lib

COLLABORATORS			
	TITLE : lib		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 31, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

lib

1.1 4 The tcs.library

```
4   The tcs.library


After all that theory in the previous chapter (wHaT?!? You have *not*
read a single thing of all that stuff?!? You'd better have a look at the
basic information, at least!), I guess you now want to do some practice;
there are two ways:

 a) hard: you digest the whole documentation and write your own routines
 b) easy: you simply use the tcs.library I wrote for you!!!

No more things to say about a); so let's get started with b):

 4.1   Preliminary Operations
 4.2   General Guidelines
 4.3   Declarations Description
 4.4   Functions for Displays
 4.5   Functions for Color/Palette Control
 4.6   Functions for Graphics
 4.7   Functions for Picture Files
 4.8   Simple Meta-Example
 4.9   Known Bugs & Problems



 - to be able to follow the links here and in the following sub-sections
   you should have the files "tcs.i" and "tcs_lib.i" in the directory
   "INCLUDES:libraries/"
```

1.2 4.1 Preliminary Operations

```
4.1   Preliminary Operations
```

Installing the tcs.library is of course a matter of seconds: just put the file "tcs.library" in your "LIBS:" drawer.
Then copy the files tcs.i and tcs_lib.i anywhere you prefer: they are the only two includes you'll need to write your programs (for instance you could keep them in the directory "libraries/" in the same drawer where all the other AmigaOS includes (exec/, intuition/, etc.) are stored.

The files supplied in TCS/pal/ are the palettes (IFF/ILBMs with the BMHD * and CMAP chunks only) needed for creating new pictures or remapping pre-existent ones to the built-in RGBx formats. These files are _not_ used by the library, so you can store them wherever you want (you can even delete them - however, I don't recommend this).

1.3 4.2 General Guidelines

4.2 General Guidelines

Generally, you'll have to perform your operations in this order:

1. initialize one or more displays
2. show it/them
3. work with it/them
4. hide it/them
5. free it/them

Although a bit complicated, luckily for users, initializations are made transparent to them by the function TCS_InitDspl(): here we just need to say that this function returns a pointer to a structure where you can find the address of the chunky buffer relative to the display; moreover, this pointer is fundamental as almost any other function needs it: store it in a safe place, you'll often have to use it!
Once the display has been set up, you can anytime activate it; note that nothing stops you from having different displays initialized at the same time (provided you have enough memory!): the one which is (or will be) actually shown will not be affected at all.
Certainly, once you are finished with the display(s) you've created, all the resources they have been assigned must be released: the means to do this is provided by another function exactly opposite to the one that performs the allocations.

As concerns Cross Playfield, you can operate in two ways:

a) initialize separately two normal displays and then put them together with TCS_EnbXPfld()
b) initialize a normal display, a "front-playfield-only" display and then put them together with TCS_EnbXPfld()

the difference between the two methods is that b) uses much less memory (the drawback is that the second display cannot be shown alone).

To use a picture, you can:

 a) load it with TCS_LdILBM()
 b) convert it to raw chunky with your preferred tool and load it as plain
    binary data

To remap a picture to an RGBx mode of your choice, follow these steps:

 1. run an image-processing program
 2. load the picture you want to convert
 3. load the chosen RGBx palette from TCS/pal/
 4. use the program's "remap" (or whatever it is called) function
 5. save the new picture in ILBM format

Finally, here are some other important notes:

 - tcs.library functions, as any other library's, don't guarantee that
   the content of the registers d0, d1, a0, a1 is preserved
 - when needed, always check the validity of the value returned in d0
   or in the ccr before going further
 - be aware that, unless otherwise stated, functions will *never* perform
   checks on input values correctness! It's all up to you!!!
 - some functions expressely require the AmigaOS to be ON and some others
   to be (partially) OFF: pay attention to their description
 - don't call [de-]allocation functions from interrupts, because they use
   exec.library AllocMem() and FreeMem()!

1.4 4.3 Declarations Description

4.3   Declarations Description

Before going on, I'd better shortly illustrate the meaning of some declarations of public interest in tcs.i (which you have to refer to as only small parts of that code will be reproduced here); for the declarations not explained here, please try to understand the comments: the items which could interest you are usually rather self-explaining.

- no description of the TCSBase structure is given because it holds

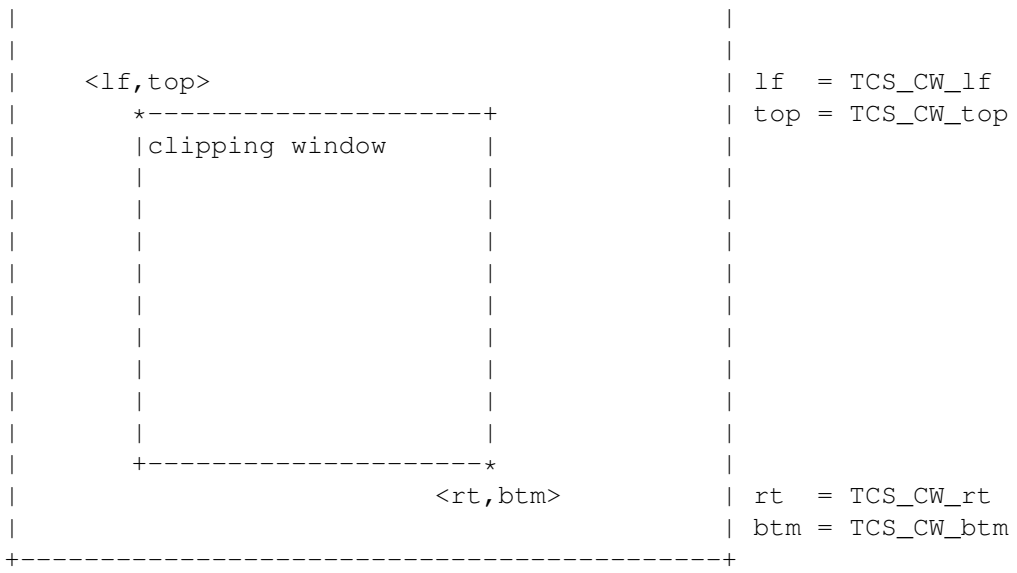
```
  no relevant item; the library name, as usual, is given as a short macro
- tcs_lib.i, as you may expect, contains the Library Vector Offsets
```

1.5 4.3.1 VideoModes definitions, bits & flags

```
4.3.1   VideoModes definitions, bits & flags


These definitions are very important as they are needed to specify which
video mode (VdoMode) to assign to a display at its initialization.
They are given in the common bits & flags fashion and can be used in the
classic way: to form the desired bit-map value put the appropriate flags
together by and/or-ing the TCS_VMf_#? values and/or by bclr/bset-ing the
relevant bits using the TCS_VMb_#? bit indexes.

A VdoMode definition consists of any TCS_VM_RGBx value (that selects the
RGBx method wanted), optionally or-ed with:

 TCS_VMf_MskPln : enable MskPln (HalfRes only)

 TCS_VMf_chqr   : enable ChqrMode (HalfRes only)

 TCS_VMf_HScrl  : enable horizontal scrolling (HalfRes only)

 TCS_VMf_FullRes: enable FullRes video mode (clears: TCS_VMf_MskPln,
                  TCS_VMf_chqr, TCS_VMf_HScrl)

 TCS_VMf_BltFRP : enable Blitter-assisted FullRes (FullRes only)

 TCS_VMf_DubBuf : enable double buffering

 TCS_VMf_TriBuf : enable triple buffering (clears TCS_VMf_DubBuf)

 TCS_VMf_FPfld  : use display exclusively as front playfield when the
                  Cross Playfield mode is active (this display cannot be
                  shown alone)
```

1.6 4.3.2 ClippingWindow structure

```
4.3.2   ClippingWindow structure


This structure is used to define the only area of the screen which can be
affected by the graphic functions.
The fields indicate the coordinates of the sides of the clipping window
as follows:

<0,0>
  *-------------------------------------------+
  |chunky screen                              |
```



1.7 4.3.3 GraphicContext bits, flags and structure

4.3.3 GraphicContext bits, flags and structure

A Graphic Context is a structure which dictates how the graphics drawn by Graphic Context -sensitive functions must look.

The structure fields are:

TCS_GC_flgs : bitmap made of any combination of the TCS_GCf_#? flags

TCS_GC_ClpWin: ClippingWindow structure to use when the TCS_GCb_clp bit of TCS_GC_flgs is set

1.8 4.3.4 DisplayDeclaration structure

4.3.4 DisplayDeclaration structure

This structure tells the library what kind of display you want to open, so you must know perfectly the meaning of every item:

TCS_DD_VdoMode: see here

TCS_DD_UsrLst0: address of the first user copperlist: when _some_ early settings have been done by the main copperlist and _before_ the first line is drawn on the screen and passing to chequer and/or scroll settings (when needed), this address is loaded to COP2LC and then the Copper is forced to jump with a write to COPJMP2.

```
                To resume main copperlist execution, the user copperlist
                *must* end with COPMOVE to COPJMP1 (ex.: dc.w $88,1).
                The user copperlist(s) can freely redifine COP2LC, but
                *cannot* touch COP1LC, as it holds the main copperlist
                resume address.
                The user copperlist is always executed before the line
                indicated by TCS_DD_DsplY0 and should return (i.e. must
                be closed by the declaration above) before such line.
                Set it to 0 if no user copperlist is required

TCS_DD_UsrLst1: address of the second user copperlist: when _all_ the
                settings have been done by the main copperlist, this
                address is loaded to COP2LC and then the Copper is forced
                to jump with a write to COPJMP2.
                There is no particular restriction on how this copperlist
                must end.
                This copperlist is _always_ executed at the end of the
                main copperlist, so note that if ChqrMode is ON then it
                will be started after almost the whole screen has been
                drawn (only part of the last rasterline remains).
                Set it to 0 if no user copperlist is required

TCS_DD_DsplX0,
TCS_DD_DsplY0 : the coordinates of the top-left corner of the display
                window in SHRES pixels as in DIWSTRT+DIWHIGH (just the
                values, not the format!)
                 - TCS_DD_DsplX0 >= TCS_DL_MinX0
                 - TCS_DD_DsplY0 >= TCS_DL_MinY0

TCS_DD_DsplX1,
TCS_DD_DsplY1 : the coordinates of the bottom-right corner of the display
                window in LORES pixels as in DIWSTOP+DIWHIGH (just the
                values, not the format!)
                 - TCS_DD_DsplX1 <= TCS_DL_MaxX1
                 - TCS_DD_DsplY1 <= TCS_DL_MaxY1
                 - TCS_DD_DsplX1-TCS_DD_DsplX0 >= TCS_DL_MinWd
                 - TCS_DD_DsplY1-TCS_DD_DsplY0 >= TCS_DL_MinHt

TCS_DD_ScrWd  : width in TCS pixels of the screen to open (if necessary,
                it will be rounded to next multiple of 8)
                 - TCS_DD_ScrWd >= DsplWd/8 [HalfRes]
                 - TCS_DD_ScrWd >= DsplWd/4 [FullRes]
                 where DsplWd = (TCS_DD_DsplX1-TCS_DD_DsplX0) rounded to
                 the next multiple of 64 because 64-bit burst for bit-
                 plane data fetch is used

TCS_DD_ScrHt  : height in pixels of the screen to open
                 - TCS_DD_ScrHt >= TCS_DD_DsplY1-TCS_DD_DsplY0

TCS_DD_brtns  : brightness degree in [0...256] of display at startup

TCS_DD_GfxCtxt: GraphicContext structure for the default Graphic Context
                assigned to the display
```

1.9 4.3.5 DisplayInfo structure

```
4.3.5    DisplayInfo structure


This structure is really important as it is used by almost any function.
It's created and mantained automatically and normally you shouldn't really
feel the need to access it. Yet, it contains precious info, so I'm going
to talk about the most important fields:

 TCS_DI_MainLst: address of the main copperlist of the display.
                 There is a single copy of this copperlist per display,
                 regardless of the screen buffering method chosen

 TCS_DI_UsrLst0: a longword that points to the first user copperlist: a
                 simple write in this field will not produce any effect
                 (the jump to the user copperlist is auto-coded inside the
                 main copperlist by InitDspl()). If you really need to
                 change the address of this copperlist, you'll have to do
                 it yourself using the MainCopperList structure

 TCS_DI_UsrLst1: a longword that points to the second user copperlist: a
                 simple write in this field will not produce any effect
                 (the jump to the user copperlist is auto-coded inside the
                 main copperlist by InitDspl()). If you really need to
                 change the address of this copperlist, you'll have to do
                 it yourself using the MainCopperList structure

 TCS_DI_CSAdr  : fundamental field: it always holds the address of the
                 screen buffer you can write/read pixels to/from.
                 *Always* use this value and forget about the many others
                 defined in the same structure!

 TCS_DI_CSWd   : unsigned word field that holds the chunky screen width
                 in bytes

 TCS_DI_CSHt   : unsigned word field that holds the chunky screen height
                 in pixels
```

1.10 4.3.6 MainCopperList structure

```
4.3.6   MainCopperList structure


This structure is used to gain "clean" access to the copperlists generated
by InitDspl() (if you feel "obliged" to put your hands on this part of
Copper code, make sure you know exactly what you're doing).

Copperlists are handled in this way: there is a "master copperlist" (the
one whose structure we're dealing with here - note that it always runs
using COP1LC, so other copperlists can't modify this register) which
performs some settings and calls all the other copperlists dedicated to
```

screen buffering (TCS_MCL_BufLst), Cross Playfield (TCS_MCL_FPfldLst), palette (TCS_MCL_PalLst) and user-defined settings (TCS_MCL_UsrLst0/1).

Since the few other fields are rather self-explaining (and, generally, you should not be interested much), I'll dwell upon only the most relevant of those just listed:

TCS_MCL_UsrLst0: this is a three-longword field used for three COPMOVEs that load the COP2LC register with TCS_DD_UsrCopLst0 and then start such copperlist by writing to COPJMP2. If no user copperlist is required, it is written with a COPMOVE to the strobe register COPJMP1 to continue with the main copperlist execution

TCS_MCL_UsrLst1: this is a three-longword field used for three COPMOVEs that load the COP2LC register with TCS_DD_UsrCopLst1 and then start such copperlist by writing to COPJMP2. If no user copperlist is required, it is written with a simple "COPWAIT forever" ($fffffffe)

1.11 4.3.7 ILBMInfo structure

4.3.7 ILBMInfo structure

This structure has few rather self-explaining fields and is the one you get after loading an ILBM:

TCS_II_UsrBufAdr : user-specified address of the chunky buffer for the raster data (0 if none specified)
TCS_II_GfxAdr : address of the chunky buffer for the raster data (same as TCS_II_UsrBufAdr, if this is not 0)
TCS_II_PalAdr : address of the original 256 entries, raw, 24-bit palette
TCS_II_PalDiff : degree of mismatch between the original palette and the mode indicated in TCS_II_RGBxMode (0 in case of exact match)
TCS_II_wd : picture's width in pixels
TCS_II_ht : picture's height in pixels
TCS_II_PlnsNo : picture's number of planes
TCS_II_RGBxMode : RGBx mode automatically selected by TCS_LdILBM()

1.12 4.4 Functions for Displays

4.4 Functions for Displays

The following functions allow you to create, modify, use, destroy all the displays you want (and your machine permits!):

```
TCS_InitDspl()
TCS_ShwDspl()
TCS_HideDspl()
TCS_FreeDspl()
TCS_SetRGBxMode()
TCS_C2PPass0()
TCS_CPUFRPass0()
TCS_CPUFRPass1()
TCS_CPUFRPass2()
TCS_BltFRPass0()
TCS_BltFRPassHndlr()
TCS_WtBltFRPass()
TCS_DubSwp()
TCS_TriSwp()
TCS_TriUpd()
TCS_WtTriSwp()
TCS_SttcSwp()
TCS_GetVdoBufsAdrs()
TCS_EquVdoBufs()
TCS_SetPlnsPos()
TCS_SetPlnsVPos()
TCS_EnbXPfld()
TCS_DsbXPfld()
TCS_EnbDXPfld()
TCS_DsbDXPfld()
TCS_SetFPfldOpct()
TCS_SetGfxCtxt()
```

1.13 TCS_InitDspl()

```
TCS_InitDspl()



INFO

 Reserves and initializes all the memory buffers needed for the bitplanes,
 copperlists and data structures required to create a display.


SYN

 DIAdr = TCS_InitDspl(DDAdr)

 d0.l                 a0.l


IN

 DDAdr   pointer to DisplayDeclaration structure of the desired display


OUT

 DIAdr   pointer to DisplayInfo structure (0=ERROR)
```

```
NOTE

 - after the call, you can find the address of the buffer to use as chunky
   screen in DIAdr.TCS_DI_CSAdr (it's *not* adviceable to use any other
   pointer that can be found in that structure)
 - DDAdr.ScrWd is always rounded to next multiple of 8 (if necessary)
 - error returned if (.x = DDAdr.x):
    a) .DsplX0 < TCS_DL_MinX0
    b) .DsplX1 > TCS_DL_MaxX1
    c) .DsplY0 < TCS_DL_MinY0
    d) .DsplY1 > TCS_DL_MaxY1
    e) .DsplX1-.DsplX0 < TCS_DL_MinWd
    f) .DsplY1-.DsplY0 < TCS_DL_MinHt
    g) - [HalfRes] .ScrWd < DsplWd/8
       - [FullRes] .ScrWd < DsplWd/4
       (DsplWd = (.DsplX1-.DsplX0) rounded to the next multiple of 64)
    h) .ScrHt < .DsplY1-.DsplY0
    i) not enough memory
 - use TCS_FreeDspl() to deallocate
 - uses exec.library's AllocMem(), thus it can't be called from interrupts
```

1.14 TCS_ShwDspl()

```
TCS_ShwDspl()



INFO

 Shows on the monitor a display.


SYN

 success = TCS_ShwDspl(DIAdr)

 ccr                   a0.l


IN

 DIAdr     display DisplayInfo structure pointer


OUT

 success   ne = display started successfully
           eq = ERROR


NOTE

 - error returned if:
```

```
    a) the display was already shown
    b) DIAdr relative to a display initialized as "front-playfield-only"
       and the other playfield was hidden
    c) DIAdr is not a valid DI structure pointer
 - make sure the AmigaOS is OFF and you have control over the hardware!
 - if there is another display being shown, it is hidden first
 - Cross Playfield mode is restored if other playfield already shown
 - activates all the needed DMA channels (bitplanes, Copper and, in case
   of Blitter-assisted FullRes conversion, Blitter)
 - it always switches to PAL
 - use TCS_HideDspl() to hide the display
```

1.15 TCS_HideDspl()

```
TCS_HideDspl()



INFO

 Hides the desired display.


SYN

 success = TCS_HideDspl(DIAdr, NewCopLst)

 ccr                    a0.l   a1.l


IN

 DIAdr       display DisplayInfo structure pointer
 NewCopLst   address of the copperlist to be executed after hiding the
             display (0 = blacken the monitor screen)


OUT

 success     ne = display hidden successfully
             eq = ERROR


NOTE

 - error returned if:
    a) display was already hidden
    b) DIAdr is not a valid DI structure pointer
 - if NewCopLst=0 then Copper and bitplanes DMAs are turned OFF
 - use TCS_ShwDspl() to make the display visible again
 - in Cross Playfield mode the other playfield remains visible unless decla-
   red "front-playfield-only" (in which case NewCopLst is used)
```

1.16 TCS_FreeDspl()

```
TCS_FreeDspl()



INFO

 Frees all the resources allocated for a display.


SYN

 success = TCS_FreeDspl(DIAdr)

 ccr                    a0.l


IN

 DIAdr     display DisplayInfo structure pointer


OUT

 success   ne = resources released successfully
           eq = ERROR


NOTE

 - error returned if:
    a) the display is currently being shown (hide it, first)
    b) Cross Playfield mode is active (disable it, first)
    c) DIAdr is not a valid DI structure pointer
 - uses exec.library's FreeMem(), thus it can't be called from interrupts
 - you *must* wait for Blitter-assisted FullRes conversion to end (if
   active) before calling this function! The reason is that while this
   function releases all the allocated buffers, it could be that a long
   blit is still being performed on one of them!
```

1.17 TCS_SetRGBxMode()

```
TCS_SetRGBxMode()



INFO

 Sets the RGBx mode and palette of a display.


SYN
```

```
 TCS_SetRGBxMode(DIAdr, RGBxID, brtns)

                 a0.l   d0.b    d1.w


IN

 DIAdr    display DisplayInfo structure pointer
 RGBxID   one of the TCS_VM_RGBx values
 brtns    brightness degree: [0 ... 256] = [min ... max]


NOTE

 - since there is a single palette copperlist, its changes are immedia-
   tely visible despite buffering
 - brightness control permits to achieve easily simple fade in/out effects
 - if the display is in Cross Playfield mode, the new palette for both
   playfields is re-calculated, too
 - relatively expensive
```

1.18 TCS_C2PPass0()

```
TCS_C2PPass0()



INFO

 Executes the conversion chunky screen -> display logical planes.
 The user does not have to bother about whether the conversion is actually
 needed or not, and about which TCS_#?Pass0() function to call.


SYN

 TCS_C2PPass0(DIAdr)

              a0.l


IN

 DIAdr   display DisplayInfo structure pointer


NOTE

 - if Blitter-assisted conversion is required, the handler must be
   properly set by the user anyway
 - nothing is done if the screen width and height don't match the display
   area ones: ScrWd = (DsplX1-DsplX0)/4; ScrHt = DsplY1-DsplY0 (the iden-
   tifiers belong to the DisplayDeclaration structure)
 - see also TCS_CPUFRPass0(), TCS_CPUFRPass1(), TCS_CPUFRPass2() and
```

TCS_BltFRPass0()

1.19 TCS_CPUFRPass0()

```
TCS_CPUFRPass0()



INFO

 This function is useful only if a FullRes video mode has been activated
 as it executes the conversion chunky screen -> display logical planes.


SYN

 TCS_CPUFRPass0(DIAdr)

                a0.l


IN

 DIAdr   display DisplayInfo structure pointer


NOTE

 - *NEVER* call if DI's video mode is not FullRes!!!
 - this routine has to deal with lotsa data and writes to _slow_ CHIP ram,
   so don't expect to be lightning fast!
   Anyway, I can't really see how to make it faster (at least on my 030 -
   I tried thousands of different implementations!!!)
 - call *only* when the screen width and height match exactly the display
   area ones: ScrWd = (DsplX1-DsplX0)/4; ScrHt = DsplY1-DsplY0 (the iden-
   tifiers belong to the DisplayDeclaration structure)
 - see also TCS_CPUFRPass1(), TCS_CPUFRPass2() and TCS_BltFRPass0()
```

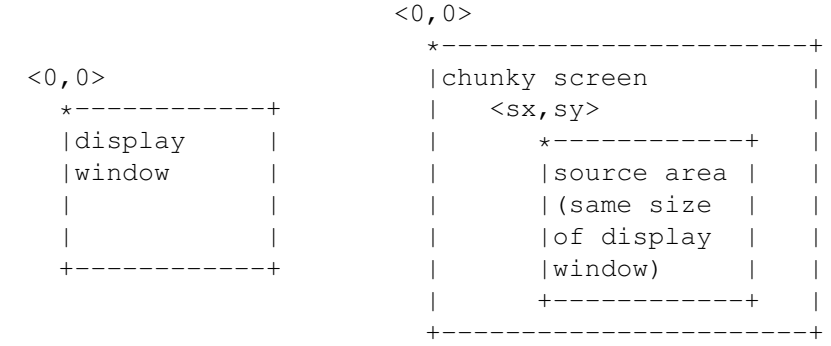
1.20 TCS_CPUFRPass1()

```
TCS_CPUFRPass1()



INFO

 This function is useful only if a FullRes video mode has been requested
 as it executes the conversion chunky screen -> display logical planes.


SYN
```

```
TCS_CPUFRPass1(DIAdr, sx,  sy)

               a0.l   d0.w d1.w
```

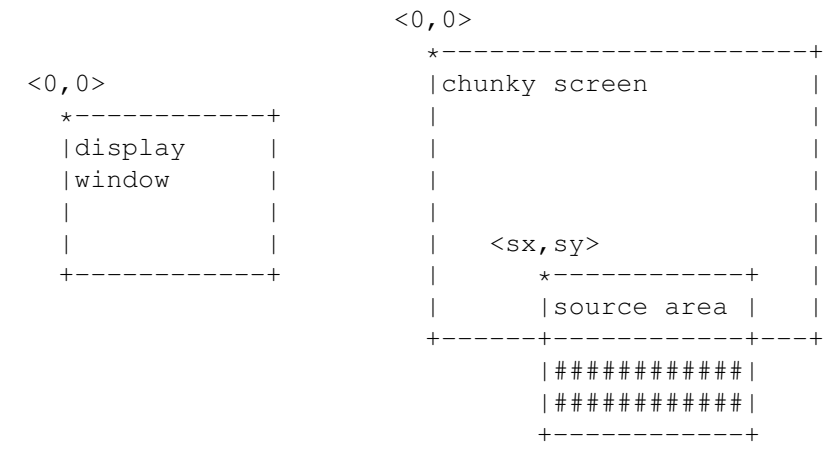
IN

```
DIAdr   display DisplayInfo structure pointer
sx,sy   coordinates of top-left pixel of area to convert referring to
        chunky screen's coordinates system (see figure)
```



NOTE

```
- it can be used to easily scroll screens larger than the display
- *NEVER* call if DI's video mode is not FullRes!!!
- best performance when sx is multiple of 4
- this routine has to deal with lotsa data and writes to _slow_ CHIP ram,
  so don't expect to be lightning fast!
  Anyway, I can't really see how to make it faster (at least on my 030 -
  I tried thousands of different implementations!!!)
- you must choose <sx,sy> carefully:
```




```
   pixels marked with a '#' will be shown on the display even if they
   don't belong to the screen

- see also TCS_CPUFRPass0(), TCS_CPUFRPass2() and TCS_BltFRPass0()
```

1.21 TCS_CPUFRPass2()

```
TCS_CPUFRPass2()
```

```
INFO
```

```
 This function is useful only if a FullRes video mode has been requested
 as it executes the conversion chunky screen -> display logical planes,
 giving the possibility of choosing the area of the screen to convert:
 this can give a significant speedup when only a part of the screen needs
 to be updated.
```

```
SYN
```

```
 TCS_CPUFRPass2(DIAdr, sx0, sy0, sx1, sy1, dx,  dy)

                a0.l   d0.w d1.w d2.w d3.w d4.w d5.w
```

```
IN
```

```
 DIAdr     display DisplayInfo structure pointer
 sx0,sy0   coordinates of top-left and bottom-right pixel of area to
 sx1,sy1   convert referring to chunky screen's coordinates system
 dx,dy     coordinates of top-left pixel of the destination area referring
           to display window's coordinates system
```


```
NOTE
```

```
 - it can be used to easily scroll screens larger than the display
 - *NEVER* call if DI's video mode is not FullRes!!!
 - if necessary, the area width (sx1-sx0+1) is rounded to the next
   multiple of 16
 - best performance when sx0 and dx are multiple of 4
 - dx is always automatically evened
 - this routine has to deal with lotsa data and writes to _slow_ CHIP ram,
```

```
   so don't expect to be lightning fast!
   Anyway, I can't really see how to make it faster (at least on my 030 -
   I tried thousands of different implementations!!!)
 - be extremely cautios when choosing the input values; the following
   situatuation causes writes to CHIP ram *not* allocated:

    <0,0>                                 <0,0>
      *-----------------------------+       *----------------------------+
      |display window               |       |chunky screen               |
      |                             |       |                            |
      |                             |       |<sx0,sy0>                   |
      |                             |       |   *------------------+     |
      |          <dx,dy>            |       |   |area to convert   |     |
      |             *---------------+---+   |   |                  |     |
      +---+         |area to convert###|   |   |                  |     |
      |###|         |               |###|   |   |                  |     |
      |###|         |               |###|   |   |                  |     |
      |###|         |               |@@@|   |   |                  |     |
      +---+---------+---------------+@@@|   |   +------------------*     |
                    |@@@@@@@@@@@@@@@@@@@|   |                  <sx1,sy1>|
                    +-------------------+   +----------------------------+

    #: these pixels "wrap around" the display and reappear on the other
       side as shown
    @: these pixels are written in CHIP ram locations not allocated
       for the display bitplanes

 - see also TCS_CPUFRPass0(), TCS_CPUFRPass1() and TCS_BltFRPass0()
```

1.22 TCS_BltFRPass0()

```
TCS_BltFRPass0()



INFO

 Executes the first part of the conversion as in TCS_CPUFRPass0() (up-
 dates VdoPln1 only).


SYN

 TCS_BltFRPass0(DIAdr)

                a0.l


IN

 DIAdr   display DisplayInfo structure pointer


NOTE
```

- *NEVER* call if DI's video mode is not FullRes and Blitter-assisted FullRes conversion has not been activated!!!
- the job is completed by TCS_BltFRPassHndlr()
- the Blitter must not be currently used by any other program: if multitasking is enabled, execute an OwnBlit() before; note that since the Blitter will also be used by TCS_BltFRPassHndlr() you can release it with DisownBlit() only after waiting for it to finish; same goes if you need the Blitter yourself: wait before accessing its registers!
- will wait for TCS_BltFRPassHndlr() to finish in case another conversion has already been started but has not terminated yet
- designed for Amigas equipped with FAST ram in mind (to let the CPU free of working in parallel), so don't use it on unxepanded machines! Anyway, use it carefully: the Blitter takes ages to update the whole screen, so make sure the CPU doesn't fall in idle state waiting for the Blitter to finish
- call only when the chunky screen width and height match exactly the display area's ones: ScrWd = (DsplX1-DsplX0)/4; ScrHt = DsplY1-DsplY0 (identifiers belong to the DisplayDeclaration structure)
- if no buffering is used the screen looks very jerky because this function works on planar basis: the Blitter first converts the whole VdoPln1 and then VdoPln0 (on the contrary, TCS_CPUFRPass?() converts both planes long by long, so no jerks are visible even without buffering)
- sets INTENA.INTEN (it should have been already set, anyway)
- see also TCS_CPUFRPass0(), TCS_CPUFRPass1() and TCS_CPUFRPass2()

1.23 TCS_BltFRPassHndlr()

```
TCS_BltFRPassHndlr()
```

```
INFO

 Executes the second part of the current FullRes conversion (updates
 VdoPln0) when done with Blitter's assistance.
```

```
SYN

 TCS_BltFRPassHndlr()
```

```
NOTE

 - *NEVER* call if DI's video mode is not FullRes and Blitter-assisted
   FullRes conversion has not been activated!!!
 - it must be called from inside a level3/BLIT interrupt handler (all the
   job will be done automatically); for example:

    Lev3Hndr   movem.l      d0-d1/d7/a0-a1/a6,-(sp)
               move.w       $dff01e,d7              ;get INTREQR
               btst.l       #6,d7
               beq.s        .exit                   ;if not BLIT...

               movea.l      TCSBase,a6
```

```
                jsr          (_LVOTCS_BltFRPassHndlr,a6)

    .exit       movem.l      (sp)+,d0-d1/d7/a0-a1/a6
                move.w       #$40,$dff09c             ;clear INTREQ.BLIT
                rte
```

(of course this code can be extended to handle all other interrupts)

- for considerations about Blitter-sharing with other tasks have a look at the notes of TCS_BltFRPass0()
- for considerations about possible interactions with screen buffering, have a look at the notes of TCS_DubSwp() and TCS_TriSwp()

1.24 TCS_WtBltFRPass()

```
TCS_WtBltFRPass()
```

```
INFO

 Waits for th current Blitter-assisted FullRes conversion to end.
```

```
SYN

 TCS_WtBltFRPass(DIAdr)

                 a0.l
```

```
IN

 DIAdr   pointer to a DisplayInfo structure or 0 to wait for the conver-
         sion completion on a specific or any display, respectively.
```

```
NOTE
```

- if you pass a non-zero DIAdr, the function will wait only if the current conversion is relative to the specified display
- make sure TCS_BltFRPassHndlr() can still be called (i.e.: the interrupt handler from which it is called is still active), otherwise a deadlock will surely happen!
- before using the Blitter yourself, you *must* call this function (using graphics.library's WaitBlit() or polling DMACONR.BBUSY is not the same nor enough!)

1.25 TCS_DubSwp()

```
TCS_DubSwp()
```

```
INFO

 Executes the screen swapping to make double buffering take place: the
 logical buffer which was in the background will be displayed after the
 Copper reloads the BPLxPT registers (during the first VBL after the call)
 and the physical buffer that was displayed until that moment will become
 available for your gfx operations.


SYN

 NewChnkScr = TCS_DubSwp(DIAdr)

 d0.l                    a0.l


IN

 DIAdr        display DisplayInfo structure pointer


OUT

 NewChnkScr   address of the chunky screen that can be used for background
              rendering after the call


NOTE

 - it makes no sense to call this function if double buffering has not
   been activated with TCS_VMf_DubBuf (TCS_VMf_TriBuf is useless either)
 - call immediately before or during a vertical blanking to have that the
   new physical copperlist is promptly used: otherwise, it could happen
   that you start rendering on the new logic buffer which is still being
   displayed, with the ensuing on-screen jerkings!
 - in case of Blitter-assisted FullRes conversion, the buffers can be
   swapped only after the Blitter is finished, thus this function could
   put itself in active wait for that event (TCS_BltFRPassHndlr() must
   be called from an enabled interrupt to be able to break that wait loop,
   otherwise a deadlock will surely happen!).
   Also consider that the wait probably will make everything go "out of
   sync", i.e. the swap could occur in the middle of a frame (without any
   visible change until the next VBL) despite you waited for the VBL be-
   fore calling the function; in such case a simple solution would be wai-
   ting or performing non-graphic operations till the next VBL
```

1.26 TCS_TriSwp()

```
TCS_TriSwp()


INFO
```

This function is the homologous of TCS_DubSwp(): to keep the triple buffering mechanism going it checks whether a screen buffer has been completely rendered and, in such case, makes the new physical buffer displayable starting from the first VBL after or during the call.

SYN

TCS_TriSwp(DIAdr)

a0.l

IN

DIAdr display DisplayInfo structure pointer

NOTE

- it makes no sense to call this function if triple buffering has not been activated with TCS_VMf_TriBuf (TCS_VMf_DubBuf is useless either)
- it makes only half of the job required for triple buffering: the rest is done by TCS_TriUpd()
- must be called from inside an enabled interrupt handler (preferably every VBL with a level3/VERTB interrupt - TCS_TriUpd() could be waiting in a tight loop)
- in case of Blitter-assisted FullRes conversion, the swapping could be delayed until the Blitter has finished its job (more precisely, to the next VBL after Blitter's rendering completion) - let's see why: normally, the sequence of the operations would be:

```
*************************************************************************
* user program main loop

 loop     <...>
          <program screen rendering>
          jsr      TCS_BltFRPass0()     ;FullRes conversion (1st)
          jsr      TCS_TriUpd()         ;triple buffering
          bra      loop

*************************************************************************
* VERTB interrupt handler


 .start   <...>
          move.w   $dff01e,d7            ;get INTREQR
          <...>

          btst.l   #6,d7
          beq.s    .VERTB                ;if not BLIT...
          jsr      TCS_BltFRPassHndlr() ;FullRes conversion (2nd)
          move.w   #$40,$dff01e          ;clear INTREQ.BLIT
          <...>

 .VERTB   btst.l   #5,d7
```

```
            beq.s    .there                ;if not VERTB...
            jsr      TCS_TriSwp()          ;triple buffering
            move.w   #$20,$dff01e          ;clear INTREQ.VERTB
     .there <...>
            rte
```

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

    to give the processor as much freedom as possible, TCS_TriUpd() works
    in a non-blocking way, in the sense that it only stops if no buffer is
    available; at the same time, the Blitter proceeds with its job concur-
    rently, out of processor's control: this means that there is no gua-
    rantee that when trying to perform the swapping inside the handler it
    has already finished (it could be in the middle of both the first and
    the second part); thus, TCS_TriSwp() is forced to ignore the request,
    as there is no buffer actually ready to be displayed (although, from
    the program's perpective, there is - as signalled with TCS_TriUpd()).
    A last note on when we clear INTREQ: the BLIT bit is cleared immedia-
    tely after the call to TCS_BltFRPassHndlr() because in case of small
    blits, the Blitter could request another interrupt before exiting this
    handler, so we must pay attention not to trash this signal (yet, since
    TCS_BltFRPassHndlr() returns just after starting the Blitter, it is
    a very unlikely event; anyway, be extremely cautious and DO NOT set
    Blitter nasty bit - DMACON.BLTPRI), which could prevent the micro from
    writing to INTREQ, especially on CHIPram-only machines!)

 - must *never* be interrupted by TCS_TriUpd() or TCS_BltFRPass0()!
```

1.27 TCS_TriUpd()

```
TCS_TriUpd()



INFO

 This is one of the functions needed to operate the triple buffering: in
 particular this routine is used to acknowledge that a screen rendering
 has been finished (in background); whenever you get to such point, just
 call this function, get hold of its return value and continue without
 giving a damn to all the rest: this function and TCS_TriSwp() will do
 everything for you.


SYN

 NewChnkScr = TCS_TriUpd(DIAdr)

 d0.l                    a0.l


IN

 DIAdr        display DisplayInfo structure pointer
```

```
OUT

 NewChnkScr   address of the chunky screen that can be used for background
              rendering after the call


NOTE

 - *NEVER* call it if triple buffering is not active or TCS_TriSwp()
   cannot interrupt its execution (i.e. TCS_TriSwp() must be called from
   inside an enabled interrupt): otherwise it would get stuck in an infi-
   nite wait loop!
 - after getting NewChnkScr you can immediately start to draw graphics
   to the buffer at this address
```

1.28 TCS_WtTriSwp()

```
TCS_WtTriSwp()



INFO

 Waits for a triple buffering swap to be performed by TCS_TriSwp().


SYN

 TCS_WtTriSwp(DIAdr)

              a0.l


IN

 DIAdr   display DisplayInfo structure pointer


NOTE

 - useful for particular synchronization needs
 - *NEVER* call it if triple buffering is not active or TCS_TriSwp()
   cannot interrupt its execution (i.e. TCS_TriSwp() must be called from
   inside an enabled interrupt): otherwise it would get stuck in an infi-
   nite wait loop!
 - the function exits immediately if no swap will ever be done (this
   happens when the next step should be done by TCS_TriUpd() and not by
   TCS_TriSwp()
```

1.29 TCS_SttcSwp()

```
TCS_SttcSwp()


INFO

 Executes a video buffers swap (as if TCS_DubSwp() or TCS_TriSwp() or
 TCS_TriUpd() had been called, but without affecting the current copper-
 list) when the buffering mechanism is enabled but not running.


SYN

 NewChnkScr = TCS_SttcSwp(DIAdr)

 d0.l                     a0.l


IN

 DIAdr        display DisplayInfo structure pointer


OUT

 NewChnkScr   address of the new logical video buffer


NOTE

 - especially useful to update the video buffers statically, out of the
   buffering cycle (see also TCS_GetVdoBufsAdrs() and TCS_EquVdoBufs())
 - don't call if swapping is currently running
```

1.30 TCS_GetVdoBufsAdrs()

```
TCS_GetVdoBufsAdrs()


INFO

 Returns the addresses of all the buffers reserved for video buffering.


SYN

 TCS_GetVdoBufsAdrs(DIAdr, DstLstAdr)

                    a0.l   a1.l


IN
```

```
 DIAdr       display DisplayInfo structure pointer
 DstLstAdr   address of a 16 bytes long vector, which will be filled with
             4 longwords representing the addresses of:

              - the current logical buffer
              - the current available buffer
              - the current physical buffer
              - the current ready buffer

              (in this order)


NOTE

 - a1.l is guaranteed to be left unmodified
 - especially useful to update the video buffers statically, out of the
   buffering cycle (see also TCS_SttcSwp() and TCS_EquVdoBufs())
 - double buffering active: available=logical, ready=physical
 - triple buffering active: available=logical and ready<>physical or
   available<>logical and ready=physical (case at initialization)
 - double/triple buffering not active: all addresses are equal
 - FullRes ON: all addresses are equal independently from the buffering
   mode selected
 - don't call if buffers swapping is currently running
```

1.31 TCS_EquVdoBufs()

```
TCS_EquVdoBufs()



INFO

 Equals the content of all the video buffers of a display to the current
 logical video buffer's.


SYN

 TCS_EquVdoBufs(DIAdr)

                a0.l


IN

 DIAdr       display DisplayInfo structure pointer


NOTE

 - especially useful to update the video buffers statically, out of the
   buffering cycle (see also TCS_SttcSwp() and TCS_GetVdoBufsAdrs())
 - in FullRes, you should update the logical video buffer by calling one
   of the TCS_???FRPass?() functions before calling this one
```

```
 - no operation is performed if the display is not buffered
```

1.32 TCS_SetPlnsPos()

```
TCS_SetPlnsPos()


INFO

 Sets the position of the bitplanes of a HalfRes display.


SYN

 TCS_SetPlnsPos(DIAdr, XPos, YPos)

                a0.l   d0.w  d1.w


IN

 DIAdr   display DisplayInfo structure pointer
 XPos    unsigned x offset in SHRES pixels from top-left corner
 YPos    unsigned y offset in pixels from top-left corner


NOTE

 - this function can be used to scroll a screen larger than the display
   area (no check is made, though - it's not dangerous, it would result
   just in an on-screen memory dump ;) )
 - don't call if horizontal scrolling has not been activated!
 - this routine is relatively slow if ChqrMode is ON, due to the fact
   that the copperlist which implements it is quite long and thus many
   writes to CHIP ram must be done (besides, as a consequence, jerkings
   are very likely to show up if double/triple buffering is not active
   and, anyway, if called while or just before the Copper executes that
   part of the copperlist - to avoid this, call just after the VBL)
 - the display DIAdr refers to needs *not* necessarily to be active
 - it affects only the current logic copperlist
 - in Cross Playfield mode it's up to you to keep the same horizontal
   position of both playfields!
 - to obtain the same effect in FullRes simply change the input values of
   TCS_CPUFRPass1() or TCS_CPUFRPass2()
```

1.33 TCS_SetPlnsVPos()

```
TCS_SetPlnsVPos()
```

```
INFO

 Sets the vertical position of the bitplanes of a HalfRes display.


SYN

 TCS_SetPlnsVPos(DIAdr, YPos)

                 a0.l   d1.w


IN

 DIAdr   display DisplayInfo structure pointer
 YPos    unsigned y offset in pixels from top-left corner


NOTE

 - this function can be used to scroll a screen larger than the display
   area (no check is made, though - it's not dangerous, it would result
   just in an on-screen memory dump ;) )
 - don't use if horizontal scroll is ON (use TCS_SetPlnsPos(), instead)
 - to obtain the same effect in FullRes simply change the input values of
   TCS_CPUFRPass1() or TCS_CPUFRPass2()
 - the display DIAdr refers to needs *not* necessarily to be active
 - this function is fast in any video mode and thus can be always called
   without problems (unlike TCS_SetPlnsPos())
 - it affects only the current logic copperlist
```

1.34 TCS_EnbXPfld()

```
TCS_EnbXPfld()



INFO

 Enables the Cross Playfield mode by superimposing a screen (from a pre-
 viously initialized display - "front playfield") to another one (from
 another display - "back playfield").


SYN

 success = TCS_EnbXPfld(BPfld, FPfld)

 ccr                    a0.l   a1.l


IN

 BPfld     back playfield DisplayInfo structure pointer
 FPfld     front playfield DisplayInfo structure pointer
```

```
OUT

 success   ne = mode enabled successfully
           eq = ERROR


NOTE

 - error returned if:
    a) the displays are uncompatible
    b) BPfld's display is declared "front-playfield-only"
 - in HalfRes, the playfields positions are automatically reset to <0,0>
 - front playfield opacity set to 256 and Dual mode disabled by default
 - the playfields are _not_ automatically shown, but instead you must call
   TCS_ShwDspl() anytime after enabling the mode
```

1.35 TCS_DsbXPfld()

```
TCS_DsbXPfld()



INFO

 Deactivates the Cross Playfield mode.


SYN

 success = TCS_DsbXPfld(DIAdr)

 ccr                    a0.l


IN

 DIAdr     DisplayInfo structure of any playfield


OUT

 success   ne = mode disabled successfully
           eq = ERROR


NOTE

 - error returned if:
    a) Cross Playfield not enabled
    b) playfield relative to DIAdr still shown (hide it, first)
```

1.36 TCS_EnbDXPfld()

```
TCS_EnbDXPfld()



INFO

 Enables and sets the Dual modality of Cross Playfield mode to simulate
 a real Dual Playfield.


SYN

 success = TCS_EnbDXPfld(DIAdr, TrnspCol)

 ccr                      a0.l   d0.b


IN

 DIAdr      DisplayInfo structure of any playfield
 TrnspCol   front playfield RGBx color to treat as transparent regardless
            of the playfield's opacity


OUT

 success   ne = mode enabled successfully
           eq = ERROR


NOTE

 - error returned if:
    a) DIAdr doesn't belong to a display used for Cross Playfield
    b) HalfRes and the back playfield doesn't have a MskPln
 - activating this mode reduces the front playfield available colors to
   81: apart from the one specified, other 174 have some of their compo-
   nents equal to those of col, so also those components are treated as
   transparent (and thus those 174 colors don't look as they should - to
   find out which ones, use TCS_FlgDXPfldCols() or TCS_SvIFFRGBxPal())
```

1.37 TCS_DsbDXPfld()

```
TCS_DsbDXPfld()



INFO

 Disables the Dual modality of Cross Playfield mode.
```

```
SYN

 success = TCS_DsbDXPfld(DIAdr)

 ccr                    a0.l


IN

 DIAdr     DisplayInfo structure of any playfield


OUT

 success   ne = mode disabled successfully
           eq = ERROR


NOTE

 - error returned if:
    a) DIAdr doesn't belong to a display used for Cross Playfield
    b) the Dual mode was not enabled
```

1.38 TCS_SetFPfldOpct()

```
TCS_SetFPfldOpct()



INFO

 Sets the opacity of the front playfield (when Cross Playfield mode is
 active) in order to make the back playfield more/less visible through
 the pixels of front playfield.


SYN

 TCS_SetFPfldOpct(DIAdr, opct [, TrnspCol])

                  a0.l   d0.w    d1.b


IN

 DIAdr        DisplayInfo structure of any playfield
 opct         opacity degree of front playfield, belonging to [0...256]
              ( = [totally transparent ... totally opaque])
 [TrnspCol]   transparent RGBx color in Dual Cross Playfield mode


NOTE

 - no action is performed if DIAdr doesn't belong to a display used for
```

```
   Cross Playfield
 - this function makes producing cross-fading effects extra-easy...
 - ... but it's a bit expensive (if MskPln is ON, almost twice as slow)
```

1.39 TCS_SetGfxCtxt()

```
TCS_SetGfxCtxt()



INFO

 Sets the Graphic Context of a display.


SYN

 TCS_SetGfxCtxt(DIAdr, GCAdr)

                a0.l   a1.l


IN

 DIAdr   display DisplayInfo structure pointer
 GCAdr   pointer to the desired GraphicContext structure
```

1.40 4.5 Functions for Color/Palette Control

```
4.5   Functions for Color/Palette Control



These functions provide some comfortable ways to handle RGB/RGBx data:

  TCS_GetRGBBrtns()
  TCS_GetRGBxTheoBrtns()
  TCS_GetRGBxActlBrtns()
  TCS_RGBToRGBx()
  TCS_RGBxToRGB()
  TCS_MkRGBxCnvTab()
  TCS_RGBxPicToTrueCol()
  TCS_TrueColPicToRGBx()
  TCS_CLUTPicToTrueCol()
  TCS_CLUTPicToRGBx()
  TCS_RmpRGBxPic()
  TCS_FlgDXPfldCols()
  TCS_SvIFFRGBxPal()
```

1.41 TCS_GetRGBBrtns()

```
TCS_GetRGBBrtns()


INFO

 Returns the brightness of a TrueColor 24-bit pixel.

SYN

 brtns = TCS_GetRGBBrtns(RGBPxl)

 d0.w                    d0.l

IN

 RGBPxl   pixel in $00RrGgBb format

OUT

 brtns    brightness in the [0...255] range
```

1.42 TCS_GetRGBxTheoBrtns()

```
TCS_GetRGBxTheoBrtns()


INFO

 Returns the theorical brightness of an RGBx pixel.

SYN

 brtns = TCS_GetRGBxTheoBrtns(RGBxPxl, RGBxID)

 d0.w                          d0.b     d1.b

IN

 RGBxPxl   pixel in any the RGBx format specified
 RGBxID    one of the TCS_VM_RGBx values

OUT

 brtns     brightness in the [0...255] range
```

```
NOTE

 - since RGBx modes can't fully exploit all the available brightness (see
   the RGB <-> RGBx issue for details), brtns does not reflect the actual
   value of brightness as it appears on the screen (if you needed it, use
   TCS_GetRGBxActlBrtns() instead)
```

1.43 TCS_GetRGBxActlBrtns()

```
TCS_GetRGBxActlBrtns()



INFO

 Returns the actual brightness of an RGBx pixel.


SYN

 brtns = TCS_GetRGBxActlBrtns(RGBxPxl, RGBxID)

 d0.w                          d0.b     d1.b


IN

 RGBxPxl   pixel in any the RGBx format specified
 RGBxID    one of the TCS_VM_RGBx values


OUT

 brtns     brightness in the [0...127] range


NOTE

 - since RGBx modes can't fully exploit all the available brightness (see
   the RGB <-> RGBx issue for details), actually brtns is always less
   than 128
 - see also TCS_GetRGBxTheoBrtns()
```

1.44 TCS_RGBToRGBx()

```
TCS_RGBToRGBx()



INFO

 Converts a normal TrueColor 24-bit pixel to the corresponding RGBx one.
```

```
SYN

 RGBxPxl = TCS_RGBToRGBx(RGBPxl, RGBxMode)

 d0.b                    d0.l    d1.b


IN

 RGBPxl     TrueColor 24-bit ($00RrGgBb) source value
 RGBxMode   desired RGBx mode (any TCS_VM_RGBx value)


OUT

 RGBxPxl    RGBPxl encoded in the selected RGBx (8-bit)


NOTE

 - the passage from 24 to 8 bits produces an unavoidable loss of quality
 - the conversion is quite heavy for intensive real-time calculations (for
   picture remapping try figure something else (for example, hash tables
   built using this function))
```

1.45 TCS_RGBxToRGB()

```
TCS_RGBxToRGB()



INFO

 Converts a pixel in any RGBx format to TrueColor 24-bit.


SYN

 RGBPxl = TCS_RGBxToRGB(RGBxPxl, RGBxMode)

 d0.l                    d0.b     d1.b


IN

 RGBxPxl    pixel in any RGBx format
 RGBxMode   RGBx format of RGBxPxl (any TCS_VM_RGBx value)


OUT

 RGBPxl     pixel in TrueColor 24-bit format ($00RrGgBb)
```

```
NOTE

 - even if the destination is 24-bit, there's no quality improvement
 - for intensive real-time conversion, I suggest to use a look-up table
   rather than calling this function each and every time (it's slow!)
```

1.46 TCS_MkRGBxCnvTab()

```
TCS_MkRGBxCnvTab()



INFO

 Creates a look-up table for RGBx -> RGB conversion: the item #I is the
 TrueColor 24-bit value corresponding to the RGBx 8-bit value I.


SYN

 TCS_MkRGBxCnvTab(TabAdr, RGBxMode)

                  a0.l    d0.b


IN

 TabAdr     address of the buffer to fill with the data;
            each item written will be a longword containing a 24-bit RGB
            value in the format: $00RrGgBb
 RGBxMode   desired RGBx mode (any TCS_VM_RGBx value)


NOTE

 - the buffer must be at least 4*256 bytes long!
 - to convert the RGBx value V, just read the longword at the address:
   TabAdr+V*4
```

1.47 TCS_RGBxPicToTrueCol()

```
TCS_RGBxPicToTrueCol()



INFO

 Converts a raw chunky picture in RGBx format to a raw chunky TrueColor
 24-bit picture.
```

```
SYN

 TCS_RGBxPicToTrueCol(PicAdr, DstBufAdr, PicRGBxMode, PicWd, PicHt)

 d0.l                 a0.l    a1.l       d0.b         d1.w   d2.w


IN

 PicAdr         pointer to picture's RGBx data
 DstBufAdr      pointer to buffer for calculated TrueColor 24-bit data
 PicRGBxMode    picture's RGBx mode (any TCS_VM_RGBx value)
 PicWd          picture's width in pixels
 PicHt          picture's height in pixels


NOTE

 - make sure that PicWd & PicHt don't exceed the actual dimensions of the
   picture (otherwise meaningless data will be converted, too) or the di-
   mension of the destination buffer (at least PicWd*PicHt*3 bytes -
   - otherwise dangerous writes to RAM will be done)
 - just simple pixel-by-pixel-based remapping
 - source and destination buffers cannot overlap
```

1.48 TCS_TrueColPicToRGBx()

```
TCS_TrueColPicToRGBx()



INFO

 Converts a raw chunky TrueColor 24-bit picture to a raw chunky picture in
 RGBx format.


SYN

 TCS_TrueColPicToRGBx(PicAdr, DstBufAdr, DstRGBxMode, PicWd, PicHt)

                      a0.l    a1.l       d0.b         d1.w   d2.w


IN

 PicAdr         pointer to picture's TrueColor 24-bit data
 DstBufAdr      pointer to buffer for calculated RGBx data
 DstRGBxMode    desired RGBx mode (any TCS_VM_RGBx value)
 PicWd          picture's width in pixels
 PicHt          picture's height in pixels


NOTE
```

```
 - make sure that PicWd & PicHt don't exceed the actual dimensions of the
   picture (otherwise meaningless data will be converted, too) or the di-
   mension of the destination buffer (at least PicWd*PicHt bytes - other-
   wise dangerous writes to RAM will be done)
 - source and destination buffers can overlap as long as the destination
   buffer comes before the source one (DstBufAdr<=PicAdr)
 - just simple pixel-by-pixel-based remapping
```

1.49 TCS_CLUTPicToTrueCol()

```
TCS_CLUTPicToTrueCol()



INFO

 Converts a color indexed raw chunky 8-bit picture to a raw chunky
 TrueColor 24-bit picture.


SYN

 TCS_CLUTPicToTrueCol(PicAdr, DstBufAdr, CLUTAdr, PicWd, PicHt)

                      a0.l    a1.l       a2.l     d0.w   d1.w


IN

 PicAdr         pointer to picture's 8-bit data
 DstBufAdr      pointer to buffer for calculated TrueColor 24-bit data
 CLUTAdr        pointer to picture's 24-bit color look-up table
 PicWd          picture's width in pixels
 PicHt          picture's height in pixels


NOTE

 - make sure that PicWd & PicHt don't exceed the actual dimensions of the
   picture (otherwise meaningless data will be converted, too) or the di-
   mension of the destination buffer (at least PicWd*PicHt bytes - other-
   wise dangerous writes to RAM will be done)
 - source and destination buffers cannot overlap
 - just simple pixel-by-pixel-based remapping
```

1.50 TCS_CLUTPicToRGBx()

```
TCS_CLUTPicToRGBx()



INFO
```

```
 Converts a color indexed raw chunky 8-bit picture to a raw chunky picture
 in RGBx format.


SYN

 TCS_CLUTPicToRGBx(PicAdr, DstBufAdr, CLUTAdr, DstRGBxMode, PicWd, PicHt)

                   a0.l    a1.l       a2.l     d0.b         d1.w   d2.w


IN

 PicAdr         pointer to picture's 8-bit data
 DstBufAdr      pointer to buffer for calculated RGBx data
 CLUTAdr        pointer to picture's 24-bit color look-up table
 DstRGBxMode    desired RGBx mode (any TCS_VM_RGBx value)
 PicWd          picture's width in pixels
 PicHt          picture's height in pixels


NOTE

 - make sure that PicWd & PicHt don't exceed the actual dimensions of the
   picture (otherwise meaningless data will be converted, too) or the di-
   mension of the destination buffer (at least PicWd*PicHt bytes - other-
   wise dangerous writes to RAM will be done)
 - source and destination buffers can overlap as long as the destination
   buffer comes before the source one (DstBufAdr<=PicAdr)
 - just simple pixel-by-pixel-based remapping
```

1.51 TCS_RmpRGBxPic()

```
TCS_RmpRGBxPic()



INFO

 Remaps the raw chunky data of a picture from an RGBx mode to another.


SYN

 TCS_RmpRGBxPic(PicAdr, DstBufAdr, SouRGBxMode, DstRGBxMode, PicWd, PicHt)

                a0.l    a1.l       d0.b         d1.b         d2.w   d3.w


IN

 PicAdr         pointer to picture's RGBx (SouRGBxMode) data
 DstBufAdr      pointer to buffer for calculated RGBx (DstRGBxMode) data
 SouRGBxMode    picture's RGBx mode (any TCS_VM_RGBx value)
```

```
 DstRGBxMode   desired RGBx mode (any TCS_VM_RGBx value)
 PicWd         picture's width in pixels
 PicHt         picture's height in pixels


NOTE

 - make sure that PicWd & PicHt don't exceed the actual dimensions of the
   picture (otherwise dangerous writes to RAM will be done)
 - source and destination buffers can overlap as long as the destination
   buffer comes before the source one (DstBufAdr<=PicAdr)
 - just simple pixel-by-pixel-based remapping
```

1.52 TCS_FlgDXPfldCols()

```
TCS_FlgDXPfldCols()



INFO

 Returns the colors that look good/bad in Dual Cross Playfield mode given
 a certain RGBx transparent color.


SYN

 TCS_FlgDXPfldCols(FlgsAdr, col)

                   a0.l    d0.b


IN

 FlgsAdr   address of the buffer that will be filled as follows:
            - FlgsAdr[x].b=-1: x is a good-looking color
            - FlgsAdr[x].b=0: x is a bad-looking color
 col       transparent RGBx color


NOTE

 - a0.l is guaranteed to be left unmodified
 - be sure that the buffer is at least 256 bytes long
```

1.53 SvIFFRGBxPal()

```
TCS_SvIFFRGBxPal()



INFO
```

```
 Saves palette of a given RGBx mode to an IFF file.


SYN

 success = TCS_SvIFFRGBxPal(FlNm, RGBxID, BadCols, TrnspCol, DummyVal)

 ccr                        a0.l  d0.b    d1.b     d2.b      d3.l


IN

 FlNm       name of the file where to save the palette to
 RGBxID     one of the TCS_VM_RGBx values
 BadCols    if not 0, the bad-looking colors in Dual Cross Playfield mode
            will be marked as specified by the next parameters
 TrnspCol   transparent RGBx color in Dual Cross Playfield mode (only if
            BadCols<>0)
 DummyVal   24-bit RGB value to assign to bad-looking colors (only if
            BadCols<>0)


OUT

 success    ne = palette saved successfully
            eq = ERROR


NOTE

 - error returned if:
   - a) could not open file for output
   - b) could not write [all] data to file
   - c) could not allocate temporary memory
 - the AmigaOS must be ON because of possible disk activity
 - uses exec.library's FreeMem(), thus it can't be called from interrupts
 - existing files will be overwritten
```

1.54 4.6 Functions for Graphics

```
4.6   Functions for Graphics


The following functions allow to handle graphics almost effortlessly:

  Graphic Primitives
  Brush-handling Functions
  Miscellaneous Functions



 - function naming convention: for each Graphic Context-sensitive func-
```

tion there are always several other derivate functions which differ for just the trailing digit, which dictates the Graphic Context they act according to; in other words, if there is a Graphic Context-sensitive function called TCS_fnc(), there are also other functions of the form TCS_fncX(), which operate as required by the Graphic Context corresponding to X. For example, TCS_PltPxl0() simply plots a pixel without any particular operation, whereas TCS_PltPxl1() plots a pixel considering the Clipping Window which has been passed as additional argument (in fact, TCS_GCf_clp=1); analogously, TCS_DrwFrm3() will draw a filled frame taking into account the clipping limitations (3 = TCS_GCf_clp + + TCS_GCf_fill = 1 + 2).
This proves to be helpful to avoid the overhead of Graphic Context -sensitive functions (the simpler the function, the heavier the overhead!), when working with a fixed Graphic Context
- unsupported Graphic Contexts are simply ignored
- when not using clipping, make sure that the graphics to be drawn will fit in the destination screen
- never pass negative values (unless differently specified)
- the functions are as much general as possible (and, anyway, do not pretend to be the fastest in the world), so for better performance it is recommendable writing custom/specific routines

1.55 4.6.1 Graphic Primitives

4.6.1 Graphic Primitives

This group of functions is dedicated to basic graphics drawing:

function	supported Graphic Contexts
TCS_PltPxl()	normal, clipping, inverse
TCS_PltPxl0()	
TCS_PltPxl1()	
TCS_PltPxl4()	
TCS_PltPxl5()	
TCS_DrwLn()	normal, clipping, inverse
TCS_DrwLn0()	
TCS_DrwLn1()	
TCS_DrwLn4()	
TCS_DrwLn5()	
TCS_DrwHrzLn()	normal, clipping, inverse
TCS_DrwHrzLn0()	
TCS_DrwHrzLn1()	
TCS_DrwHrzLn4()	
TCS_DrwHrzLn5()	
TCS_DrwVrtLn()	normal, clipping, inverse
TCS_DrwVrtLn0()	
TCS_DrwVrtLn1()	
TCS_DrwVrtLn4()	
TCS_DrwVrtLn5()	
TCS_DrwFrm()	normal, clipping, filling, inverse
TCS_DrwFrm0()	
TCS_DrwFrm1()	

```
 TCS_DrwFrm2()
 TCS_DrwFrm3()
 TCS_DrwFrm4()
 TCS_DrwFrm5()
 TCS_DrwFrm6()
 TCS_DrwFrm7()
TCS_DrwSqr()        normal, clipping, filling, inverse
 TCS_DrwSqr0()
 TCS_DrwSqr1()
 TCS_DrwSqr2()
 TCS_DrwSqr3()
 TCS_DrwSqr4()
 TCS_DrwSqr5()
 TCS_DrwSqr6()
 TCS_DrwSqr7()
TCS_DrwTrngl()      normal, clipping, filling, inverse
 TCS_DrwTrngl0()
 TCS_DrwTrngl1()
 TCS_DrwTrngl2()
 TCS_DrwTrngl3()
 TCS_DrwTrngl4()
 TCS_DrwTrngl5()
 TCS_DrwTrngl6()
 TCS_DrwTrngl7()
TCS_DrwPlgn()       normal, clipping, filling, inverse
 TCS_DrwPlgn0()
 TCS_DrwPlgn1()
 TCS_DrwPlgn2()
 TCS_DrwPlgn3()
 TCS_DrwPlgn4()
 TCS_DrwPlgn5()
 TCS_DrwPlgn6()
 TCS_DrwPlgn7()
TCS_DrwOpnPlgn()    normal, clipping, inverse
 TCS_DrwOpnPlgn0()
 TCS_DrwOpnPlgn1()
 TCS_DrwOpnPlgn4()
 TCS_DrwOpnPlgn5()
TCS_DrwCrcl()       normal, clipping, filling, inverse
 TCS_DrwCrcl0()
 TCS_DrwCrcl1()
 TCS_DrwCrcl2()
 TCS_DrwCrcl3()
 TCS_DrwCrcl4()
 TCS_DrwCrcl5()
 TCS_DrwCrcl6()
 TCS_DrwCrcl7()
TCS_DrwElps()       normal, clipping, filling, inverse   (UNAVAILABLE)
 TCS_DrwElps0()                                            (UNAVAILABLE)
 TCS_DrwElps1()                                            (UNAVAILABLE)
 TCS_DrwElps2()                                            (UNAVAILABLE)
 TCS_DrwElps2()                                            (UNAVAILABLE)
 TCS_DrwElps4()                                            (UNAVAILABLE)
 TCS_DrwElps5()                                            (UNAVAILABLE)
 TCS_DrwElps6()                                            (UNAVAILABLE)
 TCS_DrwElps7()                                            (UNAVAILABLE)
TCS_FillArea()      normal, clipping, inverse
```

```
   TCS_FillArea0()
   TCS_FillArea1()
   TCS_FillArea4()
   TCS_FillArea5()
```

```
 - try to keep coordinates below 1024 (precise limitations will be given
   for each function as soon as possible)
```

1.56 TCS_PltPxl()

```
TCS_PltPxl()
```

```
INFO

 Plots a pixel on a logical screen.
```

```
SYN

 TCS_PltPxl(DIAdr, x,   y,   col  [, ClpWin])

            a0.l   d0.l d1.w d2.b    a3.l
```

```
IN

 DIAdr      screen display DisplayInfo structure pointer
 x,y        coordinates of the pixel
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

```
NOTE

 - calling a function for a simple pixel-plotting produces a great over-
   head, so you'd better write your own custom routine (if you need speed)
 - for speed, it is *absolutely granted* that this function will trash
   only d1 (i.e. all the other registers are left unmodified)
 - x is declared as .l for speed's sake (to avoid an "ext.l")
```

1.57 TCS_DrwLn()

```
TCS_DrwLn()
```

```
INFO
```

```
 Draws a line on a logical screen.


SYN

 TCS_DrwLn(DIAdr, x0,   y0, x1,  y1,  col  [, ClpWin])

           a0.l   d0.w d1.w d2.w d3.w d4.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 x0,y0      signed coordinates of the first pixel of the line
 x1,y1      signed coordinates of the last pixel of the line
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.58 TCS_DrwHrzLn()

```
TCS_DrwHrzLn()



INFO

 Draws a horizontal line on a logical screen.


SYN

 TCS_DrwHrzLn(DIAdr, x0,  x1,  y,   col  [, ClpWin])

              a0.l   d0.w d1.w d2.w d3.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 x0,x1      x coordinates of the first and last pixels
 y          y coordinate of both pixels
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.59 TCS_DrwVrtLn0()

```
TCS_DrwVrtLn()



INFO
```

```
 Draws a vertical line on a logical screen.


SYN

 TCS_DrwVrtLn(DIAdr, y0,   y1,  x,   col  [, ClpWin])

              a0.l   d0.w  d1.w d2.w d3.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 y0,y1      y coordinates of the first and last pixels
 x          x coordinate of both pixels
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.60 TCS_DrwFrm()

```
TCS_DrwFrm()



INFO

 Draws a rectangle on a logical screen.


SYN

 TCS_DrwFrm(DIAdr, x0,  y0,  x1,  y1,  col  [, ClpWin])

            a0.l   d0.w d1.w d2.w d3.w d4.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 x0,y0      coordinates of any corner
 x1,y1      coordinates of the opposite corner
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.61 TCS_DrwSqr()

```
TCS_DrwSqr()



INFO
```

```
 Draws a square on a logical screen.


SYN

 TCS_DrwSqr(DIAdr, x,   y,   SideLen, col  [, ClpWin])

            a0.l   d0.w d1.w d2.w     d3.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 x,y        coordinates of top-left corner
 SideLen    length of a side in pixels (>0)
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.62 TCS_DrwTrngl()

```
TCS_DrwTrngl()



INFO

 Draws a triangle on a logical screen.


SYN

 TCS_DrwTrngl(DIAdr, VtxsAdr, col  [, ClpWin])

              a0.l   a1.l     d0.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 VtxsAdr    pointer to 3 couples of the kind <x,y> where each couple indi-
            cates the signed coordinates of a vertex; components are .w
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure


NOTE

- in case of clipping, there must be some more than 4*h bytes free in the
  stack (h=|uy-dy|, hy = y of uppermost vertex, dy = y of downmost vertex)
```

1.63 TCS_DrwPlgn()

```
TCS_DrwPlgn()


INFO

 Draws a closed polygon on a logical screen.


SYN

 TCS_DrwPlgn(DIAdr, VtxsAdr, col  [, ClpWin])

             a0.l   a1.l     d0.b   a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 VtxsAdr    pointer to sequence of couples of the kind <x,y> where each
            couple indicates the signed coordinates of a vertex;
            each component is .w;
            the list must end with *two* NULL longwords
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure


NOTE

 - the polygon is automatically "closed", so you need not to (and, indeed,
   you should not) set the last vertex equal to the first
 - there *must* be at least three vertexes defined in the list!
 - when filling:
    - up to 64*ht+3912 bytes (ht = [clipped] polygon height) in the stack
      could be needed
    - no more than 31 polygon edges can "cross" a single rasterline: to be
      sure, simply keep the number of the edges below 32
 - best performance when the vertexes list is longword aligned
```

1.64 TCS_DrwOpnPlgn()

```
TCS_DrwOpnPlgn()


INFO

 Draws an open (i.e. last edge omitted) polygon on a logical screen.


SYN

 TCS_DrwOpnPlgn(DIAdr, VtxsAdr, col  [, ClpWin])
```

```
                a0.l   a1.l     d0.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 VtxsAdr    pointer to sequence of couples of the kind <x,y> where each
            couple indicates the signed coordinates of a vertex;
            each component is .w;
            the list must end with *two* NULL longwords
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure


NOTE

 - there *must* be at least one vertex defined in the list!
 - best performance when the vertexes list is longword aligned
```

1.65 TCS_DrwCrcl()

```
TCS_DrwCrcl()



INFO

 Draws a circle on a logical screen.


SYN

 TCS_DrwCrcl(DIAdr, cx,  cy,  rad, col  [, ClpWin])

             a0.l   d0.w d1.w d2.w d3.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 cx,cy      coordinates of the circle centre
 rad        circle radius length in pixels (>=0)
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.66 TCS_DrwElps()

```
TCS_DrwElps()



INFO
```

```
 Draws an ellipse on a logical screen.


SYN

 TCS_DrwElps(DIAdr, cx,  cy,  BAxis, SAxis, col  [, ClpWin])

             a0.l   d0.w d1.w d2.w   d3.w   d4.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 cx,cy      coordinates of the circle centre
 BAxis      bigger axis length in pixels (>=0)
 SAxis      smaller axis length in pixels (>=0)
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure
```

1.67 TCS_FillArea()

```
TCS_FillArea()



INFO

 Fills an area of a logical screen with a given RGBx color.


SYN

 TCS_FillArea(DIAdr, x,   y,   col  [, ClpWin])

              a0.l   d0.w d1.w d2.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 x,y        coordinates of the first pixel to fill (all pixels adjacent
            to this one and with the same color will be filled)
 col        color value in RGBx format
 [ClpWin]   pointer to ClippingWindow structure


NOTE

 - BE CAREFUL! The screen edges are not considered as limits!
 - this functions requires some room in the stack; more precisely, up to
   8*wd*ht bytes could be needed (wd & ht are the dimensions of the rec-
   tangle your polygon can be inscribed into or of the clipping window,
   if the latter is smaller). Generally this figure is much lower and de-
   pends on the shape of the polygon and the starting pixel; as a general
```

```
   rule try to start from the "centre" of the polygon (example: to fill a
   square (the worst case), 1.9*wd$^2$ bytes are required if starting from
   the top-left or bottom-right corner; just wd$^2$ are required if starting
   from the centre). Note that a better memory usage means also more speed
   (and not just the time spared for writes)
 - no operation is performed if:
    a) normal drawing: the pixel color at <x,y> is equal to col
    b) inverse drawing: col=0
```

1.68 4.6.2 Brush-handling Functions

```
4.6.2   Brush-handling Functions


The following functions work on complex graphic data like brushes:


 function                 supported Graphic Contexts

  TCS_GetBrsh()          none
  TCS_MkBrshMsk()        none
  TCS_MrgBrshs()         none
  TCS_MixBrshs()         none                          (UNAVAILABLE)
  TCS_FreeBrsh()         none
  TCS_WrtBrsh()          normal, clipping, masking
   TCS_WrtBrsh0()
   TCS_WrtBrsh1()
   TCS_WrtBrsh8()
   TCS_WrtBrsh9()
  TCS_FastWrtBrsh()      normal, masking
   TCS_FastWrtBrsh0()
   TCS_FastWrtBrsh8()
  TCS_WrtBrshZn()        normal, clipping, masking
   TCS_WrtBrshZn0()
   TCS_WrtBrshZn1()
   TCS_WrtBrshZn8()
   TCS_WrtBrshZn9()
  TCS_FastWrtBrshZn()    normal, masking
   TCS_FastWrtBrshZn0()
   TCS_FastWrtBrshZn8()
  TCS_FitBrsh()          normal, masking
   TCS_FitBrsh0()
   TCS_FitBrsh8()
  TCS_FastFitBrsh()      normal, masking
   TCS_FastFitBrsh0()
   TCS_FastFitBrsh8()
  TCS_FitBrshZn()        normal, masking
   TCS_FitBrshZn0()
   TCS_FitBrshZn8()
  TCS_FastFitBrshZn()    normal, masking
   TCS_FastFitBrshZn0()
   TCS_FastFitBrshZn8()
  TCS_RotZmBrsh()        normal, masking
   TCS_RotZmBrsh0()
```

```
   TCS_RotZmBrsh8()
  TCS_FastRotZmBrsh()    normal, masking
   TCS_FastRotZmBrsh0()
   TCS_FastRotZmBrsh8()
```

- many of the functions above require as arguments the brush dimensions: never pass a null or negative value for width or height!

1.69 TCS_GetBrsh()

```
TCS_GetBrsh()


INFO

 Cuts out a chunky brush from a logical screen.


SYN

 BrshAdr = TCS_GetBrsh(DIAdr, BufAdr, x,   y,   BrshWd, BrshHt)

 d0.l                   a0.l  a1.l    d0.w d1.w d2.w    d3.w


IN

 DIAdr     screen display DisplayInfo structure pointer
 BufAdr    address of buffer for snapped brush (0 = automatic allocation)
 x,y       coordinates of top-left corner on source screen
 BrshWd    width of brush in pixels
 BrshHt    height of brush in pixels


OUT

 BrshAdr   address of new brush (0=ERROR)


NOTE

 - if BufAdr=0:
    - error returned if could not allocate a buffer for the brush
    - the buffer must be de-allocated with TCS_FreeBrsh()
    - best-memory-first allocation
    - uses exec.library's AllocMem(), thus it can't be called from inter-
      rupts
 - if BufAdr<>0:
   - BrshAdr=BufAdr
   - make sure that the brush fits in the specified buffer
 - best performance when the source or destination data (better if both)
```

```
   are longword aligned (even better if the width is a multiple of 4)
```

1.70 TCS_MkBrshMsk()

```
TCS_MkBrshMsk()


INFO

 Creates a chunky mask for a brush.


SYN

 MskAdr = TCS_MkBrshMsk(BrshAdr, BufAdr, ptrn, TrnspCol, BrshWd, BrshHt)

 d0.l                   a0.l     a1.l    d0.b  d1.b      d2.w    d3.w


IN

 BrshAdr    address of brush
 BufAdr     address of buffer for calculated mask (0 = automatic alloca-
            tion)
 ptrn       filling pattern for non-zero pixels of mask (usually $ff)
 TrnspCol   color of brush pixels to be considered transparent
 BrshWd     width of brush in pixels
 BrshHt     height of brush in pixels


OUT

 MskAdr     address of new mask (0=ERROR)


NOTE

 - if BufAdr=0:
    - error returned if could not allocate a buffer for the mask
    - the buffer must be de-allocated with TCS_FreeBrsh()
    - best-memory-first allocation
    - uses exec.library's AllocMem(), thus it can't be called from inter-
      rupts
 - if BufAdr<>0:
    - MskAdr=BufAdr
    - make sure that the mask fits in the specified buffer
 - a mask can be used just like a brush because it *is* a brush
 - by playing with the fill pattern value, you can easily obtain pseudo-
   transparency effect:
    - $88 = include red component
    - $44 = include green component
    - $22 = include blue component
    - $11 = include extra component
   examples:
```

```
    - $ff: include all the components, non-trasparent pixels of the brush
           will appear totally opaque (just like they are)
    - $77: the brush pixels will lose the red component ($ff-$88=$77) and
           will let the same component of the background pixels to be vi-
           sible through them
```

1.71 TCS_MrgBrshs()

```
TCS_MrgBrshs()



INFO

 Merges two different brushes together.


SYN

 BrshAdr = TCS_MrgBrshs(Brsh0Adr, Brsh1Adr, Msk0Adr, BufAdr,

 d0.l                   a0.l      a1.l      a2.l     a3.l

                        BrshsWd, BrshsHt)

                        d0.w    d1.w


IN

 Brsh0Adr   address of first brush
 Brsh1Adr   address of second brush
 Msk0Adr    address of first brush mask
 BufAdr     address of destination buffer for merged brush (0 = automatic
            allocation)
 BrshsWd    width of brushes in pixels
 BrshsHt    height of brushes in pixels


OUT

 BrshAdr   address of merged brush (0=ERROR)


NOTE

 - if BufAdr=0:
    - error returned if could not allocate a buffer for the brush
    - the buffer must be de-allocated with TCS_FreeBrsh()
    - best-memory-first allocation
    - uses exec.library's AllocMem(), thus it can't be called from inter-
      rupts
 - if BufAdr<>0:
   - BrshAdr=BufAdr
   - make sure that the brush fits in the specified buffer
```

```
    - destination buffer may coincide with one of the sources
  - best performance when the source or destination data (better if both)
    are longword aligned (even better if the width is a multiple of 4)
  - see also TCS_MixBrshs
```

1.72 TCS_FreeBrsh()

```
TCS_FreeBrsh()


INFO

 Frees the memory buffer allocated for a brush by a brush-maker function.


SYN

 TCS_FreeBrsh(BrshAdr)

              a0.l


IN

 BrshAdr   address of brush


NOTE

 - useless if the brush buffer had been allocated by the user
 - safe to call even if BrshAdr is wrong
 - uses exec.library's FreeMem(), thus it can't be called from interrupts
```

1.73 TCS_WrtBrsh()

```
TCS_WrtBrsh()


INFO

 Writes a chunky brush to a logical screen.


SYN

 TCS_WrtBrsh(DIAdr, BrshAdr, x,   y,   BrshWd, BrshHt

             a0.l   a1.l     d0.w d1.w d2.w    d3.w

             [, MskAdr] [, ClpWin])
```

```
                a2.l       a3.l

IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 x,y        coordinates of top-left corner on destination screen
 BrshWd     width of brush in pixels
 BrshHt     height of brush in pixels
 [MskAdr]   address of mask
 [ClpWin]   pointer to ClippingWindow structure


NOTE

 - if clipping is unused, the brush must lie entirely inside the screen!
 - if used, the mask must have the same dimensions of the brush
 - best performance when the source or destination data (better if both)
   are longword aligned (even better if the width is a multiple of 4)
 - slower than TCS_FastWrtBrsh()
```

1.74 TCS_FastWrtBrsh()

```
TCS_FastWrtBrsh()



INFO

 Writes a chunky brush to a logical screen.


SYN

 TCS_FastWrtBrsh(DIAdr, BrshAdr, x,   y,   BrshWd, BrshHt [, MskAdr])

                  a0.l   a1.l     d0.w d1.w d2.w    d3.w      a2.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 x,y        coordinates of top-left corner on destination screen
 BrshWd     width of brush in pixels; must be multiple of 16
 BrshHt     height of brush in pixels
 [MskAdr]   address of mask


NOTE

 - make sure that the brush lies entirely inside the screen!
 - if used, the mask must have the same dimensions of the brush
```

```
 - best performance when the source or destination data (better if both)
   are longword aligned
 - faster than TCS_WrtBrsh()
```

1.75 TCS_WrtBrshZn()

```
TCS_WrtBrshZn()



INFO

 Writes a rectangular zone of a chunky brush to a logical screen.


SYN

 TCS_WrtBrshZn(DIAdr, BrshAdr, sx0, sy0, sx1, sy1, dx,  dy,

               a0.l   a1.l     d0.w d1.w d2.w d3.w d4.w d5.w

               BrshWd [, MskAdr] [, BrshHt, ClpWin])

               d6.w      a2.l       d7.w    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 sx0,sy0    coordinates of source rectangle top-left corner
 sx1,sy1    coordinates of source rectangle bottom-right corner
 dx,dy      coordinates of destination rectangle top-left corner
 BrshWd     width of source brush in pixels
 [MskAdr]   address of mask
 [BrshHt]   height of source brush in pixels
 [ClpWin]   pointer to ClippingWindow structure


NOTE

 - if clipping is unused, the zone must lie entirely inside the screen!
 - if used, the mask must have the same dimensions of the brush
 - best performance when the source or destination data (better if both)
   are longword aligned (even better if the width is a multiple of 4)
 - see also TCS_FastWrtBrshZn()
```

1.76 TCS_FastWrtBrshZn()

```
TCS_FastWrtBrshZn()
```

```
INFO

 Writes a rectangular zone of a chunky brush to a logical screen.


SYN

 TCS_FastWrtBrshZn(DIAdr, BrshAdr, sx0, sy0, sx1, sy1, dx,  dy,  BrshWd

                   a0.l   a1.l     d0.w d1.w d2.w d3.w d4.w d5.w d6.w

                   [, MskAdr])

                      a2.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 sx0,sy0    coordinates of rectangle top-left corner
 sx1,sy1    coordinates of rectangle bottom-right corner
 dx,dy      coordinates of destination rectangle top-left corner
 BrshWd     width of source brush in pixels
 [MskAdr]   address of mask


NOTE

 - make sure that the zone lies entirely inside the screen!
 - the zone width (=sx1-sx0+1) must be multiple of 16
 - if used, the mask must have the same dimensions of the brush
 - best performance when the source or destination data (better if both)
   are longword aligned
 - see also TCS_WrtBrshZn()
```

1.77 TCS_FitBrsh()

```
TCS_FitBrsh()



INFO

 Given an 8-bit chunky brush, "fits" it to a rectangular zone of any size
 in a logical screen.


SYN

 TCS_FitBrsh(DIAdr, BrshAdr, VtxsAdr, BrshWd, BrshHt [, MskAdr])

             a0.l   a1.l     a2.l     d0.w    d1.w      a4.l
```

```
IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 VtxsAdr    pointer of a structure of this kind:

             offset   content

             0,2      x0,y0: coordinates of top-left pixel of destination
                             rectangle
             4,6      x1,y1: coordinates of bottom-right pixel of destina-
                             tion rectangle

 BrshWd      width of brush in pixels
 BrshHt      height of brush in pixels
 [MskAdr]   address of mask


NOTE

 - make sure that the destination zone lies completely inside the screen!
 - if used, the mask must have the same dimensions of the brush
 - see also TCS_FastFitBrsh()
```

1.78 TCS_FastFitBrsh()

```
TCS_FastFitBrsh()



INFO

 Given an 8-bit chunky brush, "fits" it to a rectangular zone of any size
 in a logical screen.


SYN

 TCS_FastFitBrsh(DIAdr, BrshAdr, VtxsAdr, BrshWd, BrshHt [, MskAdr])

                 a0.l   a1.l     a2.l     d0.w    d1.w      a4.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 VtxsAdr    pointer of a structure of this kind:

             offset   content

             0,2      x0,y0: coordinates of top-left pixel of destination
                             rectangle
             4,6      x1,y1: coordinates of bottom-right pixel of destina-
```

```
                              tion rectangle

 BrshWd      width of brush in pixels
 BrshHt      height of brush in pixels
 [MskAdr]    address of mask


NOTE

 - make sure that the destination zone lies completely inside the screen!
 - destination zone width (=x1-x0+1) must be multiple of 4 (automatic
   rounding to the next multiple always performed)
 - if used, the mask must have the same dimensions of the brush
 - best performance when x0 is multiple of 4
 - see also TCS_FitBrsh()
```

1.79 TCS_FitBrshZn()

```
TCS_FitBrshZn()



INFO

 Given an 8-bit chunky brush, "fits" a rectangular zone of any size from
 such brush into another rectangular zone of any other size in a logical
 screen.


SYN

 TCS_FitBrshZn(DIAdr, BrshAdr, VtxsAdr, BrshWd [, MskAdr])

               a0.l   a1.l     a2.l     d0.w      a4.l


IN

 DIAdr       screen display DisplayInfo structure pointer
 BrshAdr     address of brush
 VtxsAdr     pointer of a structure of this kind:

              offset   content

              0,2      sx0,sy0: coordinates of top-left pixel of source
                                rectangle
              4,6      sx1,sy1: coordinates of bottom-right pixel of source
                                rectangle
              8,10     dx0,dy0: coordinates of top-left pixel of destina-
                                tion rectangle
              12,14    dx1,dy1: coordinates of bottom-right pixel of desti-
                                nation rectangle

 BrshWd      width of brush in pixels
 [MskAdr]    address of mask
```

```
NOTE

 - make sure that the destination zone lies completely inside the screen!
 - if used, the mask must have the same dimensions of the brush
 - see also TCS_FastFitBrshZn()
```

1.80 TCS_FitBrshZn()

```
TCS_FastFitBrshZn()



INFO

 Given an 8-bit chunky brush, "fits" a rectangular zone of any size from
 such brush into another rectangular zone of any other size in a logical
 screen.


SYN

 TCS_FastFitBrshZn(DIAdr, BrshAdr, VtxsAdr, BrshWd [, MskAdr])

                   a0.l   a1.l     a2.l     d0.w      a4.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 BrshAdr    address of brush
 VtxsAdr    pointer of a structure of this kind:

             offset   content

             0,2      sx0,sy0: coordinates of top-left pixel of source
                               rectangle
             4,6      sx1,sy1: coordinates of bottom-right pixel of source
                               rectangle
             8,10     dx0,dy0: coordinates of top-left pixel of destina-
                               tion rectangle
             12,14    dx1,dy1: coordinates of bottom-right pixel of desti-
                               nation rectangle

 BrshWd     width of brush in pixels
 [MskAdr]   address of mask


NOTE

 - make sure that the destination zone lies completely inside the screen!
 - destination zone width (=dx1-dx0+1) must be multiple of 4 (automatic
   rounding to the next multiple always performed)
 - if used, the mask must have the same dimensions of the brush
```

```
 - best performance when dx0 is multiple of 4
 - see also TCS_FitBrshZn()
```

1.81 TCS_RotZmBrsh()

```
TCS_RotZmBrsh()



INFO

 Rotates and scales a chunky brush into a rectangular zone of a logical
 screen.


SYN

 TCS_RotZmBrsh(DIAdr, BrshCntrAdr, x0,  y0,  x1,  y1,  ang, scl,

               a0.l   a1.l         d0.w d1.w d2.w d3.w d4.l d5.w

               BrshWdLog [, MskCntrAdr])

               d6.w         a2.l


IN

 DIAdr          screen display DisplayInfo structure pointer
 BrshCntrAdr    address of brush rotation centre
 x0,y0          coordinates of top-left pixel of destination rectangle
 x1,y1          coordinates of bottom-right pixel of destination rectangle
 ang            anti-clockwise rotation angle in unsigned degrees
 scl            scale factor in [0...65535] (a factor x scales by 256/x;
                e.g: x=128 -> $\times$2, x=256 -> $\times$1, x=512 -> $\times$0 ↩
                    .5)
 BrshWdLog      base 2 logarithm of width of brush in pixels (>0)
 [MskCntrAdr]   address of mask rotation centre


NOTE

 - make sure that the destination zone lies completely inside the screen!
 - of course brush width must be a power of 2
 - if used, the mask must have the same dimensions of the brush
 - slower than TCS_FastRotZmBrsh()
```

1.82 TCS_FastRotZmBrsh()

```
TCS_FastRotZmBrsh()
```

```
INFO

 Rotates and scales a chunky brush into a rectangular zone of a logical
 screen.


SYN

 TCS_FastRotZmBrsh(DIAdr, BrshCntrAdr, x0,  y0,  x1,  y1,  ang, scl,

                   a0.l   a1.l         d0.w d1.w d2.w d3.w d4.l d5.w

                   BrshWdLog [, MskCntrAdr])

                   d6.w         a2.l


IN

 DIAdr           screen display DisplayInfo structure pointer
 BrshCntrAdr     address of brush rotation centre
 x0,y0           coordinates of top-left pixel of destination rectangle
 x1,y1           coordinates of bottom-right pixel of destination rectangle
 ang             anti-clockwise rotation angle in unsigned degrees
 scl             scale factor in [0...65535] (a factor x scales by 256/x;
                 e.g: x=128 -> $\times$2, x=256 -> $\times$1, x=512 -> $\times$0 ↩
                     .5)
 BrshWdLog       base 2 logarithm of width of brush in pixels (>0)
 [MskCntrAdr]    address of mask rotation centre


NOTE

 - make sure that the destination zone lies completely inside the screen!
 - destination zone width (=x1-x0+1) must be multiple of 4 (automatic
   rounding to the next multiple always performed)
 - of course brush width must be a power of 2
 - if used, the mask must have the same dimensions of the brush
 - best performance when x0 is multiple of 4
 - faster than TCS_RotZmBrsh()
```

1.83 4.6.3 Miscellaneous Functions

```
4.6.3   Miscellaneous Functions


Almost uncategorizable:


 function                supported Graphic Contexts

  TCS_ClrScr()          normal, clipping
   TCS_ClrScr0()
```

```
   TCS_ClrScr1()
  TCS_CpyScr()          normal, clipping
   TCS_CpyScr0()
   TCS_CpyScr1()
  TCS_CpyScrZn()        normal, clipping
   TCS_CpyScrZn0()
   TCS_CpyScrZn1()
  TCS_FastCpyScrZn()    none
  TCS_ClpLn()           none
  TCS_FillBuf()         none
```

1.84 TCS_ClrScr()

```
TCS_ClrScr()


INFO

 Clears with a given RGBx color a logical screen.


SYN

 TCS_ClrScr(DIAdr, col  [, ClpWin])

            a0.l   d0.b    a3.l


IN

 DIAdr      screen display DisplayInfo structure pointer
 col        RGBx value of the color the screen has to be cleared with
 [ClpWin]   pointer to ClippingWindow structure
```

1.85 TCS_CpyScr()

```
TCS_CpyScr()


INFO

 Copies a logical screen to another logical screen.


SYN

 success = TCS_CpyScr(SouDIAdr, DstDIAdr [, ClpWin])

 ccr                  a0.l      a1.l        a3.l
```

```
IN

 SouDIAdr   source screen display DisplayInfo structure pointer
 DstDIAdr   destination screen display DisplayInfo structure pointer
 [ClpWin]   pointer to ClippingWindow structure


OUT

 success    ne = screen copied successfully
            eq = ERROR


NOTE

 - error returned if any of the dimensions of the screens are different
 - Blitter not used even if the screens' buffers are in CHIP ram
```

1.86 TCS_CpyScrZn()

```
TCS_CpyScrZn()



INFO

 Copies a rectangular zone of a logical screen of a display to the logical
 screen of another display.


SYN

 TCS_CpyScrZn(SouDIAdr, DstDIAdr, sx0, sy0, sx1, sy1, dx,  dy   [, ClpWin])

              a0.l      a1.l      d0.w d1.w d2.w d3.w d4.w d5.w    a3.l


IN

 SouDIAdr   source screen display DisplayInfo structure pointer
 DstDIAdr   destination screen display DisplayInfo structure pointer
 sx0,sy0    source rectangle top-left corner coordinates
 sx1,sy1    source rectangle bottom-right corner coordinates
 dx,dy      destination rectangle top-left corner coordinates
 [ClpWin]   pointer to ClippingWindow structure for destination screen


NOTE

 - if clipping is unused, the zones must lie entirely inside the screens;
   otherwise source and destination zones may lie (partially) outside of
   the respective screen (the coordinates are signed)
 - source and destination may overlap as long as destination comes before
   source (i.e. dy<sy0 | (dy=sy0 & dx<=sx0))
```

```
 - best performance when the source or destination data (better if both)
   are longword aligned (even better if the width is a multiple of 4)
 - Blitter not used even if the screens' buffers are in CHIP ram
 - see also TCS_FastCpyScrZn()
```

1.87 TCS_FastCpyScrZn()

```
TCS_FastCpyScrZn()



INFO

 Copies a rectangular zone of a logical screen of a display to the logical
 screen of another display.


SYN

 TCS_FastCpyScrZn(SouDIAdr, DstDIAdr, sx0, sy0, sx1, sy1, dx,  dy)

                  a0.l      a1.l      d0.w d1.w d2.w d3.w d4.w d5.w


IN

 SouDIAdr   source screen display DisplayInfo structure pointer
 DstDIAdr   destination screen display DisplayInfo structure pointer
 sx0,sy0    source rectangle top-left corner signed coordinates
 sx1,sy1    source rectangle bottom-right corner signed coordinates
 dx,dy      destination rectangle top-left corner signed coordinates


NOTE

 - the zone width (sx1-sx0+1) must be a multiple of 16 (automatic rounding
   to the previous multiple always performed)
 - the zones must lie entirely inside the screens
 - source and destination may overlap as long as destination comes before
   source (i.e. dy<sy0 | (dy=sy0 & dx<=sx0))
 - best performance when the source or destination data (better if both)
   are longword aligned
 - Blitter not used even if the screens' buffers are in CHIP ram
 - see also TCS_CpyScrZn()
```

1.88 TCS_ClpLn()

```
TCS_ClpLn()



INFO
```

```
 Clips a line without drawing it.


SYN

 inside, cx0, cy0, cx1, cy1 = TCS_ClpLn(x0,  y0,  x1,  y1,  ClpWin)

 ccr     d0.w d1.w d2.w d3.w            d0.w d1.w d2.w d3.w a3.l


IN

 x0,y0     signed coordinates of the first pixel of the line
 x1,y1     signed coordinates of the last pixel of the line
 ClpWin    pointer to ClippingWindow structure


OUT

 inside    ne = the line lies (partially) inside the ClippingWindow
           eq = the line lies completely outside the ClippingWindow
 cx0,cy0   signed coordinates of the first pixel of the clipped line
 cx1,cy1   signed coordinates of the last pixel of the clipped line


NOTE

 - <cx0,cy0> and <cx1,cy1> are equal to <x0,y0> and <x1,y1>, respectively,
   if inside=eq (cx0 and cy0 are sign-extended to 32 bits)
 - guaranteed not to trash any register
```

1.89 TCS_FillBuf()

```
TCS_FillBuf()



INFO

 Fills a buffer with a given pattern (useful for filling mask planes).


SYN

 TCS_FillBuf(BufAdr, BufSz, ptrn)

              a0.l    d0.l   d1.l


IN

 BufAdr   buffer address
 BufSz    buffer size in bytes
 ptrn     bit-pattern
```

```
NOTE

 - quite good, but for extra-extra-extra-extra-fast performance, write
   yourself a function which fits _perfectly_ your needs
```

1.90 4.7 Functions for Picture Files

```
4.7   Functions for Picture Files



These functions allow you to quickly access IFF files from/to which load/
/save graphics:

  TCS_LdRGBx()       (UNAVAILABLE)
  TCS_UnLdRGBx()     (UNAVAILABLE)
  TCS_SvRGBx()       (UNAVAILABLE)
  TCS_SvScr2RGBx()   (UNAVAILABLE)
  TCS_LdILBM()
  TCS_UnLdILBM()
  TCS_SvILBM()       (UNAVAILABLE)
```

1.91 TCS_LdILBM()

```
TCS_LdILBM()



INFO

 Loads an IFF InterLeaved BitMap file into a chunky buffer.


SYN

 ILBMStruc = TCS_LdILBM(FlNm, BufAdr, BufLen)

 d0.l                    a0.l  a1.l    d0.l


IN

 FlNm     name of the file to load
 BufAdr   address of buffer for raster data (0 = automatic allocation)
 BufLen   size in bytes of destination buffer (only if BufAdr<>0)


OUT

 ILBMStruc   pointer to an ILBMInfo structure or a TCS_PE_#? errcode
```

```
NOTE

 - the RGBx mode returned in the structure is the one which best matches
   the ILBM palette saved in the CMAP chunk of the IFF (the ILBM palette
   generally should be one of those in the TCS/pal/ directory): in case
   there is not an exact match, the best RGBx mode is chosen, but *no*
   remapping is performed! - the degree of mismatch can be found in the
   TCS_II_PalDiff field
 - only 24-bit color values in the CMAP chunk of the IFF are correctly
   interpreted (old 12-bit CMAPs don't work!)
 - currently only 8-bitplane, non-masked, non-HAM, ILBMs supported (unsup-
   ported formats generate a TCS_PE_BADILBM error)!
 - if the specified destination buffer is too small, a TCS_PE_LOWMEM
   error will be returned
 - ILBM body data is converted on line basis, so you don't need twice
   the memory for just loading
 - deallocate memory only with TCS_UnLdILBM()
 - make sure to pass correct values for BufAdr and BufSz!
 - make sure the AmigaOS is ON because of possible disk activity
 - uses exec.library's FreeMem(), thus it can't be called from interrupts
```

1.92 TCS_UnLdILBM()

```
TCS_UnLdILBM()



INFO

 Frees the memory allocated by LdILBM().


SYN

 TCS_UnLdILBM(ILBMStruc)

              a0.l


IN

 ILBMStruc   address of an ILBMInfo structure


NOTE

 - safe to call even if ILBMStruc is wrong/corrupted (at most you'll end
   up with a memory leak due to the failed de-allocation of memory)
 - uses exec.library's FreeMem(), thus it can't be called from interrupts
 - of course, the eventual user-specified raster buffer won't be freed
```

1.93 4.8 Simple Meta-Example

```
4.8    Simple Meta-Example



I'd better give directly a "concrete" example, I guess.
This mainly serves the purpose of showing the usage of the simplest (and
most important!) functions to create a display:

 < your code starts here >
 < ... >
 TCSBase = OpenLibrary("tcs.library",1)         ;get lib pointer
 < ... >
 < declare a proper DD structure and call it "MyDD" >
 < ... >
 DIAdr = TCS_InitDspl(MyDD)                     ;init your own display
 if DIAdr<>0                                     ;if succedeed
    < ... >
    ChnkScr = DIAdr.TCS_DI_CSAdr                 ;address of chunky screen
    < ... >
    < take control over Amiga hardware in the cleanest way possible! >
    < ... >
    TCS_ShwDspl(DIAdr)                          ;activate display
    < ... >
    < write/read whatever you want in the buffer pointed by ChnkScr >
    < ... >
    < OK, enough >
    < ... >
    TCS_HideDspl(DIAdr,0)                       ;deactivate display
    < ... >
    < restore AmigaOS here >
    < ... >
    TCS_FreeDspl(DIAdr)                         ;free display resources
    < ... >
 endif
 < ... >
 CloseLibrary(TCSBase)
 < ... >
 < your code ends here >



  - working examples may be found in the TCS/examples/ directory
```

1.94 4.9 Known Bugs & Problems

```
4.9    Known Bugs & Problems



- HalfRes+MskPln does not work correctly in (Dual) Cross Playfield mode.
  I do not absolutely know why this happens... it does make perfectly
```

sense that this particular mode gives problems (it’s the only mode using five bitplanes), yet it nonetheless becomes nonsense each time I look at the code and all I can come up with is: "Hey, it’s perfect! It takes into account the difference with the other modes everywhere needed and always does the right thing in the right place! Moreover, the rest of the code works perfectly in the other modes, so maybe... am I missing something in the theory? But... what...?"

- due to clipping approximation, clipped lines could be drawn slightly differently from unclipped ones; some bad side effects:

  - clipped lines may not completely coincide with unclipped ones
  - clipped polygons could show a few unfilled pixels by the borders (to be fixed with Sutherland-Hodgman clipping)

- I suspect there is something wrong in RGB -> RGBP conversion... I will check it again later

- generally graphic functions have this problem with inverse drawing: if the same pixel is plotted an even number of times, it’s as if it had never been drawn (example: corners of a frame). It’s a rather silly bug, but fixing it sometimes may be very painful!

This version of the library has been tested only on an A1200 [+Bz1230+16Mb of 60 ns RAM], so it may well fail to work correctly on your equipment.