutzt ?**

Prinzipiell in allen Prozessen, bei der Kommunikation zwischen den Prozessen aber auch bei der Speicherung eines Journals .

## Koordinatensysteme

Zu Beginn der Entwicklung des OFR-Systemes waren PCs auf INTEL-iX86-Basis noch vorwiegend 16-Bit Maschinen, weswegen die verfügbaren Grafikbibliotheken nur mit

maximal 16-Bit-Integer-Arithmetik arbeiteten. Dementsprechend wurde die Softwarekomponente auf dem PC, PAPER, so ausgelegt, daß der ansprechbare Blattbereich kaum größer als der sichtbare Bereich (also in der Größenordnung 640x480 Punkte) war. Die Idee des *virtuellen Arbeitsblattes*, das deutlich größer als der sichtbare Bildschirmbereich ist, wurde völlig in der Journalverwaltung versteckt. Somit existiere n zwei Koordinatensysteme: einerseits die (kleinen, realen) Koordinaten auf PC-Seite und andererseits die (großen, virtuellen) Koordinaten innerhalb des Journals Das folgende Bild zeigt, in welche Prozessen welches Koordinatensystem benutzt wird:




Die Umrechnung zwischen den beiden Koordinatensystemen wird alleine von Journal erledigt

## Journale

Eine Aufgabe der Journalverwaltung ist es, Arbeitssitzungen (= eine Sammlung von Seiten/Blättern mit den entsprechenden Interpretationen und Antworten) dauerhaft zu speichern und ggf. wieder einzuladen. Um diese Sicherungen auch für den Benutzer lesbar zu halten, wurde ein textorientiertes Format gewählt. Bestimmte Abschnitte einer Sicherung werden durch Schlüsselwörter eingeleitet, die am Anfang einer Zeile stehen. Solche Schlüsselwörter und ihre Bedeutung sind:

#USER Auf dieses Schlüsselwort folgt, durch ein Leerzeichen getrennt,

| | |
|---|---|
| | der Name (die UNIX-Benutzerkennung/der loginname) des Benutzers. diese nformation wird nur als Orientierungshilfe in das Journal geschrieben und beim Einlesen ignoriert. |
| #DATE | Ebenfalls nur ein Hinweis. Datum/Uhrzeit, zu dem die Sicherung geschrieben wurde |
| #NOTE | ein beliebiger Kommentar |
| #BLT | die Zahl der in dieser Sitzung gespeicherten Blätter. Muß am Anfang stehen (vor dem ersten #CNT/#COUNT) und wird beim Einlesen dazu benutzt, die Zahl der einzulesenden Blätter zu bestimmen. |

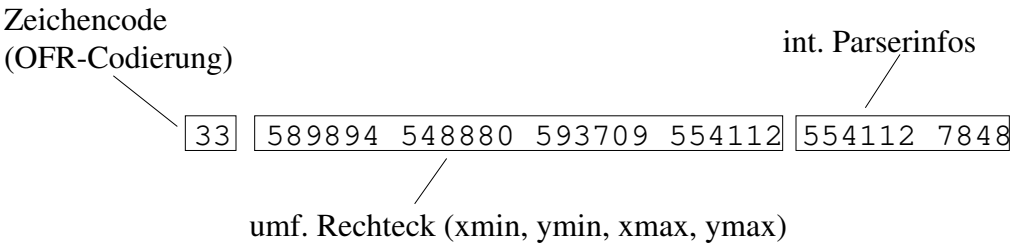
Während die bisherigen Schlüsselfelder i.A. nur einmal pro Arbeitssitzung vorkommen, erscheinen die folgenden je einmal pro Seite:

| | |
|---|---|
| #RED | die Reduce-Klartextdarstellung der Formel |
| #TeX | die TeX-Version der Formel |
| #CNT | die Zahl der Zeichen in der Formel |
| bzw. #COUNT | die Zahl der Zeichen in der Formel. Auf die Zeile mit diesem Schlüsselwort folgen die Zeichen, aus denen die Formel besteht. Jedes Zeichen wird mit seinem umfassenden Rechteck und weiteren Informationen in einer eigenen Zeile angegeben. |

Ein Beispiel:

```
#USER marvin.
#DATE Sitzung vom  7. APR. 1991
#NOTE wie tief kann man schachteln ? - ein Kettenbruch
#COUNT 13
27 531906 528170 536157 536999 536999 1323
105 540735 532857 545204 533838 533838 981
28 571364 516834 578558 525663 525663 13243
109 555450 530023 599813 530895 530895 872
29 557630 534274 561663 542231 542231 11935
109 567985 538743 570819 539288 539288 545
30 582700 532748 586297 537435 537435 7030
109 574089 539397 595235 540596 540596 1199
31 573980 543648 578776 550297 550297 9973
105 583136 545937 585643 546155 546155 218
32 589894 540487 592292 545501 545501 7521
109 587714 547245 594908 547681 547681 436
33 589894 548880 593709 554112 554112 7848
```

Die einzelnen Komponenten einer zeichenbeschreibenden Zeile haben dabei folgende Bedeutung:




## Koord/Musterfiles, Netze

*siehe Dokumentation zu Neuron.*