VISION BASED SYSTEMS FOR HARDNESS TESTING AND NDT

by

Ian Colin Smith, B.Eng., M.Sc.(Eng.).

Department of Electrical Engineering and Electronics.

October 1990.

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor of Philosophy.

APPENDICES

APPENDIX IV

FRAME STORE CONTROL REGISTERS

Video Input Control/Status Register 1 (INCSR1)

The INCSR1 register is a read/write register that interfaces with and controls the DT2853 frame store. This register can be read and written at any time.

Bits 15-8 - RESERVED. Read as 1/Write ignored.

These bits are unassigned and read back as ones. All writes to these bits are ignored.

Offset from base address (hex)	Register Name	Register Function
0	Video Input Control/Status Register 1 (INCSR1)	Controls the video input
2	Video Input Control/ Status Register 2 (INCSR2)	Controls the video input
4	Video Output Control/ Status Register 2 (OUTCSR)	Controls the video output
6	Cursor (CURSOR)	Contains the cursor position
8	Index (INDEX)	Contains the LUT index
A	Input Look-Up Table Entry (INLUT)	Contains the LUT entry
C	Red-Green Output Look-Up Table Register (REDGRN)	Contains the red and green LUT entry
E	Blue Output Look-Up (BLUE)	Contains the blue LUT entry

Table A4.1: Frame Store Control Register Summary

Bit 7 - BUSY, Read/Write.

Setting BUSY starts the operation selected by MODE (bits 4 to 6 of INCSR2). While BUSY remains set, the selected operation is in progress. If the ENSTOP (Enable Stop) bit is set, BUSY clears automatically at the completion of an operation. Otherwise BUSY remains set and multiple operations occur. The board may be brought to an immediate stop by writing a '1' to the ENSTOP bit, then writing a '0' to the BUSY bit. This is not recommended by the manufacturers except when initialising the board. BUSY is clear on power-up and cannot be set when in "reserved" operating mode (MODE equals 010 or 011 binary).

The "video in" and "feedback" operations are synchronised with the board's video timing. When BUSY is set, "video in" and "feedback" modes do not actually begin until the end of an even field vertical sync. BUSY is automatically cleared at the beginning of the next even field vertical sync. If ENSTOP is clear, BUSY sets again to perform another operation. ENSTOP must be set for BUSY to remain clear and stop operations. When ENSTOP is clear, a done interrupt is generated each time BUSY temporarily clears (if interrupts are enabled).

If ENSTOP is clear, BUSY remains set. BUSY cannot be cleared unless ENSTOP is set first. To achieve this, ENSTOP must be set by writing to INCSR1 (BUSY then clears and stays clear after the operation is complete). If no operation is pending, another write to INCSR1 is necessary to clear BUSY with ENSTOP kept set.

Bit 6 - INTERRUPT ON DONE (DONEINT), Read/Write

When set, DONEINT enables an interrupt to occur at the end of any operation.

Bit 5 - EXTTRG - read only.

Anything written to this bit will be ignored. The bit is the logical OR of two external trigger inputs (not used here).

Bit 4 - RESERVED, Read as 1/Write ignored.

These bits are unassigned and read back as ones. All writes to these bits are ignored.

Bit 3 - ENABLE STOP (ENSTOP), Read/Write.

When set, writing a 1 to BUSY (bit 7) enables a single operation. The operation starts as specified in the description of BUSY. When the operation is completed, BUSY automatically clears. When ENSTOP is clear, BUSY automatically sets, regardless of the status of board operations. BUSY remains set despite any attempts to clear BUSY by writing to the INCSR1 register. This allows multiple operations without having to set busy each time.

Bits 2-0 - INPUT LUT SELECT (ISEL2,ISEL1,ISEL0), Read/Write.

These bits select one of eight input look-up tables to be used during input operations (see table A4.2). All input data is transformed through these tables before being written to the frame store image buffer.

	Input LUT Select Bits		
Input LUT	**ISEL2**	**ISEL1**	**ISEL0**
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Table A4.2 Input LUT Select

Video Input/Control Status Register 2 (INCSR2)

The INCSR2 register is a read/write register which interfaces with and controls the DT2853. The register can only be written to if BUSY (bit 7 of INCSR1) is clear; otherwise writes are ignored. INCSR2 can be read at any time.

Bits 15-8 - RESERVED, Read as 1/Write ignored.

These bits are unassigned and are read back as ones. All writes to these bits are ignored.

Bit 7 - BUFFER SELECT (BUFSEL), Read/ Write.

This bit determines which frame store image buffer is used during the execution of any of the operating modes. When set, this bit specifies frame store image buffer 1. When clear, this bit specifies frame store image buffer 0. (Note: a bit in the OUTCSR, DISBUF selects the buffer used for display and feedback data).

Bits 6-4 - MODE (MODE2,MODE1,MODE0), Read/Write.

These bits control which operation occurs when BUSY (bit 7 of INCSR1) is set (see table A4.3).

Bit 3 - WRITE PROTECT 3 (WP3), Read/ Write.

When set, this bit disables all write operations to bit planes 4, 5, 6 and 7 of the frame store image buffers. Bus reads and writes are affected as well as video acquisition.

Modes	MODE2 Bit 6	MODE1 Bit 5	MODE0 Bit 4
Feedback	0	0	0
Video In	0	0	1
Reserved	0	1	0
Reserved	0	1	1
Load LUT	1	0	0
Trigger In	1	0	1
Reserved	1	1	0
Reserved	1	1	1

Table A4.3 Mode Select

Bit 2 - WRITE PROTECT 2 (WP2), Read/ Write.

When set, this bit disables all write operations to bit planes 2 and 3 of the frame store image buffers. Bus reads and writes are affected as well as video acquisition.

Bit 1 - WRITE PROTECT 1 (WP1), Read/ Write.

When set, this bit disables all write operations to bit plane 1 of the frame store image buffers. Bus reads and writes are affected as well as video acquisition.

Bit 0 - WRITE PROTECT 0 (WP0), Read/ Write.

When set, this bit disables all write operations to bit plane 0 of the frame store image buffers. Bus reads and writes are affected as well as video acquisition.

Video Output Control/ Status Register (OUTCSR)

The OUTCSR register is a read/write register which interfaces and controls the video output portion of the DT2853. This register can be read or written at any time.

Bit 15 - VERTICAL SYNC (VSYNC), Read/Write ignored.

When set, this bit indicates that a vertical sync is occurring in the video timing. Changes to the input and output look-up tables, cursor and BUSY (bit 7 of INCSR1) may be made without showing up on the output display, if done during the vertical sync. VSYNC is set for 10 lines at the beginning of each field. After the bit changes from 1 to 0, there is an additional 10 lines before active video display begins. This guarantees at least 630 μs of blanking after the bit is checked and found to be set.

Bit 14 - INTERRUPT ON SYNC (SYNCINT), Read/Write.

When set, this bit allows an interrupt to occur on the next clear to set transition of VSYNC.

Bit 13 - FIELD, Read/Write ignored.

When this bit is set, it indicates that the odd field is being digitised. When clear, this bit indicates that the even field is being digitised.

Bit 12 - EXTTRG, Read/Write ignored.

This is a duplicate of the EXTTRG bit in INCSR1.

Bit 11-8 - RESERVED, Read as 1/Write ignored.

These bits are unassigned and read back as ones. All writes to these bits are ignored.

Bit 7 - DISPLAY (DISP), Read/Write.

When set, data is taken from the selected frame store image buffer (DISBUF of OUTCSR), transformed by the selected output look-up table (OSEL0, OSEL1, OSEL2 and OSEL3 of OUTCSR), and presented to the inputs of the red, green and blue digital-to-analogue converters. When clear, a black image is displayed on the monitor.

Bit 6 - CURSOR (CURS), Read/Write.

When set, a full-screen cross-hair cursor is presented to the video outputs to be displayed.

Bit 5 - EXTERNAL TIMING (EXTTIM), Read/Write.

When set, video timing is synchronised to the input signal through a phase locked loop. When clear, video timing is generated by an internal crystal-controlled clock.

Bit 4 - DISPLAY BUFFER SELECT (DISBUF), Read/Write.

This bit determines which of the two frame store image buffers is selected for display and for the feedback data source. When set, this bit specifies frame store image buffer 1. When clear, this bit specifies frame store memory 0.

Bits 3-0 OUTPUT LUT SELECT (OSEL3, OSEL2, OSEL1, OSEL0), Read/Write.

These bits select one of eight output look-up tables to be used for the output display (see table A4.4). OSEL3 is unassigned and reads back as zero. All writes to this bit are ignored.

Cursor

The cursor register contains the pixel and line position of the cursor divided by two. This allows 8 bits for the pixel address (bits 0 to 7) and eight bits for the line address (bits 8 to 15). The cursor can only be set to even pixels and even lines. The resulting full-screen, cross-hair cursor is two pixels wide and two pixels high. This cursor covers not only the selected line and pixel, but the odd line and pixel as well.

Output LUT	OSEL3 Bit 3	OSEL2 Bit 2	OSEL1 Bit 1	OSEL0 Bit 0
Feedback	0	0	0	0
Video In	0	0	0	1
Reserved	0	0	1	0
Reserved	0	0	1	1
Load LUT	0	1	0	0
Trigger In	0	1	0	1
Reserved	0	1	1	0
Reserved	0	1	1	1

Table A4.4 Output LUT Select

Index

The INDEX register is a read/write register which determines the LUT index to be used when reading or writing look-up table entries of the INLUT, REDGRN or BLUE registers. This register can be read and written only when the board is in "load LUT" mode (MODE = 100).

Input Look-Up Table Entry (INLUT)

The INLUT register is a read/write register which contains the data to be written to or the data that was read from the selected LUT (specified by ISEL2, ISEL1 and ISEL0 of INCSR1) entry (specified by INDEX). This register can only be accessed when the board is in "load LUT" mode (mode = 100).

Redgreen Output Look-Up Table Register (REDGRN)

The REDGRN register is a read/write register which contains the data relating to the red and green outputs of the selected look-up table entry.

Bits 0-7 contain the data of the RED output look-up table selected by the values of OSEL2, OSEL1, and OSEL0 (specified by INDEX). If the bits are all zeros, the lowest intensity (black) is produced; if they are all ones, full intensity is produced. This register can be accessed only when the board is in the Load LUT mode (mode = 100).

Blue Output Look-Up Table Register (BLUE)

The BLUE register is a read/write register which contains the data relating to the blue output of the selected LUT (specified by INDEX). If the bits are all zeros, the lower intensity (black) is produced; if they are all ones, full intensity is produced. This register can be accessed only when the board is in the Load LUT mode (MODE = 100).

APPENDIX V

8086 ASSEMBLER CODE INTERFACE ROUTINES

function: camera (input_buffer, input_lut, output_buffer, output_lut)

arguments:

	int input_buffer	image buffer into which images are captured
	int input_lut	input LUT used to modify input image data
	int output_buffer	image buffer in which the displayed image is stored
	int output_lut	output LUT used to modify output image data
returns:	none	

description:

This function causes the frame store to repeatedly capture frames from the video source. The buffer into which the video data is written and the input LUT by which the data is modified can be selected using the input_buffer and input_lut arguments. The function also allows the output buffer from which the output video data is read and output LUT through which the data is modified to be selected using the output_buffer and output_lut arguments. If the buffers used for reading and writing are the same then a real-time image is produced on the output monitor.

8086 assembler code:

The caller's stack frame pointer is first saved and the function's stack pointer established. ENSTOP (bit 3 of INCSR1) is then set allowing BUSY (bit 7 of INCSR1) to clear at the end of the present operation. This allows the operating mode of the frame store to be changed.

Once the BUSY bit has cleared, indicating that the previous operation has been completed, then the first argument (input_buffer) is read from the stack and it's least significant bit written to BUFFER SELECT (INCSR2 bit 7). The MODE bits (INCSR2 bits 4 to 6) are then set to 001 giving "video in" operation. The write protection bits (least significant nibble of INCSR2) are cleared to give full access to the selected image buffer.

The second argument (input_lut) is then read from the stack and it's least significant nibble written to INPUT LUT SELECT (bits 0 to 3 of INCSR1), so that the desired input LUT is placed in the input data path. By clearing bit 6 of INCSR1, the DONE INTERRUPT is then disabled so that the host computer processor cannot be interrupted by the frame store.

The output buffer and LUT are set up by writing to the OUTCSR register, the third argument (output_buffer) is read from the stack and it's least significant bit written to OUTPUT BUFFER SELECT (OUTCSR bit 4) so that the desired output LUT is placed in the output data path. The final argument (output_lut) is then read from the stack and it's least significant nibble written to output LUT select (bits 0 to 3 of OUTCSR). In the present application the cursor and external timing functions of the frame store are not required and are disabled by clearing bit 5 (ENABLE EXTERNAL TIMING) and bit 6 (CURSOR) of OUTCSR. Finally ENSTOP is cleared and BUSY set so that repeated frame capture occurs.

function: capture (input_buffer, input_lut, output_buffer, output_lut)

arguments:

	int input_buffer	image buffer into which an image is captured
	int input_lut	input LUT used to modify input image data
	int output_buffer	image buffer in which the displayed image is stored
	int output_lut	output LUT used to modify output image data
returns:	none	

description:

This function causes a single frame to be captured, into the image buffer specified by input_buffer, on each invocation while displaying the contents of the image buffer specified by output_buffer.

8086 assembler code:

The assembler code for this routine is identical to that of the camera() function with the exception that ENSTOP is set, rather than cleared, at the end of the routine. This allows BUSY to clear at the end of the operation so that only a single frame is captured.

function: date()

arguments: none

returns:

long int date present date

description:

This function returns the present data as a long integer in the form:

CC | YY | MM | DD

where:

CC = century

YY = year

MM = month

DD = date

(all the values are in BCD)

8086 assembler code:

Interrupt 1AH is used to trap to the system BIOS with the AH contents set to 04H on entry to indicate that the date is to be read from the real-time clock. On returning from the BIOS routine CH holds the century reading, CL holds the year reading, DH holds the months reading and DL holds the date reading. These values are copied to the BH, BL, AH and AL registers, respectively, so that the result is returned in the form given in section 3.4.1.

function: ilut (table, index, value)

arguments:

	int table	input LUT who's contents are to be modified
	int index	table index number (0 to 255)
	int value	value to be written at LUT location given by index
returns:	none	

description:

This function allows the contents of the input LUTs to be modified so that the input grey scale mapping can be altered.

8086 assembler code:

The caller's stack frame pointer is first saved and the function's stack pointer established. The BUSY bit is then polled to check if there is an on-going operation. When this bit clears the contents of the INCSR1 and INCSR2 registers are saved. The INCSR2 contents are then modified by writing 0040H to the register; this sets the MODE bits to 100 and selects "load LUT" operation.

The LUT to be written to is selected by adding the first argument (table) to the contents of INCSR1 and writing the results to the register. This modifies the LUT SELECT bits (bits 0 to 2) according to the LUT specified in the function argument. The table element is selected by writing the second argument (index) to the INDEX register and the LUT contents then modified by writing the third argument (value) to the INLUT register.

Finally the INCSR1 and INCSR2 register contents are restored in order to return the frame store to it's previous operating mode.

function: ltime (centre, ncolumns)

arguments:

centre	column number of first column to be transferred
int ncolumns	number of groups of four columns to be transferred

returns: none

description:

This function reads the data contained within groups of four columns from frame store buffer 0 and writes them to consecutive locations in the host computer memory. The locations used, in base memory, are the same as those used for down-loaded image frames (see mov_buf()).

8086 assembler code:

The assembler code used for this function is similar to that of readpixl(). The physical address of the source is found in the same manner as in readpixl() with the first argument (centre) used as the y value and the x value set to zero. The physical address of the target is calculated as in readpixl().

The frame store is then configured for continuous frame capture operation as in the camera() function. Here image buffer 0 is configured for read/write access and the data routed via input LUT 2 and output LUT 0. A variable containing the number of captured frames is then initialised and the OUTCSR register polled for the start of an odd field. This is signalled by VSYNC and FIELD (bits 13 and 15 respectively) both being set. Once this condition has been detected the BIOS "move block" function is called with the block size set to 400H (i.e. 2kbyte = 4 rows of pixels). This transfers the grey levels of four columns of pixels to base memory within a single video blanking period. A check is then made to see if sufficient frames have so far been captured. If this is not the case control is transferred back to the polling loop, otherwise continuous frame capture operation is disabled by setting ENSTOP and the routine is exited.

function: mov_buf (buffer)

arguments:

	int buffer	number of buffer who's contents are to be transferred
returns:	none	

description:

This function down-loads the contents of an entire image buffer (specified by the buffer argument) to the host computer memory. The data is stored in the host computer memory at locations 5C000H - 9BFFFH.

8086 assembler code:

The routine moves data in 64 kbyte blocks from the frame store image buffer store to the host. BIOS calls are required to achieve this since the image buffers form part of the host's extended memory. Interrupt 15H is used trap to the BIOS and the required "move block" BIOS function is selected by setting the accumulator to 87H on entry (i.e. before the interrupt occurs). The number of words in the block to be transferred is stored in the CX register. The BIOS function also requires that a descriptor table (GDT) be set up; this includes, amongst other things, the physical addresses of both the source and target locations. The GDT is described fully in the IBM Technical Reference Manual [47]. The 24 bit physical address of the source, (A00000H for buffer 0, A40000H for buffer 1) is calculated as follows. The least significant word of the address is first set to 0000H and the first function argument (the buffer number) read from the stack. This, two byte, value is then multiplied by 4 and A0H added. The most significant byte of the (three byte) result is used as the most significant byte of the physical address. The source address, pointed to by read_add, is then complete.

The next step is to calculate the 24 bit physical address of the target location. Of this address, pointed to by point_tgt, only the least significant 20 bits of the address are in fact significant since the target location is in base memory (i.e. the first Megabyte). A value of 5C000H was chosen for the target location and this value is written to the GDT in the same manner as the source address. Before the BIOS function is called the location of top of the GDT needs to be passed to the function in the form of segment-plus-offset. These values are calculated and placed in the ES and SI registers respectively. The size of the block to be moved is then specified as 32k words (64k pixels) and the BIOS function invoked.

The above process is repeated until the entire contents of the image buffer (256 kbyte) have been transferred. This requires a total of eight block transfers so the above

operation is repeated a further seven times with the source and target addresses incremented by 1000H (64k) after each BIOS call.

function: olut (table, index, value)

arguments:

	int table	output LUT who's contents are to be modified
	int index	table index number (0 to 255)
	int value	value to be written at LUT location given by index
returns:	none	

description:

This function allows the contents of the output LUTs to be modified so that the output grey scale mapping can be altered.

8086 assembler code:

The assembler code for this function is almost identical to that of the ilut() function. The major differences are that the output LUT is selected by writing to the OSEL bits (bits 0 to 3 of OUTCSR), rather than ISEL, and that the specified table value is written to both the blue and red/green LUTs using the BLUE and REDGRN registers. This gives a full intensity image when the output video is fed to a monochrome monitor.

function: read_pix (x, y, buffer)

arguments:

	int x	x co-ordinate of pixel to be read
	int y	y co-ordinate of pixel to be read
	int buffer	buffer containing the specified pixel

returns:

	int pixel	grey scale of pixel at location (x,y)

description:

This function reads the grey scale value of a single pixel, located at (x,y), in a specified image buffer.

8086 assembler code:

The code for this function is similar to that used in the mov_buf() routine and in fact this function uses the same GDT. The offset of the pixel address from the image buffer base address is calculated and the result added to the image buffer base address to give the required source address. Since the video signal is digitised row by row the offset of the pixel location from the base can be calculated by forming the product of the row length (here 200H = 512) and the y co-ordinate then adding the result to the image buffer base address. The assembler code implements this equation and adds the result to either A00000H (for buffer 0) or A40000H (for buffer 1). In both cases the physical address of the source is pointed to by read_add.

The next stage is to calculate the physical address of the target. This occupies three bytes of storage and is pointed to by point_tgt. The (data) segment value of the target and it's offset are first determined, the segment value is then multiplied by 16 and added to the offset in order to obtain the 24 bit physical address. The most significant 4 bits of this address are always zero since the target is located in base memory (i.e. the first Megabyte).

The location of the GDT is then passed to the BIOS "move block" function as in mov_buf() and the function called. The BIOS function actually transfers the grey scale values of two pixels on each invocation, since only word transfers are permitted. The most significant byte of the target data is set to zero before being being passed to the calling program.

function: readpixl (x, y)

arguments:

	int x	x co-ordinate of pixel to be read
	int y	y co-ordinate of pixel to be read

returns:

	int pixel	grey level of the pixel located at (x,y)

description:

This routine is similar in effect to read_pix(), however here pixel data is read from a previously down-loaded image frame, held in base memory, rather than the frame store image buffer.

8086 assembler code:

The stack pointer is first set up in the same manner as in the other routines. The offset of the start of the row, specified in the second function argument, from the base address of the data block is calculated by multiplying the length of a row (200H = 512 pixels) by the y co-ordinate. Of the 32 bit result, stored in the (DX,AX) register combination, only 12 bits of the value are in fact significant; the least significant 8 bits are always zero (since the lower 8 bits of the multiplier are zero) and the most significant 12 bits always zero (since the result is always less than 00040000H). The 12 significant bits are shifted to the most significant bit locations of DX and the base address divided by 16 (i.e. 5C00H) is added. The result gives the physical address of start the specified row divided by 16. Clearly, by specifying this result as a segment value and the x co-ordinate (first function argument) as an offset the memory location holding the pixel value is addressable. In this routine the current data segment value is saved onto the stack, the data segment register contents are then overwritten by the required segment value and the offset (x co-ordinate) written to the base index (BX) register. An indirect move is then used to fetch the pixel value before the previous data segment value is restored.

function: res_buf (buffer)

arguments:

int buffer	number of buffer who's contents are to be over written

description:

The function restores the contents of a previously down-loaded image buffer, held in base memory, to the frame store image buffer.

8086 assembler code

The assembler code and GDT used for this routine are virtually identical to that used for the mov_buf() function, the only difference being that the pointers to the source and target locations are interchanged.

function: time()

arguments: none

returns:

long int time	present time

description:

This function returns the present time as a long integer in the form:

time: 00 | HH | MM | SS

where: HH = hours

MM = minutes

SS = seconds

(all the values are in BCD)

8086 assembler code:

Interrupt 1AH is used to trap to the system BIOS with the AH register contents set to 02H on entry to indicate that the time is to be read from the real-clock clock. On returning from the BIOS routine, CH holds the hours reading, CL the minutes reading and DH the seconds reading. The values are copied to the AL, BH, and BL registers respectively so that the result is returned in the same form as in section 3.4.1.

function: wrt_pix (x, y, value, buffer)

arguments

int x	x co-ordinate of pixel location
int y	y co-ordinate of pixel location
int value	byte value to be written to image buffer

description:

This function changes the grey level of a pixel located at (x,y) by writing a single byte to the specified image buffer.

8086 assembler code:

The assembler code for this function is similar to that used in readpixl() and the GDT is the same as that used by res_buf(). The routine is complicated by the fact that only word transfers of data are possible using the BIOS "move block" function. Consequently two bytes need to be read from the frame store image buffer, the byte corresponding to the specified pixel over written and the resulting word then returned to the buffer.

function: wrtpixl (x, y, value)

arguments:

int x	x co-ordinate of pixel location
int y	y co-ordinate of pixel location
int value	byte value to be written to base memory

description:

This routine is similar in effect to read_pix(), however here the pixel data is written to a previously down-loaded image frame, held in base memory, rather than the frame store image buffer.

8086 assembler code:

The code is almost identical to readpixl(), the only difference being that a byte value is written to the base memory rather than read from it.

```
;------------------------------------------------------------------------
; This program contains functions which initialise the control registers
; and Look Up Tables of the DT2853 frame store.It also controls the
;  operation of DT2853 via functions CAMERA, CAPTURE, DISPLAY.
; DATA transfer is also facilitated between the frame store and programs.
; It is to be called from a C program which supplies the values for
; initialisation and control.
;------------------------------------------------------------------------
;
 TITLE Board initialisation and data access
 SUBTTL 29/1/88
 NAME dt2853
;
 INCLUDE DOS.MAC
   PUBLIC read_pix,wrt_pix,camera,capture,show_buf,ilut,olut,pcapture,fback,sync
   PUBLIC turn_on,turn2,turn3,cam2
;
;NOTE: The base address for control registers has been CHANGED to 250h
; For details of registers see DATA TRANSLATION manual pp 5-5 to 5-30
;
; Define register addresses:
;
 base_add equ 250h
 incsr1  equ base_add ; Video Input Control/Status 1
 incsr2  equ base_add  +  2 ; Video Input Control/Status 2
 outcsr  equ base_add  +  4 ; Video Output Control/Status
 cursor  equ base_add  +  6 ; Cursor control
 index  equ base_add  +  8 ; Contains the LUT index
 inlut  equ base_add  +  0aH ; Contains LUT entry
 redgrn  equ base_add  +  0cH ; Contains the red & green LUT
 blue  equ base_add  +  0eH ; Contains the blue LUT entry
;
;------------------------------------------------------------------------
;
;      DATA items now declared and descriptor tables set up.
; For details of descriptor table see IBM tech. ref. pp 5-149,5-150.
;
 DSEG
PUBLIC  tgt_data
;
; The following is the descriptor table for reading from the frame store
;
  dummy  DQ 0

  gdt_loc  DQ 0

  src_seg_limit DW 0ffffH
  src_base_add DB ?,?,?; DT base address  =  A00000H
;      To be filled by pointer
  src_dat_ri DB 93H
  src_dat_res DW 00H

  tgt_seg_limit DW 0ffffh
  tgt_base_add DB ?,?,?; To be filled by pointer
  tgt_dat_ri DB 93H
  tgt_dat_res DW 00H
```

```
bios_cs  DQ 0

temp_ss  DQ 0
;
;
;                              next location holds 100 rows of pixels
          tgt_data DW 100 DUP (256 DUP (?))
;
gdt     DD dummy ; gdt points (seg,offset) to top
;      of table
read_add  DW src_base_add ; points(offset) to source
;      base address
 point_tgt       DW      tgt_base_add ; pointer defined for
;      filling target address
;
          xtemp          DW      ?  ; store for x window co ord
;
          ytemp          DW      ?  ; store for y window co ord
;
          pixcnt         DW      ?  ; number of white pixels counted
;
          rows           DB      ?  ; number of rows to be read
;
          columns        DB      ?  ; number of colums be read
;
          nwords         DW      ?  ; total number of words to be moved
;
;---------------------------------------------------------------------------
; Write descriptor table now setup
;
wdummy  DQ 0
;
wgdt_loc DQ 0
;
wsrc_seg_limit DW 0ffffH
wsrc_base_add DB ?,?,?; Equivalent to tgt_base_add
wsrc_dat_ri DB 93H
wsrc_dat_res DW 00H

wtgt_seg_limit DW 0ffffh
wtgt_base_add DB ?,?,?; Dt board;To be filled by pointer
wtgt_dat_ri DB 93H
wtgt_dat_res DW 00H

wbios_cs DQ 0

wtemp_ss DQ 0
;
;
write_add DW wtgt_base_add ;pointer to DT board
;      address location
wgdt     DD wdummy ; wgdt points (seg,offset) to top
;      of table
wrt_scr_add  DW wsrc_base_add ; points(offset) to source
;      base address
tgt_dat_add DW ?      ;variable to hold pixel value
;      address
```

```
            xtempx          DB      ?
            ytempx          DB      ?
;--------------------------------------------------------------------------

  ; define variables for control purposes
;
inc1_val DW ? ; value in INCSR1
inc2_val DW ? ; value in INCSR2
out_val  DW ? ; value in OUTCSR
;
ENDDS
;
;--------------------------------------------------------------------------
 PSEG  ; program segment starts

;--------------------------------------------------------------------------
;
; This procedure continually captures frames
;
BEGIN turn3
     int     3
push bp
 mov bp,sp



;
;
;   Busy must be clear to proceed
mov dx,incsr1
mov ax,08H
out dx,ax  ; Set ENSTOP
out dx,ax  ; Clear BUSY
zcheck_b:
in  ax,dx
and ax,080H  ; Check BUSY is clear
jnz zcheck_b
;
;   value for inscr2 now calculated
 mov dx,incsr2
mov ax,0ff10H ; MODE  =  001, no write protect
      out dx,ax
mov dx,outcsr
mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
out dx,ax      ;
;
mov dx,incsr1
mov ax,0ff82H ; Set BUSY, Clear  ENSTOP; Input LUT  =  002
out dx,ax           ; board set to capture continuously
;
                             ;
                             ; calculate number of pixels to be checked
;       mov     ax,[bp+ 10]      ; xmax into ax
;      clc
;       sbb    ax,[bp + 8]        ; subtract xmin to give number of pixels
;       mov     pixcnt,ax        ; number of pixels to pixcnt
;       mov     cl,1            ; divide by two to give number of words
;       shl    ax,cl
```

```
;       mov     nwords,ax       ; number of words to nwords
 mov ax,200H  ; length of one line i.e 512 pixels
 mov  bx,[bp+6]       ; L memory model used; bx = y position
 mul bx  ; calculate y position in memory,result in DX:AX
;       add     ax,[bp+8]       ; include x offset - xmin
        add     dx,0a0H
 mov bx,read_add ; bx points to base address in GDT
 mov [bx+2],dl ; put hi byte pixel address calculated into GDT
 mov [bx+0],ax ; put low word of pixel address into GDT
;    ; 24 bit address of target now calculated
 mov ax,ds  ; ax
.SI
 loaded with (segment:offset) of
 lea si,tgt_data ; intended location of pixel value
;
 mov bx,10H  ; ax now shifted 4 places left and result
 mul bx  ; placed in DX:AX
;
 add ax,si  ; offset address now added to lower word
 adc dx,0H  ; and any carry resulting added to DX
;
 mov     bx,point_tgt ; bx points to target base address
 mov [bx],ax  ; tgt_base_add now filled with
 mov [bx+2],dl ; low word, high byte
                                ; wait for video blanking period
zstart:
        mov     dx,outcsr
zwait_sync:
        in      ax,dx
        and     ax,08000H       ; mask off vsync -bit 15
        cmp     ax,08000H       ; vsync set ?
        jne     zwait_sync      ; no, hang on
                                ; yes, start processing

 les si,gdt  ; es
.SI
 now points to top of GDT
;                               BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,256    ; number of  double pixels to be moved
 int 15H  ; BIOS interrupt


;
;                               got the data now sort it out
;
                                ; extablish start of data
      lea       bx,tgt_data   ; bx holds start address of data block
      mov       dx,0            ; dx holds number of white pixels found
                                ; count reset here
      mov       cx,512          ; cx holds total number of pixels to be read
next_pix:
      mov       al,[bx]         ; get the data
      inc       bx              ; update data address
      cmp       al,0            ; pixel white ?
      je        zblack          ; no
      inc       dx              ; yes, increment pixel count
zblack:
```

```
     loop      next_pix       ; get next pixel value
;                        check threshold
      cmp     dx,[bp+8]       ; no of white pixels > threshold
      jge    zfinish         ; yes, finished
      jmp     zstart          ; no - try  again
zfinish:
 pop bp              ;  return to calling prog
 ret
turn3 ENDP

;-----------------------------------------------------------------------
;
; This procedure continually captures frames
;
 BEGIN turn2
 push bp
  mov bp,sp



;
;   calculate number of rows to be moved
      mov     bx,[bp + 12]  ; ymax into bx
      clc
      sbb     bx,[bp + 8]       ; number of rows into bx  =  ymax - ymin
      mov      rows,bl          ; number of rows into rows - one byte only
;
;                         calculate number of columns to be moved
      mov     bx,[bp + 10]  ; xmax into bx
      clc
      sbb     bx,[bp + 6]    ; number of columns into bx  =  xmax - xmin
      mov      columns,bl      ; number of columns into columns
                          ; - one byte only
;
;                          calculate total number of words to be moved
   mov     al,rows         ; total number of pixels  =  rows * 512
      mov      ah,0
      mov      bx,100H
      mul     bx             ; total number of words   =  rows * 256
      mov      nwords,ax
;
;   Busy must be clear to proceed
 mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
xcheck_b:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz xcheck_b
;
;   value for inscr2 now calculated
  mov dx,incsr2
 mov ax,0ff10H ; MODE  =  001, no write protect
      out dx,ax
 mov dx,outcsr
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 out dx,ax      ;
```

```
;
 mov dx,incsr1
 mov ax,0ff82H ; Set BUSY, Clear  ENSTOP; Input LUT = 002
 out dx,ax          ; board set to capture continuously
;
                                ; wait for video blanking period
xstart:
      mov    bx,0          ; reset pixel count
      mov    pixcnt,bx
      mov    dx,outcsr
xwait_sync:
      in     ax,dx
      and    ax,08000H       ; mask off vsync -bit 15
      cmp    ax,08000H       ; vsync set ?
      jne    xwait_sync      ; no, hang on
                                ; yes, start processing
                                ;
                                ;
                                ; calculate start address of block to be moved
 mov ax,200H  ; length of one line i.e 512 pixels
 mov  bx,[bp+8]       ; L memory model used; bx = ymin position
 mul bx  ; calculate y position in memory,result in DX:AX
      add     dx,0a0H
 mov bx,read_add ; bx points to base address in GDT
 mov [bx+2],dl ; put hi byte pixel address calculated into GDT
 mov [bx+0],ax ; put low word of pixel address into GDT
;   ; 24 bit address of target now calculated
 mov ax,ds  ; ax
.SI
 loaded with (segment:offset) of
 lea si,tgt_data ; intended location of pixel value
;
 mov bx,10H  ; ax now shifted 4 places left and result
 mul bx  ; placed in DX:AX
;
 add ax,si  ; offset address now added to lower word
 adc dx,0H  ; and any carry resulting added to DX
;
 mov     bx,point_tgt ; bx points to target base address
 mov [bx],ax  ; tgt_base_add now filled with
 mov [bx+2],dl ; low word, high byte

 les si,gdt  ; es
.SI
 now points to top of GDT
;                               BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,nwords    ; number of  double pixels to be moved
 int 15H  ; BIOS interrupt

;
;                               got the data now sort it out
;
;                               make temporary store for rows
     mov     bl,rows
     dec     bl             ; subtract 1 - make 0 the start
     mov     bh,0             ; set high byte zero
```

```
      mov     ytemp,bx
      mov     bl,columns     ; temp store for columns
      dec     bl             ; make 0 the start
      mov     bh,0           ; set high byte zero
      mov     xtemp,bx
                             ; extablish start of data
      lea     dx,tgt_data    ; dx holds start address of data block
      add     dx,[bp + 6]    ; add xmin to extablish first pixel in window
next_col:
      mov     bx,dx
      add     bx,xtemp       ; add the x offset
      mov     al,[bx]        ; get the data

      cmp     al,0           ; pixel white ?
      je      xblack         ; no
      inc     pixcnt         ; yes, increment pixel count
xblack:
      dec     xtemp          ; update x co-ord
      cmp     xtemp,0        ; start of row ( going backwards ) ?
      jge     next_col       ; no - carry on
                             ; yes - start the next row
      mov     bl,columns     ; reset xtemp
      dec     bl             ;
      mov     bh,0           ;
      mov     xtemp,bx
      add     dx,200H        ; add 512 pixels to start next row
      dec     ytemp          ; decrement row count
      cmp     ytemp,0        ; all rows done ?
      jge     next_col       ; no - do another
                             ; yes - finished
                             ;
;                            check threshold
      mov     bx,[bp + 14]
      cmp     bx,pixcnt      ; no of white pixels > threshold
      jle     xfinish        ; yes, finished
      jmp     xstart         ; no - try  again
xfinish:
 pop bp           ;  return to calling prog
 ret
turn2 ENDP

;-----------------------------------------------------------------------
;
; This procedure continually captures frames
;
 BEGIN turn_on
 push bp
 mov bp,sp
;    Busy must be clear to proceed
 mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
bcheck_b:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz bcheck_b
```

```
;
;   value for inscr2 now calculated
  mov dx,incsr2
 mov ax,0ff10H ; MODE = 001, no write protect
       out dx,ax
 mov dx,outcsr
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 out dx,ax      ;
;
 mov dx,incsr1
 mov ax,0ff82H ; Set BUSY, Clear  ENSTOP; Input LUT = 002
 out dx,ax          ; board set to capture continuously
;
                           ; wait for video blanking period
start:
       mov    bx,0          ; reset pixel count
       mov    pixcnt,bx
       mov    dx,outcsr
wait_sync:
       in     ax,dx
       and    ax,08000H      ; mask off vsync -bit 15
       cmp    ax,08000H      ; vsync set ?
       jne    wait_sync      ; no, hang on
                           ; yes, start processing
       mov    bx,[bp+6]      ; initialise co-ords
       mov    xtemp,bx
       mov    bx,[bp+8]
       mov    ytemp,bx
get_data:
                           ; get pixel value
 mov ax,200H  ; length of one line i.e 512 pixels
 mov  bx,ytemp  ; L memory model used; bx = y position
 mul bx  ; calculate y position in memory,result in DX:AX
 mov bx,xtemp ; L memory model used; bx = x position
 add ax,bx    ; calculate true location in memory:include x
       add    dx,0a0H  ; include offset of DT2853 base address
       mov bx,read_add ; bx points to base address in GDT
 mov [bx+2],dl ; put hi byte pixel address calculated into GDT
 mov [bx+0],ax ; put low word of pixel address into GDT
;    ; 24 bit address of target now calculated
 mov ax,ds  ; ax
.SI
 loaded with (segment:offset) of
 lea si,tgt_data ; intended location of pixel value
;
 mov bx,10H  ; ax now shifted 4 places left and result
 mul bx  ; placed in DX:AX
;
 add ax,si  ; offset address now added to lower word
 adc dx,0H  ; and any carry resulting added to DX
;
 mov     bx,point_tgt ; bx points to target base address
 mov [bx],ax  ; tgt_base_add now filled with
 mov [bx+2],dl ; low word, high byte

 les si,gdt  ; es
.SI
```

```
 now points to top of GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,1  ; number of pixels to be moved
 int 15H  ; BIOS interrupt
 mov ax,tgt_data   ; pixel value put into AL for return to C
 xor ah,ah  ; clear AH
;
       cmp    ax,0           ; pixel white ?
       je     black          ; no
       inc    pixcnt         ; yes, increment pixel count
black:
       inc    xtemp          ; update x co-ord
       mov    bx,xtemp
       cmp    [bp+10],bx     ; x = xmax ?
       jne    get_data       ; no, keep going
;
                             ; yes
       mov    bx,[bp+6]      ; reset x co-ord
       mov    xtemp,bx
       inc    ytemp          ; update y coord
       mov    bx,ytemp
       cmp    [bp+12],bx     ; y = ymax ?
       jne    get_data       ; no, keep going
                             ; yes, finished
       mov    bx,[bp+14]
       cmp    bx,pixcnt      ; no of white pixels > threshold
       jle    finish         ; yes, finished
       jmp    start          ; no - try  again
finish:
 pop bp              ;  return to calling prog
 ret
turn_on ENDP
;-------------------------------------------------------------------------
; This function initialises the input look up tables
; Form ilut(table_number,index,value)
;
 BEGIN ilut
 push bp
 mov bp,sp  ; establish this program's stack frame pointer
;
       mov    dx,outcsr      ; clear OUTCSR bit7 to turn off the display
       in     ax,dx
       and    ax,1111111101111111b
       out    dx,ax
;
;
;
;    To program input LUT, BUSY must first be clear
 mov dx,incsr1
 in ax,dx
 and ax,080H  ; check bit 7 (BUSY)
 jz continue        ; Busy clear,continue
 mov ax,08H  ; Set ENSTOP to allow
 out  dx,ax  ; busy to be cleared
 out dx,ax  ; Clear BUSY
continue:
```

```
;   Input LUT programming begins
;
 in ax,dx  ; read INCSR1
 mov  inc1_val,ax ; store INCSR1 value at inc1_val
 mov dx,incsr2
 in ax,dx  ; read INCSR2
 mov inc2_val,ax ; store INCSR2 value at inc2_val
;
 mov ax,040H  ;
 out dx,ax  ; set INCSR2 to load LUT : MODE 100b
 mov bx,[bp+6] ; fetch table number
 mov  dx,incsr1
 in ax,dx    ; read incrs1
      and    ax,1111111111111000b        ; set LUT bits to 000
 add ax,bx  ; include LUT number
 out dx,ax
;
 mov ax,[bp+8] ; fetch index
 mov  dx,index
 out dx,ax  ; output LUT index to INDEX register
;
 mov dx,inlut
 mov  ax,[bp+10] ; fetch lut entry
 out dx,ax  ;
;
      mov     dx,outcsr       ; set OUTCSR bit7 to enable the display
      in      ax,dx
      or      ax,0000000010000000b
      out     dx,ax
;
 pop bp
 ret
ilut ENDP
;------------------------------------------------------------------------
;
;  This function loads output LUT
; Form olut(table_number,index,value)
;
 BEGIN olut
 push bp
 mov bp,sp  ; establish this program's stack frame pointer
;
 mov dx,incsr1
 in ax,dx
 and ax,080H  ; check bit 7 (BUSY)
 jz cntnue          ; Busy clear,continue
 mov ax,08H  ; Set ENSTOP to allow
 out  dx,ax  ; busy to be cleared
 out dx,ax  ; Clear BUSY
cntnue:
;   output LUT programming begins
;
 mov dx,outcsr
 in ax,dx  ; read OUTCSR
 mov  out_val,ax ; store OUTCSR value at out_val
 mov dx,incsr2
 in ax,dx  ; read INCSR2
```

```
 mov inc2_val,ax ; store INCSR2 value at inc2_val
;
 mov ax,040H  ;
 out dx,ax  ; set INCSR2 to load LUT : MODE 100b
 mov bx,[bp+6] ; fetch table number
 mov  dx,outcsr
 in ax,dx    ; read outcsr
      and     ax,1111111111111000b        ; set LUT bits to 000;
 add ax,bx  ; include LUT number
 out dx,ax
;
 mov ax,[bp+8] ; fetch index
 mov  dx,index
 out dx,ax  ; output LUT index to INDEX register
;
 mov dx,redgrn
 mov  ax,[bp+10] ; fetch lut entry
 mov ah,al  ; GREEN LUT = RED LUT
 out dx,ax  ;
 pop bp
 ret
olut ENDP
;
;-----------------------------------------------------------------------
;
;-----------------------------------------------------------------------
; The following functions capture(ip_buff,ILUT,op_buff,OLUT)
; display(buffer,OLUT), camera(buffer,inlut,outlut) capture and display
; the specified buffer or camera through the specified LUTs
;-----------------------------------------------------------------------
;
; This procedure captures one frame
;
 BEGIN capture
 push bp
 mov bp,sp
;   Busy must be clear to proceed
 mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
check_b:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz check_b
;
;   value for inscr2 now calculated
 mov cl,7
 mov bx,[bp+6] ; fetch input buffer number
 shl bx,cl  ; shift bx 7 places left
  mov dx,incsr2
 mov ax,0ff10H ; MODE = 001, no write protect
 add ax,bx
      out dx,ax
 mov dx,outcsr
 mov bx,[bp+10] ; fetch display buffer
      mov     cl,4
```

```
        shl    bx,cl          ; shift buffer select to bit 4
        add    bx,[bp + 12] ; fetch output LUT
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 add ax,bx
 out dx,ax     ;
;
 mov dx,incsr1
 mov bx,[bp+8] ; fetch INPUT LUT
 mov ax,0ff88H ; Set BUSY,ENSTOP; Input LUT = 000
 add ax,bx
 out dx,ax
;
check_by:
 in  ax,dx
 and ax,080H
 jnz check_by
;       Frame captured, return to caller
 pop bp
 ret
capture ENDP
;
;------------------------------------------------------------------------
; The following functions capture(ip_buff,ILUT,op_buff,OLUT)
; display(buffer,OLUT), camera(buffer,inlut,outlut) capture and display
; the specified buffer or camera through the specified LUTs
; The most significant four bits are write protected.
;------------------------------------------------------------------------
;
; This procedure captures one frame
;
 BEGIN pcapture
 push bp
 mov bp,sp
;    Busy must be clear to proceed
 mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
check_b2:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz check_b2
;
;    value for inscr2 now calculated
 mov cl,7
 mov bx,[bp + 6] ; fetch input buffer number
 shl bx,cl  ; shift bx 7 places left
  mov dx,incsr2
 mov ax,0ff18H ; MODE = 001, ms nibble write protected
 add ax,bx
       out dx,ax
 mov dx,outcsr

 mov bx,[bp + 10] ; fetch display buffer
 add bx,[bp + 12] ; fetch output LUT
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 add ax,bx
```

```
 out dx,ax      ;
;
 mov dx,incsr1
 mov bx,[bp+8] ; fetch INPUT LUT
 mov ax,0ff88H ; Set BUSY,ENSTOP; Input LUT = 000
 add ax,bx
 out dx,ax
;
check_by2:
 in  ax,dx
 and ax,080H
 jnz check_by2
;       Frame captured, return to caller
 pop bp
 ret
pcapture ENDP

;--------------------------------------------------------------------------
;
; This procedure displays the captured frame
;
 BEGIN show_buf
 push bp
 mov bp,sp
 mov dx,outcsr
 mov bx,[bp+6]          ; fetch buffer number
 mov cl,4               ; shift to bit 4 - o/p buf select
 shl bx,cl
 add bx,[bp+8]          ; fetch output lut
 mov  ax,0000111110100000b
 add ax,bx
 out dx,ax   ; CURSOR OFF,BUF = 0,SYNC INT,O/P LUT=0
 pop bp
 ret
show_buf ENDP
;


;--------------------------------------------------------------------------
;
; This procedure feeds back a frame through the input lut
; usage :   camera (i/p buffer, i/p lut o/p buffer, o/p lut )
;
 BEGIN fback
 push bp
 mov bp,sp
;   Busy must be clear to proceed
 mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
chk_by3:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz chk_by3
;
;
```

```
;
 mov dx,outcsr
 mov bx,[bp+10] ; fetch display buffer
 add bx,[bp+12] ; fetch output LUT
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 add ax,bx
 out dx,ax      ;
;
;   value for inscr2 now calculated
 mov cl,7
 mov bx,[bp+6] ; fetch input buffer number
 shl bx,cl  ; shift bx 7 places left
  mov dx,incsr2
 mov ax,0ff00H ; MODE = 000, no write protect
 add ax,bx
       out dx,ax
;
;
   mov dx,incsr1
 mov bx,[bp+8] ; fetch INPUT LUT
 mov ax,0ff88H ; Set BUSY, set ENSTOP
 add ax,bx
 out dx,ax
;
       mov     dx,incsr1
chk_by4:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz chk_by4
 pop bp
 ret
fback ENDP

;
;--------------------------------------------------------------------------
;
; This procedure displays the camera output
;
 BEGIN cam2
 push bp
 mov bp,sp
;   Busy must be clear to proceed
  mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
achk_by:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz achk_by
;
;   value for inscr2 now calculated
 mov cl,7
 mov bx,[bp+6] ; fetch input buffer number
 shl bx,cl  ; shift bx 7 places left
  mov dx,incsr2
 mov ax,0ff18H ; MODE = 001, ms nibble write protected
```

```
 add ax,bx
       out dx,ax
 mov dx,outcsr
 mov bx,[bp+10] ; fetch display buffer
 add bx,[bp+12] ; fetch output LUT
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 add ax,bx
 out dx,ax      ;
;
 mov dx,incsr1
 mov bx,[bp+8] ; fetch INPUT LUT
 mov ax,0ff80H ; Set BUSY; Input LUT = 000
 add ax,bx
 out dx,ax
;
 pop bp
 ret
cam2 ENDP


;
;------------------------------------------------------------------------
;
; This procedure displays the camera output
;
 BEGIN camera
 push bp
 mov bp,sp
;   Busy must be clear to proceed
  mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
chk_by:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz chk_by
;
;   value for inscr2 now calculated
 mov cl,7
 mov bx,[bp+6] ; fetch input buffer number
 shl bx,cl  ; shift bx 7 places left
  mov dx,incsr2
 mov ax,0ff10H ; MODE = 001, no write protect
 add ax,bx
       out dx,ax
 mov dx,outcsr
 mov bx,[bp+10] ; fetch display buffer
 add bx,[bp+12] ; fetch output LUT
 mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
 add ax,bx
 out dx,ax      ;
;
 mov dx,incsr1
 mov bx,[bp+8] ; fetch INPUT LUT
 mov ax,0ff80H ; Set BUSY; Input LUT = 000
 add ax,bx
```

```
 out dx,ax
;
 pop bp
 ret
camera ENDP

;
;------------------------------------------------------------------------
;
; This procedure selects the internal sync source
;
 BEGIN sync
 push bp
 mov bp,sp
;   Busy must be clear to proceed
  mov dx,incsr1
 mov ax,08H
 out dx,ax  ; Set ENSTOP
 out dx,ax  ; Clear BUSY
chk2:
 in  ax,dx
 and ax,080H  ; Check BUSY is clear
 jnz chk2
;
 mov dx,outcsr
       in      ax,dx
       and     ax,1111111111011111b; set bit 5 - internal sync
 out dx,ax     ;
;
;
 pop bp
 ret
sync    ENDP
;
;------------------------------------------------------------------------
; This is function "read_pix(x,y,buffer)" which obtains the pixel value
; at position (x,y) in bufferand returns this value (in ax) to the caller.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.

;      This function returns the pixel value in AL; AH defines status
; of BIOS call:see IBM tech. ref.

 BEGIN read_pix
 push bp  ; save caller's stack frame pointer
 mov bp,sp  ; establish this function's stack
 mov ax,200H  ; length of one line i.e 512 pixels
 mov  bx,[bp+8] ; L memory model used; bx = y position
 mul bx  ; calculate y position in memory,result in DX:AX
 mov bx,[bp+6] ; L memory model used; bx = x position
 add ax,bx    ; calculate true location in memory:include x
 mov cx,[bp+10] ; fetch buffer no.
 jcxz  b0
 add   dx,0a4H  ; start of buffer 1
 jmp b1
b0: add dx,0a0H  ; include offset of DT2853 base address
b1: mov bx,read_add ; bx points to base address in GDT
 mov [bx+2],dl ; put hi byte pixel address calculated into GDT
```

```
 mov [bx+0],ax ; put low word of pixel address into GDT
;    ; 24 bit address of target now calculated
 mov ax,ds  ; ax
.SI
 loaded with (segment:offset) of
 lea si,tgt_data ; intended location of pixel value
;
 mov bx,10H  ; ax now shifted 4 places left and result
 mul bx  ; placed in DX:AX
;
 add ax,si  ; offset address now added to lower word
 adc dx,0H  ; and any carry resulting added to DX
;
 mov     bx,point_tgt ; bx points to target base address
 mov [bx],ax  ; tgt_base_add now filled with
 mov [bx+2],dl ; low word, high byte

 les si,gdt  ; es
.SI
 now points to top of GDT
;      BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,1  ; number of pixels to be moved
 int 15H  ; BIOS interrupt
 mov ax,tgt_data   ; pixel value put into AL for return to C
 xor ah,ah  ; clear AH
 pop bp
 ret
 read_pix ENDP
;
;-------------------------------------------------------------------------
; This is function "write_pix(x,y,z,buffer)" which writes the pixel value
; z to position (x,y) in buffer.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.
; Because the BIOS interrupt moves one word of memory,writing to one
; byte involves reading two bytes,at x and x+1,modifying byte at x
; and rewriting to x and x+1
;-------------------------------------------------------------------------
 BEGIN wrt_pix
 push bp  ; save caller's stack frame pointer
 mov bp,sp  ; establish this function's stack
;     First read the memory word starting
;    at the pixel location required
 mov ax,200H  ; length of one line i.e 512 pixels
 mov  bx,[bp+8] ; L memory model used; bx = y position
 mul bx  ; calculate y position in memory,result in DX:AX
 mov bx,[bp+6] ; L memory model used; bx = x position
 add ax,bx    ; calculate true location in memory:include x
 int 3
 mov cx,[bp+12]
 jcxz bu0
 add dx,0a4H
 jmp bu1
bu0: add dx,0a0H  ; include offset of DT2853 base address
bu1: mov bx,read_add ; bx points to SOURCE address in READ GDT
 int 3
 mov [bx+2],dl ; put hi byte pixel address calculated into GDT
```

```
 mov [bx + 0],ax ; put low word of pixel address into GDT
       mov bx,write_add ; bx points to TARGET address in WRITE GDT
 mov [bx + 2],dl ; put hi byte pixel address calculated into GDT
 mov [bx + 0],ax ; put low word of pixel address into GDT
;    ; 24 bit address of target now calculated
 mov ax,ds  ; ax
.SI
 loaded with (segment:offset) of
 lea si,tgt_data ; intended location of pixel value
 mov tgt_dat_add,si ; store location in tgt_dat_add
;
 mov bx,10H  ; ax now shifted 4 places left and result
 mul bx  ; placed in DX:AX
;
 add ax,si  ; offset address now added to lower word
 adc dx,0H  ; and any carry resulting added to DX
;
 mov     bx,point_tgt ; bx points to TARGET base address in READ GDT
 mov [bx],ax  ; tgt_base_add now filled with
 mov [bx + 2],dl ; low word, high byte
 lea bx,wsrc_base_add; bx points to SOURCE address in WRITE GDT
 mov [bx],ax  ; wsrc_base_add now filled with
 mov [bx + 2],dl ; low word, high byte

 les si,gdt  ; es
.SI
 now points to top of READ GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,1  ; number of pairs of pixels to be moved
 int 15H  ; BIOS interrupt
 mov bx,[bp + 10] ; value to be written to frame store in bx
 lea si,tgt_data ;
 mov [si],bl  ; byte overwritten in tgt_data
 les si,wgdt  ; es
.SI
 now points to top of WRITE GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,1  ; number of pixels to be moved
 int 15H  ; BIOS interrupt
 pop bp
 ret
wrt_pix ENDP
;
 ENDPS
 END
```

```
;------------------------------------------------------------------------
; This program moves DATA between the frame store and system memory.
;  If the whole buffer is to be moved,5C000 to 9BFFF is occupied.
; Otherwise 64K blocks are sequentially copied to block starting
; at 8C000.
; It is to be called from a C program which supplies the values for
; buffer moves.
;------------------------------------------------------------------------
;
TITLE Data access
SUBTTL 22/2/88
NAME dt_mov
;
INCLUDE DOS.MAC
      PUBLIC  mov_buf,res_buf,readpixl,wrtpixl,ltime



;
;     DATA items now declared and descriptor tables set up.
; For details of descriptor table see IBM tech. ref. pp 5-149,5-150.


;
DSEG

;  addresses used for control purposes
;
base_add equ 250h
incsr1  equ base_add ; Video Input Control/Status 1
incsr2  equ base_add  +  2 ; Video Input Control/Status 2
outcsr  equ base_add  +  4 ; Video Output Control/Status
cursor  equ base_add  +  6 ; Cursor control
index  equ base_add  +  8 ; Contains the LUT index
inlut  equ base_add  +  0aH ; Contains LUT entry
redgrn  equ base_add  +  0cH ; Contains the red & green LUT
blue  equ base_add  +  0eH ; Contains the blue LUT entry
;
;------------------------------------------------------------------------


;
; The following is the descriptor table for reading from the frame store
;
 dummy  DQ 0

 gdt_loc  DQ 0

 src_seg_limit DW 0ffffH
 src_base_add DB ?,?,?; DT base address = A00000H
;      To be filled by pointer
 src_dat_ri DB 93H
 src_dat_res DW 00H

 tgt_seg_limit DW 0ffffh
 tgt_base_add DB ?,?,?; To be filled by pointer
 tgt_dat_ri DB 93H
 tgt_dat_res DW 00H
```

```
 bios_cs  DQ 0

 temp_ss  DQ 0
;
;
;
 gdt      DD dummy ; gdt points (seg,offset) to top
;      of table
 read_add  DW src_base_add ; points(offset) to source
;      base address
  point_tgt       DW      tgt_base_add ; pointer defined for
;      filling target address
          frames          DW      ?
;------------------------------------------------------------------------
; Write descriptor table now setup
;
 wdummy  DQ 0
;
 wgdt_loc DQ 0

 wsrc_seg_limit DW 0ffffH
 wsrc_base_add DB ?,?,?; Equivalent to tgt_base_add
 wsrc_dat_ri DB 93H
 wsrc_dat_res DW 00H
;
 wtgt_seg_limit DW 0ffffh
 wtgt_base_add DB ?,?,?; Dt board;To be filled by pointer
 wtgt_dat_ri DB 93H
 wtgt_dat_res DW 00H
;
 wbios_cs DQ 0
;
 wtemp_ss DQ 0
;
;
 write_add DW wtgt_base_add ;pointer to DT board
;      address location
 wgdt     DD wdummy ; wgdt points (seg,offset) to top
;      of table
 wrt_src_add  DW wsrc_base_add ; points(offset) to source
;      base address
;
;
ENDDS
;
;------------------------------------------------------------------------
 PSEG  ; program segment starts
;
; This is function "mov_buf(buff)" which moves DT buffer "buff"
; to system memory.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.


BEGIN mov_buf
push bp  ; save caller's stack frame pointer
mov bp,sp  ; establish this function's stack
mov bx,read_add ; bx points to base address of DT
```

```
mov ax,00H
mov [bx],ax  ; put low word of pixel address into GDT
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a0H
 mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,point_tgt ; bx points to target base address
mov ax,0c000H
mov [bx],ax   ;
mov al,05H
mov [bx+2],al ; low word, high byte

les si,gdt  ; es
.SI
now points to top of GDT
;     BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
int 15H  ; BIOS interrupt
;
mov bx,read_add ; bx points to base address in GDT
mov ax,00h
mov [bx],ax  ; put low word of pixel address into GDT
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a1H
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,point_tgt ; bx points to target base address
mov ax,0c000h
mov [bx],ax   ;
mov al,06h
mov [bx+2],al ; low word, high byte

les si,gdt  ; es
.SI
now points to top of GDT
;     BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
int 15H  ; BIOS interrupt
;
mov bx,read_add ; bx points to base address in GDT
mov ax,00h
mov [bx],ax  ;
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a2H
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;   ; 24 bit address of target now calculated
;
mov     bx,point_tgt ; bx points to target base address
mov ax,0c000h
```

```
 mov [bx],ax  ;
 mov al,07h
 mov [bx+2],al ; low word, high byte

 les si,gdt  ; es
.SI
 now points to top of GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,8000H ; number of pixels to be moved:64k
 int 15H  ; BIOS interrupt
;
 mov bx,read_add ; bx points to base address in GDT
 mov ax,00h
 mov [bx],ax  ; put low word of pixel address into GDT
 mov ax,[bp+6] ; fetch buffer no.
 mov cl,4
 shl ax,cl  ; Do adjustment for buffer number
 add al,0a3H
 mov [bx+2],al ; put hi byte pixel address calculated into GDT
;   ; 24 bit address of target now calculated
;
 mov     bx,point_tgt ; bx points to target base address
 mov  ax,0c000h
 mov [bx],ax   ;
 mov al,08h
 mov [bx+2],al ; low word, high byte
 les si,gdt  ; es
.SI
 now points to top of GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,8000H ; number of pixels to be moved:64k
 int 15H  ; BIOS interrupt
 pop bp
 ret
mov_buf ENDP
;
;--------------------------------------------------------------------------
; This is function "res_buf(buff)" which restores the buffer from system
; memory to buffer buff on DT2853.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.
;--------------------------------------------------------------------------
;
 BEGIN res_buf
 push bp  ; save caller's stack frame pointer
 mov bp,sp  ; establish this function's stack
 mov bx,write_add ; bx points to base address in GDT
 mov ax,00H
 mov [bx+0],ax ; put low word of pixel address into GDT
 mov ax,[bp+6] ; fetch buffer no.
 mov cl,4
 shl ax,cl  ; Do adjustment for buffer number
 add al,0a0H
  mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
 mov     bx,wrt_src_add ; bx points to target base address
```

```
mov ax,0c000H
mov [bx],ax   ;
mov al,05H
mov [bx+2],al ; low word, high byte

les si,wgdt  ; es
.SI
now points to top of GDT
;     BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
int 15H  ; BIOS interrupt
;
mov bx,write_add ; bx points to base address in GDT
mov ax,00h
mov [bx],ax  ; put low word of pixel address into GDT
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a1H
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,wrt_src_add ; bx points to target base address
mov ax,0c000h
mov [bx],ax   ;
mov al,06h
mov [bx+2],al ; low word, high byte

les si,wgdt  ; es
.SI
now points to top of GDT
;     BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
int 15H  ; BIOS interrupt
;
mov bx,write_add ; bx points to base address in GDT
mov ax,00h
mov [bx],ax
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a2H
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,wrt_src_add ; bx points to target base address
mov ax,0c000h
mov [bx],ax  ;
mov al,07h
mov [bx+2],al ; low word, high byte

les si,wgdt  ; es
.SI
now points to top of GDT
;     BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
```

```
int 15H  ; BIOS interrupt
;
mov bx,write_add ; bx points to base address in GDT
mov ax,00h
mov [bx],ax  ; put low word of pixel address into GDT
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
shl ax,cl  ; Do adjustment for buffer number
add al,0a3H
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,wrt_src_add ; bx points to target base address
mov  ax,0c000h
mov [bx],ax   ;
mov al,08h
mov [bx+2],al ; low word, high byte

les si,wgdt  ; es
.SI
now points to top of GDT
;      BIOS call now set up
mov ah,87H  ; BIOS function 87H
mov cx,8000H ; number of pixels to be moved:64k
int 15H  ; BIOS interrupt
pop bp
ret
res_buf ENDP
;
;---------------------------------------------------------------------------
;
; This is function "mov_blk(buff,blk)" which moves DT block "blk" in
; "buff" to system memory.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.


BEGIN mov_blk
push bp  ; save caller's stack frame pointer
mov bp,sp  ; establish this function's stack
mov bx,read_add ; bx points to base address of DT
mov ax,00H
mov [bx],ax  ; put low word of pixel address into GDT
int  3
mov ax,[bp+6] ; fetch buffer no.
mov cl,4
mul cl
add al,0a0H
add al,[bp+8] ; add block number to al
mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
mov     bx,point_tgt ; bx points to target base address
mov ax,0c000H
mov [bx],ax   ;
mov al,08H
mov [bx+2],al ; low word, high byte

les si,gdt  ; es
.SI
```

```
 now points to top of GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,8000H ; number of pixels to be moved:64k
 int 15H  ; BIOS interrupt
;
 pop bp
 ret
mov_blk ENDP
;
;--------------------------------------------------------------------------
; This is function "res_blk(buff,blk)" which restores DT block "blk" in
; "buff" from system memory.
; It utilises BIOS INT 15H, Function 87H, to access the frame store.


 BEGIN res_blk
 push bp  ; save caller's stack frame pointer
 mov bp,sp  ; establish this function's stack
 mov bx,write_add ; bx points to base address of DT
 mov ax,00H
 mov [bx],ax  ; put low word of pixel address into GDT
 mov ax,[bp+6] ; fetch buffer no.
 mov cl,4
 mul cl
 add al,0a0H
 add al,[bp+8] ; add block number to al
  mov [bx+2],al ; put hi byte pixel address calculated into GDT
;
 int 3
 mov     bx,wrt_src_add ; bx points to target base address
 mov ax,0c000H
 mov [bx],ax   ;
 mov al,08H
 mov [bx+2],al ; low word, high byte

 les si,wgdt  ; es
.SI
 now points to top of GDT
;     BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,8000H ; number of pixels to be moved:64k
 int 15H  ; BIOS interrupt
;
 pop bp
 ret
res_blk ENDP
;
;--------------------------------------------------------------------------
; Function wrtpixl(x,y,z): writes value z to location x,y in buffer
; currently in system memory
;
 BEGIN wrtpixl
 push bp
 mov bp,sp
  mov ax,200H  ;length of one line
 mov bx,[bp+8]
```

```
 mul bx  ;result in DX:AX
 mov cl,12
 shl dx,cl
 mov bx,ax  ;temp store of ax
 mov cl,4
 shr ax,cl
 add dx,ax
 add dx,5c00H ;include memory offset
 mov ax,bx  ; restore ax
 and ax,000fh ; clear top 12 bits of ax
 mov bx,[bp+6]
 add bx,ax  ;
  mov al,[bp+10] ;value to be written
  push  ds
 mov ds,dx
 mov [bx],al
 pop ds
  pop  bp
 ret
wrtpixl ENDP
;
;-----------------------------------------------------------------------
; Function readpixl(x,y): reads value at location x,y in buffer
; currently in system memory.
;
 BEGIN readpixl
 push bp
 mov bp,sp
  mov ax,200H  ;length of one line
 mov bx,[bp+8]
 mul bx  ;result in DX:AX
 mov cl,12
 shl dx,cl
 mov bx,ax  ;temp store of ax
 mov cl,4
 shr ax,cl
 add dx,ax
 add dx,5c00H ;include memory offset
 mov ax,bx  ; restore ax
 and ax,000fh ; clear top 12 bits of ax
 mov bx,[bp+6]
 add bx,ax  ;
  push ds
 mov ds,dx
 mov ax,[bx]
 xor ah,ah
 pop ds
  pop bp
 ret
readpixl ENDP
;
;-----------------------------------------------------------------------
; Function wrt_blk(x,y,z): writes value z to location x,y in block
; currently in system memory
;
 BEGIN wrt_blk
 push bp
```

```
 mov bp,sp
  mov ax,200H  ;length of one line
 mov bx,[bp+8]
 mul bx  ;result in DX:AX
 mov cl,12
 shl dx,cl
 mov bx,ax  ;temp store of ax
 mov cl,4
 shr ax,cl
 add dx,ax
 add dx,8c00H ;include memory offset
 mov ax,bx  ; restore ax
 and ax,000fh ; clear top 12 bits of ax
 mov bx,[bp+6]
 add bx,ax  ;
  mov ax,[bp+10] ;value to be written
  push ds
 mov ds,dx
 mov [bx],ax
 pop ds
  pop  bp
 ret
wrt_blk ENDP
;
;-------------------------------------------------------------------------
; Function read_blk(x,y,): reads value at location x,y in block
; currently in system memory.Returns value in AL.
;
 BEGIN read_blk
 push bp
 mov bp,sp
  mov ax,200H  ;length of one line
 mov bx,[bp+8]
 mul bx  ;result in DX:AX
 mov cl,12
 shl dx,cl
 mov bx,ax  ;temp store of ax
 mov cl,4
 shr ax,cl
 add dx,ax
 add dx,8c00H ;include memory offset
 mov ax,bx  ; restore ax
 and ax,000fh ; clear top 12 bits of ax
 mov bx,[bp+6]
 add bx,ax  ;
  push ds
 mov ds,dx
 mov ax,[bx]
 pop ds
 xor ah,ah
  pop  bp
 ret
read_blk ENDP

;-------------------------------------------------------------------------
;
;
```

```
BEGIN ltime
     int    3
push bp
 mov bp,sp


;
;
;
;   set up source and target base addresses
mov     ax,200H          ; length of one row - 512 pixels
      mov  bx,[bp+6]       ; bx = y position
      mul    bx            ; calculate y posn in memory result in DX:AX
      add    dx,0a0h       ; include buffer0 offset address (a00000H)
mov bx,read_add  ; bx points to base address of DT
mov [bx],ax    ; put low word of pixel address into GDT
 mov [bx+2],dl       ; put hi byte pixel address calculated into GDT
;
mov     bx,point_tgt ; bx points to target base address
mov ax,0c000H
mov [bx],ax   ;
mov al,05H
mov [bx+2],al ; low word, high byte

      mov     frames,000000H   ; frame count set to zero




;
mov dx,incsr1
mov ax,08H
out dx,ax  ; Set ENSTOP
out dx,ax  ; Clear BUSY
check_b:
in  ax,dx
and ax,080H  ; Check BUSY is clear
jnz check_b
;
;   value for inscr2 now calculated
 mov dx,incsr2
mov ax,0ff10H ; MODE = 001, no write protect
      out dx,ax
mov dx,outcsr
mov ax,0000111110100000b; Set DISPLAY,EXTERNAL TIMING,O/P BUF=0
out dx,ax     ;
;
mov dx,incsr1
mov ax,0ff82H ; Set BUSY, Set   ENSTOP; Input LUT = 002
out dx,ax           ; board set to single continuously
;
;
next_frame:
      mov   dx,outcsr
check_b2:
```

```
 in  ax,dx  ; poll vsync and field  - bit 15/13
 and ax,0a000H ; vsync set and odd field ?
  cmp     ax,08000H
 jne check_b2        ; no - wait
                           ; yes - continue
    ; get the data
 les si,gdt  ; es
.SI
 now points to top of GDT
;      BIOS call now set up
 mov ah,87H  ; BIOS function 87H
 mov cx,400H        ; number of double pixels to be moved
 int 15H  ; BIOS interrupt
;
        inc     frames          ; update frame count
    mov    dx,[bp+8]       ; check frame count
        cmp     dx,frames       ; all done ?
        je      fini            ; yes - finished
                           ; no - update target address


        mov     bx,point_tgt    ; get low word
        mov     ax,[bx]
        mov     dl,[bx+2]       ; get high, low byte
        add     ax,800H         ; add 2k - pixel count for 4  rows
        mov     [bx],ax         ; return result to pointer low word
        adc     dl,000H         ; add any carry to high word, low byte
        mov     [bx+2],dl       ; return result to pointer
        jmp     next_frame
fini:
        pop bp
        ret
ltime   ENDP

        ENDPS
        END
```

```
;
TITLE Clock access
SUBTTL 1/12/88
NAME clock
;
INCLUDE DOS.MAC
      PUBLIC  time,date


;
;
;-----------------------------------------------------------------------
  PSEG  ; program segment starts
;


 BEGIN time
 push bp  ; save caller's stack frame pointer
  mov     bp,sp
      mov ah,02h          ; ah  =  3; read real time clock
      int      1ah          ; interrupt -time of day
      mov     ah,00h
      mov     al,ch          ; hours (BCD) into al
      mov     bh,cl          ; minutes (BCD) into bh
      mov     bl,dh          ; seconds (BCD) into bl
   ; result returned to C as:
                              ; bit31                        bit0
      pop bp               ;     00 | hours | minutes | seconds
 ret

time ENDP


;-----------------------------------------------------------------------

       BEGIN  date
       push   bp  ; save caller's stack frame pointer
       mov    bp,sp

    mov ah,04h  ; ah = 4; read date from clock
int      1ah  ; interrupt - time of day
 mov ax,cx  ; year(BCD) into ax
      mov bx,dx  ; month(BCD)  and day(BCD)  into bx
   ; result returned to C as
               ; bit 31
   ;  century | year | month | date
 pop bp


ret

date  ENDP

;-----------------------------------------------------------------------
      ENDPS      ; PROGRAM SEGMENT ENDS

      END
```

APPENDIX VI

C PROGRAM CODE FOR THE SHORE SYSTEM

function: calibration (scale,centre)

non-pointer arguments: none

pointer arguments:

	float scale	horizontal scaling factor for the image in mm/pixel
	centre	column containing the guide tube centre line in the frame store image buffer.
returns:	none	
called functions:	cursor()	

description:

This function displays the calibration menu, containing the "set centre line" and "set scaling" options. The first option allows the position of the guide tube centre line in the captured image to be passed to the C program by the user. A grey scale image is first captured and a moveable cursor then superimposed on the output image by calling the cursor function. The user is then prompted to place the cursor at the guide tube centre as seen on the output monitor.

If the set scaling option is selected then the user is given the option of either typing in the vertical scaling factor in mm/pixel or determining it interactively using a pair of cursors. In the latter case, the cursor() function is called twice to establish two reference points on the test rig scale, as seen on the output monitor. The user is then instructed

to enter the scale values at these two locations. The scale factor can be calculated using the formula:-

$$\text{scale factor} = \frac{v1 - v2}{x1 - x2} \qquad [A6.1]$$

where v1 and v2 are the scale readings corresponding to the cursor locations (x1,y) and (x2,y).

function: cross_section (buffer, hsection, co_ord)

non_pointer arguments:

	int buffer	number of frame store buffer containing the image data
	int hsection	TRUE: if horizontal cross section required, FALSE: if a vertical cross section is required.
	int co_ord	number of required row or column containing the image data
pointer arguments:	none	
returns:	none	
called functions:	none	

description:

This function produces a plot of grey level along a row or column of a captured image on the computer monitor. The function makes extensive use of the HALO software library to supply the graphics functions. These routines are highly specific and will not

be discussed in detail (for further information see [46]). Essentially, the axes are plotted first, followed by the tick marks and finally the data.

function: cursor (x,y,buffer,two_pass,pass)

non-pointer arguments:

	int buffer	displayed image buffer
	two_pass	1 - if first cursor is to be placed; 2-if second cursor is to be placed.
	int pass	TRUE - if two cursors are to be placed; FALSE - if only one cursor is to be placed.

pointer arguments:

	int x	cursor column number
	int y	cursor row number
returns:	none	
called functions:	none	

description:

The function produces a "+" shaped cursor, 40 pixels square in size. It's movement can be controlled using the w, a, s and z keys on the keyboard with the SHIFT key used to speed up the movement and the "*" key is used to mark the final cursor position. In the case of a single cursor, the following algorithm is used to move the cursor:

```
<store pixel values at present cursor location>
<over write cursor at present location>

WHILE (<key pressed> AND <key is not "*" key >)
  IF (<key is a movement key>)
     <restore pixel values at present location>
     <get direction of movement>
     <up-date co-ordinates of cursor>
     <store pixel values at new cursor location>
     <over write cursor at new cursor location>
  ENDIF
ENDWHILE

<restore pixel values at current location>
```

If two cursors are to be placed the first cursor is not overwritten at the end of the first function call. This ensures that both cursors are visible on the output image when the second cursor is placed. Once the second cursor has been placed, both cursors are overwritten by the original pixel values. The grey scale values of the pixels covered by the first cursor are stored in a global array so that they may be recalled on the second function call.

function: def()

arguments: none

returns none

called functions: none

description:

This function sets up the initial contents of the input LUTs on the frame store. LUT0 is configured to give a direct grey scale mapping, LUT1 an inverted grey scale mapping, and LUT2 a binary mapping (for thresholding operations).

function histogram (buffer,xminm,xmaxm,ymaxm)

non-pointer arguments:

	int buffer	image buffer containing the required data
	int xminm	co-ordinates of the lower left
	int yminm	hand window corner
	int xmaxm	co-ordinates of the upper
	int ymaxm	right hand window corner
pointer arguments:	none	
returns:	none	
called functions:	none	

description:

This function produces a grey level frequency plot, for a captured image, on the computer monitor. The histogram frequency table, contained within an array, is calculated by reading the grey level of every pixel, located within the specified window, from a previously down-loaded image frame. From this data the mode grey level and number of pixels having this grey level are determined. The function makes extensive of the HALO software library to supply the graphics functions. These routines are highly specific and will not be discussed in detail (for further information see [46]);

essentially, the axes are plotted first, followed by the tick mark and the histogram bars. Finally the mode grey level and number of pixels having this grey level are given.

function: measurement (centre,scale)

non-pointer arguments:

	int centre	co-ordinate of guide tube centre line
	float scale	camera scaling factor in mm/pixel
pointer arguments:	none	
returns:		
	float shorenum	calculated Shore number
called functions:	none	

description:

This function contains the image analysis software for measuring the rebound height of the indenter. The assembler function ltime() is first called so that 128 consecutive frames are captured from the video camera. As indicated in section 3.4.2, the assembler function transfers the contents of four rows of pixels, from each captured frame, to base memory to form a contiguous data block which is analysed by the measurement function. The window limits are calculated as in section 3.3.2. If the left hand limit cannot be found then an error is signalled and the function is exited, however, if the right hand limit cannot be found a worst-case estimate is instead used (Appendix II). The rebound height is found in the same manner as described in section 3.3.2, if either the rising or falling part of the initial rebound cannot be found then an error is signalled and

the function is exited. Finally, if no errors are found, the Shore number is calculated using the following formula (see section 2.2.4):

$$\text{Shore number} = 140 \times \frac{250}{h \times scale} \qquad [A6.2]$$

where *scale* is the scaling factor of the camera in mm/pixel and *h* is the rebound height in pixels.

function: measmenu(scale,centre)

non-pointer arguments:

	int scale	horizontal scaling factor for the image in mm/pixel
	int centre	row containing the guide tube centre line in the image array
pointer arguments:	none	
returns:	none	
called functions:	measurement()	

description:

This function displays the measurements menu containing the "take a single measurement", "take multiple measurements" and "write results to file" options. The measurement options both use the measurement() function to return the Shore hardness for a given test. A returned value of -1 indicates that an error has occurred and in such cases the measurement is repeated.

If the "take multiple measurements" option is selected the user is prompted for the number of measurements to be taken and the measurement() function called the required number of times. The Shore hardness obtained from each measurement is stored in an array from which the group mean and standard deviation are calculated according to:-

$$\text{mean,} \quad \bar{x} = \frac{1}{N}\sum_{i=1}^{i=N} x_i \qquad [A6.3]$$

$$\text{standard deviation,} \quad \sigma = \frac{1}{N-1}\sum_{i=1}^{i=N} |\bar{x} - x_i| \qquad [A6.4]$$

where x_i is the result of the ith measurement and N the number of results in the group.

The "write results to file" option allows the results of single or group measurements to be included in a file stored on the hard disk. The code used to perform this operation makes extensive use of Lattice "C" library functions [41], not all of which are present in the Kernigan and Ritchie [35] "standard".

The user is first prompted for a file name and the specified file opened for read/ write access using the library function "open" which returns the file handle (channel number). If the named file does not already exist then a new file is created. A header containing a file description string, supplied by the user, is then written to file using the "write" library function. In order that the results may be referenced at a later date, the present date and time are written to file.

The, user-supplied, date() and time() functions return the date and time as long integers. The C program breaks down the date and time variables into short (two byte) integers

containing year/month/date and hours/minutes/seconds respectively. These variables are then converted to string form before being recorded on file. The measurement results are then written to file in string form. The conversion process involves an intermediate step of changing the results in integer form to floating point, using a cast, before conversion to a string using the "ecvt" library function. This is because there is no library function supplied by Lattice to convert an integer directly into a string. Finally the results file is closed using the "close" library function.

function: parameters( )

arguments: none

returns: none

called functions: thresh2(), section(), histogram()

description:

This function displays the parameters menu, containing the "histogram", "grey level cross-section" and "alter thresholds" options. If the "histogram" option is selected then the capture() function is first called causing a single image frame to be digitised. The histogram() function is then called to produce a grey level frequency plot for the captured image. If the "grey level cross-section" option is selected a frame is captured and the cross_section() function called. This produces a plot of grey level against co-ordinate for a row or column of the image array. Finally, if the "alter thresholds" option is selected, then the thresh2() function is called. This produces a real time binary image for which the threshold limits can be altered interactively.

function: section(buffer)

non-pointer arguments:

	int buffer	buffer containing the required image
pointer arguments:	none	
returns	none	
called functions:	cross_section(), cursor()	

description:

This function simply prompts the user to indicate whether a plot of grey level against co-ordinate for a captured image is required in the horizontal or in the vertical direction. The cross_section() function is then called with the hsection argument set according to whether a vertical or horizontal section is required.

function: threshl (lower,upper)

non-pointer arguments: none

pointer arguments:

	int lower	lower threshold limit
	int upper	upper threshold limit
returns:	none	
called function:	none	

description:

This function writes to input LUT2, using the ilut() function, to give a binary grey scale mapping. Each input grey scale value g is mapped to a new value g' according to:-

$$g' = \begin{cases} 255, & t_1 < g < t_2 \\ 0, & t_1 \geq g \\ 0, & t_2 \leq g \end{cases} \qquad [A6.5]$$

where t_1 and t_2 are the lower and upper threshold limits, respectively.

function: thresh2()

arguments:	none
returns:	none
called functions:	thresh1()

description:

This function produces a real time binary image on the output monitor and allows the user to alter the threshold limits interactively. The user is first prompted to supply the upper and lower threshold limits and the thresh1() function is then called with these values as arguments. The thresh1() function writes to input LUT2 on the frame store to give the required grey scale mapping and the camera() function is used to produce a real time binary image. The user is then asked whether the threshold limits are to be altered. If they are then the limits can be changed interactively using the d, a, l and j keys on the keyboard; otherwise the function is exited. In the former case the program waits until a valid keystroke is detected and then updates the limits as follows:-

a decrease lower limit

d increase lower limit

j decrease upper limit

l increase upper limit

Traps are included to ensure that the lower limit is greater than or equal to zero, the upper limit is less than 255 and that the upper limit is always greater than the lower limit.

```
.li on
#include  "\lc\stdio.h"
#include  "\lc\dos.h"
#include  "\lc\math.h"
#include  "\lc\fcntl.h"
#include  "\lc\string.h"

#define   TRUE    1
#define  FALSE      0
#define         BLACK                     0
#define         WHITE                     255


extern void camera(); /* real time display ( i/p buffer, i_lut, o/p buffer, o_lut ) */
extern  void    capture(); /* real time display ( i/p buffer, i_lut, o/p buffer, o_lut )
 */
extern int read_pix(); /* read pixel from buffer ( x, y, buffer ) */
extern void wrt_pix(); /* write pixel ( x, y, value, buffer ) */
extern void olut();  /* set the output lut ( table, index, value ) */
extern  void    ilut();  /* set the input lut ( table, index, value ) */
extern void mov_buf(); /* moves a whole frame (buff) */
extern void res_buf(); /* restores a whole frame (buff) */
extern void mov_blk(); /* moves a section (buff,section) */
extern void res_blk(); /* restores a section (buff,section) */
extern void wrtpixl(); /* writes to the local buffer (x,y,value) */
extern int readpixl();  /* reads the pixel from the local buffer (x,y) */
extern  void    pcapture();     /* as for capture, ms nibble write protected  */
extern  void    fback();        /* feeds back a frame through the lut - same args as cam
era */
extern  void    show_buf();     /* displays captured frame show_buf (buffer,olut)*/
extern  void    sync();         /* select internal sync  */
extern  void    turn_on();
extern  void    turn2();
extern  void    turn3();
extern  void    cam2();
extern  void    ltime();
extern  long    date();
extern  long    time();


float   measurement();

main()
{
  int  option;
  int  error;
  int  centre = 205;
  float scale = 0.0;
      def ();

      do
      {
        error = system ("CLS");
        printf ("                    OPENING MENU \n");
        printf ("                    = = = = = = = =  = = = =  \n\n\n");
        printf ("              r   display Real time image  \n\n");
        printf ("              p   adjust  Parameters       \n\n");
```

```
        printf ("           c  Calibration           \n\n");
        printf ("           m  Measurements             \n\n");
        printf ("           q  quit                  \n\n");
        option = getch();
        error  = system("cls");
        switch ( option )
        {
            case 'r':  camera(0,0,0,0);
                       break;
            case 'p':  parameters();
                       break;
            case 'm':  measmenu(centre,scale);
                       break;
            case 'c':  calibration(&centre,&scale);
                       break;
        }
    } while ( option .

}  /* end of main */



measmenu(centre,scale)
int centre;
float scale;
{
   int  option;
   int  error;
   float hard;
   int     nvalues;
   float   value[100];
   float   mean,stddev;
   int     i;
   float   total;
   int     file;
   char    *filename;
   int     mode;
   int     status;
   int     length;
   char    *p;
   double  dvalue;
   int     ndig;
   int     *dec = 0;
   int     *sign = 0;
   char    *dummy;
   int     j;
   int     valnum;
   int     hours,minutes;
   int     year,month,day;

      do
      {
        error = system ("CLS");
        printf ("                 MEASUREMENTS MENU \n");
        printf ("                 = = = = = = = = = = = = =  = = = =  \n\n\n");
        printf ("           s  take a Single measurement\n\n");
        printf ("           m  take Multiple measuremnts\n\n");
```

```
        printf ("            w   Write results to file    \n\n");
        printf ("            q   Quit - return to main menu  \n\n");
        option = getch();
        error  = system("cls");
        switch ( option )
        {
            case 's':  hard = measurement(centre,scale);
                      printf (" --- press any key to continue --- \n");
                      error = getch();
                      break;
            case 'm':  printf (" number of measurements to be taken <ret> ?\n");
                      scanf ("%d",&nvalues);
                      printf ("\n\n\n");
                      for ( i=1; i < = nvalues; i+ + )
                      {
                          printf ("\n\n ***** measurement %d of %d *****\n\n",i,nva
lues);
                          value [i] = measurement (centre,scale );

                          if ( value[i] < 0.0 )
                              i--; /* trap any erroneous results */
                      }

                      /* calculate mean */

                      total = 0.0;
                      for ( i = 1; i< = nvalues; i+ + )
        total = value[i] + total;
                      mean = total / (float)nvalues;


                      /* calculate standard deviation */

                       total = 0.0;
                      for ( i = 1; i< = nvalues; i+ + )
                      total = ((value[i] - mean) > 0 ) ? value[i]-mean+total : mea
n -value[i] + total;
                      stddev= total / (float)nvalues;
                      printf (" mean value = %6.1f \n\n",mean);
                      printf (" standard deviation = %6.1f \n\n",stddev);
                      printf (" --- press any key to continue --- \n");
                      error=getch();
                      break;
            case 'w':  /* open results file - read / write access */
                      /* seek end of file before each write      */

                      error=system("CLS");
                      printf ("\n  name of results file ?");
                      filename = gets (dummy);
                      mode = O_RDWR | O_APPEND | O_CREAT;
                      file = open ( filename, mode );

                      /* write out the header */

                      dummy = "\n\n\n************************************************
**********************\n\n";
```

```
                    length = strlen(dummy);
                    status = write ( file, dummy, length );

                    printf (" enter identifier for the results \n");
   p = gets(dummy);
                    length = strlen ( p );
                    status = write ( file, p, length );

                    dummy = "\n\n****************************************************
*******************\n\n\n";
                    length = strlen (dummy);
                    status = write ( file, dummy, length );

    /* write the date and time */

  /* extract date first */

  year = (  date() & 0xFFFF0000 ) / 0x10000;
   year =  1000 * ( ( year & 0xF000 ) / 0x1000 ) +
                      100  * ( ( year & 0x0F00 ) / 0x100  ) +
                      10   * ( ( year & 0x00F0 ) / 0x10   ) +
                      ( year & 0x000f );

  month = ( date() & 0x0000FF00 ) / 0x100;
   month =  10 * ( ( month & 0xF0 ) / 0x10 ) +
                      ( month & 0x0f );

  day  = date() & 0x000000FF ;
   day =  10 * ( ( day & 0xF0 ) / 0x10 ) +
                      ( day & 0x0f ) ;

   /* now the time */

  hours = (  time() & 0xFFFF0000 ) / 0x10000;
   hours =  1000 * ( ( hours & 0xF000 ) / 0x1000 ) +
                       100  * ( ( hours & 0x0F00 ) / 0x100  ) +
                       10   * ( ( hours & 0x00F0 ) / 0x10   ) +
                      ( hours & 0x000f );

  minutes = ( time() & 0x0000FF00 ) / 0x100;
   minutes =  10 * ( ( minutes & 0xF0 ) / 0x10 ) +
                      ( minutes & 0x0f );

  /* write time and date to file */

  dummy = " results recorded on  ";
                   length = strlen(dummy);
                   status = write ( file, dummy, length );


   length = stcu_d ( dummy, day );
                   status = write ( file, dummy, length );

                   dummy = "/";
                   status = write ( file, dummy, 1 );
```

```
length = stcu_d ( dummy, month );
                status = write ( file, dummy, length );

                dummy = "/";
                status = write ( file, dummy, 1 );


/*    length = stcu_d ( dummy, year, 5 );
                status = write ( file, dummy, length );

                dummy = "   at   ";
                length = strlen(dummy);
                status = write ( file, dummy, length );
*/


length = stcu_d ( dummy, hours );
                status = write ( file, dummy, length );

                dummy = ":";
                status = write ( file, dummy, 1 );


                if ( minutes < 10 )
{
   dummy="0";
   status = write ( file, dummy, 1 );
}

length = stcu_d ( dummy, minutes );
                status = write ( file, dummy, length );

                dummy = "\n\nSHORE NUMBERS\n\n";
length = strlen(dummy);
                status = write ( file, dummy, length );


                error = system("cls");
                printf (" writing to file %s ",filename);
               /* print out the measurements */

                nvalues = 10;

                for ( valnum=1; valnum < = nvalues; valnum+ + )
                {
                   dvalue=3.14159;
                   ndig=8;

                   /* convert shore number (float) to a string */

                   p="";
                   dummy="";
                   p = ecvt ( dvalue, ndig, dec, sign );

                   /* include the decimal point */
                   i = j = 0;
```

```
                    do
                    {
          = *dec )     if ( i
                          *(dummy+i) = *(p+j);
                       else
                       {
                          *(dummy+i) = '.';
                          j--;
                       }
                       i++;
                       j++;
                    } while ( i <= ndig );
                    *(dummy+i)= '\n';

                    /* write string to file */

                    length = 10;
                    status = write ( file, dummy, length );


                 } /* end of write measurements loop */


                 break;
         }
      } while ( option

   } /* end of measmenu */


   parameters()
   {
     int option;
     int error;

       do
       {
         error = system ("CLS");
         printf ("                    PARMETERS MENU \n");
         printf ("                    ========== ==== \n\n\n");
         printf ("            t   alter thresholds    \n\n");
         printf ("            g   grey level cross section  \n\n");
         printf ("            h   histogram             \n\n");
         printf ("            q   return to main menu  \n\n");
         option = getch();
         error  = system("cls");
         switch ( option )
         {
             case 't':  thresh2();
                        break;
             case 'g':  capture(0,0,0,0);
                        section(0);
                        break;
             case 'h':  capture(0,0,0,0);
                        histogram(0,0,0,511,511);
                        break;
```

```
        }
    } while ( option

} /* end of parameters */


calibration(centre,scale)
float *scale;
int *centre;
{
int x,y;
int option;
int error;
int xmax, xmin;
int maxmm,minmm;
    do
    {
      error = system ("CLS");
      printf ("                        CALIBRATION MENU \n");
      printf ("                        = = = = = = = = = = = = =  = = = = \n\n\n");
      printf ("            c  set Centre line    \n\n");
      printf ("            s  set Scaling         \n\n");
      printf ("            q  return to main menu  \n\n");
      option = getch();
      error  = system("cls");

      if ( option = = 'c' )
      {
        capture (0,2,0,0);
        printf (" use cursor to mark the centre of the ball at rest \n\n");
        x=y=256;
        cursor (&x,&y,0,FALSE,1);
        *centre=y;
      }
      if ( option = = 's' )
      {
        printf (" use cursor (y/n)? \n");
        x= getch();
        if ( x= ='y' )
        {
          capture (0,0,0,0);
          printf (" use cursor to locate largest value on graticule\n\n");
          x=y=256;
          cursor (&x,&y,0,FALSE,1);
          xmin=x;
          printf (" enter the value in mm  <ret> \n\n");
          scanf ("%d",&maxmm);
          printf (" use cursor to locate smallest value on graticule \n\n");
          x=y=256;
          cursor (&x,&y,0,FALSE,1);
          xmax=x;
          printf (" enter the value in mm  <ret> \n\n");
          scanf ("%d",&minmm );

          /* scale factor in mm per pixel */
```

```
            *scale =  ( (float)maxmm  - (float)minmm ) / ( (float)xmax - (float)xmin
);

            printf (" scale factor = %f  mm / pixel \n\n",*scale);
            printf (" any key to continue ... \n");
            x=getch();
          }
          else
          {
            printf (" enter scale factor in mm / pixel ");
            scanf  ("%f",scale);
          }
        }
      } while ( option



}





section (buffer)
int buffer;
{
 int x,y;
 int option;

   printf ("          GREY LEVEL CROSS SECTION\n\n\n");
   printf ("            h  - horizontal section     \n\n ");
   printf ("            v  - vertical section       \n\n ");


   x=y=256;
   do
   {
      switch ( option = getch() )
      {
         case 'h' :  printf (" use cursor to indicate required ordinate");
                     cursor (&x,&y,buffer,FALSE,1);
                     cross_section (buffer,TRUE,y);
                     break;
         case 'v' :  printf (" use cursor to indicate required abcissa ");
                     cursor (&x,&y,buffer,FALSE,1);
                     cross_section (buffer,FALSE,x);
                     break;
      }
   } while (( option ) && ( option     =  'h' ) );
```

```
}


def ()
{
  int  i;


      /* set up luts */

      /* LUT 0 - direct grey scale mapping */

      for ( i= 0; i < = 255; i+ + )
        ilut (0,i,i);

     /* LUT 1 - inverted grey scale */

      for ( i = 0; i < = 255; i+ + )
        ilut (1,i,255-i);

     /* LUT 2 - thresholding */

      for ( i = 0; i < 128; i+ + )
        ilut (2,i,255);
      for ( i = 128; i < = 255; i+ + )
        ilut (2,i,0);

     /* LUT 3 - XORing */

      ilut (3,0,0);
      ilut (3,15,255);
      ilut (3,240,255);
      ilut (3,255,0);


      camera (0,0,0,0 );


} /* end of default */




/**********************************************************************/
/*                                                                    */
/* name:      thresh2( lower, upper, n )                              */
/* sends:     lower  - initial lower threshold bound  | MUST BE       */
/*            upper  - initial upper threshold bound  | POINTERS      */
/* returns:   lower  - final lower threshold bound                    */
/*            upper  - final threshold bound                          */
/* synopsis:  as thresh1 but thresholds can be changed interactively  */
/*            using a d l j keys.                                     */
/*            lut n.                                                  */
/*            *upper > *lower    0 < = n < = 6                        */
```

```
/*                                                              */
/**********************************************************************/


thresh2 ()
{
int    move;
int    lower;
int    upper;
int    answer;
     do
     {
        printf ("Enter lower bound  <return> \n");
        scanf("%d",&lower);
        printf ("Enter upper bound  <return> \n");
        scanf("%d",&upper);
     } while ( ( upper  <  lower ) || ( upper  >  255 ) ||  ( lower  <  0 ) );

     camera (0,2,0,0);
     thresh1 (&lower,&upper);
     camera (0,2,0,0);

     do
     {
        printf ("Adjust thresholds ( y/n )  ? \n");
        answer  =  getch();
     }  while ('(y)ans&& (answer      =  'n') );

     if ( answer  = =  'n' )
        return;


      printf (" d to increase lower bound \n");
      printf (" a to decrease lower bound \n");
      printf (" l to increase upper bound \n");
      printf (" j to decrease upper bound \n");
      printf (" * to quit  \n\n");

      while (=(move  =  getch())
        {
           switch (move)
           {
           case 'j' :  (upper)--;
                       break;
           case 'l' :  (upper)++;
                       break;
           case 'a' :  (lower)--;
                       break;
           case 'd' :  (lower)++;
                       break;
           }
           if ( upper  >  255 )
              upper  =  255;
           if ( lower  <  0   )
              lower  =  0;
           if ( upper  <  lower )
           {
```

```
            lower = upper;
            upper = lower;
          }
          printf ("                          \r");
          printf ("lower= %d  upper= %d\r",lower,upper);

          camera (0,2,0,0);
          thresh1 (&lower,&upper);
          camera (0,2,0,0);
       }

} /* end of thresh2 */



/*********************************************************************/
/*                                                                   */
/* name:     thresh1( lower, upper)                                  */
/* sends:    lower  - lower threshold bound                          */
/*           upper  - upper threshold bound                          */
/* returns:  none                                                    */
/* synopsis: converts all pixel with grey level between lower and    */
/*           upper to white, all others to black by altering input   */
/*           lut n.                                                  */
/*           upper  > lower    0 < = n < = 6                         */
/*                                                                   */
/*********************************************************************/


thresh1 ( lower, upper )
int  *lower;
int  *upper;
{
  int i;

      for ( i=0; i<  *lower; i+ + )
         ilut ( 2, i, 0 );
      for  ( i  =  *lower ; i  < =  *upper; i+ +)
         ilut ( 2, i, 255);
      for  ( i  =  *upper+ 1; i< =255; i+ +  )
         ilut ( 2, i, 0);


} /* end of thresh1 */



/*********************************************************************/
/*                                                                   */
/* name:     cursor(x,y,buffer )                                     */
/* sends:    x      - initial cursor column number |  MUST BE        */
/*           y      - initial cursor row number    |  POINTERS       */
/*           buffer - displayed buffer number - 0 or 1               */
/*           two_pass - true if routine to be called twice           */
```

```
/*                    e.g for windowing                         */
/*          pass  - number of times cursor invoked              */
/* returns:  x     - final  cursor column number                */
/*          y     - final  cursor row    number                 */
/* synopsis:  displays white cursor over displayed buffer       */
/*          cursor may be moved using w a s z keys              */
/*                                                              */
/*                0 <=  x, y <=  511                            */
/*                                                              */
/*****************************************************************************/


cursor ( x, y, buffer, two_pass, pass )
int *x;
int *y;
int buffer;
int two_pass;
int pass;

{
  static int store[40];
        int wspace[40];
        int i,j,move,oldx,oldy;

        printf  (" move cursor using:- \n\n");
        printf  ("        w        \n");
        printf  ("    a     s      \n");
        printf  ("        z        \n");
        printf  (" use SHIFT to speed up movement\n");
        printf  ("\n type * to  quit ");
        printf  ("\n\n");

          /* read old pixel values */

          oldx = *x;
          oldy = *y;

          j = 0;
          for ( i = -9; i <= 9; i++ )
          {
            wspace[j] = read_pix( *x + i, *y, buffer );
            j++ ;
          }
          j = 20;
          for ( i = -9; i <= 9; i++ )
           {
            wspace[j] = read_pix( *x, *y + i, buffer );
            j++;
           }


           /* over write cursor */
           for ( i = -9; i <= 9; i++ )
              wrt_pix ( *x, *y + i, 255, buffer );
           for ( i = -9; i <= 9; i++ )
              wrt_pix ( *x + i, *y ,255, buffer );
```

```
while ((move = getch())
  {

     if ((move = = 'a')|| (move = = 's')||(move = = 'w')||(move = = 'z')||
        (move = = 'A')||(move = = 'S')||(move = = 'W')||(move = = 'Z') )
     {
        /* resore previous pixels */

        j = 0;
        for ( i = -9; i < = 9; i + + )
        {
           wrt_pix( *x + i, *y, wspace[j],  buffer );
           j + + ;
        }
        j = 20;
        for ( i = -9; i < = 9; i + + )
        {
           wrt_pix( *x, *y + i, wspace[j],  buffer );
           j + + ;
        }
       switch ( move )
        {

         case 'a' :  (*x)--;
                     break;
         case 's' :  (*x) + + ;
                     break;
         case 'w' :  (*y)--;
                     break;
         case 'z' :  (*y) + + ;
                     break;
         case 'A' :  (*x)- = 10;
                     break;
         case 'S' :  (*x) + = 10;
                     break;
         case 'W' :  (*y)- = 10;
                     break;
         case 'Z' :  (*y) + = 10;
                     break;

        }
        if ( *x  <  0 )
           *x = 511;
        if ( *x  >  511 )
           *x = 0;
        if (*y  >  511 )
           *y = 0;
        if (*y  <  0 )
           *y = 511;


       /* read old pixel values */
```

```
        j=0;
        for ( i=-9; i< =9; i+ + )
        {
          wspace[j] = read_pix( *x+i, *y, buffer );
          j+ + ;
        }
        j=20;
        for ( i=-9; i< =9; i+ + )
        {
          wspace[j] = read_pix( *x, *y+i, buffer );
          j+ +;
        }


        /* over write cursor */

        for ( i=-9; i< =9; i+ + )
           wrt_pix ( *x+i, *y, 255, buffer );

        for ( i=-9; i< =9; i+ + )
          wrt_pix ( *x, *y+i, 255, buffer );


        printf("                \r");
        printf("x=  %d  y=%d\r",*x,*y);

      } /* endif */
  } /* end of while */


  if ( two_pass = = FALSE )
  { /* restore previous pixels */
     j=0;
     for ( i=-9; i< =9; i+ + )
     {
        wrt_pix( *x+i, *y, wspace[j],  buffer );
        j+ + ;
     }
     j=20;
     for ( i=-9; i< =9; i+ + )
     {
       wrt_pix( *x, *y+i, wspace[j],  buffer );
       j+ +;
     }
  }
  else
  {
     if ( pass = = 1 )
     { /* write old pixel values to store */
        for ( i = 0; i < = 40; i+ + )
            store[i] = wspace[i];
     }
     else
     {  /* reinstate old values */
        j=0;
        for ( i=-9; i< =9; i+ + )
```

```
            {
               wrt_pix( *x+i, *y, wspace[j], buffer );
               j++ ;
            }
            j=20;
            for ( i=-9; i<=9; i++ )
            {
               wrt_pix( *x, *y+i, wspace[j], buffer );
               j++;
            }
            j = 0;
            for ( i=-9; i<=9; i++ )
            {
               wrt_pix( oldx+i, oldy, store[j], buffer );
               j++ ;
            }
            j=20;
            for ( i=-9; i<=9; i++ )
            {
               wrt_pix( oldx, oldy+i, store[j], buffer );
               j++;
            }
         }
      }

}  /*end of cursor */

histogram(buffer,xminm,yminm,xmaxm,ymaxm)
int buffer;
int xmaxm,ymaxm,yminm,xminm;
{
   int four=4;
   long  data[256];
   int i,x,y;
   long maxm;
   int  mode;
   float x1, x2, y1, y2;
   float  xmax, ymax, xmin, ymin;
   int colour;
   static char text[13][10] ={
                              {"0"},  {"20"}, {"40"}, {"60"}, {"80"},
                              {"100"}, {"120"}, {"140"}, {"160"}, {"180"},
                              {"200"}, {"220"}, {"240"}
                             };
    static char anykey[30] = {"PRESS ANY KEY TO CONTINUE"};


    for ( x=0; x<=255; x++ )
      data[x] = 0;

    printf ("\n\nCollecting data, please wait ...\n\n");

    mov_buf (buffer);
    for ( x=xminm; x<=xmaxm; x++)
     for ( y=yminm; y<=ymaxm; y++)
        data[ readpixl(x,y) ]++;
```

```
maxm = 0;
for ( x= 1; x< =254; x+ + )
 if ( data[x] > maxm )
 {
    maxm = data[x];
    mode = x;
 };



setdev("haloibmc.dev");
initgraphics(&four);

x1 = 0.0 ; x2 = 276.0; y1= 0.0; y2 = 120.0;
setworld ( &x1, &y1, &x2, &y2 );
clr();
xmax = 276.0; xmin = 20.0; /* histogram bounds */
ymax = 110.0; ymin = 25.0;



colour=0;
setcolor(&colour);
bar (&x1,&y1,&x2,&y2);
colour=6;
setcolor(&colour);
x=1;
setlnstyle(&x);
setlnwidth(&x);

/* x axis */

movabs (&xmin,&ymin);
lnabs (&xmax,&ymin);

/* y axis */

movabs (&xmin,&ymin);
lnabs (&xmin,&ymax);


/* x tick marks */

y1=ymin-3.0;
for ( x1 = xmin; x1 < = xmax; x1+ =2 )
{
   movabs (&x1,&ymin);
   lnabs (&x1,&y1);
}
y1=ymin-7.0;
for ( x1 = xmin; x1 < = xmax; x1+ =10 )
{
   movabs (&x1,&ymin);
   lnabs (&x1,&y1);
}
```

```
/* y tick marks */

x1 = xmin-3.0;
for ( y1 = ymin; y1 <= ymax; y1+ = 1.7 )
{
   movabs (&xmin,&y1);
   lnabs  (&x1,&y1);
}
x1 = xmin-7.0;
for ( y1 = ymin; y1 <= ymax; y1+ =8.5 )
{
   movabs (&xmin,&y1);
   lnabs  (&x1,&y1);
}
colour= 1;
setcolor(&colour);
for ( x=1; x<= 254; x++ )
{
  x1 = (float)x + xmin;
  movabs (&x1,&ymin);
  y1 = ymin +  (ymax-ymin)*data[x]/maxm;
  lnabs (&x1,&y1);
}

/* add the text */

x = 0;
y = 6;
ftcolor ( &y, &x );

x = 2;
y = 16;
ftsize(&x,&y);
ftinit();

y = 20;
for ( i = 0; i <= 12; i++ )
{
  x = 6 + (70*i)/12;
  ftlocate ( &y, &x );
  ftext ( text[i] );
}




x = 4;
y = 0;
ftcolor ( &x,&y);
x = 2;
y = 16;
ftsize(&x,&y);
ftinit();
```

```
    y = 22;
    x = 15;
    ftlocate (&y, &x);
    ftext ( anykey );
    printf ("mode at grey level %d:- %d pixels", mode, maxm);
    x = getch();
    closegraphics();



}


cross_section(buffer,hsection,coord)
int buffer;
int hsection;
int coord;
{

  int four = 4;
  int  data[512];
  int i,x,y;
  float x1, x2, y1, y2;
  float  xmax, ymax, xmin, ymin;
  int colour;
  static char xtext[13][10] = {
                          {"0"},  {"50"}, {"100"}, {"150"}, {"200"},
                          {"250"}, {"300"}, {"350"}, {"400"}, {"450"},
                          {"500"}, {"550"}
                         };
  static char ytext[14][10] = {
                          {"0"},  {"20"}, {"40"}, {"60"}, {"80"},
                          {"100"}, {"120"}, {"140"}, {"160"}, {"180"},
                          {"200"}, {"220"}, {"240"}
                         };



    /* get the data */

    if ( hsection = TRUE )
      for ( x = 0; x < 512; x++ )
        data[x] = read_pix (x,coord,buffer);
    else
      for ( y = 0; y < 512; y++ )
       data[y] = read_pix (coord,y,buffer);


    /* get the graphics going */


    setdev("haloibme.dev");
    initgraphics(&four);


    x1 = 0.0 ; x2 = 600.0; y1 = 0.0; y2 = 430.0;
    setworld ( &x1, &y1, &x2, &y2 );
```

```
clr();
xmax = 562.0; xmin = 40.0;  /* histogram bounds */
ymax = 315.0; ymin = 60.0;




colour = 0;
setcolor(&colour);
bar (&x1,&y1,&x2,&y2);
colour = 6;
setcolor(&colour);
x = 1;
setlnstyle(&x);
setlnwidth(&x);

/* x axis */

movabs (&xmin,&ymin);
lnabs (&xmax,&ymin);

/*  y axis */

movabs (&xmin,&ymin);
lnabs (&xmin,&ymax);


/* x tick marks */

y1 = ymin-5.0;
for ( x1  =  xmin; x1  < =  xmax; x1 + = 10.0 )
{
   movabs (&x1,&ymin);
   lnabs  (&x1,&y1);
}
y1 = ymin-10.0;
for ( x1  =  xmin; x1  < =  xmax; x1 + = 50.010 )
{
   movabs (&x1,&ymin);
   lnabs  (&x1,&y1);
 }


/* y tick marks */

x1 = xmin-5.0;
for ( y1  =  ymin; y1  < =  ymax; y1 + = 2.0 )
{
   movabs (&xmin,&y1);
   lnabs  (&x1,&y1);
}
x1 = xmin-10.0;
for ( y1  =  ymin; y1  < =  ymax; y1 + = 10.0 )
{
   movabs (&xmin,&y1);
   lnabs  (&x1,&y1);
}
```

```
    colour = 1;
    setcolor(&colour);

    /*data*/

    for ( x=0;  x< = 512;  x+ + )
    {
      x1 = (float)x + xmin;
      movabs (&x1,&ymin);
      y1 = ymin +  (ymax-ymin)*data[x]/255;
      lnabs (&x1,&y1);
    }

    /* add the text */

    x = 0;
    y = 6;
    ftcolor ( &y, &x );

    x = 1;
    y = 8;
    ftsize(&x,&y);
    ftinit();

    y = 40;
    for ( i = 0; i < = 12; i+ +)
    {
      x = 6 + 6 * i ;
      ftlocate ( &y, &x );
      ftext ( xtext[i] );
    }

    x=1;
    for ( i = 0; i < = 12; i+ +)
    {
      y = 38 - i * 2;
      ftlocate ( &y, &x );
      ftext ( ytext[i] );
    }

    x = 4;
    y = 0;
    ftcolor ( &x,&y);
    x = 2;
    y = 16;
    ftsize(&x,&y);
    ftinit();
    printf ("Press any key to continue");
    x=getch();
    closegraphics();
}


float measurement(centre,scale)
int  centre;
float scale;
```

```
{
float heightmm;
float shorenum;

int  x,y;
int  dummy;
int  nwhite;
int  ymin,ymax;
int  success;
int  yt,i;
int  npoints;
int  maxima[128];
int  diff[128];
int  lower,upper;
int  least_diff;
int  turning_pt;
int  maximum;
int  zero;
int  zero2;

   printf(" --- press any key to start measurement ---\r");
   x=getch();

   printf ("                                          \r");
   printf (" processing raw data ..... \r");
   if ( centre=%0 )
      centre++;
   ltime (centre,128);
   capture(0,0,0,0);
   res_buf(0);
   printf ("                                          \r");
   printf (" sorting odd and even fields \r");


   /* sort groups of four rows into groups of two - */
   /* odd and even fields */

   for ( y=0; y<=511; y+=4 )
     for ( x=0; x<=511; x++ )
       {
          dummy = readpixl ( x, y+2 );
          wrtpixl ( x, y+2, readpixl ( x, y+1 ) );
          wrtpixl ( x, y+1, dummy  );
       }


   /* OR corresponding pairs of rows */

   res_buf (0);
   printf ("                                          \r");
   printf (" OR ing corresponding rows \r");

   for ( y = 0; y <= 508; y+=2 )
     for ( x = 0; x <= 511; x++ )
     {
        dummy = readpixl ( x, y )  | readpixl (x,y+1) ;
        wrtpixl ( x, y , dummy );
```

```
      wrtpixl ( x, y+1 , dummy );
   }

   res_buf (0);
   printf ("                              \r");
   printf (" calculating zero level \r" );

   /* find first column with number of white pixels > 25 % */
   /* this gives the position of the base of the stand      */

   success = FALSE;
   for ( x =511; ( x >= 0 ) && ( success= =FALSE ) ; x-- )
   {
     npoints = 0;
     for ( y = 0; y <= 511; y++ )
        if ( readpixl(x,y) == WHITE )
          npoints++;
     if ( npoints > 128 )
         success = TRUE;
   }

    dummy= x;
    zero2=dummy;

   /* find first column with number of white pixels < 25 % */
   /* this gives the position of the top of the ball bearing */
   /* at rest */

   success = FALSE;
   for ( x =dummy; ( x >= 0 ) && ( success= =FALSE ) ; x-- )
   {
     npoints = 0;
     for ( y = 0; y <= 511; y++ )
        if ( readpixl(x,y) == WHITE )
          npoints++;
     if ( npoints < 128 )
         success = TRUE;
   }

    zero = x -1;
    zero2= ( zero2 + zero ) / 2;



  printf ("                              \r");
  printf ("zero level at x = %d \n ",zero);
  printf (" calculating window \r" );


  /* find where intial descent occurred */
  /* first row with >= 10% white pixels */


 success= FALSE;
 for ( y=0; ( y<=508 ) && ( success= = FALSE ); y+ =2 )
 {
   nwhite = 0;
```

```
    for ( x = 0; x <= 511; x++ )
    {
       if ( readpixl(x,y) == WHITE )
          nwhite++;
       if ( nwhite > 50 )
          success=TRUE;
    }
}
  ymin = y;
  if ( success == TRUE )
     for ( x = 0; x<=511; x++ )
        wrt_pix ( x, ymin, 255, 0 );
 else
 {
    printf ("ERROR - unable to locate lower window bound\n\n");
    printf (" --- press any key --- \n");
    x = getch();
    return(-1.0);
 }
/* find end point */

success=FALSE;
for (  y = ymin;  ( y <= 511 ) && ( success==FALSE ) ; y+=2 )
{
   nwhite = 0;
   for ( dummy = 0; dummy <= 50; dummy++ )
   {
      if ( readpixl( zero2, y - dummy ) == WHITE )
         nwhite++;
      if  ( nwhite > 40 )
         success=TRUE;
   }
}

 if ( success==FALSE )
 {
    printf ("upper window bound not found;  using worst case estimate instead\n\n");
    ymax = ( y > 256 ) ? 511 : y + 256 ;
 }
 else
    ymax=y;

 for ( x = 0 ; x <= 511 ; x++ )
    wrt_pix  ( x , ymax, 255, 0 );


printf ("                                   \r");
printf (" removing noise - vertical lines \r" );

for ( x=0; x<=511; x++ )
{
  nwhite =0;
  for ( y = ymin; y<= ymax; y+=2 )
     if ( readpixl(x,y) == WHITE )
        nwhite++;
  if ( nwhite > (ymax-ymin)/8 )
     for ( yt = ymin; yt <= ymax; yt++ )
```

```
        wrtpixl ( x, yt, BLACK );
    }

    res_buf(0);
    printf ("                                    \r");
    printf (" calculating rebound height   \r");


    /* get white pixel with max height for each row */

    i=0;
    for ( y = ymin; y <= ymax; y+=2 )
    {
        success=FALSE;
        for  ( x= 0; ( x<=511 ) && (success==FALSE) && ( i< 128) ; x++ )
        {
          npoints=0;
          for ( dummy=0;  dummy  <=  10; dummy++ )
             if   (  readpixl(x+dummy,y)  ==  WHITE )
                 ++npoints;
          if ( npoints >= 5 )
              success=TRUE;
        }
        if ( success == TRUE )
        {
          maxima[i] = x;
          i++;
        }
    }
   npoints = i-1;

   /* calculate differences  */

   for ( i=0; i<= npoints-1; i++ )
      diff[i] = maxima[i] - maxima [i+1];

   for (i = 0; i <= npoints-1 ; i++ )

  /* find first three points with +ve differences */

  success = FALSE;
  for ( i = 0; (i <= npoints-2) && (success==FALSE); i++ )
     if ( (diff[i] >= 0) && (diff [i+1] >= 0) && ( diff[i+2] >= 0 ) )
     {
        success = TRUE;
        lower = i;
     }
  if ( success == FALSE )
  {
       printf ("ERROR - unable to find first positive gradient\n\n");
       printf (" --- press any key --- \n");
       x = getch();
       return(-1.0);
  }
  /* find first three points with -ve differences */
```

```
success = FALSE;
for ( i = lower; (i < = npoints-2) && (success= =FALSE); i+ + )
  if (  (diff[i]  < =  0) && (diff[i+ 1]  < =   0) && ( diff[i+2]  < =  0 ) )
  {
     success = TRUE;
     upper = i;
  }
 /* convert all differnces to  + ve */

 if ( success = =  FALSE )
 {
    printf ("ERROR - unable to find first negative gradient\n\n");
    printf (" --- press any key --- \n");
    x = getch();
    return(-1.0);
 }
 for (i=lower; i< =upper; i+ + )
   diff[i] =   ( diff[i]  >  0 ) ? diff[i] : -diff[i];



/* find least differnce between lower and upper - turning point */


least_diff= 1000;
for ( i = lower; i  < =  upper; i+ +  )
  if ( diff[i]  <  least_diff  )
  {
    least_diff=diff[i];
    turning_pt  =  i;
   }

/* find the value corresponding to turning_pt */

maximum  =  ( maxima[turning_pt]  <  maxima[turning_pt + 1] ) ?  maxima[turning_pt] : maxima[t
urning_pt+ 1];

for ( y  =  0; y  < =  511; y+ +  )
  wrt_pix ( maximum, y, 255, 0 );

 heightmm  =  ( (float)zero - (float)maximum )  *  scale;
 shorenum  =  heightmm  *  ( 140.0 / 250.0 );

 printf ("                                         \r");
 printf (" rebound height  =  %d  pixels \n",zero-maximum );
 printf ("                 =  %6.1f  mm      \n\n",heightmm);
 printf (" Shore number    =  %6.1f \n\n ",shorenum );
 printf (" any key to continue \n  ");

 return(shorenum);

}
```

APPENDIX VII

C PROGRAM CODE FOR THE VICKERS SYSTEM

function: average(data)

non-pointer arguments: none

pointer arguments: int data

returns: none

called functions: none

description:

This function produces a running-averaged version of the input array. Each element of the array is replaced with the average of it's own contents and those of the following two elements.

function: binary()

arguments: none

returns:

int error_status — FALSE: no errors found, image successfully thresholded; NON_FATAL: image thresholded but poor contrast present in original image; FATAL: no threshold found.

called functions: average(), get_thresh()

description:

The function first captures a single frame and transfers it to base memory. Each pixel of the frame is read to produce a frequency table of grey levels within the image, held within a 256 element array. Small fluctuations in the frequency table are removed by calling the average() function and the image is thresholded by calling the get_thresh() function. The latter function also returns the error status.

function: blob (x_bar, y_bar)

non-pointer arguments: none.

pointer arguments:

	int x_bar	x co-ordinate of the centroid
	int y_bar	y co-ordinate of the centroid

pointer arguments: none

returns:

	error_status	TRUE if excessive number of blobs found; FALSE otherwise.

called functions: none

description:

This function implements the region labelling and label collection operations described in section 4.4.5. Instead of assigning each object pixel in the binary image a label (which is inefficient), labels are written to which ever memory partition is not being used by the current run-length code. The label list is arranged in such a way that each label has the same index in the list as the corresponding run in the run-length code. This is achieved as follows; for each row number (RN) codeword in the run-length code, a "blank" is written to the label list (the actual value being 2 000). For each TW1 codeword in the

run-length code the row number of the following run, plus 1 000, is written to the label list. For each TW2 codeword the value of the label corresponding to the run is written to the label list. In this manner all the runs corresponding to a given label can easily be located.

At the same time as region labelling is taking place the areas of labelled blobs are calculated using [4.74]. Each blob area area is stored in an array indexed by label. If the number of labels at any time exceeds 256 then the function is exited with error_status returned TRUE.

The label collection operation is as described in section 4.4.5, with one exception; instead of using separate stacks to store equivalent labels, an array is used. Equivalent labels are stored in consecutive elements in the array with different groups of labels separated by an element holding an illegal value (actually 300).

Once the label collection operation has finished, the function sums the areas of connected blobs to give region areas. The largest region and largest blob are then found. If the largest blob has the greater area then the centre of area equations [4.72],[4.73] are applied to all the runs making up the blob so that the centroid co-ordinates may be determined.

The same method is used if the largest region has the greater area however, here the centre of area equations are applied to all runs of all constituent blobs. In both cases the centroid co-ordinates are returned by the function.

function: centre_of_area (x,y)

non-pointer arguments: none

pointer arguments:

	int x	x co-ordinate of the centroid
	int y	y co-ordinate of the centroid

returns: none

called functions: none

description:

Reads each pixel of the thresholded image and calculates the centroid co-ordinates by implementing eqns [4.10],[4.11],[4.12].

function: code (no_of_trans)

non-pointer arguments: none

pointer arguments:

	int no_of trans	number of codewords in the resulting run length code.

returns:

	int error_status	TRUE if excessive number of runs found; FALSE otherwise.

called functions: none

description:

This function forms the run-length code for a down-loaded binary image. The run-length code is stored in the same part of extended memory as for down-loaded

images to save space in base memory. The reserved space for down-loaded images is 256 kbyte. This is divided into to equal partitions so that two run-length codes can simultaneously be stored. This allows for "ping-pong" processing i.e. taking data from one part of memory, operating on it and then transferring it to a different part of memory. The data flow is then reversed for the next operation. Each codeword is stored within a single int (two bytes) of storage. Codewords are represented as follows:

RN = y co-ordinate of the run

TW1 = left co-ordinate of the run + 1000

TW2 = right co-ordinate of the run + 2000

This allows the shrink(), expand() and delete() functions to distinguish between different types of codeword.

The function sets the first and last columns of the binary image to white so that each row begins with a white to black transition and ends with a black to white transition. The image is scanned row by row and as each transition is found the corresponding codeword is written to memory. The run-length code starts on row 0 however, the image scanning starts on row 2. By this arrangement the image data can never be over-written by the run-length code prior to it being read.

If more than 20 000 codewords are recorded then error_status is returned true. This value was set not by storage limitations but was determined empirically. Images requiring more than this number of codewords were found to be of such poor quality that further processing did not yield any significant improvements.

function: cursor

description:

This function is identical to the function used in the Shore hardness program, described in Chapter 3.

function: delete (no_of_trans,n)

pointer arguments:

	int n	maximum length of runs to be erased
pointer arguments:		
	int no_of_trans	number of codewords in the resulting run-length code
returns:	none	
called functions:	none	

description:

The function reads each pair of transition codewords (TW1,TW2) from the first memory location and writes them to the second memory location if they are of length greater than n.

function: draw_line (x1,y1,x2,y2)

non-pointer arguments:

	int x1	x co-ordinate of first point
	int y1	y co-ordinate of first point
	int x2	x co-ordinate of second point
	int y2	y co-ordinate of second point
pointer arguments:	none	

returns: none

called functions: none

description:

This function draws a straight line from (x1,y1) to (x2,y2) by writing to the frame store image buffer. Those pixels forming the line are set to grey level WHITE.

function: expand (no_of_trans)

pointer arguments: none

pointer arguments:

int no_of_trans number of codewords in the run length code

returns: none

called functions: none

description:

This function implements the expansion operation described in section 4.4.3 The x-direction expansion is first achieved by reading, from the first memory partition, a pair of transition codewords (TW1,TW2), altering the x co-ordinates according to [4.19],[4.20] and then writing the resulting codeword pair to the second memory partition. If the resulting run "touches" another run on the same line the two runs are condensed into one.

The expand operation in the -y direction (a +y direction expansion was not found to be necessary) is then implemented as follows. A pair of codewords (TW1,TW2 are taken from the run-length code corresponding to a run on the (i+1)th row. The codewords (TW1,TW2) corresponding to all the runs on the ith row which overlap the run on the (i+1)th row are then found. If a simple configuration is found (configurations 1 to 5 in section 4.4.3) then the codewords corresponding to the run on the ith row are modified according to [4.28]-[4.33]. and written to the second part of the memory partition.

If a block is found then all connected runs on both the ith and (i+1)th row are located. A single pair of codewords are then written to the second memory partition. This gives a single run which replaces all the connected runs on the ith row. The process is repeated for the next run on the (i+1)th row and then on subsequent rows.

function: get_thresh (data,error)

non-pointer arguments:	none	
pointer arguments:		
	int data	array holding the grey level frequency table
	int error	FALSE: no errors found,image successfully thresholded; NON_FATAL: image thresholded but poor contrast present in original image; FATAL: no threshold found.
returns:	none	
called functions:	thresh()	

description:

The function implements the automatic thresholding algorithm given in section 4.3.2. The mode value is found from the grey level frequency table and the search is first carried out on the lower side of the mode to locate the other mode and minimum. If a threshold is found, it is judged to be acceptable if:

$$(n(g_1)/n(g_3) > 2) \cap (n(g_2)/n(g_3) > 2) \cap$$
$$(n(g_1) > 100) \cap (n(g_2) > 100) \qquad [A7.1]$$

This condition is stricter that in section 4.4.6. The conditions $n(g_1) > 100$ and $n(g_2) > 100$ are necessary to prevent the algorithm taking small modes caused by noise as belonging to the object. If the above condition is met then the input LUT is set up by calling the thresh function, error is set to FALSE and the routine is exited. If a threshold is found but:-

$$(n(g_1)/n(g_3) \leq 2) \cap (n(g_2)/n(g_3) \leq 2) \cap$$
$$(n(g_1) > 100) \cap (n(g_2) > 100) \qquad [A7.2]$$

then the threshold is noted so that it may be used later with error set to NON_FATAL to warn of poor contrast.

A search is then made on the upper side of the mode. If a threshold is found and the condition given in [A7.1] holds then the thresh() function is called with the threshold as an argument and error is set to FALSE. If a threshold is found and [A7.2] holds but [A7.1] does not then the thresh() function is called with the threshold as an argument and error is set to NON_FATAL. Finally if no threshold is found on the upper side of the mode then the function checks to see if a threshold was found on the lower side of the mode. If it was, then the previous threshold is used as an argument for the thresh()

function and error is set to NON_FATAL. If no threshold at all is found then error is returned to TRUE and the thresh() function is not called.

function: new_centre (nvalues,x,y)

non-pointer arguments:

	int nvalues	number of boundary points for the indentation

pointer arguments:

	int x	x co-ordinate of the centroid
	int y	y co-ordinate of the centroid
returns:	none	
called functions:	none	

description:

The function calculates the centroid of the indentation based on the boundary pixels of the indentation. The centre of area equations [4.10],[4.11],[4.12] are applied to the boundary co-ordinate data (held in two arrays) with the area taken as the number of boundary points (nvalues).

perimeter(x,y,nvalues)

non-pointer arguments

	int x	x co-ordinate of the centroid
	int y	y co-ordinate of the centroid

pointer arguments:

	int nvalues	number of boundary pixels for the indentation
returns:		
	error_status	TRUE if indentation boundary meets the image border; FALSE otherwise.
called functions:	none	

description:

The function scans the image radially, starting from the centre of area, at 1 degree increments. The nominal radius increment is 1 pixel, however the polar (radius, angle) form is converted to rectangular co-ordinates in order to address each pixel. A pixel is taken as a boundary point if the following 5 contiguous pixels in a given radial direction are found to be white (i.e. have the value GREY). If the radius at any angle is found to extend to the image boundary then the error_status variable is set TRUE. The co-ordinates of the boundary points are stored in two arrays (one for x and one for y) indexed by angle in degrees.

functions: screen1(), screen2() ... etc, perimeter_error(), thresh_error(), etc

arguments:	none
returns:	none
called functions:	none

description:

These functions provide prompts and program status information for the user by writing to the VDU.

function shrink (no_of_trans)

non-pointer arguments: none

pointer arguments:

	int no_of_trans	number of codewords in the resulting run-length code

returns: none

called functions: none

description:

This function implements the shrink operation described in section 4.4.4. The x-direction shrink is first achieved by reading from the first memory partition each pair of transition codewords (TW1,TW2), altering the x co-ordinates according to [4.45],[4.46] and then writing the resulting codeword pair to the second memory partition. If the original run was of length two or less then the codewords are not written to the second memory partition.

The shrink operation in the -y direction is then implemented as follows. A pair of codewords are taken from the run-length code corresponding to a run on the ith row. The codewords of all runs on the ith row which overlap the run on the (i+1)th row are then located. If a simple configuration is found (configurations 1 to 5) then the codewords corresponding to runs on the (i+1)th row are modified according to [4.47]-[4.50] and written to the second memory partition.

If a block is found then all runs on both the ith and (i+1)th lines which are connected are located. A series of transition codewords are then created according to [4.53]-[4.63] and written to the second memory partition. The process is then repeated for each new run on the (i+1)th line.

function: set_up()

arguments:	none
returns:	none
called functions:	none

description:

This function writes to the frame store input look-up table to give a direct grey scale mapping for captured frames.

function: sig (nvalues, x, y, ang1, ang2, ang3, ang4)

non-pointer arguments:		
	int nvalues	number of boundary points for the indentation
pointer arguments:		
	int x	x co-ordinate of the centroid
	int y	y co-ordinate of the centroid
	int ang1..ang4	angles of the indentation corners from the horizontal
returns:	none	

called functions: none

description:

The function locates the corners of the indentation from the data held in the boundary co-ordinate arrays. The signature is first calculated by computing the distance of each boundary pixel from the centroid. The distance values are stored in two arrays indexed by angle in degrees. The distance between two co-ordinates $(x_1 y_1)$ and $(x_2 y_2)$ is given by:-

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \qquad [A7.3]$$

however, a modified form of this equation is used in the program so that the distance is normalised with respect to the aspect ratio. The modified equation is:-

$$r[i] = \sqrt{(\bar{x} - x[i])^2 + \left(\frac{(\bar{y} - y[i])}{aspect} \right)^2} \qquad [A7.4]$$

where $r[i]$ is the distance of a boundary pixel at $(x[i], y[i])$ from the centroid, $(\bar{x}, \bar{y})$ are the centroid co-ordinates and *aspect* is the aspect ratio (approximately 4/3). To reduce the effects of small fluctuations in the signature (which could cause problems when determining the peaks in the signature) each distance value is replaced by the average taken over five elements, i.e. a new set of signature values,

$r_{av}[i]$, are created such that:-

$$r_{av}[i] = \frac{1}{5}(r[i-2] + r[i-1] + r[i] + r[i+1] + r[i+2]) \qquad [A7.5]$$

To locate the first corner of the indentation the maximum distance value in the signature is found. The index of the array element holding this value is then the angle of the first corner to the horizontal and is recorded as angl. The location of the second corner is

then found by scanning the signature array over the interval [$ang1 + 45$, $ang1 + 135$] and locating the maximum distance. The index of this value gives the angle of the second corner. The process is repeated until all four corners have been found.

function: sig_check (nvalues, x, y)

non-pointer arguments:

	int nvalues	number of boundary points for indentation
	int x	x co-ordinate of centroid
	int y	y co-ordinate of centroid
pointer arguments:	none	
returns:		
	error_status	TRUE: if the centroid is outside the indentation or too close to the boundary; FALSE: otherwise.
called functions:	none	

description:

The function calculates the distance of each boundary point from the centroid. If more the 180 points have a distance of 10 pixels or less from the centroid, it is assumed that the centre of area is outside the indentation (since the background is white, the perimeter function will, ideally, record zero distance for each boundary point). In this case the error_status is set to TRUE and the routine is exited. The ratio of maximum boundary distance to radius for each boundary point is recorded. If this ratio is greater than 5 for more than 10 boundary pixels (these values were determined empirically) it is

assumed that the centroid is too close to the boundary. Here the error_status variable is set to TRUE.

function: smooth()

arguments: none

int error_status TRUE: if excessive number of transitions found; FALSE: otherwise.

called functions: code(), delete(), expand(), shrink()

description:

This function performs the binary smoothing operation discussed in section 4.4.1. The code() function is first called to produce the run-length code from data in a down-loaded binary image. The expand() function carries out the expansion of section 4.4.3 and is then called twice. The shrink() function, which implements the shrink operation of section 4.4.4 is then called twice. Finally any runs of length 10 or less are erased from the code by calling the delete() function. This has effect of removing small groups of black pixels which have been erroneously thresholded.

function: stretch()

arguments: none

returns: none

called functions: none

description:

This function "stretches" the grey scale of an image so that the entire grey scale range of the frame store (here 0 to 255) is used by the image. A frame is first captured and down-loaded. Every pixel is then read to ascertain the maximum grey level (g_{max}) and minimum grey level (g_{min}). Each grey level is then mapped to a new grey level according to:-

$$g \rightarrow 255\left(\frac{g - g_{\min}}{g_{\max} - g_{\min}}\right) \qquad [A7.6]$$

This is achieved by writing the transformed grey scale to the input LUT and then capturing another image.

function: thresh(t1,t2)

pointer arguments:

	int t1	lower threshold
	int t2	upper threshold
pointer arguments:	none	
returns:	none	
called functions:	none	

description:

This function writes to the frame store look-up table to give the required grey scale mapping for capturing binary images. Any pixels having grey levels between t1 and t2 are converted to black while those outside this range are converted to white.

function: verify (magnification,weight)

non-pointer arguments:

	int magnification	microscope magnification
	int weight	applied load in kgf.
pointer arguments:	none	
returns:	none	
called functions:	stretch(), cursor()	

description:

The function first prompts the user if she/he wants the grey scale of the image to be "stretched". If so then the stretch() function is called. The user is then prompted to place four cursor at the corners of the indentation as seen on the output monitor. Cursor movement is performed by the cursor() function which also returns the cursor co-ordinates once placed. The diagonal lengths are then calculated as in the vickers() function [A7.7] and the corresponding hardness calculated using eqn [4.14].

function: vickers (magnification, weight, angle1, angle2, angle3, angle4)

non-pointer arguments:

	int magnification	microscope magnification
	int weight	applied load in Kgf
	int angle1...angle4	angles of indentation corners
pointer arguments:	none	
returns:	none	
called functions:	draw_line()	

description:

The function first finds the co-ordinates of the indentation corners from the boundary co-ordinate arrays using the corners angles. The diagonal lengths in millimetres are then calculated using the following formula:-

$$d = \sqrt{\left(\frac{x1 - x2}{Sx}\right)^2 + \left(\frac{y1 - y2}{Sy}\right)^2} \qquad [A7.7]$$

where d is the diagonal length, (x1,y1) and (x2,y2) are the co-ordinates of the corners and Sx and Sy are the x and y scaling factors in pixels/mm. The Vickers hardness is then calculated using [4.14]. Before the function is exited the set_up() function is called to produce a real-time grey scale picture on the output monitor.

```
.li on
#include "\lc\stdio.h"
#include "\lc\dos.h"
#include "\lc\math.h"

extern void camera(); /* real time display(i/p buff,ilut,o/p buff,olut) */
extern void capture(); /* real time capture (i/p buff,ilut,o/p buff,olut) */
extern void ilut();  /* i/p look up table (table,index,value) */
extern void olut();  /* o/p look up table (table,index,value) */
extern void mov_buf(); /* move buffer to local buffer (buff) */
extern void res_buf(); /* restore to buffer(buff)*/
extern int read_pix(); /* read (x,y) pixel from buffer (x,y,buff) */
extern int readpixl(); /* read (x,y) pixel from local buffer (x,y) */
extern void wrt_pix(); /* write pixel to buffer (x,y,value,buffer) */
extern void wrtpixl(); /* write pixel to local buffer (x,y,value) */
extern void write2_word(); /* read/writes direct to buffer */
extern int  read1_word();
extern void write2_word(); /* (index,value) */
extern int  read2_word();  /* (value)       */
extern void header();
extern void verify();
extern void thresh_error();
extern int smooth();
extern int blob();
extern void vickers();


#define    WHITE    255
#define    BLACK    0
#define    GREY     150
#define    TRUE     1
#define    FALSE    0
#define    FATAL    1
#define    NON_FATAL 2
#define    NAVERAGE 3   /* number of histogram averages */
#define    ASPECT   1.362
#define    SCALE    454     /* pixel/mm in x direction */
#define    SCALEX   454.0
#define    SCALEY   673.0
#define    DISTANCE_RATIO 5
#define    MAX_POINTS 10

int _stack = 32000;
int xcoord[360];
int ycoord[360];
int  distance[360];
int  data[256];


main()
{
int option;
int magnification,weight;
int i,j;
int  x,y;
int  nvalues;
```

```
int   ang1,ang2,ang3,ang4;
int   error_status;

      /* default settings */

START: magnification = 10;
      weight = 20;

      ang1 = ang2 = ang3 = ang4 = 0;

      header(magnification,weight);
do{
 option = getch();
            printf ("\x07");
            switch (option)
            {
 case 'm':
                  printf ("%c[44m",27);
                  system("cls");
                  printf ("%c[41m",27);
                  printf ("%c[37m",27);
                  printf ("%c[12;18f",27);
                  printf("                                        ");
                  printf ("%c[13;18f",27);
  printf("   Enter new magnification factor  <ret>      ");
                  printf ("%c[14;18f",27);
                  printf("                                        ");
                  printf ("%c[13;59f",27);
  scanf("%d",&magnification);
                  header(magnification,weight);
            break;
            case 'v':
                  system("cls");
                  verify(magnification,weight);
                  header(magnification,weight);
                  break;
case 'w':
                  printf ("%c[44m",27);
                  system("cls");
                  printf ("%c[41m",27);
                  printf ("%c[37m",27);
                  printf ("%c[12;25f",27);
                  printf("                                ");
                  printf ("%c[13;25f",27);
 printf("   Enter new weight in Kgf  <ret>      ");
                  printf ("%c[14;25f",27);
                  printf("                                ");
                  printf ("%c[13;59f",27);
 scanf("%d",&weight);
                  header(magnification,weight);
 break;
case 'p':
                  screen1();
                  screen2();

                  error_status = binary();
                  if ( error_status = = FATAL )
```

```
        {
          thresh_error();
          goto START;
        }
        else
          if ( error_status = = NON_FATAL )
          {
              screen7();
              error_status = smooth();
              if ( error_status )
                goto START;
              screen8();
              error_status = blob(&x,&y);
              if ( error_status )
                goto START;
              capture(0,0,0,0);
              mov_buf(0);
          }
          else
          {
            screen3();
            centre_of_area(&x,&y);
          }
        screen4();
        error_status = perimeter (x,y,&nvalues);

        if ( error_status )
          {
            perimeter_error();
            goto START;
          }

        error_status = sig_check(nvalues,x,y);
        if ( error_status )
        {
          screen7();
          error_status = smooth();
          if ( error_status )
             goto START;
          screen8();
          error_status = blob(&x,&y);
          if ( error_status )
             goto START;

          capture(0,0,0,0);
          mov_buf(0);
          perimeter(x,y,&nvalues);

        }
        screen5();
        new_centre(nvalues,&x,&y);
        sig(nvalues,x,y,&ang1,&ang2,&ang3,&ang4);
        screen6();
        vickers(magnification,weight,ang1,ang2,ang3,ang4);
        header(magnification,weight);
        break;
case 'x' :
```

```
                printf("%c[0m",27);
                system("cls");
                break;
        default:
                printf ("%c[46m",27);
                printf ("\r");
                printf ("     ");
                printf ("%c[41m",27);
                printf ("%c[37m",27);
                printf ("                          --- invalid option ---
        ");

                for ( i=0; i< =30000; i+ +)
                  for ( j=0; j< =5; j+ +);

                printf ("%c[46m",27);
                printf ("\r");
                printf ("     ");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("                          please enter an option
        ");


                break;
  }
 } while (option              ='x');
}
 /*  end of main */



set_up()       /* sets up the input LUT to give direct grey scale mapping */
{
int i;

 for (i=0;i< =255;i+ +)
  ilut(0,i,i);
 camera(0,0,0,0);
}
/*  end of setup*/




binary()            /* calculates grey level histogram, prints it out */
               /* and finds the threshold bounds                 */


{
register x,y;
int data[256];    /* frequency values */
int  error;

      /* move a frame down to base memory */
```

```
        capture(0,0,0,0);
        mov_buf(0);

       /* clear the tally table */

        for ( x=0; x<256; x++ )
          data[x] =0;


       /* calculate the frequencies for the tally table */
       /* note only 1/4 of the pixels are read         */

        for ( x = 0; x < 512; x+ =2 )
           for ( y = 0; y < 512; y+ =2 )
              data[ readpixl(x,y) ]++;

       /* remove fluctuations from the tally table by averaging */
        average(data);



       /* get the threshold bounds and threshold image */
        get_thresh(data,&error);


        camera(0,0,0,0);

         /* clear the histogram from the display */

        capture(0,0,0,0);

        return(error);


}
/*  end of binary */




thresh(t1,t2)   /* alter the input LUT values */
              /* grey level betweem t1 and t2 map to black(0) */
              /* those outside to white */

int t1,t2;
{
 int i;

 for (i=0;i< =255;i++){
  if ((i> =t1) && (i< =t2)){
   ilut(0,i,BLACK);
  }
  else
   ilut(0,i,GREY);
```

```
 }
}
/* end of thresh */




average(data)
int data[];
{
int i,j;

     for (j=1; j<=NAVERAGE; j++ )
     for ( i=0; i<=253; i++ )
         data[i] = ( data[i] + data[i+1] + data[i+2] ) / 3;

}


get_thresh(data,error)              /* computes the threshold values */
int  data[];
int  *error;
                    /* return values for error:                  */
                    /*    FALSE      - no error, thresholds found    */
                    /*    NON_FATAL - error, both thresholds found */
                    /*    FATAL      - error, no thresholds found    */

{

long max_frequency1;
long min_frequency;
long max_frequency2;
int  i;
int  mode1,mode2;
int  old_thresh;
int  thresh1;
int interval;

          /* find  mode value */

          max_frequency1 = 0;
          old_thresh = 0;

          for ( i=0; i<250; i++ )
            if ( data[i] > max_frequency1 )
            {
                max_frequency1 = data[i];
                mode1 = i;
            }

         interval = 20;

          do
          {
```

```
/* find other mode - must be at least interval pixels */
/* distant from 1st mode                               */

max_frequency2 = 0;

for ( i = 0; i < = mode1-interval; i + + )
  if ( data[i] > = max_frequency2 )
  {
    max_frequency2 = data[i];
    mode2 = i;
  }

/* find "valley" between modes - this lowest frequency */

  min_frequency = 300000;

    for ( i = mode2; i < = mode1; i + + )
    {
     if ( data[i] < min_frequency )
     {
       min_frequency = data[i];
       thresh1 = i;
     }
    }

interval + = 10;

} while (          ( (min_frequency < max_frequency1) &&
          (min_frequency < max_frequency2)  ) && (mode1-interval > 0) );

 if (   ( min_frequency < max_frequency1 ) && ( min_frequency < max_frequenc
y2 )
     && ( max_frequency1 > 100 ) && ( max_frequency2 > 100 ) )
   old_thresh = thresh1;

if ( ( mode1-interval > 0 ) && ( max_frequency1/min_frequency > 2 ) &&
    ( max_frequency1 > 100 ) && ( max_frequency2 > 100 ) )
{
  *error = FALSE;
}
else
{
   interval = 20;

   do
   {
      /* find other mode - must be at least interval pixels */
      /* distant from 1st mode                               */

      max_frequency2 = 0;
```

```
        for ( i=mode1+interval; i<= 255;  i++)
          if ( data[i] >= max_frequency2 )
          {
            max_frequency2 = data[i];
            mode2=i;
          }


        /* find "valley" between modes - this lowest frequency */


        min_frequency=300000;

        for ( i=mode1; i<=mode2; i++ )
        {
          if ( data[i] < min_frequency )
          {
            min_frequency = data[i];
            thresh1 = i;
          }
        }

        interval += 10;

    } while (              ( (min_frequency<max_frequency1) &&
             (min_frequency<max_frequency2)  ) && (mode1+interval  < 250) );



    if ( ( max_frequency1/min_frequency > 2 ) && ( mode2 < 250 ) &&
        ( max_frequency1 < 100 )  && ( max_frequency2 < 100 ) )
    {
      *error = FALSE;
    }
    else
      if ( ( mode2 < 250 ) && ( max_frequency1 < 100 )  && ( max_frequency2 < 100
))
      {
        *error = NON_FATAL;
      }
      else
       if ( old_thresh > 0 )
       {
          thresh1=old_thresh;
          *error = NON_FATAL;
       }
       else
       {
        *error = FATAL;
        return;
       }

  }
  /* lower threshold = 0 */
```

```
        capture(0,0,0,0);
        thresh(0,thresh1);
        capture(1,0,0,0);

   }






draw_line(x1,y1,x2,y2)   /* draws a line on the output image from */
int x1,y1;               /* (x1,y1) to (x2,y2)                    */
int  x2,y2;
{
float grad;
float offset;
float y;
int   x;

        /* calculate gradient */

        grad = (float)( y1 - y2 ) / (float)( x1 - x2 );

        /* calculate intercept on y axis */

        offset =  (float)y1 - grad*(float)x1;

        if ( x1 < x2 )              /* check for smallest x coord */
          for  ( x= x1; x< =x2; x+ + )
          {
             y =  grad*(float)x + offset;
             wrt_pix(x,(int)y,WHITE,0);
          }
        else
          for  ( x= x1; x> =x2; x-- )
          {
             y =  grad*(float)x + offset;
             wrt_pix(x,(int)y,WHITE,0);
          }

}  /* end of draw lines */


perimeter(xbar,ybar,nvalues)  /* calculates the outside of the indetation    */
                              /* by a radial line scan centred on (xbar,ybar)*/
                              /* nvalues returns the number of boundary      */
                              /* pixels actually found                       */
                              /* the coordinates of the boundary pixels are  */
                              /* recorded in the xcoord and ycoord arrays    */
                              /* returns TRUE if error found                 */
                              /* FASLE otherwise                             */
int *nvalues;
```

```
int xbar,ybar;
{
double pi;
register    x,y;
register    r;
int    theta_deg;
double theta_rad;
int    success;
double value_sin;
double value_cos;
int    nwhite;

      pi = 4 * atan(1.0);
      x=y=256;


      *nvalues =0;


      /* scan radii at 1 degree increments */

      for ( theta_deg = 0; theta_deg < 360; theta_deg++ )
      {
         theta_rad = (double)theta_deg * ( pi / 180.0 );
         value_sin = sin(theta_rad);
         value_cos = cos(theta_rad);
         success=FALSE;
         x=xbar;
         y=ybar;
         nwhite=0;

         /* scan radius at 1 pixel increments - check point is  */
         /* actually within the image limits                    */


         for ( r = 0; ( r < 512 ) && ( x >= 0 ) && ( y >= 0 ) &&
                  ( x <= 511 ) && ( y <= 511 ) && (success==FALSE); r++)
         {

            /* polar to rectangular conversion */

              x = xbar + (int)((double)r*value_cos);
              y = ybar + (int)((double)r*value_sin);

            /* first five white pixels mark the boundary */

             if ( (readpixl(x,y) == GREY ) )
               nwhite++;
             else
               nwhite=0;

             if ( ( nwhite == 5 ) || ( x<=0) || (x>=512) || (y<=0) || (y>=512) )
             {
                if ( ( x<=0) || (x>=512) || (y<=0) || (y>=512) )
                  return(TRUE);
                success = TRUE;
                *(xcoord+(*nvalues)) = xbar + (int)((double)(r-5)*value_cos);
```

```
            *(ycoord + (*nvalues)) = ybar + (int)((double)(r-5)*value_sin);
            (*nvalues)++;


        }
    }
}

return(FALSE);

} /* end of perimeter */





new_centre(nvalues,xbar,ybar) /* calculates the centre of area based on the*/
                              /* boundary pixels ( coordinates stored in   */
                              /* xcoord and ycoord arrays )                */
                              /* nvalues is the number of boundary points  */
                              /* (xbar,ybar) returns the centre of area    */


int nvalues;
int *xbar,*ybar;
{
int   x,y;
long sum;




/* area = no of values*/

/* get centre of area */


    sum = 0;

    for ( x= 0; x <= nvalues; x++ )
       sum = sum + xcoord[x];



    *xbar = sum / nvalues;


    sum = 0;

    for ( x = 0; x <= 512; x++ )
         sum = sum + ycoord[x];
```

```
        *ybar = sum / nvalues;

 }  /* end of new_centre */


sig_check(nvalues,xbar,ybar)
int xbar;
int ybar;
int nvalues;
{
int error;
int dummy;
int i;
int minm;
int maxm;
int count;

        count = 0;
        minm = 32000;
        maxm = 0;
        error = FALSE;
        for ( i = 0; i < = nvalues-1; i + + )
        {
           dummy = (int)(sqrt((double)(xcoord[i]-xbar)*(double)(xcoord[i]-xbar)
                       + (double)(ycoord[i]-ybar)*(double)(ycoord[i]-ybar)/(ASPEC
T*ASPECT) ) );
           if ( dummy > maxm )
              maxm = dummy;
           if ( dummy < 10 )
              count + + ;

        }
           if ( count > 180 )
              return(TRUE);

        count = 0;
        for ( i = 0; i < = nvalues-1; i + + )
        {
           dummy = (int)(sqrt((double)(xcoord[i]-xbar)*(double)(xcoord[i]-xbar)
                       + (double)(ycoord[i]-ybar)*(double)(ycoord[i]-ybar)/(ASPEC
T*ASPECT) ) );
           if ( dummy < 1 )
             count + + ;
           else
              if ( maxm/dummy > DISTANCE_RATIO )
                 count + + ;
        }

          if ( count > MAX_POINTS )
            error = TRUE;

        return(error);
}
```

```
sig(nvalues,xbar,ybar,angle1,angle2,angle3,angle4)

int nvalues;
int xbar,ybar;            /* calculates signature of coordinate array */
int *angle1,*angle2,*angle3,*angle4;

{
int i;
int minm;
long sum;
double mean;
long  area;
int  dummy[360];

   int four = 4;
   int x,y;
   long maxm;
   int  mode;
   float x1, x2, y1, y2;
   float  xmax, ymax, xmin, ymin;
   int colour;
   static char text[13][10] ={
                              {"0"},  {"20"},  {"40"},  {"60"},  {"80"},
                              {"100"}, {"120"}, {"140"}, {"160"}, {"180"},
                              {"200"}, {"220"}, {"240"}
                             };
   static char anykey[30] = {"PRESS ANY KEY TO CONTINUE"};


      /* adjust the y values to allow for aspect ratio */


       nvalues--;
       for ( i=0; i< =nvalues; i+ + )
       {
          dummy[i] = (int)(sqrt((double)(xcoord[i]-xbar)*(double)(xcoord[i]-xbar)
                       + (double)(ycoord[i]-ybar)*(double)(ycoord[i]-ybar)/(ASPEC
T*ASPECT) ) );
       }


    for (i=2; i< =nvalues-2; i+ +)
       distance[i] = ( dummy[i] + dummy[i-1] + dummy[i-2]
                        + dummy[i+1] + dummy[i+2] ) / 5;

    distance[0] = ( dummy[0] + dummy[nvalues]   + dummy[nvalues-1]
                       + dummy[1]  + dummy[2] ) / 5;

    distance[1] = ( dummy[1] + dummy[nvalues]   + dummy[0]
                       + dummy[2]  + dummy[3] ) / 5;

    distance[nvalues-1] = ( dummy[nvalues-1] + dummy[nvalues-2]   + dummy[nvalues-3]
                       + dummy[nvalues]  + dummy[0] ) / 5;

    distance[nvalues] = ( dummy[nvalues] + dummy[nvalues-1]   + dummy[nvalues-2]
                       + dummy[0]  + dummy[1] ) / 5;
```

```
maxm = 0;
minm = 32000;
for ( x=1; x<= 254; x++ )
{
  if ( distance[x] > maxm )
  {
     maxm = distance[x];
     mode = x;
  }
  if ( distance[x] < minm )
     minm = distance[x];
}

if ( maxm / minm >= DISTANCE_RATIO )
  return(TRUE);


/* calculate mean of distances */


sum=0;
for (i=0; i<=nvalues; i++)
  sum+=distance[i];
mean = (double)sum/(double)nvalues;

area= (long)(4.0*(mean/1.1221997)*(mean/1.1221997));


/* find first corner - scan whole of distance array - 360 deg */


maxm = 0;
for ( x=0; x<360; x++ )
 if ( distance[x] > maxm )
 {

    maxm = distance[x];
    *angle1 = x;
 }

/* find second corner - scan over angle+45 deg -> angle+135 deg*/


maxm = 0;
for ( x=*angle1+45; x<=*angle1+135; x++ )
{

   if ( x >= 360 )
     y=x-360;
   else
     y=x;
```

```
        if ( distance[y] > maxm )
        {
          maxm = distance[y];
          *angle2 = y;
        }
      }

    /* third corner */



    maxm = 0;
    for ( x = *angle2 + 45; x < = *angle2 + 135; x + + )
     {

        if ( x > = 360 )
          y = x-360;
        else
          y = x;

        if ( distance[y] > maxm )
        {
          maxm = distance[y];
          *angle3 = y;
        }
     }

    /* fourth corner */


    maxm = 0;
    for ( x = *angle3 + 45; x < = *angle3 + 135; x + + )
     {

        if ( x > = 360 )
          y = x-360;
        else
          y = x;

        if ( distance[y] > maxm )
        {
          maxm = distance[y];
          *angle4 = y;
        }
     }

} /* end of signature */
```

```
centre_of_area(xbar,ybar)          /* calculates  the C of A for an entire */
int *xbar,*ybar;                   /* thresholded frame, coords of centre  */
{                   /* are returned as (xbar,ybar)          */
long   area;
register  x,y;
long    sumx;
long    sumy;



/* calculate area and centre of area */

        mov_buf(0);

        area = 0;
        sumx = 0;
        sumy = 0;

        for  ( x = 0; x < = 512; x+ + )
          for ( y= 0; y < = 512; y+ + )
              if  ( readpixl(x,y) = = BLACK )
              {
                 area+ +;
                 sumx + = x;
                 sumy + = y;
              }

        *xbar = sumx / area;
        *ybar = sumy / area;



}  /* end of centre_of_area */



screen1()
{

                        printf ("%c[44m",27);
                        system("cls");
                        printf ("%c[41m",27);
                        printf ("%c[37m",27);
                        printf ("%c[12;10f",27);
                        printf("
  ");
                        printf ("%c[13;10f",27);
   printf("    position the sample and press any key to continue      ");
                        printf ("%c[14;10f",27);
                        printf("
  ");
                        printf ("%c[13;63f",27);
                    getch();

}
```

```
screen2()
{
                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("              collecting grey level data
  ");
                printf ("%c[14;10f",27);
                printf("
  ");
                printf ("%c[13;63f",27);
}

screen3()
{
                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("         calculating the rough centre of area
  ");
                printf ("%c[14;10f",27);
                printf("
  ");
                printf ("%c[13;63f",27);
}

screen4()
{


                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("        calculating the perimeter and signature
  ");
                printf ("%c[14;10f",27);
                printf("
  ");
                printf ("%c[13;63f",27);
}
```

```
screen5()
{

                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("         calculating the accurate centre of area
  ");
                printf ("%c[14;10f",27);
                printf("
  ");
                printf ("%c[13;63f",27);

}

screen6()
{

                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("            fitting lines to the object
  ");
                printf ("%c[14;10f",27);
                printf("
  ");
                printf ("%c[13;63f",27);
}


screen7()
{

                printf ("%c[46m",27);
                system("cls");
                printf ("%c[44m",27);
                printf ("%c[37m",27);
                printf ("%c[12;10f",27);
                printf("
  ");
                printf ("%c[13;10f",27);
                printf("            binary smoothing the image
  ");
                printf ("%c[14;10f",27);
                printf("
```

```
 ");
                    printf ("%c[13;63f",27);
}


screen8()
{

                    printf ("%c[46m",27);
                    system("cls");
                    printf ("%c[44m",27);
                    printf ("%c[37m",27);
                    printf ("%c[12;10f",27);
                    printf("
 ");
                    printf ("%c[13;10f",27);
                    printf("            finding the largest region
 ");
                    printf ("%c[14;10f",27);
                    printf("
 ");
                    printf ("%c[13;63f",27);
}



perimeter_error()
{


        printf ("%c[41m",27);
        system("cls");
        printf ("%c[47m",27);
        printf ("%c[30m",27);
        printf ("%c[6;12f",27);
        printf("                                    ");
        printf ("%c[7;12f",27);
        printf("            PERIMETER CALCULATION ERROR            ");
        printf ("%c[8;12f",27);
        printf("                                    ");
        printf ("%c[9;12f",27);
        printf("  possible causes:-                    ");
        printf ("%c[10;12f",27);
        printf("    poor illumination or sample not in field of view    ");
        printf ("%c[11;12f",27);
        printf("                                    ");
        printf ("%c[18;12f",27);
        printf("            press any key to continue            ");
        printf ("%c[18;57f",27);
        printf("\07");
        printf("\07");

        getch();
        printf ("%c[0m",27);
        system("cls");
}
```

```
#include "\lc\stdio.h"
#include "\lc\dos.h"
#include "\lc\math.h"

extern void camera(); /* real time display(i/p buff,ilut,o/p buff,olut) */
extern void capture(); /* real time capture (i/p buff,ilut,o/p buff,olut) */
extern void ilut();  /* i/p look up table (table,index,value) */
extern void olut();  /* o/p look up table (table,index,value) */
extern void mov_buf(); /* move buffer to local buffer (buff) */
extern void res_buf(); /* restore to buffer(buff)*/
extern int read_pix(); /* read (x,y) pixel from buffer (x,y,buff) */
extern int readpixl(); /* read (x,y) pixel from local buffer (x,y) */
extern void wrt_pix(); /* write pixel to buffer (x,y,value,buffer) */
extern void wrtpixl(); /* write pixel to local buffer (x,y,value) */
extern void write2_word(); /* read/writes direct to buffer */
extern int  read1_word();
extern void write2_word(); /* (index,value) */
extern int  read2_word();  /* (value)       */

#define   WHITE    255
#define   BLACK    0
#define   GREY     150
#define   TRUE     1
#define   FALSE    0
#define   NAVERAGE 3   /* number of histogram averages */
#define   ASPECT   1.362
#define   SCALE    454     /* pixel/mm in x direction */
#define   SCALEX   454.0
#define   SCALEY   673.0


extern int xcoord[360];
extern int ycoord[360];


header(magnification,weight)
int magnification,weight;
{

        printf ("%c[46m",27);
        system("cls");
        set_up();
        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[2;6f",27);
        printf ("
   \n");
        printf ("%c[3;6f",27);
        printf ("                 Vickers Hardness Tester v4.0
   \n");
        printf ("%c[4;6f",27);
        printf ("          (c) 1989 I.C.Smith, University of Liverpool
   \n");
        printf ("%c[5;6f",27);
        printf ("
   \n");
        printf ("%c[46m",27);
```

```
            printf ("\n\n");
            printf ("%c[31m",27);
            printf ("       present settings:-  magnification = %d,  applied weight
 = %d Kgf\n",magnification,weight);
            printf ("\n\n");
            printf ("%c[30m",27);
            printf ("                    m");
            printf ("%c[34m",27);
            printf ("  alter Magnification \n\n");
            printf ("%c[30m",27);
            printf ("                    w");
            printf ("%c[34m",27);
            printf ("  alter applied Weight \n\n");
            printf ("%c[30m",27);
            printf ("                    p");
            printf ("%c[34m",27);
            printf ("  Process a sample \n\n");
            printf ("%c[30m",27);
            printf ("                    v");
            printf ("%c[34m",27);
            printf ("  verify hardness manually \n\n");
            printf ("%c[30m",27);
            printf ("                    x");
            printf ("%c[34m",27);
            printf ("  eXit program     \n\n");
            printf ("\n\n");

            printf ("     ");
            printf ("%c[44m",27);
            printf ("%c[37m",27);
            printf ("                    please enter an option
      ");

}



verify(magnification,weight)
int magnification,weight;
{

int x,y;
long x1,x2,x3,x4;
long y1,y2,y3,y4;
int diagonal1,diagonal2;
int mean_diagonal;
int answer;
double diag1,diag2,mean_diag;
int hardness;

        printf ("%c[44m",27);
        system("cls");
        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[12;18f",27);
```

```
 printf("                              ");
 printf ("%c[13;18f",27);
 printf("    Stretch the grey level scale (y/n) ?    ");
 printf ("%c[14;18f",27);
 printf("                              ");
 printf ("%c[13;59f",27);

if ( (answer=getch()) == 'y' )
  stretch();

printf ("%c[44m",27);
system("cls");
printf ("%c[41m",27);
printf ("%c[37m",27);
printf ("%c[7;15f",27);
printf("                                        ");
printf ("%c[8;15f",27);
printf("    Locate the corners of the indentation using    ");
printf ("%c[9;15f",27);
printf("            the following cursor keys:              ");
printf ("%c[10;15f",27);
printf("                                        ");
printf ("%c[11;15f",27);
printf("                  W                       ");
printf ("%c[12;15f",27);
printf("                A     S                   ");
printf ("%c[13;15f",27);
printf("                  Z                      ");
printf ("%c[14;15f",27);
printf("                                        ");
printf ("%c[15;15f",27);
printf("         Use the * key to mark the location         ");
printf ("%c[16;15f",27);
printf("    Use the SHIFT key to speed up the movement    ");
printf ("%c[17;15f",27);
printf("                                        ");

printf ("%c[20;15f",27);
printf("            press any key to continue              ");
getch();

capture(0,0,0,0);


printf ("%c[44m",27);
system("cls");
printf ("%c[41m",27);
printf ("%c[37m",27);
printf ("%c[11;15f",27);
printf("                                        ");
printf ("%c[12;15f",27);
printf("            locate the first corner              ");
printf ("%c[13;15f",27);
printf("                                        ");
x=y=255;
cursor(&x,&y,0,TRUE,1);
```

```
        x1=x;
        y1=y;


        printf ("%c[44m",27);
        system("cls");
        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[11;15f",27);
        printf("                                    ");
        printf ("%c[12;15f",27);
        printf("        now the opposite corner          ");
        printf ("%c[13;15f",27);
        printf("                                    ");
        cursor(&x,&y,0,TRUE,1);
        x2=x;
        y2=y;

        printf ("%c[44m",27);
        system("cls");
        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[11;15f",27);
        printf("                                    ");
        printf ("%c[12;15f",27);
        printf("         locate another corner          ");
        printf ("%c[13;15f",27);
        printf("                                    ");
        x=y=255;
        cursor(&x,&y,0,TRUE,1);
        x3=x;
        y3=y;


        printf ("%c[44m",27);
        system("cls");
        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[11;15f",27);
        printf("                                    ");
        printf ("%c[12;15f",27);
        printf("        now the opposite corner          ");
        printf ("%c[13;15f",27);
        printf("                                    ");
        cursor(&x,&y,0,TRUE,1);
        x4=x;
        y4=y;

        diagonal1=(int)(sqrt((double)((x2-x1)*(x2-x1)+(double)((y2-y1)*(y2-y1))/(ASPECT*A
SPECT))));
        diagonal2=(int)(sqrt((double)((x4-x3)*(x4-x3)+(double)((y4-y3)*(y4-y3))/(ASPECT*A
SPECT))));

        diag1= sqrt(        (double)(x2-x1)*(double)(x2-x1)/(SCALEX*SCALEX)
                          +(double)(y2-y1)*(double)(y2-y1)/(SCALEY*SCALEY) );
        diag2= sqrt(        (double)(x4-x3)*(double)(x4-x3)/(SCALEX*SCALEX)
                          +(double)(y4-y3)*(double)(y4-y3)/(SCALEY*SCALEY) );
```

```
    mean_diagonal = ( diagonal1 + diagonal2 )/2;
    mean_diag= ( diag1 + diag2 ) / 2.0;

    hardness = (int)(( 2.0 * weight * sin(68*3.1416/180) ) / ( mean_diag * mean_diag
 ) );



    printf ("%c[46m",27);
    system("cls");

    printf ("%c[41m",27);
    printf ("%c[37m",27);
    printf ("%c[2;6f",27);
    printf ("                                                          \n
");
    printf ("%c[3;6f",27);
    printf ("              Vickers Hardness Tester v4.0                \n
");
    printf ("%c[4;6f",27);
    printf ("          (c) 1989 I.C.Smith, University of Liverpool     \n
");
    printf ("%c[5;6f",27);
    printf ("                                                          \n
");
    printf ("%c[30m",27);
    printf ("%c[46m",27);

    printf("\n\n\n");
    printf ("\t        first diagonal length  = %d pixels\n",diagonal1);
    printf ("\t                               = %4.3f mm \n\n",diag1);
    printf ("\t        second diagonal length = %d pixels\n",diagonal2);
    printf ("\t                               = %4.3f mm \n\n",diag2);
    printf ("\t        mean diagonal length   = %d pixels\n",mean_diagonal);
    printf ("\t                               = %4.3f mm \n\n",mean_diag);
    printf ("\t        Hardness  =  %d HV%d",hardness,weight);
    printf ("%c[23;10f",27);
    printf ("%c[44m",27);
    printf ("%c[37m",27);
    printf ("            press any key to continue              ");

    getch();

    set_up(); /* restore default grey scale mapping */
}



vickers(magnification,weight,angle1,angle2,angle3,angle4)
int magnification,weight;
int angle1,angle2,angle3,angle4;
{
```

```
long x1,x2,x3,x4;
long y1,y2,y3,y4;
int diagonal1,diagonal2;
int mean_diagonal;
double diag1,diag2,mean_diag;
int hardness;



        x1 = xcoord[angle1];
        y1 = ycoord[angle1];
        x2 = xcoord[angle2];
        y2 = ycoord[angle2];
        x3 = xcoord[angle3];
        y3 = ycoord[angle3];
        x4 = xcoord[angle4];
        y4 = ycoord[angle4];


        draw_line((int)x1,(int)y1,(int)x2,(int)y2);
        draw_line((int)x2,(int)y2,(int)x3,(int)y3);
        draw_line((int)x3,(int)y3,(int)x4,(int)y4);
        draw_line((int)x4,(int)y4,(int)x1,(int)y1);


        diagonal1 = (int)(sqrt((double)((x3-x1)*(x3-x1) + (double)((y3-y1)*(y3-y1))/(ASPECT*A
SPECT))));
        diagonal2 = (int)(sqrt((double)((x4-x2)*(x4-x2) + (double)((y4-y2)*(y4-y2))/(ASPECT*A
SPECT))));

        diag1 = sqrt(       (double)(x3-x1)*(double)(x3-x1)/(SCALEX*SCALEX)
                          + (double)(y3-y1)*(double)(y3-y1)/(SCALEY*SCALEY) );
        diag2 = sqrt(       (double)(x4-x2)*(double)(x4-x2)/(SCALEX*SCALEX)
                          + (double)(y4-y2)*(double)(y4-y2)/(SCALEY*SCALEY) );

        mean_diagonal = ( diagonal1 + diagonal2 )/2;
        mean_diag = ( diag1 + diag2 ) / 2.0;

        hardness = (int)(( 2.0 * weight * sin(68*3.1416/180) ) / ( mean_diag * mean_diag
) );



        printf ("%c[46m",27);
        system("cls");

        printf ("%c[41m",27);
        printf ("%c[37m",27);
        printf ("%c[2;6f",27);
        printf ("                                                  \n
");
        printf ("%c[3;6f",27);
        printf ("            Vickers Hardness Tester v4.0          \n
");
        printf ("%c[4;6f",27);
        printf ("      (c) 1989 I.C.Smith, University of Liverpool  \n
```

```
");
     printf ("%c[5;6f",27);
     printf ("                                                    \n
");
     printf ("%c[30m",27);
     printf ("%c[46m",27);

     printf("\n\n\n");
     printf ("\t        first diagonal length  = %d pixels\n",diagonal1);
     printf ("\t                              = %4.3f mm \n\n",diag1);
     printf ("\t        second diagonal length = %d pixels\n",diagonal2);
     printf ("\t                              = %4.3f mm \n\n",diag2);
     printf ("\t        mean diagonal length   = %d pixels\n",mean_diagonal);
     printf ("\t                              = %4.3f mm \n\n",mean_diag);
     printf ("\t        Hardness   =   %d HV%d",hardness,weight);
     printf ("%c[23;10f",27);
     printf ("%c[44m",27);
     printf ("%c[37m",27);
     printf ("              press any key to continue              ");

     getch();

     set_up(); /* restore default grey scale mapping */
}




/**********************************************************************/
/*                                                                    */
/* name:     cursor(x,y,buffer )                                      */
/* sends:    x      - initial cursor column number |  MUST BE         */
/*           y      - initial cursor row number    |  POINTERS        */
/*           buffer - displayed buffer number - 0 or 1                */
/*           two_pass - true if routine to be called twice            */
/*                  e.g for windowing                                 */
/*           pass   -  number of times cursor invoked                 */
/* returns:  x      - final   cursor column number                    */
/*           y      - final   cursor row    number                    */
/* synopsis: displays white cursor over displayed buffer              */
/*           cursor may be moved using w a s z keys                   */
/*                                                                    */
/*                0 < =  x, y < = 511                                 */
/*                                                                    */
/**********************************************************************/


cursor ( x, y, buffer, two_pass, pass )
int *x;
int *y;
int buffer;
int two_pass;
int pass;
```

```
{
  static int store[40];
        int wspace[40];
        int i,j,move,oldx,oldy;


          /* read old pixel values */

          oldx = *x;
          oldy = *y;

          j = 0;
          for ( i = -9; i < = 9; i + + )
          {
            wspace[j] = read_pix( *x + i, *y, buffer );
            j + + ;
          }
          j = 20;
          for ( i = -9; i < = 9; i + + )
           {
            wspace[j] = read_pix( *x, *y + i, buffer );
            j + + ;
           }


          /* over write cursor */
          for ( i = -9; i < = 9; i + + )
              wrt_pix ( *x, *y + i, 255, buffer );
          for ( i = -9; i < = 9; i + + )
              wrt_pix ( *x + i, *y ,255, buffer );




      while ( (move = getch())!='*')
        {

          if ((move = = 'a')|| (move = = 's')||(move = = 'w')||(move = = 'z')||
             (move = = 'A')||(move = = 'S')||(move = = 'W')||(move = = 'Z') )
          {
             /* resore previous pixels */

             j = 0;
             for ( i = -9; i < = 9; i + + )
             {
                wrt_pix( *x + i, *y, wspace[j],  buffer );
                j + + ;
             }
             j = 20;
             for ( i = -9; i < = 9; i + + )
             {
                wrt_pix( *x, *y + i, wspace[j],  buffer );
                j + + ;
             }
            switch ( move )
             {
```

```
case 'a' : (*x)--;
           break;
case 's' : (*x)++;
           break;
case 'w' : (*y)--;
           break;
case 'z' : (*y)++;
           break;
case 'A' : (*x)-=10;
           break;
case 'S' : (*x)+=10;
           break;
case 'W' : (*y)-=10;
           break;
case 'Z' : (*y)+=10;
           break;

}
if ( *x < 0 )
   *x=511;
if ( *x > 511 )
   *x=0;
if (*y > 511 )
   *y=0;
if (*y < 0 )
   *y=511;


/* read old pixel values */

j=0;
for ( i=-9; i<=9; i++ )
{
  wspace[j] = read_pix( *x+i, *y, buffer );
  j++ ;
}
j=20;
for ( i=-9; i<=9; i++ )
{
  wspace[j] = read_pix( *x, *y+i, buffer );
  j++;
}


/* over write cursor */

for ( i=-9; i<=9; i++ )
   wrt_pix ( *x+i, *y, 255, buffer );

for ( i=-9; i<=9; i++ )
   wrt_pix ( *x, *y+i, 255, buffer );


   } /* endif */
} /* end of while */
```

```
    if ( two_pass = = FALSE )
    { /* restore previous pixels */
        j=0;
        for ( i=-9; i< =9; i+ + )
        {
            wrt_pix( *x+i, *y, wspace[j], buffer );
            j+ + ;
        }
        j=20;
        for ( i=-9; i< =9; i+ + )
        {
            wrt_pix( *x, *y+i, wspace[j], buffer );
            j+ +;
        }
    }
    else
    {
        if ( pass = = 1 )
        { /* write old pixel values to store */
            for ( i = 0; i < = 40; i+ + )
                store[i] = wspace[i];
        }
        else
        {  /* reinstate old values */
            j=0;
            for ( i=-9; i< =9; i+ + )
            {
                wrt_pix( *x+i, *y, wspace[j], buffer );
                j+ + ;
            }
            j=20;
            for ( i=-9; i< =9; i+ + )
            {
                wrt_pix( *x, *y+i, wspace[j], buffer );
                j+ +;
            }
            j = 0;
            for ( i=-9; i< =9; i+ + )
            {
                wrt_pix( oldx+i, oldy, store[j], buffer );
                j+ + ;
            }
            j=20;
            for ( i=-9; i< =9; i+ + )
            {
                wrt_pix( oldx, oldy+i, store[j], buffer );
                j+ +;
            }
        }
    }

} /*end of cursor */
```

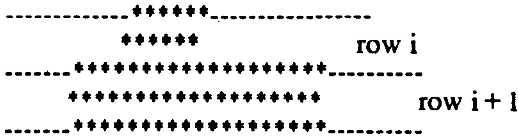
```
stretch()

/*
   alter the imput LUT grey level mapping
   so that all of the 256 possible grey levels
   are used in the image

*/

{
int x,y,max_grey,min_grey;
float scale;
int   grey_data;

  camera(0,0,0,0);
  capture(0,0,0,0);
  mov_buf(0);


  /* find highest grey level present */


  printf ("%c[46m",27);
  system("cls");
  printf ("%c[44m",27);
  printf ("%c[37m",27);
  printf ("%c[12;10f",27);
  printf("                                          ");
  printf ("%c[13;10f",27);
  printf("          collecting grey level data              ");
  printf ("%c[14;10f",27);
  printf("                                          ");
  printf ("%c[13;63f",27);

  max_grey = 0;
  min_grey = 255;

  for ( x=0; x<512; x++ )
   for ( y=0; y<512; y++ )
   {
     if ( readpixl(x,y) > max_grey )
          max_grey = readpixl(x,y);
     if ( readpixl(x,y) < min_grey )
          min_grey = readpixl(x,y);
   }


  /* scale grey LUT values */

  scale = 255.0 / ( (float)max_grey - (float)min_grey );

  for ( x = 0; x <=255; x++ )
  {
       grey_data = (int)((float)x * scale )-min_grey;
      if ( (grey_data >= 0) && (grey_data <=255) )
         ilut ( 0,x,grey_data );
```

```
        else
        {
          if ( grey_data < 0 )
            ilut ( 0,x,0 );
          else
            ilut ( 0,x,255 );
        }
    }

    capture(0,0,0,0);


} /* end of stretch */

thresh_error()
{


        printf ("%c[41m",27);
        system("cls");
        printf ("%c[47m",27);
        printf ("%c[30m",27);
        printf ("%c[6;12f",27);
        printf("                                              ");
        printf ("%c[7;12f",27);
        printf("              THRESHOLD CALCULATION ERROR              ");
        printf ("%c[8;12f",27);
        printf("                                              ");
        printf ("%c[9;12f",27);
        printf("      possible causes:-                               ");
        printf ("%c[10;12f",27);
        printf("         poor illumination or sample quality           ");
        printf ("%c[11;12f",27);
        printf("                                              ");
        printf ("%c[18;12f",27);
        printf("              press any key to continue               ");
        printf ("%c[18;57f",27);
        printf("\07");
        printf("\07");

        getch();
        printf ("%c[0m",27);
        system("cls");
}
```

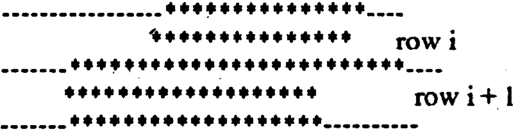
```
#include "\lc\stdio.h"
#include "\lc\dos.h"
#include "\lc\math.h"

#define TRUE   1
#define FALSE  0
#define GREY   150
#define BLACK  0
#define MAX_TRANS 20000 /* maximum permitted number of transitions */
#define MAX_LABEL 255   /* maximum permitted number of labels       */

code(count)
int *count;
{
int x,y;
int pixel1,pixel2;


    /* set last column all white */

    for ( y=0; y < 512; y++ )
      wrtpixl(511,y,GREY);


    *count = 0;

    /* run length code image */


    /* set first two rows blank ( white ) */

    write1_word(*count,0);
    (*count)++;
    write1_word(*count,1);
    (*count)++;

    for ( y=2; y < 510; y++ )
    {
      pixel1 = GREY;      /* first column all white */
      write1_word(*count,y);
      (*count)++;
      if ( *count > MAX_TRANS )
        return(TRUE);
      for ( x=1; x < 512; x++ )
      {
        pixel2 = readpixl(x,y);
        if ( (pixel1==BLACK) && (pixel2==GREY) )
        {
          write1_word(*count,x+1000); /* black to white transition */
          (*count)++;
        }
        else
          if ( (pixel2==BLACK) && (pixel1==GREY) )
          {
            write1_word(*count,x+2000); /* white to black  transition */
            (*count)++;
          }
```

```
            pixel1 = pixel2;
        }
    }

    /* image ends with two white lines - no runs */

    write1_word(*count,510);
    (*count) + + ;
    write1_word(*count,511);

    return(FALSE);

} /* end of code */




smooth()            /* returns TRUE if maximum number of transitions exceeded */
                   /*        FALSE othewise                                   */
{
int count;
int error;

    count = 0;
    error =  code(&count);
    if ( error )
    {
        trans_error();
        return(error);
    }
    expand(&count);
    expand(&count);
    shrink(&count);
    shrink(&count);
    delete(&count,10);

    return(error);

} /* end of smooth */


shrink(no_of_trans)
int *no_of_trans;
{
int i;          /* various loop variables                    */
int j,k;
int line_number1; /* index of ith line                       */
int line_number2; /* index of (i+1)th line                   */
int r1;          /* index of a run on the ith line           */
int r2;          /* index of a run on the (i+1)th line       */
                 /* each row must start with a W > B transition  */
                 /* each row must end with a B > W transition    */
                 /* OR be completely white - contain no runs     */
```

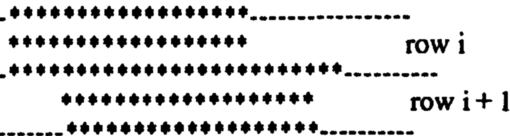
```
            /* W > B (left) transitions have odd indices          */
            /* W > B transitions have 2000 added to the x co-ord   */
            /* B > W (right) transitions have even indices        */
            /* B > W transitions have 1000 added to the x co-ord   */
int left;   /* left most pixel of new run written to second array */
int right;  /* right most "  "  "  "  "  "  "  "  */

int condition2;
int condition3;  /* connectivity conditions */
int condition4;


/* condition1 got lost for historical reasons

/**************************** condition2 ********************************
```



```
*************************************************************************/


/**************************** condition3 ********************************
```



```
*************************************************************************/


/**************************** condition4 ********************************
```



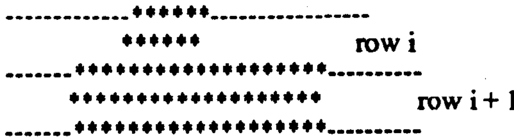
```
*************************************************************************/

    j = 0;
```

```
for (i=0; i<=*no_of_trans; i++)
{
  if ( read1_word(i) < 1000 )
  {
    write2_word(j,read1_word(i));
    j++;
  }
  else
   if (( read1_word(i) > 2000 ) && ( read1_word(i) < 3000 ) )
   {
     if ( read1_word(i) - 1000 != read1_word(i + 1)
     {
      /* white to black transition */

      write2_word(j,read1_word(i) + 1);
      j++;
     }
     else
         /* isolated black pixel */
         i++;

   }
   else
      if (( read1_word(i) < 2000) && ( read1_word(i) > 1000 ) )
        {
           /* black to white transition */

           write2_word(j,read1_word(i)-1);
           j++;
        }
}

*no_of_trans=j-1;

for ( i=0; i<=*no_of_trans; i++ )
   write1_word(i,read2_word(i) );


/* shrink in the y-direction */

j=0;  /* index for first run-length code array */
k=0;  /* index for second run-length code array */

line_number1=0;


while (1)
{
  /* write line number to second array */

  write2_word(k,read1_word(line_number1));
  k++;
  if ( read1_word(line_number1) == 511)
        break;

  /* get the first line index */
```

```
j++;
while ( read1_word(j) >= 1000)
j++;
line_number2=j;

if ( line_number2 - line_number1 > 1 ) /* if not a white line */
{
  r2 = 1;
  r1 = 1;
  while ( ( read1_word(line_number2+r2) > 1000 )   )
  {
    /* while there's a run on the (i+1)th line */

    /* check for end of file */

   condition2=condition3=condition4=FALSE;

   while (   ( read1_word(line_number2+r2+1)-1000
               >= read1_word(line_number1+r1)-2000 )
         && ( read1_word(line_number1+r1) > 1000   ) )
    {
      /* while there's a run on the ith line     */
      /* and rgt( s[i+1,q] ) >= lft( s[i,r] ) */


      condition2=condition3=condition4=FALSE;

      /* check for condition2 or condition3 */
      /* lft( s[i+1,q] ) <= lft( s[i+1,r] < rgt( s[i+1,q] ) */

      if ( ( read1_word(line_number2+r2)-2000 <= read1_word(line_number1+r1)-
2000 ) &&
          ( read1_word(line_number1+r1)-2000  < read1_word(line_number2+r2+1
)-1000 ) )
      {
            /* rgt( s[i+1,q] ) >= rgt( s[i,r] ) */

        if ( read1_word(line_number2+r2+1)-1000   <=   read1_word(line_number1
+r1+1)-1000 )
        {
          condition2=TRUE;
          left = read1_word(line_number1+r1);
          right = read1_word(line_number1+r1+1);
        }
        else
          {
           condition3=TRUE;
           left = read1_word(line_number1+r1);
           right = read1_word(line_number2+r2+1);
          }
     }
     else

       /* lft( s[i+1,q] ) < rgt( s[i,r] ) < rgt( s[i+1,q] ) */
       /* lft( s[i,r] ) < rgt ( s[i+1,q] ) */
```

```
            if ( ( read1_word(line_number2+r2)-2000 < read1_word(line_number1+r1+1
)-1000 ) &&
                 ( read1_word(line_number1+r1+1)-1000 < read1_word(line_number2+r2
+1)-1000 ) &&
                 ( read1_word(line_number1+r1)-2000 < read1_word(line_number2+r2)-
2000) )
            {
              condition4=TRUE;
              left  = read1_word(line_number2+r2);
              right = read1_word(line_number1+r1+1);
            }
          /* do the shrink */man

          if ( condition2 || condition3 || condition4 )
          {
            write2_word(k,left);
            k++;
            write2_word(k,right);
            k++;
          }
          else
            /* lft( s[i,r] ) <= lft ( s[i+1,q] ) */
            /* rgt( s[i,r] ) >= rgt ( s[i+1,q] ) */

            if ( ( read1_word(line_number1+r1)  <=  read1_word(line_number2+r2)
 ) &&
                 ( read1_word(line_number1+r1+1) >=  read1_word(line_number2+r2+
1)) )
            {
              write2_word(k,read1_word(line_number2+r2));
              k++;
              write2_word(k,read1_word(line_number2+r2+1));
              k++;
            }

          r1+=2; /* up date the ith row pointer */
        }
        r1-=2; /* point to last run on i not to the right of the run */
              /* on i+1 */
        if ( r1 < 1 )
          r1 = 1;

      r2+=2; /* up-date the (i+1)th row pointer */

      } /* end while */

    } /* end if */

    /* up date the line index */

    line_number1=line_number2;

  } /* end while */
```

```
    /* restore array */

    *no_of_trans=k-1;

    for ( i=0; i< =*no_of_trans; i+ + )
       write1_word(i,read2_word(i) );

} /* end of shrink */




expand (no_of_trans)  /* expands the bounday of every black pixel by   1 */


int *no_of_trans; /* total number of transitions                        */
{
int i;            /* various loop variables                        */
int j,k;
int line_number1; /* index of ith line                              */
int line_number2; /* index of (i+1)th line                          */
int r1;          /* index of a run on the ith line                 */
int r2;          /* index of a run on the (i+1)th line             */
                 /* each row must start with a W>B transition       */
                 /* each row must end with a B>W transition         */
                 /* OR be completely white - contain no runs        */
                 /* W>B (left) transitions have odd indices         */
                 /* W>B transitions have 2000 added to the x co-ord  */
                 /* B>W (right) transitions have even indices       */
                 /* B>W transitions have 1000 added to the x co-ord  */
int left;        /* left most pixel of new run written to second array */
int right;        /* right most  "  "  "  "   "   "   "   "  */

int condition2;
int condition3;   /* connectivity conditions */
int condition4;
int condition5;


/* condition1 got lost for hist0rical reasons

/**************************** condition2 *********************************
```




```
*************************************************************************/
```

```
/**************************** condition3 *********************************

            ----------------***************----
                           ***************    row i
            -------**************************----
                  *******************           row i+1
            -------*******************---------

*************************************************************************/

/**************************** condition4 *********************************

            -******************----------------
             ******************                row i
            -*************************---------
                  *******************          row i+1
            -------*******************---------


*************************************************************************/
/**************************** condition5 *********************************

            -*********************************---
             *********************************  row i
            -*********************************---
                  *******************          row i+1
            -------*******************---------


*************************************************************************/


    j = 0;

    for (i=0; i<=*no_of_trans; i++)
    {
      if ( read1_word(i) < 1000 )
      {
        write2_word(j,read1_word(i));
        j++;
      }
      else
        if ( read1_word(i) > 2000 )
        {
          /* white to black transition */
          if ( read1_word(i) > 2001 )
             write2_word(j,read1_word(i)-1);
          else
```

```
            write2_word(j,read1_word(i));
          j++;
        }
      else
          if ( read1_word(i) > 1000 )
            if ( ( read1_word(i+1))-read1_word(i))
            {
               /* black to white transition */

              if ( read1_word(i) < 1500 )
                write2_word(j,read1_word(i) + 1);
              else
                write2_word(j,read1_word(i));
              j++;
            }
            else
            {
              /* single white pixel */

              write2_word(j,read1_word(i+2) );
              i+=2;
              j++;
            }
  }

  *no_of_trans=j-1;

  for ( i=0; i<=*no_of_trans; i++ )
     write1_word(i,read2_word(i) );



 /* expand  in the y-direction */

j=0;  /* index for first run-length code array */
k=0;  /* index for second run-length code array */

line_number1 = 0;

while (1)
{
   /* write line number to second array */

   write2_word(k,read1_word(line_number1));
   k++;
   if ( read1_word(line_number1) == 511 )
      break;

   /* get the first line index */

  j++;
  while ( read1_word(j) >= 1000 )
  j++;
  line_number2=j;
```

```
if ( ( line_number2 - line_number1 > 1 ) && ( read1_word(line_number2) > 10 ) )
  /* if not a white line */
  {
    r2 = 1;
    r1 = 1;
    while ( ( read1_word(line_number2+r2) > 1000 )   )
    {
      /* while there's a run on the (i+1)th line */

      left = read1_word(line_number2+r2);
      right = read1_word(line_number2+r2+1);

     condition2=condition3=condition4=condition5=FALSE;

     while (   ( read1_word(line_number2+r2+1)-1000
                 >= read1_word(line_number1+r1)-2000 )
            && ( read1_word(line_number1+r1) > 1000   )
            && ( read1_word(line_number2+r2) > 1000   ) )
      {
        /* while there's a run on the ith line    */
        /* and rgt( s[i+1,q] ) >= lft( s[i,r] ) */

        /* initialise left and right to lft and rgt of */
        /* run on (i+1)th line                         */


        condition2=condition3=condition4=condition5=FALSE;


        /* check for condition2 or condition3 */
        /* lft( s[i+1,q] ) <= lft( s[i+1,r] <= rgt( s[i+1,q] ) */

        if ( ( read1_word(line_number2+r2)-2000 <= read1_word(line_number1+r1)-
2000 ) &&
             ( read1_word(line_number1+r1)-2000  <= read1_word(line_number2+r2+
1)-1000 ) )
        {
            /* rgt( s[i+1,q] ) >= rgt( s[i,r] ) */

          if ( read1_word(line_number2+r2+1)-1000  >=  read1_word(line_number1
+r1+1)-1000 )
            condition2=TRUE;
          else
            {
             condition3=TRUE;
             right = read1_word(line_number1+r1+1);


            }
       }
       else

         /* lft( s[i+1,q] ) <= rgt( s[i,r] ) < rgt( s[i+1,q] ) */
         /* lft( s[i,r] ) < rgt ( s[i+1,q] ) */

         if ( ( read1_word(line_number2+r2)-2000 <= read1_word(line_number1+r1+
1)-1000  ) &&
```

```
                ( read1_word(line_number1+r1+1)-1000 < read1_word(line_number2+r2
+1)-1000 ) &&
                ( read1_word(line_number1+r1)-2000 < read1_word(line_number2+r2)-
2000)  )
            {
              condition4=TRUE;
              left = read1_word(line_number1+r1);
            }

            else

             /* lft( s[i,r] ) < lft( s[i+1,q] ) */
             /* rgt( s[i,r] ) > rgt( s[i+1,q] ) */

             if ( ( read1_word(line_number1+r1) < read1_word(line_number2+r2 ) )
 &&
                ( read1_word(line_number1+r1+1) > read1_word(line_number2+r2+1)
))

            {
              condition5=TRUE;
              left = read1_word(line_number1+r1);
              right = read1_word(line_number1+r1+1);
            }


           /* if run on ith row not connected to run on (i+1)th row */
           /* then write it to the second array                    */

          if (             condition2 &&      condition3 && condition4 && condition5 )
          {
            write2_word(k,read1_word(line_number1+r1));
            k++;
            write2_word(k,read1_word(line_number1+r1+1));
            k++;
/*
printf ("\n 1: line=%d left= %d, right= %d, r1=%d, r2=%d",read1_word(line_number1),
read1_word(line_number1+r1)-2000,read1_word(line_number1+r1+1)-1000,r1,r2);
*/
          }



         r1+=2; /* up date the ith row pointer */
         }

         /* check for a branch point */

        if ( condition2 || condition4)
        {
          /* do the expansion */

          write2_word(k,left);
          k++;
          write2_word(k,right);
          k++;
```

```
/*
printf ("\n 2: line = %d  left =  %d,  right =  %d, r1 = %d, r2 = %d",read1_word(line_number1),
left-2000,right-1000,r1,r2);
*/
          }
          if ( condition3 || condition5 )
          {
             do
             {
                /* rgt( s[i,r] ) > = lft( s[i + 1,q] ) and not end of line */

                r2 + = 2;
             }
             while (  ( read1_word(line_number1 + r1-1)-1000  > =  read1_word(line_number
2 + r2)-2000 )
                 &&( read1_word(line_number2 + r2 )  >  1000 ) );

            r2- = 2;
            if ( r2  <  1 )
               r2  =  1;

                /* rgt( s[i,r] ) > rgt( s[i + 1,q] */

            if (  read1_word(line_number1 + r1 + 1)  > =  read1_word(line_number2 + r2 + 1) )
            {

              write2_word(k,left);
              k + + ;
              write2_word(k,right);
              k + + ;
/*
printf ("\n 3: line = %d  left =  %d,  right =  %d, r1 = %d, r2 = %d",read1_word(line_number1),
left-2000,right-1000,r1,r2);
*/
            }

            else
            {
             r1- = 2;
             r2- = 2;
             write1_word(line_number1 + r1,left);
/*
             if ( r2  <  1 )
                r2  =  1;
*/
            }

          }

        r2 + = 2; /* up-date the (i + 1)th row pointer */

        if ( r2  <  1 )
            r2  =  1;

        } /* end while */

      } /* end if */
```

```
      /* up date the line index */

      line_number1 = line_number2;

    }  /* end while */

    *no_of_trans = k-1;

    /* restore array */

    for ( i=0; i< = *no_of_trans; i+ + )
       write1_word(i,read2_word(i) );


  } /* end of expand  */



delete ( no_of_trans, n ) /* deletes all runs of length n or less */

int *no_of_trans;
int n;
{
int j,k;


     j=0;
     k=0;


     do
     {
       if ( read1_word(j) <  1000 )
       {
         write2_word(k,read1_word(j));
         j+ +;
         k+ +;
       }
       else

           if ( ( read1_word(j+1) - read1_word(j) + 1000 ) < = n  )
               j+ =2;
            else
            {
               write2_word(k,read1_word(j));
               j+ +;
               k+ +;
               write2_word(k,read1_word(j));
               j+ +;
               k+ +;
            }
    } while ( j< = *no_of_trans );

    *no_of_trans=k-1;
```

```
    for ( j=0; j<=*no_of_trans; j++ )
       write1_word(j,read2_word(j) );


} /* end of delete */


blob(x_bar,y_bar)        /* perform region labelling */
int *x_bar;
int *y_bar;

{
int label;    /* label number */
int j;        /* index for run length code list */
int y;        /* row number */
int i;
int connection;
int r1;
int r2;
int line_number1;
int line_number2;
long blob_area[256];
long region_area[256];
int  convlist1[256];
int  convlist2[256];
int  stack_address[256];
int  no_of_stacks;
int  in_list;
long max_region_area;
long max_blob_area;
int  blob_index;
int  region_index;
int  dummy;
int  no_of_converge;
int  x;
long x_sum;
long  y_sum;


          /* label list has corresponding indices to run list */
          /* value = 2000 represents a blank */
          /* 1000 <= value < 2000  y co-ordinate */
          /* value < 1000          label number  */




    /* intialise the area table */

   for ( j=0; j<256; j++ )
     blob_area[j] =0;
```

```
j = 0;
line_number1 = 0;
no_of_converge = 0;
label = 0;
write2_word(0,2000); /* first line */


while (1)
{


   /* get the first line index */


   j++;
   while ( read1_word(j) >= 1000 )
      j++;
   line_number2 = j;
   y = read1_word(line_number2);

   /* write blank to label list */


  write2_word(line_number2,2000);
  if ( y == 511 )
    break;

  if ( line_number2 - line_number1 > 1 )
  {
    r2 = 1;
    r1 = 1;
    while ( read1_word(line_number2 + r2) > 1000 )
    {
      /* while there's a run on the (i + 1)th line */

     connection = FALSE;

     while (  ( read1_word(line_number2 + r2 + 1)-1000
                 >= read1_word(line_number1 + r1)-2000 )
           && ( read1_word(line_number1 + r1) > 1000  ) )
      {
        /* while there's a run on the ith line    */
        /* and rgt( s[i + 1,q] ) >= lft( s[i,r] ) */

        /* test for a connection */
        /* lft ( s[i + 1,q] ) <= rgt ( s[i,r] ) <= rgt ( s[i + 1,q] ) */
     /* OR  lft ( s[i + 1,q] ) <= lft ( s[i,r] ) <= rgt ( s[i + 1,q] ) */
     /* OR  lft ( s[i + 1,q] ) < lft ( s[i,r] ) AND rgt( s[i + 1,q] ) > rgt ( s[i,r]
) */


          if ( ( read1_word(line_number2 + r2)-2000   <= read1_word(line_number1 + r1
)-2000   ) &&
               ( read1_word(line_number1 + r1)-2000   <= read1_word(line_number2 + r2
+ 1)-1000  ) ||
               ( read1_word(line_number2 + r2)-2000   <= read1_word(line_number1 + r1
```

```
+ 1)-1000  )  &&
                  ( read1_word(line_number1 + r1 + 1)-1000  < =  read1_word(line_number2 + r2
+ 1)-1000  )  ||
                  ( read1_word(line_number1 + r1)          <  read1_word(line_number2 + r2
)        )  &&
                  ( read1_word(line_number1 + r1 + 1)          >  read1_word(line_number2 + r2 +
1)        )  )

              {
                 if ( connection  = =  FALSE )  /* test for previous connection */
                 {
                    /* assign run on i + 1 with label of run on i  */

                    write2_word (line_number2 + r2,read2_word(line_number1 + r1));

                    write2_word (line_number2 + r2 + 1,y + 1000);
                    connection  =  TRUE;
                    blob_area[read2_word(line_number1 + r1)] + = ( read1_word(line_numbe
r2 + r2 + 1) - read1_word(line_number2 + r2)  +  1000  );
                 }
                 else
                       /* sort out convergence */

                     if ( read2_word(line_number2 + r2)[illegible]r1 + r
1))
                     {
                        in_list =  FALSE;
                        for ( i = 0; i  <  no_of_converge; i +  +  )

                         /* is this convergence in the list */

                         if ( ( convlist1[i]  = =  read2_word(line_number1 + r1) ) &
& ( convlist2[i]  = =  read2_word(line_number2 + r2) ) ||
                              ( convlist1[i]  = =  read2_word(line_number2 + r2) ) &
& ( convlist2[i]  = =  read2_word(line_number1 + r1) ) )
                         {
                            in_list  =  TRUE;
                            break;
                         }

                        if (  ( in_list  = =  FALSE ) || ( no_of_converge  = =  0 ) )
                        {
                           convlist1[no_of_converge] = read2_word(line_number2 + r2
);
                           convlist2[no_of_converge] = read2_word(line_number1 + r1
);
                           no_of_converge +  + ;

                        }

                     }
              }
           r1 +  = 2; /* up date the ith row pointer */
           }
           r1- = 2; /* point to last run on i not to the right of the run */
                 /* on i + 1 */
```

```
        if ( r1  <  1 )
          r1  =  1;


      if ( connection  = =  FALSE )
      {
          /* if no connection found for this run assign a new label */


          write2_word (line_number2 + r2,label);
          write2_word (line_number2 + r2 + 1,y + 1000);
          blob_area[label] + = ( read1_word(line_number2 + r2 + 1) - read1_word(line_numbe
r2 + r2)  +  1000  );
          label + + ;
          if ( label  = =  MAX_LABEL )
          {
             label_error();
             return(TRUE);
          }
      }





      r2 + = 2; /* up-date the (i + 1)th row pointer */

      } /* end while */

    } /* end if */
    else
      if ( read1_word(line_number2 + 1)  > =  1000 )
      {
        r2 = 1;
       while ( read1_word(line_number2 + r2)  > =  1000 )
       {
           write2_word (line_number2 + r2,label);
           write2_word (line_number2 + r2 + 1,y + 1000);
           blob_area[label] + = ( read1_word(line_number2 + r2 + 1) - read1_word(line_nu
mber2 + r2)  +  1000  );
           label + + ;
           if ( label  = =  MAX_LABEL )
           {
              label_error();
              return(TRUE);
           }
           r2 + = 2;
       }
      }

    /* up date the line index */

    line_number1 = line_number2;

  }  /* end while */
```

```
label--;


capture(0,2,0,0);
for (i=0;i<255;i++)
    ilut(2,i,GREY);
capture(0,2,0,0);


 if ( no_of_converge > 0 )
 {
    /* sort out equivalent stacks */

    for ( i=0; i<=255; i++ )
      stack_address[i] = 300;    /* set all addresses to illegal values */

    no_of_stacks = -1;

    for ( i=0; i<=no_of_converge; i++ )    /* search the label lists */
    {
      if (    ( stack_address[convlist1[i]] == 300 )
          && ( stack_address[convlist2[i]] == 300 ) )
      {
        /* neither label is in a stack */
        /* assign both labels to a new stack */

        no_of_stacks++;
        stack_address[convlist1[i]] = stack_address[convlist2[i]]
                                   = no_of_stacks;


      }
      else
        if (    ( stack_address[convlist1[i]]
            && ( stack_address[convlist2[i]]
        {
          /* both labels are in stacks */
          /* change addresses of all labels in the same stack as */
          /* the first label to that of the second  */

          if ( stack_address[convlist2[i]] > stack_address[convlist1[i]] )
          {
             dummy = stack_address[convlist2[i]];
             for ( j=0; j<=label; j++ )
                if ( stack_address[j] == dummy )
                    stack_address[j] = stack_address[convlist1[i]];
          }
          else
          {
             dummy = stack_address[convlist1[i]];
             for ( j=0; j<=label; j++ )
                if ( stack_address[j] == dummy )
                   stack_address[j] = stack_address[convlist2[i]];
          }
          for ( j=0; j<=label; j++ )
```

```
            if ( (stack_address[j]!=300)&&(stack_address[j] > dummy ) )
                stack_address[j]--;
            no_of_stacks--;


        }
        else
          if ( stack_address[convlist1[i]]!=300)

            /* first label is in a stack - second isn't */
            /* assign second label to same stack as first */

            stack_address[convlist2[i]] = stack_address[convlist1[i]];

        else

           /* second label is in a stack - first isn't */
           /* assign first label to same stack as second */

           stack_address[convlist1[i]] = stack_address[convlist2[i]];
    } /* end  for */

} /* end if */

max_region_area = 0;

if ( no_of_converge > 0 )
{
   /* add up the blob areas to give region areas */


   for ( j=0; j<=no_of_stacks; j++)
      region_area[j]=0;

   for ( j=0; j<=no_of_stacks; j++)
      for ( i=0; i<=label; i++ )
         if ( stack_address[i] == j )
            region_area[j] += blob_area[i];


   /* find the largest region */


   for ( j=0; j<=no_of_stacks; j++ )
      if ( region_area[j] > max_region_area )
      {
         max_region_area=region_area[j];
         region_index = j;
      }

}


/* find the largest blob */
```

```
max_blob_area = 0;
for ( j=0; j< =label; j+ + )
  if ( blob_area[j] > max_blob_area )
  {
      max_blob_area=blob_area[j];
      blob_index =j;
  }


/* locate largest region/blob */
/* calculate it's centre of area */




x_sum=y_sum=0;

if ( max_region_area > max_blob_area )
{
   for ( i=0; i< =label; i+ + )
   {
      if ( stack_address[i] = = region_index )
      {
        j=0;
        while ( read1_word(j))
        {
           if ( read2_word(j) = = i )
           {
             y = read2_word(j+1)-1000;
             for ( x=read1_word(j)-2000; x< =read1_word(j+1)-1000; x+ + )
             {
                y_sum+ =y;
                x_sum+ =x;
             }

/*              y_sum+ =(read2_word(j+1)-1000);
             x_sum+ =(( (long)( read1_word(j+1)-read1_word(j)+1001 )
                      *(long)( read1_word(j+1)+read1_word(j)+1000 ) ) / 2); */

           }

          j+ +;
        }
      }
   }
   *x_bar = x_sum / max_region_area;
   *y_bar = y_sum / max_region_area;
}
else
{
   j=0;
   while ( read1_word(j)!11 )
   {
      if ( read2_word(j) = = blob_index )
      {
         y = read2_word(j+1)-1000;
         for ( x=read1_word(j)-2000; x< =read1_word(j+1)-1000; x+ + )
```

```
            {
              y_sum + = y;
              x_sum + = x;
             }
/*            y_sum + = (read2_word(j + 1)-1000);
            x_sum + = (( (long)( read1_word(j + 1)-read1_word(j) + 1001 )
                  *(long)( read1_word(j + 1) + read1_word(j) + 1000 ) ) / 2); */
          }
          j + + ;
        }
        *x_bar = x_sum / max_blob_area;
        *y_bar = y_sum / max_blob_area;
      }



      return(FALSE);

} /* end of blob */



trans_error()
{


      printf ("%c[41m",27);
      system("cls");
      printf ("%c[47m",27);
      printf ("%c[30m",27);
      printf ("%c[6;12f",27);
      printf("                                        ");
      printf ("%c[7;12f",27);
      printf("            TOO MANY TRANSITIONS                  ");
      printf ("%c[8;12f",27);
      printf("                                        ");
      printf ("%c[9;12f",27);
      printf("     possible causes:-                      ");
      printf ("%c[10;12f",27);
      printf("       analogue pick up, poor sample quality,        ");

      printf ("%c[11;12f",27);
      printf("       dust on microscope lenses                ");

      printf ("%c[12;12f",27);
      printf("                                        ");
      printf ("%c[18;12f",27);
      printf("            press any key to continue             ");
      printf ("%c[18;57f",27);
      printf("\07");
      printf("\07");

      getch();
      printf ("%c[0m",27);
```

```
        system("cls");
}


label_error()
{

        printf ("%c[41m",27);
        system("cls");
        printf ("%c[47m",27);
        printf ("%c[30m",27);
        printf ("%c[6;12f",27);
        printf("                                        ");
        printf ("%c[7;12f",27);
        printf("              TOO MANY LABELS                  ");
        printf ("%c[8;12f",27);
        printf("                                        ");
        printf ("%c[9;12f",27);
        printf("      possible causes:-                      ");
        printf ("%c[10;12f",27);
        printf("         poor sample quality, crystal defects          ");

        printf ("%c[11;12f",27);
        printf("                                        ");
        printf ("%c[18;12f",27);
        printf("             press any key to continue              ");
        printf ("%c[18;57f",27);
        printf("\07");
        printf("\07");

        getch();
        printf ("%c[0m",27);
        system("cls");
}
```

APPENDIX VIII

C PROGRAM CODE FOR THE QA SYSTEM

function: calculate_profile (distance, x_bar, y_bar, minor, major, rotation border_x, border_y)

non-pointer arguments:-

int x_bar	x co-ordinate of the ellipse centroid
int y_bar	y co-ordinate of the ellipse centroid
int minor	ellipse minor axis length
int major	ellipse major axis length
int rotation	major axis rotation from the horizontal in degrees

pointer arguments:

int distance	array holding deviation values
int border_x	array holding x co-ordinates of border pixels
int border_y	array holding y co-ordinates of border pixels

returns: none

called functions: none

description:

This function is used to calculate the deviation of the inner edge of the disc from the fitted ellipse. The deviation of each border point is found by applying equations [5.26]-[5.35] to the co-ordinates of the border points and the corresponding points on the fitted ellipse. The deviation values are returned in a 360 element array indexed as rotation angle in degrees about the ellipse centroid from the horizontal.

function: centre_of_area (xbar,ybar)

non-pointer arguments:

	int xbar	x co-ordinate of the disc centroid
	int ybar	y co-ordinate of the disc centroid

pointer arguments: none

returns: none

called functions: none

definition:

This function makes an estimate of the centroid of the disc based on all of the pixels within the image using equations [5.1],[5.2]. Here:-

f(x,y) = 1 for object pixels (labelled as BLACK)

f(x,y) = 0 for background pixels (labelled as GREY)

function: chip_border_follow (x_bar, y_bar, npoints, length)

non-pointer arguments:

	int x_start	x co-ordinate of the border starting point
	int y_start	y co-ordinate of the border starting point

pointer arguments:

	int npoints	number of pixels in the border contour
	int length	border length

returns:

	int error	TRUE- if seed pixels are encircled by contour; FALSE- otherwise.

called functions: none

description:

This function implements the border following algorithm given in section 5.3.4 to trace the border of a chip or crack feature. Each border pixel is assigned grey level CHIP and the length of the border is calculated using equation [5.36].

function: chip_fill()

arguments: none

returns: none

called functions: chip_border_follow(), fill_chips()

description:

This function is used to identify chip and crack regions within the disc, calculate size and shape parameters for each feature and present the data to the user in tabular form. The image array is scanned row by row from top to bottom until a suitable pair of seed pixels are found using the requirements shown in Fig 5.14. A pair of pixels which may be used as starting points for the border following algorithm given in section 5.3.4 are then located by scanning further along the row containing the lower seed pixel until a pair of adjacent dissimilar pixels are found. The border of the region is traced by calling the chip_border_follow() function and a check made to ensure that the seed pixels are encircled by the border contour. If this is not the case then scanning continues, otherwise the region enclosed by the contour is filled by calling the chip_fill() function. For each contour which is filled the area of the region is calculated along with the maximum and minimum bounding radius (from the region centroid) of the contour, the position of the region centroid and the compactness (*perimeter²/area*).

function: draw_profile (distance, draw_inner)

non-pointer arguments:

	int draw_inner	TRUE- if inner edge profile is to be plotted; FALSE- otherwise.

pointer arguments:

	int distance	pointer to array holding deviation values

called functions: none

returns: none

description:

This function plots the profile of a disc edge using the edge deviation data held in an array. The mean, maximum, minimum, and RMS deviations are also calculated and presented to the user. The function makes extensive use of routines from the HALO graphics library to plot the data. These routines are highly specific and will not be discussed here; for a detailed description see [46].

function: ellipse_results (sum_x, sum_y, sum_x_squared, sum_y_squared, sum_xy, area, x_bar, y_bar, major, rotation)

non-pointer arguments:

	double sum_x	$\sum x$ for region
	double sum_y	$\sum y$ for region
	double sum_x_squared	$\sum x^2$ for region
	double sum_y_squared	$\sum y^2$ for region
	double sum_xy	$\sum xy$ for region

pointer arguments:

	int x_bar	x co-ordinate of ellipse centroid
	int y_bar	y co-ordinate of ellipse centroid
	int major	major axis length of ellipse
	int minor	minor axis length of ellipse
	int rotation	major axis rotation from the horizontal

returns: none

called functions: none

description:

This function calculates the parameters of an approximating ellipse for a region given the summation parameters ($\sum x$ etc) using equations [5.17]-[5.25].

function: fill_chips (x_bar, y_bar, x_centre, y_centre, area)

non-pointer arguments:

	int x_bar	x co-ordinate of lower seed pixel
	int y_bar	y co-ordinate of lower seed pixel

pointer arguments:

	int x_centre	x co-ordinate of region centroid
	int y_centre	y co-ordinate of region centroid
	int area	area of region

called functions: link2(), left2(), lright2(), push2(), push2d(), pop2(), pop2d()

returns: none

description:

This function fills a chip or crack region using the region filling algorithm given in section 5.3.5. Each pixel within the bounding contour is assigned grey level CHIP2 and equations [5.1],[5.2] are used to calculate the centroid of the filled region.

function: fill_hole (xbar, ybar, sum_x, sum_y, sum_x_squared, sum_y_squared, sum_xy, area)

non-pointer arguments:

int xbar	x co-ordinate of the disc centroid
int ybar	y co-ordinate of the disc centroid

pointer arguments:

double sum_x	$\sum x$
double sum_y	$\sum y$
double sum_x_squared	$\sum x^2$
double sum_y_squared	$\sum y^2$
double sum_xy	$\sum xy$
double area	number of pixels in the filled region

returns: none

called functions: left(), link(), push(), pushd(), pop(), popd()

description:

This function fills the hole region of the disc using the algorithm given in section 5.3.5, each pixel within the filled region being assigned grey level ALMOST_WHITE. As each row is filled the summations required for calculating the ellipse parameters (equations [5.8] - [5.10]) are updated using equations [5.17] - [5.25].

The stack holding the seed pixel addresses is realised as a pair of arrays (one for the x co-ordinates and one for the y co-ordinates) and is manipulated using the push() and pop() functions. The stack holding the scanning directions for seed pixels is realised using another array and is manipulated using the popd() and pushd() functions. Two global variables are used to hold the stack pointers so that the positions are not lost during calls to other functions.

function: inner_border_follow (xbar,ybar)

non-pointer arguments:

	int xbar	co-ordinate of the disc centroid
	int ybar	y co-ordinate of the disc centroid

pointer arguments: none

returns:

int error	TRUE- if inner border and outer borders are traced; FALSE- if only inner border is traced.

called functions: none

definition:

This function traces the inner edge of the disc using the algorithm given in section 5.3.4. The image array is first scanned in the + x direction, starting at the centroid, until five contiguous pixels labelled as BLACK are found. The first BLACK pixel and it's left hand neighbour are then used as the starting point for the border. Each border pixel is assigned grey level WHITE and for each pixel the distance from the centroid is calculated using equation [5.28] and the angle of rotation about the centroid from the horizontal is calculated using equation [5.31].

A 360 element array, indexed by angle in degrees, is used to store the distance values. If the distance from the centroid of a given border point is greater than the value held in the array element having the same angle then the array value is overwritten by the new distance value (all the array values are initialised to zero at the start of the function). The co-ordinates of the border pixel are written to two arrays (one for x and one for y) in this case. These co-ordinate arrays contain 360 elements and are also indexed by rotation angle in degrees.

The final action of the function is to calculate the mean border distance from the centroid and check if this is greater than the mean of the nominal inner and outer edge radius. As indicated in section 5.3.4, if this is so then both the inner and outer edges of

the disc must have been traced therefore the disc contains a disc from the inner to the outer edge. In this case the error variable is returned TRUE otherwise it is returned FALSE.

function: left (x,y)

non-pointer arguments:

	int x	x co-ordinate of pixel address, p
	int y	y co-ordinate of pixel address, p
pointer arguments:	none	
returns:		
	int x	x co-ordinate of pixel address, LEFT(p)
called functions:	none	

description:- This function implements the LEFT function described in section 5.3.5.

function: lright (x,y)

non-pointer arguments:

	int x	x co-ordinate of pixel address, p
	int y	y co-ordinate of pixel address, p

pointer arguments: none

returns:

	int x	x co-ordinate of pixel address, LRIGHT(p)

called functions: none

description:

This function implements the LRIGHT function described in section 5.3.5.

function: link (x, y, above, below, e1, e2, p1)

non-pointer arguments:

	int x	x co-ordinate of input pixel address
	int y	y co-ordinate of input pixel address

pointer arguments:

	int above, below	as defined in section 5.3.5
	int e1, e2, p1	as defined in section 5.3.5

returns: none

called functions: none

description:

This function implements the LINK algorithm given in section 5.3.5. If any of the pixel addresses e1, e2, or p1 are undefined an illegal value (=-1) is returned.

functions: link2(), left2(), lright2(), push2(), pop2() push2d(), pop2d()

description:

These functions operate in the same manner as link(), left() etc but are used in conjunction with the fill_chips() function rather than the fill_whole() and fill_disk() functions.

function: outer_border_follow (xbar,ybar)

non-pointer arguments:

	int x_bar	x co-ordinate of the centroid
	int y_bar	y co-ordinate of the centroid
pointer arguments:	none	
returns:	none	
called functions:	none	

description:

This function traces the outer border of the disc using the algorithm given in section 5.3.4. The image array is first scanned in the +x direction, starting at the first column, until five contiguous pixels labelled as BLACK are found. The first BLACK pixel and it's left hand neighbour are then used as the starting point for the border. Each border pixel is assigned grey level WHITE and for each pixel the distance from the centroid is calculated using equation [5.28] and the angle of rotation about the centroid from the horizontal is calculated using equation [5.31].

A 360 element array, indexed by angle in degrees, is used to store the distance values. If the distance from the centroid of a given border point is less than the value held in the array element having the same angle then the array value is overwritten by the new distance value (all of the array values are initialised to largest possible int at the start of the function). The co-ordinates of the border pixels are written to two arrays (one for x and one for y) in this case. These co-ordinate arrays contain 360 elements and are also indexed by rotation angle in degrees.

function: pop (x_stack, y_stack, px, py)

pointer arguments:

int x_stack	array holding the x co-ordinates of the pixel addresses
int y_stack	array holding the y co-ordinates of the pixel addresses

	int px	x co-ordinate of pixel address read from the stack
	int py	y co-ordinate of pixel address read from the stack
non-pointer arguments:	none	
called functions:	none	
returns:	none	

description:

This function simulates a stack poping operation using a pair of arrays. On each invocation a variable corresponding to the stack pointer is decremented if it's value is greater than zero (i.e. the stack is not empty) and the px and py arguments are read from the arrays having an index equal to the stack pointer value. If the stack is empty, px and py are returned as illegal values (= -1).

function: pop_d (d_stack,d)

pointer arguments:

	int d	direction variable to be read from the stack
	int d_stack	array array holding the stack elements.
non-pointer arguments:	none	

called functions:	none
returns:	none

description:

This function is identical to the pop() function with the exception that it acts on one array.

function: push (x_stack, y_stack, px, py)

non-pointer arguments:

int x	x co-ordinate of pixel address
int y	y co-ordinate of pixel address

pointer arguments:

int x_stack	array holding x co-ordinates of pixel addresses
int y_stack	array holding y co-ordinates of pixel addresses

returns:	none
called functions:	none

description:

This function simulates a stack pushing operation using a pair of arrays. On each invocation a variable corresponding to the stack pointer is incremented and the arguments of the function (x,y) are written to the elements of the arrays having an index equal to the stack pointer value.

function: pushd (d_stack,d)

non-pointer arguments:

	int d	variable holding direction (UP or DOWN) to be placed on stack.

pointer arguments:

	int d_stack	array holding stack elements

returns:	none
called functions:	none

description:

This function is identical to push() with the exception that it acts only on one array.

function: set_up()

arguments: none

returns: none

called functions: none

description:

This function sets up input LUT 0 on the frame store to give a direct grey scale mapping.

function: thresh (t1,t2)

non-pointer arguments:

	int t1	lower threshold limit
	int t2	upper threshold limit

pointer arguments: none

returns: none

called functions: none

description:

This function sets up input LUT 0 on the frame store for thresholding operations. For each captured frame the original grey level of a pixel, g, is mapped to a new grey level g' according to:

$$g' = \begin{cases} \text{BLACK } (=0), & t_1 > g \\ \text{GREY } (=20), & t_1 \le g \le t_2 \\ \text{BLACK } (=0), & t_2 < g \end{cases} \qquad [A8.1]$$

```
.li on
#include "\lc\stdio.h"
#include "\lc\dos.h"
#include "\lc\math.h"

extern void camera(); /* real time display(i/p buff,ilut,o/p buff,olut) */
extern void capture(); /* real time capture (i/p buff,ilut,o/p buff,olut) */
extern void ilut();  /* i/p look up table (table,index,value) */
extern void olut();  /* o/p look up table (table,index,value) */
extern void mov_buf(); /* move buffer to local buffer (buff) */
extern void res_buf(); /* restore to buffer(buff)*/
extern int read_pix(); /* read (x,y) pixel from buffer (x,y,buff) */
extern int readpixl(); /* read (x,y) pixel from local buffer (x,y) */
extern void wrt_pix(); /* write pixel to buffer (x,y,value,buffer) */
extern void wrtpixl(); /* write pixel to local buffer (x,y,value) */
extern int  disk2_read(); /* read image from disc and threshold between t1 and t2 (t1,t2
)*/
extern void centre_of_area();
extern void fill_hole();
extern void fill_disc();
extern int inner_border_follow();
extern void outer_border_follow();
extern void fit_ellipse();

                    /* various grey scales, for:-   */
#define    WHITE   255          /* inner and outer border       */
#define    WHITE2  254          /* inner ellipse                */
#define    WHITE3  253          /* outer ellipse                */
#define    ALMOST_WHITE 128       /* hole region                */
#define    ALMOST_WHITE2 127      /* disk region ( minus chips ) */
#define    BLACK    0           /* inital disk grey scale       */
#define    GREY    64           /* background grey scale        */
#define    GREY2    1           /* chip regions                 */
#define    TRUE     1
#define    FALSE    0 .
#define    ONE     64
#define    SX      1.33       /* x direction scale factor          */
#define    SY      1.00       /* and y direction factor for aspect ratio */
#define    UP       1
#define    DOWN     0
#define    ILLEGAL -1
#define    ONTOP    1
#define    UNDER    0
#define    CHIP    252        /* chip borders */
#define    CHIP2   251        /* chip colour after filling */
#define    HORIZONTAL 0
#define    VERTICAL  1
#define    DIAGONAL  2


_stack = 32000;

int inner_distance[360];
int outer_distance[360];
int inner_border_x[360];
int inner_border_y[360];
```

```
int outer_border_x[360];
int outer_border_y[360];
int chip_border_x[256];
int chip_border_y[256];

static int stack_pointer2;
static int x_stack2[100];
static int y_stack2[100];
static int sp2;
static int d_stack2[100];



main()
{
int error;
int option;
int x,y;
int count;
int x_bar1,y_bar1,major1,minor1;
int rotation1;
int x_bar2,y_bar2,major2,minor2;
int rotation2;
int px,py;
double sum_x1;
double sum_y1;
double sum_x_squared1;
double sum_y_squared1;
double sum_xy1;
double area1;
double sum_x2;
double sum_y2;
double sum_x_squared2;
double sum_y_squared2;
double sum_xy2;
double area2;
double chip_area;
int  threshold;


    x=y=255;

START: do{
            system("cls");
            set_up();

            printf ("\n          Defect Analyser v3.0 I.C Smith, University of Liver
pool ");
            printf ("\n          ------------------------------------------------
------");
            printf ("\n\n\n\n\n\n");
            printf ("\n\n                          p  process ");
            printf ("\n\n                          x  exit    ");
 option=getch();
            switch (option)
            {
 case 'p':
```

```
                    thresh(0,20);
                    capture(0,0,0,0);
                    system("cls");
                    printf ("              calculating centre of area ");
                    centre_of_area(&x,&y);
                    res_buf(0);
                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n");
                    printf ("                        tracing the border     ")
;
                    error = inner_border_follow(x,y);
                    if (error)
                    {
                       system("cls");
                       printf("\n\n\n\n\n\n\n\n\n\n\n\n");
                       printf("              disk contains crack from outer
to inner edge\n");
                       printf("\n\n\n                   --- press any key t
o continue -- ");
                       getch();
                       goto START;
                    }
                    mov_buf(0);
                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n\n");
                    printf ("                  filling the closed region ");
                    fill_hole(x,y,&sum_x1,&sum_y1,&sum_x_squared1,&sum_y_squared1
,&sum_xy1,&area1);
                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n\n");
                    printf ("                        tracing the border     ")
;
                    outer_border_follow(x,y);
                    mov_buf(0);
                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n\n");
                    printf ("                  filling the closed region ");
                    fill_disk(x,y,&sum_x2,&sum_y2,&sum_x_squared2,&sum_y_squared2
,&sum_xy2,&area2,&chip_area);
                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n\n");
                    printf ("                   calculating ellipse parameters
\n");
                    ellipse_results(sum_x1,sum_y1,sum_x_squared1,sum_y_squared1,s
um_xy1,area1,
                              sum_x2,sum_y2,sum_x_squared2,sum_y_squared2,s
um_xy2,area2,chip_area,
                              &x_bar1,&y_bar1,&minor1,&major1,&rotation1,
                              &x_bar2,&y_bar2,&minor2,&major2,&rotation2 );


                    header();
                    printf("\n\n\n\n\n\n\n\n\n\n\n");
                    printf ("                   calculating inner edge profile
\n");
                    calculate_profile(inner_distance,x_bar1,y_bar1,minor1,major1,
rotation1,
```

```
                        inner_border_x, inner_border_y);
                draw_profile(inner_distance,TRUE);
                header();
                printf("\n\n\n\n\n\n\n\n\n\n");
                printf ("                calculating outer edge profile
\n");
                calculate_profile(outer_distance,x_bar2,y_bar2,minor2,major2,
rotation2,
                        outer_border_x, outer_border_y);
                draw_profile(outer_distance,FALSE);
                chip_fill();
                break;

            }
    } while (option    = 'x');

}
 /* end of main */



set_up()      /* sets up the input LUT to give direct grey scale mapping */
{
int i;

 for (i = 0;i < = 255;i + + )
  ilut(0,i,i);
 camera(0,0,0,0);
}
/* end of setup*/

header()
{
    system("cls");
    printf ("\n        Defect Analyser v3.0, I.C Smith University of Liverpool ");
    printf ("\n        ------------------------------------------------------");

    printf ("\n\n");
}



thresh(t1,t2)  /* alter the input LUT values */
            /* grey level betweem t1 and t2 map to black(0) */
            /* those outside to white */

int t1,t2;
{
 int i;
      capture(0,0,0,0);
 for (i = 0;i < = 255;i + + ){
  if ((i > = t1) && (i < = t2)){
   ilut(0,i,GREY);
  }
  else
```

```
    ilut(0,i,BLACK);
  }
}
/*  end of thresh */







draw_profile(distance,draw_inner)    /* draw the profile using HALO garphics functions *
/
int *distance;
int draw_inner;
{
   long sum;
   int rms;
   int max_deviation;
   int min_deviation;
   int maxm,minm;
   float scale;
   int four=4;
   int i,x,y;
   float ifloat;
   int  mode;
   float x1, x2, y1, y2;
   float  xmax, ymax, xmin, ymin, yzero;
   int colour;
   float dummy1,dummy2;
   static char text[13][10] ={
                                {"0"},  {"20"},  {"40"},  {"60"},  {"80"},
                                {"100"}, {"120"}, {"140"}, {"160"}, {"180"},
                                {"200"}, {"220"}, {"240"}
                              };
   static char anykey[30] = {"PRESS ANY KEY TO CONTINUE"};

     max_deviation = 0;
     min_deviation = 0;
     sum = 0;

     maxm=minm=0;

     for (i=0; i<360; i++ )
     {

       if ( distance[i] > maxm )
         maxm=distance[i];
       if ( distance[i] < minm )
         minm=distance[i];
     }

     setdev("haloibme.dev");
     initgraphics(&four);
```

```
x1 = 0.0  ; x2 = 400.0;  y1 = 0.0; y2 = 120.0;
setworld ( &x1, &y1, &x2, &y2 );
clr();
xmax = 380.0;    xmin = 20.0;  /* histogram bounds */
ymax = 100.0;  ymin = 0.0;




scale = ( ymax - ymin ) / (float)(maxm-minm);
yzero = ymin - (float)minm * scale;

colour = 0;
setcolor(&colour);
bar (&x1,&y1,&x2,&y2);

x = 1;
setlnstyle(&x);
setlnwidth(&x);


colour = 7;
setcolor(&colour);

/* data */

for ( i = 0; i < 360; i++ )
{
   if ( distance[i] > max_deviation )
      max_deviation = distance[i];
   if ( distance[i] < min_deviation )
      min_deviation = distance[i];
   sum += ( (long)distance[i] * (long)distance[i] );

   dummy1 = (float)yzero + (float)distance[i] * scale;
   ifloat = (float)i + xmin;
   movabs(&ifloat,&yzero);
   lnabs(&ifloat,&dummy1);
}


colour = 4;
setcolor(&colour);

/* x axis */

movabs (&xmin,&yzero);
lnabs (&xmax,&yzero);

/*  y axis */

movabs (&xmin,&ymin);
lnabs (&xmin,&ymax);

/* x tick marks */
```

```
    dummy1 = yzero + 5.0;
    dummy2 = yzero-5.0;

    /* x tick marks */

     for ( i=20; i< =380; i+ =20 )
     {
       ifloat = (float)i;
       movabs(&ifloat,&dummy1);
       lnabs(&ifloat,&dummy2);
     }

     /* y tick marks */

    dummy2 = xmin-5;

    for ( i=0; i< =(maxm-minm); i+ + )
    {
      dummy1 = ymin  + (float)i * scale;
      movabs(&xmin,&dummy1);
      lnabs(&dummy2,&dummy1);
    }


    rms = (int) sqrt ( (double) sum / 360.0);

    if ( draw_inner )
      printf ("                    Inner Edge Profile \n");
    else
      printf ("                    Outer Edge Profile \n");

    printf (" each vertical division represents 1 pixel, each horizontal division 20 de
grees\n");

    printf (" maximum deviation  =  + %d/%d pixels, rms deviation  =  %d pixels\n",
            max_deviation, min_deviation, rms );

    printf ("                    press any key to continue");
    getch();
    closegraphics();


} /* end of draw_profile */


get_profile(xbar1,ybar1,minor1,major1,rotation1,
        xbar2,ybar2,minor2,major2,rotation2,distance1,distance2)
int *distance1;
int *distance2;
int xbar1,ybar1;
int major1,minor1;
int rotation1;
int xbar2,ybar2;
int major2,minor2;
int rotation2;
{
int   x,y;
```

```
int    r;
int    theta_deg;
double theta_rad;
int    success;
double value_sin;
double value_cos;
int radius;
float rot_rad1;
float rot_rad2;
double x_component,y_component;
double vector;
double delta_radius;
double norm_radius;
double angle;


      rot_rad1= (double)rotation1 * ( PI / 180.0 );
      rot_rad2= (double)rotation2 * ( PI / 180.0 );

      /* scan radii at 1 degree increments */

      for ( theta_deg = 0; theta_deg < 360; theta_deg++ )
      {

         theta_rad = (double)theta_deg * ( PI / 180.0 );
         value_sin = sin(theta_rad);
         value_cos = cos(theta_rad);

         success=FALSE;

         /* scan radius at 1 pixel increments - check point is  */
         /* actually within the image limits                    */

         x=y=256;

         for ( r = 0; ( r < 512 ) && ( x >= 0 ) && ( y >= 0 ) &&
                   ( x <= 511 ) && ( y <= 511 ) && (success==FALSE); r++ )
         {

             /* polar to rectangular conversion */

               x = xbar1 + (int)((double)r*value_cos);
               y = ybar1 + (int)((double)r*value_sin);

                  if ( readpixl(x,y)ALMOST_WHITE )
                  {
                    success=TRUE;
                    radius=r;
                  }
        }


        norm_radius = (double)radius * sqrt( SX * SX * value_cos * value_cos
                                            + SY * SY * value_sin * value_sin );

        angle = atan2( ( SY * value_sin ) , ( SX * value_cos ) );
```

```
        x_component = (double)major1 * cos( angle - rot_rad1 );
        y_component = (double)minor1 * sin( angle - rot_rad1 );


        vector = sqrt( ( x_component * x_component ) + ( y_component * y_component )
);

        delta_radius = vector - norm_radius;

        distance1[theta_deg] = (int)delta_radius;

    }


    for ( theta_deg = 0; theta_deg < 360; theta_deg++ )
    {

        theta_rad = (double)theta_deg * ( PI / 180.0 );
        value_sin = sin(theta_rad);
        value_cos = cos(theta_rad);

        success=FALSE;

        /* scan radius at 1 pixel increments - check point is  */
        /* actually within the image limits                    */

        x=y=256;

        for ( r = 0; ( r < 512 ) && ( x >= 0 ) && ( y >= 0 ) &&
                ( x <= 511 ) && ( y <= 511 ) && (success==FALSE); r++ )
        {

            /* polar to rectangular conversion */

              x = xbar2 + (int)((double)r*value_cos);
              y = ybar2 + (int)((double)r*value_sin);
                if ( readpixl(x,y) == GREY )
                  success=TRUE;
        }


        norm_radius = (double)r * sqrt( SX * SX * value_cos * value_cos
                                      + SY * SY * value_sin * value_sin );

        angle = atan2( ( SY * value_sin ) , ( SX * value_cos ) );

        x_component = (double)major2 * cos( angle - rot_rad2 );
        y_component = (double)minor2 * sin( angle - rot_rad2 );


        vector = sqrt( ( x_component * x_component ) + ( y_component * y_component )
);

        delta_radius = vector - norm_radius;

        distance2[theta_deg] = (int)delta_radius;
```

```
    }

} /* end of inner profile */


/* print out ellipse results */

ellipse_results(sum_x1,sum_y1,sum_x_squared1,sum_y_squared1,sum_xy1,area1,
        sum_x2,sum_y2,sum_x_squared2,sum_y_squared2,sum_xy2,area2,chip_area,
        x_bar1,y_bar1,minor1,major1,rotation1,
        x_bar2,y_bar2,minor2,major2,rotation2 )


double sum_x1,sum_y1,sum_x_squared1,sum_y_squared1,sum_xy1,area1;
double sum_x2,sum_y2,sum_x_squared2,sum_y_squared2,sum_xy2,area2,chip_area;
int *x_bar1,*y_bar1,*minor1,*major1,*rotation1;
int *x_bar2,*y_bar2,*minor2,*major2,*rotation2;
{
double total_area;
int offset;
double ratio;

    sum_x2 += sum_x1;
    sum_y2 += sum_y1;
    sum_x_squared2 += sum_x_squared1;
    sum_y_squared2 += sum_y_squared1;
    sum_xy2 += sum_xy1;
    total_area = area1 + area2;

    header();
    printf("inner edge parameters:- \n");
    fit_ellipse( sum_x1, sum_y1, sum_x_squared1, sum_y_squared1, sum_xy1, area1,
           x_bar1, y_bar1, minor1, major1, rotation1);
    printf("\n outer edge parameters \n");
    fit_ellipse( sum_x2, sum_y2, sum_x_squared2, sum_y_squared2, sum_xy2, total_area,

           x_bar2, y_bar2, minor2, major2, rotation2);

    offset = (int)sqrt ( (double)( ( *x_bar2 - *x_bar1 ) * ( *x_bar2 - *x_bar1 ) +
                      ( *y_bar2 - *y_bar1 ) * ( *y_bar2 - *y_bar1 ) ) );

    ratio = 100.0 * chip_area / area2;

   printf("\n\n");
   printf("\t offset between centres = %d \n",offset);
    printf("\t total chip area = %d = %3.1f %% of disc area \n",(int)chip_area,(float
)ratio);

   printf ("\n\n");
   printf ("     all dimensions in pixels normalised w.r.t aspect ratio \n");
```

```
     printf("                   press any key to continue ");
     getch();

}


chip_fill()
{
int x,y;
int area;
int norm_area;
int length;
int noofpoints;
int noofchips;
int xbar,ybar;
int error;
int x1,y1;
int x2;
int ratio;
int x_bar,y_bar;
int i;
int max_radius;
int min_radius;
int radius;
int nborder;
int rad_ratio;
   noofchips = 0;

   header();
   printf ("chip parameters :-     \n\n");
   printf ("area\tperimeter\tP*P/A\tposition\tmax r\tmin r\t max / min \n");
   printf ("----\t---------\t-----\t--------\t-----\t-----\t----------- \n");


   mov_buf(0);

   for (y = 2; y < = 509; y + + )
   {
     x = 2;
     while( x < = 509)
     {
       /* find an interior point */

       if ( ( readpixl(x,y) = = GREY2 ) && ( readpixl(x + 1,y) = = GREY2  ) &&
            ( readpixl(x-1,y) = = GREY2 )   && ( readpixl(x,y-1) = = GREY2 )   &&
            ( readpixl(x-1,y-1) = = GREY2 ) && ( readpixl(x + 1,y-1) = = GREY2 ) &&
            ( readpixl(x,y + 1) = = GREY2 )   && ( readpixl(x + 1,y + 1) = = GREY2 ) &&
            ( readpixl(x-1,y + 1) = = GREY2 ) && ( readpixl(x,y + 2) = = GREY2 ) &&
            ( readpixl(x-1,y + 2) = = GREY2 ) && ( readpixl(x + 1,y + 2) = = GREY2 )
          )
       {
         /* take this point as the seed point */

         x1 = x;
         y1 = y;
```

```
/* find a boundary point */

while ( readpixl(x,y) = = GREY2 )
  x+ +;
x--;


/* trace the boundary */
chip_border_follow(x,y,&noofpoints,&length,&xbar,&ybar,&nborder);
mov_buf(0);


/* check for false perimeter tracing */


   for ( x2=x1; ( x2 > 0 ) && ( readpixl(x2,y) = = GREY2 );  x2-- );
   if ( readpixl(x2,y) = = ALMOST_WHITE2 )
      error = TRUE;
   else
      error = FALSE;




if (                  error )
{
   noofchips+ +;
   fill_chips(x1,y1,&area,&x_bar,&y_bar);
   norm_area = (int)( (float)(area + noofpoints) * SX * SY );
   ratio = (int)((long)length * (long)length / (long)norm_area);



   max_radius = 0;
   min_radius = 32000;

   for ( i = 0; i<nborder; i+ + )
   {
      radius = (int) sqrt ( (double)( ( chip_border_x[i] - x_bar ) * (
chip_border_x[i] - x_bar ) ) * (SX*SX)
                              +(double)( ( chip_border_y[i] - y_bar ) * (
chip_border_y[i] - y_bar ) ) * (SY*SY) );


      if ( radius > max_radius )
         max_radius = radius;
      if ( radius < min_radius )
         min_radius = radius;
   }
   rad_ratio = max_radius / min_radius;

   printf ("%d\t%d\t\t%d\t%d,%d\t\t%d\t%d\t%d\n",norm_area,length,ratio,
x_bar,y_bar,max_radius,min_radius,rad_ratio);
   if ( noofchips % 10 = = 0)
   {
```

```
                printf ("\n\n");
                printf ("          all dimensions in pixels normalised w.r.t aspec
t ratio \n");
                printf ("                        press any key to continue
      \n");
                printf ("\n                                              conti
nued... \n");
                getch();
                header();
                printf ("chip parameters :-      \n\n");
                printf ("area\tperimeter\tP*P/A\tposition\tmax r\tmin r\t max / mi
n \n");
                printf ("----\t---------\t-----\t--------\t-----\t-----\t---------
-- \n");
              }
            }
        }
        x++;
       } /* end while */
    } /* end for */


    printf ("\n\n");
    printf ("          all dimensions in pixels normalised w.r.t aspect ratio \n");
    printf ("                        press any key to continue                \n");
    getch();


}



chip_border_follow(x_bar,y_bar,npoints,length,xbar2,ybar2,nborder)
                            /* traces the chip borders in the disc */
int x_bar,y_bar;
int *npoints,*length;
int *xbar2,*ybar2;
int *nborder;
{
int cx,cy;
int dx,dy;
int x,i,k;
int index;
int ex[8];
int ey[8];
int success;
static int x_offset[16] = { 0, 1, 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1 };
static int y_offset[16] = { 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1, 0, 1 };
int c;
int finished;
int nblack;
int nhorizontal;
int nvertical;
int ncrossing;
int cx_last;
int cy_last;
int found_seed;
```

```
int direction;
int previous_direction;
int points;


      points = 0;
      direction = previous_direction = HORIZONTAL;

      nhorizontal = 0;
      nvertical = 0;
      ncrossing = 0;

      cx_last = 0;
      cy_last = 0;


      cx  =  x_bar;
      cy  =  y_bar;

      dx  =  x_bar + 1;
      dy  =  y_bar;

      /* set c to 3 */

      wrtpixl(cx,cy,3);

      /* set d to 2 */

      wrtpixl(dx,dy,2);


      /* get direction of d */


  finished  =  FALSE;
  do
  {
     success = FALSE;
     for ( i = 0; (i < = 7) && (success = = FALSE); i + + )
       if ( (dx-cx  = =  x_offset[i]) && ( dy-cy  = =  y_offset[i]) )
       {
         index  =  i;
         success = TRUE;
       }

     /* assign e(i) */

    success = FALSE;
    for ( i = 0; (i < = 7) && (success = = FALSE); i + + )
    {
      ex[i]  =  cx  +  x_offset[index];
      ey[i]  =  cy  +  y_offset[index];

      if (  ( readpixl( ex[i],ey[i] )  = =  GREY2 ) ||
            ( readpixl( ex[i],ey[i] )  = =  3   ) ||
            ( readpixl( ex[i],ey[i] )  = =  4   ) )
```

```
        success = TRUE;
    else
        index + + ;
}

 k = i-1;


if ( ( (c = readpixl(cx,cy)) = = 3 ) && (readpixl(ex[k],ey[k]) = = 4) )
   for ( i = 0; (i  <  k) && (finished = = FALSE); i + +  )
      if ( readpixl(ex[i],ey[i])  = =  2 )
      {
          wrtpixl( cx, cy, 4 );
          wrtpixl( ex[i], ey[i], 0 );
          finished  =  TRUE;
      }

if(finished = = FALSE)
{
   if ( c = =  GREY2 )
      wrtpixl(cx,cy,4);


      wrt_pix(cx,cy,CHIP,0);
      chip_border_x [points]  =  cx;
      chip_border_y [points]  =  cy;
      points + + ;

   if ( cx_last             =  0 )
   {
      previous_direction = direction;
      if ( cy  = =  cy_last )
         direction = HORIZONTAL;
      else
      {
         if ( cx  = =  cx_last )
            direction = VERTICAL;
         else
            direction = DIAGONAL;
      }

      if ( direction  = =  DIAGONAL )
      {
         ncrossing + + ;
         nhorizontal + + ;
         nvertical + + ;
      }
      else
         if ( direction    =  previous_direction )
             ncrossing + + ;

      if ( direction  = =  HORIZONTAL )
          nhorizontal + + ;
      if ( direction  = =  VERTICAL )
          nvertical + + ;

   }
```

```
            cx_last = cx;
            cy_last = cy;

             cx  =  ex[k];
             cy  =  ey[k];

             dx  =  ex[k-1];
             dy  =  ey[k-1];


        }


      } while ( finished = =  FALSE );


      *nborder =   points;
      *npoints  =  ncrossing  +  nvertical  +  nhorizontal;
      *length  =  (int) (0.984059 * ( SX * (float)nhorizontal +  SY * (float)nvertical -
                                 0.5 * ( SX  +  SY - sqrt ( SX*SX  +  SY*SY ) ) *
                                 (float)ncrossing ) );



}
 /* end of border follow */





left2(x,y)
int x;
int y;
{
int i;
       if ( readpixl(x,y)          =  CHIP )
       {
          for ( i = x; (i > = 0)&&[illegible]pixl(i)y)
          return(i + 1);
       }
       else
       {
          for ( i = x; (i > = 0)&&(readpixl(i,y) = = CHIP); i--);
          return(i + 1);
       }

}


lright2(x,y)
int x;
int y;
{
int i;
int dummy;
```

```
        return( left2( left2(x,y)-1, y ) - 1 );

}


/*                      fill chips()

fills in the hole region of the disk - pixels forming part of the region
are set to CHIP2 - border pixels are taken as being CHIP

inputs:-

xbar,ybar  - co-ords of centre of area of entire image, used as a seed
        point by the filling algorithm

outputs (used for calculating ellipse parameters):-

sum_x
sum_y
sum_x_squared
sum_y_squared
sum_xy
area

*/

fill_chips(xbar,ybar,area,x_centre,y_centre)
int xbar,ybar;
int *x_centre;
int *y_centre;
int *area;
{
int px_right,py_right;
int px,py;
int other;
int u;
int dir;
int above,below;
int e1,e2;
int x;
int px_next,py_next;
int px1,py1;
double xs;
double xf;
double ys;
long x_sum,y_sum;

    mov_buf(0);
    stack_pointer2=sp2=0;
    *area=0;
    x_sum = 0;
    y_sum = 0;


    push2(x_stack2,y_stack2,left2(xbar,ybar),ybar);
    push2d(d_stack2,DOWN);
```

```
push2(x_stack2,y_stack2,left2(xbar,ybar + 1),ybar + 1);
push2d(d_stack2,UP);
px = xbar;
py = ybar;

while(stack_pointer2 > 0)
{
  pop2(x_stack2,y_stack2,&px_next,&py_next);
  pop2d(d_stack2,&dir);
  px_right = 511;
  py_right = py;

  if ( dir = = DOWN )
  {
    other = UNDER;
    u = 2;
  }
  else
  {
    other = ONTOP;
    u = 1;
  }

  while(1)
  {
    if ( ( px_next = = ILLEGAL ) ||
        ( readpixl(px_next,py_next) = = CHIP2 ) )
    goto FINI2;

    px = px_next;
    py = py_next;


    link2(px-1,py,&above,&below,&e1,&e2,&px1);
    py1 = py;


    if ( ( ( dir = = DOWN ) && ( above > 1 ) ||
        ( dir = = UP   ) && ( below > 1 )   ) && ( px > px_right ))
      goto FINI2;

    if ( ( above > 0 ) && ( below > 0 ) )
    {
      if ( u = = 1 )
      {
        px_next = e1;
        py_next = py + 1;
      }
      else
      {
        px_next = e2;
        py_next = py-1;
      }
    }
    else
    {
      if ( readpixl(px,py) CHIP2 )
```

```
{
     push2(x_stack2,y_stack2, left2( lright2(px,py),py ) , py );
     push2d(d_stack2,dir);
}
if ( ( above = =(0))&&&&((below = DOWN ) ||
       ( above   = 0 ) && ( below = = 0 ) && ( dir = = UP  ) )
 {
    if ( u = = 1 )
    {
     px_next = e1;
     py_next =  py + 1;
    }
    else
    {
     px_next = e2;
     py_next = py-1;
    }
 }
 else
   if ( ( other== (CHIPTOP|) && ( readpixl(px,py + 1)

       ( other== (CHIPUNDER) ) && ( readpixl(px,py-1)
   {
     if ( ( other==CHIPSTOP ) && ( readpixl(px,py + 1)
     {
        px_next = left2(px,py + 1);
        py_next =  py + 1;
     }
     else
      if ( ( other==CHIPUNDER ) && ( readpixl(px,py-1)
      {
         px_next = left2(px,py-1);
         py_next = py-1;
      }
   }
   else
     px_next = ILLEGAL;
}
for ( x = px; ( x < 512 )CHIP()readpixl(x,py)
  px_right = x-1;
  py_right = py;
for ( x = px; x < = px_right; x+ + )

   wrtpixl(x,py_right,CHIP2);

*area + = (px_right-px + 1);
x_sum  + =  (long)(px_right  +  px ) * (long)( px_right - px  +  1 );
y_sum  + =  (long)py_right * (long)( px_right - px  +  1 );



link2 (px_right + 1,py_right,&above,&below,&e1,&e2,&px1);
py1 = py_right;


if ( ( ( above = = 0 ) || ( below = = 0 ) ) &&
    ( readpixl(px1,py1)CHIP2 ) )
```

```
          {
            push2(x_stack2,y_stack2,px1,py1);
            if ( above > 0 )
              push2d(d_stack2,UP);
            else
              push2d(d_stack2,DOWN);
          }
        }
FINI2: ;
    }
    res_buf(0);

    *x_centre = x_sum / ( 2 * *area );
    *y_centre = y_sum / ( *area );



} /* end of fill_chips */



link2(x,y,above,below,e1,e2,p1)
int x,y;
int *above,*below;
int *e1,*e2,*p1;
{
int dx,dy;
int x1,x2;
int i;

     x=left2(x,y);

    *above=*below=0;
    dx=dy=1;

    x1=x2=ILLEGAL;
    *e1=*e2=ILLEGAL;

    if ( readpixl( x-dx,y+dy ) = = CHIP )
    {
      (*above)+ +;
      x1=x-dx;
    }
    if ( readpixl( x-dx,y-dy ) = = CHIP )
    {
      (*below)+ +;
      x2=x-dx;
    }


    while ( readpixl(x,y) = = CHIP )
    {


    if ( ( readpixl( x,y+dy ) = = CHIP )
       &&( readpixl( x-dx,y+dy ) CHIP ) )
      (*above)+ +;
```

```
    if ( ( readpixl( x,y-dy ) = = CHIP )
       &&( readpixl( x-1,y-dy != )CHIP ) )
      (*below) + + ;
     if ( readpixl(x,y + dy) = = CHIP )
       x1 = x;
     if ( readpixl(x,y-dy) = = CHIP )
       x2 = x;
     x + + ;
    }
    *p1 = x;

    if ( readpixl(x,y + dy) = = CHIP )
       x1 = x;
    if ( readpixl(x,y-dy) = = CHIP )
       x2 = x;

    if ( x1                    = ILLEGAL )
    {
       for ( i = x1; (i < 512 ) && ( readpixl(i,y + dy) = = CHIP ); i + + );
       *e1 = i;
    }

    if ( x2                    = ILLEGAL )
    {
      for ( i = x2; (i < 512 ) && ( readpixl(i,y-dy) = = CHIP ); i + + );
      *e2 = i;
    }

    if ( ( readpixl( x-dx,y + dy != )CHIP )
       &&( readpixl( x,y + dy ) = = CHIP ) )
      (*above) + + ;

    if ( ( readpixl( x-dx,y-dy != CHIP )
       &&( readpixl( x,y-dy ) = = CHIP ) )
      (*below) + + ;

}

push2(x_stack2,y_stack2,px,py)
int px,py;
int *x_stack2,*y_stack2;
{
int i;

   stack_pointer2 + + ;
   *(x_stack2 + stack_pointer2) = px;
   *(y_stack2 + stack_pointer2) = py;

}

pop2(x_stack2,y_stack2,px,py)
int *px,*py;
int *x_stack2,*y_stack2;
{
  if (stack_pointer2 > 0)
  {
    stack_pointer2--;
```

```
        *px = *(x_stack2 + stack_pointer2 + 1);
        *py = *(y_stack2 + stack_pointer2 + 1);
    }
    else
      *px = *py = -1;
}



push2d(d_stack2,d)
int d;
int *d_stack2;
{
    sp2++;
    *(d_stack2 + sp2) = d;
}

pop2d(d_stack2,d)
int *d,*d_stack2;
{
   if (sp2 > 0)
   {
      sp2--;
      *d = (*(d_stack2 + sp2 + 1));
   }
   else
     *d = -1;
}



calculate_profile(distance,xbar,ybar,minor,major,rotation,border_x,border_y)

int *distance;
int xbar,ybar;
int major,minor;
int rotation;
int *border_x;
int *border_y;
{
int    x,y;
int    theta_deg;
double theta_rad;
double value_sin;
double value_cos;
double radius;
float rot_rad;
double x_component,y_component;
double vector;
double delta_radius;
double norm_radius;
double angle;
```

```
rot_rad = (double)rotation * ( PI / 180.0 );

for ( theta_deg=0; theta_deg < 360; theta_deg++ )
{

  theta_rad = (PI/180.0) * theta_deg;

  value_cos = cos(theta_rad);
  value_sin = sin(theta_rad);

  /* polar to rectangular conversion */

  x = border_x[theta_deg];
  y = border_y[theta_deg];

  radius = sqrt( (float)(x - xbar)*(float)(x-xbar) +
                 (float)(y - ybar)*(float)(y - ybar) );

  norm_radius = radius * sqrt( SX * SX * value_cos * value_cos
                      + SY * SY * value_sin * value_sin );

  angle = atan2( ( SY * value_sin ) , ( SX * value_cos ) );

  x_component = (double)major * cos( angle - rot_rad );
  y_component = (double)minor * sin( angle - rot_rad );

  vector = sqrt( ( x_component * x_component ) + ( y_component * y_component ) )
;

  delta_radius = vector - norm_radius;

  distance[theta_deg] = (int)delta_radius;

}

} /* end of inner profile */
```

```
#include "\lc\stdio.h"
#include "\lc\dos.h"
#include "\lc\math.h"

extern void camera(); /* real time display(i/p buff,ilut,o/p buff,olut) */
extern void capture(); /* real time capture (i/p buff,ilut,o/p buff,olut) */
extern void ilut();  /* i/p look up table (table,index,value) */
extern void olut();  /* o/p look up table (table,index,value) */
extern void mov_buf(); /* move buffer to local buffer (buff) */
extern void res_buf(); /* restore to buffer(buff)*/
extern int read_pix(); /* read (x,y) pixel from buffer (x,y,buff) */
extern int readpixl(); /* read (x,y) pixel from local buffer (x,y) */
extern void wrt_pix(); /* write pixel to buffer (x,y,value,buffer) */
extern void wrtpixl(); /* write pixel to local buffer (x,y,value) */
extern void disk2_read(); /* read image from disc and threshold between t1 and t2 (t1,t2
)*/

                /* various grey scales, for:-   */
#define   WHITE   255        /* inner and outer border      */
#define   WHITE2  254        /* inner ellipse            */
#define   WHITE3  253        /* outer ellipse            */
#define   ALMOST_WHITE 128      /* hole region              */
#define   ALMOST_WHITE2 127     /* disk region ( minus chips )  */
#define   BLACK   0          /* inital disk grey scale     */
#define   GREY    64         /* background grey scale      */
#define   GREY2   1          /* chip regions             */
#define   TRUE    1
#define   FALSE   0
#define   ONE     64
#define   SX      1.33      /* x direction scale factor          */
#define   SY      1.00      /* and y direction factor for aspect ratio */
#define   UP      1
#define   DOWN    0
#define   ILLEGAL  -1
#define   ONTOP   1
#define   UNDER   0


static int stack_pointer;
static int x_stack[100];
static int y_stack[100];
static int sp;
static int d_stack[100];




centre_of_area(xbar,ybar)       /* calculates  the C of A for an entire */
int *xbar,*ybar;              /* thresholded frame, coords of centre  */
{                 /* are returned as (xbar,ybar)        */
long  area;
register  x,y;
long   sumx;
long   sumy;
```

```
/* calculate area and centre of area */

        mov_buf(0);

        area = 0;
        sumx = 0;
        sumy = 0;

        for ( x = 0; x < = 512; x+ + )
          for ( y= 0; y < = 512; y+ + )
            if ( readpixl(x,y) = = BLACK )
            {
                area+ +;
                sumx + = x;
                sumy + = y;
            }

        *xbar = sumx / area;
        *ybar = sumy / area;


        /* plot centre on the screen */


        for ( x=0; x< =511; x+ +)
         wrt_pix(x,*ybar,WHITE,0);


        for ( y=0; y< =511; y+ +)
         wrt_pix(*xbar,y,WHITE,0);

} /* end of centre_of_area */



inner_border_follow(x_bar,y_bar) /* traces the inner border of the disc */
int x_bar,y_bar;
{
int cx,cy;
int dx,dy;
int x,i,k;
int index;
int ex[8];
int ey[8];
int success;
static int x_offset[16] = { 0, 1, 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1 };
static int y_offset[16] = { 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1, 0, 1 };
int c;
int finished;
int nblack;
int angle;
int radius;
int distance[360];
```

```
extern int inner_border_x[];
extern int inner_border_y[];
long sum_distance;

      for ( angle=0; angle < 360; angle++)
         distance[angle] =0;


      /* find first border point */
      /* - first one with 5 black pixels */

      success=FALSE;
      nblack=0;

      for ( x = x_bar; ( x <= 511 ) && (success==FALSE); x++)
      {
         if ( (readpixl(x,y_bar) == BLACK) )
            nblack++;
         else
            nblack=0;

         if ( nblack== 5)
            success = TRUE;
      }

         x-=5;

         cx = x-1;
         cy = y_bar;

         dx = x;
         dy = y_bar;

         /* set c to 3 */

         wrtpixl(cx,cy,3);

         /* set d to 2 */

         wrtpixl(dx,dy,2);


         /* get direction of d */


    finished = FALSE;
    do
    {
        success=FALSE;
        for ( i=0; (i<=7) && (success==FALSE); i++ )
           if ( (dx-cx == x_offset[i]) && ( dy-cy == y_offset[i]) )
           {
              index = i;
              success=TRUE;
           }
```

```
/* assign e(i) */

success=FALSE;
for ( i=0; (i<=7) && (success==FALSE); i++ )
{
  ex[i] = cx + x_offset[index];
  ey[i] = cy + y_offset[index];

  if( ( readpixl( ex[i],ey[i] ) == ONE ) ||
      ( readpixl( ex[i],ey[i] ) == 3  ) ||
      ( readpixl( ex[i],ey[i] ) == 4  ) )
     success=TRUE;
  else
     index++;
}

k=i-1;


if ( ( (c=readpixl(cx,cy))==3 ) && (readpixl(ex[k],ey[k])==4) )
  for ( i=0; (i < k) && (finished==FALSE); i++ )
    if ( readpixl(ex[i],ey[i]) == 2 )
    {
        wrtpixl( cx, cy, 4 );
        wrtpixl( ex[i], ey[i], 0 );
        finished = TRUE;
    }

if(finished==FALSE)
{
  if ( c== ONE )
    wrtpixl(cx,cy,4);
  wrt_pix(cx,cy,WHITE,0);

  radius=(int)sqrt ( (float)(cx-x_bar)*(float)(cx-x_bar)*(SX*SX) +
                     (float)(cy-y_bar)*(float)(cy-y_bar)*(SY*SY) );

  angle=(int)( (180.0/PI) *
       (PI + atan2((float)(cy-y_bar)/SY,(float)(cx-x_bar)/SX )));


  if ( radius > distance[angle] )
  {
    inner_border_x[angle] = cx;
    inner_border_y[angle] = cy;
    distance[angle] = radius;
  }
  cx = ex[k];
  cy = ey[k];

  dx = ex[k-1];
  dy = ey[k-1];


}
```

```
    } while ( finished == FALSE );

    for ( angle=1; angle<359; angle++ )
       if ( distance[angle] == 0 )
       {
          inner_border_x[angle] = (inner_border_x[angle+1]+inner_border_x[angle-1] )
/ 2;
          inner_border_y[angle] = (inner_border_y[angle+1]+inner_border_y[angle-1] )
/ 2;
       }

   if ( distance[0] == 0 )
   {
      inner_border_x[0] = (inner_border_x[1]+inner_border_x[359]) / 2;
      inner_border_y[0] = (inner_border_y[1]+inner_border_y[259] ) / 2;
   }

   if ( distance[359] == 0 )
   {
      inner_border_x[359] = (inner_border_x[0]+inner_border_x[358]) / 2;
      inner_border_y[359] = (inner_border_y[0]+inner_border_y[358] ) / 2;
   }


   sum_distance = 0;
   for ( angle=0; angle <360; angle++ )
     sum_distance += distance[angle];

   if ( sum_distance / 360 > 160 )
     return(TRUE);
   else
     return(FALSE);
} /* end of border follow */


outer_border_follow(x_bar,y_bar) /* traces the outer border of the disc */
int x_bar,y_bar;
{
int cx,cy;
int dx,dy;
int x,i,k;
int index;
int ex[8];
int ey[8];
int success;
static int x_offset[16] = { 0, 1, 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1 };
static int y_offset[16] = { 1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1, 0, 1 };
int c;
int finished;
int nblack;
extern int outer_border_x[];
extern int outer_border_y[];
int radius;
int angle;
int distance[360];
```

```
    for ( angle = 0; angle < 360; angle + + )
      distance[angle] = 32000;


  /* find first border point */
  /* - first one with 5 black pixels */

  success = FALSE;
  nblack = 0;

  for ( x = 0; ( x < = x_bar ) && (success = = FALSE); x + + )
  {
    if ( (readpixl(x,y_bar) = = BLACK) )
      nblack + + ;
    else
      nblack = 0;

    if ( nblack = = 5)
      success = TRUE;
  }

    x- = 5;

    cx = x;
    cy = y_bar;

    dx = x-1;
    dy = y_bar;

    /* set c to 3 */

    wrtpixl(cx,cy,3);

    /* set d to 2 */

    wrtpixl(dx,dy,2);


    /* get direction of d */


finished = FALSE;
do
{
    success = FALSE;
    for ( i = 0; (i < = 7) && (success = = FALSE); i + + )
      if ( (dx-cx = = x_offset[i]) && ( dy-cy = = y_offset[i]) )
      {
        index = i;
        success = TRUE;
      }

    /* assign e(i) */
```

```
success=FALSE;
for ( i=0; (i< =7) && (success= =FALSE); i+ + )
{
   ex[i] = cx + x_offset[index];
   ey[i] = cy + y_offset[index];

   if (  ( readpixl( ex[i],ey[i] ) = =  BLACK) ||
         ( readpixl( ex[i],ey[i] ) = =  3   ) ||
         ( readpixl( ex[i],ey[i] ) = =  4   ) )
      success=TRUE;
   else
      index+ +;
}

k=i-1;


if ( ( (c=readpixl(cx,cy))= =3 ) && (readpixl(ex[k],ey[k])= =4) )
  for ( i=0; (i  <  k) && (finished= =FALSE); i+ +  )
    if ( readpixl(ex[i],ey[i]) = =  2 )
    {
        wrtpixl( cx, cy, 4 );
        wrtpixl( ex[i], ey[i], 0 );
        finished  =  TRUE;
    }

if(finished= =FALSE)
{
   if ( c= =  BLACK )
      wrtpixl(cx,cy,4);
     wrt_pix(cx,cy,WHITE,0);


   radius=(int)sqrt ( (float)(cx-x_bar)*(float)(cx-x_bar)/(SX*SX) +
                     (float)(cy-y_bar)*(float)(cy-y_bar)/(SY*SY) );
   angle=(int)( (180.0/PI) *
        (PI + atan2((float)(cy-y_bar)/SY,(float)(cx-x_bar)/SX )));


   if ( radius  <  distance[angle] )
   {
      outer_border_x[angle]  =  cx;
      outer_border_y[angle]  =  cy;
      distance[angle]  =  radius;
   }


for ( angle=1; angle<359; angle+ +  )
   if ( distance[angle]  = =  32000 )
   {
     outer_border_x[angle]  =  (outer_border_x[angle+1]+outer_border_x[angle-1] )
/ 2;
     outer_border_y[angle]  =  (outer_border_y[angle+1]+outer_border_y[angle-1] )
/ 2;
   }
```

```
if ( distance[0] = = 32000 )
{
   outer_border_x[0] = (outer_border_x[1]+outer_border_x[359]) / 2;
   outer_border_y[0] = (outer_border_y[1]+outer_border_y[259] ) / 2;
}

if ( distance[359] = = 32000 )
{
   outer_border_x[359] = (outer_border_x[0]+outer_border_x[358]) / 2;
   outer_border_y[359] = (outer_border_y[0]+outer_border_y[358] ) / 2;
}



         cx = ex[k];
         cy = ey[k];

         dx = ex[k-1];
         dy = ey[k-1];


      }


    } while ( finished = = FALSE );

} /* end of border follow */


/*             fit ellipse

fits an ellipse given the region parameters

inputs ( region parameters ):

sum_x
sum_y
sum_x_squared
sum_y_squared
sum_xy
area

outputs ( ellipse parameters )

x_bar      co-ords of the centre
y_bar
minor      minor axis length |  normalised w.r.t
major      major axis length |      aspect ratio
rotation   major axis rotation from horizontal

*/

fit_ellipse( sum_x, sum_y, sum_x_squared, sum_y_squared, sum_xy, area,
          x_bar, y_bar, minor, major, rotation)
double   sum_x;
```

```
double  sum_y;
double  sum_x_squared;
double  sum_y_squared;
double  sum_xy;
double  area;
int *x_bar;
int *y_bar;
int *minor;
int *major;
int *rotation;
{
double a,b,c,e,f;
double theta_rad;
double rot_rad;
int theta_deg;
long norm_area;
int x,y;

/* calculate area and centre of area */


        norm_area = (long)(SX*SY * area);

        *x_bar =  sum_x / area;
        *y_bar =  sum_y / area;


        a =  (4.0/PI) * SX*SX*SX*SY * ( sum_x_squared- sum_x * sum_x / (double)area);

        b =  (4.0/PI) * SX*SY*SY*SY * ( sum_y_squared - sum_y * sum_y / (double)area);

        c =  (4.0/PI) * SX*SX*SY*SY * ( sum_xy - sum_x * sum_y / (double)area);

        e =  sqrt ( (a-b)*(a-b) + (4.0/PI)*c*c );

        f =  sqrt ( sqrt ( a*b - c*c ) );


        *major = (int)(sqrt ( ( a + b + e ) / (2*f) ));
        *minor = (int)(sqrt ( ( a + b - e ) / (2*f) ));

        *rotation = (int)( (90.0/PI) * atan( 2*c/(a-b) ) );

        rot_rad = *rotation * PI /180 ;


        /* plot it out */


        for ( theta_deg = 0; theta_deg < 360; theta_deg++ )
        {
            theta_rad = theta_deg * PI / 180;
            x= *x_bar + (*major/SX) * cos(theta_rad-rot_rad);
            y= *y_bar + (*minor/SY) * sin(theta_rad-rot_rad);
            wrt_pix ( x, y, WHITE2, 0 );
        }
```

```
    printf("\t centre at %d,%d \n",*x_bar,*y_bar);
    printf("\t major axis length  =  %d \n",*major);
    printf("\t minor axis length  =  %d \n",*minor);
    printf("\t major axis rotation  =  %d degress \n",*rotation);


 }  /* end of ellipse */




lcft(x,y)
int x;
int y;
{
int i;
        if ( readpixl(x,y)          = WHITE )
        {
          for ( i = x; (i > = 0)&&(readpixl(i,y)[illegible]);
          return(i + 1);
        }
        else
        {
          for ( i = x; (i > = 0)&&(readpixl(i,y) = = WHITE); i--);
          return(i + 1);
        }

}


lright(x,y)
int x;
int y;
{
int i;
int dummy;

        return( left( left(x,y)-1, y ) - 1 );

}


/*                         fill hole()

fills in the hole region of the disk - pixels forming part of the region
are set to ALMOST_WHITE - border pixels are taken as being WHITE

inputs:-

xbar,ybar  - co-ords of centre of area of entire image, used as a seed
             point by the filling algorithm

outputs (used for calculating ellipse parameters):-
```

```
sum_x
sum_y
sum_x_squared
sum_y_squared
sum_xy
area

*/

fill_hole(xbar,ybar,sum_x,sum_y,sum_x_squared,sum_y_squared,sum_xy,area)
int xbar,ybar;
double *sum_x,*sum_y;
double *sum_x_squared,*sum_y_squared;
double *sum_xy;
double *area;
{
int px_right,py_right;
int px,py;
int other;
int u;
int dir;
int above,below;
int e1,e2;
int x;
int px_next,py_next;
int px1,py1;
double xs;
double xf;
double ys;

    mov_buf(0);
    stack_pointer = sp = 0;

    *sum_x = 0.0;
    *sum_y = 0.0;
    *sum_x_squared = 0.0;
    *sum_y_squared = 0.0;
    *sum_xy = 0.0;
    *area = 0.0;

    push(x_stack,y_stack,left(xbar,ybar),ybar);
    pushd(d_stack,DOWN);
    push(x_stack,y_stack,left(xbar,ybar + 1),ybar + 1);
    pushd(d_stack,UP);
    px = xbar;
    py = ybar;

    while(stack_pointer > 0)
    {
      pop(x_stack,y_stack,&px_next,&py_next);
      popd(d_stack,&dir);
      px_right = 511;
      py_right = py;

      if ( dir == DOWN )
      {
        other = UNDER;
```

```
      u = 2;
    }
    else
    {
      other = ONTOP;
      u = 1;
    }

    while(1)
    {
      if ( ( px_next = = ILLEGAL ) ||
           ( readpixl(px_next,py_next) = = ALMOST_WHITE ) )
      goto FINI;

      px = px_next;
      py = py_next;


      link(px-1,py,&above,&below,&e1,&e2,&px1);
      py1 = py;


      if ( ( ( dir = = DOWN ) && ( above > 1 ) ||
             ( dir = = UP   ) && ( below > 1 )    ) && ( px > px_right ))
        goto FINI;

      if ( ( above > 0 ) && ( below > 0 ) )
      {
         if ( u = = 1 )
         {
           px_next = e1;
           py_next = py + 1;
         }
         else
         {
           px_next = e2;
           py_next = py-1;
         }
      }
      else
      {
         if ( readpixl(px,py) ALMOST_WHITE )
         {
              push(x_stack,y_stack, left( lright(px,py),py ) , py );
              pushd(d_stack,dir);
         }
         if (  ( above = = 00))&&&&((bdlow = = DOWN )  ||
                ( above   = 0 ) && ( below = = 0 ) && ( dir = = UP   )  )
           {
              if ( u = = 1 )
              {
                px_next = e1;
                py_next = py + 1;
              }
              else
              {
                px_next = e2;
```

```
            py_next = py-1;
          }
        }
        else
          if ( ( other == ALMOST_WHITE ) && ( readpixl(px,py + 1)

             ( other == ALMOST_WHITE ) && ( readpixl(px,py-1)
          {
            if ( ( other == ALMOST_WHITE ) && ( readpixl(px,py + 1)
)
            {
              px_next = left(px,py + 1);
              py_next =  py + 1;
            }
            else
              if ( ( other == ALMOST_WHITE ) && ( readpixl(px,py-1)
) )
              {
                px_next = left(px,py-1);
                py_next = py-1;
              }
          }
          else
            px_next = ILLEGAL;
      }
      for ( x = px; ( x < 512 ) && ( readpixl(x,py)
        px_right = x-1;
        py_right = py;
      for ( x = px; x < = px_right; x + + )
        wrtpixl(x,py_right,ALMOST_WHITE);

      /* fill region from px to px_right then calculate region
        parameters */

      xs = (double)px;
      xf = (double)px_right;
      ys = (double)py_right;

      *area + = (xf-xs + 1);

      *sum_x + = ( 0.5 * ( xf - xs + 1.0 ) * ( xf + xs ) );

      *sum_y + = ( ys * ( xf - xs + 1.0 ) );

      *sum_x_squared + = (( 1.0 / 6.0 ) *
                    ( xf * ( xf + 1.0 ) * ( 2.0 * xf + 1.0 ) -
                     xs * ( xs - 1.0 ) * ( 2.0 * xs - 1.0 )   ));

      *sum_y_squared + = ( ys * ys *  ( xf - xs + 1.0 ) );

      *sum_xy + = ( 0.5 * ys * ( xf - xs + 1.0 ) * ( xf + xs ) );

      link (px_right + 1,py_right,&above,&below,&e1,&e2,&px1);
      py1 = py_right;
```

```
        if ( ( ( above = = 0 ) || ( below = = 0 ) ) &&
           ( readpixl(px1,py1)ALMOST_WHITE ) )
        {
          push(x_stack,y_stack,px1,py1);
          if ( above > 0 )
            pushd(d_stack,UP);
          else
            pushd(d_stack,DOWN);
        }
      }
FINI: ;
    }
    res_buf(0);
}



/*                    fill_disk()

fills in the disc region of the disk - pixels forming part of the region
are set to ALMOST_WHITE2, chips (dark areas) are set to GREY2
 - border pixels are taken as being WHITE

inputs:-

xbar,ybar  - co-ords of centre of area of entire image, used
            to find seed point for fillin algorithm
            -  seed point taken as mid point
            between inner and outer borders

outputs (used for calculating ellipse parameters):-

sum_x
sum_y
sum_x_squared
sum_y_squared
sum_xy
area              actual disk area - including chips
chip_area           chip area

*/

fill_disk(xbar,ybar,sum_x,sum_y,sum_x_squared,sum_y_squared,sum_xy,area,chip_area)

int xbar,ybar;
double *sum_x,*sum_y;
double *sum_x_squared,*sum_y_squared;
double *sum_xy;
double *area;
double *chip_area;
{
int px_right,py_right;
int px,py;
int other;
int u;
int dir;
```

```
int above,below;
int e1,e2;
int x;
int px_next,py_next;
int px1,py1;
int success;
int x1,nblack;
double xs;
double xf;
double ys;


    mov_buf(0);
    stack_pointer = sp = 0;

    *sum_x = 0.0;
    *sum_y = 0.0;
    *sum_x_squared = 0.0;
    *sum_y_squared = 0.0;
    *sum_xy = 0.0;
    *area = 0.0;
    *chip_area = 0.0;


    /* get start point */

     success = FALSE;
     nblack = 0;

     for ( x = 0; ( x < = xbar ) && (success = = FALSE); x+ + )
     {
       if ( (readpixl(x,ybar) = = BLACK) )
         nblack+ +;
       else
         nblack = 0;

       if ( nblack = = 5)
         success = TRUE;
     }

     x1 = x;

     for ( x = 0; ( x < = xb[illegible](x,ybar)

     xbar = ( x1 + x ) / 2;

    push(x_stack,y_stack,left(xbar,ybar),ybar);
    pushd(d_stack,DOWN);
    push(x_stack,y_stack,left(xbar,ybar + 1),ybar + 1);
    pushd(d_stack,UP);
    px = xbar;
    py = ybar;

    while(stack_pointer > 0)
    {
      pop(x_stack,y_stack,&px_next,&py_next);
      popd(d_stack,&dir);
```

```
px_right = 511;
py_right = py;

if ( dir = = DOWN )
{
  other = UNDER;
  u = 2;
}
else
{
  other = ONTOP;
  u = 1;
}

while(1)
{
  if ( ( px_next = = ILLEGAL ) ||
      ( readpixl(px_next,py_next) = = ALMOST_WHITE2) ||
      ( readpixl(px_next,py_next) = = GREY2        ) )
  goto FINISH;

  px = px_next;
  py = py_next;


  link(px-1,py,&above,&below,&e1,&e2,&px1);
  py1 = py;


  if ( ( ( dir = = DOWN ) && ( above > 1 ) ||
       ( dir = = UP   ) && ( below > 1 )    ) && (  px > px_right ))
    goto FINISH;

  if ( ( above > 0 ) && ( below > 0 ) )
  {
    if ( u = = 1 )
    {
      px_next = e1;
      py_next = py + 1;
    }
    else
    {
      px_next = e2;
      py_next = py-1;
    }
  }
  else
  {
    if ( ( readpixl(px,py)=ALMOST_WHITE2 ) &&
       ( readpixl(px,py)=GREY2 ) )
    {
        push(x_stack,y_stack, left( lright(px,py),py ) , py );
        pushd(d_stack,dir);
    }
    if (  ( above = = 0) ) && ( below = = DOWN )  ||
         ( above   = 0 ) && ( below = = 0 ) && ( dir = = UP   )  )
     {
```

```
            if ( u = = 1 )
            {
              px_next = e1;
              py_next = py + 1;
            }
            else
            {
              px_next = e2;
              py_next = py-1;
            }
          }
          else
            if ( ( other == ALMOST_WHITE2 ) && ( readpixl(px,py + 1)
&&
                        = GREY2 ) readpixl(px,py + 1)

               ( other == ALMOST_WHITE2 ) && ( readpixl(px,py-1)
&&
                        = GREY2 ) readpixl(px,py-1)
            {
              if ( ( other == ALMOST_WHITE [illegible] ( readpixl(px,py + 1)
2 ) &&
                        = GREY2 ) readpixl(px,py + 1)

              {
                 px_next = left(px,py + 1);
                 py_next =  py + 1;
              }
              else
               if ( ( other == ALMOST_WHI [illegible] && ( ( readpixl(px,py-1)
TE2 ) &&
                        = GREY2 ) () readpixl(px,py-1)
)
               {
                  px_next = left(px,py-1);
                  py_next = py-1;
               }
            }
            else
              px_next = ILLEGAL;
        }
        for ( x = px; ( x < 512 ) && ( WHITE readpixl(x,py) [illegible]
          px_right = x-1;
          py_right = py;

        /* fill the area */

        for ( x = px; x < = px_right; x + + )
        {
          if ( readpixl(x,py_right) == GREY )
            wrtpixl(x,py_right,ALMOST_WHITE2);
          else
          {
            *chip_area + = 1.0;
            wrtpixl(x,py_right,GREY2);
          }
        }
```

```
        /* calculate region parameters */


        xs = (double)px;
        xf = (double)px_right;
        ys = (double)py_right;


        *area += (xf-xs+1);

        *sum_x += ( 0.5 * ( xf - xs + 1.0 ) * ( xf + xs ) );

        *sum_y += ( ys * ( xf - xs + 1.0 ) );

        *sum_x_squared += (( 1.0 / 6.0 ) *
                         ( xf * ( xf + 1.0 ) * ( 2.0 * xf + 1.0 ) -
                          xs * ( xs - 1.0 ) * ( 2.0 * xs - 1.0 )   ));

        *sum_y_squared += ( ys * ys * ( xf - xs + 1.0 ) );

        *sum_xy += ( 0.5 * ys * ( xf - xs + 1.0 ) * ( xf + xs ) );


        link (px_right+1,py_right,&above,&below,&c1,&c2,&px1);
        py1=py_right;


        if ( ( ( above == 0 ) || ( below == 0 ) ) &&
            ( ( readpixl(px1,py1)!=ALMOST_WHITE2 ) &&
              ( readpixl(px1,py1)!=GREY2 ) ) )
        {
          push(x_stack,y_stack,px1,py1);
          if ( above > 0 )
            pushd(d_stack,UP);
          else
            pushd(d_stack,DOWN);
        }
      }
FINISH: ;
    }
    res_buf(0);
}




push(x_stack,y_stack,px,py)
int px,py;
int *x_stack,*y_stack;
{
int i;
```

```
    stack_pointer++;
    *(x_stack + stack_pointer) = px;
    *(y_stack + stack_pointer) = py;

}

pop(x_stack,y_stack,px,py)
int *px,*py;
int *x_stack,*y_stack;
{
  if (stack_pointer > 0)
  {
    stack_pointer--;
    *px = *(x_stack + stack_pointer + 1);
    *py = *(y_stack + stack_pointer + 1);
  }
  else
    *px = *py = -1;
}



pushd(d_stack,d)
int d;
int *d_stack;
{
   sp++;
   *(d_stack + sp) = d;
}

popd(d_stack,d)
int *d,*d_stack;
{
  if (sp > 0)
  {
    sp--;
    *d = (*(d_stack + sp + 1));
  }
  else
    *d = -1;
}

link(x,y,above,below,e1,e2,p1)
int x,y;
int *above,*below;
int *e1,*e2,*p1;
{
int dx,dy;
int x1,x2;
int i;

      x = left(x,y);

    *above = *below = 0;
    dx = dy = 1;
```

```
x1=x2=ILLEGAL;
*e1=*e2=ILLEGAL;

if ( readpixl( x-dx,y+dy ) == WHITE )
{
  (*above)++;
  x1=x-dx;
}
if ( readpixl( x-dx,y-dy ) == WHITE )
{
  (*below)++;
  x2=x-dx;
}


while ( readpixl(x,y) == WHITE )
{
if ( ( readpixl( x,y+dy ) == WHITE )
   &&( readpixl( x-dx,y+dy)WHITE ) )
  (*above)++;
if ( ( readpixl( x,y-dy ) == WHITE )
   &&( readpixl( x-1,y-dy )WHITE ) )
  (*below)++;
 if ( readpixl(x,y+dy) == WHITE )
  x1=x;
 if ( readpixl(x,y-dy) == WHITE )
  x2=x;
 x++;
}
*p1=x;

if ( readpixl(x,y+dy) == WHITE )
  x1=x;
if ( readpixl(x,y-dy) == WHITE )
  x2=x;

if ( x1                    = ILLEGAL )
{
  for ( i=x1; (i<512 ) && ( readpixl(i,y+dy) == WHITE ); i++ );
  *e1=i;
}

if ( x2                    = ILLEGAL )
{
 for ( i=x2; (i<512 ) && ( readpixl(i,y-dy) == WHITE ); i++ );
 *e2=i;
}

if ( ( readpixl( x-dx,y+dy)WHITE )
   &&( readpixl( x,y+dy ) == WHITE ) )
  (*above)++;

if ( ( readpixl( x-dx,y-dy WHITE )
   &&( readpixl( x,y-dy ) == WHITE ) )
  (*below)++;
}
```