Display stream package

A library package is now available which provides a capability for display streams with considerably more flexibility than the current operating system streams. Additional features include multiple fonts, repositioning to any bit position in the current line (or, under proper circumstances, any line), selective erasing and polarity inversion, and better utilization of the available bitmap space.

The package consists of two files, DSTREAM and DHANX. In addition, the file DISP.D provides useful parameter and structure declarations, in particular the parameters lDCB and lDS mentioned below. DSTREAM is written in Bcpl and occupies about 2.4K (octal). DHANX is written in assembly language and occupies about .1K (octal). The package does not require any routines other than those in the operating system.

1. Creating a display stream

CreateDstream(v, nl, ds, options)
creates a display stream. V is the same 4-word vector currently used for creating display streams through the operating system (see section 3.2.3 of the O.S. manual), but, unlike the O.S., CreateDstream requires that v be supplied and that v!0 and v!1 delimit a storage area. Nl is the number of lines for the stream: it is completely independent of the amount of space supplied for bitmap and DCBs. Ds is a pointer to a block of lDS words which will be used to store the stream itself. If ds is omitted, CreateDstream will try to obtain such a block from the O.S. using CREATES(v, DISPLAYOPEN): this procedure is not recommended in view of the small number of display streams provided by the O.S. The value returned by CreateDstream is the stream (ds if ds was supplied).

+3

LDCB = 4

The minimum space for a display stream is lDCB*nl+fh*nwrds, where fh is the height of the standard system font and nwrds is v!3 if v!3 is non-zero, otherwise 38. This, however, only provides enough bitmap for a single line. A space allocation of lDCB*nl+fh*nwrds*nl guarantees enough bitmap for all nl lines. The display stream package uses all the available space and then, if necessary, deletes lines starting from the top to make room for new data.

Options, if supplied, controls the action of the stream under various exceptional conditions. The various options have mnemonic names (defined in DISP.D) and may be added together. Here is the list of options:

DScompactleft: allows the bitmap space required for a line to be reduced when scrolling by eliminating multiples of 16 initial blank bit positions and replacing them with the display controller's "tab" feature. However, a line in which this has occurred may not be overwritten later (with SetLinePos, see below).

DScompactright: allows the bitmap space for a line to be reduced when scrolling by eliminating multiples of 16 blank bit positions on the right. Overwriting is allowed up to the beginning of the blank space, i.e. you cannot make a line longer by overwriting if you select this option.

DSstopright: causes characters to be discarded when a line becomes full, rather than scrolling onto a new line.

DSstopbottom: causes characters to be discarded in preference to

losing data from the screen. This applies when either all nl lines are occupied, or when the allocated bitmap space becomes full.
If options is not supplied, it defaults to DScompactleft+DScompactright.

2. Current-line operations

GetFont(ds)
returns the current font of ds.

SetFont(ds, pfont)
changes the font of the display stream ds. Pfont is a pointer to word 2 of a font, which is compatible with GetFont. Characters which have been written into the stream already are not affected; future characters will be written in the new font. If the font is higher than the font initially specified, writing characters may cause unexpected alteration of lines other than the line being written into.

GetBitPos(ds)
returns the bit position in the current line. The bit position is normally initialized to 8.

SetBitPos(ds, pos)
sets the bit position in the current line to pos and returns true, if pos is not too large; otherwise, returns false. Pos must be less than 606 (the display width) minus the width of the widest character in the current font. Resetting the bit position does not affect the bitmap; characters displayed at overlapping positions will be "or"ed in the obvious manner.

GetLinePos(ds)
returns the line number of the current line. Unlike the present operating system display streams, which always write into the bottom line and scroll up, the display streams provided by this package start with the top line and only scroll when they reach the bottom.

EraseBits(ds, nbits, flag)
changes bits in ds starting from the current position. Flag=0, or flag omitted, means set bits to 0 (same as background); flag=1 means set bits to 1 (opposite from background); flag=-1 means invert bits from their current state. If nbits is positive, the affected bits are those in positions pos through pos+nbits-1, where pos is GetBitPos(ds); if nbits is negative, the affected positions are pos+nbits through pos-1. In either case, the final position of the stream is pos+nbits.

Here are two examples of the use of EraseBits. If the last character written on ds was ch, EraseBits(ds, -CharWidth(ch, GetFont(ds))) will erase it and back up the current position (see below for CharWidth). If a word of width ww has just been written on ds, EraseBits(ds, -ww, -1) will change it to white-on-black.

3. Inter-line operations

GetLinePos(ds)
returns the current line position within ds. The top line in the stream is numbered 0.

SetLinePos(ds, pos)

sets the current line position in ds to pos. If the line has not yet been written into, or if it has zero width, or if it is indented as the result of compacting on the left, SetLinePos has no effect and returns false; otherwise, SetLinePos returns true. Note that if you want to get back to where you were before, you must remember where that was (using GetLinePos and GetBitPos).

4. Scrolling

The display stream package writes characters using a very fast assembly language routine until either the current line is full or it encounters a control character. In either of these situations it calls a scrolling procedure whose address is a component of the stream. The scrolling procedure is called with the same arguments as PUTS, i.e. (ds, char), and is expected to do whatever is necessary. The standard procedure takes the following action:

1) Null (code 0) is ignored.
2) New line (code 15b) causes scrolling.
3) Tab (code 11b) advances the bit position to the next multiple of 8 times the width of "blank" (code 40b) in the current font: if this would exceed the right margin, just puts out a blank.
4) Other control characters (codes 1-10b, 12b-14b, 16b-37b) print as "↑" followed by their letter equivalent.
5) If a character will not fit on the current line, scrolling occurs and the character is printed at the beginning of the new line (unless the DSstopright option was chosen, in which case the character is simply discarded).

The scrolling procedure is also called with arguments (ds, -1) whenever a contemplated scrolling operation would cause information to disappear from the screen, either because nl lines are already present or because the bitmap space is full (unless the DSstopbottom option was chosen, in which case the procedure is not called and the action is the same as if it had returned false). If the procedure returns true, the scrolling operation proceeds normally. If the procedure returns false, the scrolling does not take place, and the character which triggered the operation is discarded.

The user may supply a different scrolling procedure simply by filling it into the field ds
DS.scroll.

5. Miscellaneous

GetLmarg(ds)
returns the left margin position of ds. The left margin is initialized to 8 (about 1/10" from the left edge of the screen).

SetLmarg(ds, pos)
sets the left margin of ds to pos.

GetRmarg(ds)
returns the right margin position of ds. The right margin is initialized to the right edge of the screen: this is the value of the displaywidth parameter in DISP.D.

SetRmarg(ds, pos)
sets the right margin of ds to pos.

ResetLine(ds)
is equivalent to EraseBits(ds, GetLmarg(ds)-GetBitPos(ds)), i.e. it erases the current line and resets to the left margin.

CharWidth(char, pfont)
returns the width of the character char in the font pfont.

```
// New display streams
// last edited 31 aug 75 17:00

        get "disp.d"

external        // entries
[       CharWidth       // (char, font) -> width
        ResetLine       // (ds)
        CreateDstream   // (v, nl, ds[, options]) -> ds
        DPUT    // (ds, char), in DHANX.A
        GetFont // (ds) -> font
        SetFont // (ds, font)
        GetBitPos       // (ds) -> pos
        SetBitPos       // (ds, pos)
        GetLmarg        // (ds) -> pos
        SetLmarg        // (ds, pos)
        GetRmarg        // (ds) -> pos
        SetRmarg        // (ds, pos)
        GetLinePos      // (ds) -> lpos
        SetLinePos      // (ds, lpos) -> true/false
        Scroll  // (ds[, char])
        EraseBits       // (ds, nbits[, flag])
]

external        // O.S.
[       BMOVE; BSTORE
        CREATES; PUTS
        DSP
        SWAT
]
manifest
[       leftmargin = 8
        rightmargin = displaywidth
]


let CharWidth(char, font) = valof
[       let w, cw = 0, nil
         [ cw = font!(font!charÃchar)
           if (cw & 1) ne 0 then break
           w, char = w+16, cw rshift 1
         ] repeat
        resultis w + cw rshift 1
]

and ResetLine(ds) be
[       SetBitPos(ds, ds>>DS.rmarg)
        EraseBits(ds, ds>>DS.lmarg-ds>>DS.rmarg)
]

and CreateDstream(v, nl, ds, options; numargs na) = valof
```

```
// v is the same parameter as for the O.S. CREATES call
// nl is the number of lines desired
// ds is storage for the stream (1DS words)
[       v!0 = (v!0+1)&(not 1)
        v!1 = (v!1)&(not 1)
        BMOVE(DSP, ds, 1DS-1)
        ds>>DS.ssa, ds>>DS.esa = v!0, v!1
        ds>>DS.pfont = (v!2 eq 0? DSP>>DS.pfont, v!2)
        ds>>DS.nwrds = (v!3 eq 0? (displaywidth+31)/16, (v!3+1))&(-2)
        let ssa, esa = ds>>DS.ssa, ds>>DS.esa
        let wpsl = ds>>DS.nwrds
        let ht = (ds>>DS.pfont!(-2)+1) rshift 1
        let bsz = wpsl*ht*2
        if esa-ssa ls nl*1DCB+bsz then SWAT("Dstream too small")
        ds>>DS.puts = DPUT
        ds>>DS.opens = linkds
        ds>>DS.closes = unlinkds
        ds>>DS.resets = cleards
        ds>>DS.scroll = Scroll
        let edcb = ssa+nl*1DCB
        let ldcb = edcb-1DCB
        ds>>DS.fdcb, ds>>DS.ldcb = ssa, ldcb
        ds>>DS.blksz = bsz
        let bda = edcb
        ds>>DS.fmp = esa-bsz
        ds>>DS.bda = bda
        for p = ssa by 1DCB to ldcb do
           p>>DCB.next, p>>DCB.height = p+1DCB, ht
        ldcb>>DCB.next = 0
        ds>>DS.lmarg, ds>>DS.rmarg = leftmargin, rightmargin
        ds>>DS.options = ((na ge 4) & (options ne -1)? options, DScompactleft+DSc
                                                **ompactright)
        SetFont(ds, ds>>DS.pfont)
        cleards(ds)
        resultis ds
]

and cleards(ds) be
[       let fdcb, fmp = ds>>DS.fdcb, ds>>DS.fmp
        for dcb = fdcb by 1DCB to ds>>DS.ldcb do
         [ dcb>>DCB.parwd = 0
           dcb>>DCB.bitmap = fmp
         ]
        ds>>DS.cdcb = fdcb
        ds>>DS.tdcb = fdcb
        fdcb>>DCB.width = ds>>DS.nwrds
        ds>>DS.mwp = ds>>DS.bda
        clearmap(ds)
]

and clearmap(ds) be
[       ds>>DS.cdcb>>DCB.indwidth = ds>>DS.nwrds
        BSTORE(ds>>DS.fmp, 0, ds>>DS.blksz-1)
        SetBitPos(ds, ds>>DS.lmarg)
]

and linkds(ds) be
if prevdcb(ds>>DS.fdcb) eq 0 then
[       ds>>DS.ldcb>>DCB.next = 0
        prevdcb(0)>>DCB.next = ds>>DS.fdcb
]
```

```
and unlinkds(ds) be
[       let pdcb = prevdcb(ds>>DS.fdcb)
        if pdcb ne 0 then pdcb>>DCB.next = ds>>DS.ldcb>>DCB.next
]

and prevdcb(dcb) = valof
[       let org = DCBchainHead-(offset DCB.next/16)
        while org>>DCB.next ne dcb do
         [ if org eq 0 then resultis 0
           org = org>>DCB.next
         ]
        resultis org
]

and GetFont(ds) = ds>>DS.pfont

and SetFont(ds, pfont) = valof
[       let ht = (pfont!(-2)+1) rshift 1
        ds>>DS.pfont = pfont
        SetRmarg(ds, ds>>DS.rmarg)
        resultis ht le ds>>DS.cdcb>>DCB.height
]

and GetBitPos(ds) = ds>>DS.bsofar

and SetBitPos(ds, pos) = valof
[       ds>>DS.bsofar = pos
        ds>>DS.dba = (not pos) & #17
        let cdcb = ds>>DS.cdcb
        ds>>DS.bwrds = cdcb>>DCB.width
        ds>>DS.wad = cdcb>>DCB.bitmap-cdcb>>DCB.width+pos rshift 4
        resultis pos le ds>>DS.bstop
]

and GetLmarg(ds) = ds>>DS.lmarg

and SetLmarg(ds, pos) be
[       ds>>DS.lmarg = pos
        SetBitPos(ds, pos)
]

and GetRmarg(ds) = ds>>DS.rmarg

and SetRmarg(ds, pos) be
[       ds>>DS.rmarg = pos
        ds>>DS.bstop = pos-(ds>>DS.pfont!(-1) & #77777)
]

and GetLinePos(ds) = (ds>>DS.cdcb-ds>>DS.fdcb)/lDCB

and SetLinePos(ds, lpos) = valof
[       let dcb = ds>>DS.fdcb+lpos*lDCB
        if dcb gr ds>>DS.ldcb resultis false
        if dcb>>DCB.indentation ne 0 resultis false
        if dcb>>DCB.width eq 0 resultis false
        ds>>DS.cdcb = dcb
        ds>>DS.bwrds = dcb>>DCB.width
        SetBitPos(ds, ds>>DS.bsofar)
        resultis true
]
```

```
and Scroll(ds, char; numargs na) = valof
[       if na ge 2 then switchon char into
         [ case $*N:
              endcase
           case #11:    // tab
            [ let sp8 = CharWidth($*S, ds>>DS.pfont)*8
              if not SetBitPos(ds, (ds>>DS.bsofar/sp8+1)*sp8) then PUTS(ds, $*S)
              resultis char
            ]
           case 0:      // null
              resultis char
           case -1:     // about to lose data
              resultis true
           default:
              test char ls #40
              ifso [ PUTS(ds, $↑); PUTS(ds, char+#100) ]
              ifnot
               [ let rpos = CharWidth(char, ds>>DS.pfont)+ds>>DS.bsofar
                 test rpos gr ds>>DS.rmarg
                 ifnot [ ds>>DS.bstop = rpos; PUTS(ds, char) ]
                 ifso if (ds>>DS.options&DSstopright) eq 0 endcase
               ]
              resultis char
         ]
        unless compact(ds) resultis char
        let cdcb, ldcb = ds>>DS.cdcb, ds>>DS.ldcb
        test cdcb eq ldcb
        ifnot
         [ cdcb = cdcb>>DCB.next
           ds>>DS.cdcb = cdcb
         ]
        ifso
         [ let dcb = ds>>DS.fdcb
           if dcb eq ds>>DS.tdcb then unless freebitmap(ds) resultis char
           while dcb ne ldcb do
            [ BMOVE(dcb+(1DCB+1), dcb+1, 1DCB-2)        // assumes next in word 0
              dcb = dcb+1DCB
            ]
           ds>>DS.tdcb = ds>>DS.tdcb-1DCB
           cdcb>>DCB.indwidth = ds>>DS.nwrds
           cdcb>>DCB.bitmap = ds>>DS.fmp
         ]
        test cdcb>>DCB.bitmap eq ds>>DS.fmp
        ifso clearmap(ds)
        ifnot ResetLine(ds)
        if char ne $*N then PUTS(ds, char)
        resultis char
]
```

case 12B // linefeed

```
and compact(ds) = valof
[       let dcb = ds>>DS.cdcb
        let ht = dcb>>DCB.height*2
        let onw = dcb>>DCB.width
        let nw = ((ds>>DS.options&DScompactright) ne 0? (ds>>DS.bsofar+15) rshift
                                        ** 4, onw)
        let old = dcb>>DCB.bitmap       // = ds>>DS.fmp
        let d = 0
        if (ds>>DS.options&DScompactleft) ne 0 then
         [ while d ne nw do
            [ let p = old+ds>>DS.blksz+d
              for i = 1 to ht do
               [ p = p-onw
                 if @p ne 0 then goto used
               ]
              d = d+1
            ]
used:    ]
        unless (nw eq onw) & (d eq 1) do
           nw, old = (nw-d+1)&(-2), old+d
        let p = getmapspace(ds, nw*ht)
        test p eq 0
        ifso    // not enough room
           dcb>>DCB.indwidth = 0
        ifnot test p eq -1
        ifso    // don't scroll
           resultis false
        ifnot
         [ let new = p
           if nw ne 0 then for i = 1 to ht do
            [ BMOVE(old, new, nw-1)
              old, new = old+onw, new+nw
            ]
           dcb>>DCB.width = nw
           dcb>>DCB.indentation = d
         ]
        dcb>>DCB.bitmap = p
        resultis true
]

and getmapspace(ds, nw) = valof
[       let wp = nil
   [    wp = ds>>DS.mwp
        let rp = ds>>DS.tdcb>>DCB.bitmap
        test wp gr rp
        ifso
         [ if ds>>DS.fmp-wp gr nw break
           ds>>DS.mwp = ds>>DS.bda
           if rp eq ds>>DS.bda then unless freebitmap(ds) resultis -1
         ]
        ifnot
         [ if rp-wp gr nw break
           unless freebitmap(ds) resultis -1
           if rp eq ds>>DS.fmp resultis 0       // not enough room
         ]
   ] repeat
        ds>>DS.mwp = wp+nw
        resultis wp
]
```

```
and freebitmap(ds) = valof
[       if (ds>>DS.options&DSstopbottom) ne 0 resultis false
        unless (ds>>DS.scroll)(ds, -1) resultis false
        let dcb = ds>>DS.tdcb
        ds>>DS.tdcb = dcb+1DCB
        dcb>>DCB.indwidth = 0
        resultis true
]

and EraseBits(ds, nbits, flag; numargs na) = valof
[       if na ls 3 then flag = 0
        let pos = GetBitPos(ds)+nbits
        test nbits ls 0
        ifso
         [ SetBitPos(ds, pos)
           EraseBits(ds, -nbits, flag)
           SetBitPos(ds, pos)
         ]
        ifnot
         [ let cdcb = ds>>DS.cdcb
           let wpsl, ht = cdcb>>DCB.width, cdcb>>DCB.height*2
           while nbits gr 0 do
            [ let map = ds>>DS.wad
              let dba = ds>>DS.dba
              let nb = (nbits gr dba? dba+1, nbits)
              let mask = MaskTab!dba - MaskTab!(dba-nb)
              for i = 1 to ht do
               [ map = map+wpsl
                 @map = (flag eq 0? @map & not mask,
                    flag ls 0? @map xor mask, @map % mask)
               ]
              SetBitPos(ds, ds>>DS.bsofar+nb)
              nbits = nbits-nb
            ]
         ]
        resultis pos
]
```

```
; Alto display handler
; Modified for stream-dependent scrolling
; last edited 10-APR-75 22:00

        .TITL DHANX
        .ENT DPUT


.DALC   SKL=ADCZ# 0,0,SNC
.DALC   SKLE= SUBZ# 0,0,SNC
.DALC   SKE= SUB# 0,0,SZR
.DALC   SKNE=SUB# 0,0,SNR
.DALC   SKGE=ADCZ# 0,0,SZC
.DALC   SKG=SUBZ# 0,0,SZC


; STRUCTURE OF STREAM

PFONT=0         ;POINTER TO FONT
NWRDS=20.       ;WORDS PER SCAN LINE
DBA=21.         ;DESTINATION BIT ADDRESS
WAD=24.         ;DESTINATION WORD ADDRESS
BSOFAR=23.      ;BITS USED SO FAR IN THIS LINE
BSTOP=22.       ;BIT POSITION AT WHICH TO STOP WRITING
SCROLL=25.      ;SCROLLING ROUTINE
SAVAC2=26.      ;TEMP FOR AC2 DURING CONVERT

; STRUCTURE OF TEMPS IN FRAME
LINK=1
ST=2
TEMP=3




;       DPUT(ST, CHAR)
;Writes a character on a display stream.
;All control characters are passed to the scroller.


        .SREL
DPUT:   DPUTC

        .NREL
DPUTC:  STA 3 LINK,2
        STA 0 ST,2
        MOV 0 3         ;USE AC3 TO ADDRESS STREAM
        LDA 0 C377      ;MASK CHARACTER TO 8 BITS
        AND 0 1
        LDA 0 C40       ;CHECK FOR CONTROL CHARACTERS
        SKL 1 0
        JMP PUT0        ;NOT ONE
DSCR:   LDA 0 SCROLL,3
        STA 0 TEMP,2
        MOV 3 0
        LDA 3 LINK,2
        JMP @TEMP,2     ;GO DO SCROLL INSTEAD

DRET:   LDA 3 LINK,2
        JMP 1,3         ;SKIP RETURN

PUT0:   STA 1 TEMP,2
```

```
;TEST IF THE CHARACTER WILL FIT.  BSTOP = NWRDS * 16 - THE MAX. WIDTH
;OF ANY CHARACTER IN THE FONT

        LDA 0 BSOFAR,3
        LDA 1 BSTOP,3
        SKG 1 0
        JMP SCR1
        LDA 1 TEMP,2
        STA 2 SAVAC2,3
        MOV 3 2

PUT1:   LDA 0 WAD,2     ;WORD ADDRESS IN THIS LINE
        LDA 3 PFONT,2   ;POINTER TO FONT
        ADD 1 3 ;POINTS TO CHARACTER SLOT
        CONVERT NWRDS   ;BINGO!
        JMP PUT2        ;CHARACTER HAS AN EXTENSION

        LDA 0 BSOFAR,2  ;NO EXTENSION, AC3=WIDTH, AC1=DBA AND 17B
        ADD 3 0
        STA 0 BSOFAR,2

        SUBZ 3 1 SZC
        JMP PUT3        ;DIDN'T OVERFLOW A WORD BOUNDARY
        ISZ WAD,2       ;INCREMENT WORD ADDRESS
        LDA 0 C20
        ADD 0 1         ;UPDATE DBA, WHICH IS NOW NEGATIVE
PUT3:   STA 1 DBA,2
        LDA 2 SAVAC2,2
        JMP DRET

SCR1:   LDA 1 TEMP,2    ;PICK UP CHARACTER
        JMP DSCR        ;GO SCROLL

PUT2:   ISZ WAD,2       ;HANDLE EXTENSION.  AC3 HAS PSEUDO-CHARACTER CODE
        LDA 0 BSOFAR,2
        LDA 1 C20
        ADD 1 0
        STA 0 BSOFAR,2
        MOV 3 1
        JMP PUT1


C377:   377
C20:    20
C40:    40


        .END
```

```
// Definitions for display utilities
// last edited 30 aug 75 16:10


manifest
[       DCBchainHead = #420
        MouseXLoc = #424
        MouseYLoc = #425
        CursorXLoc = #426
        CursorYLoc = #427
        CursorMap = #431
        CursorMapSize = #20
        MaskTab = #460
        ButtonsLoc = #177030
        KeysLoc = #177034
]
manifest
[       displayheight = 808
        displaywidth = 606
        cursorheight = 16
        cursorwidth = 16
]

structure BUTTONS:      // hardware button data
[       blank bit 8
        keyset bit 5    // complemented
        mouse bit 3     // complemented, button order is 4-1-2
]

structure DCB:  // display control block
[       next word
         [ resolution bit 1
           background bit 1
            [ indentation bit 6
              width bit 8
            ] = indwidth bit 14
         ] = parwd word
        bitmap word
        height word
]
manifest lDCB = (size DCB)/16

structure DS:   // display stream (modified - see DHANX.A)
```

```
// numbered entries are used by assembly code
// starred entries are modified
[       pfont word      // 0, pointer to font
        opens word      // OPENS
        closes word     // CLOSES
        gets word       // GETS
        puts word       // PUTS
        resets word     // RESETS
        putback word    // PUTBACK
        error word      // ERRORS
        endofs word     // ENDOFS
        stateofs word   // STATEOFS
        lmarg word      // * left margin
        rmarg word      // * right margin
        options word    // * option flags
        nwrds word      // * words per full scan line
        blank word
        ssa word        // 15, start of storage area
        esa word        // 16, end of storage area
        fdcb word       // 17, first DCB
        ldcb word       // 18, last DCB
        blksz word      // 19, block size for text line
        bwrds word      // *20, words per scan line
        dba word        // 21, destination bit address
        bstop word      // *22, bit where to stop writing
        bsofar word     // 23, bits so far in this line
        wad word        // 24, dest. word address
        scroll word     // *25, scrolling routine
        savac2 word     // *26, temp for AC2
        cdcb word       // * current DCB
        fmp word        // * pointer to full text line of bitmap
        bda word        // * beginning of bitmap data area
        tdcb word       // * top DCB with data
        mwp word        // * bitmap writer pointer
]
manifest lDS = (size DS)/16

manifest        // DS options
[       DScompactleft = 1       // eliminate leading blank words
        DScompactright = 2      // eliminate trailing blank words
        DSstopright = 4 // discard rather than scroll on line overflow
        DSstopbottom = 8        // discard rather than lose screen data
]
```