ISSN 1369-1961

Department of
Computer Science

Technical Report No. 757

ML-PVA User's Manual

D. Turgay Altilar

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

January 1997

ML-PVA

USER'S MANUAL

D Turgay ALTILAR

January 1997

1 INTRODUCTION **1**

2 SYSTEM ARCHITECTURE OVERVIEW **3**

2.1 ML-PVA 3
2.2 WORKSTATION 4
2.3 PC 4
2.4 NETWORKING 5
2.5 PASSIVE DEVICES 5

3. PARALLEL VIDEO ACCELERATOR ENVIRONMENT **7**

3.1 BOOTING ML-PVA 9
3.2 THE FRAME BUFFER 10
3.3 A COMMAND INTERPRETER FOR THE FRAME BUFFER MANIPULATIONS : ISPMAIN 14

4. WORKSTATION ENVIRONMENT **17**

4.1 ADMINISTRATION TOOL (ADMTOOL) 17
4.2 ISP 18
4.3 OTHER APPLICATIONS 19

5. PC ENVIRONMENT **21**

5.1 CONTENTS OF THE TEMPLATE USER DIRECTORY 21
5.2 SHAREABLE AND ADMINISTRATIVE FILE STRUCTURE OF PC 25

6 THE PROGRAMMING MODEL OF ML-PVA : SAPS **29**

7 DEVELOPING AN APPLICATION FOR ML-PVA **31**

7.1 WRITING A NON-SERVER PROGRAM 31
7.2 THE USE OF LOCATOR FILES 34
7.3 ACCESSING FRAME BUFFER FROM SYSTEM HOST 40
7.4 WRITING A CLIENT-SERVER PROGRAM 48

APPENDIX A -LIBRARY CALLS FOR DSP PROGRAMMING (WKLIB) **53**

APPENDIX B -LIBRARY CALLS FOR DSP PROGRAMMING (WK2LIB) **55**

APPENDIX C LIBRARY CALLS FOR SYSTEM HOST PROGRAMMING **61**

BIBLIOGRAPHY **63**

1 Introduction

This document has been prepared to help MonaLisa Parallel Video Accelerator (ML-PVA) users in program development and running . It explains only the basic concepts. Further details exist in the forms of reports, manuals, and published papers given in the Bibliography section.

MonaLisa (**M**odelling **NA**tura**L** **I**mages for **S**ynthesis and **A**nimation) is a EU supported RACE II project whose main goal is to develop a virtual reality platform by means of both hardware and software, for TV studio production and post production by mixing environment for using synthetically generated images and real images together. ML-PVA, the parallel architecture for video processing has been developed at Queen Mary and Westfield College (QMW) under MonaLisa project.

This manual replaces the first version of "MonaLisa Accelerator User's Manual" , written by A V Sahiner and P Lefloch. However, to support the previously developed applications, the former directory structures has been kept. Users are expected to follow this manual from now on, eventhough backward support exists.

Although ML-PVA is the actual parallel machine, there are two more separate platforms, performing different tasks, to build the complete system. The first one is a standard workstation running a number of user interface programs. The second one is a PC for compiling DSP programs. They are all attached to Ethernet and they all have TCP/IP based communication support. This manual explains the systems information upto Section 6. Programming ML-PVA is the main concern of the Section 6 and 7. Thus a user having systems information and willing to exercise programming could skip the first six chapter and begin with the Section 6.

For any further help concerning the use of actual machine, providing related documentation send an e-mail to {altilar,turgay}@dcs.qmw.ac.uk. Bugs, malfunctions and critics will be welcomed at the same addresses.

2 System Architecture Overview

The system consists of three main computing environments which are;

- Parallel Video Accelerator (ML-PVA)
- Workstation (UNIX based)
- PC (having windowing and multitasking utilities).

It also comprises passive peripherals like DAC (Digital to Analogue Converter) , ADC (Analogue to Digital Converter), video recorder, and monitor. A system user is expected to know some information about these three platforms and about the use of passive peripherals.

2.1 ML-PVA

The following quick overview about the ML-PVA is to make user familiar with the system architecture. The current architecture of accelerator, ML-PVA, in QMW comprises the following parts. (Fig. 2.1)

- A frame buffer of 32Mbytes

A bank of DRAMs on the high speed bus, accessible via IOPs where video sequences are kept.

- 5 IOPs (Input Output Processors)

Intelligent input output processors to access the frame buffer. A standard interface is provided by the use of this hardware. Generically all IOPs are the same accept a few add on devices for the ones attached to converters.

- A 68030 main processor (PVA Host) configured with
  - A local hard disk of 40Mbytes
  - An external hard disk of 320Mbytes (backup disk)
  - A floppy disk driver

A card computer running a real-time operating system, controlling the ML-PVA, and responsible for the management of pool processors (DSPs), IOPs, and the frame buffer.

- 8 DSPs (DSP96002)

Processing units on which given tasks run. Attached to the frame buffer via IOPs to acquire global data. Two of them located on the same board, sharing the resources like global memory and buffers, on the board. Two boards constitutes a cluster which has single IOP to access the frame buffer.

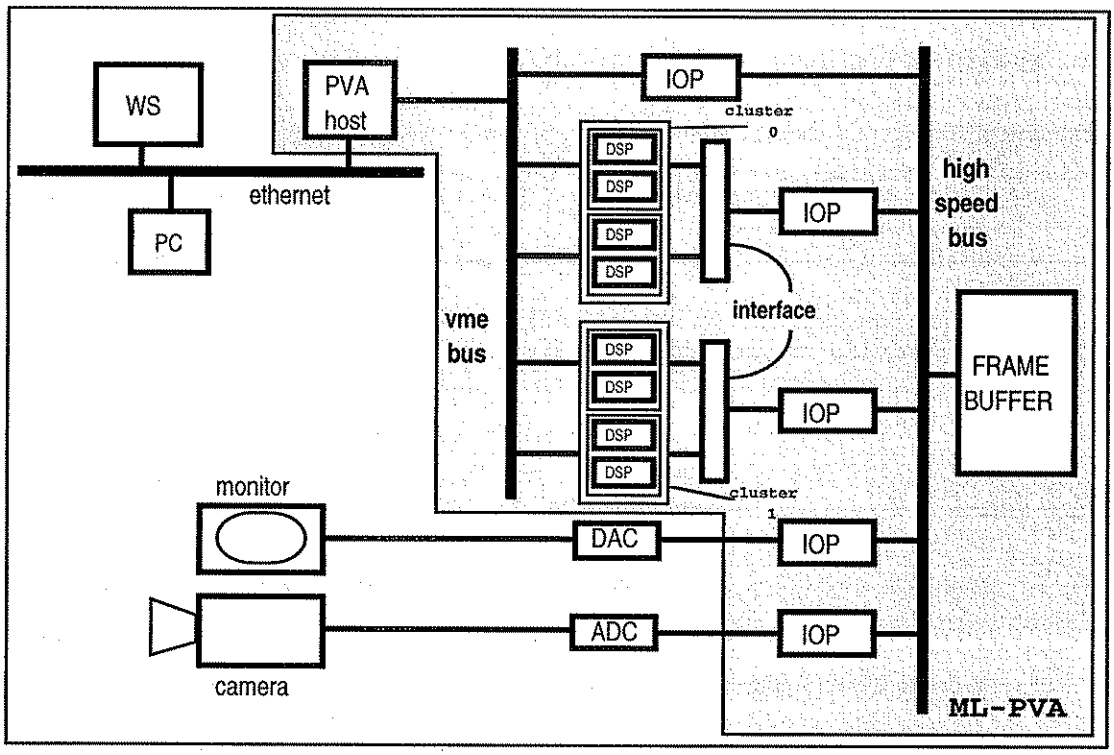

Figure 2.1 Systems architecture and ML-PVA

2.2 Workstation

The workstation which acts as a system host controls ML-PVA through Ethernet via PVA Host. Workstation has two distinct tasks; providing a user interface to ML-PVA and acting as a client for client-server based programming. Socket based communication, using some previously set addresses, exists between them. (See communication library documents for detailed information about the socket based communication.) As usual, any other workstations or X-terminals having access to this machine could run these applications, as well.

2.3 PC

PC provides an environment for compiling DSP codes by running a special C compiler for DSPs. After compilation, the produced codes should be uploaded to PVA-Host. There is no further communication or interaction between PC and any other system platforms. Besides a compilation environment, it provides a debugging facility. A debugging tool, ADS, and related hardware have been installed on the PC for debugging DSPs via a direct communication link. This debugging tool addresses low level problems and unlikely to be used by the programmers unless they come across problems while they are dealing with low level programming like direct memory addressing.

2.4 Networking

All of the three platforms are attached to the Ethernet. This provides user to carry out all tasks via a workstation on which user interface modules run. DVX, a multi user environment provider program, run on the PC, a user can have a number of PC windows on a workstation or an x-terminal screen. The PVA-Host also supports TCP/IP communication, which enables users to Telnet or to ftp to and from. Although the names and addresses are known by the Departmental name server, it would be better to use the network names and the addresses given in Table 2.1. A user needs this information while booting the PVA-Host via NFS, as well (Section 3.1). A user should bear in mind that accessing **`kiri`** and **`topaz`** is possible only for the machines on the same sub-net which means having the Ethernet address which begins with **`138.37.88.`**.

Network Name	Machine	Ethernet Address
kiri	Parallel Video Accelerator	138.37.88.142
ankara	SGI Workstation	138.37.88.73
topaz	PC	138.37.88.78

Table 2.1. Network names and addresses of system platforms

2.5 Passive Devices

As seen in Fig. 2.1, there are four passive devices related to the ML-PVA.

- A camera having an RGB output. (If camera has composite output, either introduce a composite to RGB converter between camera and ADC or work with gray level images by connecting camera output to G/Y input of ADC)

- An analogue to digital converter accepting RGB (or YUV) signals and converts them into first digital (4:2:2) serial then digital (4:2:2) parallel to be written on the frame buffer through the IOP.

- A digital to analogue converter accepting RGB signals in digital form and converts them to analogue signals and synchronisation signal to feed monitor.

- A monitor which is capable of accepting RGB signals with external synchronisation signal.

For detailed information about ADC and DAC refer to the manuals given in the Bibliography Section.

3. Parallel Video Accelerator Environment

PVA is the heart of the system, providing parallel processing power and frame buffer capability. Parallel video accelerator environment comprises four main units;

- Parallel Video Accelerator Host (PVA-Host)
- Digital Signal Processor Boards (DSPs)
- Intelligent I/O Processing Boards (IOPs)
- Frame Buffer (FB)

PVA-Host is a 68030 based card computer running a real time operating system, OS9. PVA-Host is the host of the accelerator which

- monitors the DSPs,
- dispatches tasks to DSPs,
- communicates with DSPs and System-Host,
- initialises IOPs ,
- controls frame buffer accesses.

These functionalities are performed by a number of programs. All of the program codes are available on the disk. As the file structure of the PVA-Host is complex, only a portion of the file structure which is closely related to programming and using ML-PVA is taken into consideration. /dd/MONALISA is the main directory of OS9 software developed at QMW. The directory structure located under /dd/MONALISA is depicted in Table 3.2. A user should realise that any changes in these codes would affect all other users of the system as well.

```
/dd
  +---MONALISA
       +---Comms
       |    +---RPC
       |    +---SharedMem
       |    +---Signals
       |    +---UnixComms
       |    +---WKernelComms
       |
       +---DEBUG
       +---defs
       +---DRIVERS
       |    +---DEFS
       |    +---Dspr
       |    +---Dsps
       |
       +---LIB
       +---PoolMan
       |    +---Loader
       |
       +---SAPS
       +---Utils
       \---Various
```

Table 3.2. Directory structure under /dd/MONALISA

The directories and information about their contents are as follows:

/dd/MONALISA/Comms: All files regarding communication of PVA-Host with the other environments are collected under this directory under five sub-directories. Directories and overall definitions of existing files are as follows.

RPC - Remote procedure call mechanism
SharedMem - Support of shared memory mechanism for RPC
Signals - Signal management support for interprocess comm.
UnixComms - UNIX-OS9 communication functions(TCP/IP based)
WKernelComms - DSP-OS9 communication functions

/dd/MONALISA/DEBUG : Contains the files which provide a number of functions to debug OS9 programs.

/dd/MONALISA/defs : All of the header files that have been developed at QMW located under this directory. (These definitions must match with the ones on the workstation and on PC.)

/dd/MONALISA/DRIVERS : PVA-Host accesses DSPs via device drivers like all other devices attached to it. There are two different way of accessing a DSP; as a serial device or as a random block device. The directories and description of them are as follows.

/dd/MONALISA/DRIVERS/DEFS : Contains device drivers' header files.

/dd/MONALISA/DRIVERS/Dsps : Contains the source code files for accessing DSP as a serial device.

/dd/MONALISA/DRIVERS/Dspr : Contains the source code files for accessing DSP as a random block device.

/dd/MONALISA/LIB : Libraries, use for compiling application programs on PVA-Host stays here. Makefiles has lines to use the libraries in this folder while linking.

/dd/MONALISA/PoolMan : Besides the pool management tasks like processor initialising, program loading, pool monitoring, The Pool Manager performs the tasks of a scheduler, and dispatcher. This directory is the one most likely to be updated frequently. An implementation of a new call, or even adding a few debug lines would lead some changes in the files under this directory.

/dd/MONALISA/PoolMan/Loader : This directory comprises the source codes for downloading programs (SAPS) to DSPs via global static and dynamic rams of the DSP cards.

/dd/MONALISA/SAPS : This directory is the home of the application codes. There should be two copies, one for even numbered DSPs and one for odd numbered DSPs, (Fig. 3.1) of any application to run on DSPs. Be sure that these file names could be produced by concatenating "a.hex" or "b.hex" to the name written in "SAPS" file on the workstation. (E.g. If it is `hello` in SAPS file on ankara under /usr/pva_adm, **`helloa.hex`** and **`hellob.hex`** should be the files under this directory.) There are two special files, **`wkernela.hex`** and **`wkernelb.hex`**, both of which have to be there forever as they are boot programs of DSPs.

/dd/MONALISA/Utils : This directory contains several utilities for the system like **`dbv96dbg`** , a low level debugger for DSPs; **`dbv96loader`**, DSP application (SAPS) loader .

/dd/MONALISA/Various : This directory is the host of debugging information files like **`pmdebug`**, **`lddebug`** and **`printdebug`**. A user should keep in mind that the processes on ML-PVA triggered by a daemon (**`ispd`**) and input and output devices are directed to a null device. Thus one needs to write into a file for debugging purposes. It is suggested to erase these files frequently as messages are always appended.

The given directory structure comprises the codes that have been developed at QMW. In addition, there are two other parties owning the files on PVA-Host; Microware Systems Corporation and DVS. OS9 operating system files are a commercial product developed by Microware Systems Corporation. DVS from Germany developed programs, libraries and device drivers to operate their specially developed hardware like IOPs and address generator for the project.

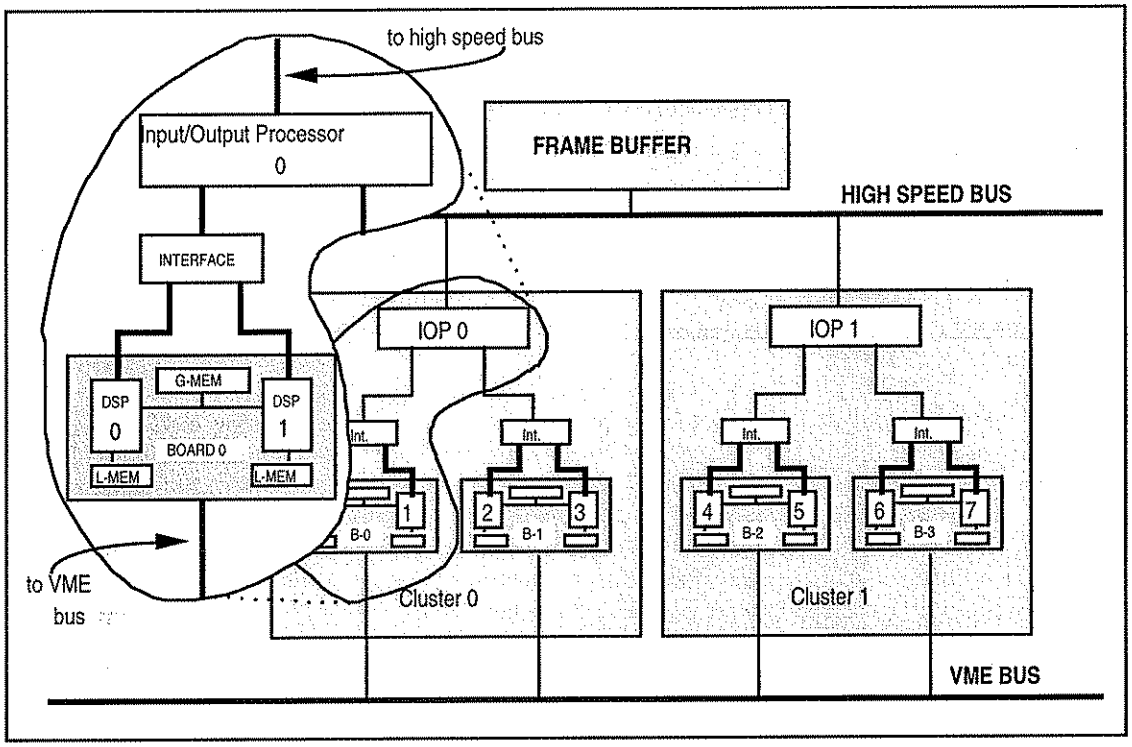

Figure 3.1 : DSPs, Boards, IOP numbering and connections

3.1 Booting ML-PVA

The PVA-Host goes through the booting phase when it is powered up. It initially searches for a OS9 Kernel and a valid boot file in the memory (EEPROM). There are two sources to boot the machine; local disk or any remote disk via NFS. This boot file keeps the latest saved configuration to decide which device would be the booting source. After reading the boot file, the up-to-date information taken from the boot file is displayed on the screen to inform the user. It waits for ten seconds to be interrupted to reconfigure booting device. If it is not interrupted by pressing "control-d", it keeps on booting accepting that the displayed parameters are correct.

Consider that the active boot file is to boot PVA-Host via NFS. A user will see an output on the console screen as in Fig. 3.2. The Internet address of ISP200 is the address of PVA-Host, namely `kiri`, in this specific case. The address for HOST is the address of System-Host which is ankara in this case. Home directory points the directory on the PVA-Host under which relevant files could be found, `/usr/dd_824` in this example. (Make sure that this directory has been exported on the UNIX site. Look up UNIX manuals for the file `/etc/exports` and the command `exportfs` to export the directory if needs. Typing `exportfs` on a UNIX system, displays the exported directories.) And finally, startup file on host is the name of the OS9 shell script to be run at the beginning of the boot session. (When it is booted via NFS, new environment takes over the PVA-Host's responsibilities and tasks)

```
Bootstrap program version      : 1.8/921203 (MVME147)
Protocol for bootstrap         : "NFS"
Internet Address for ISP200    : "192.135.231.215"
Internet Address for HOST      : "192.135.231.204"
Home directory on HOST         : "/usr/dd_824"
Startup file on HOST           : "startup"
```

Figure 3.2. NFS bootstrap information

If the last saved configuration holds local booting information, user sees an output as in Fig 3.3.

```
Bootstrap program version      : 1.8/921203 (MVME147)
Protocol for bootstrap         : "LOCAL"
Startup file on HOST           : "startup"
```

Figure 3.3. LOCAL bootstrap information

During the count-down, a user could interrupt to reconfigure the system booting parameters or even system booting protocol. After interrupting the system boot, boot program asks for the protocol first expecting one of two strings to be typed; **LOCAL** or **NFS**. At any input stage of reconfiguration a plain ENTER means accepting the existing value. Having the protocol, reconfiguration goes through the relevant questions to acquire the new information. Finally it asks for saving this newly generated protocol. It does not save the configuration to be used in the further booting sessions unless the user confirms to save the changes however the changed values will be valid for only the next booting.

By the end of the count-down, boot program goes for the given values. If a user types **-l** for the startup file on host, an OS9 shell prompt will appear without running any script. In other words, this parameter enables a user to access OS9 shell immediately. At the end of the shell session, it will proceed booting with the previous parameters.

According to the boot file, local or remote disk is attached first. It searches for /dd directory then it runs the given startup file. The PVA-Host has been configured to minimize user or administrator interaction as much as possible. For the sake of simplicity, two more script files , developed for this system, run automatically as they are called from the startup file, **/dd/startup**. These two files are **/dd/MONALISA/init.mlpva**, and **/dd/MONALISA/DRIVERS/startup.dsp**. Although there are two more files **start.isp** (start.nfs) and **finish.nfs** (finish.isp) which run system daemons, it is not our concern to go through these files as they are there to initialise basic OS9 daemons.

The startup file, **startup** (Fig. 3.4), sets environmental parameters, loads and initialises system daemons, loads and initialises DVS specific device drivers, loads **ispmain**, which is a command interpreter for programming the ISP System. It calls **startup.dsp** (Fig. 3.5) to load and initialise drivers to DSP boards. Finally it calls **init.mlpva** (Fig. 3.6) to load the programs developed at QMW.

Booting session ends by OS9 login prompt on the console screen. If any fatal error happens during booting, it goes to the beginning of the boot file and runs again.

More detailed information about booting of PVA can be found in ISP200 Installation and Setup Guide.

3.2 The Frame Buffer

The frame buffer is 32MByte RAM attached to the high speed bus. Although it is RAM, OS9 treats as if a file system mounted as **/y0**. Relevant device drivers to support this facility have been developed by DVS. These drivers are activated at bootstrap. As it is a file system a user could run standard OS9 commands like **dir**, **copy**, **del**, **rename**, **makdir**, and **deldir**, on the frame buffer. The video sequences (data objects) act as files under located under the directory **/y0**. It is rare to access the frame buffer as a file system, a user would better to use the command interpreter **isp** on OS9 or **ispmain** on the system host.

Data transfers both to and from the frame buffer are done under the control of PVA-Host by programming the intelligent input/output processors (IOPs). PVA-Host makes the address generator

produce addresses for the transfer and initialises IOPs. IOPs' have a FIFO buffer to keep data. Considering data transfer from frame buffer to DSPs' memories, the first step is done between the frame buffer and IOPs' FIFO buffers via high speed bus. Data transfer between IOPs' FIFO buffers and attached DSPs' memory is the second step via a special interface. It is happened in the reverse order for data transfer from DSPs' memory to the frame buffer. This two stepped data transfer model provides concurrent data transfers to and from the frame buffer.

To realise this model, DSPs feed PVA Host with the name of the file to be attached, data transfer direction, size of data to be transferred. These steps are transparent to a user unless a user implements a new scheme of data transfers. A user to program at such level is encouraged to go through the implementation of **`WKGetData`** and **`WKPutData`** commands for details.

```
-ntnp
*
*   startup     The commands in this file are highly system dependent and
should
*               only be modified by an experienced user.
*
*   960104 SdH  changed NFS entry for local boot
*   960228 DTA  ISP daemon path has changed to the domestic one
*
tmode -w=1 nopause
-tnp

setenv PORT /term
setenv TERM ansi
setenv PATH /dd/DVS/CMDS:/dd/ISP/CMDS:/dd/NFS/CMDS:/dd/DVS
setenv DVS_STARTUP      /dd/DVS/isp_startup
setenv DVS_HELPMSG      /dd/DVS/SYS/helpmsg
setenv DVS_ERRMSG       /dd/DVS/SYS/errmsg
setenv DVS_REFFILEPATH /dd/DVS/CMDS/REF_TZ40
setenv DVS_FMUXUPPATH   /dd/DVS/CMDS/MUX
*
chd /dd/CMDS/BOOTOBJS
load -d t1 ; iniz t1
*load -d h1.vccs.A201 ; iniz h1 ;        * backup disk
*load -d rbteac d0.teac.a201 ; iniz d0; * floppy disk
*
* start (TCP/IP) or (TCP/IP ; NFS) for LOCAL Boot
((chd /dd/ISP ; start.isp) ; (chd /dd/NFS ; start.nfs)) <>>>/nil &
*
* begin of ISP specific startup
*
load -d /dd/DVS/V0i_1.1/BIN/scbbus  ;*  BBAG interface
load -d /dd/DVS/V0i_1.1/BIN/v0_ipca_E08 ; iniz v0 ;* ipca200
load -d /dd/DVS/I0i_1.6a/scisp
load -d /dd/DVS/I0i_1.3/i0_ipca_bbe08 ; iniz i0 ;* ipca200
*
initag5.0 >>>/nil
(list /dd/DVS/EXE/agconf.200.32 ! agt) >>>/nil
*
/dd/DVS/CMDS/loadfmux -b=00 -d=/dd/DVS/CMDS/MUX/y0uprog.200
/dd/DVS/CMDS/loadfmux -b=08 -d=/dd/DVS/CMDS/MUX/d1iuprog;* DSP IOP0
/dd/DVS/CMDS/loadfmux -b=10 -d=/dd/DVS/CMDS/MUX/d1iuprog;* 656 Dev1
/dd/DVS/CMDS/loadfmux -b=18 -d=/dd/DVS/CMDS/MUX/d1iuprog;* DSP IOP1
/dd/DVS/CMDS/loadfmux -b=20 -d=/dd/DVS/CMDS/MUX/d1iuprog;* 656 Dev2
*
load -d /dd/DVS/W0i_1.2/BOOTOBJS/scfint
load -d /dd/DVS/W0i_1.3/BOOTOBJS/w0_i200 ; iniz w0 ;* ipca200
*
load -d /dd/DVS/Y0_2.5/BOOTOBJS/rbisp
load -d /dd/DVS/Y0_2.5/BOOTOBJS/y0_200mb.ap0 ; iniz y0
load -d /dd/DVS/ISP824/ispmain
*
* start daemon for server on Ethernet using ispmain
* host will select command and data channel
*
load -d /dd/DVS/CMDS/genproc
load -d /dd/MONALISA/NameServer/ispd
ispd -s=ispmain -e <>>>/nil &
*
/dd/MONALISA/DRIVERS/startup.dsp
*
/dd/MONALISA/init.mlpva
chd /dd ; chx /dd/CMDS ; -nt
tmode -w=1 xon=11 xoff=13 abort=03 quit=05 ; -t
ex tsmon -dp /term <>>>/nil &
*
*============================ end of startup ====================*
```

Figure 3.4 The current startup file for local booting.

```
-t
*  startup.dsp     load and initialise drivers to DBV96 boards
*                  (this shell script is called during bootstrap
*                  by the main startup file (see /dd/startup))
*
*  960112  DTA     Modified for initiailising 8 processors.
*
*
*  SCF drivers (One driver per processor)
*
load /dd/MONALISA/DRIVERS/Dsps/scdsp -d
*
load /dd/MONALISA/DRIVERS/Dsps/dsps0 -d; iniz dsps0;
load /dd/MONALISA/DRIVERS/Dsps/dsps1 -d; iniz dsps1;
load /dd/MONALISA/DRIVERS/Dsps/dsps2 -d; iniz dsps2;
load /dd/MONALISA/DRIVERS/Dsps/dsps3 -d; iniz dsps3;
load /dd/MONALISA/DRIVERS/Dsps/dsps4 -d; iniz dsps4;
load /dd/MONALISA/DRIVERS/Dsps/dsps5 -d; iniz dsps5;
load /dd/MONALISA/DRIVERS/Dsps/dsps6 -d; iniz dsps6;
load /dd/MONALISA/DRIVERS/Dsps/dsps7 -d; iniz dsps7;
*
* RBF drivers (One driver per board)
*
load /dd/MONALISA/DRIVERS/Dspr/rbdsp -d
load /dd/MONALISA/DRIVERS/Dspr/rbdsp.stb -d
*
load /dd/MONALISA/DRIVERS/Dspr/dspr0 -d; iniz dspr0;
load /dd/MONALISA/DRIVERS/Dspr/dspr2 -d; iniz dspr2;
load /dd/MONALISA/DRIVERS/Dspr/dspr4 -d; iniz dspr4;
load /dd/MONALISA/DRIVERS/Dspr/dspr6 -d; iniz dspr6;
-nt
```

Figure 3.5 The current startup.dsp file

```
-t
*
*  init.mlpva     Parallel Video Accelerator Init. Script
*
*  960322 DTA     Previously init.elseta
*
*
* load MLPVA system
*
load -d /dd/MONALISA/PoolMan/PoolMan
load -d /dd/MONALISA/Comms/UnixComms/CM_UnixOS9
load -d /dd/MONALISA/PoolMan/Loader/Loader
*
*
* load Camera Tracking module (This module is called by ispd as well)
*
* load -d /dd/MONALISA/CamTrack/CamTrack
*
* loader and debugger utilities
*
load -d /dd/MONALISA/Utils/dbv96loader
load -d /dd/MONALISA/Utils/dbv96dbg
*
* evdir :            lists all events
* evdel <event>   :  deletes event
load -d /dd/MONALISA/Utils/evdir
load -d /dd/MONALISA/Utils/evdel
*
-nt
```

Figure 3.6 The current init.mlpva file

3.3 A command interpreter for the frame buffer manipulations : ispmain

The ispmain software is a command interpreter that runs on OS9 to manipulate the frame buffer. It accepts a set of predefined commands and follows a particular syntax for programming the ISP System. It allows to set-up the system statically whereas the Host Computer video library permits to do it dynamically through UNIX-based applications. This library is based on a single routine `vdlib()` that accepts different kinds of arguments with respect to the commands to be executed on the remote ISP System. (See section 7.3 for programming details. See DVS ISP200/ISP400 Reference Manual for a detailed description of all commands and related parameters.)

In order to download or record a video sequence, a user should initialise the codec (coder/decoder), after typing **`ispmain`** to get the application shell. ADC and DAC are attached to the IOPs which can be accessed as logical devices called codec via this interface. Each codec has a number. If no codec is present in the system, then special device number 0 can be used for simple frame buffer work like putting data into the frame buffer, processing and saving the results in files. There are three devices in the current system.

Dev 0 Default device for basic input output
Dev 1 Attached to DAC which drives monitor
Dev 2 Attached to ADC which is connected to camera

Initialising the codec sets the colour mode of operation (B/W, RGB or YUV), the video synchronizing raster, the sampling frequency and the format of the data to be registered within the frame buffer. Initialising is the first command to be issued before any further commands. Once initialised, the quantifiers are effective till the next initialisation for the device. Each device needs to be initialised once individually. Jumping among the devices do not change the initialised values. An initialisation keeps the given qualifiers for the command **init** accepts the following parameters in the given order. The command syntax is

```
init [mode[frame_rate[interlace-id[linesperframe[freq[npel[nlin]]]]]]]
```

To set device (codec) 0 to record a video sequence having the following properties;

- 25 frames per second (standard PAL rate)
- a frame size of 720x576
- two field interlaced frames
- colour mode of black and white
- 625 lines per frame
- a sampling frequency of 27 MHz.

user should type

```
sdev 0
init 0 25 2 625 27. 720 576
```

To replace colour mode with YUV mode (video colour coding) and the frame size with 360x288 in the above properties (other properties kept same) user should type

```
sdev 0
init 2 25 2 625 27. 360 288
```

or

```
sdev 0
init/656 25 625 360 288
```

Note that **`init/656`** is the same command having the qualifier **`/656`** which sets three parameters implicitly.

Once the codec has been initialised, image or data files could be created in the frame buffer. Creating a file means allocating a memory area and assigning a file name to it. To create an object (file) **testseq** in the frame buffer for storing a 25 frame (of 720x576 pixels) sequence a user should type

```
build testseq 25
```

If there is a file in the frame buffer with the given name, the qualifier **/rewrite** should be used to overwrite.

```
build/rewrite testseq 25
```

This interpreter assumes the most recent **build**ed or **attach**ed name as the current file, and performs further commands over this file. In order to remove the file `testseq` at any time, user should type

```
attach testseq 3
remove
```

A simple test to examine the functionality of the output device, one could run the following sequence.

```
sdev 1
init/656
build test 1
display
pattern/ftrp
```

User would see the colour bars on the monitor. **pattern** generates a special pattern defined by the quantifier on the attached file. Quantifier **/ftrp** defines the colour bar pattern in this case.

By the end of the following sequence you will see that the image taken through the attached video camera is on the monitor with a delay of a second. (25 frame delay makes one second for the PAL standard) If you experiment the same sequence using single frame there will be no delay. (except reading/writing time which is negligible)

```
sdev 0
init 0 25 2 625 27. 720 576
build loopseq 25
sdev 2
init 0 25 2 625 27. 720 576
attach loopseq 3
record 0 -1 1 0
sdev 1
init 0 25 2 625 27. 720 576
attach loopseq
display
```

This is also the simplest way to test the functionality of **camera-ADC-IOP-Frame_Buffer-IOP-DAC-monitor** chain,

The command **file** gives the name of the currently attached file whereas **dir** lists all objects on the frame buffer. The command **help** is to list the available commands in ispmain and **help <command-name>** will bring the explanation of the command.

4. Workstation Environment

Workstation, the system host, is a standard UNIX based machine. In the current configuration we use an SGI Indigo 2, named ankara.

System host provides the user interface environment to the PVA-Host. The communication between these hosts has been realised by TCP/IP using sockets. (The programming aspect is discussed at the programming section.) AdmTool and other applications running on the system host need some environmental parameters to be defined. If these definitions are done in relevant user profile, `.cshrc`, `.login`, `.profile`. etc, user does not need to do this again. These environmental settings are;

```
ISPCONFIG    to  /usr/gfx_host/ISP/config/ispconfig.ee.d
ISPECONFIG   to  /usr/gfx_host/ISP/config/ispeconfig.ee.d
ERRMSG       to  /usr/gfx_host/ISP/errmsg
```

Use **`setenv`** command to initialise these values.

4.1 Administration Tool (AdmTool)

The basic user interface to manipulate ML-PVA environment is the **`AdmTool`**. Both source codes and executable code are available under **`/usr/pva_adm`** on ankara. (/usr/pva_adm is a linked directory)

When user runs AdmTool, a window shown in Fig. 4.1 appears on the workstation screen. If either the program terminates abnormally or there is another user currently using this tool, a proper warning message will appear mentioning that the program is non-sharable. If a user is sure that there is no one using the tool, **`-remove`** option should be used to run.

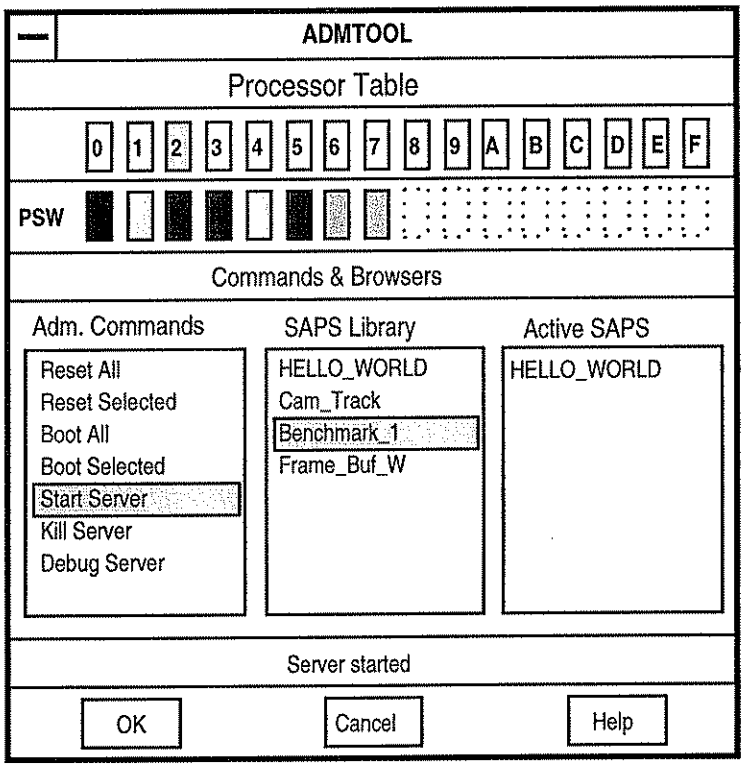

Figure 4.1 AdmTool

The Administration Tool is an OSF Motif Windows based tool that provides an interface for running applications and parallel servers on the ML-PVA. Running an application on the ML-PVA requires booting the pool processors with WKernel (micro kernel running on the pool processors to support servers). At startup the processor pool is in a reset state. It must be booted before any further actions. In

order to boot a processor, one has to select the processor or processors first by clicking on the numbered buttons then clicking on Boot Selected and OK respectively. It is also possible to boot all available processors simply just by clicking on Boot All and OK respectively. By the end of booting the selected processors, the colour bar showing the status of the processors turns into green.

The above explanation for booting processors is valid for resetting processors except user should select Reset Selected or Reset All instead of Boot Selected and Boot All respectively. Resetting a processor turns the corresponding colour bar into yellow.

The Reset command was included in the Command List as one may need to perform a software reset, if an application running on a processor fails causing either a bus error or a stack error. WKernel can not recover from these two errors and can not reboot automatically.

To start an application or a server select the command Start Server and select the name of the application/server from the SAPS Library list. Choose the target processors (one for simple applications; multiple for parallel servers) by clicking on the numbered buttons on the processor table. and finally on OK button. After loading the application to the selected processors, the corresponding colour bars turns into red while the name of the application is added to the list of Active SAPS.

Applications may be terminated using the Administration Tool, in case of servers termination stops the execution of each worker on the processor pool. To terminate a server, select Kill Server, and select the application/server to be terminated from the list of Active SAPS and press the control command OK. User is not allowed to withdraw an application just from one processor while keeping the application running on others if they started altogether. Dynamic management is allowed automatically in a different scheme. See programming section for dynamic processor allocation.

The Administration Tool includes an external window for debugging purposes. This allows users to dump selected processor's registers and memory contents. Click on a processor from the Processor Table, Debug Server and OK respectively pops up the debug window. There are buttons for halting the execution of a processor, resuming, resetting it and dumping its registers or memory address space. Before asking for any dumps, one must first halt the execution of the processor (command Halt). For a dump of the registers, select Registers from the Cascade Button (it switches from registers to memories) and Dump. The content of the registers then appear in the debug window. For a dump of the dynamic memory bank (DRAM), select Y memory from the Cascade Button, enter the address you want the dump from and press Dump.

4.2 isp

Another system host application is isp which provides the management of frame buffer. `isp` is located under `/usr/gfx_host/ISP` on ankara. (A similar program ispmain exists on OS9 for the same purpose as well. See section 3.3 for initialising, building, recording video sequences.)

The importance of this tool is due to enabling data transfers to and from the frame buffer from and to a host disk. File structures on the frame buffer and on host are different. Structure conversion is performed within this tool for both directions. To record a video sequence and to save it on the host disk user should follow the given steps;

1) Run `isp` in your home directory by writing the full path.
2) After having command interpreter's prompt indicating active device number which is 0, select device 2 to record.
3) Initialise device 2
4) Build a video sequence
5) Record a sequence

6) Save the video sequence on the disk

```
1]    a_user@ankara>/usr/gfx_host/ISP/isp
      Entering CI on  </dev/ttyq1  >/dev/ttyq1  2>/dev/ttyq1
2]    I$0>sd 2
3]    I$2>init 0 25 2 625 27.0 720 576
4]    I$2>build test 30
5]    I$2>rec
6]    I$2>save 0 30
```

It is also possible to upload or download a window taken from a whole frame. (See Host Computer Software for UNIX Operation Systems.)

4.3 Other Applications

Other applications available on the system host are

Studio; a tool for video image processing
ComTool; a higher level user interface
Camtrack; Camera tracking interface

For detailed information about these programs see the bibliography.

5. PC Environment

Although it is possible to access PC directly in the lab., a user may access PC via their workstations. The DVX program running on PC, provides remote access to PC. A user could run a **rsh** command to invoke a DOS screen on the terminal. The remotely accessible program on PC is called "dos" (Bear in mind that there is no difference between lower case and upper for PC-DOS). Thus, the command is **`rsh topaz dos`** or **`rsh topaz dos -display hostname:0.0`** where *hostname* is the name of the workstation you are using. Be sure that the system administrator has opened an account on PC (topaz) before you try to login. (Your machine should be registered by the system administrator, as well) You will reach directly to your home directory, which is **c:\users*username*** when you run rsh command.

If PC is not active (or got stuck), the power up procedure should be followed. After powering on the PC, it will go through the memory check first. It will download QuarterDeck automatically and wait for an input after prompting " Load the TCP/IP protocol? "on the screen. Press **y**(es) to continue. Then it will keep on going through the *autoexec.bat* and ends up running DVX which enables PC to be accessible via Ethernet.

The directory structure of the PC, in the ML-PVA users' point of view consists of two root directories; **c:\users** and **c:\monalisa** . \users directory is the root for all system users, where c:\monalisa comprises system-wide, user independent files, programs, tools, etc.

Users must be aware of the fact that there is no mechanism under PC-DOS to manage the accessibility of data among the users. There are no group, individual, or root rights which means there exists no hard-security and no privacy. It is users' responsibility to work properly after accessing the PC.

The system administrator creates a directory under c:\users for each user. This directory, whose name is user's name, has the structure of template that exists under **`c:\users`**. This template directory comprises everything that a user will need for DSP related compilation like **makefile** for compiling programs or creating libraries etc. . Users are expected to do all their developments under their own directories. If necessary, a number of different directories could be created under the user's home directory having the same structure. Table 5.1 gives the related directory structure under c:\monalisa and sample directory structure under c:\users .

5.1 Contents of the template user directory

Considering the C:\users\template directory, existing files and directories are explained as follows:

make.bat : A makefile to compile your DSP programs. There is only one makefile for both odd and even numbered DSPs contrary to the former makefiles described in User's Manual Version 0.1 The first parameter indicates the type of DSP where the second one file name. All information that a user may need is written in this batch file as inline commands. As it is users own makefile, one can make any change on that file like adding new libraries, changing the name of locator file, etc. Usage of this file is **`make [a] [b] [x] filename`** . *filename* is the name of the C file to be compiled without suffix **`.c`**. Option **`a`** is to compile for only the even numbered DSPs and ***`filename`*`a.hex`** is produced by the end of successful compiling. **`b`** is to compile for only the odd numbered DSPs and ***`filename`*`b.hex`** is produced by the end of successful compiling. **`x`** is for producing both files. See the inline comments in the **`make.bat`** for more detailed information considering in-between steps of the makefile (Fig. 5.1)

```
C:\
  +---MONALISA
      +---TOOLS
      |   +---ADS
      |   \---ITOOLS
      |       +---X
      |       \---TUTOR
      |
      +---USR
      |   +---LIB
      |   +---COMPILER
      |   +---BIN
      |   +---ETC
      |   \---INCLUDE
      |
      \---ADM
          +---WKLIB
          +---DVSPLIB
          +---LIBC96K
          +---WKERNEL
          |   +---DSPA
          |   +---DPSB
          |   \---WKERLIB
          |
          \---96KERNEL
```

```
C:\
  +---USERS
      +---ALIVAHIT
      |   +---MAIN
      |   |   +---LIB
      |   |   +---LIB_SRC
      |   |   +---INCLUDE
      |   |   +---LOCATOR
      |   |
      |   \---CAMTRACK
      |       +---LIB
      |       +---LIB_SRC
      |       +---INCLUDE
      |       \---LOCATOR
      |
      +---LEFLOCH
      |   +---LIB
      |   +---LIB_SRC
      |   +---INCLUDE
      |   \---LOCATOR
      |
      +---ALTILAR
      |   +---LIB
      |   +---LIB_SRC
      |   +---INCLUDE
      |   \---LOCATOR
      |
      :
      :
      |
      \---TEMPLATE
          +---INCLUDE
          +---LIB
          +---LIB_SRC
          \---LOCATOR
```

Table 5.1 ML-PVA related directory structures on PC.

LIB : It would be better to create a library of your own if you use some functions in most of your programs and do not want to compile it every time you use them. Do not forget to make the relevant changes in your makefile to take your own library or libraries into consideration. This directory inhabits the user's own assembled library files, if exits. There should be two types of files which are the library index files (probably having **`.lib`** suffix but naming of a library is user's decision) and actual codes of library functions (always have suffix **`.ln`**).

LIB_SRC : This directory comprises user's own library function calls in C or assembler and two batch files to add into a library, namely **`addlib_c.bat`** (Fig. 5.2) for adding C programs, and **`addlib_a.bat`** (Fig. 5.3) for adding assembler programs into a library. If given library does not exist under **`..\LIB`**, batch create a new file otherwise it adds or changes the given library. Do not forget that the batch file searches for the given library under **`..\LIB`** and writes back to **`..\LIB`**, as well.

```
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
: make.bat      makefile for compiling DSP programs                       :
:               make a filename  compiling for even numbered DSPs         :
:               make b filename  compiling for odd numbered DSPs          :
:                                                                         :
: 950817  DTA                                                             :
: 950904  DTA  benchmark.lib is added for getimage function              :
: 950906  DTA  local include dir. is added for directim.h                 :
: 960516  DTA  make x filename compiling for both DSPs                    :
:                                                                         :
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
echo off
if "%2" == "" goto usage
set target="%1"
echo %target%
if "%1" == "a" goto compiling
if "%1" == "b" goto compiling
if "%1" == "x" goto compiling
goto usage
:
:compiling
:
: options
: -i=interleaved assembly & c source listing -> prog.s
: -I= user include file directories
: -S=system include file directories
: -o ofn=output object file, default=prog.ol
c96002 %2.c -i -S c:\monalisa\usr\include -I .\include -o
if errorlevel == 1 goto error
if errorlevel == 2 goto error
:
:linking
:
: options
: -c cfn= locator commands from file cfn
: -L lib= library index files
: -o= write output to prog.ab
: -v= verbose
: It is possible to use your own lib. After creating it (see addlib_?.bat)
: you have to put its name after -L option.
:
if %target% == "x" goto link_b
if %target% == "b" goto link_b
:link_a
:for even numbered DSPs (DSPA)
echo llink is forking for a
echo %target%
llink %2.ol -c \monalisa\usr\bin\wkslot1a.lc -L \monalisa\usr\lib\dvsplib
\monalisa\usr\lib\wklib .\lib\benchmar.lib -o
:
goto mapping
:
:link_b
: for odd numbered DSPs (DSPb)
echo llink is forking for b
echo %target%
llink %2.ol -c \monalisa\usr\bin\wkslot1b.lc -L \monalisa\usr\lib\dvsplib
\monalisa\usr\lib\wklib .\lib\benchmar.lib -o
:
:symbolic map
:symlist %2.ab -o
:
:object size for code data and stack
:olsize %2.ab -o %1.os
```

Figure 5.1 make.bat

```
:
:mapping
echo gsmap is working
gsmap %2.ab -o
:-o= output -> prog.map
:
:formatter
echo formatter is working
form96k %2.ab -o
:-o= output -> prog.hex

if %target% == "a" goto copy_a
:copy_b
: for odd numbered DSPs (DSPB)
copy %2.hex %2b.hex
del  %2.hex
:
: if you need .map files for debugging purposes
: enable the following line to proceed
:
: copy %2.map %2b.map
:
if "%1" == "b" goto end
set target="a"
echo going to a after rba
goto link_a
:copy_a
: for even numbered DSPs (DSPA)
copy %2.hex %2a.hex
del  %2.hex
:
: if you need .map files for debugging purposes
: enable the following line to proceed
:
: copy %2.map %2a.map
:
echo going to end after a
goto end
:

:usage
echo "usage: makex [a|b|x] <filename>"
goto end
:
:error
echo failed in-between calls
:end

: One may need the following deleted files for debugging.
: In order to command out deletion, prefer to put ":" at
: the beginning of the line instead of deleting the line
: completely. Be sure that related files are produced
: by one of the above lines.
:
echo deleting files
del %2.s
del %2.ol
del %2.ab
del %2.map
del %2.ai
:
```

Figure 5.1. make.bat (cont'd)

INCLUDE : This is users own directory to put their include files. Do not forget to make relevant changes in make files. (for all batches you will use to compile source codes, like **`make.bat`**, **`addlib_a.bat`**, **`addlib_c.bat`**).

LOCATOR : As DSP has no mapping facility it is all under user's control to point the location of data structures. Although it is rare to use, the ability of defining specific addresses for data structures becomes very important for some applications like defining a shared memory, or for debugging. On the other hand one might need the physical address of a data structure which will be shared by different programs running on other DSPs or PVA host as in Camera tracking programs. Do not forget to make the relevant changes in your makefile (make.bat) to use your own locator file. The given make.bat accepts system wide known **`wkslot1a.lc`** and **`wkslot1b.lc`** unless it is declared explicitly. (See section 7.2 for the use of locator files.)

5.2 Shareable and Administrative File Structure of PC

The directory structure explained in this section is either for the use of all users or the system administrator. Users are expected to use these files as they are. Any unauthorised change might lead to a system wide failure to affect all of the users. If you need any changes of the files under C:\MONALISA, ask the system administrator.

Files for administration care are collected under the sub directory, .\ADM and user related ones are located under .\USR. Directories and their functionalities are as follows:

Under C:\MONALISA\ADM

C:\MONALISA\ADM\WKLIB : This comprises sources and batches that are used for creating user library called WKLIB. This library provides a high level interface to users with various so called system calls. New system calls should be added here, if needed to be open to all users.

C:\MONALISA\ADM\DVSPLIB : Under this directory one can find IOP communication library function sources and batches that are used to create the **dvsplib**.

C:\MONALISA\ADM\LIBC96K : This directory consists of standard library headers and library provided by Intermetrics. No source code exists, its existence is just for completeness of library structure. Header files are actually located under C:\MONALISA\USR\INCLUDE.

C:\MONALISA\ADM\WKERNEL : A low level programming interface and system calls which is a bias for the **WKLIB**.

C:\MONALISA\ADM\96KERNEL: EPROM Kernel codes exists under this directory.

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
: addlib_c.bat makefile for adding functions written in                  :
:              asm96002 programming language to the given library.       :
:              Library will be created unless it is exist                :
:              Run under "./lib_src" which is users library source       :
:              directory. New index and object files will be under       :
:              "./lib" as it is users library directory                  :
: 950817  DTA                                                            :
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
echo off
if "%2" == "" goto usage
:
:compiling
:
: options
: -i=interleaved assembly & c source listing -> prog.s
: -S= system include file directories
: -o ofn=output object file, default=prog.ol
c96002 %1.c -i -S \monalisa\usr\include -I ..\include -o
if errorlevel == 1 goto error
if errorlevel == 2 goto error
asm96002 %1.s -S \monalisa\usr\include -I ..\include -o
:
:linking
:
: options
: -o= write output to prog.ln as locate processing suppressed
: -v= verbose
: -lo = suppress locate processing
: -w  = suppress warning messages
:
llink %1.ol -lo -o -w -v
:
echo Copying the former library %2 from  ..\lib
copy ..\lib\%2
:
:librarian
:
libr %1.ln -v -L %2
:
:
: copying user library stuff (index and object) to user's lib
: directory which is suggested as "..\lib"
echo Copying %1.ln to ..\lib
copy %1.ln ..\lib
echo Copying %2 to ..\lib
copy %2 ..\lib
:
echo Deleting the temporarily created files
del *.s
del *.ol
del *.ai
del *.ln
del %2
:
goto :end
:usage
echo "usage : addlib <filename> <libname>"
goto :end
:error
echo "error : Failed inbetween calls"
:end
```

Figure 5.2 addlib_c.bat

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
: addlib_a.bat makefile for adding functions written in                  :
:              asm96002 programming language to the given library.       :
:              Library will be created unless it is exist                :
:              Run under "./lib_src" which is users library source       :
:              directory. New index and object files will be under       :
:              "./lib" as it is users library directory                  :
: 950817  DTA                                                            :
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
echo off
if "%2" == "" goto usage
:
asm96002 %1.asm -S \monalisa\usr\include -I ..\include -o
:
:linking
:
: options
: -o= write output to prog.ln as locate processing suppressed
: -v= verbose
: -lo = suppress locate processing
: -w  = suppress warning messages
:
llink %1.ol -lo -o -w -v
:
echo Copying the former library %2 from  ..\lib
copy ..\lib\%2
:
:librarian
:
libr %1.ln -v -L %2
:
:
: copying user library stuff (index and object) to user's lib
: directory which is suggested as "..\lib"
echo Copying %1.ln to ..\lib
copy %1.ln ..\lib
echo Copying %2 to ..\lib
copy %2 ..\lib
:
echo Deleting the temporarily created files
del *.ol
del *.ln
del %2
:
goto :end
:usage
echo "usage : addlib_a <filename> <libname>"
goto :end
:error
echo "error : Failed inbetween calls"
:end
```

Figure 5.3 addlib_a.bat

Under C:\MONALISA\USR

C:\MONALISA\USR\LIB : All public libraries and library objects (*.ln) are under this directory. Any change in this directory will effect all users as they all have path to this directory in their make files. No need of any changes unless a new command is for the use of all.

C:\MONALISA\USR\COMPILER: All compiling, assembling and linking programs provided by Intermetrics are under this directory. No change unless a new version of a compiler is installed.

C:\MONALISA\USR\ETC : Locator files exists under this directory for the time being. All public, text files will be located under this directory.

C:\MONALISA\USR\BIN : Empty for the time being. All public, binary files will be located under this directory.

C:\MONALISA\USR\INCLUDE : Comprises all include files, including special files for the accelerator and all include files provided by Intermetrics to access Intermetrics library.

6 The Programming model of ML-PVA : SAPS

The programming model of ML-PVA is based on Self Adaptive Parallel Servers (SAPS) model which provides Single Program Multiple Data (SPMD) parallelism. In this model, a parallel application is composed of a number of copies of the same sequential program each running on a separate processor node. This form of parallelism is widely used on message passing multiprocessor systems because it has a relatively simple and well defined structure, and it facilitates a uniform and easy-to-understand usage of communication primitives. SPMD style also provides a framework for developing parallel application software using sequential programming techniques, and therefore enables application software already developed for sequential machines to be used for parallel architectures, without undergoing major changes. (See Bibliography section for an example of transforming sequential implementation to a parallel one.)

Under the SAPS model, the mechanisms for SPMD parallelism are encapsulated within servers. A server when requested executes multiple copies of an associated sequential procedure in parallel in SPMD mode. The data is provided by the client within the service request. The interface between the application programs, as clients, and the parallel servers is conveniently hidden in the procedure call mechanism which is a well-understood facility to develop modular programs.

The interaction between a SAPS and an application as its client, for the actual provision of the service, is based on standard remote procedure call which is structured as a service-request and a service-reply is also supported by a data objects scheme which enables data decomposition. Servers fetch application data via operation invocations on data objects.

The server-application (client) duality provides the means for the separation of concerns and therefore the separation of the building of servers and their utilisation by the applications. Applications are conventional sequential programs developed independent of the concerns for parallelism. The main building block of a server is also a sequential procedure; a parallel SPMD structure is obtained when this procedure is interfaced to a standard template. This scheme allows building servers using existing sequential software without major modifications.

A SAPS has a multi-process structure. This structure contains the mechanisms for the reception of service requests, their processing, and the transmission of the results back. A dispatcher process and a pool of workers form a process farm where the dispatcher farms out work to workers and each worker when becomes idle, requests for more work from the dispatcher process. It is the multiplicity of the workers that provide parallelism within the server. Workers run on the pool processors in a one node per worker fashion. The number of nodes used by the workers of a particular server is not statically decided, it depends on run time availability of nodes. A server can start operating with a single worker and dynamically increase its worker population at run time as more nodes become available.

Within the SAPS structure it is the scheduler process which is responsible for the resource management activities. It manages the configuration (it creates the dispatcher and worker processes), and it reconfigures the SAPS structure by adding or deleting workers. The scheduler coordinates its activities with other SAPS schedulers through the pool manager.

The structure of a server is completely transparent to its clients. Each server has two message communication access points: one for handling service requests (service access point), and the other for monitoring the processor resources and coordinating its resource usage with other computing agents (resource management access point).

The SAPS structure as a whole is supported by a standard software template. The complete functionality of a SAPS as a generic server object is programmed within this template. To create a SAPS blueprint for a particular service all that is required is the interfacing of a conventional sequential procedure (the task proper) to a copy of the template. This is achieved via a local procedure call interface.

7 Developing an application for ML-PVA

There are two types of application programming;

1- PVA-Host driven (non-server) programs
2- System-Host driven (client-server) programs

The first type of application programs run over a number of DSPs under the ultimate control of PVA host. There is no direct communication between PVA host and System Host (Workstation) while the programs running over DSPs. Only the initialising actions are taken by the application (AdmTool) running on the System Host. Generic application of this kind reads data from frame buffer, performs a specific algorithm and writes results to frame buffer. This kind of application does not need any code to be run on the System Host.

The second type, client-server, performs an algorithm over a number of DSPs under the control of System-Host. The system-host, sends tasks and collects the result. It is system-host's responsibility to provide data to the server tasks. The responsibility of PVA-Host, for this type of programming, is to fetch the availability messages from DSPs and to dispatch a task.

Developing and compiling a non-server program is explained first. For client-server based programming only the differences are mentioned in the following paragraphs.

7.1 Writing a non-server program

A program can be coded in C, in assembler or in both. All the necessary steps are performed on the PC. The simplest to code would be a "Hello World" program to get familiar with the environment and compiler. (Fig. 7.1)

```
/*********************************************************
*/
/*
*/
/* hello.c    The first program on ML_PVA
*/
/*
*/
/* 960516     DTA  Implementation
*/
/*
*/
/*********************************************************
*/

main()
{
  WKprintf(0,"Hello world, I am DSP %d\n",WKmyid());
}
```

Figure 7.1. hello.c

After logging into the PC, either via an rsh command or by direct access to PC, write (or download) your program under your home directory. If it is a C program, running make.bat will be enough to compile. If it is an assembler program another makefile should be used to compile, **c96002** line should be commanded out in the make.bat. Basically, the only difference between the two Makefiles is the absence of assembler code producing step in assembler makefile. Which means that users could produce their own assembler Makefiles, especially if they have their own libraries and include files of their own.

A C program may contain some system calls specially developed for ML-PVA. (Appendix A and B). These calls are defined in the libraries which will be linked while compiling. Calls provide an interface to ML-PVA.

One could produce one's own group of commands by compiling the related sources to make personal library. (See creating a personal library and creating a new system command sections for details.)

Makefile compiles for even numbered DSPs, for odd numbered DSPs, or for both of them. The produced files, not the ones within intermediate steps, have **.hex** suffix. makefile concatenates an **a** or **b** before this suffix, depending on the type of the produced file. Be sure that file name is less than 8 to let batch file to insert a letter. Otherwise, files will overwrite the other one and the last produced one will survive without indicating any information about type it was compiled for.

Compilation ends with a text file whose extension is **.hex** . It is important to know that it is a text file as one should ftp this file to PVA-Host in ASCII mode (NOT in BINARY mode!).

The only need of accessing workstation rises here. This application should be defined in user interface, AdmTool, running on the workstation. The name of the produced code, without any suffix or a,b at the end of the file name, should be written in **/usr/pva_adm/SAPS** on ankara. Tab is essential between the file name and the application name. (This file structure will be left in the near feature and only a white-space character will be enough.)

To run this application, user should access /usr/pva_adm and run AdmTool. Select the application and run it after initialising and booting DSPs.

Here is a scheme showing all the above steps to be taken assuming that user is in front of an x-terminal (act_mach) which accesses ankara. (Pay attention to the prompts in the flow to distinguish the environments. (Fig.7.2)

This is the end of work to be done on PC. Jump into another window for ankara.

- Go to /usr/pva_adm
- add the following line in the file "SAPS" if does not exist.

`hello[tab]  HELLOWORLD` (whitespace character must be a tab)

- run AdmTool
    - boot DSPs
    - select HELLOWORLD from SAPS list
    - select DSPs to run the application
    - select Start SAPS
    - click on OK

After seeing the line "HELLOWORLD" on active SAPS window you will see output on the monitor attached to the DSP. (Bear in mind that a monitor has been attached to a DSP card which has two DSPs. If you run the application on two DSPs located on the same card concurrently , you might experience the overwritten output which seems like garbage. This is because of the existence of just one output buffer for one card. There is no signaling mechanism to prevent concurrent access to this buffer.)

```
user_a@ankara> rsh topaz DOS  -display act_mach:0.0
Remote program run successfully
user_a@ankara>
[A new DOS window will appear. Switch to this DOS window.]
C:\USERS\USER_A>edit hello.c  [vi is available on PC as well]
[Write the program given at Fig. 7.1. Save and leave the editor.]
C:\USERS\USER_A>dir
Volume in drive C is MSDOS 5.00
Volume Serial Number is 1EC5-7A0C
Directory of C:\USERS\USER_A
.                       <DIR>                   16/05/96        15:00
..                      <DIR>                   12/01/96        19:20
INCLUDE                 <DIR>                   16/05/96        15:00
LIB                     <DIR>                   16/05/96        15:01
LIB_SRC                 <DIR>                   16/05/96        15:02
INCLUDE                 <DIR>                   16/05/96        15:02
MAKE            BAT                  3288       16/05/96        15:03
HELLO           C                     525       16/05/96        16:11
       8 file(s)              3813  bytes
                          14598144  bytes free
C:\USERS\USER_A>make x hello.c
compiling...
linking for DSPb...
hex file is being produced...
linking for DSPa...
hex file is being produced...
make.bat has successfully finished.
C:\USERS\USER_A>dir
Volume in drive C is MSDOS 5.00
Volume Serial Number is 1EC5-7A0C
Directory of C:\USERS\USER_A
.                       <DIR>                   16/05/96        15:00
..                      <DIR>                   12/01/96        19:20
INCLUDE                 <DIR>                   16/05/96        15:00
LIB                     <DIR>                   16/05/96        15:01
LIB_SRC                 <DIR>                   16/05/96        15:02
INCLUDE                 <DIR>                   16/05/96        15:02
MAKE            BAT                  3288       16/05/96        15:03
HELLO           C                     525       16/05/96        16:11
HELLOB          HEX                 33094       16/05/96        16:13
HELLOA          HEX                 33094       16/05/96        16:14
       10 file(s)             70001  bytes
                          14531956  bytes free
C:\USERS\USER_A> ftp kiri
Connected to kiri
220 OS-9 ftp server V1.0 ready
Name (kiri:user_a):super
331 password required for super
Password:user
230 user super logged in
ftp> cd /dd/monalisa/saps
200 CWD command ok
ftp>put helloa.hex
local: helloa.hex remote: helloa.hex
200 PORT command ok
150 Opening data connection for helloa.hex (192.135.231.9,1828)
226 Transfer complete
33099 bytes send in 0.5 seconds (64.64 Kbytes/s)
ftp>put hellob.hex
local: hellob.hex remote: hellob.hex
200 PORT command ok
150 Opening data connection for hellob.hex (192.135.231.9,1830)
226 Transfer complete
33099 bytes send in 0.5 seconds (64.64 Kbytes/s)
ftp>quit
C:\USERS\USER_A>
```

Figure 7.2. A typical flow of compiling a SAPS and uploading it on PVA-Host

7.2 The use of locator files

Since there is no memory mapping hardware support on the DBV96 processor boards memory mapping is achieved statically at linking time. The two locator files given in tables above have been developed to implement a particular memory mapping which allow the sharing of the physical on-board memory. The user should not modify these files except for special cases. However an understanding of the memory space utilisation is necessary for debugging purposes.

Fig. 7.3 shows the processor board configuration considering memory and bus configuration. There are two local SRAM banks (each of 512K Words) assigned to each on-board processor. These are primarily used as program memory. Some WKernel management data is also stored here. There is also a global SRAM bank of 512K words which is shared by the two processors.

This bank is used in combination with the local banks to provide 64-bit wide memory for Stack and Data Storage. On each DVB96 processor board there is also a DRAM bank of 4MWords that is shared by the two on-board processors.

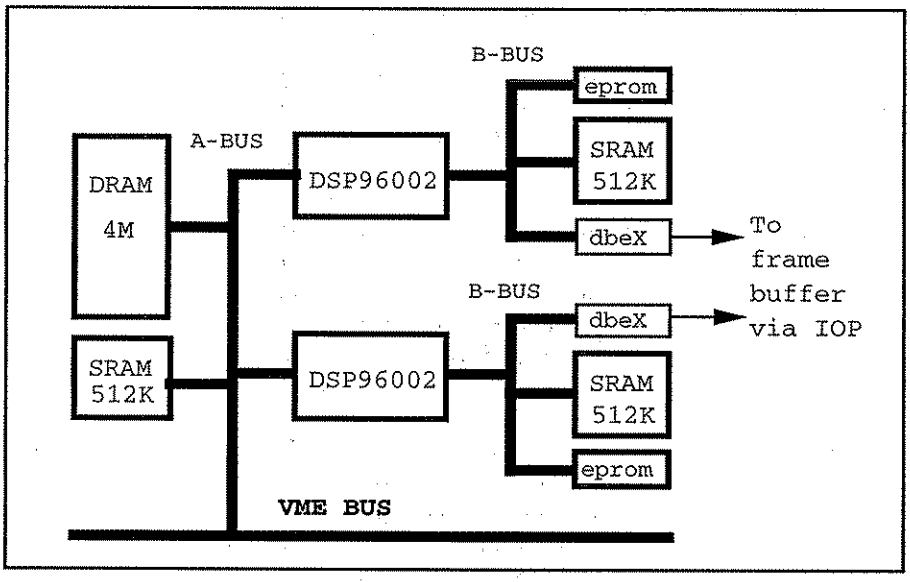

Figure 7.3. Processor board memory/bus configuration

The DRAM is mapped between the addresses 0x20000000 and 0x20400000. Type A DSPs use upto 0x20200000 and the rest is used by the type B DSPs. DRAM is used for storing uninitialised data and the heap for each processor. (Table 7.1). SRAM banks are mapped from 0x00000 to 0x40000. (Table 7.2) Note that memory usage by DSPs is word oriented that is individual bytes within a single word are not accessible directly.

Memory Address		GLOBAL DRAM	GLOBAL DRAM
From	To	(DSPA) Y:	(DSPB) Y:
0x20200000	0x203FFFFF	----RESERVED----	-- DATA AREA --
0x20000000	0x201FFFFF	-- DATA AREA --	----RESERVED----

Table 7.1 Memory map of Y: 0x20000000 to 0x203FFFFF

Memory Address		LOCAL SRAM	GLOBAL SRAM	LOCAL SRAM
From	To	(DSPA) X: P:	(DSPA & DSPB) Y:	(DSPB) X: P:
0x3EC00	0x3FFFF	----RESERVED----	----RESERVED----	----RESERVED----
0x3CC00	0x3EBFF	----RESERVED----	SLOT3 STACK for DSPB	
0x3AC00	0x3CBFF		SLOT2 STACK for DSPB	
0x38C00	0x3ABFF		SLOT1 STACK for DSPB	
0x36C00	0x38BFF	SLOT3 STACK for DSPA		----RESERVED----
0x34C00	0x36BFF	SLOT2 STACK for DSPA		
0x32C00	0x34BFF	SLOT1 STACK for DSPA		
0x32A00	0x32BFF	----RESERVED----	SLOT3 DATA AREA for DSPB	
0x32800	0x39FFF		SLOT2 DATA AREA for DSPB	
0x32600	0x327FF		SLOT1 DATA AREA for DSPB	
0x32400	0x325FF	SLOT3 DATA AREA for DSPA		----RESERVED----
0x32200	0x323FF	SLOT2 DATA AREA for DSPA		
0x32000	0x321FF	SLOT1 DATA AREA for DSPA		
0x30001	0x31FFF	Slot1Int_HandlerText	----RESERVED----	Slot1Int_HandlerText
0x30000		Slot1Int_HandlerFlag		Slot1Int_HandlerFlag
0x22000	0x2FFFF	Slot3 Text (DSPA)		Slot3 Text (DSPB)
0x12000	0x21FFF	Slot2 Text (DSPA)		Slot2 Text (DSPB)
0x02800	0x11FFF	Slot1 Text (DSPA)		Slot1 Text (DSPB)
0x02000	0x027FF		Message Buffers	
0x01901	0x01FFF	----RESERVED----		----RESERVED----
0x01900			Message Count	
0x0180B	0x018FF		Slot Table Area	
0x0180A			Active Slot	
0x01809			SSW(SignalStatusWord)	
0x01803	0x01808		----RESERVED----	
0x01802			Interrupt Count	
0x01801			Dbex in use flag	
0x01800			PSW(ProcessorStatW)	
0x01560	0x017FF	WKernel Text	----RESERVED----	WKernel Text
0x01540	0x0155F	SYSCALL Vector		SYSCALL Vector
0x01520	0x0153F	SOFT-INT Vector		SOFT-INT Vector
0x01500	0x0151F	Exception Vector		Exception Vector
0x00901	0x014FF	WKernel Error Buf.	Message Buffers	WKernel Error Buf.
0x00900		WKernel Error Cnt.	Message Count	WKernel Error Cnt.
0x0082B	0x008FF	----RESERVED----	----RESERVED----	----RESERVED----
0x0080B	0x0082A		Slot Table Area	
0x0080A			Active Slot	
0x00809			SSW(SignalStatusWord)	
0x00803	0x00808		----RESERVED----	
0x00802		Slot1 Cont. Addr.	Interrupt Count	Slot1 Cont. Addr.
0x00801		Slot1 Wait Flag	Dbex in use flag	Slot1 Wait Flag
0x00800		----RESERVED----	PSW(ProcessorStatW)	----RESERVED----
0x00400	0x007FF	ROM	----RESERVED----	ROM
0x00200	0x003ff	----RESERVED----	----RESERVED----	----RESERVED----
0x00000	0x001FF	Internal RAM	Internal RAM	Internal RAM

Table 7.2 Memory map of P:, X:, and Y: 0x00000 to 0x3FFFF

A user may need to determine the physical addresses of data structures. This kind of manipulation is available via locator files. The following example is to explain the use of locator files for such a purpose. This example code stays on the PC under C:\USERS\MAP. Locator files and makefile are modified for this specific example. There are two files to be compiled; **`map_1.c`** and **`map_2.c`**. The second one, having **`pragma`** declarations in it, is the code determining the physical addresses of two arrays on the memory explicitly. The first one has no explicit declaration and allows compiler to

determine the addresses. By comparing the map files produced after compiling user would understand the use of locator files just by seeking through the differences.

For the sake of simplicity one type of locator files exist under LOCATOR directory both of which has application specific allocations. As map_1.c will be compiled using this locator file, user will see a warning message indicating to ignore the two lines in the locator file. If another locator file having no application specific allocation definitions is used, there will be no warning messages. The source codes of two files are listed in Fig. 7.4 and Fig. 7.5

```
/*************************************************************/
/*  map_1.c  one of the two files to examine the use of      */
/*           memory mapping via locator files (the other     */
/*           one is map_2.c having pragma definitions )      */
/*                                                           */
/*  960530   DTA first implementation                        */
/*************************************************************/
float first_array[256];
float second_array[256];

void main()
{
int i;

  WKprintf(0,"DSP #%d running\n",WKmyid());
  for(i=0;i<256;i++) first_array[i]=i*0.5;
  for(i=0;i<256;i++) second_array[i]=first_array[i]*first_array[i];
  WKexit();
}
```

Figure 7.4. map_1.c code

```
/*************************************************************/
/*  map_2.c  one of the two files to examine the use of      */
/*           memory mapping via locator files (the other     */
/*           one is map_1.c having no pragma definitions)    */
/*                                                           */
/*  960530   DTA first implementation                        */
/*************************************************************/
#pragma sep_on
float first_array[256];
float second_array[256];
#pragma sep_off

void main()
{
int i;

  WKprintf(0,"DSP #%d running\n",WKmyid());
  for(i=0;i<256;i++) first_array[i]=i*0.5;
  for(i=0;i<256;i++) second_array[i]=first_array[i]*first_array[i];
  WKexit();
}
```

Figure 7.5. map_2.c code

```
-- ****************************************************************
-- *                                                              *
-- * map_a.lc      Locator file for mapping exercise              *
-- *                                                              *
-- * 951108    DTA Hardware conf:                                 *
-- *                                p with port B                 *
-- *                                x with port B                 *
-- *                                y with port A                 *
--****************************************************************

-- ================================================================
-- general configuration
-- ================================================================
reserve ( xi : 0 to #200 ) ;       -- Reserve X internal memory.
reserve ( yi : 0 to #200 ) ;       -- Reserve Y internal memory.
reserve ( pi : before #400 ) ;     -- Reserve low internal P memory

reserve ( pa : #0 to #FFFFFFFF ); -- These addresses are not available.
reserve ( xa : #0 to #FFFFFFFF ); -- These addresses are not available.
reserve ( yb : #0 to #FFFFFFFF ); -- These addresses are not available.


-- ================================================================
-- pb configuration
-- ================================================================
reserve ( pb :         #0 to      #400); -- internal.
reserve ( pb :      #400 to     #1500); -- Reserved+ WK Error
reserve ( pb :     #1500 to     #2000); -- Kernel
--      ( pb :     #2000 to    #12000); ** Slot1= 2000 - 12000
reserve ( pb :    #12000 to #FFFFFFFF);


-- ================================================================
-- ya configuration
-- ================================================================
reserve ( ya :         #0 to      #800);
reserve ( ya :      #800 to     #1800); -- DSPA kernel data
reserve ( ya :     #1800 to     #2800); -- DSPB kernel data
--      ( ya :     #2800 to    #32000); ** Application SRAM
--      ( ya :    #32000 to    #32200); ** slot1 data 0x32000-0x32200
reserve ( ya :    #32200 to #20000000);
--      ( ya : #20000000 to #20200000); ** DSP A private DRAM mem.
reserve ( ya : #20200000 to #20400000); -- DSP B private DRAM mem.
reserve ( ya : #20400000 to #FFFFFFFF); -- empty

-- ================================================================
-- xb configuration
-- ================================================================
reserve ( xb    :     #0 to      #800); -- internal,reserved,rom
reserve ( xb    :   #800 to    #32000);
--      ( xb :    #32000 to    #32200); ** slot1 data 0x32000-0x32200
reserve ( xb :    #32200 to #FFFFFFFF);

-- ================================================================
-- location directives
-- ================================================================
locate ( init_PB : b : #2000);
locate ( {code} : after #2000);
locate ( {constant} {} {data} : after #20000000);

-- ================================================================
-- Application Specific
-- ================================================================
locate ( SYfirst_array_Y   : a: #20001000);
locate ( SYsecond_array_Y  : a: #20101000);
```

Figure 7.6. map_a.lc : A locator file for DSPA with application specific definitions

The only difference between these two files are pragma declarations. Two arrays are defined between #pragma sep_on and #pragma sep_off in map_2.c. This declaration determines the physical addresses on the memory in correspondence with locator files. A locator file for DSPA is given in Fig. 7.6. See locator file for DSPB on the disk to realise the differences, Note that this locator files reflect the information given in Table 7.1 and Table 7.2.

A user would better compile the two programs and debug via AdmTool. Steps to be taken to compile and to download is previously explained (refer to Fig.7.2 for details). A user should run make.bat both for map_1 and map_2. The makefile located under c:\users\map is modified to use local locator files and to keep the map files after compiling. A user could investigate the differences between these two .map files. Parts of three of these map files, indicating the difference are in Fig. 7.7, Fig. 7.8, and Fig. 7.9.

```
Symbol Map for map_1.ab                          May 31 1996  14:18:29      Page 1

Translator : llink
Target     : 96000

Global          Address

EXIT            00000010 (16)
FWKexit         00002bb2 (11186) PB
FWKmyid         00002bae (11182) PB
TIME            00000013 (19)
Y_ctype         00032608 (206344) YA
Y_digits        20200047 (538968135) YA
Y_ldigits       20200025 (538968101) YA
Y_udigits       20200036 (538968118) YA
Yfirst_array    2020004b (538968139) YA
Ysecond_array
                2020014b (538968395) YA
Ystderr         2020004a (538968138) YA
Ystdin          20200048 (538968136) YA
Ystdout         20200049 (538968137) YA


Symbol Map for map_1.ab                          May 31 1996  14:18:29      Page 2

Segment         Address                 Length          Class     Align  Combine
Type

SWKexit_PB      00002bb2 (11186)        000002 (2)      code      word   private
PB`
SWKmyid_P       00002bae (11182)        000004 (4)      code      word   private
PB`
Sstrncpy_P      00002028 (8232)         00001e (30)     code      word   private
PB`
idata_Y         20200020 (538968096)    00002b (43)     data      word   private
YA`
ildata_L        00032600 (206336)       000008 (8)      l_data    word   private
LBA
init_PB         00002000 (8192)         000004 (4)      code      word   private
PB`
libcode_P       000022e4 (8932)         0000db (219)    code      word   private
PB`
sdata_Y         20200000 (538968064)    000020 (32)     constant  word   private
YA`
udata_Y         2020004b (538968139)    0003be (958)    data      word   private
YA`

Statistics

Segments    : 20
Globals     : 48
Code  Size  : 00000c0e (3086)
Data  Size  : 000003f1 (1009)

User Start Address = #2000
```

Figure 7.7. map_1b.map : Map file produced by compiling map_1.c for DSPB

```
Symbol Map for map_1.ab                    May 31 1996  14:19:46      Page 1

Translator : llink
Target     : 96000

Global          Address

EXIT            00000010 (16)
FWKexit         00002bb2 (11186) PB
FWKmyid         00002bae (11182) PB
TIME            00000013 (19)
Y_ctype         00032008 (204808) YA
Y_digits        20000047 (536870983) YA
Y_ldigits       20000025 (536870949) YA
Y_udigits       20000036 (536870966) YA
Yfirst_array    2000004b (536870987) YA
Ysecond_array
                2000014b (536871243) YA
Ystderr         2000004a (536870986) YA
Ystdin          20000048 (536870984) YA
Ystdout         20000049 (536870985) YA

Symbol Map for map_1.ab                    May 31 1996  14:19:46      Page 2

Segment         Address                 Length          Class     Align  Combine
Type

SWKexit_PB      00002bb2 (11186)        000002 (2)      code      word   private
PB`
SWKmyid_P       00002bae (11182)        000004 (4)      code      word   private
PB`
Sstrncpy_P      00002028 (8232)         00001e (30)     code      word   private
PB`
idata_Y         20000020 (536870944)    00002b (43)     data      word   private
YA`
ildata_L        00032000 (204800)       000008 (8)      l_data    word   private
LBA
init_PB         00002000 (8192)         000004 (4)      code      word   private
PB`
libcode_P       000022e4 (8932)         0000db (219)    code      word   private
PB`
sdata_Y         20000000 (536870912)    000020 (32)     constant  word   private
YA`
udata_Y         2000004b (536870987)    0003be (958)    data      word   private
YA`

Statistics

Segments    : 20
Globals     : 48
Code  Size  : 00000c0e (3086)
Data  Size  : 000003f1 (1009)

User Start Address = #2000
```

Figure 7.8. map_1a.map : Map file produced by compiling map_1.c for DSPA

The comparison of **map_1a.map** (Fig. 7.7) to **map_1b.map** (Fig. 7.8) gives the differences between two different type of DSPs (DSPAs and DSPBs or even numbered DSPs and odd numbered DSPs) by means of memory mappings. User will recognise that the difference between these two files comes from the offset differences of DRAMS.

The comparison of **map_1a.map** (Fig. 7.7) to **map_2a.map** (Fig. 7.9) shows that the address of data structure could be forced to a specific value via locator files. Address of first_array is 0x2000004b in map_1a.map where it is 0x20001000 as forced by pragma declaration in map_2a.map.

User could examine the location of these arrays via AdmTool's debug facility.

```
Symbol Map for map_2.ab                    May 31 1996  14:37:05     Page 1

Translator : llink
Target     : 96000

Global       Address

EXIT         00000010 (16)
FWKexit      00002bb2 (11186) PB
FWKmyid      00002bae (11182) PB
FWKprintf    00002bc5 (11205) PB
=============================================================================
Y_ldigits    20000025 (536870949) YA
Y_udigits    20000036 (536870966) YA
Yfirst_array 20001000 (536875008) YA
Ysecond_array
             20101000 (537923584) YA
Ystderr      2000004a (536870986) YA
Ystdin       20000048 (536870984) YA
Ystdout      20000049 (536870985) YA
Symbol Map for map_2.ab                    May 31 1996  14:37:05     Page 2

Segment      Address              Length         Class    Align  Combine
Type

SWKexit_PB   00002bb2 (11186)     000002 (2)     code     word   private
PB`
SWKmyid_P    00002bae (11182)     000004 (4)     code     word   private
PB`
SWKprintf_P  00002bc5 (11205)     00001e (30)    code     word   private
PB`
SY_ctype_Y   00032008 (204808)    000101 (257)   isep     word   private
YA`
SYfirst_array_Y
             20001000 (536875008) 000100 (256)   usep     word   private
YA`
SYsecond_array_Y
             20101000 (537923584) 000100 (256)   usep     word   private
YA`
S_cppad_P    000023bf (9151)      0007ef (2031)  code     word   private
PB`
=============================================================================
data_Y       20000000 (536870912) 000020 (32)    constant word   private
YA`
udata_Y      2000004b (536870987) 0001be (446)   data     word   private
YA`

Statistics

Segments   : 22
Globals    : 48
Code  Size : 00000c0e (3086)
Data  Size : 000001f1 (497)

User Start Address = #2000
```

Figure 7.9. map_2a.map : Map file produced by compiling map_2.c for DSPA

7.3 Accessing Frame Buffer from System Host

The Host Computer Software provides some functions used to set up the software and connect/disconnect to/from the ISP System. The following example in Fig. 7.10 shows the skeleton structure of an application that uses VDLIB functions.

`ispini()` initialises some variables used by VDLIB routines, **`ispdefcon()`** defines a connection the ISP System (contacting the ISP daemon running under OS9 which creates then an ISP Server used

by the application for controlling devices and transferring data). Then all control and transfer operations can be performed by the single routine **vdlib()**. At the end of the application, the connection to the ISP System has to be closed, this is done by **ispend()** and **ispdefdiscon()**. Disconnecting from the ISP System is very important at the end of an application (to avoid leaving zombie processes on the ISP), this is why it is advised to install a signal handler in the user application, so that the program will catch the and disconnects itself from the ISP System.

```
/*****************************************************************/
/*   vdlib_template.c   This is a template source code using     */
/*                      vdlib functions                          */
/*   960212  DTA                                                 */
/*****************************************************************/

#include <stdio.h>
#include <signal.h>
#include <ispudef.h>

int id;

signalhandler(int signum) /* signal handler */
{
  extern int id;

  signal(signum,signalhandler);
  printf("Signal %d has been received. Terminating program...\n",signum);
   if(ispend()==-1) printf(abnormal termination : ispend()");
  else printf("ispend() is successfull.\n");
  if(ispdefdiscon(id)==-1) printf("abnormal termination:ispdefdiscon()");
  else printf("ispdefdiscon() is successfull.\n");
  exit(-1);
}

main()
{

int i;

/* standard UNIX signals to be handled */
  for(i=1;i<18;i++)  signal(i,signalhandler);

/* isp communication initialisation */
  if(ispini()==-1) {
    prinf("ispini() : ISP not initialised !\n");
    exit(-2);
    }
  if((id=ispdefcon())==-1){
    printf("ispdefcon() : ISP connection failed !\n");
    exit(-3);
    }
  else printf("ISP Connection Initialised id = %d\n", id);


/* User special VDLIB commands should be inserted here */


/* De-initialise and Disconnect */
  if(ispend()==-1) printf(abnormal termination : ispend()");
  else printf("ispend() is successful. \n");
  if(ispdefdiscon(id)==-1) printf("abnormal termination:ispdefdiscon()");
  else printf("ispdefdiscon() is successful. \n");
  exit(0);
}
```

Figure 7.10. A template program for using VDLIB functions.

The Video Device Library (VDLIB) is a subroutine package used for controlling a video input/output device and transferring data between the frame buffer and the data arrays in the host memory. These operations are all performed by the subroutine **vdlib()** which takes a function code which determines the actual function of the command (see **ispudef.h** under **/usr/gfx_host/ISP/defs** for a list

of the function codes). The first parameter being passed to the function defines the task , the latter ones are to quantify this task. The vdlib() routine should return SUCCESS (defined in ispudef.h as 1) if the command has been successfully executed by the ISP System. If function fails, it returns 0.

A number of vdlib() call using different parameters few calls with different tasks and parameters. Try to establish connections between these codes and ispmain commands.

- To initialise the device 0 for blank and white 720x576 images:

```
vdlib(CF_DEV,0);                                    /* select
                                                          device */
vdlib(CF_INIT,0,25,2,625,27000000,720,576) /* initialise
                                                          device */
```

- To creating the object "testseq" of 25 frames:

```
vdlib(CF_BUIL,"testseq",25,720,576)      /* build file */
```

To use the frame buffer as a sort of multi-purpose image storage, two routines were developed for transferring any type of data between a frame buffer object (ISP file) and an array in the host computer memory.
These are:

- **fbwrite(array, n)** for writing **n** bytes from the buffer **array** into the currently attached object,
- **fbread(array, n)** for reading **n** bytes from the currently attached object into the buffer **array** in the host memory.

The following example creates two objects and loads 1000 float values (held in buffer) into the first created object (after attachment):

```
vdlib(CF_BUIL,"testseq1",25,720,576);
vdlib(CF_BUIL,"testseq2",10,720,288);
vdlib(CF_DETA);
vdlib(CF_ATTA,"testseq1",3);
fbwrite(buffer, 1000*sizeof(float));
```

- To read 5000 integers from an object "obj_int":

```
vdlib(CF_ATTA,"obj_int",3);
fbread(buffer, 5000*sizeof(int));
```

Those two functions work with different types of data such as: single variable, arrays and structures but not lists as they used pointers which are of course irrelevant in the frame buffer. Furthermore, when one wants to transfer several types of data (e.g. an integer and an array), it may be worth to create only a single object in the frame buffer and thus group these data into a single structure that is then transferred into the object. But an object for each data is also possible.

The following example shows how to transfer a structure into an object:

```
typedef struct {
  int   id;
  char  name[12];
  } tobj;
      .
      :

tobj        object;
      .
      :
object.id = 1;
strcpy(object.name,"image");
      .
```

```
        :
fbwrite(&object, sizeof(tobj));
```

As one could realise fbread and fbwrite commands have been developed using VDLIB functions. Actually instead of using fbwrite the following code could be used.

```
vdlib(CF_SIOC,0);
vdlib(CF_SIOP,0);
vdlib(CF_SIOD,0);
for(j=0;j<576;j+=72){
   vdlib(CF_SIOS,0,j,720,72);
   vdlib(CF_CPUW,buf);
   buf+=720*72;
   }
```

Note that a frame of 720x576 is being sent to frame buffer after dividing into ten blocks. At such a low level user should be aware of the physical constraints as well. The number of bytes permitted to be transferred is dependent on the transfer device and has been defined as MAXTRANS in ispudef.h. The typical value for the data size through Ethernet is 65024 bytes. (65536-512=65024)

The ISP software is available in the library **isp.a** under the directory **/usr/gfx_host/ISP**. This must be included in any Makefiles of any applications for the ISP System. This include option must also include the directories that contain the include files such as **ispudef.h** which defines the command function codes

An example using vdlib() functions has been coded. This example consists of three files; **fbwrite.c** (Fig. 7.11), **fbread.c** (Fig. 7.12), and **fbremove.c** (Fig. 7.13). **fbwrite.c** creates two files, namely input and output on the frame buffer. input is initialised by the multiples of a given float number (default value is 1.25). All elements of output are initialised by 3.0. **fbread.c** reads the file given at the command line and prints values on the screen. And finally, **fbremove.c** is to remove the given file on the frame buffer. It should be preferred to use a makefile to compile these three files using the given library and related include files. (Fig. 7.14)

The existence of the files on the frame buffer could be seen through an OS9 shell Frame buffer is treated as a file system having just one level directory. It is mounted to PVA-Host as **/y0**. **dir -e /y0** on PVA-Host will list the files on the frame buffer.

```
/*********************************************************************/
/* fbwrite.c    creates two files "input","output" and initialise     */
/* 950619 DTA   and initialises them                                 */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <ispudef.h>

int id;

void signalhandler(int signum)
{
  if(ispend()==-1) fprintf(stdout,"abnormal termination : ispend()");
  if(ispdefdiscon(id)==-1) fprintf(stdout,"abnormal termination:ispdefdiscon()");
  if (signum==16) exit(0) else exit(-1);
}

void fatalerror(str)
char *str;
{
  fprintf(stderr,"Fatal error : %s\n",str);
  signalhandler(17);
}

void main(count,arg)
int  count;
char **arg;
{
float   buffer[100];
float  val=1.25;
int     status,i;

/* init the value of val if given */
  if (count==2) val=atof(arg[1]);
  printf("val:%5.2f\n",val);

/* signal capture*/
  for(i=1;i<18;i++)  signal(i,signalhandler);
  for(i=23;i<25;i++) signal(i,signalhandler);
  for(i=26;i<32;i++) signal(i,signalhandler);

/* isp communication stuff*/
  if(ispini()==-1)fatalerror("ispini()");
  if((id=ispdefcon())==-1)fatalerror("ispdefcon()");

/* Codec Initialisation */
  if((status=vdlib(CF_SDEV,0))!=1) fatalerror("CF_SDEV");
  if((status=vdlib(CF_INIT,0,25,2,625,27000000,400,1))!=1) fatalerror("CF_INIT");

/* Create two new files on the Frame Buffer */
  if((status=vdlib(CF_BUIL, "input", 1,400,1))!=1) fatalerror("CF_BUILD");
  if((status=vdlib(CF_BUIL, "output", 1,400,1))!=1) fatalerror("CF_BUILD");

/* Attach a file as the work file and write data and detach */
  if(status=vdlib(CF_ATTA,"input",3))!=1) fatalerror("CF_ATTA");
  for (i=0; i<100; i++) buffer[i] = i*val;
  if((status=fbwrite((char*)buffer, 100*sizeof(float)))!=1)fatalerror("fbwrite");
  if((status=vdlib(CF_DETA))!=1) fatalerror("CF_DETA");

/* Attach a file as the work file and write data and detach */
  if((status=vdlib(CF_ATTA, "output",3))!=1)fatalerror("CF_ATTA");
  for (i=0; i<100; i++) buffer[i] = 3.0;
  if((status=fbwrite((char*)buffer, 100*sizeof(float)))!=1)fatalerror("fbwrite");
  if((status=vdlib(CF_DETA))!=1) fatalerror("CF_DETA");

/* Deinitalise and Disconnect */
  signalhandler(16);
}
```

Figure 7.11. fbwrite.c

```
/*********************************************************************/
/* fbread.c     Reads the given file and displays on the screen.     */
/* 950619 DTA   All isp library functions' return values are handled */
/*              All signals which cause exit or core handled         */
/*********************************************************************/
#include <stdio.h>
#include <signal.h>
#include <ispudef.h>

int id;

void signalhandler(int signum)
{
  signal(signum,signalhandler);
  if(ispend()==-1) fprintf(stdout,"abnormal termination : ispend()");
  if(ispdefdiscon(id)==-1)fprintf(stdout,"abnormal termination:ispdefdiscon()");
  if (signum==16) exit(0) else exit(-1);
}

void fatalerror(str)
char *str;
{
  fprintf(stderr,"Fatal error : %s\n",str);
  signalhandler(17);
}

void main(argc, argv)
int     argc;
char    **argv;
{
float   buffer[100];
int     i;
int     status;

extern void signalhandler();

  if(argc<2){printf("usage: fbread filename\n"); exit(-1);}

/* signal capture*/
  for(i=1;i<18;i++)  signal(i,signalhandler);
  for(i=23;i<25;i++) signal(i,signalhandler);
  for(i=26;i<32;i++) signal(i,signalhandler);

/* isp communication stuff*/
  if(ispini()==-1)fatalerror("ispini()");
  if((id=ispdefcon())==-1)fatalerror("ispdefcon()");
  else fprintf(stdout,"ISP Connection Initialised id = %d\n", id);

/* Codec Initialisation */
  if((status=vdlib(CF_SDEV,0))!=1)fatalerror("error vdlib(CF_SDEV,0);\n");
  if((status=vdlib(CF_INIT,0,25,2,625,27000000,400,1))!=1)fatalerror("CF_INIT");

/* Attach a file as the work file */
  if((status=vdlib(CF_ATTA, argv[1] ,3))!=1) fatalerror("CF_ATTA");

/* READ Data */
  if((status=fbread(buffer, 100*sizeof(float)))!=1) fatalerror("fbread");
  for (i=0; i<100; i++) fprintf(stdout,"%d = %f\t",i,buffer[i]);
  fprintf(stdout,"\n");

/* Detach */
 if((status=vdlib(CF_DETA))!=1)fatalerror("CF_DETA");

/* Deinitalise and Disconnect */
  signalhandler(16);
}
```

Figure 7.12. fbread.c

```
/*************************************************************************/
/* fbremove.c   removes the given file off the frame buffer              */
/* 950619 DTA                                                            */
/*************************************************************************/
#include <stdio.h>
#include <signal.h>
#include <ispudef.h>

int id;

void signalhandler(int signum)
{
  signal(signum,signalhandler);
  if(ispend()==-1) fprintf(stdout,"abnormal termination : ispend()");
  if(ispdefdiscon(id)==-1) fprintf(stdout,"abnormal termination ispdefdiscon()");
  if (signum==16) exit(0) else exit(-1);
}

void fatalerror(str)
char *str;
{
  fprintf(stderr,"Fatal error : %s\n",str);
  signalhandler(17);
}


void main(argc, argv)
int     argc;
char    **argv;
{
float   buffer[100];
int     i;
int     status;

extern void signalhandler();

  if(argc<2){ printf("usage fbremove filename\n"); exit(1); }

/* signal capture*/
  for(i=1;i<18;i++)  signal(i,signalhandler);
  for(i=23;i<25;i++) signal(i,signalhandler);
  for(i=26;i<32;i++) signal(i,signalhandler);

/* isp communication stuff*/
  if(ispini()==-1)fatalerror("ispini()");
  if((id=ispdefcon())==-1)fatalerror("ispdefcon()");
  else fprintf(stdout,"ISP Connection Initialised id = %d\n", id);

/* Codec Initialisation */
  if((status=vdlib(CF_SDEV,0))!=1)fprintf(stdout,"error vdlib(CF_SDEV,0);\n");
  if((status=vdlib(CF_INIT,0,25,2,625,27000000,100,100))!=1)fatalerror("CF_INIT");

/* Attach a file as the work file */
  if((status=vdlib(CF_ATTA, argv[1] ,3))!=1) fatalerror("CF_ATTA");

/* Remove the file */
  if((status=vdlib(CF_REMO))!=1) fatalerror("CF_REMO");
  fprintf(stdout,"<%s> is removed\n",argv[1]);

/* Deinitalise and Disconnect */
  signalhandler(16);
}
```

Figure 7.13. fbremove.c

```
# Makefile to compile vdlib based applications
# 951120 DTA

CC = cc
LIBS =/usr/monalisa/Project/GFXHost/ISP/isp.a

CFLAGS = -traditional -O2 -DLSIBOARDSYSTEM\
         -I/usr/include\
               -I/usr/monalisa/Project/GFXHost/ISP/defs

OBJS1 = fbwrite.o
OBJS2 = fbread.o
OBJS3 = fbremove.o

programs=fbwrite fbread fbremove

.c.o:
          $(CC) -c $(CFLAGS) $<

fbwrite :  $(OBJS1)  $(LIBS)
          $(CC) -o fbwrite $(OBJS1) $(LIBS)

fbread  :  $(OBJS2) $(LIBS)
          $(CC) -o fbread $(OBJS2) $(LIBS)

fbremove:  $(OBJS3) $(LIBS)
          $(CC) -o fbremove $(OBJS3) $(LIBS)
```

Figure 7.14. A Makefile for vdlib based applications (compiles the previously given three files)

As DSP programs also could access the frame buffer, frame buffer becomes a common area (physically shared memory, structurally shared file system) being used by both DSPs and workstation which potentially provides data transfer between the two platforms. In order to have an application in which data primarily send by workstation, only DSP program is missing considering the previously given three programs on workstation side.

The sample program, **fbrw.c** (Fig. 7.15), to be run on DSPs could be found under c:\users\fbaccess on the PC. Compile this program and upload to /dd/monalisa/SAPS directory on PVA-Host. Add a line for this new program into SAPS file under /usr/pva_adm on the workstation. Then follow the given directives:

- check out the /y0 directory on PVA-host by typing **dir -e .y0**. See that there is no file named either **input** or **output**. (Delete by typing the command **del /y0/input** and/or **del /y0/output** if exists to see fbwrite creates the file.)
- run **fbwrite** on the workstation
- check out /y0 to see whether files are created
- run **fbread input** to see that file **input** is initialised by the multiples of 1.25 in ascending order
- run **fbread output** to see that file **output** is initialised by 3.0
- run **AdmTool**
  - Boot processor N
  - Activate **fbrw** on Nth processor

(While it is running, it display the values of input and new values assigned to output by DSP on the monitor attached to Nth processor)

- run **fbread output** to see the recently assigned values
- back to **AdmTool** to kill the server
- run **fbremove** to erase the files on the frame buffer.
- check out /y0 to see whether files are removed

```
/*************************************************************************/
/*                                                                       */
/* fbrw.c      Reads data from frame buffer and writes back to another   */
/*             file after computation.                                   */
/*                                                                       */
/* 960505   DTA                                                          */
/*************************************************************************/


float  array1[100], array2[100];

main()
{
int    status, i;
float  *fp1,*fp2;

  if((status=WKgetdata("input",(char *)array1,100*sizeof(float)))<0){
    WKprintf(0,"WKgetdata() failed !\n");
    WKexit();
    }
  WKprintf(0,"WKgetdata() succeeded !\n");

  fp1 = array1;
  fp2 = array2;
  for(i=0; i<100; i++,fp1++,fp2++){
    *fp2 = *fp1 + 0.025;
    WKprintf(0,"%d = %f -> %f\n",i,*fp1,*fp2);
    }

  if((status=WKputdata("output",(char *)array2,100*sizeof(float)))<0){
    WKprintf(0,"WKputdata() failed :status=%d\n",status);
    WKexit();
    }
  WKprintf(0,"WKputdata() succeeded !\n");
}
```

Figure 7.15. fbrw.c; a DSP program accessing files on the frame buffer

7.4 Writing a client-server program

A client server program runs over all three platforms. Although a programmer would not need to develop any program on PVA-Host, PVA-Host takes the responsibility of communicating with System-Host and DSPs, and dispatching the tasks.

The actual implementation of a client program based upon an RPC command to PVA-Host. Before calling this RPC command, which is sending tasks to pool manager to be dispatched, client is responsible for sending the data and all relevant information to the frame buffer.

The basic protocol for a client-server based programming over this system is given in Fig 7.16.

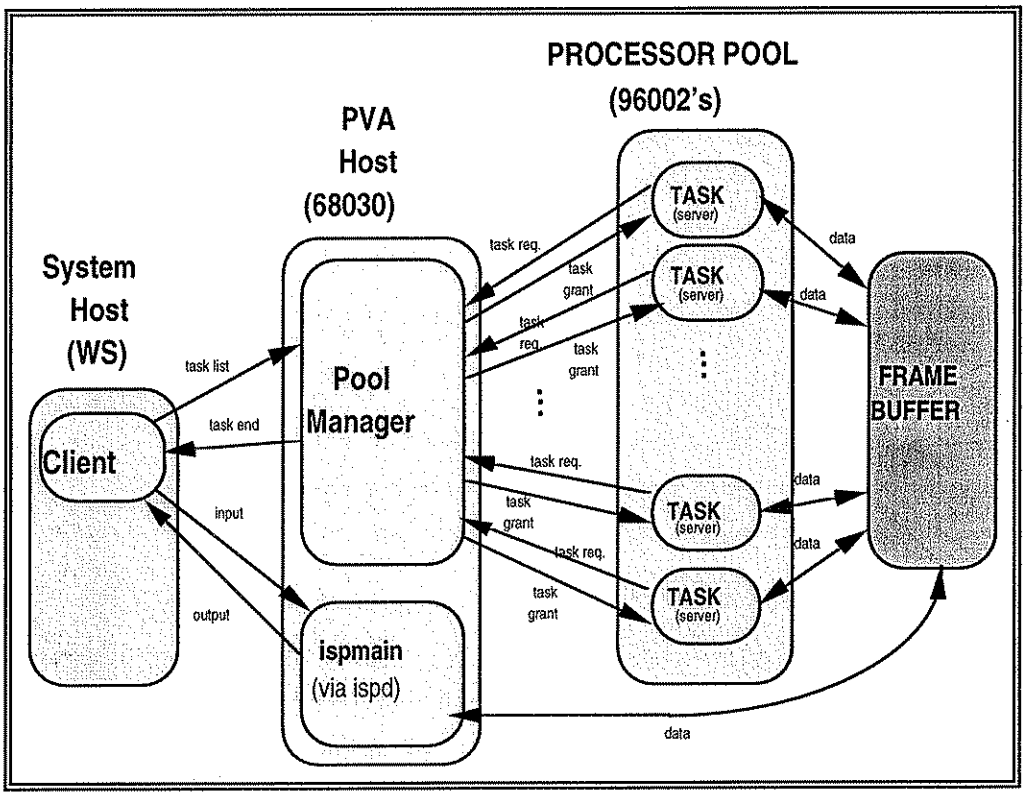

Figure 7.16. Platforms, entities and communication schemes for client-server programming

A library has been developed to provide users an easier programming environment. The relevant system call are given at Appendix B. A sample program is given in Fig. 7.17 for workstation and Fig. 7.18 for DSPs. Makefile to produce executable code is in Fig. 7.19.

The sample program running on the system host has six distinguishable parts;

- Pre-processing (a sequential part of the algorithm)
- Initialising the parallel processing environment
- Preparing and sending data
- Preparing and sending tasks (implicitly waiting for the results)
- Collecting the results.
- Post-processing of the results (sequential part of the algorithm)

The sample program running on DSP has three distinguishable parts. The third part is a forever loop comprising five sections.

- Pre-processing (a sequential part of the algorithm)
- Reading initialisation data
- Waiting for a task to be dispatched forever
    - Getting task and task information
    - Reading data
    - Computing and producing local results
    - Communicating with pool manager over the results
    - Writing results

Given parts should be considered as a superset of any program structure, as all of them are not expected to be in each program.

```
/*************************************************************/
/* client.c       A template client prog. for client server*/
/*                type of programming                        */
/*                                                           */
/* 951114    DTA  First Implementation                       */
/*************************************************************/
main(argc,argv)
int argc;
char **argv;
{
int *taskval;
int *taskvalptr;
int noofparam;

int array_a[576][720], array_b[576][720], array_c[576][720];
int constants[20];
int result[100];

/* more local variables should be here  */

/* sequential computation code should be here*/

/* preparation of data to be send to frame buffer should be here */

/* initialise  communication env. + frame buffer env */
  init_env();

/*send data to frame buffer (into four files) */

  init_fb("constants");
  send_data(constants,sizeof(constants));
  deinit_fb();

  init_fb("array_a");
  send_data(list,sizeof(array_a));
  deinit_fb();

  init_fb("array_b");
  send_data(array_b,sizeof(array_b));
  deinit_fb();

  init_fb("results");
  send_data(res,sizeof(results));
  deinit_fb();

  nooftasks=...; /* number of tasks to be dispatched         */
  noofparam=...; /* number of parameter to be passed along
                    DSP while dispatching the task           */

/* memory allocation for taskval*/
  taskvalptr=taskval=(int *)malloc(nooftasks*sizeof(int)*noofparam));

/*send tasks */
  for(i=0;i<nooftasks;i++){
    *taskval++=...;          /* fill the task parameters into the taskval */
    }

  send_task(nooftasks,noofparam,taskvalptr);

/* Receiving results step */
  receive_results("results",results,100);

  deinit();
  free(taskval);
  exit(0);
}
```

Figure 7.17. client.c; (client side of client-server type of programming)

```
/*************************************************************/
/* server.c       A template server prog. for client server*/
/*                type programming                          */
/*                                                          */
/* 951114    DTA  First Implementation                      */
/*************************************************************/
#include "elseta.h"
#define COPMAX 1

int a_a[576][720], a_c[576][720];
int a_b[576][360];
int res[100];
int cons[20];

main()
{

WKTaskItem task;
int first=0;
/* more local declarations */


  id=WKmyid();

/* pre-processing should be here*/

/* reading and initialisation of data structure should be here */

  while(1){                                  /*FOREVER LOOP*/

/*  DSP%d is waiting for a new task */
    WKtask(&task);
/*  New task dispatched */

/*  The passed task information is processed */
    task_info =*task.wkti_argv[0];

/*  Reading data just for once */
    if(!first){
      WKgetdatanpi("array_a",a_a,576*360);
      WKgetdatanpi("array_b",a_b,576*720);
      WKgetdatanpi("constants",cons,100);
      first=1;
      }

/* Reading data whenever a task is dispatched*/
    WKgetdatanpi("array_c",a_a,576*720);

/* Computation and producing local results should be here */

/* A special command to communicate with PVA-Host
   (asking poll manager for global maximum in this case)   */
    i=WKCombin(COPMAX,(float)depth,&send_depth);

/* Writing to frame buffer for client */
    WKputdatanpi("results",res,sizeof(res));

    }
}
```

Figure 7.18. server.c; (server side of client-server type of programming)

```
# Makefile to compile client-server based applications
# 951002 DTA

CC = cc
LIBS = /usr/monalisa/Project/GFXHost/ISP/isp.a
GFX_HOST = /usr/monalisa/Project/GFXHost
CFLAGS = -traditional -O2 -I/usr/include \
        -I/usr/monalisa/Project/GFXHost/ISP/defs \
        -I/usr/monalisa/Project/GFXHost/Comms/Rpc \
        -I/usr/monalisa/Project/ELSET-A/OS9software/defs
RM = rm -f

.c.o:
        $(RM) $@ $(@F)
        $(CC) $(CFLAGS) -c $<

taskserver: taskserver.o \
        $(GFX_HOST)/task/task_func.o \
        $(GFX_HOST)/ELSET_Tools/Command/Connect/Connect.o \
        $(GFX_HOST)/Comms/Rpc/Socket.o \
        $(GFX_HOST)/Comms/Rpc/Comms.o \
        $(LIBS)

        $(CC) -o taskserver taskserver.o \
                task_func.o \
                Connect.o \
                Socket.o \
                Comms.o \
                $(LIBS)
```

Figure 7.19. A Makefile for client server based applications (compiles taskserver.c)

Appendix A -Library Calls for DSP programming (WKLIB)

```
PMprint(buf)
unsigned int *buf/* print buffer */
```

This call is used for printing an ASCII buffer to the Pool Managers Standard Output. Normally used for debugging purposes.

```
PMtime()
```

This call is used for timing the execution of certain sections of code. The first call to `PMtime` sets the clock and the second call causes the time expired between the first call and the second to be printed to the debug file pmdebug in /dd/MONALISA/Various/pmdebug under OS9 file system. Timing information is printed in **secs.millisecs**.

```
WKReadImage(buffer, size)
char   *buffer      /* image buffer */
int    size         /*image buffer size*/
```

This is a special call currently used by the Camera Tracking Server. It is provided for those who want to build similar applications. When called `WKReadImage` fills in the image buffer provided with the image from the frame buffer. A special process running on the OS9 system (i.e. Camera Tracking Manager) separately arranges for a particular image sequence to be used from the frame buffer.

```
wksetdbex()
wkreldbex()
```

These are special calls used by the `GetData` and `PutData` system calls for controlling the DBex Interface. May be used by users with special Dbex usage needs.

```
wkerror(error)
int    error        /* error number*/
```

Writes a given error number to a reserved debugging area in the memory (Local SRAM). The error count is stored at address 0x900 and the dump area starts at address 0x901. Each `wkerror` call causes the error count and the dump address pointer to be incremented and the error number to be dumped into the address pointed by the dump pointer. The user should check for the limits of the dump area which is 0x1500.

```
WKexit()
```

`WKexit` is used to exit from the Worker. This should only be used for debugging purposes. Normally workers do not exit; they wait for additional work from the dispatcher; and they are terminated by the scheduler when required (i.e. the worker is sent a die signal).

```
WKfree(ptr)
char   *ptr         /* buffer pointer */
```

Returns dynamically allocated memory back to the heap. Currently there are problems in the implementation of this routine; the user should be careful when allocating and freeing large areas.

```
WKgetdata(objhandle, buf, size)
char               *objhandle  /* Data Object Id */
unsigned int       *buf        /* Input Buffer */
int                size        /* maximum number of words in buffer */
```

The buffer is filled with the data from the frame buffer associated with the Object Handle. If the frame buffer data is larger than the buffer only the amount specified by size is filled in. This call is used as the basic data distribution mechanism to distribute application data residing in the frame buffer to workers. The data object handles are sent to workers within task assignments by the dispatcher. In the current

implementation image names for the images stored in the frame buffer can be used as Data Object Handles.

```
WKmalloc(size)
int    size          /* memory size */
```

Allocate memory from the heap. Current implementation has problems; use with caution, especially for size larger than 8K.

WKmyid()

Returns the processor id No. This may be useful if workers need to identify themselves.

WKmyslotid()

Returns the Slot Id No for the active worker. Currently only one slot (i.e. Slot 1) is used for applications.

```
WKprintf(0 , format, args)
char          *format      /* format string*/
```

Printf like function for formatted print on the local monitor. The first argument must be 0; this is a current requirement that will be removed later. For `WKprintf` to work the respective board should be connected to a monitor.

```
WKputdata(objhandle, buf, size)
char                *objid       /* Data Object Handle*/
unsigned int        *buf         /* Output Buffer*/
int                 size         /* Buffer Size*/
```

`WKputdata` is used for workers to put their results in the frame buffer or in files in the OS9 file system. Data Object Handles are provided within the task assignments by the dispatcher.

```
wkrpc(dport, fcode, msgp)
int           dport        /* Destination Port*/
int           fcode        /* Function Code*/
UMsg          *msgp        /* Pointer to Msg Structure*/
```

The remote procedure call is used by `WKgetdata`,`WKputdata`,`PMTime`, and `WKtask` calls. Usually the user will not need to explicitly use this function. However it may be useful in certain special cases. The message structure is defined in the `Elseta.h` file.

wksetsignal()

This call informs the system that there is a signal handler. `WKsignal()` is the signal handler program that should be provided with the application code. When there is a soft-interrupt the system checks if a signal handler is registered; if yes it jumps to the special address (0x30001) where the signal handler is loaded. The address 0x30000 is used for the signal handler flag.

```
WKtask(rbufp)
unsigned int *     rbufp        /* Reply Buffer */
```

`WKTask` is used in the *worker template* (Server Programs) to send task requests; the reply buffer is used for storing the task assignment; it can also be used to pass immediate data to the dispatcher.

Appendix B -Library Calls for DSP programming (WK2LIB)

```
int WKgetdatanpi(image,buffer,size)
char          *image;
unsigned int *buf;
int           size;
```

A similar call to WKgetdata() except one can read the same page of the given "image" file. Fills the "buffer" with the "image" data of the size "size". Note that WKgetdata() reads next page of the same file if it is called twice.

Return value : Returns the size that is actually read if it is successfull, -1 otherwise.

See also : WKgetdata()

```
int WKputdatanpi(image,buffer,size)
char          *image;
unsigned int *buf;
int           size;
```

A similar call to WKputdata() except one can read the same page of the given "image" file. Puts the data pointed by the "buffer" into the "image" of the size "size". Note that WKputdata() writes into the next page of the same file if it is called twice.

Return value : Returns the size that is actually written if it is successfull, -1 otherwise.

See also WKputdata()

```
int wk_iop_init_to_read(image,buf)
char          *image;
unsigned int *buf;
```

Sends message to pool manager (object manager) to initialise the IOP and to leave it open for reading.

Return value : Returns the error code send from pool manager. 0 if it is successfull, <0 otherwise.

See also : wk_read_direct() wk_iop_release_read()

Remarks : This command will be unified with wk_iop_init_to_write later on.

```
int wk_iop_init_to_write(image,buf)
char          *image;
unsigned int *buf;
```

Sends message to pool manager (object manager) to initialise the IOP and to leave it open for writing.

Return value : Returns the error code send from pool manager. 0 if it is successfull, <0 otherwise.

See also : wk_write_direct() wk_iop_release_write()

Remarks : This command will be unified with wk_iop_init_to_read later on.

```
int wk_iop_release_read(image,buf)
char          *image;
unsigned int *buf;
```

Sends message to pool manager (object manager) to release the IOP which is supposed to be initialised for reading and left open before.

Return value : Returns the error code send from pool manager. 0 if it is successfull, <0 otherwise.

See also : wk_read_direct() wk_iop_init_read()

Remarks: This command will be unified with wk_iop_release_write later on.

```
int wk_iop_release_write(image,buf)
char          *image;
unsigned int *buf;
```

Sends message to pool manager (object manager) to release the iop which is supposed to be initialised for writing and left open before.

Return value : Returns the error code send from pool manager. 0 if it is successfull, <0 otherwise.

See also : wk_write_direct() wk_iop_init_write()

Remarks : This command will be unified with wk_iop_release_read later on.

```
int wk_read_direct(buffer, size)
unsigned int *buf;
int          size;
```

Reads directly from dbex port into buffer of a given size. All initialisation should be done before using that command. This command opens the dbex connection first then reads.

Return value :Returns 0 if data successfully read, 1 if interface not opened, and 2 if an unexpected error occurs.

See also :wk_iop_init_read()

```
int wk_write_direct(buffer, size)
unsigned int *buf;
int          size;
```

Writes directly to previously attached file in the frame buffer through dbex port from the buffer of a given size. All initialisation should be done before using that command.

Return value : Returns 0 if data successfully written, 1 if interface not opened, and 2 if an unexpected error occurs.

See also : wk_iop_init_write()

```
WKWriteImage(buffer,size)
unsigned int *buf;
int          size;
```

Writes to previously attached file in the frame buffer through dbex port from the buffer of a given size. This command does exactly the same task as done by wkwriteimage() except return values.

Return value : Returns 0 if data written successfully, -1 otherwise.

See also : WKReadImage() wkwriteimage()

void signal_(code)
int code;

It writes down the given 'code' into an interprocess communication register (actually IPCR3 which is located at 0xBFFF001B on VME bus) for signaling. This register is accessible both through 68030 and any one of DSPs, i.e. DSP-DSP or DSP-68030 communication can be realized by using it with wait_().

See also : wait_() waitframe()

Remarks : There is no restriction to access interprocess communication registers. Be sure that no other processor attempts to write while your program is in progress.

void wait_(code)
int code;

It waits for 'code' to be written into an interprocess communication register (actually IPCR3 which is located at 0xBFFF001B on VME bus). This register is accessible both through 68030 and any one of DSPs, ie. DSP-DSP or DSP-68030 communication can be realized by using it with signal_().

See also : signal_() waitframe()

Remarks : There is no restriction to access interprocess communication registers. Be sure that no other processor attempts to write while your program is in progress. It waits forever unless another DSP or 68030 writes proper value into that register.

void waitframe()

It waits for "FRAME_AVAILABLE" to be written into interprocess communication register (actually IPCR4 which is located at 0xBFFF001C on VME bus). FRAME_AVAILABLE is defined in *handshake.h*. After it gets the signal it clears it immediately and returns to caller program. This commands is designed to work in correspondence with frame buffer hardware which signals IPCR4 when a new frame is available.

See also : signal_() wait_()

Remarks : There is no restriction to access interprocess communication registers. Be sure that no other processor attempts to write while your program is in progress. It waits forever unless another DSP or 68030 writes proper value into that register.

void pmtimestamp()

Sends a message to pool manager to put a time stamp for it that will be read and evaluated later. It saves only the number of seconds since midnight and number of ticks (ticks/second=100 in the current configuration but this value is printed while dumping recorded stamps, as well.)

See also : pmtimedump() pmtimediffdump()

Remarks : Time information written into memory in order to prevent delays. For the time being the pool manager have 100 rooms to store time stamp.

void pmtimedump()

Sends a message to pool manager to write all time stamps into "pmdebug" which is located at /dd/monalisa/various under kiri. After writing into the file, pool manager list is cleaned.

A sample output

```
DSP 0 requested a timedump
tick/second                :   100
# of recorded timestamps :     2
============================================================
DSP 5 @ 55153.53
DSP 5 @ 55220.36
============================================================
```

See also : pmtimestamp() pmtimediffdump()

Remarks : As only seconds since midnight and number of ticks are recorded, it is users responsibility to evaluate the correct difference if the first time stamp is greater than the second one. Bear in mind that second range is between 0 and 86399.

void pmtimediffdump()

Sends a message to pool manager to write all differences between two consecutive timestamp calls of the same DSP into "pmdebug" which is located at /dd/monalisa/various under kiri. After writing into the file, pool manager list is cleaned. If the number of timestamps of a DSP is odd, then the last one is skipped. The number of recorded stamps is also given even all off them is not taken into consideration.

A sample output

```
DSP 0 requested a timedump
tick/second                :   100
# of recorded timestamps :     2
============================================================
DSP 5 @ 00066.83
============================================================
```

See also : pmtimestamp() pmtimedump()

Remarks : As only seconds since midnight and number of ticks are recorded, it is users responsibility to evaluate the correct difference if the difference is a negative value which is possible if two consecutive time stamp calls are done over midnight. Bear in mind that second range is between 0 and 86399.

int wkopenimage(rw)
int rw;

Opens the dbex connection and set it in use. The same command as WKOpenImage() except return values.

Return value : Returns 0 if it is successfully opened, 1 if interface is either busy or not ready, 2 if illegal parameter is sent.

See also : WKOpenImage()

void wkcloseimage()

Closes the dbex connection and releases the dbex. This command is as same as WKCloseImage() and is implemented just for unity of a group of commands.

See also :WKCloseImage()

```
wkwriteimage(buffer, size)
unsigned int *buf;
int          size;
```

Uses directly dbex_write and returns the same results as WKWriteImage() .

See also : WKWriteImage()

```
wkreadimage(buffer, size)
unsigned int *buf;
int          size;
```

Uses directly dbex_read and returns the same results as WKReadImage()

See also : WKReadImage()

```
int WKvmegetdata(objname,buf,size)
char         *objname;
unsigned int *buf;
int          size;
```

A similar call to WKgetdatanpi() except this call uses VME bus to transfer data instead of dBex connection.

Return values :

0: Data transferred successfully
-1: Data transfer could not be completed
-2: Object not exist
-50: ambiguous parameter

See also : WKvmeputdata() WKgetdatanpi()

```
int WKvmeputdata(objname,buf,size)
char         *objname;
unsigned int *buf;
int          size;
```

A similar call to WKputdatanpi() except this call uses VME bus to transfer data instead of dBex connection.

Return values :

0: Data transferred successfully
-1: Data transfer could not be completed
-2: Object not exist
-50: ambiguous parameter

See also : WKvmeputdata() WKgetdatanpi()

Appendix C Library Calls for System Host Programming

int init_env()

Creates poll manager connection, initialises the device (0th device) to provide frame buffer accesses. This function should be called once before any function in this library.

Return values:
- 0 : successful initialisation
- -1 : Pool Manager connection failed
- -2 : ISP connection failed
- -3 : Device selection failed (these devices are non-sharable)
- -4 : Device initialisation failed

int init_fb(filename)
```
char *filename
```

As any kind of data on the frame buffer is accessible through their names, this initialisation function creates a file on the frame buffer using the given name. A file should be created prior to sending data. If the file with the given name exists on the frame buffer, it deletes the former file without any warning.

Return values:
- 0 : successful initialisation
- -1 : can not create the file
- -2 : can not attach the file

int send_data(dataptr,sizeofdata)
```
int *dataptr;
int sizeofdata;
```

Transfers data into the file on the frame buffer which is opened previously by *init_fb* command. This function uses another underlying library function called *fbwrite()*. This underlying library dependency by-passed by the command *send_data_v2()* .

Return values:
- 0 : successful upload
- -1 : can not transfer the data

See also : init_fb(), send_data_v2()

int deinit_fb()

Detaches the currently accessible file and allow another file to be attached.

Return values:
- 0 : detachment successful
- -1 : can not detach the active file

int send_data_v2(dataptr,sizeofdata)
```
int *dataptr;
int sizeofdata;
```

Transfers data into the file on the frame buffer which is opened previously by *init_fb* command.

Return values:
- 0 : successful upload
- -1 : channel definition failure
- -2 : page definition failure
- -3 : can not initialise the transfer direction
- 4 : data segment error
- -5 : data writing error

See also : init_fb(), send_data_v2()

```
int  send_task(nooftasks,tasklistsize,listptr)
int  nooftasks;
int  tasklistsize;
int  *listptr;
```

This command sends the tasks to the pool manager running on OS9 after packing 16 of them together.
T

Return values:
- 0 : sending tasks successful and the results ready
- -1 : pmRpc failed
- -2 : task handling on pool manager failed

```
int  receive_results(filename,resultbuffer,size)
char *filename;
int  *resultbuffer;
int  size;
```

After returning back from the send_task call, the results are ready to be read. On the other hand user should aware of the ability of accessing files on the frame buffer concurrently. One could read temporary results without interrupting the run of any server.

Return values:
- 0 : read successfully
- -1 : can not attach to the given file
- -2 : can not read the file
- -3 : can not detached the file

```
int  deinit()
```

De-initialises the frame buffer environment, releases the pool manager connection.

Return values:
- 0 : disconnected successfully
- -1 : abnormal termination (ispend)
- -2 : abnormal termination (ispdefdiscon)
- -3 : can not disconnect pool manager.

Bibliography

OS9 Operating System
Microware System Corp.: *OS9 Operating System Manuals*, 1991
Microware System Corp.: *OS9 Language Manuals*, 1991
Microware System Corp.: *Using OS9/Internet*, 1989

DSP Programming
Intermetrics MicroSystems Software Inc.: *InterTools 96002 C Compiler/Assembler User's Manual*, 1991
Motorola: *Motorola DSP 96002 User's Manual*, 1989

PVA Host Programming
DVS Digitale Videosysteme: *ISP200/ISP400 Installation and Setup Guide* , 1992
DVS Digitale Videosysteme: *Host Computer Software for UNIX Operating Systems*, 1992

Systems Design
D.T. Altilar, Y. Paker, A.V.Sahiner : *A Parallel Architecture for Video Processing*, Proceedings of Conference on High Performance Computing and Networking Europe'97, 1997.
Blonde L., Buck M., Galli R., Niem W., Paker Y., Schmidt W., Thomas G., A Virtual Studio for Live Broadcasting: The MonaLisa Project, IEEE Multimedia, Summer 1996.
D.T. Altilar: *The First Year Report of PhD Research*, Queen Mary and Westfield College, January 1996.
A.V.Sahiner,P. Lefloch, Y. Paker: *A Parallel Accelerator for Using Synthetic Images for TV and Video Production*, Image Processing for Braodcast and Video Production, Hamburg 1994.
P. Lefloch, A.V.Sahiner, Y. Paker: *Visual Tools for Parallel Server Handling*, Image Processing for Braodcast and Video Production, Hamburg ,1994

Application
P. Lefloch: *Studio:* A Tool for image sequence handling, QMW, 1993

Application Programming
D.T. Altilar: *Parallelisation and implementation of a sequential longest gray code search program for ML-PVA*, Queen Mary and Westfield College, December,1995.
D Routsis, A.V.Sahiner, Y. Paker: *Real Time Camera Tracking Server on the ELSET Accelerator*, Image Processing for Braodcast and Video Production, Hamburg 1994