03ce4198-0

COLLABORATORS

	TITLE : 03ce4198-0	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 27, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

03ce4198-0

1.1 Index

```
                PPC680x0 Small Documentation For Promo Edition
-----------------------------------------------
   Written in March-May 1999 by Coyote Flux



        Prologue

        NEW~Important~notes~and~Info

        Missing~Functions

        What~is~PPC680x0?
                              How does it work?
                -
        External
                                -
        Internal
                              Usage
                -
        General
                                -
        Programming~Environments
                                -
        Instruction~Set
                                -
        Optimizing

        Quick~Guide

        Notes~&~Thank~You's

        Ordering~PPC680x0

        Copyright?

        Authors
```

History

Our~Great~Amiga

PC~Must~Die!

Amiga~Format~Attitude
Good luck!

1.2 Prologue

Prologue

Welcome to
PPC680x0
! What you have here is the first source-code convertor for the Amiga-PPC platform. This version is for promotional use ONLY and should not be spread as a Public Domain utility. This promo version MIGHT have some little bugs in the output codes although they weren't spotted yet. It should have NO fatal bugs, though.

Make sure you have read the
missing~section
as this also contains
information on what is new in this version of PPC680x0. The new features are not discussed in the rest of the guidefile...yet.

Let's hope this utility will make PPC programming easier for all of you!

Just a little note: PPC680x0 is a big library function that is called everytime you push a button on the PPC680x0 interface. The library version will be released together with the final version, allowing realtime conversion etc.

Also remember that we are always happy to see bug reports!!!

1.3 Missing Functions

WHAT'S MISSING IN THIS VERSION

This promo-version has nearly all 68000 instructions enabled except for:

* link/unlk
* some extremely low-level instructions which include trapping and exception instructions.

But there's a lot of other things missing, such as:

* Intelligent label detection:
  At this moment PPC680x0 does not detect brackets in labels

```
    which results in errors. The final version will detect labels
    as well as their perfect sizes for optimizations
 * Support for equr, fequr equrl and reg directives
 * 68020+ (EXTRA) instructions support
   (which means the promo does support 32-bits mulu etc. but
   no bitfield commands etc.. 64->32 bits divisions are also left
   out)
 * 68881+ instructions support
 * low-level CARRY emulation (now works for about 70%)
 * low-level SR/CCR emulation
 * low-level register emulation
 * extra WarpUp/PowerUp goodies
 * rest emulation for divisions/multiplications
 * next 50% of SUPER architecture instructions
 * More error messages for the interface
 * Amiga-guide instruction file with a LOT of more information in it
 * IFF-pictures with schedules and conversion methods
 * full-color interface on 8 bitplane+ Amiga's.
 * PPC680x0.library for low-level usage

AND MOST IMPORTANT:

 * modular interface, which means you can always get free patches,
   optimizations, additional instructions etc. from Aminet.
 * all functions tested 100%
 * output-organ extensions: There's a possibility that there will
   be new output languages for PPC680x0, which means you can then
   convert your codes to PowerPC, ColdFire, AmigaNG etc..

REMOVED IN PROMO 1.7 (FROM COMMERCIAL RELEASE)

 - FPU
 - some instructions for bitfields etc.
 - SUPER2 architecture
 - rest emulation
 - good guidefile
 - extensive internal interfacing

ADDED IN COMMERCIAL RELEASE 1.0 (7th July 1999)

 - A LOT EXTRA!!!
 - rN suffix
 - good guidefile
 - most bitfield commands
 - optimizations (too much to list!)
 - bug-fixes (too much to list!)
 - dctv.library conversion routines succesfully converted

ADDED IN COMMERCIAL PRE-RELEASE EDITION 0.82  (5th July 1999)

 - link/unlk set to enabled
 - return terminators added to prevent spaces in between code
 - rest emulation added
 - ext carry emulation fixed
 - many tiny fixes for parts that created trash or forgotten tabs
 - Bit Fields are now in the work-in-progress state, they don't
  give an error, but don't give an output either. Expect them
```

```
  tomorrow...!

ADDED IN COMMERCIAL PRE-RELEASE EDITION 0.8

 - link, unlk
 - 68060 Floating Point Unit working and enabled
 - bug-fixes for all conditional and branching instructions
 - optimizations in loading/storing
 - 40% EXTRA of Super Architecture
 - new guidefile
 - better immediate handling
 - modular interface native functions
 - X=D button
 - header questions enable/disable
 - carry emulation now better
 - 'not' fixed
 - better low-level register handling

ADDED IN PROMO 1.5:

 - Bug Fixes and optimizations:
  - exg defaulted to .W FIXED
  - move.l   d0,(a1) caused problems
  - rol      #imm,<D> now faster and FIXED
  - mulu/muls/divu/divs FIXED (big bug!)
  - mathematical model caused trash or unnecessary instructions
  - equr/equrl/reg/fequr now set to promo status. Disabled! The final
    version will have full support for them. They were already
    disabled but they didn't give an ERROR message as they do now!
  - (sp) usage caused trouble sometimes
  - moveq fixed (no more .w usage)
  - asr.l    #imm,<D>
  - shifting is now much faster and works(!) much better
  - moveq    #0,<D>  now set to clear status
  - simple range checking enabled so that immediate values now
   default to quick format if low enough. This is not yet as
   complicated as in the final version though! Simple means that
   it detects binary/octal/hex/decimal NUMBERS ONLY. No sums, that
   is....
  - moveq now has detection for zeroes
  - PPC immediate value support for shifting/rotating added
  - more??? Probably! (rlwimi's, exts's etc. removed or changed to a
    working version!)

 - History link added to guide
 - New Important Notes and Information added to guide

ADDED IN PROMO 1.4:

 - lea/pea/jsr/jmp <label>.w produced trash
 - interleaving went active after a few lines when it was disabled
 - new extremely useful PPCDIRECT command added:

 The smove command is an extremely handy substitute for all normal
 load/store commands of the PPC. It works as follows:

  smove.size     (<label>,reg1),reg2
```

```
  smove.size     (reg1,reg2),reg3
  smove.size     reg1,(<label>,reg2)
  smove.size     reg1,(reg2,reg3)

 A plus behind the brackets means update enabled. Example:

  smove.l        (label,d3)+,a4

 means <ea> is label+d3. This is loaded in a4. d3 is updated and
 keeps the <ea> used for this mode which is label+d3.

 This command is directly translated to one single PPC instruction

ADDED IN PROMO 1.3:

 - There were terrible bugs in 'Full' addressing modes. Changing a
   slight part of the source code caused everything to change in a
   direction with bad side-effects. This is now fixed.
 - Double extension bug fixed
 - Many little bugs and logical parts fixed
 - Missing dot for subroutines added: This caused a bug that
   destroyed subroutine emulation
 - 'head' and 'tail' command added. This makes the promised
   'autoheader' questions obsolete as the 'head' instruction makes
   an PPC680x0 Start Header and the 'tail' instructions an PPC680x0
   End Header. Can be used only once!!! (Although using it twice
   won't give any errors!!!)
 - PPC680x0 source code succesfully converted to PPC using this
   software!
 - Changes in guide-file: Missing section, Prologue, Copyright and
   Ordering.

ADDED IN PROMO 1.2:

 - Macro support for non-68k-command macro's
 - New address-loading mechanism (USE Full/QB button for old one
   where QB falls to the old mode. Experience needed!)
 - Help button calls guidefile when available in directory
   where PPC680x0: is assigned to
 - Power StormASM Predefined Symbols added
 - =/== added
 - Block comments added
 - Floating Joint natives added. No effect though!!!
```

1.4 WhatIsPPC680x0

```
                WHAT IS PPC680x0???

(If you just want to start, refer to the
                quick~guide
                )

PPC680x0 is a source-level converting utility (which means you can convert
your old 680x0 source codes straight to PowerPC source-codes) and a highly
powerful 680x0-based programming  language.
```

There's a lot of programmers who don't seem to be happy with the programming architecture of the Motorola/IBM PowerPC family. Although the PPC is an extremely powerful piece of silicon, it suffers a bit from its not-exactly-user-friendly design.

Although many people (and even universities) seem to deny it, machine language is the only way to make programs run at maximum speed. Just take a look at the demo-scene, where coders push the most advanced effects through a 'theoretically slow' 68k CPU.

PPC680x0 is a program that should help out coders in a easy but (when used in the right way) extremely powerful way. You can actually treat the PowerPC CPU as a very big 68060.

Try to imagine yourself an 68060 with over 27 general purpose registers (which means it doesn't matter if you use one as an address or data register), many extra instructions, more addressing modes, Quad-word operations, etc. etc... It is now possible, thanks to PPC680x0...

PPC680x0 outputs 100% Storm PowerAsm compatible code.

1.5 External Workings

HOW DOES IT WORK???

(GENERAL)

Before we start with the inside details, lets first tell you how the interface and the general environment works:

Well, very simple, in fact... You write your source-code in any editor (or in an 680x0 assembler if you are not using any extensions), which means you have the following, for example:

```
;Source code for predictable change volume ;By Ernie & Bert in 1999

 x
        movem.l d0-d3/d14/d16,table
        move.l  a0,d0
        move.l  a1,d1
        clr.l   d3
        move.w  (d0)+,d2
        asl.l   #16,d2
        divs.l  d14,d2
        asr.l   #16,d2
        addx.l  d3,d2
        move.w  d2,(d1)+
        add.l   d2,d14
        add.l   d14,d16
        bmi.s   quit
        clr.l   notneeded
 quit   movem.l table,d0-d3/d14/d16
        lastrts
```

```
 table  ds.l    18
 notneeded
        dc.l    0
```

Now you have a source code which uses special PPC680x0 extensions. Of course it could also be any source code without those! You want to convert this source code to a PPC source code. So what do you do? You load up PPC680x0, select the things you prefer and the optimizations. Select this as a source file, select a destination file and finally press the
convert
button. You
get some questions, and after giving the answers to them you can load your destination file in a PowerPC assembler and run it (or look at it in any editor!)... Of course you should make sure that you only call PPC based kernel routines such as WarpUp/PowerUp. You can use 'call' for this, which is automatically converted to the right instruction...

It is as simple as that. Just try it out! If there's something you don't seem to understand you can use the
quick~guide
or simply read the entire
manual...

1.6 Internal Workings

HOW DOES IT WORK???

(INSIDE DETAILS)

The inside activities of PPC680x0 are much more difficult to understand, though! Lets try to give you an idea of how it works:

PPC680x0 works with a mathematical model of an 68k CPU. The 68k is split up in parts, where some of them built a formula on a complex plane (which could be called the graph of a 'complex' formula, based on the square-roots of -1).

For example, we could take the pixels on a monitor as some points of a complex plane. Every region of the monitor corresponds with a region of the 68k CPU (which can be any different mode). We could say that in the entire complex plane, all possible addressing modes are placed changing slightly when moving from point to point. (For example: When moving a relatively small end from the point corresponding with the addressing-mode d0,d4 and in the right direction, you get d0,d5). This interpretation method is possible because the 68k CPU is built in a very flexible way.

Perhaps you are wondering why we took so much time in creating a mathematical model of the chip. Well, its quite easy to guess why if you take a look in terms of performance. The earliest versions of PPC680x0 were based on simple straight conversions. As a result, in couldn't predict every combination possible, meaning that so called trash was generated. For example, the following mode:

```
        move.l  (worst.b,d2.l),d1
```

could be converted to something like

```
        addi    r28,worst
        add     r28,r28,r5      (where r3 corresponds with d0)
        lwz     r4,0(r28)
```

Now it seems logical that a program detects everything, but with linear programming you cannot make sure this is done right unless you put a special routine in your code that helps. PPC680x0's mathematical model takes samples of the code, analyses the samples and finally converts it to the PPC model. The output code will be

```
        lwz     r4,worst(r5)
```

which is the most efficient way. Besides, linear code is not as flexible as analysing code. PPC680x0 allows the following:

```
move.l ([worst.q,d0*16,d4.l*2],help.b,table.w,d14.b*32,a2),(a0)+
```

As a matter of fact, sampling goes in an infinite way, which means indirect addressing modes can be seen as:

```
        a0([b1,b2,b3,b4,b5,...],a1,a2,a3,a4,a5,...)
```

which means

```
        load    b1+b1+b2+b3+... in IOREGISTER
        load    IOREGISTER+a0+a1+a2+a3+a4+... in IOREGISTER
```

(and where a/b are registers.size*multi or label.size)

Every part of the 68k CPU is treated this way to optimize output-codes.

Explanation of some terms concerning PPC680x0:

```
 INTEGRATING     - sampling of the source-code
 TRANSFORMING    - changing the samples to a standard form
 DIFFERENTIATING - converting the transformed samples to the
                   output source-code
 LAYERS          - the layers in which the 68k is split up
 BLOCKS          - a block of 100 lines of code for analysing
 LINE            - 1/100th of a block, one line of the input
                   source-code
 L. AREA         - Label area (starting at first char of line)
 I. AREA         - Instruction area (starting after L. AREA)
 BWL AREA        - area that holds size (.b/.w/.l/.q/.s/.d/.x)
 T. AREA         - Type area (starting after BWL area)
 C. AREA         - Comment area (starts with ';', '*', etc.)
```

1.7 UsageIndex

PPC680x0 - THE USAGE

Pfffffff! Sometimes I’m wondering why I started making this program! There’s infinite combinations and it’s impossible to test them all! Happily, this is just a promo-version with a lot of options disabled! The final version has a modular interface, which means you can always get free patches from the Aminet, containing optimizations, bug-fixes (if any) and extra instructions! For more info on what’s missing in this version, please refer to the

Missing

Options
!

If you ’just want to convert a file’ please refer to the

quick~guide
!

Requirements

General~Usage

Buttons

Button~1~~-~Two~arrows

Button~2~~-~r1

Button~3~~-~a7=sp

Button~4~~-~Wup

Button~5~~-~Gpr

Button~6~~-~?ERR

Button~7~~-~Proc~Dir

Button~8~~-~32

Button~9~~-~00~0

Button~10~-~Rest~Off

Button~11~-~Stat~Off

Button~12~-~Frc

Button~13~-~Asl~In

Button~14~-~Asl~Out

Button~15~-~Big~’M’

Display~Information

Error~Messages

1.8 Usage

Requirements:

- An Amiga
- At least 430kb of Fast memory
- The Fonts: drawer installed
- Kickstart 2.04+ (Including libraries etc.)
- MagicWB

General Usage:

Perhaps we should start with the
programming~environments
, but as there's
quite a lot of users who want to use this as a conversion utility, we have placed a special
Programming~Environment
section in this guide.

So, you can install the font by simply double-clicking on the 'Install Font' icon or by copying the fonts into your Fonts: drawer. After you have copied the fonts, you can either double click on the PPC680x0 icon to start the program or drag the icon to a place on your harddisk where you would like to have it.

After you have double-clicked on the icon, you get a window with some buttons, a picture and a scrolltext

Pressing any button will cause the picture to change into an
information

display
.

Buttons

There's 12 buttons on top-left of the window and a few more outside of that cluster. The buttons mean the following:

Button 1 - The two arrows

The two arrows select if you want interleaving enabled. Interleaving means that the original 68k source code is inserted throughout the output-file. With interleaving disabled (arrows point in opposite direction, you get something like:

```
x
        move.l  (d13),d1
        add.l   d1,d2
```

With interleaving enabled (default), the code is converted to:

```
;x
x
;       move.l  (d13),d1
        lwz     r4,0(r10)
;       add.l   d1,d2
        addco.  r5,r5,r4
```

Button 2 - r1

With this button you can select the stackpointer. Pressing this button will make it change to r27, which equals a7. You should use this button if you don't want to use the userstack which is given to you by PowerUp/WarpUp. The user-stackpointer for this userstack is normally r1.

When you select r27 as a stack pointer, PPC680x0 automatically adds a space in the end header of the output file. This space is the size of the stackpointer in longwords as selected in the string-gadget named 'stacksize a7'.

Button 3 - a7=sp

This button is used in combination with button 2. 'a7=sp' means that using 'sp' in your code is treated the same as using 'a7'.

Why should you use this? Well, it's all quite logical: Because of the advanced register model of PPC680x0, it could be possible that you want to use a7 as a general purpose register instead of a stackpointer.

For example, using:

```
        move.l  (a7),d0
```

can have different effects. The command 'move.l (a7),d0' means that you want to copy the indirect value of a7 into d0 with a longword size. In original 680x0 code, a7 is always used as the stackpointer, meaning that the instruction is entirely the same as 'move.l (sp),d0'. Disabling the 'a7=sp' button means that you can use a7 and sp as two different registers. Of course this button has no effect if button 2 is set to 'r27', which means that 'sp' is always decoded to r27. When converting original 680x0 code, this button should always be set to 'a7=sp'.

Button 4 - Wup

Select goodies for WarpUp (Wup) or PowerUp (Pup). Not implemented in promotional version.

Button 5 - Gpr

Display the General Purpose Registers or Floating Point Registers in the information window. There's a part in the information window called '680x0 register usage'. This button switches between displaying the amount of GPR's used and the amount of FPR's used. Of course, the amount of FPR's is always 0 in the promo version.

Button 6 - ?ERR

When enabled (not masked out) all totally unknown errors (which are usually trash) are copied as ERRORUNKNOWN. When disabled (masked out) these errors are ignored.

Button 7 - Proc Dir

Process Directory. Pushing this button will start the process directory procedure, allowing the user to convert an entire directory. Please note that the input dir should not have binary files as they will be treated as source-codes, resulting in null-files or a trash-error.

So, how does it work? When pressing the button, you get an asl-requester where you should select the input directory. After selecting the input directory, you get a requester asking for the output directory (which should of course not be the same as the input directory). When you have selected the right directories, you are asked the following:

'Should I Add Headers To All Files?'

Selecting 'Yes' means that every file in the directory gets a stack-pointer (when enabled) and a startup-code from PPC680x0. This should only be used if every source-code is an individual program, meaning that you can assemble every file individually.

Normally, you should select 'No', meaning that no file gets a header. You should convert the file that needs a header afterwards, using the normal 'convert' button.

Button 8 - 32

Selects if your output code should be PowerPC 32-bits implementation or 64-bits implementation compatible. If you don't know in what mode your PPC is running you should check the WarpUp/PowerUp manual.

Button 9 - 00 0

Carry Emulation On/Off. When set to '00 1', carry emulation is enabled. Carrying goes in quite a different way on the PowerPC, as there's no carry in the Status Register anymore but in a special register called the XER. Selecting carry emulation means that almost all instructions change the carry in the way they should change it on an 680x0 CPU. This option is implemented for about 70% in the promo edition.

Button 10 - Rest Off

Rest emulation for divisions. Not implemented in promo version

Button 11 - Stat Off

Status Register/Condition Code/Lowlevel Emulation. Implemented only for Condition Codes in the promo version. Beware: CCR emulation is not complete in the promo version and does not suport any 'cmd from/to CCR' modes.

Button 12 - Frc

Now this is quite a useful button! When selected (default), all address accesses are forced to 32-bits when working in 64-bits implementations. This

is useful if your PPC does not support more than a 32-bits address-bus. Effective Address calculation then automatically zeroes the upper 32-bits of the address. This button has no effect if Button 8 is set to 32-bits implementations.

Button 13 - Asl In

This is the disk-button on left of the input selection string. Pressing this button pops up an ASL requester to select the source file.

Button 14 - Asl Out

This is the disk-button on left of the output selection string. Pressing this button pops up an ASL requester to select the destination file.

Button 15 - Big 'M'

Press this button to start converting the files selected using buttons 13 and 14. When pressing this button you get a requester prompting:

'Should I Make A Start Header?'

Selecting 'Yes' means that there will be a header on top of the file allowing the PPC assembler to start this file. The return address is taken from the link register and placed in the count register. You should only select 'Yes' if this is the file that will be assembled and runned in the assembler. You should select 'No' for everything else such as includes etc.

After selecting this, PPC680x0 will start converting. When it's ready, you get a requester:

'Should I Make An End Header?'

You should select 'Yes' if you want PPC680x0's internal data to be placed on bottom of this output file. It is recommended to put this on bottom of the startup/close file or in the data includes.

When you have selected this, your file is converted and can be loaded with any editor...!

DISPLAY INFORMATION

The display information means the following:

'errors' counts the amount of errors occured

'predec' counts the amount of pre-decrements used
'pstinc' counts the amount of post-increments used
'integr' counts the amount of integrate-modes used
'immval' counts the amount of immediate values used
'direct' counts the amount of direct registers used

'normal' counts the amount of normal instructions used
'single' counts the amount of single instructions used
'genera' counts the amount of general instructions used
'specia' counts the amount of special instructions used
'psuper' counts the amount of super instructions used

```
 'implie' counts the amount of implied instructions used
 'branch' counts the amount of branching instructions used
 'extend' counts the amount of extend instructions used

 'lineno' displays the line being processed. (every 128)

  The 680x0 register usage counts all the registers that are used during the
code. 0 equals d0 (=r3) and 17-24 equals a0-a7  (=r20-r27).
                Button~5
                 selects
if this part displays the GPR's or FPR's used in the code.


ERROR MESSAGES

At this moment, PPC680x0 has 25 error messages. Here's a quick guide through
them:

GENERAL ERRORS

 * ERROROVERFLOW

Although theoretically impossible, this error detects a buffer overflow.
PPC680x0 allocates 256kb for 100 lines of decoding which is impossible to
achieve, except if all 100 lines use 2*32 indices etc.. This error is always
on bottom of the file.

 * ERRORUNKNOWN

This is an error that is usually caused by trash in the instruction area.
For example:

 mofe.l  (a0),d0

causes this error to pop up. It might also be any other undetected error...

OTHER ERRORS (ALL USER ERRORS!)

 * ERROR1

This means that there a problem in the BWL area of the instruction. This can
be caused by things as 'move.k' or 'move.'. Should always be a user error.

 * ERROR2

Instruction not really implied. You have placed a non-implied instruction in
an implied way. For example:

 divu.l

causes this error to come, as there should be some input for the command.
'divu' is not an implied instruction.

 * ERROR3

Instruction not supported. The instruction you have used is not supported by
the promo version  of PPC680x0.
```

 * ERROR4

Unknown. Probably trash or forgotten comma.

 * ERROR5

Bracket expected after label. PPC680x0 is scanning for a bracket but cannot find one...

 * ERROR6, ERROR7, ERROR8

Error in element A, B or C respectively. This is an error caused by bugs in the elements of the integrator unit input. (See
InsideDetails
) Elements are placed in the following way:

  A([B1,B2,B3,B4,...],C1,C2,C3,C4,C5,...)

 * ERROR9

64-bits in 32-bits mode. This error pops up when you are using 64-bits instructions without 64-bits implementations (
Button~8
) enabled.

 * ERROR10

Wrong lea mode. The lea (load effective address) mode you are using is wrong.

 * ERROR11

Destination immediate. Come on...! A destination cannot be immediate!!! (i.e. move.l d0,#13)

 * ERROR12

Super Group Unknown. The super instruction you are using is not known by PPC680x0.

 * ERROR13

Super Command Bad. The super instruction you are using has register-only operation for the selected field.

 * ERROR14

Super Command Bad. The super instruction you are using has label-only operation for the selected field.

 * ERROR15

Super Command Bad. Both fields (label and register) are bad.

 * ERROR16

```
Bad Super Command Nth element. The super command you are using has a bug in
one of the higher elements.

 * ERROR17

Dh:Dl expected. You have placed a ':' behind a register but no second
register behind the ':'.

 * ERROR18

Movem register bug. Something is wrong with your 'movem' instruction...

 * ERROR19

(movem) '-' , '/' , ',' expected.

 * ERROR20

Movem needs directfield and integration field.

 * ERROR21

Floating point .s/.d/.x detected without FPU enabled

 * ERROR22

Floating point .p not supported

 * ERROR23

Floating point unit not found (PROMO-ONLY)
```

1.9 Programming Environments

```
                PROGRAMMING ENVIRONMENTS & GUIDELINES

This part differs a lot from the guidefile that is given with the original
version of PPC680x0. Many of the tips, tricks and details of PPC680x0-coding
are left out and this section is more an introduction and general
description of PPC680x0 instead of a big instruction.
                Optimizing
                 and

                Instruction~Set
                 are also incomplete.

Introduction

Because PPC680x0 uses a mathematical model of an 680x0 CPU, it is able to
detect almost every addressing mode for every command. Although there's a
few exceptions to this rule, it can be said that 'single' and 'double'
instructions always support all (addressing & operation) modes.

The commands of PPC680x0 are split in the following groups:
```

```
 normal
 single
 general
 special
 super
 implied
 branch
 extend

 Here's a small description of these modes:

 normal  - Commands that support the standard  'cmd    S,D'
           instruction form, where S and D stand for Source and
           Destination
 single  - Commands that support the standard  'cmd    D'
           instruction form, where D stands for Destination
 general - Commands that are general assembler instructions.
           These instructions are normally assembler directives
 special - Commands that have special treatment (bitfields etc.)
 super   - PPC680x0's own direct PPC instruction set
 implied - Commands that are implied (like rts, nop etc.)
 branch  - All Flow Control instructions (bcc, dbcc, jsr etc.)
 extend  - Extension instructions (ext, extb)

Normal and single modes use the pre-integrator unit to extract the
addressing modes. The main 'parts' of the integrator are  known as the
following:

 predec  - Pre-decrement mode. Internally known as i(+x)
 pstinc  - Post-increment mode. Internally known as -i(+x)
 integr  - Integrator mode. Internally known as 0+(x)
 immval  - Immediate value. Internally known as (-)2i(+x)
 direct  - Direct value. Internally known as absolute 0

The x-part of the formula is not important right now, but it might be on
later revisions if they will support displays of the
                complex~plane
                 etc.

Pre-decrementing is known as -(reg) and post-incrementing is known as
(reg)+. Integrating is known as every mode that uses labels and is not
immediate. So actually any kind of non-single-register memory access is
decoded in this mode. Immediate values quite speak for theirself, except
that PPC680x0 does not support immediate values that keep termination codes.
The following is NOT supported:

        move.l  #'AB,C',d0

So beware! Finally, direct modes are known as all modes that use direct
register access.

So..... Now we are not going to tell you how to program a 680x0 CPU! If you
have zero experience with the 68k, you should get some document files from
Aminet or check Motorola as they might have some books for you!

PPC680x0 PROGRAMMING
```

What we ARE going to tell you are the differences between 680x0 and PPC680x0 machine language. We are of course also giving you the super (directPPC)

instruction~set
as far as supported in the promo edition! Don’t forget that this is a short guide!!!

Differences

* PPC680x0 supports all addressing modes for normal & single instructions. (Few exceptions to the rule!)
* PPC680x0 has an additional instruction set
* PPC680x0 has true quadword support (you can use .Q now)
* PPC680x0 supports unlimited indices
* PPC680x0 supports 28 registers maximally
* PPC680x0 supports scaling modes from *1 to *32
* PPC680x0 can enable/disable native carry/sr/ccr support
* PPC680x0 can use a7 as GPR
* PPC680x0 is generally faster in .L/.Q access than .B/.W access
* PPC680x0 supports all non-macro Storm PowerASM commands (except one that has been renamed!!!)
* PPC680x0 supports quick values from 0-65535
* PPC680x0 uses no extend bit. (all done via carry)

You can actually program PPC680x0 in the same way as a normal 68k CPU except that optimizing rules have changed.

(68000) instructions
not~supported
in the promo-version:

all fpu and control instructions
divisions with Dh:Dl (68020+)
all trapping instructions
bkpt
cas
chk
link
movec
movep
reset
rtr
stop
unlk

When you have selected 64-bits implementations, it is possible to use .Q operations with any command. Scaling and rotation in 64-bits is not supported, though.

So... How do you program PPC680x0? Well, just act like you are using an 68000 Amiga with extra instructions. The 68000 registers are treated in the following way:

68k	PPC
d0-d24	r3-r27

```
 d25-d28     r28-r31  (Internal I/O registers. Don't use)
 a0-a7       r20-r27
```

Yes, you are right, a0-a7 are exactly the same as d17-d24. This means you may use both the 'd' or 'a' suffix. Using the 'a' suffix for addressing is easier as it defaults to 32/64-bits. Data registers always default to 16-bits if no size is given. This means that:

```
 move.l (a0),d0
```

Is the same mode as

```
 move.l (d17.l),d0
```

and is executed in the same amount of time. Using:

```
 move.l (d17),d0
```

defaults to word access, meaning that you are loading a value from a 16-bits address...!

PPC680x0 is very flexible. You can use as much indices (up to 32) as you want, meaning that the following is legal:

```
  add.q   label1([label2.b,d0.l*32,d1],label3.w,label4.l),d23
```

A sum is created in the following way:

```
 label1+label3&$ffff+label4&$ffffffff=replacement
 load replacement in I/O
 I/O+label2&$ff+d0&$ffffffff<<5+d1&$ffff=replacement
 load replacement in d23
```

All the normal and single instructions support all addressing modes. You should always keep in mind that the default modes are always the same as with the 680x0 CPU. (no size = .w except for address registers and some special modes)

The
mathematical~model
has some side-effects, meaning that some commands will default to the best mode. For example: When you are using 64-bits implementations without 32-bits Force enabled, a lea without size will default to .q as this is the most logical size.

VERY IMPORTANT

PPC680x0's machine language does not return to the program that runs it when using an 'rts'. You will even get a crash!!! You should use the 'lastrts' command for the exiting 'rts'. This means that using 'rts' is strictly for subroutines and 'lastrts' is to exit your code.

The Storm PowerASM command 'rs' is renamed to 'rs2' to keep compatibility with the command 'rs' used in many 68k assemblers.

There's also new rules for quick access. Moveq, addq, subq now support immediate values from 0-65535.

The exceptions to the rule:

- lea (b/w/l/q) is treated different. They only support the 'normal' 680x0 addressing modes but have .q and integrator access added
- bcd instructions ALWAYS default to byte access
- pea (b/w/l/q) is treated different. They only support the 'normal' 680x0 addressing modes but have .q and integrator access added

Working without carry emulation:

When carrying is not emulated, there's only a few commands with support for carrying:

roxr, roxl
additions (with or without extend)
substractions (with or without extend)

Other instructions only change the Positive, Negative, Zero and Overflow bits in CR0 of the PPC status register.

THERE'S A LOT MORE, BUT...!!!

From now on, we can recommend you to play around with PPC680x0 and find out it is very flexible. The original version will give you a big guide around the instructions and the way they are emulated, but we will now stop just here... (If we would tell you everything possible, there would be less reason to buy the program and this could not be called a promo anymore!)

1.10 Instruction Set

(Unofficial) PPCDirect Extended Instruction Set

This part describes the super (PPCDirect) instruction set but is cut down in the promo version. It does NOT describe any of the 680x0 instructions as it is not essential information for the promo edition.

These instructions are only useful if you want to optimize your codes or if you want to know a few extra handy instructions. There's a special section in the final guidefile where important or extremely useful commands are highlighted.

The
final~version
of PPC680x0 supports more than twice the amount of super-instructions given here. It will also describe all differences between original 68k instructions and PPC680x0's advanced architecture.

All super PPCDirect instructions are indicated with an 's' in front as the instruction. Instruction may look tricky at first sight, but when you cover up the first letter you will find out they look similar to 68k instructions.

You can select for every command (except a few where noted) if you want the Condition Register to be updated. It works in the same way as with PPC machine-code by putting a dot at the end of the instruction. No dot means no Condition Register update.

Please remember that the PPC Condition Register has no carry flag! This is done via a special register called XER. The CCR has bits for Positive, Negative, Zero and Overflow. The Overflow option of some of the instruction down here mean they set both the overflow bit of the CCR and the XER.

These instructions (except lastrts) should not be used together with CARRY emulation enabled if you don't have little knowledge of the PPC processor and architecture.

```
 SIMM  means Signed Immediate Value, which is always 16-bits
 UIMM  means Unsigned Immediate Value, which is always 16-bits
 rA,rB means any register
 rD    means any register (destination operand)
 M     means mask (see instruction for description)
```

LIST OF INSTRUCTIONS (CUT DOWN)

Non-PPCDirect Instructions

Section 1 - Implied

lastrts - quits source code. should be the last instruction your code executes.
The instruction is decoded to the PPC code:

```
mfctr  0
mtlr   0
blr
```

Section 2 - Kernal Instructions

call - calls kernal routine. equals 'bl'

PPCDirect Instructions

Section 3 - Arithmetics Instructions

Simple description of the letters behind the names:

s = shifted 16 bits to the left
v = XER overflow enabled
c = XER carry enabled
m = minus one
z = zero
x = extended

NAME	FORMAT	DESCRIPTION
sadd	rA,rB,rD	- place the sum rA+rB in rD

```
sadd    #SIMM,rB,rD  - place the sum SIMM+rB in rD
                       (DOES NOT SUPPORT CONDITION UPDATE)

sadds   #SIMM,rB,rD  - SIMM is a signed immediate value and is
                       shifted left 16 bits. The sum is
                       (SIMM<<16)+rB and is placed in rD
                       (DOES NOT SUPPORT CONDITION UPDATE

saddv   rA,rB,rD     - place the sum rA+rB in rD with overflow
                       enabled

ssub    rA,rB,rD     - place the sum rA-rB in rD

ssubv   rA,rB,rD     - place the sum rA-rB in rD with overflow
                       enabled

saddc   rA,rB,rD     - place the sum rA+rB in rD with carrying
                       enabled

saddc   #SIMM,rB,rD  - place the sum SIMM+rB in rD with carrying
                       enabled

ssubc   rA,rB,rD     - place the sum rA-rB in rD with carrying
                       enabled

ssubc   #SIMM,rB,rD  - place the sum SIMM-rB in rD with carrying
                       enabled

ssubcv  rA,rB,rD     - place the sum rA-rB in rD with carrying and
                       overflow enabled

saddcv  rA,rB,rD     - place the sum rA+rB in rD with carrying and
                       overflow enabled

saddx   rA,rB,rD     - place the sum rA+rB+CARRY(XER) in rD

saddxv  rA,rB,rD     - place the sum rA+rB+CARRY(XER) in rD with
                       overflow enabled

ssubx   rA,rB,rD     - place the sum rA-rB+CARRY(XER) in rD

ssubxv  rA,rB,rD     - place the sum rA-rB+CARRY(XER) in rD with
                       overflow enabled

saddm   rA,rD        - place the sum -1+rA+CARRY(XER) in rD

saddmv  rA,rD        - place the sum -1+rA+CARRY(XER) in rD with
                       overflow enabled

ssubm   rA,rD        - place the sum -1-rA+CARRY(XER) in rD

ssubmv  rA,rD        - place the sum -1-rA+CARRY(XER) in rD with
                       overflow enabled

saddz   rA,rD        - place the sum 0+rA+CARRY(XER) in rD
```

```
saddzv  rA,rD        - place the sum 0+rA+CARRY(XER) in rD with
                       overflow enabled

ssubz   rA,rD        - place the sum 0-rA+CARRY(XER) in rD

ssubzv  rA,rD        - place the sum 0-rA+CARRY(XER) in rD with
                       overflow enabled

Section 4 - Logical Instructions

Small description of letters behind names:

v   = overflow
l   = longword operation
q   = quadword operation
s   = shifted 16 bits to the left
i   = immediate
<<n = shifted left n times

sneg    rA,rD        - negate rA in rD

snegv   rA,rD        - negate rA in rD with overflow enabled

scmp.l  rA,rD        - compare rA/SIMM with rD treating operands
        #SIMM,rD       as signed 32-bits integers
                       (DOES NOT SUPPORT CONDITION UPDATE)

scmpl.l rA,rD        - compare rA/UIMM with rD treating operands
        #UIMM,rD       as unsigned 32-bits integers
                       (DOES NOT SUPPORT CONDITION UPDATE)

scmp.q  rA,rD        - compare rA/SIMM with rD treating operands
        #SIMM,rD       as signed 64-bits integers
                       (DOES NOT SUPPORT CONDITION UPDATE)

scmpl.q rA,rD        - compare rA/UIMM with rD treating operands
        #SIMM,rD       as unsigned 64-bits integers
                       (DOES NOT SUPPORT CONDITION UPDATE)

sandi   #UIMM,rB,rD  - UIMM AND rB, place result in rD. All other
                       bits than UIMM are set to zero
                       (ALWAYS SUPPORTS CONDITION UPDATE)

sandis  #UIMM,rB,rD  - UIMM<<16 AND rB. place result in rD. All
                       other bits than UIMM are set to zero
                       (ALWAYS SUPPORTS CONDITION UPDATE)

sori    #UIMM,rB,rD  - UIMM OR rB, place result in rD
                       (DOES NOT SUPPORT CONDITION UPDATE)

soris   #UIMM,rB,rD  - UIMM<<16 OR rB, place result in rD
                       (DOES NOT SUPPORT CONDITION UPDATE)

seori   #UIMM,rB,rD  - UIMM EOR rB, place result in rD
                       (DOES NOT SUPPORT CONDITION UPDATE)

seoris  #UIMM,rB,rD  - UIMM<<16 EOR rB, place result in rD
```

(DOES NOT SUPPORT CONDITION UPDATE)

```
sand    rA,rB,rD     - rA AND rB, place result in rD

sor     rA,rB,rD     - rA OR rB, place result in rD

seor    rA,rB,rD     - rA EOR rB, place result in rD

snand   rA,rB,rD     - rA NAND rB, place result in rD

snor    rA,rB,rD     - rA NOR rB, place result in rD

seqv    rA,rB,rD     - rA EOR rB, then complement and place result
                       in rD

sandc   rA,rB,rD     - Logical AND rA with the one's complement of
                       rB and place result in rD

sorc    rA,rB,rD     - Logical OR rA with the one's complement of
                       rB and place result in rD

scountl rA,rD        - count the amount of leading zero's in
                       longword rA and place that amount in rD

scountq rA,rD        - count the amount of leading zero's in
                       quadword rA and place that amount in rD
```

Section 5 - Shift & Rotate Instructions

Small description of letters behind names:

l = longword operation
q = quadword operation

Note about mask generation: On PPC, the leftmost bit is called bit 0 and the rightmost bit 63 or 31 depending on QUAD or LONG operation. This means that the most significant bit is always bit0

```
srolcl  rA,rB,rD,M         - rotate rB left rA/SIMM times, AND the
        #SIMM,rB,rD,M        result with the generated mask and
                             place it in rD. The mask is generated
                             having 1 bits from M to bit 31 and
                             0 bits elsewhere (quadword)
                             (NOT WORKING ON PPC603/604)

srolcr  rA,rB,rD,M         - rotate rB left rA/SIMM times, AND the
        #SIMM,rB,rD,M        result with the generated mask and
                             place it in rD. The mask is generated
                             having 1 bits from bit 0 to M and
                             0 bits elsewhere (quadword)
                             (NOT WORKING ON PPC603/604)

srolc   rA,rB,rD,M         - rotate rB left rA/SIMM times, AND the
        #SIMM,rB,rD,M        result with the generated mask and
                             place it in rD. The mask is generated
                             having 1 bits from M to bit 63-SIMM
                             and 0 bits elsewhere (quadword)
```

```
                              (NOT WORKING ON PPC603/604)

srolal  #SIMM,rB,rD,M1,M2  - rotate rB left rA/SIMM times, AND the
                             result with the generated mask and
                             place it in rD. The mask is generated
                             having 1 bits from M1 to M2 and 0
                             bits elsewhere (longword)

srolmil #SIMM,rB,rD,M1,M2  - rotate rB left rA/SIMM times. The
                             result inserted in rD at the place of
                             a mask generated from M1 to M2
                             (longword)

 Example:    srolmil #4,d1,d2,28,31

 rotate register d1 4 times. The mask is set from bit 28 to bit
 31, meaning that only the least significant 4 bits are replaced
 with the least significant 4 bits of the result.

slsll   rA,rB,rD           - shift left rB by rA times and place
                             the result in rD (longword)

slsrl   rA,rB,rD           - shift right rB by rA times and place
                             the result in rD (longword)

sasll   rA,rB,rD           - shift left rB by rA times and place
                             the result in rD (longword)

sasrl   rA,rB,rD           - shift right rB by rA times with
                             arithmetical sign extension and place
                             the result in rD (longword)

srolmiq #SIMM,rB,rD,M1,M2  - rotate left rB by SIMM times. The
                             result inserted in rD at the place of
                             a mask generated from M1 to M2
                             (quadword/NOT WORKING ON PPC603/604)

 Example:    srolmiq #4,d1,d2,60,63

 rotate register d1 4 times. The mask is set from bit 60 to bit
 63, meaning that only the least significant 4 bits are replaced
 with the least significant 4 bits of the result.


slslq   rA,rB,rD           - shift left rB by rA times and place
                             the result in rD (quadword)

slsrq   rA,rB,rD           - shift right rB by rA times and place
                             the result in rD (quadword)

saslq   rA,rB,rD           - shift left rB by rA times and place
                             the result in rD (quadword)

sasrq   rA,rB,rD           - shift right rB by rA times with
                             arithmetical sign extension and place
                             the result in rD (quadword)
```

1.11 Optimizing

```
Optimizing

This part is ridicously big with the original version and is extremely
cut-down in the promo version.

Some small ways of optimizing PPC680x0 code:

 1* Use as much .max access as possible with instruction forms.
    .max means that when you are using 32-bits implementations
    you should use as much .L instructions as possible and with
    64-bits implementations you should use as much .Q as
    possible, except for shifting and rotations, that may have
    .L-usage maximally
 2* For displacements, you should use as much .B/.W sizes as
    possible. So this means that when using indices it is
    recommended to use 16 or 8-bits as much as possible. This is
    only for the indices, NOT the instruction size! (See rule 1)
 3* Use quick operands for move, add, sub when you are using
    immediate values from 0-65535.
 4* Use as much forms with the label lower or the same as
    'move    label.w(regS.l),regD.l' for all kinds of memory
    access. This can be translated to a single instruction.

    example:   move.l   label(a0),d3

    is translated to:   lwz    r6,label(r20)

    Make sure register usage is .max
 5* Try coding without carry emulation
 6* Pre-decrement is faster than post-increment
 7* Use FRC when your PPC does not support 64-bits addressing
 8* Use r1 as stack-pointer
 9* Try using some super instructions as they can drastically
    improve performance

SORRY! We can't tell you everything already!!!
```

1.12 Quick Guide

```
                PPC680x0 QUICK GUIDE

Wanna have a quick start? Good... Here we go then. Go to Workbench and copy
the PPC680x0:Fonts/ directory into the SYS:Fonts/ directory or simply
double-click the 'install font' icon. You can now double the 'PPC680x0' icon
from anywhere.

After double-clicking the PPC680x0 icon, you get a window with some buttons,
strings, a scroller and a picture.

Pressing any button will remove the picture and
                display~all~the~information
                on the place of the picture.
```

The buttons mean the following:

Button 1 - two arrows: This means interleaving on/off. Interleaving means that the original 68k source-code file is placed throughout the file as comments. Default is on.

Button 2 - r1 : This button selects the stackpointer register. Default is r1 (the official stackpointer) but you can change it to r27 (=a7) to create a user-stackpointer with the size (in longwords) as selected in the 'stacksize a7' string. This will put a user-stack on bottom of the output source-code. (ds.l size)

Button 3 - a7 sp : This means that (a7) is treated exactly the same as (sp). Default is on. When disabled, you can use (a7) as a general purpose register (if r1 is selected as the stackpointer).

Button 4 - Wup : WarpUp goodies/PowerUp goodies. Default is WarpUp. Sadly, this is disabled in the promo version!

Button 5 - Gpr : Display General Purpose Registers or Floating Point registers in the '680x0 register usage' part. Default is General Purpose Registers. You can switch in realtime, but the promo-version has no FPU detection or support.

Button 6 - ?ERR : Display unknown errors or trash as ERRORUNKNOWN. When disabled, unknown errors or trash is ignored.

Button 7 - Proc Dir : Process Directory. Allows you to convert an entire directory.

Button 8 - 32 : Switch between 32/64-bits implementations. Default is 32-bits.

Button 9 - 00 0 : Carry emulation on/off. Default is off. The promo version has about 80% of the carry emulation implemented.

Button 10 - Rest Off : Rest emulation for divisions. Not working in the promo version...

Button 11 - Stat Off : Status register/Condition Code register/ Lowlevel emulation on/off. Only works with Condition codes in the promo version! (but no instruction to/from CCR/SR!)

Button 12 - Frc : Force 32-bits addressing modes. Default is on. This selects if you want to use 32-bits addresses as standard when working in 64-bits implementations. This is useful if you are using a PPC processor with a 32-bits address-bus. Has no effect when working in 32-bits implementations.

Button 13 - (disk 1) : ASL requester for source-file selection

Button 14 - (disk 2) : ASL requester for destination-file selection

Button 15 - M : Big Motorola M. Push to start converting

Before you start, change the quitting 'rts' in your source-code to 'lastrts'. The quitting 'rts' is the one that returns to the program that executed it. (Debugger/Workbench etc.)

So... Now you can start converting! Select the source and destination file and press the M button. You now get a requester:

'Should I Make A Start Header?'

You should only select 'Yes' if this is the first file (or only file) of the source-code (so actually the file that is normally assembled and runned). For anything else, press 'No'...

The convertion starts... You finally get a requester:

'Should I Make An End Header?'

Select 'Yes' if this is the file where you want the PPC680x0 internal data to be placed. (Such as user stack)

You are now ready! Load your file with any editor or try to assemble it! The output codes are Storm PowerAsm compatible.

1.13 Notes & Thank you's

NOTES AND THANK YOU'S

We would like to thank and greet the following people:

Teemu Suikki and all of Petsoff Limited Partnership for giving us the ability to support the best sound card on the Amiga: Delfina

Sam Jordan from Haage and Partner for giving us the ability to add Storm PowerAsm support to PPC680x0.

Ian Greenaway from White Knight Technology for helping and supporting us. Thanks for everything!!!

EBV Elektronik for sending us nearly ALL PowerPC/56x0y books and almost infinite information on all CPU's available!!!

Sander Assenbroek Machielsen for being one of the very last good Amiga freaks...

Bieuw (AKA Bart Neumann) for being a cool dude and of course a good Amiga freak as well!!!

*Garcon Fygar (AKA Jonathan Van Dijk) for being the happy co-pot-smoker!!!

(anyone we forgot)

We would like to send a very angry 'WAKE UP' to Simon N. Goodwin of Amiga

```
                Format
                 for telling us we can't do anything and are just claiming we will
release stuff that does not exist (without wanting to receive demos and
preview-editions)... Also we want to 'thank' him for his faith in the Amiga
scene and calling Nerve Axis a bunch of crap coders... SHIIIIIIIIIT!!!
```

1.14 Ordering PPC680x0

```
                Ordering PPC680x0

PPC680x0 will be released at the World Of Amiga 1999 show, held  on the 24th
and 25th of July 1999 in London.

Pre-ordering or buying PPC680x0 at, before, or until a week after the World
Of Amiga show will be very cheap. The price of  the full version will
normally be priced around $70, but at or  before the WOA show, you can get
it for only $45. Make sure you  will be there in time!!!

You can always
                contact us
                 for questions!

Beware: We are on vacation from the 8th of July until the 25th.
```

1.15 Copyright

```
                Copyright?

This promo is now freeware!

Copying is now legal without permission from
                the~authors
                .


REMEMBER:

A PROGRAM WORTH USING IS A PROGRAM WORTH BUYING!!!


OTHER RIGHTS:

 PowerUp is courtesy of phase5
 WarpUp & Storm PowerAsm is courtesy of Haage&Partner
```

1.16 Author

Contacting Us!

Do you want the full version of PPC680x0??? Or perhaps you have questions or want to talk about Mighty Mouse or anything... Well, you can always contact us at:

Coyote Flux WHQ
Rijenpad 21
1324 WC Almere
Holland

or call:

+31(0)36-5334238

or E-mail:

sraghoeb@ocenl.nl

THE AMIGA & C64 ARE THE GREATEST MACHINES EVER BUILT!!!

Coyote Flux has been with the C64 for about 12 years and with the Amiga for about 8. The Amiga is the best machine on the planet and can do anything every other machine can do. If you don't agree with us please contact us and we will tell you why you're wrong! PC SUXX!!!

COMING SOON:

CoyoteSound - The fastest and best sound-editor on the Amiga. Free Public Domain Mono version available. This program plays 14-bits sounds in 58 kHz on an 7.14 MHz 68000 Amiga and can play REAL 16-bits sound via Paula. It is also the only package that fully supports the Delfina 56k DSP in parallel to the Amiga.

Burnin' Rubber 2000 - A game that fits 2 CD's. Works on all ECS+ Amiga's with at least an 68030/50MHz, 4-speed CD-ROM, 1 MB Chip/4 MB Fast, harddisk and supports and fully utilitises Delfina DSP for 4 speaker surround sound, DSP screen and sound effects etc. The 2D section will have 4096 colors on screen with many lines of parallax scrolling, zooming, rotation etc. and the 3D section should beat EVERY 3D game on the other platforms! (unlimited polygons, morphing, phong, gouraud, fog, 24-bits, bumpmapping etc.)

Contact us for questions or information sheets!

TIME TO SHOW THE PC-GEEKS THAT AMIGA IS STILL NO #1 !!!!!!

1.17 AMIGA RULEZ!

OUR GREAT AMIGA

This program is dedicated to all the Amigans still living today and of course to the great Jay Miner. Thanks to all of you!

Amiga shall rise again!!!

1.18 PC SUXX!

PC MUST DIE!

Don’t you think so?

Let’s help the Amiga and kick all pirates to the pathetic PC platform...

PIRACY KILLS!

For all who truly love the Amiga and didn’t stop pirating: We truly don’t believe you ’love the Amiga’ so get yourself a baseball bat and do something about yourself or get a life...

1.19 Amiga Format...

Amiga Format Attitude

There’s so much we have to say about Amiga Format, but we have decided to give them a final chance. This area of the guide-file might be gone next time if AF takes the time to apology for their pathetic behaviour towards us in the last few months.

We have been buying AF since issue 1 and are terribly disappointed in their ways of treating the Amiga-scene and products in the last few (about two) years and know there are too many others agreeing with us. Sadly, we are one out of the few(?) who had personal trouble with AF in which they were highly insulting without any good reason.

Lets hope we don’t need to fill this part in the final version...

Beware Format, time is running out...

Please tell us if we only need to blame Mr. Goodwin...

1.20 New Notes

NEW IMPORTANT NOTES AND INFORMATION

Here’s some important information that you should know about PPC680x0...

PPC680x0 is a ’Pass 0’ assembler, which means that it does not detect or display all the reported bugs. When you will assemble the output file using an assembler it may give some other bugs concerning an area that is incorrect. This should always be a mistake in the input 68k source-code that caused PPC680x0 to convert it in the wrong way.

Labels do not enter the intelligent decoder yet so that complex parts may cause problems. Brackets in labels are disabled in the promo. Beware...

The promo has no support for 100% optimizations. Some of the instructions will cause more lines of code than needed but this wont be this way in the final version.

Block dependent decoding is available in the final version but will be released as seperate modules. So please understand that PPC680x0 WILL detect blocks of code!!!

Please understand that the code in this program is not yet 100% optimized! This is left over for the final version!