PCHG

COLLABORATORS			
	TITLE : PCHG		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		January 7, 2023	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

PCHG

1.1 PCHG.guide

This file contains the PCHG (Palette CHanGe) IFF chunk specs. ←
The
new IFF chunk PCHG allow to specify line-by-line palette changes in a
simple way which is independent of the video mode; while it allows up
to 65536 registers, it's usually shorter of an equivalent CTBL or SHAM
chunk. Library code with full source and documentation is provided for
a straightforward implementation in your programs. PCHG has been
developed in BIX through an open discussion of many Amiga programmers,
and it is our hope that it will become the Amiga standard for palette
change technology.

A Look at the Past
        What was before PCHG

Why PCHG?
                An introduction to the reasons for a new format

Design goals
            What we hope to do with PCHG

Informal description
        What's inside a PCHG chunk

Compression
                The compression technique used in PCHG

Writing changes
            The current hardware limitations of Copper lists

Code and Tools
            The PCHG library

Formal specification
        An (almost) precise regular grammar for PCHG

Author Info

1.2   PCHG.guide/A Look at the Past

A Look at the Past
****

   The capabilities of the Amiga old and enhanced chip sets are, in
terms of palette depth and on-screen available colors, not different
from those of a brand new '85 Amiga 1000. Hardware design limitations,
such as Chip RAM available bandwidth, forced Commodore designers to
left untouched these limitations until the delivery of the AGA chip
set. Nonetheless, not being able to display more than 16 colors in high
resolution is a deadly limitation for many applications, and the
installed base of Amigas with old or enhanced chip sets is very large.

   In the last years, several people tried to enhance the Amiga video
display by using line-by-line palette change technology. In short, by
using the Copper (and possibly the CPU) the colors stored in the
palette are changed while the video beam sweeps across the screen. As a
result, if you synchronize correctly the changes with the
vertical/horizontal blanking time you get a (partially) different
palette on each video line.

   The first known attempt in this direction was Rhett Anderson's
Sliced HAM, commonly named SHAM by the identifier of the corresponding
IFF ILBM chunk. SHAM fixes the color 0 to pure black, and reloads the
other 15 registers of the HAM palette with different values on each
line (if the image is interlaced, the colors are reloaded each two
lines instead).

   The following attempts (by NewTek and ASDG mainly) are generally
known as Dynamic HiRes, or Advanced HAM, et cetera (we will refer to
these standards as "dynamic modes"). They store in a chunk named CTBL
(ConTrol BLock I believe) a full 16 color palette for each line of a
picture. The CPU is involved in loading the palette, since there is not
enough time to do this with the Copper only.

   However, SHAM and dynamic modes have major faults both in the
definition of their IFF format and in their implementation.

   First of all, the original computation that led the SHAM designer to
think there was enough time to stuff data in 15 registers is plainly
wrong. The Copper needs some Chip memory accesses in order to fetch its
instructions. While a picture is displayed, part of these accesses can
be delayed because the video DMA requires Chip memory access, and of
course it has precedence over the Copper.

   When displaying a 320 pixel wide lo-res picture, the video DMA is
active during 160 clocks of the 226 available (HAM doesn't steal all of
the DMA accesses available, but we are interested in changing the
palette colors during the horizontal blanking, i.e., when the image is
not displayed). Thus, it could seem we have 66 clocks left. If we

subtract 6 clocks for a Copper 'WAIT' instruction, we are left with
exactly 15 'MOVE' instructions.

Unfortunately, this considerations skip over a major point of the
Amiga video DMA design: the video data fetch starts before the actual
display start, and ends a before the actual display end. This implies
that we have some clock less because of the early data fetch start, but
that we can't recover it after the data fetch end, because for some
clocks the video hardware will be still displaying our picture, and
writing in the palette registers at this point would cause an immediate
change of the video chip output. A more realistic computation leads to
about 13 color changes.

But this is not enough. When the Copper executes a comparison with
the vertical beam, it has a resolution of 8 bits. When it arrives at
the 255th video line, it wraps up to 0. Thus, if you try to build a
user Copper list which goes beyond the 255th video line, the system
places a 'WAIT(226,255)' Copper instruction in order to wait correctly
for the following lines.

If you want 13 changes, you have to start poking the color registers
with the Copper just after a video line is finished (as SHAM). But on
the 255th video line, 'MrgCop()' will merge your user Copper list with
the system one in such a way that the 'WAIT(226,255)' will happen after
the video vertical position passed 255, so that the Copper will be
locked until the next vertical blank. As a result, the following color
changes won't be executed, and some trash will be displayed at the
bottom of the screen (this indeed happens with SHAM). The reason no one
noticed this problem is that SHAM was originally hardwired to 200 lines
pictures (400 if laced). On an NTSC screen, and in the video default
position, the last line of such a picture won't usually pass the 255th
video line. However, on a PAL screen the 255th video line is at about
3/4 of the screen: the bug is readily noticeable as soon as you drag a
SHAM screen.

In order to avoid this, it is necessary to use only 'WAIT'
instructions which specify 0 as horizontal wait position. Then, the
time available before the display data fetch start allows only 7 color
changes.

Another problem with SHAM is that even if you set up correctly a
Copper list in such a way that the palette changes will start just
after the end of the display, under Release 2 the user has the
possibility of horizontally dragging an Intuition Screen. MrgCop() is
enough kind to renumber your user Copper lists when you drag vertically
a screen, but not when you drag it horizontally: if you do so, a series
of fringes will appear on one side of the SHAM picture. Again, this
problem can be avoided by waiting always for horizontal position of 0,
and by limiting the number of color changes.

Dynamic modes, on the other hands, have programmatically refused to
use the Copper only. As a result, you can certainly amaze friends and
relatives with stunning images, but you can't use any of those
beautiful pictures in an application. Moreover, the display programs
commonly used for such pictures rely on CPU busy loops which have to be
rewritten each time a new Amiga with a new CPU or a new clock rate
comes out. I still haven't found one such program which will work on my

A3000T (this is probably one of the reasons why ASDG dropped the
support for such modes in 'ADPro').

   On the IFF side, I can only say that all these formats are hardwired
to sixteen 12 bit color registers, and SHAM even disallows screens of
height different from 200 video lines.

   These constraints are unacceptable for serious purposes. For
instance, while preparing a CDTV video catalog the programmer could not
set up the user interface event he prefers for stopping the display of
a picture, because it would have to use a display program which would
freeze the machine; and the he would have to rewrite parts of the
display code in order to put in his preferred exit event.

1.3   PCHG.guide/Why PCHG?

Why PCHG?
****

   After struggling a lot with CTBL, SHAM, and whatever else was
invented for specifying palette changes in order to implement them in
'Mostra' (my ILBM viewer), I decided there was no way to make them
really work. Each program uses them in a different way, with different
non-documented specifications. SHAM is hardwired to 200 lines, and the
color of the last pixels of a screen depends on the horizontal position
of the screen itself because of a wrong computation of the free Copper
DMA slots. CTBL is theoretically undisplayable without freezing
everything and yet all images I ever saw changed much less than 15
colors per scan line, which you can perfectly do with the Copper
(thanks to ASDG's 'DDHR' utility for this info). There is moveover a
great confusion about the role of the CMAP chunk with respect to all
those guys.

   Yet the technology is very simple. Just change some color register
each scan line. Very Amiga specific, but it works, and it works really
well.

   This document describes the PCHG (Palette Changes) chunk, an ILBM
property chunk for controlling efficiently and reasonably the palette
changes at each scan line. Also, I included technical info and code
about the current allowable per line palette changes. A picture with a
PCHG chunk is called a "multi-palette" picture, just like a picture
with a CTBL chunk is called a "dynamic" or "enhanced" picture.
Multi-palette pictures are not restricted to a particular video mode.
You can have EHB, hires, HAM, etc. multi-palette pictures.

   This proposal is a team work. It was lively discussed with many other
people, including Joanne Dow, Andy Finkel, J. Edward Hanway, Charles
Heath, David Joiner, Jim Kent, Ilya Shubentsov, Mike Sinz, Loren
Wilton. There is certainly some other people I'm forgetting to mention
though.

   What's good in what follows was suggested by them. I'm responsable
for any error, omission, bad English and bad design.

1.4   PCHG.guide/Design goals

```
Design goals
************
```

  * Being able to specify *only* the changes which are really required.

  * Being able to specify 24-bit precision color changes, and an alpha
    channel.

  * Specifying correctly the relation PCHG/CMAP.

  * Getting a chunk which is usually smaller than SHAM or CTBL.

  * Having a policy about Copper-only displayability.

  * Being able to change 65536 registers.

  * Specifying two storage formats: a very dense 4-bit 32 register
    format for current technology, and an open-ended, 24-bit+alpha
    channel, 65536 register format with compression for all future
    uses.

  * Distributing public domain code for PCHG compression/decompression
    and Copper list building.

1.5   PCHG.guide/Informal description

```
                Informal description
********************
```

   PCHG starts with the following header:

```
   struct PCHGHeader {
       UWORD Compression;
       UWORD Flags;
       WORD  StartLine;
       UWORD LineCount;
       UWORD ChangedLines;
       UWORD MinReg;
       UWORD MaxReg;
       UWORD MaxChanges;
       ULONG TotalChanges;
   };
```

The only 'Compression' values currently defined are 'PCHG_COMP_NONE'
and 'PCHG_COMP_HUFFMANN'. The 'Flags' field has three bits currently
defined, 'PCHGF_12BIT', 'PCHGF_32BIT', 'PCHGF_USE_ALPHA'. The
'StartLine' and 'LineCount' fields specify the range controlled by the

line mask, as we will see later. The 'ChangedLines' field specify the
number of lines on which at least a change happens (i.e., the number of
1's in the line mask). The 'MinReg' and 'MaxReg' fields specify the
minimum and the maximum register changed in the chunk: their purpose is
to allow optimization (such as grouping of the modified registers in
some special bank). The 'MaxChanges' field specify the maximum number
of changes on a single line. The 'TotalChanges' field specify the total
number of color changes in the whole PCHG chunk.

   If Compression is 'PCHG_COMP_HUFFMANN', the rest of the chunk is a
compressed format. It is formed by a

```
    struct PCHGCompHeader {
        ULONG CompInfoSize;
        ULONG OriginalDataSize;
    };
```

followed by 'CompInfoSize' bytes which contain the decompression tree,
after which there is the compressed chunk (originally
'OriginalDataSize' bytes long). For information about the coding used
by PCHG, see
                    Compression
                        .

   The unpacked data (or the data following the 'PCHGHeader' in the
non-compressed case) are divided as follows.

   First of all, there is a array of '(LineCount+31)/32' longwords (that
is, a bit mask of LineCount bits rounded up to the nearest longword).
Each bit in the mask tells you if there are palette changes in the
corresponding line. Bit 0 of the mask (i.e., bit 31 of the first
longword) corresponds to line 'StartLine', bit 1 (i.e., bit 30 of the
first longword) to line 'StartLine+1' and so on. The number of 1's will
be equal to 'ChangedLines'. Note that 'StartLine' is a (possibly
negative) offset from the top of the screen.

   The information about the palette changes is stored immediately
after the bit mask. For each bit set to 1 in the mask there is a
variable length structure. These structures are recorded contiguously,
and they are different depending on the 'PCHGF_12BIT' or the
'PCHGF_32BIT' flags being set. In the first case, we use

```
    struct SmallLineChanges {
        UBYTE ChangeCount16;
        UBYTE ChangeCount32;
        UWORD PaletteChange[];
    };
```

The 'PaletteChange' array contains 'ChangeCount16+ChangeCount32'
elements. For each element, the lower 12 bits specify a color in 4-bit
RGB form, while the upper 4 bits specify the register number. More
precisely, for the first 'ChangeCount16' elements you take as register
number the upper 4 bits, and for the following 'ChangeCount32' elements
you take as register number the upper 4 bits+16. Thus, you can address
a 32 register palette.

   In the second case, we use

```
    struct BigLineChanges {
        UWORD ChangeCount;
        struct BigPaletteChange PaletteChange[];
    };
```

where

```
    struct BigPaletteChange {
        UWORD Register;
        UBYTE Alpha, Red, Blue, Green;
    };
```

The array 'PaletteChange' contains 'ChangeCount' elements. For each
elements, Register specifies the register number, while the 'Alpha',
'Red', 'Blue', 'Green' values specify the 8-bit content of the
respective channels. 'Alpha' should be parsed only if the
'PCHGF_USE_ALPHA' flag is set in the header. The meaning of the 'Alpha'
bits is currently undefined; it will be specified later. For they time
being, they *must* be set to 0.

    CMAP and PCHG don't interfere. It's up to the intelligence of the
IFF ILBM writer using CMAP for the first line color register values,
and then specifying the changes from line 1 (2 for laced pictures)
onwards using PCHG. CMAP has to be loaded, as specified by the IFF ILBM
specs.

    Note that PCHG is mainly a time saver chunk. The "right thing" for a
program should be generating at run-time the palette changes when a
picture with more colors than available on the hardware has to be
shown. However, the current computational power make this goal
unrealistic. PCHG allows to display in a very short time images with
lot of colors on the current Amiga hardware. It can be also used to
write down a custom Copper list (maybe changing only the background
color register) together with an image.

    Some politeness is required from the PCHG writer. PCHG allows you to
specify as many as 65535 per line color changes, which are a little bit
unrealistic on the current hardware. Programs should never save with a
picture more changes than available by using Copper lists only. This
issue is thoroughly explained in
                Writing changes
                . Moreover, under
Release 2 you may want to set the 'USER_COPPER_CLIP' of your 'ColorMap'
(via the 'VideoControl()' function); this will stop your Copper list
from debording on another screen.

    This kind of politeness is enforced by the specification. I have yet
to see people which is interested in freezing their machine just in
order to view a picture. DMA contention is a thing, lockup is another
one. PCHG chunks which do not conform to the rules explained below are
to be considered syntactically incorrect. If you want specify more
changes than available through the system Copper macro calls
'CWAIT/CMOVE', please have good reasons (such as a new Commodore chip
set or OS upgrade); otherwise, please use another chunk and don't mess
up the PCHG interpretation.

1.6 PCHG.guide/Compression

```
Compression
***********
```

   (Caveat: you don't need to read this if you're not really interested
because there are ready-to-use C functions for compression and
decompression; moreover, 4-bit PCHG chunks are usually so entropic that
the size gain is less than the size of the tree, so you shouldn't
compress them.)

   PCHG uses a classical static Huffmann encoding for the line mask and
the `LineChanges' array. The coding tree is recorded just before the
compressed data in a form which takes 1022 bytes or less (usually
~700). Its (byte) length is stored in the `CompInfoSize' field of the
`PCHGCompHeader' structure. Moreover, this form is ready for a fast and
short decompression algorithm--no preprocessing is needed. For
references about the Huffmann encoding, see Sedgewick's `Algorithms in
C'. Note that the number of compressed data bits stored is rounded up
to a multiple of 32 (the decompression routine knows the original
length of the data, so the exceeding bits won't be parsed). Note also
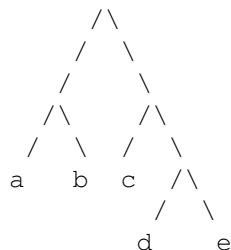that left branches are labelled by 0, right branches are labelled by 1.

   The format of the tree is recursive. We start to code from the end
of a 511 `WORD' array, and we work backwards. To code an internal node
at the position `WORD *Pos', the left subtree is recorded at `Pos-1'
with a code of length `t', and the left subtree is coded at `Pos-1-t'.
Then an offset `(-t-1)*2' is stored in `Pos', and the length of the
resulting coding is `1+t+'length of the left subtree code. An external
node is coded as the character associated with the ninth bit set. As a
final optimization, if the left subtree to code is an external node, we
just store the character associated in the place of the negative offset

   For instance, the tree

```
                    /\
                   /  \
                  a    b
```
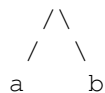
is coded as the word array `[a | 0x100] [b]'. Note that without the
ninth bit trick, it would be impossible to store this tree, since it
would be confused with the tree formed by the external node b only.

   Another simple example:

```
                    /\
                   /  \
                  /    \
                 /\    /\
                /  \  /  \
               a    b c  /\
                        /  \
                       d    e
```

is coded as

        [d | 0x100] [e] [c | 0x100] [-4] [a | 0x100] [b] [-6].

    Decompression is very easy. We start from the end of the tree code.

    If we pick a 0 bit in the packed data, we move a word to the left
and fetch the current word. If it's positive and with the ninth bit set
the tree is finished and we store to the destination the lower byte of
the word we fetched, otherwise we pick another bit.

    If we pick a 1 bit, we fetch the current word. If it's positive, we
store it. Otherwise we add it to the current position and we pick
another bit. (Here you can see the reason why the offset is not stored
as a word offset, but rather as a byte offset. We avoid a conversion
word->byte offset for each bit set to 1 of the source).

1.7   PCHG.guide/Writing changes

Writing changes
****

    PCHG is a machine-independent format. Nonetheless, it's been
developed mainly for supporting the Amiga Copper list palette changes.
Thus, it's not a surprise to find included with the format definition a
policy about the amount of color changes which you should write.

    Under the current Amiga hardware and system software, you should
never generate more than 7 (seven) changes per line. Moreover, in laced
pictures the changes can only happen on even lines. Thus, for a 400
lines laced picture you have 200*7=1400 color changes at lines 0, 2, 4,
etc., while for a 256 lines non laced picture you have 256*7= 1792
color changes at lines 0, 1, 2, etc. Of course you can save less
changes, or no changes at all on some lines. (When displaying a
scrolling playfield the DMA prefetch needed by the video hardware
limits the changes to 5. High resolution screens with more than 5
changes can exhibit glitches in some positions if scrolled, for
instance, with a viewer which supports the new 'AUTOSCROLL' screens.
This constraint, however, is not relevant enough to force the general
bound to 5 changes.)

    The point here is that you shouldn't save more changes than that. If
you want to write a picture with more changes, or changes on odd laced
lines, please make aware the user of the fact that probably most viewer
supporting PCHG won't be able to display it. The Amiga community has
been already bitten by the problems of SHAM and CTBL, and we have
neither need nor willing of repeating the experience. (Of course, this
limitation pertains to the old chip set and to the enhanced chip set
only; see the end of this section for a discussion about the AGA chip
set.)

    If you have a technical background about the Copper, that's why:

The Copper y register has 8-bits resolution. When it arrives at the
255th video line, it wraps up to 0. Thus, the system places a
'WAIT(226,255)' Copper instruction in order to stop correctly the video
display on PAL screens.

   If you want more than 7 changes, you have to start poking the color
registers with the Copper just after a video line is finished. But on
the 255th video line, 'MrgCop()' will merge your user Copper list with
the system one in such a way that the 'WAIT(226,255)' will happen
after the counter wrapped, so the Copper will be locked until the
next vertical blank. As a result, the following color changes won't be
executed, and some trash will be displayed at the bottom of the screen
(this indeed happens with SHAM).

   In order to avoid this, it is necessary to use only 'WAIT(0,<line>)'
instructions. The time available before the display data fetch start
allows only 7 color changes, and wide range experiments confirmed this.

   Finally, due to a limitation of 'MrgCop()', it's not possible
specifying 'WAIT' instructions on odd interlaced lines (it is because
interlaced screens are displayed in two passes).

   The AGA chip set, and future chip sets, have of course much less
stringent limitations. However, Commodore did not still upgrade the
Copper and the system interface to Copper lists. This means that, for
the time being, you cannot produce 24 bit color changes. Moreover, the
power of the new chip sets makes unnecessary to diddle with palette
changing. If, and when, new approaches to Copper programming will be
available, this proposal will be updated and redistributed.

1.8   PCHG.guide/Code and Tools

Code and Tools
****

   This specification is distributed with a complete set of C functions
which take care of compression, decompression and Copper list building.
Adding support for PCHG in your programs should be pretty
straightforward: you simply have to link with 'pchg.lib' (or
'pchgr.lib', if you want to use register parameter passing under
SAS/C). Documentation is provided in the standard Amiga autodoc format
(thus, you can turn it in an AmigaGuide hypertext document via the
suitable utility).

   A simple utility, 'ToPCHG', allows to produce sample multipalette
pictures of any kind starting from SHAM or CTBL pictures. It can be
used in order to test esoteric features.

   Ed Hanway's 'HamLab' and Steven Reiz's 'wasp' currently support
conversion to multipalette. The images which (should) accompany this
documentation were produced by 'HamLab'.

1.9 PCHG.guide/Formal specification

```
Formal specification
********************

     struct PCHGHeader {
        UWORD Compression;
        UWORD Flags;
        WORD  StartLine;
        UWORD LineCount;
        UWORD ChangedLines;
        UWORD MinReg;
        UWORD MaxReg;
        UWORD MaxChanges;
        ULONG TotalChanges;
     };

     struct PCHGCompHeader {
        ULONG CompInfoSize;
        ULONG OriginalDataSize;
     };

     struct SmallLineChanges {
        UBYTE ChangeCount16;
        UBYTE ChangeCount32;
        UWORD PaletteChange[];
     };

     struct BigLineChanges {
        UWORD ChangeCount;
        struct BigPaletteChange PaletteChange[];
     };

     struct BigPaletteChange {
        UWORD Register;
        UBYTE Alpha, Red, Blue, Green;
     };

     PCHG ::= "PCHG" #{ (struct PCHGHeader) (LINEDATA | HUFFCOMPLINEDATA) }


     HUFFCOMPLINEDATA ::= { (struct PCHGCompHeader) TREE HUFFCOMPDATA }
     TREE ::= { UWORD* }
     HUFFCOMPDATA ::= { ULONG* }

     HUFFCOMPDATA, when unpacked, gives a LINEDATA.


     LINEDATA ::= { LINEMASK ((struct SmallLineChanges)* |
                             (struct BigLineChanges)*) }
     LINEMASK ::= { ULONG* }

     The following relations hold:

     #LINEDATA == PCHGCompHeader.OriginalDataSize
     #TREE  == PCHGCompHeader.CompInfoSize
```

```
        #LINEMASK == ((PCHGHeader.LineCount+31)/32)*4
```

PCHG is a property chunk. For the meaning of the above grammar, see the
IFF documentation (the grammar does not give account for all the
aspects of PCHG though). Note that my use of the '[]' notation for
variable length arrays is not a C feature, but a shorthand.

1.10 PCHG.guide/Author Info

Author Info
****

        Sebastiano Vigna
        Via California 22
        I-20144 Milano MI

        BIX: svigna
        INTERNET: vigna@ghost.sm.dsi.unimi.it
        UUCP:cbmehq!cbmita!sebamiga!seba@cbmvax.cbm.commodore.com
            ...{uunet|pyramid|rutgers}!cbmvax!cbmehq!cbmita!sebamiga!seba