

IN_GO Dokumentation

COLLABORATORS

	<i>TITLE :</i> IN_GO Dokumentation		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		May 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	IN_GO Dokumentation	1
1.1	IN_GO 3.1 REASSEMBLER DOKUMENTATION	1
1.2	Tutorium1	1
1.3	Häufige Fragen zu In_Go	1
1.4	Wie wird Data bestimmt	1
1.5	Wie wird Code bestimmt ?	2
1.6	Was ist In_Go	2
1.7	Einführung	3
1.8	Sinn & Zweck	7
1.9	Copyright & Distribution	8
1.10	COPYRIGHT	8
1.11	DISTRIBUTION	8
1.12	DISCLAIMER	9
1.13	SHAREWARE	9
1.14	Installation	9
1.15	Voraussetzungen	9
1.16	Autor	9
1.17	Updates	10
1.18	Wo ich Hilfe benötige	10
1.19	DANKE	10
1.20	GLOSAR A-Z	10
1.21	AI Aktuelle Zeile	11
1.22	Amigaguide	11
1.23	Icon	11
1.24	FAST-Memory	11
1.25	CHIP-Memory	11
1.26	DATA Segment	11
1.27	CODE Segment	12
1.28	BSS Segment	12
1.29	Hunk	12

1.30 In_Go File	13
1.31 Objectfile	13
1.32 Opcode	13
1.33 RELOC_XX	14
1.34 DRELOC_32	14
1.35 Segment	14
1.36 Marken	14
1.37 Symbolischer Processor	15
1.38 Die	15
1.39 Das Action Menü	15
1.40 Lade Quelle	15
1.41 Lade File	16
1.42 Lade Task	16
1.43 Lade ROM	16
1.44 Lade RAM	17
1.45 Speichere In_Go File	17
1.46 Assembler Textfile abspeichern	17
1.47 Amiga-Guide-File abspeichern	17
1.48 Lauffähiges Programm abspeichern	18
1.49 Object-File abspeichern	18
1.50 Voreinstellung abspeichern	18
1.51 Disassembliere	19
1.52 Drucke sichtbaren Text	19
1.53 Iconifiziere In_Go	19
1.54 Help	19
1.55 New	19
1.56 Jump Menü	19
1.57 Setze Marke	20
1.58 Zeige Marke oder Segment	20
1.59 Springe zum linken Operanden	20
1.60 Springe zum Rechten Operanden	20
1.61 Springe Zurück	20
1.62 Springe zum naechsten Data/Code Wechsel	20
1.63 Springe nächste Tabelle	21
1.64 Springe zum nächsten Jmp(ax)	21
1.65 Springe zum naechsten Fehler	21
1.66 Springe zum nächsten Label	21
1.67 Springe zum vorherigen Label	21
1.68 Springe zum naechsten Langwort-Label	21

1.69	Springe zum nächsten Kommentar	21
1.70	Springe zum Segmentbeginn	21
1.71	Springe zum Segmentende	21
1.72	Springe zum Label	22
1.73	Springe zur Referenz...	22
1.74	Edit Menü	22
1.75	Mache Code zu Data von AI an	22
1.76	Lösche Zeilenlabel in der AI	22
1.77	Lösche alle Code Bereiche	23
1.78	Erzeuge Datalabel in der AI	23
1.79	Erzeuge Jumplabel in der AI	23
1.80	Erzeuge Jumplabel in der AI und Reassembliere	23
1.81	Erzeuge Jumplabel in der AI, reassembliere,nächstes Data	23
1.82	Erzeuge Interuptlabel/Normallabel in der AI	23
1.83	Reassemblerhilfe in der AI für linken Operanden	24
1.84	Reassemblerhilfe in der AI für rechten Operanden	24
1.85	Tabellenhilfe in der AI	24
1.86	Erzeuge lokale Labeln	24
1.87	Benenne typische Amiga Strings	24
1.88	Finde Reloc_32	24
1.89	Vergleiche Reloc_32	25
1.90	Suche Vorwärts	25
1.91	Suche Rückwärts	25
1.92	Line Menü	25
1.93	ZeilenAdressanzeige	25
1.94	Zeige Data als	26
1.95	Zeilenlabelbreite	26
1.96	Data Optionen	26
1.97	Zeige Opcode in Hex	26
1.98	Zeige Library Interpretation	26
1.99	Zeige Wort Interpretation	27
1.100	Zeige Langwort Interpretation	28
1.101	Zeige indirekt-indirekte Zeiger	28
1.102	Zeige lokale Labelform	28
1.103	MISC-Menü	28
1.104	Lerne Strings	28
1.105	Lerne Libs	28
1.106	Lerne libs Prefs	28
1.107	Auto Rea Klick	29

1.108Cleanup	29
1.109Requ Menü	29
1.110Quelle in Hex und ASCII	29
1.111Quelle als Text	29
1.112Bildschirm, Speicher	29
1.113Assembler Voreinstellung	30
1.114Indirekt a4 - a6	30
1.115Pseudo a0-a6	31
1.116Reassembler Voreinstellung	32
1.117Rechner	33
1.118Inhalte	33
1.119Symbolischer Processor	33
1.120Suchen	33
1.121In den Speicher schreiben	34
1.122Speicheranzeige in Hex und ASCII	34
1.123Optimierungen aus der Quelle	34
1.124Prozessor Menü	35
1.125Hunk/Segment Menü	36

Chapter 1

IN_GO Dokumentation

1.1 IN_GO 3.1 REASSEMBLER DOKUMENTATION

Was ist In_Go
Sinn & Zweck
Tutorium1
Einleitung
Installation
Voraussetzungen
Häufige Fragen
Glosar
Menü
Copyright & Distribution
Updates

Autor
Wo ich Hilfe benötige !
Dank an ...

1.2 Tutorium1

Da immer wieder die gleichen Fragen auftauchen habe ich mich entschieden
Ein kleines Programm zu schreiben welches die Funktion von In_Go verdeutlicht.
Text, Programm, Assemblertext finden Sie im Verzeichnis Tutorium.

1.3 Häufige Fragen zu In_Go

Wie kann ich In_Go mitteilen daß ein Bereich Code beinhaltet ?
Wie kann ich In_Go mitteilen daß ein Bereich Data enthält ?

1.4 Wie wird Data bestimmt

Wie kann ich Code in Data umwandeln

Dies sollte normalerweise nur nach einem Reassemblerfehler auftreten !

Daher es wird Code (überwiegend schwarz) angezeigt, obwohl der Bereich als Data ↵
von Ihnen

erkannt wurde !

Gehen Sie mit der Aktuellen Zeile auf den Anfang des falschen Codebereiches,

Wählen Sie nun im Editmenü 'Mache Code zu Data von AL an';

Nun wird der ganze Bereich danach wieder in Data verwandelt.

Dies sollte nur im Notfall angewendet werden !

1.5 Wie wird Code bestimmt ?

Wie kann man In_Go mitteilen daß ein Bereich Code oder Data beinhaltet ?

In_Go kennt nur Code-Bereiche, der Rest wird als Data betrachtet !

Nach erstmaligem Laden existieren keinerlei Code-Bereiche (Anzeige gelb) !

In_Go zeigt zwar den Code an (gelb) er ist jedoch noch nicht bearbeitet !

Durch Drücken der 'Space' Taste wird die eigentliche Reassemblierung (↵
Bearbeitung !) gestartet .

Nun durchforstet In_Go die geladene Quelle von der Startadresse (Segment 0, ↵
Offset 0) beginnend.

Dabei werden immerzu neue Code-Bereiche gefunden. Nach einiger Zeit stoppt In_Go,
weil alle direkten Aufrufe gefunden sind hörbar mit einem Beep.

Erkannter Code wird schwarz (Wenn sich keine andere Farbänderung ergibt) ↵
ausgegeben .

Mit 'd' können Databereiche als Data oder als Code ausgegeben werden
('d' schaltet Ein und Aus) !

Wenn ich nun die Aktuelle Zeile auf den Anfang eines unerkannten Codebereiches ↵
stelle,

wird mit Amiga Taste + 'j' in der Aktuellen Zeile die Reassemblierung erneut ↵
gestartet.

Der Bereich wird nun von In_Go als Code betrachtet !

Dies ist mit Vorsicht anzuwenden !

1.6 Was ist In_Go

Wenn Sie nicht wissen was ein Reassembler ist dürfte dieses Programm
relativ uninteressant für Sie sein.

Mit In_Go können Sie Programme in die kleinsten Schritte der Programmierung
auflösen. Diese Sprache der kleinsten Schritte bezeichnet man als 'Assembler'
Nach der Reassemblierung (Rückübersetzung), kann ein Textfile erzeugt werden,
welches von einem geeigneten Übersetzungsprogramm wieder in lauffähigen Code
umgewandelt werden kann.

Das sind keine einfachen Zusammenhänge !

Wenn Sie damit nicht vertraut sind . . . Finger weg (Reise nach Indien) !

Weitere Informationen finden sie in der Einführung

1.7 Einführung

Rekonstruktion

Stellen sie sich vor: Sie finden in Griechenland die Hand einer Statue, und wollen nun 'anhand' dieser Information die Statue rekonstruieren. Ein ähnliches Problem versucht In_Go zu lösen. Dabei können Sie entscheidend mitwirken.

Drama: Die Jungfrau und der Hammer

Wie jedes andere Werkzeug können Sie auch In_Go mißbrauchen. Sie Können sich Einblicke in anderer Leute Programm verschaffen (die Jungfrau mit dem Hammer erschlagen), aber auch Ihre eigenen Machwerke 'durchsichtig' machen. Die meisten größeren Programme werden ja Heute in Compilersprachen geschrieben. Hier kommt es nicht selten zu merkwürdigen Problemen infolge falschen Implementationen. In_Go ermöglicht Ihnen nun einen Einblick auf unterster Ebene, dabei können Sie auch laufende Tasks beobachten, und deren Variablen auslesen.

Was kostet die Welt ?

2.147.483.647 das ist die größte vorzeichenbehaftete Zahl die der 68000er intern verarbeiten kann: der implementierte Kalkulator von In_Go beherrscht diesen Bereich mit ca. 40 verschiedenen logischen und arithmethischen Funktionen.

Zeige mir den Zeiger

Die erste Betrachtung des Reassemblercodes führt oft zu Enttäuschungen; man kann insbesondere die indirekten Zugriffe über die Adressregister, zb. CLR -\$7800(A5) nicht richtig zuordnen. In_Go kann diese Adressen richtig zuordnen, wenn die Basis eine Konstante ist. Für A6 jedoch können nur positive Werte verarbeitet werden, negative werden ja als Libraryaufruf interpretiert. Zusätzlich können die gültigen Bereiche für diese Auslegung begrenzt werden.

Wohin soll Das noch führen ?

Nach dem Laden, (D)Reloc_32 und Namen werden eingelesen, kann durch Anwahl von 'Disassemble' (Space !) die Reassemblierung gestartet werden. Dabei wird von der Startadresse an versucht, die Codeteile zu finden. Beim einlesen von 'RTS' oder 'JMP' wird die Reassemblierung an anderer Stelle fortgeführt. Dabei werden immerzu neue Labels und Bereiche gefunden bis auch der letzte direkte Aufruf bearbeitet ist. In den meisten Fällen ist nun keineswegs das ganze Programm decodiert, da In_Go bei den gefundenen Adressen eine strenge Prüfung vornimmt und nur ausdrückliche (BSR, BRA, Bxx, JSR, JMP) Sprungadressen automatisch reassembliert. Verschiedene Codebereiche bleiben bei diesem Algorithmus unentdeckt. Mit hilfe von 'n' können solche Bereiche gefunden werden und, interaktiv, durch Amiga + 'j' reassembliert werden.

Wer ist tabellensüchtig ?

Tabellen kommen in manigfaltigen Variationen vor. In_Go bietet geeignete Instrumente zum finden und einlesen derselben:

- relative Wortreferenzen zu einer Basisadresse, die Basisadresse muß dabei nicht identisch mit dem Tabellenanfang sein, dies wird jedoch als der Normalfall angesehen.
- absolute Langwortadressen.
- selbstbezügliche Wortreferenzen.
- Labelerzeugung in eingestellter Distanz.
- Verkettete Listen mit unterschiedlich langen Elementen.

Die Distanzen können in 2 Byte Abständen frei gewählt werden.

Die Ausgabe im Display zeigt dabei in welcher Form die Labelcreation zustande kam: zB. dc.w L100-L103+2

Sie können auch die Labelart der errechneten Label bestimmen !

Da solches Tun den höheren Künsten zuzurechnen ist, kann

In_Go ca 100 verschiedenartige Tabellen erkennen !

Wie heißt das Kind ?

Sie können jede Adresse innerhalb der Quelle benennen.

Die Programmzeile kann mit einem Kommentar versehen werden.

Libraryaufrufe werden nach Festlegung der Basen als Mnemonics angezeigt. Die Lerne Funktion unterstützt Sie bei der Namensgebung jener kleinen Librarysubroutinen die für viele Kompilate typisch sind, der Name wird dabei erweitert.

Wie wärs mit Abschalten ?

Wenn Sie die Bearbeitung zu einem anderen Zeitpunkt fortführen wollen, können Sie Ihre Geistesblitze abspeichern, dies ist auch bei Tasks möglich !

Doch er fand das Label nicht....

Mit dem implementierten Sucher kann fast alles, auch indirekte und relative Aufrufe gefunden werden. Die Suche kann auf das momentane Segment begrenzt werden. Ein Menü hilft Wichtiges (Openlibrary) zu finden. Da In_Go keine Texte im Hintergrund bereithält kann ein Ausdruck wie 'add #5,d0' nur in der Form seines Opcodes gefunden werden !

Und wer will mich ?

Nach erfolgreichem Reassemblieren liegen alle Referenzen vor.

Bewege ich nun den Zeilenkursor auf ein bestimmtes Label, kann mit 'r' ein Requester aufgerufen werden der alle Aufrufe Dieser Adresse ausgibt.

Nur durch Reloc_32 gefundenen Label bleiben unberücksichtigt.

Ich würde es ja verstehen: Wenn ich nur die Source hätte !

Dem kann abgeholfen werden: Nachdem möglichst alle Codebereiche gefunden wurden, kann ein Textfile erzeugt werden.

Was heißt hier Früchte, ich will Bananen !

Leider sind Assembler ziemlich unterschiedlich, Einige fordern dies, andere geben bei jenem eine Fehlermeldung aus.

Daher sind verschiedene Assecoirs schaltbar.

Fehler über Fehler

Beim Reassemblieren wird jeder nichterlaubte Opcode angezeigt.
In vielen Fällen kann so auch selbstmodifizierender Code gefunden werden (Anzeige mit 'e').

Gehts nicht noch'n wenig kleiner ?

Lokale Label zu erzeugen, ist alles andere als einfach. Dabei müssen jegliche Zugriffe innerhalb Programms erkannt werden !
In_Go hats !

Jetzt hab ichs ! ... und weiß genau soviel wie vorher.

Assembler Prgs sind oft nach Spagetti-Manier geschrieben: Da ein Sprüngchen ... auf einen anderen Sprung der wiederum....usw
Wie soll man da zusammenhängendes erkennen ?
In_Go bietet die Möglichkeit lokale Label anzulegen !
Dieses mehr unscheinbare Tool hats in sich:
Zusammenhängendes wird nicht mehr auseinandergerissen...

Wie löß ich den Knoten ?

Alle mit In_Go erzeugten Amigaguide-Files enthalten 'Referenzen' am Ende jeder Node, die fast alle Aufrufe dieser Node enthalten, ausgenommen sind Aufrufe über einige Tabellen (selten !).
In Form eines Amigaguide-Files kann so eine kaum übertreffbare Transparenz erreicht werden ! Guide Files können auch ohne lokale Label ausgegeben werden, es mangelt jedoch an ... Transparenz.

Panne ? Hät' ich doch ein Reserverad !

Viele Programme allocieren erst während des ablaufens Speicher, in dem dann Variable angelegt und erwartet werden.
Da dieser Bereich dynamisch angelegt sein kann, scheitert jeder Versuch einer absoluten lokalisierung..
Für solche Fälle können im F3 Requester Pseudo-Bereiche angelegt werden. In diesen Pseudobereichen werden dann während des Reassemblierens Variable angelegt die man auch benennen kann.

Jetzt wirst Du gelinkt

Insbesondere 'C' besorgt sich den Bereich für lokale Variablen mit Hilfe von 'Link' Ax ! Solche 'Proceduren' können von In_Go im 'F3' Requester gesucht werden und als 'Bereiche' angelegt werden. Dies erfolgt weitgehend selbsttätig für Register A4 - A6 , nach Anklicken der entsprechenden Option.

Unterprogramm... ins Nichts ?

Manche Programme benutzen Fehlerrouitinen (oder ähnliches !) die mit Bsr oder Jsr aufgerufen werden, nach Aufruf jedoch wird die Herkunftsadresse vom Stack genommen und zu einem anderen Programmteil verzweigt. Wenn solche Label in der aktuellen Zeile mit

'Erzeuge Interruptlabel' Behandelt werden kann ein Reassemblerfehler verhindert werden.

Reloc oder Dreloc, ist das Was von Shakespeare ?

In_Go bietet die Möglichkeit ein lauffähiges Modul mit Labeln und Namen abzuspeichern. (Sie haben sich nicht verlesen !)

Verschiedene Optionen werden unterstützt:

- Nur eigene (spezifische) Namen in Code als Symbolhunk für einen Profiler
- Alle eigenen Namen als Symbolhunk
- Alle Label als Symbolhunk

Optimierung Reloc_32 nach Dreloc_32 ist möglich wenn die Offsets >=0 und <65500 sind, jedoch nur Global.

Dies ist jedoch meist nur bei kleineren Prgs möglich !

Auch mit 'In den Speicher schreiben' vorgenommene Änderungen werden abgespeichert.

(Jedoch bitte keine A_ Adressen oder (D)reloc_32 Einträge verändern !)

Der Name des geladenen Prgs wird dabei mit 'q' erweitert.

Das ROM als lauffähiges Programm ?

Sicherlich das nicht; aber Überschaubarkeit. Die im Rom gelegenen Libraries werden als Mnemonic angelegt (ca. 1000) dies sollte auch bei nachgeladenen Kickstarts funktionieren.

Indirekt indirekt - ist das ein Witz ?

Nun jedenfalls gibt es das, und das kann einem ganz schön plagen, da das eigentliche Sprungziel 'verschleiert' ist.

Mit Einschalten von <indirekt indirekt' kein Problem.

Solche Fälle werden auch ins Amigaguidefile mit übernommen.

Enforce den Enforcer ?

Der Enforcer kann Enforcerhits eines Prgs mit Hunk + Offset ausgeben, wenn Segtracker installiert ist.

Bei In_Go ist diese Adresse durch Umschaltung mit 'z' leicht zu erkennen !

Kein Papagei ist so bunt !

Farben erleichtern die Übersicht. Da läßt sich schnell erkennen ob dieses oder jenes in der Zeile ist. Das ist auch der Grund weshalb In_Go einen Screen mit 8 Farben öffnet !
Farbliche Unterschiede finden sich auch in Assembler-Guidefiles.

So primitiv war noch kein Symbolprocessor !

Aber doch.. er hilft einfache Sachverhalte zu erkennen.
Insbesondere erkennt er das Hin-und Herschieben im Processor.
Dies können Library-Basen sein oder ähnliches, die für eine weitergehende Interpretation benötigt werden.

Darfs ein bißchen mehr sein ?

Neuere Programme sind oft mit nur auf dem 68020 lauffähigen Code versehen. Nach Einschalten der entsprechenden Option werden diese richtig (Schreib 'nen Brief wenn nicht !) decodiert.

Ich wills fließend...

Viele Amigas besitzen einen Math-Coprocessor. Nach Einschalten der entsprechenden Option werden diese Opcodes richtig zugeordnet. Die gängigsten Real-Formate werden ebenfalls decodiert.

1.8 Sinn & Zweck

Philosophie

Beim Interpretieren eines fremden Sachverhaltes treten zwei grundsätzliche Prinzipien in Erscheinung:

- a) 'Gewissheit'
- b) 'Wahrscheinlichkeit'

Fall a) ist langwierig, er erfordert die volle Durchmusterung der Quelle: Jeder Fall muß entschieden werden !

Fall b) Es wird eine Interpretation nach der Wahrscheinlichkeit durchgeführt. - Dies kann jedoch zu Fehlern führen

In_Go arbeitet nach dem zweiten Prinzip. Beispiel:

```
movea.l $4,A6          ; 1.Stufe jetzt befindet sich _ExecBase in A6
....
jsr      _LVOGetMsg(A6) ; 2.Stufe Aufruf der Exec Library
....
           ; 3.Stufe Dieser Aufruf liefert
....
           ; eine Message-Struktur in D0
movea.l D0,A0          ; 4.Stufe MessageStruktur nun auch in A0
move.l seconds(a0),a2  ; 5.Stufe wäre nur bei einer Intuition_Msg
.....
           ; richtig. Bei einer WBStartup-Msg wäre
           ; sm_ArgList(a0) richtig !
```

Sie sehen also, Fehler sind möglich....

Richtige Namensgebung kann für eine automatische Interpretation nützlich sein !

Beispiel:

```
move.l Unser_Screen,a0
```

der Preprocessor geht nun davon aus das a0 auf eine Screenstruktur zeigt.

Durch den Kommentar '[d0]=_WBStartup' läßt sich ein Register des Symbolischer Processor richtigstellen !
Der Kommentar ist jedoch erst in der Folgezeile wirksam.

In_Go speichert keine Texte, die zur Ausgabe bereitgehalten werden,

Der jeweilige sichtbare Text wird erst bei der Ausgabe erzeugt. Im Interpretationsfalle kann so von falschen Registerinhalten ausgegangen werden. Der große Vorteil einer solchen Handhabung: bearbeitbare Prgs können als Textfile wesentlich größer als der zur Verfügung stehende Speicher sein. Der Nachteil: Fehler bei der sichtbaren Ausgabe sind möglich !

Übrigens auch dieser Text ist noch in Arbeit Stand April 97

1.9 Copyright & Distribution

COPYRIGHT

DISTRIBUTION

DISCLAIMER

SHAREWARE

1.10 COPYRIGHT

In_Go
ist copyrighted (C) 1996 by Ingo Molter.
Das bedeutet, daß es NICHT ERLAUBT ist,
irgendeinen Bestandteil des Programms, der Anleitung
oder sonst einem enthaltenen File in irgendeiner Weise
zu verändern. Außerdem darf weder diese Anleitung noch
ein anderes File aus dem Archiv gelöscht werden.

Es ist NICHT erlaubt, dieses Programm in einer
anderen Weise zu benutzen, als es in dieser Anleitung
beschrieben ist.

In_Go ist SHAREWARE!

1.11 DISTRIBUTION

In_go darf frei kopiert werden.
Das bedeutet, daß jeder dieses Archiv verbreiten darf,
solange folgende Punkte erfüllt sind:

- Das Archiv muß intakt bleiben, es ist NICHT ERLAUBT,
irgendwelche Files aus diesem Archiv zu löschen
oder hinzuzufügen.
 - Dieses Archiv darf frei über Mailboxen,
das InterNet sowie über PD-Serien vertrieben werden.
Der Vertrieb auf anderen Datenträgern bedarf
meiner Genehmigung.
 - Disketten-Magazine oder Anbieter,
-

die spezielle Gebühren für den File-Transfer verlangen dürfen das Archiv NICHT ohne schriftliche Genehmigung von Ingo Molter verbreiten. Bei Aufnahme von In_Go in eine PD-Library, eine CD-ROM o.ä. würde ich mich sehr über ein Freiemplar oder zumindest über eine Benachrichtigung freuen.

1.12 DISCLAIMER

Der Benutzer dieses Produkts übernimmt die VOLLE Verantwortung für Schaden und Fehler aller Art, die durch die unsachgemäße Anwendung dieses Produkts entstehen könnten. Der Autor dieses Programms kann dafür NICHT verantwortlich gemacht werden.

1.13 SHAREWARE

Dieses Programm ist SHAREWARE.
Frei kopierbar bedeutet nur, daß es erlaubt ist, die unregistrierte Version des Programms zu kopieren. Das Programm darf für 30 Tage getestet werden, danach sollte man sich registrieren LINK UPD} lassen.

1.14 Installation

Einfach das In_Go Archiv auf Ihren Datenträger ziehen

1.15 Voraussetzungen

Mindestens Kickstart 2.04
Mindestens 2 MB Ram

1.16 Autor

Ingo Molter

Dunzweilerstr.42
D-66903 Dittweiler
Tel. 06386-1312
Aber: Sende einen Brief !

1.17 Updates

Bitte senden Sie 30 Mark oder 20 \$ an mich.
Sie erhalten dann eine ausführliche Bedienungsanleitung !

Ingo Molter
Dunzweilerstr.42

D-66903 Dittweiler

Ich benötige jedoch auch Hilfe

1.18 Wo ich Hilfe benötige

In_Go ist keineswegs perfekt
Es wurde zwar ausgiebig getestet,

Wenn Sie einen Tip haben, senden Sie mir einen an Sie
adressierten mit 3 DM frankierten Briefumschlag.
Sie erhalten dann die neueste Version

Ich benötige auch jemand der mir hilft eine
Englische Version herauszugeben.

1.19 DANKE

Siehe im eingebauten Helpertext von In_Go

1.20 GLOSAR A-Z

AL AKTUELLE ZEILE
AMIGAGUIDE
AMIGAGUIDEFILE SPEICHERN
BSS

CHIP RAM
CODE

DATA
DRELOC_32
FAST

HUNK

IN_GO FILE

LAUFFÄHIGES PROGRAMM ABSPEICHERN

MARKE

```
OBJECTFILE
OPCODE

PROFILER

RELOC_8
RELOC_16
RELOC_32

SEGMENT
SYMBOLISCHER PROCESSOR
```

1.21 AI Aktuelle Zeile

Als Aktuelle Zeile wird diejenige Zeile angesehen die von einem roten Cursor eingerahmt ist.

1.22 Amigaguide

Als Amigaguide-File wird ein File bezeichnet bei dem Querverweise durch Anklicken angezeigt werden können.
Dieses Helpfile ist ein Amigaguide-File !

1.23 Icon

Alle von In_Go erzeugte Textfiles können mit Icons versehen werden.

1.24 FAST-Memory

Meist ist das nachträglich eingefügte RAM Modul FAST-RAM
Der 680X0 Processor hat hier den alleinigen Zugriff !

1.25 CHIP-Memory

Der 500 er hatte in seiner Originalversion 500 kB CHIP-RAM

Der 1200 er hat 2 MB CHIP-RAM
Der Grahpic-Processor kann nur auf CHIP-RAM zugreifen.

1.26 DATA Segment

Ein lauffähiges Prg besitzt meist auch Datasegmente.
Dies ist nie das erste Segment !
Datasegmente können jedoch auch Code enthalten !

1.27 CODE Segment

Ein lauffähiges Prg besteht aus mindestens einem CODE Segment

1.28 BSS Segment

Ein BSS Segment wird erst von der Amiga-Laderoutine belegt. Im Abgespeicherten Prg wird nur ein Hunk belegt.

Eine Adresskorrektur im CODE Segment erfolgt dann durch die Loadseg Routine der Dos Library.

Dies ist im Grunde ein Data Segment, nur das hier alle vorgegebenen Daten uninitialized oder Null sind !

Beim Laden eines linkbaren Objectmodules expandiert In_Go diese Bereiche nicht, so das Offsets ausgegeben werden.

1.29 Hunk

Der Begriff Hunk ist teilweise mit dem Begriff Segment identisch.

Lauffähige Module bestehen meist aus mehreren Segmenten wobei allgemein 3 Arten unterschieden werden

CODE	Hier kann alles stehen auch Data !
DATA	Hier werden Daten ausgelagert.
BSS	Hier werden nur Speicherbereiche in der geforderten Größe angefordert.

Der tiefere Sinn einer solchen Aufteilung:

- Der angeforderte Speicherbereich kann kleiner als die gesamte Programmlänge sein; Dies ist insbesondere bei stark fragmentiertem Speicher notwendig.
- BSS Bereiche machen das abgespeicherte Programm nur unwesentlich länger. Dieser Speicher wird erst beim laden des Prgs durch die Loadseg Routine allokiert.
- Für manche Anwendungen ist eine bestimmte Speicherart notwendig. Manche Coprocessoren haben nur Zugriff auf das CHIP-RAM.
- Die CPU hat zwar Zugriff zu allen Speicherarten jedoch erfolgt ein Zugriff aufs FAST-RAM meist schneller als aufs CHIP-RAM, da die Coprocessoren teilweise Parallel aufs CHIP-RAM zugreifen können !

Da der Amiga ein Multitask Computer ist müssen alle Programme an verschiedenen Adressen lauffähig sein Daher enthält fast jedes Programm auch Tabellen von Adressen, die vor dem Starten berichtigt werden müssen.

Diese Tabellen werden zum Beispiel Reloc_32 Hunk genannt.
In linkbaren Objectfiles können noch andere Hunks auftauchen.

1.30 In_Go File

Ein .In_Go File ermöglicht das Zwischenspeichern
Ihrer Arbeit mit In_Go.

Normalerweise wird das In_Go File im gleichen
Pfad wie das bearbeitete File abgespeichert.

Ein In_Go File enthält:

Marken
Reloc_32

1.31 Objectfile

Es gibt grundsätzlich drei Arten:

- Lauffähige Objectfiles
Diese können von der Workbench oder dem CLI gestartet werden.
Die Hunkrelation stimmt hier.
- Linkfähige Objectfiles die ein lauffähiges Objectfile ergeben
Die letzte Stufe einer Compilation die nur noch mit einer
bestimmten Library gelinkt werden müssen.
Die Hunkrelation ist hier richtig:
Ein Verweis auf Hunk 2 stimmt hier
Im Laderequester kann diese Art angeklickt werden !
- Objectfile-Kollektionen wie zB die Amiga-Lib
Hier werden viele Objectfiles aneinandergefügt. Die Hunkrelation
ist falsch: Ein Verweis auf Hunk 2 zeigt hier NICHT auf den
2. Hunk der Kollektion.

Erst durch Linken der entsprechenden Teile mit anderen Teilen
ergibt sich als Endergebnis ein lauffähiges Modul.

1.32 Opcode

Wenn eine bestimmte Instruktion im Speicher als eine
Folge von Hexzahlen existiert, so wird diese Hexzahl
von mir als Opcode bezeichnet.

Beispiel: RTS erscheint als \$4e75

1.33 RELOC_XX

In lauffähigen Files sorgen Reloc_32 Tabellen für Lauffähigkeit an verschiedenen Adressen
Es handelt sich dabei um eine Tabelle die anzeigt welche Adresse 'Angepasst' werden muß

Reloc_8 und Reloc 16 Tabellen tauchen nur in linkbaren Objektfiles auf.

Siehe auch Hunk LINK HUNKG}

1.34 DRELOC_32

In lauffähigen Files sorgen Dreloc_32 Tabellen für lauffähigkeit an verschiedenen Adressen.

Dreloc_32 Hunks sind platzsparender gegenüber Reloc_32 Hunks
Diese Hunkart ist nur bei kleineren Prgs möglich !

1.35 Segment

Ein Segment in linkbaren Objectfiles kann aus mehreren Hunks bestehen.

Besteht ein Segment nur aus einem Hunk sind beide Begriffe identisch.

Die Umschaltung der Segment erfolgt mit <+> und <->

Siehe auch In_Go File

1.36 Marken

Marken sind vom Benutzer festgelegte Punkte
Sie sind nur für die Anzeige von Bedeutung.
In_Go unterstützt 10 Marken von 0 bis 9.

Marken liegen grundsätzlich innerhalb der geladenen Segmente !

Setzen einer Marke mit 'Amiga Taste' + '0-9' Taste
Abrufen einer Marke mit Taste '0-9'

Sind Marken nicht belegt wird zu Segmentnummer verzweigt

Siehe auch In_Go File

1.37 Symbolischer Processor

Der Symbolische Processor versucht das Hin- und Herschieben von Registerinhalten zu simulieren.

Dies hat natürlich dort seine Grenzen wo auf Speicherinhalte, Stack usw. zurückgegriffen wird.

Wie auch immer... mit F6 kann der Inhalt für jede angezeigte Zeile angezeigt werden.

Mit einem Kommentar kann eine Belegung erzwungen werden !

Beispiel: Kommentar [A0]=_WBStartup

Die Belegung ist erst in der folgenden Zeile wirksam.

In der vorliegenden Version hat diese Belegung keinen Einfluß auf die Wortinterpretation der Bereiche im F3 Requester da dieser eine höhere Priorität zugeordnet ist !

1.38 Die

- Acti
- Jump
- Edit
- Line
- Misc
- Requ
- Processor
- Hunk/Segment

1.39 Das Action Menü

- Laden
- .In_Go abspeichern
- Assembler Textfile abspeichern
- Amiga-Guide File abspeichern
- Lauffähiges Modul Abspeichern
- Objectfile abspeichern
- Voreinstellung Abspeichern
- Disassembliere
- Drucke sichtbaren Text
- Iconify
- Help
- New

1.40 Lade Quelle

- Lade File
- Lade Task

Lade ROM
Lade RAM

Nach dem Laden werden hier die Segmente angezeigt.
Dabei kann auch die Hunkart durch Doppelklick editiert werden.

Beim erstmaligen Laden wird zum Reassemblerpreferencesrequester verzweigt.

1.41 Lade File

Hier existieren 5 Schalter:

- 'Interpretiere Reloc_32'
Sollte immer eingeschaltet sein !
- 'Interpretiere Langtabellen in Codesegmenten, als Sprung'
Einige Compilare legen Langworttabellen in Codesegmenten an,
die hiermit als Reassemblerrestartadressen interpretiert werden.
Mit Vorsicht anwenden !
- 'Lese Symbol Hunk Units'
Sollte immer angeschaltet sein !
- 'Sprünge in Data Segmente existieren'
Einige Compiler generieren Sprünge in Data Segmente, dies führt
dann zu Interpretationsfehlern wenn der Codeteil im Datasegment
Reloc_32 Adressen beinhaltet.
Normalerweise immer aus !
- 'Reales Prg. (nur für linkfähige Objectfiles)
Nur von Interesse wenn linkfähige Objectfiles geladen werden !
Hier wird zwischen einer Objectfilecollection und der letzten
Stufe eines Compilers unterschieden.
Allgemein gilt: Stimmen die Hunkrelationen des Objectfiles dann
einschalten !

1.42 Lade Task

Wenn Sie bereits ein 'in_go' File der entsprechenden Task gespeichert
haben, so sollten Sie erst den In_Go Pfad anwählen.
Selectieren Sie die Task durch Anklicken im Requester

Bedenken Sie Bitte das eine Task keinerlei Hinweise auf Reloc_32
Daten enthält !

1.43 Lade ROM

In_Go ermöglicht auch das ROM einzuladen !
Dabei werden ca. 1000 Symbole erzeugt.
Dies sollte auch bei nachgeladenem ROM funktionieren.

Dabei kann es zu Enforcerhits kommen da In_Go auch die Zero-Page auswertet. Beim erstmaligen Einladen sollten Sie alle Schaltmöglichkeiten einschalten.

1.44 Lade RAM

Wenig empfehlenswert !
Ergebnis Kann nicht abgespeichert werden !

1.45 Speichere In_Go File

Hier kann ein File mit allen Labeln, Referenzen, Relocs, Labelnamen usw. abgespeichert werden. Die sollte in der Regel im gleichen Verzeichnis wie das geladene File, passieren.
Dort wird es bei nochmaligen Laden auch gesucht.
Dieses File enthält nicht das eigentliche Programm !
Durch laufendes Speichern kann eine Art UNDO erreicht werden, dies sollte insbesondere bei unsicherem Tabelleneinlesen erfolgen.

1.46 Assembler Textfile abspeichern

Hier kann ein ASCII-File der geladenen Quelle abgespeichert werden. dabei sind verschiedene Optionen möglich:

- Liste mit Segmenten Einfügen.
- Liste der Code Bereiche einfügen.
- Labelliste einfügen.
- Sende keine undefinierten Segmente.
Dies ermöglicht die Beseitigung von nicht benötigten Segmenten.
- Versionstring einfügen
- Icon mitspeichern.

1.47 Amiga-Guide-File abspeichern

Hier kann ein ASCII-File der geladenen Quelle im Amigaguide-Format abgespeichert werden.
dabei sind verschiedene Optionen möglich:

- Liste mit Segmenten Einfügen.
 - Liste der Code Bereiche einfügen.
 - Labelliste einfügen.
 - Sende keine undefinierten Segmente.
Dies ermöglicht die Beseitigung von nicht benötigten Segmenten.
-

- Versionstring einfügen
- Icon mitspeichern.

1.48 Lauffähiges Programm abspeichern

Hier kann ein geladenes lauffähiges Programm wieder abgespeichert werden. Dabei sind verschiedene Optionen möglich:

- Debug-Hunks mit allen Labeln einfügen
- Debug-Hunks mit Namen in Code einfügen. Diese Option ermöglicht das sparsame speichern von eigenen Labelnamen die in Code stehen. Der tiefere Sinn dabei; ein Profiler, also ein Programm das die Laufzeiten der verschiedenen Programmteile aufzeichnet wird nicht durch balastartige Datalabel verwirrt !
- Debug Hunks nur mit eigenen Namen einfügen. Hier werden also keine In_Go erzeugten Labelnamen in der Form (A) (J) (L)+Hunk+'_'+'Offset abgespeichert.

Wenn es sich um kleinere Programme handelt kann In_Go die Reloc_32 Einträge auch in Dreloc_32 Einträge umwandeln. Das spart Speicherplatz! Diese Option ist nur für das ganze Prg verfügbar.

1.49 Object-File abspeichern

Auch hier existieren wieder verschiedene Modi.

Damit lassen sich Objectfilecollectionen editieren !

- | | |
|-------------------------------|---------------------------------------|
| - Alles zusammen | Selbstredend |
| - Nur sichtbares Segment | Selbstredend |
| - Sichtbares Segment nicht | Unterdrücke sichtbares Segment |
| - Alles spliten | Jedes Segment separat |
| - Tausche sichtbaren Hunk aus | Das sichtbare austauschen mit anderem |
| - Füge anderes Object ein | Ein Segment einfügen |

1.50 Voreinstellung abspeichern

Die Voreinstellung wird im S: Verzeichnis abgespeichert.

1.51 Disassembliere

Nach Anwahl dieses Punktes startet die Reassemblierung.

Erreichbar durch Leertaste

Abbruch durch Mausklick in die Scrollbox.

1.52 Drucke sichtbaren Text

Der gerade in der Anzeige sichtbare Text wird ausgegeben.

1.53 Iconifiziere In_Go

In_Go schließt seinen Screen und ist auf der Workbench als Icon wieder aufrufbar.

1.54 Help

Ein Helptext wird angezeigt.

Nur in der Hauptanzeige wirksam.

1.55 New

Alle Ressourcen werden zurückgegeben, Ingo startet erneut.

1.56 Jump Menü

Setze Marke
Zeige Marke oder Segment
Springe zum linken Operanden
Springe zum rechten Operanden
ZurückSpringen
Springe zum nächsten Data/Code Wechsel
Springe nächste Tabelle
Springe nächstes jmp(Ax)
Springe nächsten Fehler
Springe zum nächsten Label
Springe zum vorherigen Label
Springe zum nächsten Langwort Label
Springe zum nächsten Kommentar
Springe zum Segmentbeginn
Springe zum Segmentende
Springe nach Label ...
Springe zur Referenz ...

1.57 Setze Marke

In_Go unterstützt bis zu 10 Marken LINK MARKE} im Programm
Durch Drücken von Amigataste + 0-9 wird eine Marke
gesetzt.

Durch Drücken der Taste 0-9 wird die entsprechende
Marke angesprungen.

Marken werden auch im In_Go File abgespeichert

Siehe auch Zeige Marke oder Segment

1.58 Zeige Marke oder Segment

Durch Drücken der Taste 0-9 wird die entsprechende
Marke angesprungen.

Ist eine Marke unbelegt wird zum entsprechenden
Segment verzweigt.

1.59 Springe zum linken Operanden

Es wird versucht zum linken Operanden zu springen.
Die Letzte Anzeigeadresse wird abgespeichert.
Mit <Kursor links> kann wieder zur
letzten Anzeigeadresse zurückgesprungen werden.
max. Tiefe 10

1.60 Springe zum Rechten Operanden

Es wird versucht ab dem rechten Operanden anzuzeigen.
Die Letzte Anzeigeadresse wird abgespeichert.
Mit <Kursor links> kann wieder zur
letzten Anzeigeadresse zurückgesprungen werden.
max. Tiefe 10

1.61 Springe Zurück

Tastatur <Kursor links>

1.62 Springe zum naechsten Data/Code Wechsel

Funktioniert immer vorwärts !

1.63 Springe nächste Tabelle

Es wird versucht eine mögliche Sprungtabelle anzuzeigen !

1.64 Springe zum nächsten `Jmp(ax)`

Springt zum nächsten Indirekten Sprung `jmp (ax)` oder `jsr (ax)`

1.65 Springe zum naechsten Fehler

Alle Reassemblerfehler werden nacheinander angezeigt.

1.66 Springe zum nächsten Label

``>`` Aktuelle Zeile zum nächsten Label

1.67 Springe zum vorherigen Label

``<`` aktuelle Zeile zum letzten Label

1.68 Springe zum naechsten Langwort-Label

Springt zum naechsten Langwortlabel

1.69 Springe zum nächsten Kommentar

Springe nächsten Kommentar an

1.70 Springe zum Segmentbeginn

selbstredend

1.71 Springe zum Segmentende

selbstredend

1.72 Springe zum Label ...

Ein Requester wird angezeigt: Das angeklickte Label wird angezeigt

1.73 Springe zur Referenz...

Existieren Referenzen auf die Aktuelle Zeile,
Wird ein Auswahl-Requester angezeigt.

Durch anklicken wird die aufrufende Stelle angezeigt.

1.74 Edit Menü

Mache Code zu Data von A1 an
Lösche Zeilenlabel in der A1
Lösche alle Codebereiche
Erzeuge Datalabel in der A1
Erzeuge J1 in der A1
Erzeuge J1 in der A1 und reassembliere
Erzeuge J1 in der A1, reassemble und springe nach Data
Erzeuge Interruptlabel/NormalLabel in der A1
Reassemblerhilfe in der A1 für Operand 1
Reassemblerhilfe in der A1 für Operand 2
Tabellenhilfe in der A1
Erzeuge lokale labeln
Benenne typische Amiga Strings
Finde Reloc32 (nur für Tasks)
Vergleiche Reloc_32 (nur für Tasks)
Suche vorwärts
Suche rückwärts

1.75 Mache Code zu Data von A1 an

Bitte nur im Notfall benutzen da vorher gefundene Referenzen
nicht mitgelöscht werden.
Dies kann zu Problemen führen...

1.76 Lösche Zeilenlabel in der A1

Wenig empfehlenswert:
Referenzen bleiben erhalten.
Das Löschen von durch Reloc_32 erzeugten Labeln ist nicht möglich !

1.77 Lösche alle Code Bereiche

Werden im F3 Requester neue Bereiche eingetragen so ist eine erneute Reassemblierung fällig !
Die Labeln werden beibehalten.

1.78 Erzeuge Datalabel in der AI

Ein Label vom Typ 'L' wird angelegt
Diese Label wird NICHT automatisch vom Reassembler angesprungen.

1.79 Erzeuge Jumplabel in der AI

Ein Label vom Typ 'JL' wird angelegt.
Beim Nächsten Reassemblerstart wird ab diesem Label reassembliert.

1.80 Erzeuge Jumplabel in der AI und Reassembliere

Wie Erzeuge Jumplabel in der AI
Anschließend wird jedoch Reassembliert

1.81 Erzeuge Jumplabel in der AI, reassembliere,nächstes Data

Wie Erzeuge Jumplabel in der AI
Nach dem reassemblieren wird zum nächsten Data/Code Wechsel gesprungen.
Damit können auch unsichere unerkannte Codeteile reassembliert werden.

1.82 Erzeuge Interruptlabel/Normallabel in der AI

Manche Programme verzweigen mit 'bsr xxx' oder 'jsr xxx' zu bestimmten ↵
Unterprogrammen.
Innerhalb dieser Unterroutinen wird die Herkunftsadresse vom Stack geholt, und zu einem anderen Programmteil verzweigt.
Befinden sich hinter dem Aufruf Datas führt dies zu Reassemblerfehlern.

Fahren sie mit der Aktuellen Zeile auf die Routine auf dieses Label (nicht auf ↵
den Aufruf !),
wählen Sie diesen Punkt an, beantworten Sie den Requester Richtig.
Trifft der Reassembler nun auf einen Aufruf jsr/bsr
wird dahinter nicht automatisch weiter reassembliert.

Bei nochmaliger Anwahl kann dieser Zustand wieder beseitigt werden.

1.83 Reassemblerhilfe in der AI für linken Operanden

Der linke Operand wird im Reassemblerhelprequester zur Edition angeboten.

Handelt es sich um ein Register oder ähnliches wird der rechte Operand genommen !

1.84 Reassemblerhilfe in der AI für rechten Operanden

Der rechte Operand wird im Reassemblerhelprequester zur Edition angeboten.

Handelt es sich um ein Register oder ähnliches wird der linke Operand genommen !

1.85 Tabellenhilfe in der AI

Hier wird ein Tabellenhilfsrequester angezeigt
Die Aktuelle Zeile wird als Adresse übergeben.

1.86 Erzeuge lokale Labeln

Sollte erst nach einer vollständigen Reassemblierung angewendet werden.

Diese unscheinbare Tool hats in sich . . .

1.87 Benenne typische Amiga Strings

Mit diesem Aufruf werden Amigatypische Strings benannt.
Sie werden also als Label abgespeichert.

1.88 Finde Reloc_32

Da bei Tasks keine Reloc_32 Tabelle existiert
werden hiermit alle Segmente auf Adressen untersucht
die in die Segmente zeigen.

Durch Laden an verschiedenen Adressen und anschliessendes
Aufrufen der Funktion Vergleiche Reloc_32
werden alle rein zufällig in die Segmente zeigenden Zeiger
eliminiert.

Dies ist alles etwas problembehaftet!

Diese Funktion kann nur bei Tasks angewählt werden !

1.89 Vergleiche Reloc_32

Da bei Tasks keine RELOC_32 Tabelle existiert werden hiermit alle mit Finde Reloc_32 gefundenen Reloc_32 verglichen.

Durch Laden an verschiedenen Adressen (der Task mit .In_Go File) und anschliessendes Aufrufen dieser Funktion, werden alle rein zufällig in die Segmente zeigenden Zeiger eliminiert.

Dies ist alles etwas problembehaftet!
Diese Funktion kann nur bei Tasks angewählt werden !

1.90 Suche Vorwärts

Selbstredend

1.91 Suche Rückwärts

selbstredend

1.92 Line Menü

Zeilenadressanzeige
Zeige Data als
Zeilenlabelbreite
Data Optionen
Zeige Opcode in Hex
Zeige Library interpretation
Zeige Wort interpretation
Zeige Langwort interpretation
Zeige indirekt-indirekte Zeiger
Zeige Lokale Labelform

1.93 ZeilenAdressanzeige

Mit verschiedenen Modi

- Absolut als Speicheradresse, bei Objectfiles sinnlos
- Relativ zum Segmentanfang
- Keine Anzeige

Mit 'z' weiterschaltend umschaltbar.

1.94 Zeige Data als

Einmal erkannter Code wird immer als Code dargestellt (Überwiegend schwarz).

Datas oder unerkannter Code können hiermit in wählbarer Anzeigeart dargestellt werden (überwiegend gelb).

Wird ein Codeteil gelb dargestellt wird er jedoch immer als Data in ein Textfile ausgegeben !

Auch wenn die Anzeige dies nicht tut !

1.95 Zeilenlabelbreite

selbstredend

1.96 Data Optionen

Datas können verschieden Dargestellt werden !

- Anzeige von Strings
- Nullen sammeln

Im Gegensatz zu anderen Reassemblern kann bei In_Go die Data-Art nur Global eingestellt werden.

Wenn Sie eine bestimmte Adresse benötigen die durch die Anzeigemodi normal irgendwie nicht zugänglich ist, bringen Sie diese Adresse oberhalb des sichtbaren Bereiches.

durch Rückschritt (er findet in Wordbreite statt) kommt diese dann auch zur Anzeige ↔

Dies scheint mir verbesserungswürdig !

1.97 Zeige Opcode in Hex

Die ersten 10 Bytes werden als Hexzahl ausgegeben.

1.98 Zeige Library Interpretation

Dieser Schalter ermöglicht das Ein- und Ausschalten der Library- Interpretation. Zum richtigen Interpretieren eines Library-Aufrufes müssen die Variablen welche die Basen beinhalten richtig benannt werden !

Durch richtige Namensgebung der Librarybasen erkennt der Symbolische Processor von In_Go, die aufgerufene Library.

Richtige Namensgebung wird durch eine Benennung erreicht, welche den Namen im Reassemblerhilfsrequester entspricht.

Daher sollten Sie zuerst die Openlibrary Funktionen suchen.

In_Go hat im Suchrequester (F7) ein Menüeintrag der dies initiiert. Die Suche selbst

findet in der Hauptanzeige durch Amiga + 'f' oder Amiga + 'b'

Erst sollten sie aber 'Finde Typische Amiga Strings' aufrufen.

dieses wird Durch Amiga +Shift + '\$' erreicht.

Nun haben wir also einen OpenLibrary Aufruf gefunden:

Beispiel:

```

movea.l    $4.W,A6      ;Execbase nach A6
lea        dosname,A1   ;Name der zu öffnenden Library nach A1
moveq.l    #$25,D1      ;Die minimal geforderte Version
jsr        OpenLibrary(A6) ;Aufruf der Exec Funktion
                        ;wenn erfolgreich befindet sich nun die
                        ;LibraryBase in D0
Irgendwo darunter...   ;jetzt wird meist irgendwie getestet ob
...                  ;das öffnen erfolgreich war
...                  ;
move.l     D0,Lxxxxxxx  ;Hier beginnt die für uns interessante
                        ;Zeile: Die Base wird abgespeichert !

```

Nun klicken wir das Label Lxxxxxxx an; Der erscheinende Reassembler-

hilfsrequester ermöglicht uns eine Benennung dieses Labels. Im

Falle Dos geben wir den Namen '_DOSBase' ein.

Nach erfolgreicher Benennung erkennt der symbolische Processor nun

überall im Programm wenn Eine Dos Funktion Aufgerufen wird !

Manche Programme speichern auch die Execbase in einer Variablen ab

(dies wird von Commodore empfohlen) diese Abspeichern

erfolgt meist direkt am Anfang des Programms Beispiel:

```

        bsr INIT
        .....
INIT:   movea.l $4,A6      ;ExecBase nach A6
        move.l A6, Lxxxxxx ;ExecBase wird in Variable abgespeichert
        rts

```

Klicken Sie Lxxxxxxx An; Der erscheinende Reassemblerhilfsrequester ermöglicht nun die benennung dieses Labels in 'ExecBase'

Sollten Die Basen Indirekt(Ax) oder Pseudo(Ax) abgespeichert werden

muß dieser Fall erst im F3 Requester bearbeitet werden !

1.99 Zeige Wort Interpretation

Mit Hilfe des In_Go internen Symbolprocessors können Strukturoffsets als Symbol ausgegeben werden.

Hiermit kann diese Interpretation Aus- und Eingeschaltet werden.

1.100 Zeige Langwort Interpretation

In In_Go sind bereits gängige Langwortkonstanten abgespeichert. Hiermit kann die Interpretation Ein- und Ausgeschaltet werden.

1.101 Zeige indirekt-indirekte Zeiger

Zeigen Sprungadressen auf andere Sprungadressen kann durch Einschalten dieser Option das eigentliche Sprungziel ausgegeben werden. Dies sollte auch bei Langwort-Datas funktionieren.

1.102 Zeige lokale Labelform

Wenn lokale Label errechnet wurden, kann mit diesem Menüpunkt die Ausgabe Ein- und Ausgeschaltet werden. Bedenken Sie Bitte Das durch Eingabe eines einzigen (!) Labels alle lokalen Labeln erneut berechnet werden sollten.

1.103 MISC-Menü

Lerne Strings

Lerne Libs

Lerne Libs Prefs

Auto Rea Klick

Cleanup

1.104 Lerne Strings

Siehe In_Go internen Helptext

1.105 Lerne Libs

Siehe In_Go internen Helptext

1.106 Lerne libs Prefs

Siehe In_Go internen Helptext

1.107 Auto Rea Klick

Durch Anklicken eines Zeilenlabels wird die Reassemblierung gestartet.
Wenig empfehlenswert !

1.108 Cleanup

Diesen Eintrag sollten Sie eigentlich nie aufrufen müssen !

1.109 Requ Menü

Quelle in Hex und ASCII
Quelle als Text
Bildschirm, Speicher
Assembler Voreinstellung
Reassembler Voreinstellung
Rechner
Inhalte
Symbolischer Processor
Suchen
In den Speicher schreiben
Speicheranzeige Hex/ASCII
Optimierung aus der Quelle

1.110 Quelle in Hex und ASCII

Hier wird ein Requester ausgegeben der die geladene Quelle in Hex und ASCII ↔
ausgibt,
Durch '+' und '-' kann zwischen den verschiedenen Segmenten umgeschaltet werden.

1.111 Quelle als Text

Hier wird ein Requester ausgegeben der die geladene Quelle als Text interpretiert ↔
.
Durch '+' und '-' kann zwischen den verschiedenen Segmenten umgeschaltet werden.

1.112 Bildschirm, Speicher

In diesem Requester können Sie verschiedene Einstellungen bezüglich Speicher
und Anzeige editieren.

Die Scrollbox kann 2 Modi annehmen

- Grenzen: Hier werden Die maximalen und aktuellen Limits der In_Go internen

- Variablen ausgegeben. Eine Edition ist nicht möglich.
- Farben : In_Go ermöglicht die Ausgabe von verschiedenen Farben. Darum öffnet In_Go auch einen Screen in 8 Farben !
In vielen Fällen lassen sich hier die Farben der Zeilenausgabe ändern !
nach dem anklicken der entsprechenden Zeile.
- `Wähle Screen` der von In_Go geöffnete Screen kann bestimmt werden.
- `Wähle Font` der in den Scrollboxen verwendete Font kann eingestellt werden.
Dabei können nur lineare Fonts benutzt werden !
- `Internen Speicher definieren`
In_Go benutzt keine dynamischen Speicherbereiche.
Zum minimalen funktionieren werden jeweils 100000 Byte für Labeln und Referenzen benötigt.
Allgemein gilt Speicher für Labeln und Xrefs jeweils so groß wie die zu ladende Quelle.
Nach einer Neudefinition ist ein New fällig: Alles geht verloren !

1.113 Assembler Voreinstellung

eigentlich selbstredend
einzig: Wählen Sie Ihren Assembler. . .

1.114 Indirekt a4 - a6

Hier wird auf ein bereits beim Laden existierenden Speicherbereich zugegriffen, meist ist dies ein BSS Segment.
Viele Programme benutzen Register mit einem Zeiger auf Data oder Code
Diese Initierung erfolgt meist am Anfang von Segment 0
Ein Beispiel: `lea BaseA4,A4` ;hier wird die Adresse von Label BaseA4 in A4 abgelegt
Danach können alle Aufrufe die A4 enthalten auf jenen Speicherbereich zugreifen
auf den A4 zeigt:
Beispiel.: `movea.l \$xxxx(A4),A1` `addq.l #\$1,\$xxxx(A4)` `move.w xxxx(a4), yyy(a4)`
Sie sehen also, hier wird ein bestimmter Wert in A4 erwartet.
Diese \$xxxx und \$yyyy stehen für einen vorzeichenbehafteten Wert von Wortlänge.
Der Vorteil einer solchen Adressierung: es wird keine Korrektur erforderlich, da
alle Aufrufe relativ zu A4 erfolgen.
Da dieser `Wert in Wortlänge` vorzeichenbehaftet ist, wird oft noch ein zweiter Wert
dazuaddiert so dass sich Adressierungsmöglichkeiten im Bereich von 2^{16} also 65535 Byte
ergeben.
In_Go unterstützt Sie bei der Suche nach solchen Zeigern mit dem Schalter
`Suche indirekte Basen`. Sollten Sie sich im `Pseudo a0-a6` Modus befinden wird
dabei Automatisch in den `indirekt a4-a6` Modus zurückgeschaltet.
Meist werden dabei zuviele Einträge gefunden. Es bleibt Ihnen dann überlassen

durch Nachforschung in der Quelle die zuviel gefundenen Bereiche nach Anklicken mit dem Schalter 'Del' zu eliminieren. Meist ist nur ein einziger Zeiger auf A4 gültig.

Ein Eintrag für Indirekt 'a4-a6' enthält folgende Angaben

- Das Register für welches diese Definition gültigkeit hat.
- Zeiger auf das Label auf das A4 zeigt (oben BaseA4)
- Den Anfang des Codebereiches ab dem diese Definition gültig ist(oder 0 dann Global).
- Das Ende des Codebereiches ab dem diese Definition gültig ist(oder 0 dann Global).
- Ein eventueller Offset in Wortform der dazuaddiert wird (meist \$7FFE, wenn überhaupt)
- Ein Schalter der diese Wortadition einschaltet !
- Ein Schalter der Sprunglabelcreation zulässt (Dies ist äußerst selten).
- Ein Schalter der Datalabelcreation zulässt (Dies ist der Normalfall).
- Ein Schalter der diesen Eintrag insgesamt aktiviert

Sie können diese Einträge auch Ändern mit Schalter 'Edit'

Neue Einträge hinzufügen mit Schalter 'New'

Wenn Sie in der Hauptanzeige im Misc Menü Variablen belegt haben Können Sie mit 'Vare'

Diese in einen neuen Eitrag Übernehmen.

1.115 Pseudo a0-a6

'Pseudo a0-a6' Diese Einträge beinhalten, ähnlich wie der vorherige Modus, indirekte

Aufrufe, Diese indirekte Aufrufe zeigen jedoch nicht auf einen Teil der Quelle !

Typisch für Pseudo-Bereiche sind Link-Unlink Module von 'C' - Compile , aber auch reentrante Programme die Ihren Speicher erst während des Laufens allocieren.

In_Go unterstützt Sie bei der Suche nach solchen Bereichen mit dem Schalter 'Suche Link Module'.Sollten Sie sich im 'Indirekt a4-a6' Modus befinden, wird dabei Automatisch in den 'Pseudo a0-a6' Modus geschaltet.

Ein Eintrag für Pseudo 'a0-a6' enthält folgende Angaben

- Das Register für welches diese Definition gültigkeit hat.
- Den Anfang des Codebereiches ab dem diese Definition gültig ist(oder 0 dann Global).
- Das Ende des Codebereiches ab dem diese Definition gültig ist(oder 0 dann Global).
- Ein Schalter der diesen Eintrag insgesamt aktiviert.

Sprunglabel werden innerhalb dieses Moduses nicht unterstützt.

Sie können diese Einträge auch Ändern mit Schalter 'Edit' oder Anklicken

Neue Einträge hinzufügen mit Schalter 'New'

Wenn Sie in der Hauptanzeige im Misc Menü Variablen belegt haben, Können Sie mit 'Vare'

diese in einen neuen Eintrag übernehmen.

1.116 Reassembler Voreinstellung

Hier können verschiedene Einstellungen vorgenommen werden, die das Reassemblieren betreffen.
Für die Scrollbox existieren dabei zwei Modi:

- Indirekt a4-a6 mit Unteroption 'Suche indirekte Basen'
- Pseudo a0-a6 mit Unteroption 'Suche Link Module'

Durch Anklicken der Einträge oder 'New' können neue Bereiche in einem dann erscheinenden Requester eingegeben werden.

Allgemein Wirksam sind folgende Optionen:

- Tabellenautomatik
Mit diesem Einsteller kann das automatische Tabelleneinlesen ↔ beeinflusst werden.

Stufe 1: Sicheres Einlesen verschiedener Tabellen
Stufe 2: Einige Etwas unsichere Tabellen werden nun auch automatisch ↔ eingelesen.

Das automatische Tabelleneinlesen erfolgt beim Reassemblieren ('Space' ↔ Taste !)
- Keine Kreation wenn: Diese Schalter Unterdrücken die Labelcreation beim ↔ Reassemblieren
Dadurch werden Fehler vermieden, die durch zufällig in die Segmente ↔ zeigenden Datas verursacht werden. Beispiel move #\$100000,D0 : Wenn nun Segment 0 von ↔ \$99000 - \$101000 gelegen ist würde an Adresse 100000 ein Label generiert werden. Dabei sind folgende Schaltmöglichkeiten vorhanden:
 - Pea.l
 - Effektive Adresse
 - Unmittelbar.l Instruktionen
 - Langwort allgemein (Schließt die letzten 3 Optionen Mit ein !)
- Reassembler restart wenn: Diese Schalter ermöglichen einen Reassembler ↔ Restart auch an nicht ausdrücklich als Sprungadresse gekennzeichneten Adressen, die nur ↔ indirekt aufgerufen werden. Dabei sind folgende Schaltmöglichkeiten vorhanden:
 - move.l A6,-(A7)
 - movem allgemein (nicht jedoch: PC)
 - Link allgemein
- Owninstruktion Macro.w Ermöglicht die Generierung von einer eigenen ↔ Instruktion
in Wortbreite: dabei kann Wert und Namen Editiert werden. Mit einem

Schalter kann diese Option eingeschaltet werden.
Normalerweise werden Sie diese Option nicht benötigen .

1.117 Rechner

Rechnen auf Processorebene
ca 40. verschiedene Funktionen.
Errechnete Werte können weiterverarbeitet werden:
- eine Taste belegen
- als Jumplabel (Reassemblerrestartadresse) anlegen.
- als Data Label anlegen.

1.118 Inhalte

Hier können verschiedene Inhalte gesichtet werden.

- | | |
|-----------------------|--|
| - Selectionsschlüssel | Nur für mich von Bedeutung. |
| - Codebereiche | Erkannte Bereiche die Code enthalten. |
| - Labels,Flags | Erkannte Label und deren Interpretation. |
| - Labelnamen | Eigene Namen. |
| - Status Script | Fileeigenschaften, Fehler usw. |
| - Strukturen | Bekannte Strukturen, Die aktuelle Zeile.
erlaubt eine Einsicht von Unterstrukturen. |
| - Langwortkonstanten | Langwortkonstanten und deren Interpretation. |
| - Reloc_32 Referenzen | Alle eingeladenen (D)Reloc_32 Referenzen. |
| - Andere Referenzen | Alle während des Reassemblierens gefundenen Refs. |
| - Pseudo Bereiche | Alle Pseudobereiche. |
| - Kommentare | Alle Kommentare. |

1.119 Symbolischer Processor

Hier können die Inhalte des symbolischen Processors
eingesehen (nicht geändert !) werden.

1.120 Suchen

Hier wird eine Suche initiiert.
dabei kann folgendes gesucht werden:

- 1 Byte
- 1 Wort
- 1 Langwort
- String
- Label

Dabei können auch einige Erweiterungen angefordert werden

- `relativ.l (PC) ` für relative Aufrufe ab 68020
- `relativ.w (PC) ` für relative Aufrufe
- `relativ.b (PC) ` für relative Aufrufe
- `indirekt(Ax) ` für indirekte Aufrufe wie im F3 Requester eingegeben.
- `global ` ` die Suche wird auf alle folgenden Segmente ausgedehnt.

Die eigentliche Suche wird nur in der Hauptanzeige durchgeführt !

Amiga + f sucht vorwärts.

Amiga + b sucht rückwärts.

Einige wichtige Suchwerte können per Menü angefordert werden.

- OpenLibrary
- AllocMem
- lea labelx

1.121 In den Speicher schreiben

Hier wird die aktuelle Adresse zum überschreiben angeboten.
dabei kann folgendes geschrieben werden:

- 1 Byte
- 1 Wort
- 1 Langwort
- String mit max. 40 Zeichen
- CString mit max. 40 Zeichen (Hier wird ein 0 Byte angefügt)

1.122 Speicheranzeige in Hex und ASCII

Mit dieser Anzeige kann eine frei wählbarer Speicherbereich angesehen werden.

1.123 Optimierungen aus der Quelle

Aufruf durch F10 oder Menü.

Im Hauptmenü kann diese Option durch `o` geschaltet werden

Der Requester bietet folgende Schlaltmöglichkeiten:

- ext.w Dx gefolgt von ext.l Dx --> extb.l Dx
(spart 2 Byte bei 68020 Processor)
- jmp/jsr Langwort --> bra.l/bsr.l
(spart bis zu 8 Byte bei 68020 Processor)
- <asl.l #1,dx> gefolgt von <move.l \$cc(Ay,Dx),Dz>
--> <move.l \$cc(Ay,Dx*2),Dz> das <asl.l #1,dx>
wird dann wegelassen.
Diese Option wird nicht nur bei obiger Instruktion
wirksam, diese wurde nur als Beispiel gewählt !

Diese Option kann zu Fehlern führen da das Ergebnis von dx verworfen wird !

- <add.l dx,dx> gefolgt von <move.l \$cc(Ay,Dx),Dz>
--> <move.l \$cc(Ay,Dx*2)> das <add.l Dx,Dx> wird weglassen.
Diese Option kann zu Fehlern führen da das Ergebnis von dx verworfen wird !
Diese Option wird nicht nur bei <move.l> wirksam,
dieses wurde nur als Beispiel gewählt !
- <move.l cc(Ax),Ay> gefolgt von <move.x (Ay),Dz>
--> <move [(cc,Ax)],Dz>
Diese Option kann zu Fehlern führen da das Ergebnis von Ay verworfen wird. !
Diese Option wird nicht nur bei <move.l> wirksam,
dieses wurde nur als Beispiel gewählt !

Optimierung mit Hilfe von Makros

Makros erhöhen die Übersichtlichkeit in der Anzeige !

- Makrooptimierung Aus- und Einschalten
Mit diesem Schalter kann bestimmt werden ob In_Go überhaupt Makros erzeugen soll !
- Multi Stack Makros
Stackbefehle werden zusammengefasst
- Instruktion + Branch Befehle
Die Instruktion und der Branchbefehl werden zusammengefasst
- Gfa-Makros
Einige spezielle Gfa-Makros werden ausgegeben,sie sind für andere Quellen uninteressant.
- Optimierung Ein/Aus
Diese Funktion ist mit der Funktion Taste <o> in der Hauptanzeige identisch.

1.124 Prozessor Menü

Zur Zeit unterstützt In_Go die folgenden Processoren:

- 68000
- 68010
- 68020

Als Koprozessoren werden unterstützt:

- MMU
- Math

Wenn die entsprechende Option eigestellt ist erkennt In_Go die entsprechenden Erweiterungen; andernfalls wird ein Fehler angenommen und an anderer Stelle wird die Reassemblierung fortgeführt.

1.125 Hunk/Segment Menü

Es wird je nach Auswahl zum entsprechenden Segment/ Hunk verzweigt.

Ein bestimmtes Segment kann auch mit Hilfe der Tastatur Marken 0 – 9 angesprungen werden, wenn diese nicht belegt sind.

Dies funktioniert jedoch nur bei Hunk 0-9