# Mac2E

| COLLABORATORS | | | |
|---|---|---|---|
| | *TITLE* :<br><br>Mac2E | | |
| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
| WRITTEN BY | | November 28, 2024 | |

| REVISION HISTORY | | | |
|---|---|---|---|
| NUMBER | DATE | DESCRIPTION | NAME |
| | | | |

# Contents

# Chapter 1

# Mac2E

## 1.1 Mac2E

```
*****************************************************************************

                                Mac2E (v3.0)
                    Macro PreProzessor zur Amiga E Sprache
                            Archiv vom 10 März 1994
                    © Copyright 1993, 1994, Lionel Vintenat

*****************************************************************************

Achtung ! Alle Programme in diesem Archiv benötigen Workbench 2.0 oder
später, um zu starten.

 Einleitung           einleitende Worte
 Was ist ein Makro ?  Syntax-Definitionen
 Mac2E-Benutzung      Wie startet man das ?
 Mac2E und MUI        Hinweise zu MUI
 Bekannte Fehler      hoffentlich keine
 History              Was war wann ?
 Zukunftsmusik        Was soll noch sein ?
 Verteilung           Wer darf's kopieren ?
 Der Autor            Ich über mich.
 Danksagungen         Danke, Jungs.
```

## 1.2 Einleitung

```
 Dieser Abschnitt ist unterteilt in drei größere Bereiche, die
folgende Themen abdecken:

 - Warum ausgerechnet Mac2E ?
 - Was macht Mac2E ?
 - Wie macht es das ?

 Wie alles anfing...    Warum das alles ?
 Allgemeines            Kurzbeschreibung
 Motivationsgründe      Was ist beabsichtigt ?
```

## 1.3 Wie alles anfing...

Am Anfang war Amiga E und ich. Wir hatten eine schöne Zeit, wir zwei und schrieben wundervolle Programme in kürzester Zeit. Da ich aber keine RKMs hatte (und immer noch nicht habe), waren diese Programme häßlich, ohne graphische Benutzeroberflächen, aber trotzdem, es war eine schöne Zeit.

Dann aber kam MUI und nichts zwischen Aminga E und mir sollte mehr so sein wie zuvor. Warum ? Nun, Amiga E erlaubt nicht den Gebrauch von Makros und MUI Programmierung ohne Makros ist einfach verrückt ! Andererseits wäre es unvernünftig gewesen, so etwas wie MUI vorbeiziehen zu lassen. Deswegen zog ich mich eine Zeit lang auf die Sprache C zurück: dies war der Beginn einer dunklen Zeit für meinen Amiga...

Dann bekam ich Zugang zum InterNet. Ich sprach über mein Problem mit Wouter und er empfahl mir, einen C Preprozessor zu benutzen: das war eine hervorragende Idee. Aber nachdem ich es einige Zeit probiert hatte, stellte sich dieses Verfahren als umständlich heraus: die Übersetzungs-zeit war 100 mal so lang wie vorher und der Compiler spuckte nicht die richtige Zeilenzahl für die Fehlermeldungen aus. Da bekam ich die Idee zu Mac2E...

## 1.4 Allgemeines

Mac2E ist ein Preprozessor für den Amiga E Compiler von Wouter van Oortmerssen, der jedoch nur eine Funktion beherrscht: Makros in einem E Quelltext ersetzen. Mit anderen Worten, die Inkludierung von Dateien und konditionale Übersetzung werden von Mac2E nicht behandelt, so wie es andere C Preprozessoren tun.

Oh ! Fast hätte ich es vergessen: alle Executables in diesem Archiv sind selbstverständlich in Amiga E geschrieben !

## 1.5 Motivationsgründe

Ich habe Mac2E mit drei Ideen im Hinterkopf entwickelt:

- etwas einfaches im Gebrauch zu schaffen (im Sinne von Amiga E)

- die Probleme, die ich mit dem C Preprozessor hatte, zu lösen (siehe @{ "Wie alles anfing ..." LINK comment_tout_a_commencé})

- einen Preprozessor zu entwickeln, durch den der Quelltext nicht abhängig von ihm ist. Mit anderen Worten: wenn eine neue Version von Amiga E entwickelt ist, die einen Preprozessor enthält, soll eine Umsetzung der existierenden Quelltexte kaum Änderungen ver-langen.

Ich glaube, daß in der Version 3.0 diese drei Ideen effektiv implementier sind:

- Mac2E hält sich sehr nahe an die Bedienung eines C Preprozessors, so

daß sich Programmierer schnell an ihn gewöhnen können

- Mac2E benötigt zur Bearbeitung einer Datei ungefähr genauso viel Zeit, wie Amiga E selbst, was, gemessen an der Geschwindigkeit von Amiga E selbst auf kleinen Prozessoren, akzeptierbar sein sollte

- Mac2E fügt keine Linefeeds ein, wenn es ein Makro ersetzt, so daß die Fehlermeldungen auch die richtige Zeilennummer erhalten

- Makro-Definitionen werden in einer separaten Datei vorgenommen, die direkt an das Programm Mac2E übergeben wird, so daß eine spätere Anpassung an Amiga E _mit_ Preprozessor keine Schwierigkeit bietet, da nichts Mac2E-spezifisches im Quelltext zu finden ist

## 1.6 Was ist ein Makro ?

Einfach gesagt, definieren wir ein Makro dadurch, daß wir einem Bezeichner (dem Makronamen) eine Zeichenkette (der Rumpf des Makros) zuweisen. In unserem Quelltext verwenden wir anstelle des Rumpfes nur den Bezeichner und der Preprozessor ersetzt den Bezeichner durch den Rumpf.

Meiner Meinung nach ist der Gebrauch von Makros in drei Fällen sehr sinnvoll:

- um das Mehrfachschreiben eines Quelltext-Teils zu vermeiden
  -> siehe Beispiel 1

- um den Gebrauch abstrakter Werte zu vereinfachen
  -> siehe Beispiel 2

- um einen Programmteil logisch umzugruppieren
  -> siehe Beispiel 3

Natürlich ist dies nur ein oberflächlicher Überblick über die Syntax der Makros. Makros, so wie sie in fast allen Preprozessoren heutzutage implementiert sind, erlauben noch viel mehr. Die folgenden Abschnitte erklären den Gebrauch von Makros im Detail:

Ein Makro definieren
Ein Makro benutzen
Fortgeschrittener Gebrauch

## 1.7 Beispiel 1

Man stelle sich ein Programm vor, das Speicher sequentiell ausließt. Dazu sind zwei Variablen definiert:

```
DEF memory_pointer : PTR TO CHAR, character
```

Um ein Byte anzusprechen, ist folgende Programmzeile notwendig:

```
 character:=Char(memory_pointer++)
```

Ohne Makros müßten wie diese Zeile jedesmal schreiben, wenn wir ein Byte einlesen wollen. Wenn Lesezugriffe in verschiedenen Prozeduren geschehen, kann das sehr schnell ermüdend werden. Die Lösung ist die Definition eines Makros mit dem Namen ReadMemory und dem Rumpf character:=Char(memory_pointer++). Nun müssen wir nur noch ReadMemory eingeben, um ein Byte zu lesen.

## 1.8 Beispiel 2

Um eine Library zu öffnen, muß man an die Funktion OpenLibrary() den Namen der Library in Kleinbuchstaben übergeben. Wenn man

```
OpenLibrary('Dos.library',0)
```

schreibt, wird zwar keine Fehlermeldung generiert, aber die Library wird während der Laufzeit nicht geöffnet werden. Die Lösung hierfür ist die Definition eines Makros mit dem Namen DosLibraryName und dem Rumpf 'dos.library'. Dadurch kann es zu keinem Tippfehler beim Aufruf durch

```
OpenLibrary(DosLibraryName,0)
```

kommen.

## 1.9 Beispiel 3

Um sicherzugehen, daß stdout nicht null ist, empfiehlt die Amiga E Dokumentation zu Beginn des Programms ein WriteF('') zu setzen. Eine viel elegantere Möglichkeit ist es, ein Makro mit dem Namen OpenStdout zu definieren, das im Rumpf WriteF('') enthält. Dann braucht man nur noch OpenStdout im Quelltext zu benutzen, was viel aussagekräftiger ist.

In diesem einfachen Beispiel kommt der Unterschied zwischen diesem Makro und den beiden vorherigen nicht besonders deutlich heraus, aber es ist wichtig, daß man versteht, daß dieses Makro nicht auf ein Programm beschränkt ist, wie Beispiel 1, sondern in allen anderen Programmen benutzt werden kann. Darüberhinaus verhält sich OpenStdout wie eine Klein-Prozedur, die eine Aufgabe erfüllt (im Gegensatz zu Beispiel 2).

## 1.10 Ein Makro definieren

Die folgenden Absätze erklären, wie ein Makro definiert wird, von der einfachsten bis zur komplexesten Form:

```
Definition eines Makros ohne Parameter
Definition eines Makros mit Parameter(n)
Fortgeschriettene Definition
```

## 1.11 Definition eines Makros ohne Parameter

Eine einfache Makrodefinition hat die folgende Syntax:

```
#define macro_name macro_body
   |   |     |    |     |    |
  (1) (2)   (3)  (2)   (4)  (5)
```

wobei

(1) #define den Beginn der Definition eines Makros definiert und irgendwo in einer Zeile stehen kann

(2) ein oder mehrere Leerzeichen ist

(3) der Name des Makros ist (jede Kombination von Zahlen, Buchstaben und dem Unterstrich)

(4) der Rumpf des Makros ist (jede Kombination von Zahlen, Buchstaben und dem Unterstrich)

(5) ein Zeilenumbruch ist

Beispiele:

```
#define ReadMemory       character:=Char(memory_pointer++)
#define DosLibraryName   'dos.library'
#define OpenStdout       WriteF('')
```

## 1.12 Definition eines Makros mit Parameter(n)

Genau wie eine Prozedur kann auch ein Makro Parameter haben. Die Syntax in diesem Falle ist die folgende:

```
#define macro_name(parameter1,parameter2,...,parameterN) macro_body
   |   |   |      |    |     |    |     |   |    |     |      |    |
  (1) (2) (3)    (4)   |    (5)   |    (5) (5)   |    (7)    (8)  (9)
                       +----------+--------------+
                                  |
                                 (6)
```

wobei

(1) #define den Beginn der Definition eines Makros definiert und irgendwo in einer Zeile stehen kann

(2) ein oder mehrere Leerzeichen ist

(3) der Name des Makros ist (jede Kombination von Zahlen, Buchstaben und dem Unterstrich)

(4) eine öffnende runde Klammer direkt nach dem Makronamen ist

(5) ein Komma, jeden Parameter vom nächsten trennt, ist

(6) ein oder mehrere Parameter (jede Kombination von Zahlen, Buchstaben und dem Unterstrich) ist; jeder Parameter kann von führenden oder folgenden Leerzeichen oder Tabulatoren begleitet sein

(7) eine schließende runde Klammer ist, gefolgt von einer beliebigen Anzahl von Leerzeichen oder Tabulatoren

(8) der Rumpf des Makros ist (jede Kombination von Zahlen, Buchstaben und dem Unterstrich)

(9) ein Zeilenumbruch ist

Beispiel:

```
#define Power2( x )                 ((x)*(x))
#define SwapVariablesXY(X,Y,TEMP)   TEMP:=X; X:=Y; Y:=TEMP
#define Max( x , y )                (IF (x)>(y) THEN (x) ELSE (y))
```

Parameter, die innerhalb der Makrodefinition spezifiziert werden, werden formale Parameter genannt.

## 1.13 Fortgeschrittene Definition

Es kann passieren, daß die Definition des Rumpfes eines Makros so lang ist, daß sie nicht in eine Zeile paßt. Es ist möglich, diese Zeile in mehrere Zeilen zu unterteilen. Um dem Preprozessor zu sagen, daß die Definition in der nächsten Zeile fortgesetzt wird, muß nur ein "\" am Ende der Zeile vor dem Umbruch geschrieben werden. Der Preprozessor wird das "\" - Zeichen und den Umbruch überspringen und die nächste Zeile weiterinterpretieren. Ein Makro kann auf diese Weise über mehrere Zeilen fortgesetzt werden.

Die Definitions-Syntax für einen solchen Fall ist:

```
#define macro_name(parameters)  body_piece1 \
                body_piece2 \
                ...
                body_pieceN
```

Achtung: das "\"-Zeichen muß unmittelbar von einem Umbruch gefolgt werden, damit der Preprozessor es richtig interpretiert.

Erstes Beispiel:

```
#define SwapVariablesXY(X,Y,TEMP) TEMP:=X; \
X:=Y; \
Y:=TEMP
```

ist ein Makro, daß als Rumpf TEMP:=X; X:=Y; Y:=TEMP hat. Beachte, daß die ";" wichtig sind.

Erstes Beispiel:

```
#define SayHello WriteF('Hello, I'm the one who wrote \
the great program Mac2E (pub) !\n')
```

ist ein Makro, daß als Rumpf

```
WriteF('Hello, I'm the one who wrote the great program Mac2E (pub) !\n')
```

hat. Beachte, daß das einzelne "\"-Zeichen als Signal für den Preprozessor interpretiert wurde, den nächsten Zeilenumbruch zu überspringen.

Drittes Beispiel:

```
#define UselessMacro [1 space ->@{uu}
 ][2 spaces ->@{uu}
  {u}][3 spaces ->@{uu}
   {u}] and that's all !
```

ist ein Makro, das als Rumpf

```
[1 space -> ][2 spaces ->  ][3 spaces ->   ] und das wars !
```

hat.

## 1.14 Using a macro

Ein Makro zu definieren ist nicht alles, denn wir wollen es schließlich auch benutzen. Um dies zu tun, müssen wir einfach den Namen des Makros dort einfügen, wo wir normalerweise den Quelltext im Rumpf hinschreiben würden. Aber Achtung, ein Makro ist keine Instruktion, die normalerweise vom Compiler erkannt werden würde. Vor der Übersetzung muß ein Programm, das Makros enthält, durch den Preprozessor geparst werden. Der Sinn dieses Programms ist es, auftretende Makros zu ersetzen. Der Rumpf des Makros sollte Quelltext enthalten, der durch den Compiler erkannt wird. Wenn der Preprozessor seine Arbeit beendet hat, kann die Datei kompiliert werden.

Die folgenden Abschnitte beschreiben im Detail, wie der Preprozessor vorgeht, um Makros zu finden und zu ersetzen.

Ein Makro finden
Parsen der Parameter
Ein Makro ersetzen

## 1.15 Ein Makro finden

Damit der Preprozessor einen Makro-Namen erkennen kann, muß der Name, der im Quelltext steht, folgende Bedingungen erfüllen:

- Er muß genau derselbe sein, der in den Definitionen definiert wurde. Dabei wird zwischen Groß- und Kleinschreibung unterschieden.

- Der Name muß von einem Zeichen, anders als ein Buchstabe, eine Zahl
  oder einem Untertrich, umschlossen sein.

Sind diese beiden Bedingungen erfüllt, wird der Preprozessor den Makro-Namen erkennen.

Beispiele:

Angenommen, wir haben einen Makronamen todo (der Rumpf soll uns diesmal nicht interessieren) definiert. Es wird in den folgenden Befehlszeilen erkannt:

```
a:=toto+1
WriteF('Silly string to introduce \d !\n',toto)
```

Jedoch nicht in den nun folgenden:

```
a:=different_than_toto+1
WriteF('Silly string to introduce \d !\n',toto1)
```

## 1.16 Parsen der Parameter

Im letzten Abschnitt haben wir gesehen, daß man Makros Parameter (formale Parameter genannt) mit auf den Weg geben kann, wie man es mit einer Prozedur machen kann. Dann, wenn wir das Makro verwenden wollen, müssen wir es mit Agumenten (reale Parameter genannt) versorgen. Der Aufrufs-Syntax für ein Makro ist:

```
macro_name(parameter1,parameter2,...,parameterN)
   |       |    |      |    |      |   |    |      |
  (1)     (2)   |     (4)   |     (4) (4)   |     (5)
                +-----------+---------------+
                            |
                           (3)
```

wobei

(1) der Name des Makros ist

(2) eine offene runde Klammer ist, die direkt auf den Namen folgt

(3) ein oder mehrere Parameter ist

(4) ein Komma ist, um jeden Parameter vom nächsten zu trennen

(5) eine schließende runde Klammer ist

Achtung: die Parameter sind durch Kommata und Klammern beschränkt. Alle Zeichen zwischen den Kommata werden als Parameter interpretiert.

Falls ein Makro ohne Parameter definiert wurde, ist die Syntax für seinen Aufruf einfach nur

{b}macro_name.

Wenn der Preprozessor ein Makro analysiert, erwartet er natürlich genau dieselbe Anzahl an realen Parametern wie formale Parameter definiert wurden. Insbesondere darf ein Makro, das ohne Parameter definiert wurde, nicht von einem "("-Zeichen gefolgt werden, da der Preprozessor sonst denkt, daß das Makro mit Parametern aufgerufen wird.

Wenn die Syntax für ein Makro in ordnung ist, assoziiert der Preprozessor jeden realen Parameter mit dem entsprechenden formalen Parameter, so, wie der Compiler es mit einer Prozedur macht.

Beispiele:

Angenommen, wir haben ein Makro toto folgendermaßen definiert:

```
#define toto(param1, param2) any_body
```

Hier ist eine Tabelle, was bei verschiedenen Aufrufen passiert:

| Aufruf | Parameter 1 | Parameter 2 |
|---|---|---|
| toto(a,1) | a | 1 |
| toto( a , 1 ) | a | 1 |
| toto( (3+2)*5 ,WriteF('Ah !\n')) | (3+2)*5 | WriteF('Ah !\n') |
| toto (a,1) | FEHLER | |
| toto(1,2,3) | FEHLER | |

## 1.17 Ersetzen eines Makros

Wenn ein zu ersetzendes Makro ohne Parameter definiert wurde, ersetzt der Preprozessor das Makro einfach durch seinen Rumpf.

Sollte das Makro jedoch mit Parametern definiert worden sein, ersetzt der Preprozessor weiterhin das Makro durch seinen Rumpf, jedoch auch die formalen Parameter durch die entsprechenden realen Parameter.

Erstes Beispiel:

Angenommen, wir haben folgende Makrodefinition:

```
#define DosLibraryName 'dos.library'
```

Dann haben wir den Aufruf OpenLibrary(DosLibraryName, der durch den Aufruf OpenLibrary('dos.library') ersetzt werden wird.

Zweites Beispiel:

Angenommen, wir haben folgende Makrodefinition:

```
#define Square(x) ((x)*(x))
```

```
#define Max(x,y) (IF (x)>(y) THEN (x) ELSE (y))
```

Dann haben wir die Aufrufe a:=Square(4+3) * Max(7,2*(8-2)) die durch a:=((4+3)*(4+3)) * (IF (7)>(2*(8-2)) THEN (7) ELSE (2*(8-2))) ersetzt werden.

Beachte, wie die große Anzahl an Klammern in den Rümpfen dieser beiden Makros die Ausführung beeinflussen. Ohne sie wäre das Resultat nicht das, was man erwartet. Auch wenn ein Makro einer Prozedur oder einer Funktion ähnelt, sind sie nicht dasselbe. Der Rumpf eines Makros wird niemals während der Ersetzung ausgewertet, sondern nur ersetzt.

## 1.18 Fortgeschrittener Gebrauch

Bis hierher haben wir gelernt, wie man mit Definitionen und Gebrauch von Makros umgeht. In den folgenden Abschnitten werden wir die technischen Aspekte des Makrogebrauchs kennenlernen, die ebenso wichtig sind.

Makros, Kommentare und Strings
Verschachtelte Makros
Makros als Makro-Argumente
Sonderzeichen

## 1.19 Makros, Kommentare und Strings

Im letzten Abschnitt wurde gesagt, daß Makros überall im Quelltext vorkommen können. Nun, das ist nicht wahr ! In Wirklichkeit sucht der Preprozessor nicht nach Makro-Aufrufen in Kommentaren und Strings. Makros sind dazu da, Teile des Quelltexts unter einem Namen zusammenzufassen, weshalb es keinen Grund für Makros in Kommentaren und Strings gibt.

In der Praxis bedeutet dies, daß in Kommentaren und Strings alles mögliche stehen darf.

## 1.20 Verschachtelte Makros

Wenn man ein Makro definiert, kann man alles nur denkbare in den Rumpf des Makros schreiben, sogar Makro-Aufrufe. Der Preprozessor wird diese Art von Aufruf während der Ersetzung des Makros, in dessen Rumpf ein weiteres Makro steht, beachten. In der Praxis ersetzt der Preprozessor soviel wie möglich, so daß zu guter Letzt kein Makro-Aufruf mehr übrig bleibt. Natürlich können die Parameter eines Makros auch aufrufende Parameter eines verschachtelten Makros werden. Es gibt keine Begrenzung der Verschachtelungstiefe.

Achtung: der Rumpf eines Makros darf keinen Aufruf von sich selber enthalten. Ansonsten würde der Preprozessor dieselbe Ersetzung immer und immer wieder machen, bis in alle Ewigkeit bzw. bis zum Ende des Speichers.

```
 Erstes Beispiel:

 Angenommen, wir haben zwei Makros definiert:

#define InfiniteValue $FFFFFFFF
#define FinitePositiveNumber(x) (((x)>0) AND ((x)<>InfiniteValue))

 Dann können wir den folgenden Aufruf machen

IF FinitePositiveNumber(A*B)=FALSE THEN WriteF('Error !\n'),

der durch

IF (((A*B)>0) AND ((A*B)<>$FFFFFFFF))=FALSE THEN WriteF('Error !\n')

ersetzt werden wird.

 Zweites Beispiel:

 Angenommen, wir haben zwei Makros definiert:

#define AbsoluteValue(x) (IF (x)>0 THEN (x) ELSE -(x))
#define MaxOfAbsoluteValues(x,y) (IF AbsoluteValue(x)>AbsoluteValue(y) THEN
(x) ELSE (y))

 Dann können wir den folgenden Aufruf machen

a:=MaxOfAbsoluteValues(5,-(A*B)),

der durch

(IF (IF (5)>0 THEN (5) ELSE -(5))>(IF (-(A*B))>0 THEN (-(A*B)) ELSE -(-(A*B)))
THEN (5) ELSE (-(A*B)))

ersetzt wird.
```

## 1.21 Macro calls as macro arguments

```
 Aus den vorherigen Beispielen hat man gesehen, daß als Argument alles
an die Makros übergeben werden kann, solang die Argumente kohärent sind.
Also kann auch ein Makro als Argument übergeben werden. Auch hierbei wird
der Preprozessor erst das eingeschlossene Makro behandeln und dann das
Aufrufende. Es giebt auch hierbei keine Begrenzung in der Verschachtelungs-
tiefe.

 Bedenke, daß der Preprozessor absolut jeden Makro-Aufruf im Quelltext
behandelt, außer in Kommentaren und Strings.

 Erstes Beispiel:

 Betrachten wir die folgenden Makros:

#define SillyValue 12
#define Double(x) (2*(x))
```

Der Aufruf

```
Double(SillyValue)
```

würde durch

```
(2*(12)).
```

ersetzt werden.

Zweites Beispiel:

Betrachten wir die folgenden Makros:

```
#define MaskWeightStrong(x) ((x) AND $FFFF)
#define Average(x,y) (((x)+(y))/2)
```

Der Aufruf

```
Average(100,MaskWeightStrong(100000))
```

würde ersetzt werden durch (((100)+((100000) AND $FFFF))/2).

## 1.22 Sonderzeichen

Es sind sicherlich schon Gedanken daran aufgetaucht, was mit den Zeichen "(", ")" oder "," passiert, wenn sie als Makro-Argumente auftauchen, wo sie doch Anfang und Ende eines Aufrufs bzw. eines Parameters kennzeichnen. Nun, der Preprozessor ist intelligent genug, um zu entscheiden, welches Zeichen zum Argument und welches zum Aufruf gehört.

Achtung: die Argumente müssen nichts desto trotz kohärent bleiben ! So muß z.B. jede offene Klammer eine korrespondierende schließende Klammer haben. Ebenso muß ein Komma oder ein String durch Anführungsstriche eingeschlossen sein.

Beispiele:

Angenommen, toto wurde definiert als ein Makro mit zwei Parametern. Hier ist eine Tabelle, was bei verschiedenen Aufrufen passiert:

| Aufruf | 1. Parameter | 2. Parameter |
|---|---|---|
| toto((3+4)*(5-6),'1, 2 et 3') | (3+4)*(5-6) | '1, 2 et 3' |
| toto((((())())()),character ",") | (((())())() | Zeichen "," |
| toto(),4) | F E H L E R | |
| toto(4,,) | F E H L E R | |

## 1.23 Benutzung von Mac2E

Um zu verstehen, was nun folgt, sollte man wissen, was ein Makro ist und besonders verstanden haben, wie man ein Makro definiert und benutzt, so, wie es in C geschieht. Sollte dies nicht der Fall sein, gehe zum Abschnitt {"Was ist ein Makro ?" Link Présentation_macro} zurück. Wenn alle diese Dinge schon bekannt sind, kann dieser Abschnitt getrost vergessen werden, außer, wenn der Synatx der Makros nicht ganz klar ist.

Im nächsten Abschnitt wird generell die Benutzung der Programme PreMac2E und Mac2E besprochen, ohne nochmal auf die Makros einzugehen.

Um nochmals zu erinnern: der Hauptsinn diese Programms ist es, die Makros in E Quelltexten zu ersetzen. Das erste, was zu tun ist, ist deswegen einige Makros zu definieren. Dieses wird in einer separaten Datei getan, die Makro-Datei genannt wird. Danach wird die Makro-Datei mit PreMac2E behandelt und schlußendlich kann mit Hilfe von Mac2E die eigentliche Ersetzungsarbeit erfolgen. Die Bearbeitung des Quelltextes benötigt demnach drei Schritte, die im Detail in den folgenden drei Abschnitten besprochen:

Makro-Dateien
PreMac2E aufrufen
Mac2E aufrufen
Fehlermeldungen

## 1.24 Makro-Dateien

Eine Makro-Datei ist eine ASCII-Datei, die nur Makro-Definitionen enthält. Zur Erinnerung: Makros können nicht innerhalb des Qulltextes definiert werden, sondern ausschließlich innerhalb dieser Makro-Dateien.

Diese Dateien können auch Kommentare enthalten. Diese können überall stehen, außer in den Makro-Definitionen. Mit andern Worten können Makros nur zwischen den einzelnen Definitionen stehen. Kommentare brauchen nicht durch Anfangs- und Endezeichen begrenzt werden.

Normalerweise werden Macro-Dateien im Unterverzeichnis MacroFiles/ abgelegt, das sich in dem Verzeichnis befindet, wo Amiga E installiert wurde.

siehe Ein Makro definieren

## 1.25 Aufruf von PreMac2E

Nachdem eine Makro-Datei erstellt wurde, ist sie noch nicht sofort für Mac2E verwendbar. Vorher muß sie durch PreMac2E vorbehandelt werden. Dieses Programm hat folgenden Syntax:

```
PreMac2E macro_file pre_analyzed_macro_file
```

wobei

- macro_file die zu behandelnde Makro-Datei ist

- pre_analyzed_macro_file die resultierende, behandelte Datei ist

Um z.B. die Makro-Datei mui.e, die in diesem Archiv enthalten ist, zu bearbeiten, habe ich

PreMac2E MacroFiles/mui.e PreAnalysedMacroFiles/mui.e

eingeben.

Normalerweise werden so behandelte Dateien im Verzeichnis PreAnalysedMacroFiles/ abgelegt, welches in derm Verzeichnis zu finden ist, in dem Amiga E installiert wurde.

Während des Bearbeitens versucht PreMac2E alle Makro-Aufrufe innerhalb eines Makros zu ersetzen. Zusätzlich hierzu werden die Makros sortiert und in eine Hash-Tabelle eingetragen. Zum Schluß wird der Inhalt dieser Tabelle in die Ausgabedatei geschrieben. Mac2E benutzt diese behandelte Datei anstelle der Quelldatei. Dadurch wird ein enormer Geschwindigkeits-gewinn erreicht.

Dieser Bearbeitungsvorgang muß nur einmal durchgeführt werden (unter der Voraussetzung, daß keine Fehler auftraten, natürlich). Danach wird nur noch die vorbehandelte Datei benutzt. Wird die Originaldatei ver-ändert, muß dieser Vorgang natürlich wiederholt werden, damit die Änder-ungen Wirkung zeigen.

Der genaue Aufruf von PreMac2E ist:

"FROM/A,TO/A,VER=VERBOSE/S,KS=KEEPSPACES/S".

VERBOSE ist im Abschnitt Fehlermeldungen. beschrieben.

KEEPSPACES zwingt PreMac2E dazu, die Leerzeichen und Tabulatoren bei-zubehalten, wenn sie zu Beginn einer Zeile stehen, wenn der Rumpfe des Makros sich über mehrere Zeilen erstreckt. PreMac2E löscht sie normalerweise, um Platz zu sparen.

## 1.26 Der Aufruf von Mac2E

Mac2E ist der eigentliche PreProzessor in diesem Archiv. Er ist es, der nach Makros im Quelltext sucht und sie ersetzt. Der Aufruf lautet wie folgt:

Mac2E FROM e_source_file TO e_destination_file WITH pre_analyzed_file_list

wobei

- e_source_file der Name der Quelldatei ist, in der die Makros ersetzt werden sollen,

- e_destination_file der Name der Zieldatei ist, in der die behandelte Quelldatei gespeichert wird,

- pre_analyzed_file_list ist eine Liste von einer oder mehrerer
  Dateien, die durch PreMac2E behandelt wurden

So aufgerufen lädt Mac2E alle vorbehandelten Dateien ein und ersetzt die gefundenen Makros nach den Regeln, die in diesen Dateien aufgestellt wurden.

Siehe Ein Makro benutzen und Fortgeschrittene Benutzung

## 1.27 Fehlermeldungen

Alle Fehlermeldungen, die durch PreMac2E und Mac2E ausgegeben werden, sind ausreichend selbsterklärend. Die Zeilennummer, in der der Fehler gefunden wurde, wird ebenfalls angegeben.

Die einzige Ausnahme hiervon ist, wenn PreMac2E geschachtelte Makroaufrufe bearbeitet. Dieses Stadium der Analyse wird erreicht, nachdem alle Makros eingelesen wurden, deshalb kann PreMac2E nicht mehr mit Zeilennummern arbeiten. Um einen Fehler zu finden, muß PreMac2E mit der VERBOSE Option gestartet werden, denn nun wird zusätzlich zum Fehler auch der Name des Makros ausgegeben. Der Fehler ist im Rumpf dieses Makros zu finden.

## 1.28 Mac2E und MUI

Wenn Du Wie alles anfing... gelesen hast, weißt Du schon, daß Mac2E seine Existenz der Tatsache verdankt, daß MUI viel einfacher mit Makros zu programmieren ist, als ohne. Dies ist der Grund, warum das erste und momentan einzige Beispiel zum Gebrauch von Mac2E sich mit MUI beschäftigt. In diesem Archiv befindet sich alles, was zum Gebrauch von MUI mit Amiga E benötigt wird. Um dies zu tun, braucht man sechs Dinge:

- PreMac2E
- Mac2E
- mui.m
- muimaster.m
- mui.e
- OptiMUI2E

Die Idee, die die Konzeption von mui.m und mui.e beeinflußte, war, so nahe wie möglich am Original-Include mui.h zu bleiben, damit man während der Arbeit mit MUI in dieses Include sehen kann, da es sehr viele Kommentare enthält, die in der E Version nicht vorhanden sind.

Das komplette Interface basiert auf MUI 2.0. In den MUI Archiven gibt es zwar schon einige Dateien für die Benutzung mit Amiga E, aber sie sind weder komplett noch so praktisch wie die Dateien, die mit diesem Archiv kommen.

Mac2E
PreMac2E
mui.m

muimaster.m
mui.e
OptiMUI2E

## 1.29 mui.m

mui.m ist, wie der Name schon sagt, ein klassisches Amiga E Include. Es enthält alle Strukturen, die in MUI.h definiert wurden mit dem Unterschied, daß alle Namen (der Strukturen und ihrer Felder) in Kleinbuchstaben konvertiert wurden. Dies ist aufgrund einer Einschränkung von IConvert geschehen.

Um die MUI Strukturen in eigenen Programmen zu verwenden, muß nur MODULE 'libraries/mui.m' an den Anfang des Quelltextes gesetzt werden.

## 1.30 muimaster.m

muimaster.m ist, wie der Name schon sagt, ein klassisches Amiga E Include. Es enthält alle Funktionsdefinitionen der muimaster.library. Die Namen sind dieselben mit der Einschränkung, daß sie alle mit Mui anstelle von MUI beginnen. Diese Einschränkung ist durch Amiga E auferlegt, da Funktionsnamen mit einem Großbuchstaben beginnen müssen und mit einem Kleinbuchstaben fortgesetzt werden.

Um die MUI-Funktionen in eigenen Programmen zu verwenden, muß nur MODULE 'libraries/muimaster.m' an den Anfang des Quelltextes gesetzt werden.

## 1.31 mui.e

mui.e ist der Schlüssel zur Benutzung dieses "MUI-Amiga E Interfaces", da es alle Makros (Konstanten und kompliziertere Dinge, wie Objekt-Definitionen) der Datei mui.h enthält, nur an die E Sprache angepaßt. Die Syntax der mui.e Makros, genau wie die Syntax ihrer Rümpfe, ist genau dieselbe, wie in mui.h.

Genauso, wie diese Datei die Vorteile von MUI demonstriert, ist sie auch ein gutes Beispiel für den Gebrauch von Makros.

Die Datei mui.e ist in zwei verschiedenen Kopien vorhanden. Einmal im Verzeichnis MacroFiles/ und ein weiteres Mal im Verzeichnis PreAnalysedMacroFiles/. Die erste Version ist in lesbarem Format vorhanden, die zweite als binäre, schon mit PreMac2E behandelte Datei.

Um die MUI-Funktionen in eigenen Programmen zu verwenden, muß Mac2E auf das eigene Programm angewandt werden:

```
Mac2E source.e destination.e PreAnalyzedMacroFiles/mui.e
```

## 1.32 OptiMUI2E

Wenn man sich mui.e ein wenig genauer ansieht, sieht man in den Rümpfen der Makros, die neue Objekte definieren "TAG_IGNORE,0", z.B.

"#define WindowObject Mui_NewObjectA('Window.mui', TAG_IGNORE, 0)"

Dieser Tag tut, wie der Name schon impliziert, absolut nichts während der Ausführung, aber ich war gezwungen, ihn zu benutzen, um denselben Syntax, wie in C zu erhalten. Hier greift OptiMUI2E ein. Seine Aufgabe ist es, diese nutzlosen "TAG_IGNORE, 0" Anweisungen aus dem E Quell-text zu löschen. Seine Aufruf-Syntax ist:

OptiMUI2E FROM e_source_file TO e_destination_file

wobei

- e_source_file den Namen der Quelldatei angibt aus der die TAG_IGNORE Anweisungen gelöscht werden sollen

- e_destination_file den Namen der Zieldatei angibt, die die Quelldatei enthalten wird, aus der alle "TAG_IGNORE, 0"'s entfernt sein werden.

Warnung: OptiMUI2E entfernt manchmal Zeilenumbrüche aus der Quell-datei, um Zeilenumbrüche bei Kommata zu beachten, obligaorisch in E. Deshalb kann es sein, daß die erzeugte Datei nicht die gleiche Anzahl an Zeilen hat, wie die Quelldatei, was wiederum Probleme mit die durch den Compiler beanstandeten Zeilennummern bringen kann. Es wird deswegen dazu geraten, OptiMUIE nur zur letzten Compilation zu verwenden, wenn das Programm getestet ist. OptiMUIE ist zur Verwendung von MUI mit E zusammen nicht notwendig sondern reduziert nur die Größe des Executables, aber nur wenig.

## 1.33 Bekannte Fehler

Keine Angst, dies sind keine Fehler, vielmehr Beschränkungen:

* PreMac2E erkennt nicht, ob Makro-Definitionen rekursiv sind. Sind sie es, bricht PreMac2E einfach ab
* PreMac2E und Mac2E überprüfen nicht, ob ein Makro mehrfach definiert wurde. In diesem Fall benutzt Mac2E einfach irgendeine Definition,
* Die Länge der Makro-Namen ist auf 256 Zeichen beschränkt.
* Die Anzahl der Argumente ist auf 32 beschränkt.
* Die Länge eines Makro-Rumpfes vor dem Ersetzen ist auf 4 kb beschränkt. Wenn das nicht ausreichen sollte, brauchst Du kein Makro sondern eine Funktion.
* Die Länge eines Makro-Rumpfes nach dem Ersetzen ist auf 64 kb be-schränkt, was ausreichen sollte.
* PreMac2E überprüft die vier vorherigen Beschränkungen nicht.

## 1.34 History

```
Version 1.0 :  - erste funktionierende Version (SEHR, SEHR LANGSAM...)

Version 2.0 :  - modifizierte Version mit Assembler-Routinen
        (10 mal schneller !)
               - OptiMUI2E v1.0 hinzugefügt
               - erste ausgelieferte Version

Version 3.0 :  - PreMac2E v1.0 hinzugefügt
               - Benutzung einer Hash-Tabelle (14 mal schneller !)
               - PreMac2E und Mac2E geben Fehlermeldungen aus
               - Speicheranforderungen werden geprüft
               - kleinere Bugfixes
               - OptiMUI2E v1.1 arbeitet nun mit 68000
               - mui.e wurde kommentiert
               - Source für doMethod() beigelegt
               - Source für alle Executables beigelegt
               - besser Dokumentation

Version 3.0 :  - kleinere Bugfixes
```

## 1.35 Verteilung

```
    Refer to the English or French documentation.
```

## 1.36 Zukunftsmusik

```
 Ich warte momentan (wie jeder andere) auf eine neue Version von AmigaE,
was laut Wouter nicht allzulange dauern soll. Natürlich warte ich auch
auf eure Vorschläge...
```

## 1.37 Der Autor

```
    Refer to the English or French documentation.
```

## 1.38 Danksagungen

```
 Heißen Dank an:
        - den Amiga als besten aller Personal Computer
        - Wouter van Oortmerssen wegen seiner Arbeit auf dem Gebiet
    des Compilerbaus (probiert FALSE, Überraschung garantiert !) im
    allgemeinen und Amiga E im besonderen
        - Brian Mury für die englische Übersetzung der Anleitung
        - Marc Schröer für die deutsche Übersetzung der Anleitung
        - Xavier Billault wegen seiner Hilfe bei der Konzeption
    dieser Anleitung
```

```
        - alle auf der French Amiga mailing list, die mir geholfen
    haben
        - alle, die Public Domain Programme schreiben

 Schlußendlich auch Dank an alle, die mir  Bugmeldungen  oder
Verbesserungsvorschläge und Verbesserungen oder Übersetzungen der Dokumen-
tation geschickt haben (siehe  Der Autor ).

    Happy E programming und...

        NEVER FORGET, ONLY AMIGA MAKES IT POSSIBLE!
```

## 1.39 Index