lib

COLLABORATORS

	TITLE : lib		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 8, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

lib

1.1 4 The tcs.library

4 The tcs.library

CURRENT RELEASE IMPORTANT INFO
<- READ HERE FIRST!!!

Well, after all that theory in the previous sections (wHaT?!? You haven't read a single thing of all that stuff?!? Ow... you'd better have a look at the basic information, at least!), I guess you now want to do some practice; there are 2 ways:

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

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

```
  - 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 CURRENT RELEASE IMPORTANT INFO

```
CURRENT RELEASE IMPORTANT INFO



This release includes an _early_/_preliminary_ version of the library
which is likely not to be (binary) compatible with future versions.
Always use the include files (tcs.i and tcs_lib.i) enclosed here and
do not make any "special" assumption: in this way, your application will
probably need just to be recompiled to work correctly (anyway, detailed
compatibility informations and directions will be given in the next re-
leases).

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.

[
 I decided to upload it in the current stage of development because I'm
 going home for Easter vacations so I won't work at it for some weeks...
 meanwhile, you can experiment a bit! I hope, when I'll be back, to find a
 lot of feedback responses: criticisms, suggestions, detailed bug reports,
 etc., I'll be glad to hear from you and I'll do my best to improve this
 piece of software following your directions (yet, if you suggest new, ad-
 ditional features, don't expect them to be implemented immediately: I
 must first complete what's been left unfinished).

 So... enjoy and let me know!
]
```

1.3 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 (to remap a picture follow these steps:

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

at this point, to use the picture with the library functions, you can:

a) load it with
TCS_LdILBM()
b) convert it to raw chunky with another tool

).

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).

1.4 4.2 General Guidelines

4.2 General Guidelines

It must be well clear right from the start that some functions expressely require the AmigaOS to be ON and some others to be OFF, so pay attention to their description.

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 & 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 far as Cross Playfield is concerned, 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).


Finally, here are some other 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!!!
 - don't call [de-]allocation functions from interrupts, because they use
   exec.library AllocMem() and FreeMem()!
 - don't use the
                functions for picture loading/saving
                 if AmigaOS is OFF!
```

1.5 4.3 Declarations Description

```
                 4.3   Declarations Description



Before going on, I'd better shortly illustrate the meaning of some decla-
rations 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.

4.3.1
                 VideoModes definitions, bits and flags
                 4.3.2
```

```
- 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
```

1.6 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_xxx values and/or by bclr/bset-ing the
relevant bits using the TCS_VMb_xxx 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)
```

1.7 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                               |
  |                                            |
  |                                            |
  |    <lf,top>                                | lf   = TCS_CW_lf
  |       *---------------------+              | top = TCS_CW_top
  |       |clipping window      |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       |                     |              |
  |       +---------------------*              |
  |                          <rt,btm>          | rt   = TCS_CW_rt
  |                                            | btm = TCS_CW_btm
  +--------------------------------------------+
```

1.8 4.3.3 GraphicContext bits, flags and structure

4.3.3 GraphicContext bits, flags and structure

The hi-level graphic functions operate on a screen according to the directions written in the display’s Graphic Context structure:

TCS_GC_flgs : bitmap made of any combination of the TCS_GCf_xxx flags

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

1.9 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 _be-
                 fore_ 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 64bit burst for bitplane
                  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 of display at startup
                  - TCS_DD_brtns in [0...256]

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

1.10 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

1.11 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)
```

1.12 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_GfxAdr  : address of the chunky buffer holding the graphics data
TCS_II_PalAdr  : address of the original raw 24-bit palette
TCS_II_wd      : width in pixels
TCS_II_ht      : height in pixels
TCS_II_PlnsNo  : number of planes
TCS_II_RGBxMode: RBBx mode automatically selected by
TCS_LdILBM()
```

1.13 4.4 Functions for TCS Displays

4.4 Functions for TCS Displays

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

4.4.1 TCS_InitDspl()
4.4.2 TCS_ShwDspl()
4.4.3 TCS_HideDspl()
4.4.4 TCS_FreeDspl()
4.4.5 TCS_CPUFRPass0()
4.4.6 TCS_CPUFRPass1()
4.4.7 TCS_CPUFRPass2()
4.4.8 TCS_BltFRPass0()
4.4.9 TCS_BltFRPassHndlr()

1.14 4.4.1 TCS_InitDspl()

```
                 4.4.1   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) .ScrWd < DsplWd/8 [HalfRes] or .ScrWd < DsplWd/4 [FullRes]
       (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
```

1.15 4.4.2 TCS_ShwDspl()

```
                4.4.2   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!
 - activates all the needed DMA channels (bitplanes, Copper and, in case
   of Blitter-assisted FullRes conversion, Blitter)
 - it always switches to PAL
 - Cross Playfield mode is restored if other playfield already shown
 - use
               TCS_HideDspl()
                to hide the display without closing it
```

1.16 4.4.3 TCS_HideDspl()

```
                4.4.3   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 or 0 to 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)
```

1.17 4.4.4 TCS_FreeDspl()

```
                 4.4.4   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!
```

1.18 4.4.5 TCS_CPUFRPass0()

```
                 4.4.5   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()
```

1.19 4.4.6 TCS_CPUFRPass1()

```
                 4.4.6   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)
```

```
                                <0,0>
                                  *-----------------------+
          <0,0>                   |chunky screen          |
            *------------+        |   <sx,sy>             |
            |display     |        |      *------------+   |
            |window      |        |      |source area |   |
            |            |        |      |(same size  |   |
            |            |        |      |of display  |   |
            +------------+        |      |window)     |   |
                                  |      +------------+   |
                                  +-----------------------+
```

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:
```

```
                                <0,0>
                                  *-----------------------+
          <0,0>                   |chunky screen          |
            *------------+        |                       |
            |display     |        |                       |
            |window      |        |                       |
            |            |        |                       |
            |            |        |   <sx,sy>             |
            +------------+        |      *------------+   |
                                  |      |source area |   |
                                  +------+------------+---+
                                         |############|
                                         |############|
                                         +------------+
```

```
    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()
```

1.20 4.4.7 TCS_CPUFRPass2()

```
                4.4.7   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

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

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()
```

1.21 4.4.8 TCS_BltFRPass0()

```
                4.4.8   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 multi-
    tasking 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 conver-
    sion 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 func-
    tion works on planar basis: the Blitter first converts the whole Vdo-
    Pln1 and then VdoPln1 (on the contrary, TCS_CPUFRPassX() 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()
```

1.22 4.4.9 TCS_BltFRPassHndlr()

```
                4.4.9   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()
```

1.23 4.4.10 TCS_WtBltFRPass()

```
                4.4.10   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 cur-
    rent conversion is relative to the specified display
  - make sure
                TCS_BltFRPassHndlr()
                can still be called (i.e.: the inter-
    rupt 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!)
```

1.24 4.4.11 TCS_DubSwp()

```
                4.4.11   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 or during
 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 despite you waited for the VBL before calling the function; in such case a (stupid) solution would be waiting or performing non-graphic operations till the next VBL

1.25 4.4.12 TCS_TriSwp()

4.4.12 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 VBL subsequent 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 part)
          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 part)
          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 concurrently, out of processor's control: this means that there is no guarantee that when trying to perform the swapping inside the handler it has already finished (it could be both in the middle of 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 immediately 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() !

1.26 4.4.13 TCS_TriUpd()

4.4.13 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
```

1.27 4.4.14 TCS_WtTriSwp()

```
                 4.4.14   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()
```

1.28 4.4.15 TCS_SttcSwp()

4.4.15 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 copperlist) 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 initialize all the video buffers before starting with the real buffering (see also
GetVdoBufs()
)
- don't call if swapping is currently running

1.29 4.4.16 TCS_GetVdoBufs()

4.4.16 TCS_GetVdoBufs()

INFO

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

SYN

```
 TCS_GetVdoBufs(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

 - especially useful to initialize all the video buffers before starting
   with the real buffering (see also TriSttcSwp())
 - 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
```

1.30 4.4.17 TCS_FillBuf()

```
4.4.17   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
```

1.31 4.4.18 TCS_SetPlnsPos()

```
                4.4.18   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!
 - to obtain the same effect in FullRes simply change the input values of

               TCS_CPUFRPass1()
                or
               TCS_CPUFRPass2()
                 - 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)
 - 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!
```

1.32 4.4.19 TCS_SetPlnsVPos()

```
                4.4.19   TCS_SetPlnsVPos()
```

INFO

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

SYN

```
TCS_SetPlnsVPos(DIAdr, YPos)

                a0.l   d0.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

1.33 4.4.20 TCS_EnbXPfld()

4.4.20 TCS_EnbXPfld()

INFO

Enables the Cross Playfield mode by superimposing a screen (from a previously 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>
 - the opacity is set to 256 and the Dual mode is turned OFF by default
 - the playfields are _not_ automatically shown, but instead you must call

               TCS_ShwDspl()
                anytime after enabling the mode
```

1.34 4.4.21 TCS_DsbXPfld()

```
                4.4.21   TCS_DsbXPfld()



INFO

 Deactivates the Cross Playfield mode.

SYN

 success = TCS_DsbXPfld(DIAdr)

 ccr                    a0.l

IN

 DIAdr
               DisplayInfo structure
                of the playfield to hide after
           disabling the mode
```

```
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)
```

1.35 4.4.22 TCS_SetFPfldOpct()

```
                4.4.22   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)

                  a0.l   d0.w

IN

 DIAdr
               DisplayInfo structure
                of any playfield
 opct    opacity degree of front playfield, belonging to [0...256]
         ( = [totally transparent ... totally opaque])

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)
```

1.36 4.4.23 TCS_EnbDXPfld()

```
                4.4.23   TCS_EnbDXPfld()
```

```
INFO

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

SYN

 success = TCS_EnbDXPfld(DIAdr, col)

 ccr                     a0.l   d0.b

IN

 DIAdr
              DisplayInfo structure
               of any playfield
 col       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 4.4.24 TCS_DsbDXPfld()

```
                4.4.24   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 4.4.25 TCS_SetGfxCtxt()

```
                4.4.25   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
```

1.39 4.5 Functions for Color/Palette Control

```
                4.5   Functions for Color/Palette Control



These functions give you control over the palette/color of displays and
provide some comfortable ways to handle RGBx data:

 4.5.1
```

1.40 4.5.1 TCS_SetRGBx()

```
                4.5.1   TCS_SetRGBx()



 INFO

  Sets the RGBx mode and palette of a display.

 SYN

  TCS_SetRGBx(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 allows to easily achive 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
```

1.41 4.5.2 TCS_GetRGBBrtns()

```
4.5.2   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 4.5.3 TCS_GetRGBxBrtns()

```
4.5.3   TCS_GetRGBxBrtns()



INFO

 Returns the brightness of an RGBx pixel.

SYN

 brtns = TCS_GetRGBxBrtns(RGBxPxl, RGBxID)

 d0.b                    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

 - brtns is a _theorical_ value, *not* the _actual_ pixel brightness as it
   appears on the screen (RGBx modes can’t fully exploit the brightness
   available - see the RGB <-> RGBx issue for details)
```

1.43 4.5.4 TCS_CnvRGB()

```
4.5.4   TCS_CnvRGB()



INFO

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

SYN

 RGBxPxl = TCS_CnvRGB(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))
 - only RGBW/RGB332 supported in this version
```

1.44 4.5.5 TCS_CnvRGBx()

```
                4.5.5   TCS_CnvRGBx()



INFO

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

SYN

 RGBPxl = TCS_CnvRGBx(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!)
```

1.45 4.5.6 TCS_MkRGBxCnvTab()

```
4.5.6   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
```

1.46 4.5.8 TCS_FlgDXPfldCols()

```
4.5.8   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=0: x is a good-looking color
            - FlgsAdr[x].b=-1: x is a bad-looking color
 col       transparent RGBx color

NOTE

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

1.47 SvIFFRGBxPal()

```
4.5.9   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 other 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 disk activity!
 - existing files will be overwritten
```

1.48 4.6 Functions for Graphics

```
               4.6   Functions for Graphics



The following functions are quickly accessible graphic primitives to draw
graphics on (logical) screens:

 4.6.1
               Hi-Level Functions
                4.6.2
               Low-Level Functions
                4.6.3
               Special Functions


 - these functions are as much general as possible (and, anyway, do not
   pretend to be the fastest in the world), so for better performance it’s
   recommendable writing custom/specific routines
 - never pass negative values (unless differently specified)
 - try to keep coordinates below 1024 (precise limitations will be given
   for each function as soon as possible)
```

1.49 4.6.1 Hi-Level Graphic Functions

```
               4.6.1   Hi-Level Graphic Functions



These are the most general functions available; they will act accordingly
to the Graphic Context of the display of the screen they are applied to:

 4.6.1.1
               TCS_PltPxl()
                4.6.1.2
               TCS_DrwLn()
                4.6.1.3
               TCS_DrwHrzLn()
                4.6.1.4
               TCS_DrwVrtLn()
                4.6.1.5
               TCS_DrwSqr()
                4.6.1.6
               TCS_DrwFrm()
                4.6.1.7
               TCS_DrwTrngl()
```

- the simpler the function, the heavier the overhead!

1.50 4.6.1.1 TCS_PltPxl()

4.6.1.1 TCS_PltPxl()

INFO

Plots a pixel on a logical screen.

SYN

```
 TCS_PltPxl(DIAdr, x,   y,   col)

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

IN

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

NOTE

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

1.51 4.6.1.2 TCS_DrwLn()

```
                 4.6.1.2   TCS_DrwLn()



 INFO

  Draws a line on a logical screen.

 SYN

  TCS_DrwLn(DIAdr, x0,   y0, x1,  y1,  col)

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

 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

 NOTE

  - Graphic Contexts supported: normal, clipping
```

1.52 4.6.1.3 TCS_DrwHrzLn()

```
                 4.6.1.3   TCS_DrwHrzLn()



 INFO

  Draws a horizontal line on a logical screen.

 SYN

  TCS_DrwHrzLn(DIAdr, x0,  x1,  y,   col)

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

 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

 NOTE
 - Graphic Contexts supported: normal, clipping
```

1.53 4.6.1.4 TCS_DrwVrtLn0()

```
                 4.6.1.4   TCS_DrwVrtLn()



INFO

 Draws a vertical line on a logical screen.

SYN

 TCS_DrwVrtLn(DIAdr, y0,   y1,  x,   col)

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

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

NOTE

 - Graphic Contexts supported: normal, clipping
```

1.54 4.6.1.5 TCS_DrwSqr()

```
                 4.6.1.5   TCS_DrwSqr()



INFO

 Draws a square on a logical screen.

SYN

 TCS_DrwSqr(DIAdr, x,   y,   SideLen, col)

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

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

NOTE

 - Graphic Contexts supported: normal, clipping, filling
```

1.55 4.6.1.6 TCS_DrwFrm()

```
                 4.6.1.6   TCS_DrwFrm()



INFO

 Draws a rectangle on a logical screen.

SYN

 TCS_DrwFrm(DIAdr, x0,  y0,  x1,  y1,  col)

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

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

NOTE

 - Graphic Contexts supported: normal, clipping, filling
```

1.56 4.6.1.7 TCS_DrwTrngl()

```
                 4.6.1.7   TCS_DrwTrngl()



INFO

 Draws a triangle on a logical screen.

SYN

 TCS_DrwTrngl(DIAdr, VrtxsAdr, col)

              a0.l   a1.l      d0.b
IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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

NOTE

 - Graphic Contexts supported: normal, clipping, filling
```

1.57 4.6.1.8 TCS_DrwPlgn()

```
                 4.6.1.8   TCS_DrwPlgn()



 INFO

  Draws a closed polygon on a logical screen.

 SYN

  TCS_DrwPlgn(DIAdr, VrtxsAdr, col)

              a0.l   a1.l      d0.b

 IN

  DIAdr      screen display
                DisplayInfo structure
                 pointer
  VrtxsAdr   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

 NOTE

  - Graphic Contexts supported: normal, clipping [, filling TBD]
  - 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 one vertex defined in the list!
```

1.58 4.6.1.9 TCS_DrwOpnPlgn()

```
                 4.6.1.9   TCS_DrwOpnPlgn()
```

```
INFO

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

SYN

 TCS_DrwOpnPlgn(DIAdr, VrtxsAdr, col)

                a0.l   a1.l      d0.b

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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

NOTE

 - Graphic Contexts supported: normal, clipping
 - there *must* be at least one vertex defined in the list!
```

1.59 4.6.1.10 TCS_DrwCrcl()

```
                4.6.1.10   TCS_DrwCrcl()



INFO

 Draws a circle on a logical screen.

SYN

 TCS_DrwCrcl(DIAdr, cx,  cy,  rad, col)

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

IN

 DIAdr   screen display
               DisplayInfo structure
                pointer
 cx,cy   coordinates of the circle centre
 rad     circle radius length
 col     color value in RGBx format

NOTE
```

```
  - Graphic Contexts supported: normal, clipping, filling
```

1.60 4.6.1.12 TCS_FillArea()

```
                4.6.1.12   TCS_FillArea()



 INFO

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

 SYN

  TCS_FillArea(DIAdr, x,   y,   col)

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

 IN

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

 NOTE

  - Graphic Contexts supported: normal, clipping
  - 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). Generally this figure is
    much smaller and depends 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 re-
    quired if starting from the centre). Note that a better memory usage
    means also more speed (and not just the time spared for writes)
```

1.61 4.6.1.13 TCS_ClrScr()

```
                4.6.1.13   TCS_ClrScr()



 INFO

  Clears with a given RGBx color a logical screen.

 SYN
  TCS_ClrScr(DIAdr, col)

             a0.l   d0.b

 IN

  DIAdr   screen display
                DisplayInfo structure
                pointer
  col     RGBx value of the color the screen has to be cleared with

 NOTE

  - Graphic Contexts supported: normal, clipping
```

1.62 4.6.2 Low-Level Graphic Functions

```
                4.6.2   Low-Level Graphic Functions



To avoid the overhead of the
               Hi-Level Graphic Functions
               , you can use the
the following functions, which don't take into account the Graphic Context
of the screen they're used onto (which, instead, must be expressely speci-
fied through the name of the function itself - see footnote):

 4.6.2.1a
                TCS_PltPxl0()
                 4.6.2.1b
                TCS_PltPxl1()
                 4.6.2.2a
                TCS_DrwLn0()
                 4.6.2.2b
                TCS_DrwLn1()
                 4.6.2.2c
                TCS_DrwHrzLn0()
                 4.6.2.2d
                TCS_DrwHrzLn1()
                 4.6.2.2e
                TCS_DrwVrtLn0()
                 4.6.2.2f
                TCS_DrwVrtLn1()
                 4.6.2.3a
                TCS_DrwSqr0()
                 4.6.2.3b
                TCS_DrwSqr1()
                 4.6.2.3c
                TCS_DrwSqr2()
                 4.6.2.3d
                TCS_DrwSqr3()
                 4.6.2.4a
                TCS_DrwFrm0()
                 4.6.2.4b
                TCS_DrwFrm1()
                 4.6.2.4c
                TCS_DrwFrm2()
                 4.6.2.4d
                TCS_DrwFrm3()
                 4.6.2.5a
                TCS_DrwTrngl0()
                 4.6.2.5b
                TCS_DrwTrngl1()
                 4.6.2.5c
                TCS_DrwTrngl2()
                 4.6.2.5d
                TCS_DrwTrngl3()
                 4.6.2.6a
                TCS_DrwPlgn0()
                 4.6.2.6b
                TCS_DrwPlgn1()
                 4.6.2.6c
                TCS_DrwPlgn2()
                (UNAVAILABLE)
 4.6.2.6d
                TCS_DrwPlgn3()
                (UNAVAILABLE)
 4.6.2.6e
                TCS_DrwOpnPlgn0()
                 4.6.2.6f
                TCS_DrwOpnPlgn1()
                 4.6.2.7a
                TCS_DrwCrcl0()
                 4.6.2.7b
                TCS_DrwCrcl1()
                 4.6.2.7c
                TCS_DrwCrcl2()
                 4.6.2.7d
                TCS_DrwCrcl3()
                 4.6.2.8a   TCS_DrwElps0()     (UNAVAILABLE)
 4.6.2.8b   TCS_DrwElps1()     (UNAVAILABLE)
 4.6.2.8c   TCS_DrwElps2()     (UNAVAILABLE)
 4.6.2.8d   TCS_DrwElps2()     (UNAVAILABLE)

 4.6.2.9a
                TCS_FillArea0()
                 4.6.2.9b
                TCS_FillArea1()
                 4.6.2.10a
                TCS_ClrScr0()
                 4.6.2.10b
                TCS_ClrScr1()
```

- the low-level function "TCS_FncNmX()" performs the same operation of the hi-level "TCS_FncNm()", except that the Graphic Context is specified by the ’X’, according to the Graphic Context flags

```
             : thus TCS_PltPxl0()
plots a pixel without any particular operation, whereas TCS_PltPxl1()
plots a pixel considering the
            Clipping Window
             you pass to it as an ar-
gument (this is because TCS_GCf_clp=1); analogously, TCS_DrwFrm3() will
draw a frame taking into account the clipping limitations and filling
the rectangle (TCS_GCf_clp=1 + TCS_GCf_fill=2 = 3).
```

1.63 4.6.2.1a TCS_PltPxl0()

```
                 4.6.2.1a   TCS_PltPxl0()



INFO

 Plots a pixel on a logical screen.

SYN

 TCS_PltPxl0(DIAdr, x,   y,   col)

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

IN

 DIAdr   screen display
               DisplayInfo structure
                pointer
 x,y     coordinates of the pixel
 col     color value in RGBx format

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, anyway, it is *absolutely granted* that this function will
   trash d1 *only* (i.e. all the other registers are left unmodified)
 - x is declared as .l for speed's sake (to avoid an "ext.l")
```

1.64 4.6.2.1b TCS_PltPxl1()

```
                 4.6.2.1b   TCS_PltPxl1()



INFO

 Plots a pixel on a logical screen with clipping.

SYN
```

```
 TCS_PltPxl1(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, anyway, it is *absolutely granted* that this function will
   trash d1 *only* (i.e. all the other registers are left unmodified)
 - x is declared as .l because it must have the upper word clean
```

1.65 4.6.2.2a TCS_DrwLn0()

```
                4.6.2.2a   TCS_DrwLn0()



INFO

 Draws a line on a logical screen.

SYN

 TCS_DrwLn0(DIAdr, x0,   y0, x1,  y1,  col)

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

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
```

1.66 4.6.2.2b TCS_DrwLn1()

```
                4.6.2.2b   TCS_DrwLn1()
```

```
INFO

 Draws a line on a logical screen with clipping.

SYN

 TCS_DrwLn1(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
```

1.67 4.6.2.2c TCS_DrwHrzLn0()

```
                4.6.2.2c   TCS_DrwHrzLn0()



INFO

 Draws a horizontal line on a logical screen.

SYN

 TCS_DrwHrzLn0(DIAdr, x0,  x1,  y,   col)

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

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
```

1.68 4.6.2.2d TCS_DrwHrzLn1()

```
                4.6.2.2d   TCS_DrwHrzLn1()
```

```
INFO

 Draws a horizontal line on a logical screen with clipping.

SYN

 TCS_DrwHrzLn1(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
```

1.69 4.6.2.2e TCS_DrwVrtLn0()

```
                4.6.2.2e   TCS_DrwVrtLn0()



INFO

 Draws a vertical line on a logical screen.

SYN

 TCS_DrwVrtLn0(DIAdr, y0,   y1,  x,   col)

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

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
```

1.70 4.6.2.2f TCS_DrwVrtLn1()

```
                4.6.2.2f   TCS_DrwVrtLn1()
```

```
INFO

 Draws a vertical line on a logical screen with clipping.

SYN

 TCS_DrwVrtLn1(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
```

1.71 4.6.2.3a TCS_DrwSqr0()

```
                4.6.2.3a   TCS_DrwSqr0()



INFO

 Draws an empty square on a logical screen.

SYN

 TCS_DrwSqr0(DIAdr, x,   y,   SideLen, col)

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

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
```

1.72 4.6.2.3b TCS_DrwSqr1()

```
                4.6.2.3b   TCS_DrwSqr1()
```

```
INFO

 Draws an empty square on a logical screen with clipping.

SYN

 TCS_DrwSqr1(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
```

1.73 4.6.2.3c TCS_DrwSqr2()

```
                4.6.2.3c   TCS_DrwSqr2()



INFO

 Draws a filled square on a logical screen.

SYN

 TCS_DrwSqr2(DIAdr, x,   y,   SideLen, col)

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

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
```

1.74 4.6.2.3d TCS_DrwSqr3()

```
                4.6.2.3d   TCS_DrwSqr3()
```

```
 INFO

  Draws a filled square on a logical screen with clipping.

 SYN

  TCS_DrwSqr3(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
```

1.75 4.6.2.4a TCS_DrwFrm0()

```
                 4.6.2.4a   TCS_DrwFrm0()



 INFO

  Draws an empty rectangle on a logical screen.

 SYN

  TCS_DrwFrm0(DIAdr, x0,  y0,  x1,  y1,  col)

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

 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
```

1.76 4.6.2.4b TCS_DrwFrm1()

```
                 4.6.2.4b   TCS_DrwFrm1()
```

```
INFO

 Draws an empty rectangle on a logical screen with clipping.

SYN

 TCS_DrwFrm1(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
```

1.77 4.6.2.4c TCS_DrwFrm2()

```
                4.6.2.4c   TCS_DrwFrm2()



INFO

 Draws a filled rectangle on a logical screen.

SYN

 TCS_DrwFrm2(DIAdr, x0,  y0,  x1,  y1,  col)

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

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
```

1.78 4.6.2.4d TCS_DrwFrm3()

```
                4.6.2.4d   TCS_DrwFrm3()
```

```
INFO

 Draws a filled rectangle on a logical screen with clipping.

SYN

 TCS_DrwFrm3(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
```

1.79 4.6.2.5a TCS_DrwTrngl0()

```
                4.6.2.5a   TCS_DrwTrngl0()



INFO

 Draws an empty triangle on a logical screen.

SYN

 TCS_DrwTrngl0(DIAdr, VrtxsAdr, col)

               a0.l   a1.l      d0.b

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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
```

1.80 4.6.2.5b TCS_DrwTrngl1()

```
                4.6.2.5b   TCS_DrwTrngl1()
```

```
INFO

 Draws an empty triangle on a logical screen with clipping.

SYN

 TCS_DrwTrngl1(DIAdr, VrtxsAdr, col, ClpWin)

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

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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
```

1.81 4.6.2.5c TCS_DrwTrngl2()

```
                4.6.2.5c   TCS_DrwTrngl2()



INFO

 Draws a filled triangle on a logical screen.

SYN

 TCS_DrwTrngl2(DIAdr, VrtxsAdr, col)

               a0.l   a1.l      d0.b

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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

NOTE

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

1.82 4.6.2.5d TCS_DrwTrngl3()

```
                4.6.2.5d   TCS_DrwTrngl3()



 INFO

  Draws a filled triangle on a logical screen with clipping.

 SYN

  TCS_DrwTrngl3(DIAdr, VrtxsAdr, col, ClpWin)

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

 IN

  DIAdr      screen display
                DisplayInfo structure
                 pointer
  VrtxsAdr   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

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

1.83 4.6.2.6a TCS_DrwPlgn0()

```
                4.6.2.6a   TCS_DrwPlgn0()



 INFO

  Draws a closed polygon on a logical screen.

 SYN

  TCS_DrwPlgn0(DIAdr, VrtxsAdr, col)

              a0.l   a1.l      d0.b

 IN

  DIAdr      screen display
                DisplayInfo structure
                 pointer
  VrtxsAdr   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

 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 one vertex defined in the list!
```

1.84 4.6.2.6b TCS_DrwPlgn1()

```
                 4.6.2.6b   TCS_DrwPlgn1()



INFO

 Draws a closed polygon on a logical screen with clipping.

SYN

 TCS_DrwPlgn1(DIAdr, VrtxsAdr, col, ClpWin)

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

IN

 DIAdr       screen display
                DisplayInfo structure
                 pointer
 VrtxsAdr    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 one vertex defined in the list!
```

1.85 4.6.2.6c TCS_DrwPlgn2()

```
                 4.6.2.6c   TCS_DrwPlgn2()



INFO
```

```
 Draws a filled polygon on a logical screen.

SYN

 TCS_DrwPlgn2(DIAdr, VrtxsAdr, SideCol, FillCol)

              a0.l   a1.l      d0.b     d1.b

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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
 SideCol    color value in RGBx format for the sides
 FillCol    color value in RGBx format for filled area

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 one vertex defined in the list!
```

1.86 4.6.2.6d TCS_DrwPlgn3()

```
                 4.6.2.6d   TCS_DrwPlgn3()



INFO

 Draws a filled polygon on a logical screen with clipping.

SYN

 TCS_DrwPlgn3(DIAdr, VrtxsAdr, SideCol, FillCol, ClpWin)

              a0.l   a1.l      d0.b     d1.b     a3.l

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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
 SideCol    color value in RGBx format for the sides
 FillCol    color value in RGBx format for filled area
 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 one vertex defined in the list!
```

1.87 4.6.2.6e TCS_DrwOpnPlgn0()

```
                 4.6.2.6e   TCS_DrwOpnPlgn0()



 INFO

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

 SYN

  TCS_DrwOpnPlgn0(DIAdr, VrtxsAdr, col)

                  a0.l   a1.l      d0.b

 IN

  DIAdr      screen display
                DisplayInfo structure
                 pointer
  VrtxsAdr   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

 NOTE

  - there *must* be at least one vertex defined in the list!
```

1.88 4.6.2.6f TCS_DrwOpnPlgn1()

```
                 4.6.2.6f   TCS_DrwOpnPlgn1()



 INFO

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

 SYN

  TCS_DrwOpnPlgn1(DIAdr, VrtxsAdr, col, ClpWin)
```

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

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 VrtxsAdr   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!
```

1.89 4.6.2.7a TCS_DrwCrcl0()

```
                4.6.2.7a   TCS_DrwCrcl0()



INFO

 Draws an empty circle on a logical screen.

SYN

 TCS_DrwCrcl0(DIAdr, cx,  cy,  rad, col)

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

IN

 DIAdr   screen display
               DisplayInfo structure
                pointer
 cx,cy   coordinates of the circle centre
 rad     circle radius length
 col     color value in RGBx format
```

1.90 4.6.2.7b TCS_DrwCrcl1()

```
                4.6.2.7b   TCS_DrwCrcl1()



INFO

 Draws an empty circle on a logical screen with clipping.
```

```
SYN

 TCS_DrwCrcl1(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
 col      color value in RGBx format
 ClpWin   pointer to
               ClippingWindow structure
```

1.91 4.6.2.7c TCS_DrwCrcl2()

```
                4.6.2.7c   TCS_DrwCrcl2()



INFO

 Draws a filled circle on a logical screen.

SYN

 TCS_DrwCrcl2(DIAdr, cx,  cy,  rad, col)

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

IN

 DIAdr   screen display
               DisplayInfo structure
                pointer
 cx,cy   coordinates of the circle centre
 rad     circle radius length
 col     color value in RGBx format
```

1.92 4.6.2.7d TCS_DrwCrcl3()

```
                4.6.2.7d   TCS_DrwCrcl3()



INFO

 Draws a filled circle on a logical screen with clipping.
```

```
SYN

 TCS_DrwCrcl3(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
 col      color value in RGBx format
 ClpWin   pointer to
               ClippingWindow structure
```

1.93 4.6.2.9a TCS_FillArea0()

```
                4.6.2.9a   TCS_FillArea0()



INFO

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

SYN

 TCS_FillArea0(DIAdr, x,   y,   col)

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

IN

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

NOTE

 - BE CAREFUL! The screen edges are not considered as limits! You should
   use
               TCS_FillArea1()
                is you are unsure or need clipping!
 - 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). Generally this figure is
   much smaller and depends 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 re-
   quired if starting from the centre). Note that a better memory usage
   means also more speed (and not just the time spared for writes)
 - best performance when the sp is longword aligned
```

1.94 4.6.2.9b TCS_FillArea1()

```
                 4.6.2.9b   TCS_FillArea1()



 INFO

  Fills an area of a logical screen with a given RGBx color accoding to the
  given clipping restrictions.

 SYN

  TCS_FillArea1(DIAdr, x,    y,    col, ClpWin)

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

 IN

  DIAdr    screen display
                DisplayInfo structure
                 pointer
  x,y      x 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

  - due to the clipping checks this function is slower than
                TCS_FillArea0()
                  - refer to
                TCS_FillArea0()
                 for more information
```

1.95 4.6.2.10a TCS_ClrScr0()

```
                 4.6.2.10a   TCS_ClrScr0()



 INFO

  Clears with a given RGBx color a logical screen.

 SYN
 TCS_ClrScr0(DIAdr, col)

             a0.l   d0.b
```

IN

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

1.96 4.6.2.10b TCS_ClrScr1()

4.6.2.10b   TCS_ClrScr1()

INFO

Clears with a given RGBx color a logical screen with clipping.

SYN

```
 TCS_ClrScr1(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
```

1.97 4.6.3 Special Graphic Functions

4.6.3   Special Graphic Functions

The following functions work on complex graphic data like pixmaps are not affected by the Graphic Context of the display:

```
 4.6.3.1
                TCS_CpyScr()
                 4.6.3.2a
                TCS_FitTxtr1()
                 4.6.3.2b
                TCS_FitTxtr2()
                 4.6.3.2c
```

```
                  TCS_FitTxtr4()
                   4.6.3.4   TCS_RotZmTxtr()  (UNAVAILABLE)
```

1.98 4.6.3.1 TCS_CpyScr()

```
                  4.6.3.1   TCS_CpyScr()



 INFO

  Copies a logical screen to another logical screen.

 SYN

  success = TCS_CpyScr(SouDIAdr, DstDIAdr)

  ccr                  a0.l      a1.l

 IN

  SouDIAdr   source screen display
                DisplayInfo structure
                 pointer
  DstDIAdr   destination screen display
                DisplayInfo structure
                 pointer

 OUT

  success    ne = screen copied successfully
             eq = error

 NOTE

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

1.99 4.6.3.2a TCS_FitTxtr1()

```
                  4.6.3.2a   TCS_FitTxtr1()



 INFO

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

 SYN
  TCS_FitTxtr1(DIAdr, TxtrAdr, TxtrWd, VrtxsAdr)

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

 IN

  DIAdr      screen display
                DisplayInfo structure
                 pointer
  TxtrAdr    address of texture's top-left corner
  TxtrWd     width of textures in pixels (bytes)
  VrtxsAdr   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

 NOTE

  - see also
                TCS_FitTxtr2()
                 and
                TCS_FitTxtr4()
```

1.100 4.6.3.2b TCS_FitTxtr2()

```
                4.6.3.2b   TCS_FitTxtr2()



INFO

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

SYN

 TCS_FitTxtr2(DIAdr, TxtrAdr, TxtrWd, VrtxsAdr)

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

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 TxtrAdr    address of texture's top-left corner
 TxtrWd     width of textures in pixels (bytes)
 VrtxsAdr   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

NOTE

 - the width of the destination rectangle (dx1-dx0+1) should be even for
   correct horizontal scaling (automatic rounding is always performed)
 - faster than
               TCS_FitTxtr1()
                 - best performance when dx0 is even
 - see also
               TCS_FitTxtr1()
                and
               TCS_FitTxtr4()
```

1.101 4.6.3.2c TCS_FitTxtr4()

```
                 4.6.3.2c   TCS_FitTxtr4()



INFO

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

SYN

 TCS_FitTxtr4(DIAdr, TxtrAdr, TxtrWd, VrtxsAdr)

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

IN

 DIAdr      screen display
               DisplayInfo structure
                pointer
 TxtrAdr    address of texture's top-left corner
 TxtrWd     width of textures in pixels (bytes)
 VrtxsAdr   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 destination rectangle
12,14	dx1,dy1:	coordinates of bottom-right pixel of destination rectangle

NOTE

- the width of the destination rectangle (dx1-dx0+1) should be multiple of 4 for correct horizontal scaling (automatic rounding is always performed)
- faster than
  TCS_FitTxtr1()
  and
  TCS_FitTxtr2()
  - best performance when dx0 is multiple of 4
- see also
  TCS_FitTxtr1()
  and
  TCS_FitTxtr2()

1.102 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:

4.7.1 TCS_LdRGBx() (UNAVAILABLE)
4.7.2 TCS_UnLdRGBx() (UNAVAILABLE)
4.7.3 TCS_SvRGBx() (UNAVAILABLE)
4.7.4 TCS_SvScr2RGBx() (UNAVAILABLE)
4.7.5
TCS_LdILBM()
4.7.6
TCS_UnLdILBM()
4.7.7 TCS_SvILBM() (UNAVAILABLE)

1.103 4.7.5 TCS_LdILBM()

4.7.5 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   destination buffer address for raster data (pass 0 if you want
          the function to allocate it for you)
 BufLen   size in bytes of destination buffer (only if BufAdr<>0)

OUT

 ILBMStruc   pointer to an
               ILBMInfo structure
                or a TCS_PE_xxx 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!)
 - 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-bitplanes, compressed, non-masked, non-HAM, ILBMs
   supported (unsupported 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 with
               TCS_UnLdILBM()
                 - make sure to pass correct values for BufAdr and BufSz!
 - make sure the AmigaOS is ON and don't call from interrupts (because of
   disk activity and memory allocation).
```

1.104 4.7.6 TCS_UnLdILBM()

```
                4.7.6   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
```

1.105 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

1.106 4.9 Known Bugs

4.9   Known Bugs

I knew sooner or later I’d have to write this section, but after almost 2 years of coding I could not imagine it would have been for such a stupid reason... anyway, the only bug I know of is that 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...?"