CERN PS-CO Note 92-002
version 1.0 29/01/1992

Using DSC at PS

User's manual and cookbook.

This manual is aimed to application developpers in the DSC environnement at PS.

This edition (version 1.0) is a first draft and a partial delivery. Additionnal chapters will follow later...

This manual is issued by:

Alain Gagnaire,
Wolfgang Heinze,
Julian Lewis,
Nicolas de Metz Noblat,
Claude-Henri Sicard.

With the contribution of: H. Abie, F. Berlin.

CERN Geneva, Switzerland, January 1992

Cern Geneva 1992

PS division, Control group.

Contents

4 Diskless LynxOS Systems

5 Backup/restore of MVME147 disks

6 Understanding VME space in a DSC

7 VME - Addressing facilities library

8 Installing a VME module in a crate

9 Loading drivers under Lynx O.S.

10 SDVME - Serial Camac interface driver

11 TSVME404 - GPIB interface driver

12 ICV196VME - ITX interface driver

13 FPIPLSVME - PLS Telegram and FPI driver

The Device Stub Controller (DSC)

The DSC represent the front end computing layer in the proposed CERN control system to upgrade the 15 years old previous ones. The chosen architecture aims at a real convergence of CERN's accelerator control system. This definition was made by the CERN PS/SL working group DWG.

Main functions of the DSC:

The DSC are distributed in the local areas, connected to the workstations via a LAN, and to the equipment either directly or via a field bus. The main functions of the DSC are derived from its place in the hierarchy of the control system. they are:

1) To provide a uniform interface to the equipment as seen from the workstations.a DSC provides a connection to the workstations via ethernet with the standard protocol TCP/IP:

   a. The programs running in the workstations can call programs in the DSC by RPC (Remote Procedure Call) but can also establish fast repetitive data transmissions from the DSC to the workstations (e.g. for repetitive display).

   b. A DSC can be reset remotely.

2) To provide direct control and acquisition for equipment like beam instrumentation, interfaced directly to the DSC, the DSC runs a real time operating system : LYNX O.S. :

   a. It allows to run several tasks concurrently, e.g. a beam measurement program together with general programs like statistics, surveillance and diagnostics .

   b. It also provide a fast and determinist response to external events which is necessary if PPM (Pulse to Pulse Modulation) equipment is directly controlled by a DSC.

3) To act as a master and data concentrator for distributed equipment, interfaced via field bus, the DSC provides hardware and software for field bus connections.

Certain application programs run in the DSC (in general they run in the workstations). For this purpose local display facilities are provided via a TV driver with standard graphic functions much simpler a X-windows. General programs such as alarms and surveillance, which scan regular intervals the equipment connected to the DSCs report their output to a server on the LAN.

Finally , the DSC provides local access to the equipment and can be seen as a "banc d'essai" in the lab to run simple test program locally using NODAL and its extended libraries.

Hardware platforms:

In the DWG final report, two different basic platforms were kept: one based on the IBM/PC architecture, the other on VME based 68030 microprocessors.

One of the reasons of the choice of the LynxOS operating system was that the same operating system was available on both platform, providing an homogeneous Unix user interface.

As this manual was mainly written during the LPI implementation, the various targets are often forgotten in the various chapters of this note. By default, all chapters describing VME modules drivers are targeted towards MC68030 based boards, especially the MVME147/SA1 as this one was at that time the only VME board supported by LynxOS. Since this date, time, LynxOS was also made available by THEMIS on TSVME13x boards.

General chapters apply to the 2 target systems: based on PC and VME, running the Unix real time operating system Lynx OS .

At PS, the first PC DSC will be based on an industrial 80486 and used as the MTG (Master Timing Generator) as its boards were designed in common with the SL/CO.

1

File-system road map

Authors : Julian Lewis
Claude-Henri Sicard

Introduction:

The file system road map determines where a program can find the objects that it needs in a given environment. There are four environments, namely:

1 The USER environment:

This environment is what you see when you log in under your user name on your office workstation. Here you create text files and edit them, you may compile and run some programs, and perform some preliminary tests.

2 The DEVELOPMENT environment:

In this environment are libraries and include files provided by other users which can be used in developing a new program, or library. Depending on what your final target is going to be, you will need different utilities, libraries etc. The final targets are:

a. MCR level workstation:

These workstations have a global view of the control system. In the event of a failure of the LAN, they can be cut off from a given accelerator, and thus should not be the exclusive owner of critical applications or data.

b. ACCELERATOR level workstation:

At this level critical data and programs are kept, which will allow a reduced operation of the accelerator during times when the LAN can not access the MCR level, thus, any critical MCR programs or data must be copied down to the accelerator levels. The accelerator levels may also contain applications specific to one accelerator, which are not logically global, and should therefore not be located at the MCR level.

c. DSC level:

Here we find real time applications accessing the hardware and data files. The DSCs themselves have no disks and run on the accelerator level servers.

3 The TEST environment:

In the test environment, applications can be tested without interfering with the actual operation of the PS complex. Here test-DSCs are provided, along with copies of any critical data which may be updated by the program under test. Some libraries may provide simulation facilities.

4 The RUNTIME environment:

This is where the applications run which are controlling the PS complex. Here the applications and their data are kept.

Objects

In each of the above environments we can find a sub set of the following list of objects:

- C-program source files.
- UIL source files.
- Include files.
- Man pages.
- Binaries.
- UID files.
- Libraries.
- Data files.
- Make files.
- Shell scripts.
- Programs to be executed.
- Nodal files.

Activities

And we can define activities using these objects in the different environments to be:

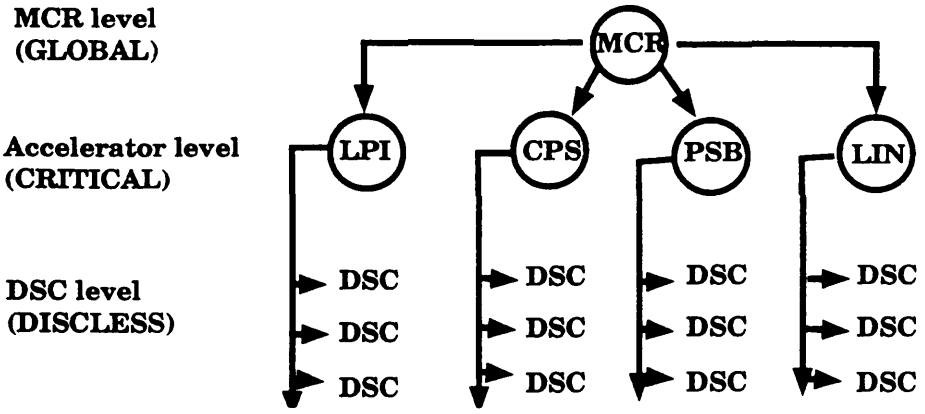
- Developing.
- Testing.
- Installing.
- Running.

What the file-system road map describes, is the set of paths and the corresponding NFS mounts which are required in order to support these activities from the various final targets.

We shall now describe each of these environments in turn starting with the simplest ...

The RUNTIME environment:

As stated above, this environment actually runs the PS complex, and hence consists of installed and tested programs with their data running on any of the three final target levels. The activity of running a program in this environment dose not in general require objects such as source files or object libraries.



DSC paths seen from the DSC and accelerator levels:

1 DSC => /*<accelerator>/data* Accelerator => */dsc/data*

Read/Write data shared between all DSCs connected to the same accelerator. For example /lpi/data/... is the path to common data seen from any LPI DSC.

2 DSC => */<accelerator/local/data* Accelerator => */dsc/<dsc name>/data*

Read/Write private data used by a particular DSC. For example /lpi/local/data ... is the path to a private data area for this DSC.

3 DSC => */<accelerator>/bin* Accelerator => */dsc/bin*

Read only access to programs shared between all DSCs connected to the same accelerator. For example, seen from a DSC, then /lpi/bin/... contains all LPI shared DSC programs.

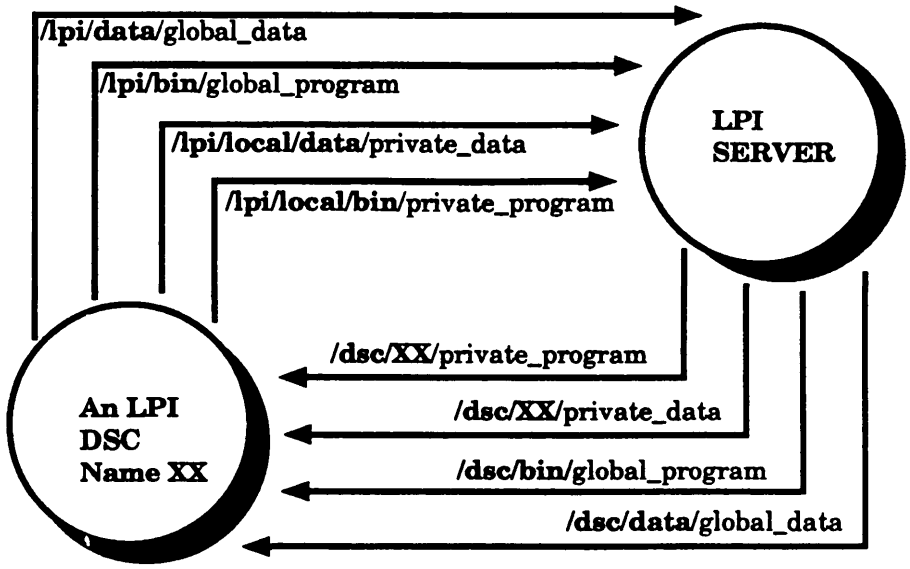
4 DSC => */<accelerator>/local/bin* Accelerator => */dsc/<dsc name>/bin*

Read only access to programs that are only able to run on this particular DSC. For example /lpi/local/bin/... is the path to private programs running only on this LPI DSC.

Symbolic links on DSC:

From the DSC we can create a symbolic link such that */dsc* is equivalent to */<accelerator>*. Thus in the following example names like ***/lpi/data/global_data*** can be replaced by the equivalent name ***/dsc/data/global_data.***

EXAMPLE:



Accelerator paths seen from the accelerator and MCR levels:

1 /mcr/data

Global data, if any of this is critical, then it must be copied to the appropriate accelerator server.

2 /mcr/bin

Workstation application programs running in a global context.

3 /<accelerator>/data

Critical data residing on an accelerator server not seen by its DSCs.

4 /<accelerator>/bin

Critical or non shared workstation application programs.

5 /<accelerator>/dsc/data

Data seen by the workstations for a given accelerator and all its DSCs.

6 /<accelerator>/dsc/bin

Programs used by all DSCs of a given accelerator.

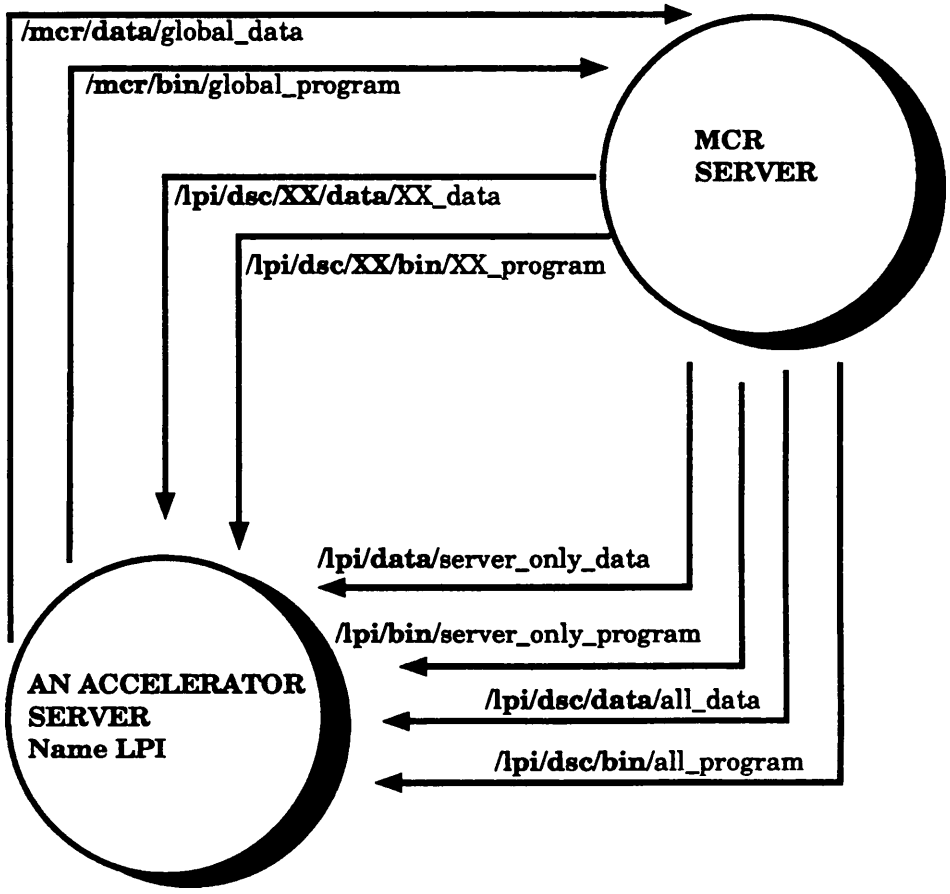
7 /<accelerator>/dsc/<dsc name>/data

Data to be shared between workstations of a given accelerator and a particular DSC.

8 /<accelerator>/dsc/<dsc name>/bin

Applications which run on a given DSC of a given accelerator.

EXAMPLE:



General remarks:

The names have been invented so that if the correct questions are asked, then the path name should be obvious, for example, an application running on an LPI DSC asks the question: Where do I store my own private data ? Answer, */dsc/local/data/...* The same DSC asks: Where do I store data to be accessed by all LPI DSCs and workstations ? Answer /dsc/data/... An application running at the MCR level asks: Where is the LPI DSCs' data with DSC name of XX ? Answer /lpi/dsc/XX/data.

The names MCR, and the accelerator names, when preceded by a "/" in fact refer to physical disc drives, and have exactly the same meaning throughout the network, so for example /mcr/data can be used by MCR applications and accelerator level programs, both referring to the same data. Copying critical data to the accelerator level will take the general form:

cp /mcr/data/... /lpi/dsc/data/... or cp /mcr/data/... /lpi/data/...

In the former case the data is seen by DSCs, in the latter only by LPI applications running on LPI workstations.

Environment variables:

In order to make life as simple as possible, to guard against future changes in the file-system road map, to make applications portable between the final target levels, and to simplify testing, we provide five environment variables:

1 ACCELLR

The name of the accelerator on which we are currently working, for example LIN, PSB, CPS, LPI, or MCR. The final name MCR means all accelerators.

2 GLOBALP

The path to global programs, on each level it will be set as follows:

a. **DSC** => */dsc/bin*

b. **Accelerator** => */mcr/bin*

c. **MCR** => */mcr/bin*

3 GLOBALD

The path to global data:

a. **DSC** => */dsc/data*

b. **Accelerator** => */mcr/data*

c. **MCR** => */mcr/data*

4 LOCALP

Local programs path:

a. **DSC** => */dsc/local/bin*

b. **Accelerator** => */<accelerator>/bin*

c. **MCR** => */mcr/bin*

5 LOCALD

Local data path:

a. **DSC** => */dsc/local/data*

b. **Accelerator** => */<accelerator>/data*

c. **MCR** => */mcr/data*

Dependence of environment variables:

The above environment variable values correspond to their values in the run time environment. During program testing however, they will not point to operational data, but to copies set up only for test purposes. This is achieved by systematically appending */test* to all data path names. It is the responsibility of the person testing an application to make sure that the data is up to date or not, as the actual operational data may in fact not be what is require to exercise the program under test.

This leads to the next environment ...

The TEST environment:

Not yet written, sorry.

The DEVELOPMENT environment:

Not yet written, sorry.

2

Local utilities on Lynx systems

Author: Nicolas de Metz-Noblat

Various utilities are installed on our Lynx systems. They are either undocumented in the standard documentation, or they are standard UNIX programs from the public domain, or they are specific software developed at CERN for the Lynx systems.

Text editors

At least three different editors are available on the Lynx systems: **emacs**, **vi** and **e**.

vi is the default (poor) UNIX editor. It is the only one available on any UNIX system and its knowledge is required at least to do some system maintenance.

emacs is product from the Free Software Foundation (the GNU project) and by definition is a free. It is one of the most popular editors, that can be customized in various ways.

e is the RAND editor, an editor which is today in the public domain and that was quickly ported to Lynx on MVME147 and PC. This editor was the most popular editor at CERN on Ultrix system a few years ago and is known by the majority.

Printers and printing utilities

There is a standard **BSD** spooler (**lpd**) and associated utilities (**lpr**, **lpq**, **lpc** and **lprm**). They allow us to see printers connected to our Ultrix cluster (located in the Terminal Room and in the Meyrin Control Room) in the same manner as from our DECstations.

lpr is the standard BSD UNIX interface to the spooler.

lpq allows you to look at the printer queue.

lprm allows you to remove one of your jobs from the printer queue.

lpc is restricted to user root and used to manage printer queues and daemons.

You can today use any printer declared in /etc/printcap. For new installations, you will get a message "your host does not have line printer access" until your host is declared in /etc/hosts.lpd of the server specified with rm= field. You can change your default printer in your .Login file modifying the line $#setenv PRINTER lnps01).

In order to simplify the usage of the program production chain, the following utilities have been installed on the system:

a2ps is an utility that does pretty-printout from source files (see **man a2ps** on a DECstation or on cernvax).

pt is a simple shell script to facilitate the usage of **a2ps**, changing the defaults options and piping its output to **lpr** utility.

The NODAL interpreter

A nodal interpreter has also been ported to the Lynx system. As this is intended to be customized depending on the application, locally available functions may vary largely from one DSC to another one. In any case, several major features can be noticed:

An EXEC and an IMEX NODAL server should be automatically started by the /var/etc/rc.local shell script.

You have to take care to the fact that file names specified to those servers should be fully qualified as they do not have any idea of defaults environment variables of caller.

See man nodal on either DECstation, or on cernvax.

Sharing files with the PS control Ultrix system.

Development DSC systems are setup in order to share home directories with the home directory you have on the PS DECstations locally on the DSC, this across NFS.

The practical result is that you can edit files from any station or development DSC, then compile it on any DSC (I recommend to use the DSCDEV as this one is never used to test applications), then run it on a target system.

With V2.0 official distribution, the local file system is still not reliable enough to be used for long-term storage, and it is much easier to be able to overwrite it on each new system release. Therefore, I discourage to use the local disk except for temporary storage (and so no backups are required).

Please take care that if you are logged in as root (as this is often required when writing a device driver), you have no write access to your files, but any try to overwrite a file will result in a file lost (root is nobody across NFS, but still has write access to directories!).

Notice also that operational DSCs should never rely on the accessibility of developper's home directory: there is a separate environment dedicated for operationnal computers on local Unix server from which the DSC will boot and we cannot keep the development environment critical for operation (even if - for commodity reasons - the development environment can be seen from operationnal DSCs).

3

LynxOS utilities

Author: Nicolas deMetz-Noblat

Here is a non-exhaustive list of LynxOS commands that was written for the V1.2.1 release but that should still be mostly valid.

File Management

cd - change current working directory

chgrp - change group ownership of files

chmod - change access mode of a file

chown - change file ownership

df - show free disk space

du - show disk usage

file - classify files

find - locate files

ln - make links between files

ls - show directory contents and file information

mkcontig - make a contiguous file !!!!

mkdir - create directories path - find path to a file

pwd - print working directory name

rm, rmdir - remove (unlink) files or directories

tar - combine files into an archive

touch - change the modify date of files

Utility Programs

btoa, atob - encode/decode binary to printable ASCII

banner - show arguments in big letters

basename, dirname - return portions of path names

bined - binary editor binfix - fix binary files

bm - search a file for a str

cal - produce a calendar

calc - simple desk calculator

cat - concatenate files

clear - clear terminal

cmp - byte-by-byte file comparison

compress, uncompress, zcat - compress and uncompress files

cp - copy files

cpio - copy file archives in and out

cs - produce file checksum

ctags - generate editor tag file

date - display or set current date and time

dlsh - Lynx shell

(dosread, doswrite, dosdir) - manipulate DOS diskettes (PC only)

dump - dump selected parts of an object file

echo - echo arguments

emit - emit characters given their ascii values

expand, unexpand - add or remove tabs from files

expr - expression evaluator and pattern matcher

finger - user information lookup

flex flook - interactively examine file blocks

grep, egrep - print lines matching a regular expression

groups - show current group membership

head - print the first few lines of a file

host, hostname - print or set name of current host

id - print current user uid, gid

ident - identify files

less - interactive paginator

lesskey - specify key bindings for less

login - login into LynxOS

more - interactive paginator

mv - move or rename files

od - file dumping utility (octal, hexadecimal or ascii)

passwd - change user password

pg - paginate files

pr - produce a formatted listing of a file

printenv - print environment variables

prof - interpret profile informations

sed - sequential editor

sh - standard command programming language shell

show - show hex and character [illegible]

size - print size of an object [illegible]

sleep - suspend execution of current process for a given interval

sort - sort or merge files

split - break stream into pieces

stty - set terminal driver options and parameters

su - set effective user id

tail - print last few lines of a file

talk - converse interactively with another user

tee - distribute data to multiples files

termcap - retrieve termcap data

test, [- simple relational expression evaluator

time - output a command's elapsed real, user and system times

tr - translitarate characters

true, false - provide truth values

tset - set up terminal line

tty - print the name of the current tty device

uniq - report repeated lines in a file

vi - visual text editor (default unix editor)

wc - count words, lines, characters in a file

who - show current users whoami - show current user name

Program development

cc, cc030 - native C compiler and linker interface

disasm - object and executable disassembler

ld - object file linker

libr, ranlib, ar - manipulate object library files

make, oldmake - automatic compilation manager

makeboot - install default bootstrap program

mkshlib - create a shared library

mktimestamp - embed time stamp in a string

nm - print symbol table

strip - remove symbols and relocation information

st - post-mortem debugging aid

awk, gawk - GNU pattern matching and processing language

bison - GNU parser generator (yacc replacement)

diff, diff3 - Gnu intelligent file comparison

emacs, ctags. etags - GNU emacs editor

gcc - Gnu Interface to C compiler

cpp - Gnu Preprocessor

cc1 - Gnu C compiler itself

as - Gnu assembler

gdb - Gnu debugger

m4 - general purpose macro-processor

rcs, rcsdiff, rlog, ci, co - RCS

sccstorcs - build RCS file from SCCS file

System Manangement

/init - master system process

/bin/rc - system startup script

/net/rc.network - network startup script

/net/rc.nfs - nfs startup script

config - produce system configuration tables

devices - show installed devices

devinstall - install physical device

drinstall - install or unload device

drivers - show system device drivers

install - intelligent copy for software installation

installvpkg - install a System V package

ipcrm - System V compatible interprocess communication facilities removal

ipcs - System V compatible interprocess communication facilities status

kconfig - produce system configuration tables

kill - terminate a process

fmtflop - format floppy diskettes

fmtscsi - format SCSI disk drives

fsck - file system check and repair

idos - identify default operating system

lptest

mkfs - make a file system

mknod - create a special file entry

mkpart - make partitions on a disk and update bad block info

setactive - set active (boot) partition

mkramdisk - dynamically install a RAM "disk" device

mkromk - make a romable kernel file

mount - mount a file system or a remote NFS directory

newconsole - choose a new console device

ps - display status of current processes

reboot - reboot the system

s5fstotar - convert System V file system to tar archive format

shownode - show inode contents

sems - user semaphore status and removal

setprio - alter priority of a process

smems - shared memory status and removal

spool - general purpose spooler spooler

stasks - display status of current stream tasks

sync, syncer - write out disk cache

tic - terminfo compiler

umount - dismount a filesystem

vmstart - start the virtual memory management

zeronode - re-initialise a disk inode

(saio) - configure analog I/O driver

/etc/mount, /etc/umount - mount a system V.3.2 filesystem

Network utilities

arp - address resolution display and control

ftp - file transfer program

ifconfig - configure network interface parameters

initrarp - initialize ARP table thought Reverse Address Resolution Protocol

kermit

lpc - Berkeley printer control program

lpq - Berkeley printer queue management

lpr - Berkeley printer print utility

lprm - Berkeley printer remove utility

netstat - show network status

pac - Berkeley printer accounting information

ping - send ICMP ECHO_REQUEST packets to network hosts

rcp - remote file copy

rlogin - remote login

route - manually manipulate the routing table

rsh - remote shell

ruptime - show host status of local machine

rwho - who's logged in on local machines

slattach - attach serial lines as network interface

telnet - user interface to the TELNET protocol

tftp - trivial file transfer program

Network servers

fingerd - remote user information server

ftpd - DARPA internet File Transfer Protocol server

inetd - internet "super-server"

lpd - Berkeley printer daemon

named - Internet domain name server

rexecd - remote execution server

rlogind - remote login server

routed - network routing daemon

rshd - remote shell server

rwhod - system status server

talkd - remote user communications server

telnetd - DARPA TELNET protocol server

tftpd - DARPA Trivial File Transfer protocol server

NFS management

exportfs - export and unexport directories to NFS clients

mountd - NFS mount request server

nfsd - NFS daemon

portmap - DARPA port to RPC program number mapper

rpcgen - an RPC protocol compiler

rpcinfo - report RPC information

showmount - show all remote mount

unfsio - network input/output deamon for NFS client support

Libraries:

_etext.o

init.o

init1.o

initn.o

pinit.o

pinit1.o

vinit.o - system V compatible C init program

vinit1.o

gnulib - small library needed by code generated by gcc

libbsd.a

libc.a

libc_nv.a

libc_p.a

libc_v.a - System V compatible interface library

libcurses.a

libm.a - Mathematical library

libnetinet.a - host databases routines

librpc.a - NFS RPC library

libtermcap.a - termcap library

Special files

/etc/exports - static export information

/etc/xtab - current state of exported directories

/sys/lynx.os/CONFIG.TBL - Lynx[illegible]

/etc/devices - dynamically loaded system [illegible] information

/etc/drivers - dynamically loaded device drivers table

/etc/fstab - table of file systems

/etc/ftpusers- table of users that cannot be accessed via ftp

/etc/magic - data for file utility

/etc/motd - "message of the day"

/etc/mtab - table of mounted file systems

/etc/group - group information file

/etc/hosts.equiv - name of hosts with "equivalent" user IDs

/etc/hosts - TCP/IP host names database

/etc/inetd.conf - inet server database

/etc/networks - TCP/IP network names database

/etc/nodetab - special file table

/etc/passwd - table of user names, passwords and login informations

/etc/pconfig - Lynx "printer" configuration file (unused)

/etc/printcap - Berkeley spooler printer database

/etc/printers - Lynx "printer" list of spoolable devices

/etc/protocols - TCP/IP protocol names database

/etc/rpc - NFS rpc names database

/etc/rmtab - NFS table of currently mounted filesystems

/etc/services - TCP/IP service names database

/etc/starttab - system startup data file

/etc/tconfig - serial port configuration file

/etc/termcap - terminal capability descriptions

/etc/ttys - login terminal information

/etc/utmp - currently logged in terminals information

~/.rhosts - user-specified file of equivalent hosts and users

~/.profile - /bin/sh initialisation file

~/.Login - /bin/dlsh login initialisation file

~/.dlshrc - /bin/dslh initialisation file

/dev/nfssvc - special chr dev to maintain NFS server data in kernel

NFS library routines

nfsmount - mount a NFS directory

getrpcent, getrpcbyname, getrpcbynumber - get RPC entry

getrpcport - get RPC port number

PS additions

/usr/local/bin/e - the RAND editor (bug on more command)

/usr/local/bin/nodal - the NODAL interpreter (bug in terminal handling)

4

Diskless LynxOS Systems

Author: Nicolas de Metz-Noblat

Target systems are normally diskless systems. In our context, there will be several servers, one per accelerator complex (e.g. LPI, ..) and one on the office LAN.

Principles of operation

On LynxOS V2.0, the MVME147 board contains four eproms. The two first ones contain the Motorola firmware (147-Bug) which is used to reset the hardware, test it, and then transfer the control to two others eproms that contains LynxOS bootstrap code.

The network boot procedure first send a Reverse Address Resolution Protocol (RARP) request, i.e. an ethernet broadcast requesting "Who knows my IP address?". The server, which knows this address replies to this request.

Once got its own address, the diskless system tries to fetch its system with tftp from the server which replied.

If transfer was successful, the loaded system is started. This system does already holds a RAM disk which contains the minimum programs required by the startup procedure.

The first startup action is to get system specific startup files (hosts and rc.network files) across tftp, always from initial boot server and then to execute them.

Execution environment

Once the startup procedure completed, following file systems are reachable from the diskless system:

- local ramdisk - with limited space (about 500Kb with around 150Kb free after startup.

- a read-only environment in /usr that does hold all standard programs and data shared by diskless DSCs.

- a read-write environment in /var that does hold everything that requires to be modified by this DSC and is equivalent to the local disk - even if located on an other system. In particular, this can hold a swap file if swapping is required for non-real-time programs such as interactive programs

- various NFS mount for access to [illegible] directories and other data that requires to be shared between the [illegible] including disk based development systems).

RAM disk contents

The ramdisk is the master directory (/) of the system. Its size has to stay limited as it effects the reliability of the downloading procedure and it consumes the physical memory.

On startup it does contains the following files:

/init This is the first dispatched program that executes the /client/rc script and then manage the interactive logins on active terminal lines.

/client This directory holds all other programs that are required by startup. It is partially cleaned up by the startup procedure in order to free some space in the ramdisk after startup.

/client/dlsh This is the startup shell and is used as default shell for login as it is already memory resident.

/client/getccf This program retrieves the /etc/hosts and /client/rc.network on startup and should be deleted after startup.

/client/hostname This is used to setup the host name and should be deleted after startup.

/client/ifconfig This is used to start TCP/IP and should be deleted after startup.

/client/mount This is used to mount (or check mounted) remote directories.

/client/route This is used to define network routes.

/client/stty This is used in root .Login file

/client/tset This is used in root .Login file.

/client/unfsio This is the NFS client program and is required to stay.

/client/rc This is the common startup shell script.

/client/rc.network This is the specific startup shell script.

/etc This directory contains configuration data required by normal Unix programs as network servers, terminal descriptions. Most files are just links to /usr/etc, with the exception of ttys and motd taken from /var/etc.

/dev This directory does contains all system special files required to access devices.

/pipe This ramdisk directory is intended for efficient creation of special files for pipes, and is publicly writable.

/sem This ramdisk directory is intended for efficient creation of special files for semaphores, and is publicly writable.

/usr This is the mount point of the read-only shared environment.

/var This is the mount point of the computer specific read-write environment.

/bin, /cc, /lib, /sys this are symbolic links to the /usr environment.

/tmp This is a symbolic link to /var/tmp and this last one is cleared on system startup.

/.Login, /.dlshrc, /.subroutines This files are required for root login.

/dsc/bin this is the mount point to the read-only application environment shared by the various DSCs of a single server

/dsc/local this is the mount point to the application environment specific to this DSC.

/dsc/data this is the mount point to the shared area between DSC's and consoles.

The read-only shared environment (/usr)

This does contains the normal system environment of a disk-based system, with the only modification that all directories in this directory must be grouped (e.g. move /bin to /usr/bin, /X to /usr/X).

As this environment is shared by all diskless systems, changes in this environment should be achieved with special care as this does affect all of them.

Few special cases must be noted:

/usr/spool has to be a symbolic link to /var/spool as this has to be unique for each system.

/usr/tmp has to be a symbolic link to /var/tmp for the same reason.

/usr/etc this directory contains all files from /etc that can be shared between the different diskless systems as hosts, passwd, termcap and other service files.

/usr/local this directory is the only one that should be modified by us in order to extend the standard environment.

/usr/local/bin this directory does hold common programs as editors and other site specific executable programs.

/usr/local/lib this directory should receive all libraries that you want to share with other users.

/usr/local/include this directory should receive the include files associated with the corresponding libraries (preferably grouped in a single subdirectory per product).

/usr/local/util This directory is intended to receive extra read-only data or programs that are not directly started by a user command (preferably grouped in a single subdirectory per product).

/usr/local/drivers This directory is intended to receive all drivers that are automatically installed on startup by the **dynaminst** program.

The read-write permanent environment (/var)

This environment is in practice the true permanent storage of the system. It will contains all data (and programs) specifics to this computer - but special care should be taken by real-time applications as access to the disk can be blocked at any time for an indeterminate amount of time (and possibly with I/O errors) if the file server is re-started.

When the system is initially installed, the following directories are created:

/var/tmp this is the normal /tmp of the system and is cleared- on every system restart. It is normally accessed across /tmp[illegible]. This directory is publicly writeable.

/var/adm this directory is intended to receive system administration statistics data and its access should be restricted to root user.

/var/spool this directory is intended for use by the spooling system and its access is restricted to root user.

/var/etc this directory contains [illegible] definitions), **rc.local** (specific startup script), **motd** (Message of the day) and **insttab** (dynamic drivers installation table).

Setting-up a new board

The first step is to note the ethernet address written on the back of the front-panel. This address will be required at different steps in the board initialization.

First plug in an up-to-date release of the Motorola firmware (today rev. 2.42) in sockets U22 and U30. Then Plug the LynxOS TFTP bootstrap eproms in sockets U1 and U15. (check for correct eprom type selection depending upon eprom as described in the MVME147S/D1 documentation page 2.4).

If this is the first installation, check that the board connector, the two flat cables and the MVME712 connection board are connected on the back - and that on the MVME712, at least serial port 1 is configured as a terminal (not modem).

Connect a terminal to the serial line and power-up the crate and check it is connected to the V24 line at 9600 bauds.

Depress together Abort and Reset switches, then release the Reset switch - maintaining the Abort switch for about ten seconds. This will reset the Motorola firmware and you should have the prompt 147-bug> on the terminal. If nothing does happens, check first the terminal, then try to cross the two Motorola Eproms.

Once you get the prompt on the terminal, the first thing to do is to check if the bootstrap eproms are in the right order. This can be achieved with the command:

MD FFA00000

That should produce a memory dump of the specified address and where the text **BOOT** can be read. (If its OBTO, you have crossed U1 and U15 eproms).

Then you have to do an exhaustive test of the board:

ENV

B (bug environment)

E (Enable tests)

E (Enable RAM test)

(then all defaults)

Depress Reset button and then The whole card is tested. Don't worry about bad NVRAM contents.

Once tests successfully completed, disable them with the command:

ENV

B (Bug environment)

B (Bypass)

(then all defaults)

Disable the Motorola system boot with the command:

NOAB

Define the boot as going to the second EPROM set with the command:

RB

R

N

FFA00000 (Boot address)

Check the Ethernet address with the command:

LSAD

Then set the board time with the command:

SETTIME

10/07/91	(07 Oct 91)
(return)	(Calibration value)
10:10:00	(HH:MM:SS)

Then depress once more the Reset button. This time it should automatically issue a GO command and go to the Lynx Monitor that you should immediately abort by pressing the Return key.

There you must issue the following commands:

R 0 0	(define the root file-system)
V	(Boot across the network).

There the system boostrap should startup, trying to reboot across the network.

Don't forget to reconnect the Ethernet cable.

Then you have to login on the server in order to declare this new DSC or to change the Ethernet address of the replaced one.

If, after bootstrap, the startup repetitively fails on getting the rc.network file, this just means that you have forgotten to issue the "R 0 0" command.

If "*" character continuously appear on the terminal, you have to check the ethernet address written on the terminal that must be equal to this one entered on the server. Notice that - for the time being at least - you cannot cross the CISCO, i.e. bootstrap from a server not located on the same IP network (128.141 or 192.91.236).

DSC configuration management on the server.

Most DSCs are diskless and are serviced by a local server.

In order to use this procedure, you have to be in the list of DSC privileged users or to be the user root of the server, as for any system management routine.

There you have to call the DSC management program:

cd /<complex>/dscenv/bin

dscconfig

This provides you the following menu:

DSC Configuration Procedure

LynxOS boot directory: /dscenv/tftp/lynx/tftpboot

l - List known DSCs

a - Add a DSC

m - Change ethernet board address of a DSC

d - Delete an existing DSC

q - Exit from this procedure

Your selection ?

By default, the response is **q**.

The **a** command is intended for declaration of a brand new DSC whose address must be already known from the stations (i.e. be declared in our YP hosts database). The only other requests from the program are the DSC name and its hardware ethernet address. In case of doubt, a generic address 08-00-3e-00-00-00 (for an MVME147 board) can be entered and then modified later with the **m** command. This will create the whole diskless environment requested by a diskless station, then declare it in the various system files as required.

The **m** command is to be used every time you do a standard exchange of a board as this is the only way to distinguish two boards on startup.

The **d** command allows you to remove a DSC. You need to be logged in as root in order to remove also all files associated with this DSC. If this is not the case, the only effect is to disable this DSC from any boot and to remove all its authorizations for NFS access to the server (the rm of the root directory refuse to work). This command should be used with special care as you normally remove all files that belongs to this specific DSC.

The **l** command is used just to consult the /etc.ethers file

Maintaining the diskless environment.

A disk based MVME147 system is required (at least for the V2.0 beta-test release), in order to prepare the boostrap image. Today only the DSCDEV (dsps07) is allowed to issue the right NFS mounts required to do this maintenance.

You will have first to login as root on dsps07 and to issue the following command:

mount XXXsrv:/XXX/dscenv /mnt (XXX= lpi, mcr or tst)

You can then regenerate the system image in the **/mnt/usr/sys/lynx.os** directory. (You will find there a copy of all system source files).

You can update the initial ramdisk contents in **/mnt/usr/root** directory, but don't forget to clean it up after modifications.

After any modification in this environment, it is required to regenerate the boostrap system image with the following commands:

cd /mnt/tftp/lynx/tftpboot

make

Please take care that during that time, no diskless 147 based DSC will be allowed to reboot and that all modifications will be valid for all 147 boards booting from this environment.

N.B.: the **/usr/local** environment of all DSCs (including disk based DSCs) is normally the same one (taken from SVPS02) and this can only be modified from DSCDEV.

5

Backup/restore of MVME147 disks

Author: Nicolas de Metz-Noblat

One very important operation is to keep up to date backups of developments systems. This can be very easely achieved using the SCSI streamer.

Disk capacity varies from one system to an other, the minimum (DSPS01 and DSPS02) being of 85Mb, most others being 150Mb.

The Streamer capacity of 150Mb can be achieved only with DC6150 tapes. Please check this with your furnisher.

Doing a system backup

In order **to do a full system backup**, first log in as root and unmount any NFS mounted file system. This is preferably achieved by rebooting the system in single user mode with the following command:

reboot -f

Once the system restarted, it automatically enter the single user mode , a mode in wich the network is stopped and no user has access to the system. Note that the virtual memory system is then not active. You can then plug your cartridge (not hardware procted, i.e. not in the "safe" position) inside the streamer. Once the tape rewund, type the following commands:

cd /

tar cvbf 256 /dev/rtape

This will create a full backup of the system (except contiguous files):

tar is the Tape ARchive utility,

c means create an archive,

v means verbose, i.e. all file names will be printed on terminal (this flag is optionnal).

b requests tar to write on the tape whith the specified number of 512 bytes blocks in a single block. Here, we specify 256*512=128kb block on the tape. This is very important for a streaming tape and if not specified your backup will not fit on the tape.

f allows you to force to output to the specified device (here /dev/rtape).

. is very important: all backups have to be relative in order to allow you to restore them later on a different device than the one which it was created.

Once finished, finish the system re[illegible] and answering return to all questions.

Restoring system backup:

In order to restore a full system backup, you will need a copy of the first system installation tape.

If you are going to restore a backup of another system, please donot forget to first disconnect the Ethernet cable from the crate.

If needed, you can first reformat the disk using the Motorola 147-MBUG monitor.

Go to the target system and hit any character when you get the message "type any character to break".

There you will enter the Lynx PROM monitor. insert the first system installation tape and type the following commands:

R 0 10 r

b - t2

This will boot a system from the tape.

You have now to install this minimum system on the disk:

Freshdisk

After about 30 minutes, you can go back to the PROM monitor:

reboot -

You can now remove the system installation tape and reboot from the disk:

R 0 0

b - s0

You can now plug in your full backup and restore it with the command:

Getit /dev/rtape

Once finished, check the /net/rc.network to check the host name (and the /etc/hosts file to check if it is defined), reconnect the Ethernet cable and restart restored system with:

reboot -a

System will complain about the file /swap that will be missing, so login as root and recreate it:

mkcontig /swap 20m

Dont forget to reconnect the Ethernet cable.

Other backups

As a Unix user, you should know (and use) the **tar** standard utility.

You can, using the **tar** utility, do backups of several sub-directories, either to the streaming tape, either to a simple file on tape that you will then tranfer with ftp (without forgetting to specify a binary transfer mode) to any other computer.

A simple precaution: never specify absolute path names to tar (and dont forget to specify at least one file name as .) in order to be able to recover them in another directory.

A good practice is to check the contents of the archive (at least its beginning) with a **tar tf** command.

example:

tar cbf 16 /tmp/mine	save the contents of current directory
tar xvf /tmp/mine	restore it to the current directory.

6

Understanding VME space in a DSC

Author: Alain gagnaire

This note is an introduction to VME space addressing in a DSC. To get a full description of the VME bus protocol and CPU board addressing in a DSC see:

- VMEbus SPECIFICATION MANUAL (ANSI/IEEE STD1014-1987)
- CPU board reference manual: depending on the target board

Reminder of basic VME addressing from a DSC processor (CPU):

Addressing mechanisms principle in a DSC:

In a DSC, a running program, in supervisor mode or not, references addresses in **what is called its virtual address space**.

When executing a program, **the CPU addressing mechanism converts program virtual addresses in 32 bits physical addresses.** This convertion uses a mapping table attached to the program.

For a normal user program (non privileged) this mapping is restricted to the virtual space of its data and code. At run time of a program the O.S. allocates for each virtual page of data or code, a physical page in the memory.

For a system program this mapping includes visibility of private system areas and of the VME bus address space areas (see O.S. reference manual).

The physical address space in a VME CPU board consists of the normal memory address space corresponding to RAM, EPROM, local I/O and different ranges corresponding to the different VME bus addressing areas: short, standard, extended.

The mapping of the actual physical space on the VME areas depends on the CPU board : see VME CPU addressing in a DSC.

VME access :

A VME access can be :

- **a data access**: e.g. :read/write I/O registers, memory in a VME module, etc...
- **a program access:** fetching of instructions from a VME module: e.g. from a library in the Eprom of a VME module. the instructions are picked up from the VME space; an example is the graphic module TSVME600 which provides the access library to the basic functions in it's EPROM

An adddress on the VME bus is [illegible] the physical space as seen from the CPU accerding to the mapping [illegible] space in the CPU.

To access a VME module, the CPU must generate an address in the range of the mapping of VME address space.

When a 32 bit CPU falls into the range of the VME space, the corresponding access is processed by theVME chip interface, which is based on groups of lines:

- **Address Modifier lines (AM)**: these 6 lines are set up automatically by addressing mechanisms (they can also be programmed explicitely if needed, see in

the CPU reference manual instruction set). A VME target module acknowledges the addressing only when the AM lines fit its AM requirements. These lines are used to give information on the type of the adressing mode, to filter the access to modules and protect modules from program access. The 64 different configurations of the AM lines are organised into 3 categories (Defined, Reserved, User-defined) out of which we have to consider only the defined one made of 3 subsets:

  * **Short** addressing: 16 address lines used A15-A02 lines
  * **Standard** addressing: 24 address lines A23-A02 lines
  * **Extended** addressing: 32 address lines A31-A02

- **The Data lines: 8, 16, 32** according to the data width of the access (byte, short, long).
- **The address lines: 16, 24, or 32** according to the associated AM lines.

The automatically generated AM depends also on the CPU state on access, this value is :

- **in case of data access:**

Address Size	supervisor	non privileged
short (16 bits)	$2D	$29
Standard (24 bits)	$3D	$39
Extended (32 bits)	$0D	$09

- **in case of program access:**

Address Size	supervisor	non privileged
Standard (24 bits)	$3E	$3A
Extended (32 bits)	$0E	$0A

Reminder : The target module requires a certain AM. Most of the VME modules have got a strap to allow the user to partially set up the AM in order to define the access right: supervisor only or non privileged and supervisor.This set up will tell the user how to access the module from a program running in the DSC.

The VME space as seen from the CPU :

Depends on CPU board.

The whole physical addressing space of the CPU is shared by

- on board memory space of the system EPROM, RAM...
- on board I/O devices,
- VME space.

The layout of the addressing space is described in the CPU board reference manual.
The physical addressing on the VME bus is using the AM lines as well, therefore the user should remember on what the AM lines depends :

- the address range, determining the addressing type: short, standard or extended
- the addressing mode: data access or fetch mode, determines the access mode
- the state of CPU: user or system level determines the non privileged or supervisor access mode.

Reminder : during a VME access the data size must also fit the data port size of the module. This size can be forced by setting of theVME ship (e.g. in a the MVME147 the VMEship can be told to perform long data transfer in 2 short access cycles to fit 16 bit data port size module requirements).

The VME space as seen from the Operating System :

A program running under the O.S. cannot access the physical memory directly . Actually the program accesses its virtual space. An intermediate mapping, hiding the physicall memory topology to user programs, is used in order to associate a physical space to its virtual address space. In a DSC this mapping depends on the priviledge level of execution of the program.

- **From a non privileged program :** at this level the Operating system provides the program with the visibility of a subset of the whole virtual space corresponding to the data and code space. To access the VME space some facilties, depending on the type of Operating system, are provided to access directly the VME space, see chapter "user memory mapping".
- **From a system program** : at this level the Operating System has the visibility of the VME space in a special mapping, which can be used by system programs and drivers.

Reminder:

- The O.S. provides a special documentation for the mapping of VME space as seen from the system access (from drivers). This depends on the type of CPU board used.
- The O.S. provides user programs with facilities to directly access the VME space see chapter "user memory mapping and VME".

VME module visibility from the CPU board:

When receiving a new module and before installing it, the user will have to gather the following basic information about the module:

- AM supported by the module.
- Adressing size: short, standard or extended
- Functions supported:
  - data access
  - program access

A set of straps is inserted to filter the access according to the user's choice, e.g.: on the TSVME 600 the user can restrict access only to supervisor mode.

- Data port size: the module has got a data port of a given size, the access must follow this requirement (the VME chip interface must perform the data transfer correspondingly)..

  e.g.: the VME chip interface can be told to generate only word access,

  in this case long data transfer is done in 2 word data transfers.

- The VME base address: this is usually set by strap, it defines the module's VME offset in the corresponding VME subset range (short, standard or extended).

VME space mapping in the CPU adress space:

This is dependent on the CPU board:

For the MVME147 SYS1147U/D1 CPU board :

N.B.: see System manual in Operating instruction ch 3.3.1.1 p3-3

The map of main memory is given by the following table:

Address range	Devices accessed	Port size	Size	Notes
0-DRAMsize	On board DRAM	D32	4-32 Mb	
DRAMsize-$efff ffff	VME bus A32/A24	D32	3 Gb	1
$f000 0000-$ff7f ffff	VME A32	D16	248 Mb	
$ffff 0000-$ffff ffff	VME short	D16	64 Kb	

For the MVME147 MVME147S/D1 CPU board :

see User's manual in operating instructions ch 3 p 3-5

Address range	Devices accessed	Port size	Size	Notes
0-DRAMsize	On board DRAM	D32	4-32 Mb	
DRAMsize-$efff ffff	VME bus A32/A24	D32	3 Gb	1
$f000 0000-$f0ff ffff	VME A24	D16	16 Mb	
$f100 0000-$ff7f ffff	VME A32	D16	232 Mb	
$ffff 0000-$ffff ffff	VME short	D16	64 Kb	

NOTES: 1. This A24 only applies to VMEbus space that falls below $1000 0000. VMEbus space below $1000 0000 only occurs on versions of the MVME147 I MVME147/S that have DRAMsize smaller than 16 Mb.

For the THEMIS TSVME13x CPU board :

See the manual TSVME13x 68030 single-board computer

Address range	Devices accessed	Port size	Size	Notes
0-DRAMsize	On board DRAM	D32	1 or 4 Mb	
DRAMsize-$fcff ffff	VME bus A32	D32	3 Gb	1
$fd00 0000-$fdfe ffff	VME A24	D16	15 Mb	
$fdff 0000-$fdff feff	VME A16	D16	1 Mb	
$ffff 0000-$ffff ffff	VME short	D16	64 Kb	

Lynx O.S. facility to directly access the VME space:

A user's program normally has no access outside its memory space. To enable it to access VME space or any address range, Lynx O.S. provides a special function to extend the memory map of a non priviledged program to any physical space:

- **smem_create**: System call to get mapped onto the physical address range given in the arguments. It returns the virtual address in the caller's virtual space extention of the physical area specified in the parameters.This virtual area works like a window giving direct access to the corresponding physical area. Choosing the physical address in a VME range gives access to this VME area.

Reminder:

- The address given to **smem_create** as parameter is interpreted as a physical 32 bits address. To get a window mapped on a VME space area this value must be choosen according to CPU mapping of CPU board which depend on type of the CPU. **Note that such a program is CPU dependent ! ...** so to run it on another type of cpu board it must be recompiled with the corresponding VME range declarations.
- To prevent user program dependence on this mapping, general libraries facilities must provide dynamically this mapping.

N.B.: see LYNX OS ref. manual chapter 2 System Calls.

Hints to compute the 32 bit CPU address of a VME module :

This information is necessary for people who need to directly access a VME module using the Lynx O.S. facility in order to create an extention of a program virtual space (smem_create) mapped onto a VME space.

The physical address of a VME module as seen from the CPU is computed by adding to the base address of the range (short, standard or extended), the module base address set by the strap on the module board.

VME range CPU base address

\+ module VME base address

= VME module 32 bit CPU_address

e.g. : for the MVME147/S the VME short range is `0xffff0000`

if the module base address (given by the straps) is `0xe000`

the physical CPU address of the VME module is

`0xffff0000 + 0xe000 => 0xffffe000`

VME space visibility from Lynx O.S.:

The Lynx O.S. uses the virtual memory mechanisms, therefore it hides the physical space completely from the program. Each program is given the visibility of a virtual space corresponding to its code and data.

System virtual address mapping (mem.h) :

This may be dependent on the implementation of Lynx O.S., the mapping below correspond to the Lynx O.S based on a MVME147/S CPU:

The table below describes the address space *as seen from system level programs like drivers, it gives a mapping for the valide VME address range.*

N.B.:The source of this documentation was extracted from the C header file : **mem.h**, used to compile system kernel and drivers.

System memory mapping and VME address range:

Virtual Memory address	associated Physical Space
0xffff ffff	**VME short** I/O (A16)
0xffff 0000	On board PROM
0xff00 0000	**VME Standard** (A24)
0xfe00 0000	Reserved
0xfd00 0000	**VME Extended** (A32)
0xed00 0000	Physaddr (32 Mb)
0xeb00 0000	O.S. Addr (4 Mb)
0xeac0 0000	SpecPage/Startstack (8 Mb)
STARTPROTECT	USpecPage Copy of PSSENTRY for user to read
SHARED MEM START	
USTARTSTACK	(8 Mb)
PERLIMIT	USER AREA
0x0000 0000	

Hints to compute the system virtual address of a VME module :

This information is necessary for people writing drivers.

The virtual address of a VME module as seen from the system level is computed by adding the module base address set by the strap on the module board (The range base address is picked up from the system virtual address mapping) to the base address of the range (short, standard or extended) :

VME range system base address
+ module VME base address

= VME module system virtual address

e.g. : for the MVME147/S the VME standard range is `0xfe000000`
if the module base address (given by the straps) is `0x080000`
the physical CPU address of the VME module is

```
0xfe000000 + 0x08000 => 0xfe080000
```

7

VME - Addressing facilities library

Author: Alain Gagnaire
Habtamu Abie

The user is given an introduction to basic of VME addressing at chapter:

Understanding VME space in a DSC.

VME accesses via library calls for application portability:

In a DSC the access to the VME is available from the low and basic level of the system, using directly the system call facilities to open a virtual space window on the physical space corresponding to the VME area target. This level of programming makes the program dependent on the current O.S. features (in our case Lynx O.S. with the smem_create system call) and moreover makes the application directly dependent on the CPU mapping of the VME space.

Therefore a library interface was introduced in order to hold dependencies below application programs. It is up to the system managers to set up the library according to the actual Operating System and CPU environment.

The VME module address in the library interface:

To understand the VME addressing on a DSC see in this manual the chapter "**Understanding VME space in a DSC**".

A VME address for the user is made of 3 informations:

- The **Address Modifier (AM)** : a part of it specifies the type of address size used for the access: **short (16 bit address), standard (24 bits address), extended (32 bits address).** This is a fundamental characteristic of the addressed module to stand such or such address size and it determines which physical address range must be used by CPU to generate the associated AM lines on the VME bus by actual addressing.
- The **module base address** : usually it can be set up by strap on the module board. This must be understood as an offset in the physical address space associated to the address size type.
- The **module address offset** : it must be understood as an offset in the module address space.

The library determines the CPU base address of the VME module computed from the interface address as follows:

- The AM selected according to [illegible] to tell the library which base address of the Address size type range named AR to take, this depends on the CPU mapping .
- The VME module base address named **MBA** is given by the strap setting.

VME module physical base address = AR + MBA

The module address offset is finally added to make an access from the physical base address of the VME module.

The C library interface for the VME access facilities: (vmebuslib.o)

Using this library to access a VME module makes programs independent of the O.S. features and of the CPU physical memory mapping.

These facilities are basic functions to perform **single data accesses** in the VME space available on the DSC. A program VME address is defined, as explain above, by 3 informations:

The AM, the module VME base address and the offset.

How to use the library:

The source file of program using the library interface must declare in front the include file : **<vmebuslib.h>**

The Makefile building the user's application must include the appropriate lines to give the path of the object file : **/u/dscps/vmefclty/vmebuslib.o**

READ_VME, WRITE_VME : Read/Write from the VME bus

This functions process a read/write of a byte, word or long from/onto the VME bus at the VME address defined by the 3 informations: AM, module VME base address, user offset.

Formal C syntax definition:

```
void READ_VME(AM,Module_add,offset,ref_data,size,coco)
void WRITE_VME(AM,module_add,offset,ref_data,size,coco)
    unsigned long AM;
    unsigned long module_add;
    unsigned long offset;
    unsigned int  *ref_data;
    unsigned int size;
    int *coco;
```

Syntax of a call:

```
READ_VME(AM,module_add,offset,ref_data, size, coco);
WRITE_VME(AM,module_add,offset,ref_data, size, coco);
```

Where:

AM = The user select the AM corresponding to the type of addressing size he wants, the supported values are:

$29 to select the short addressing data access.

$39 to select the standard addressing data access.

$09 to select the extended addressing data access.

module_add = The address of the module defined by its strap.

offset = This is the offset inside the target VME module

ref_data = address of the data involved in the transaction.

size = size of the data exchanged in the transaction: 1 for a byte, 2 for a word and 4 for a long.

coco = completion code if >=0 error occurred see local error code below.

VME_MNGT : Function to get rid of module visibility after accesses.

This function is associated to the Lynx OS implementation to perform the VME access, this function release the Lynx OS resources implicitly reserved by the call to access function on a module. This function must be invoked when the access finished to release

the resources it sets up. When non released it would prevent to access other modules (maximum 8 different VME space can be simultaneously accessed by this way).

Formal C syntax definition:

```
void VME_MNGT(AM,Module_add);
    unsigned long AM;
    unsigned long module_add;
```

Syntax of a call:

```
VME_MNGT(AM,module_add);
```

Where:

`AM and module_add` = specifies the module interface address of a previously accessed module.

Error codes:

`VMEBUS_UNSIZE` = byte number not allowed

`VMEBUS_ILLADDR` = illegal base address

`VMEBUS_INVAM` = AM value not supported

`VMEBUS_SEGVIOL` = Hardware error by VME access

`VMEBUS_ILLWIN` = Cant get system resource to manage the VME access.

`VMEBUS_ILLNAME` = Internal error

`VMEBUS_INVSIZE` = size not supported

`VMEBUS_NSYSMEM` = `No more system resource for that access`

`VMEBUS_EINTER` = unexpected error code

The NODAL VME access interface:

The Nodal interpreter function extensions provide NODAL user with the VME access facilities. The on-line documentation for that purpose is available under the NODAL or the console station by invoking the 'man' program.

The NODAL function for the VME interface access are named :

VME R/W function

VMEMNGT Call function

SEE

The NODAL man pages manual :

PS/CO Note 91-0020

by F. Perriollat, G. Cuisinier, A. Gagnaire.

OR

under any console station

ask the man facility to display on line

the documentation of any NODAL function:

>man *function-name*

8

Installing a VME module in a crate

Authors: Alain Gagnaire,
Wolfgang Heinze

Inserting a board in a slot of a VME crate:

Whatever manufacturer your crate is coming from, you must always take care of the 5 standard jumpers along each slot:

Setting of the jumpers:

- IACK is the one to daisy-chain the interrupt acknowledge line, so *remove it only when a board is plugged in.*
- BG0, BG1, BG2, BG3 are the 4 ones grouped together serving the BGIN/BGOUT daisy chain. *They have to be taken out only if a module is inserted which can take mastership of the bus.* At present, the main CPU board is the only master in the crate, this make these jumpers unused in the other slots, so *leave these 4 jumpers untouched except for the CPU slot where they have to be removed.*

Attention:

There are some difference beetween crate coming from different manufacturers:

- for **WES** crate type V422 the jumpers are on the left-hand side of each slot.
- for **ROTRONIX** crates, the jumpers are on the left-hand side of each slot.
- for **MOTOROLA** crates 1147, the jumpers are on the right-hand side of each slot.

Reminder:

- remove all jumpers on the slot of CPU.
- if there is an empty slot, install the IACK jumper on that slot.
- if there is a board in a slot, remove IACK jumper from that slot.

CAUTION

Installing a module in one of the slots that use the P2 connector for another type of signal may damage all the boards in the system.

9

Loading drivers under Lynx O.S.

Author: Alain Gagnaire,
Habtamu Abie

Drivers for additional devices can be dynamically loaded under LYNX O.S. The standard LYNX O.S. documentation provides all information for that purpose but the complete recipe to follow is not given and the fundamental information is scattered in different places which make things not easy . In any case, the user must have read the LYNX O.S. standard documentation:

- LYNX O.S. User's Manual Vol 2 : Chapter 4.6 Writing Drivers
- LYNX O.S. Utility Programs Manual , see the entries: *devices, drivers, devinstall,, drinstall, drivers, mknod*
- LYNX O.S. System Calls, see the entries : *cdv_install, cdv_uninstall, bdv_install, bdv_uninstall, dr_install, dr_uninstall, mknod.* to perform the dynamical loading of driver from a program written in C.

Before you start your loading sequence the user should make some checks in the driver code in order to prevent misfunction in it:

- Check order of the driver's jump table entries which must as shown in table 4.1, page 76 in chapter 4, i.e.: ***Open, Close, Read, Write, Ioctl, Install, Uninstall !***

Installing a driver using LYNX O.S. commands:

This enables the user to dynamically install a driver by hand-typing the sequence of commands or editing a command file or a script file:

N.B.: the command lines are headed with @ as the prompt character of LYNX O.S.

1 Run your C program to build a file and fill it up with the sdata of the informa tion table required later at device installation:

If the program is named *setup_dev_info* and the file *dev_info_table* perform:

@setup_dev_info >dev_info_table

2 Load the driver object module, which result from the compilation of driver source code, into the system.

If the driver's object module is named *mydriver*, and the output of *drinstall* command is redirected into the file *driver_ident*, perform:

@drinstall -c mydriver [illegible]

3 Install the driver (as Major device) into the configuration of the system with the command:

@devinstall -c -d dev_info_table <driver_i

4 Create the name of the associated Minor device to set up a handle for this device, with the following sequence:

- Note the ID number of the corresponding Major device in the list displayed by the command *devices* , e.g *nn* the number for this example.

- Create the handle named *mydevice_handle* giving to Minor device the arbitrary value *ii* (in the range [0..255]), with the command:

@mknod mydevice_handle c nn ii

Installing a driver using LYNX O.S. system calls in a C program:

This enable the user to install a driver by calling a dedicated program. here follows, as an example, the source file of such a program:

```
/*  Example of C program installing a driver into LYNX O.S
Created: Habtamu Abie
 This program requires 3 option argument on the command line:
                -Dmydriver    to specify the module of driver in this case mydriver
                -Idev_info_table to specify dev_info_table as file to receive the information table
                -Nmydevice_handle to specify the name of the handle cre-
ated by mknod
When the program name is my_install a call looks like this:
                @my_install Dmydriver -Idev_info_table -Ndevice_handle
*/
#include <io.h>
#include <stat.h>
#include <file.h>
#include <stdio.h>
#include <ctype.h>
 /* info table: to provide hardware address */
struct {  long addr;  long ivec; } info_table = {0xfee00000, 3};
static char info_path[80], drvr_path[80], node_path[80];
int extra = 0;
/* set_path subroutine */
void set_path(path,arg)
                char.  path,[] arg[];
{int i;
                 i = 2;
                 while ( arg[i] != '\0'){
                 path[i-2] = arg[i];
                 i++;
                 }
                 path[i-2] = '\0';
 }
/* swith_path subroutine */
void switch_path(arg)
            char arg[];
{  char ch;
  ch = arg[1];
switch ((char) toupper((int)ch)) {
                case 'D':
                set_path(drvr_path, arg);
```

```
                    break;
                    case 'I':
                    set_path(info_path, arg);
                    break;
                    case 'N':
                    set_path(node_path, arg);
                    break;
                    default:
                    break;
}}
/*
void get_path(arg1,arg2,arg3)
          char  *arg1, *arg2, *arg3;
{                 switch_path(arg1);
                  switch_path(arg2);
                  switch_path(arg3);
}
/*          BODY of MAIN
*/
main (argc, argv)
int argc; char *argv
{
 int driver_ID, fd,  Majordev_ID, Minordev_ID;
 char ch[10], minor[80];
 if (argc != 4) {
                  fprintf(stderr, "Usage: %s -[Driver Info_table Path]\n",argv[0]);
                  exit(1); }
                 get_path(argv[1],argv[2],argv[3]);
                 fprintf(stderr,"Producing InfoTable: %s\n",info_path);
                 if ((fd = open(info_path,O_WRONLY | O_CREAT, 0755)) < 0) {
                                  perror("Cannot open or create INFO-table");
                                  exit(1); }
                  if (write(fd,&info_table,sizeof(info_table)) < 0) {
                                   perror("Error writing INFO-Table");
                                  exit(1); }
                 close(fd);
                 fprintf(stderr,"Installing driver : %s\n",drvr_path);          if ((driver_ID =
dr_install(drvr_path,CHARDRIVER)) < 0) {
                                   perror("Cannot install driver");
                                  exit(1); }
                 fprintf(stderr,"Driver ID = %d\n",driver_ID);
                 if ((Majordev_ID = cdv_install(info_path,driver_ID,&minor)) < 0) {
                                   perror("Cannot install driver");
                                   exit(1); }
                 fprintf(stderr,"Creating node with Majordev_ID %d with node %s\n",
                                  Majordev_ID,node_path);
                 if (mknod(node_path,S_IFCHR,Majordev_ID << 8) < 0) {
                                  perror("Error creating node");
                                   exit(1); }
                 do {              printf("Do you want make node with minor dev? (Y/N) ");
                                  scanf("%s",ch);  printf("\n");
```

```
                if ((char)toupper((int)ch[0]) == 'Y') {
                        printf("Give the minor device number: ");
                        scanf("%d",&Minordev_ID);
                        printf("\n");
                        sprintf(minor,"%s.%d", node_path,Minordev_ID);
                        if (mknod(minor,S_IFCHR,(Majordev_ID << 8 |
                                Minordev_ID)) < 0) {
                                perror("Writing to a file");
                                exit(1);}
} } while ((char)toupper((int)ch[0]) == 'Y');

                -=.=-                                                    */
```

SDVME - Serial Camac interface driver

Authors: Alain Gagnaire (software),
Wolfgang Heinze (hardware)

Introduction:

This VME module provide access to a Serial Camac loop: to Read or Write from/to a Camac module embedded in a crate in the loop, to accept Lams from Camac modules in the loop as an interrupt signal in the VME crate. A full specification of the module is given by the reference manual of the VME board provided by the developers of the module:

- **PS/CO/Note 91-022 Camac Serial Highway Driver in VME**

by L. Antonov, V. Dimitrov, W.Heinze.

The access to the driver's facilities can be done on 2 levels:

- The direct access to the driver interface based on UNIX i/o system call. This is not recommended because it makes the user program dependent on the drivers implementation and non portable.
- The library interface which provide global function hiding the UNIX system call interface and the driver specific interface to the programmer. This interface is available for C program and NODAL program.

Driver interface functionality:

These functions are provided via the user interface of the standard Unix file system.

Camac access : (ioctl function)

- Single Read/Write Camac.
- Multiple Camac access : perform a given sequence of Camac command
- Repetitive Camac access: perform the same Camac command the number of time given, on data provided/returned from/into an array.

Connection with a Camac LAM : (ioctl function)

- Connect : to get an event associated with a LAM of a given slot from a given crate of the loop. When a LAM occurred the driver will put an event, made of 4 bytes, in the ring buffer of the requesting device. The event is a sequence of 4 bytes, if we represent this event by : byte [3..0] the structure of the event is:
  - bytes [3,2]= 1 word = number of occurrences of LAM since the connection was made.
  - byte [1] = Crate number of LAM source.
  - byte [0] = Slot number issuing the LAM.

N.B.: the connect is exclusive, only one device can get the event for a given slot.

- Disconnect : to get rid of a previous LAM source connection. The in coming LAM are no longer given to this device.

Synchronization with a Camac LAM : (select, read function)

performed by means of the select or read function. The read will be used to get information on source of the incoming LAM (previously connected) .

Synchronization with Camac LAM (select, read file system call):

- Select : if the device descriptor of the sdvme driver, dedicated for synchronization is given in the list of a select, and if a connect was made before on it,for one or more LAM source, the select will return when one of this LAM arrived. The knowledge of the LAM source is acquired by reading the incoming events from the ring buffer of the corresponding device descriptor.
- Read : to get synchronized with a LAM source and to read the ring buffer in order to to know the LAM source. When ring buffer is empty the call is blocking during a laps of time, after which a time out is returned. The associated buffer must be tailored to receive at least one full event i.e.: minimum 4 bytes; the buffer is fed only with an integer number of events.

SDVME CAMAC Driver library Interface : (camaclib.o gpsynchrolib.o)

Introduction:

This interface inherited its main lines from the OS9 ESONE CAMAC IMPLEMENTATION FOR OPAL, it proposes a few subset of this whole interface and an extension with some more standard functions.

The library hides to the programmer the UNIX system call of the driver's direct interface. The user may like to have a minimum knowledge of the system resources involved by the library, for that purpose he has to read the further chapters giving the full description of the direct interface.

In few words, the user must know that a Camac access is based on UNIX i/o system call, to access the driver facilities the library has to open a device file, the currently opened device file identifier is stored in the global context of the library (local to the running process), this open is called at the first CAMAC access:

> For the CAMAC access request the library opens the device file dedicated for that purpose whose name is : **`/dev/sdvmelRW`** .
>
> For the synchronization request the library tries to get a free device file, non possible to be shared, out of the set of resources dedicated for that purpose whose names are:
>
> **`/dev/sdvmel01`** to **`/dev/sdvmel16`**

Using library access make program independent of the driver implementation and source code portable.

How to use the library :

In the source file of program include the header file associated to these libraries:

```
#include <camaclib.h>
#include  <gpsynchrolib.h>
```

The Makefile building the user program must include the following lines

- The general path to dsc library is defined in the makefile as :

```
# general path to lead to dscps library:
ROOT= /u
```

- To work with the access routine define the library path:

```
# define path to load camaclib.o
# camaclib.o full path
CAMLIB= $(ROOT)/camdfclty/camaclib.o
```

- To work with the synchronization routines define the library path:

```
#define path to [illegible]
RTLIB= $(ROOT)/r[illegible]fclty/[illegible]synchrolib.o
```

- To tell the loader to reference the cam[illegible] lib[illegible] include the $(CAMLIB) and/or $(RTLIB) in the compilation/link command line.

Primary routines :

<u>cdreg</u> : Encode a CAMAC address

This function encode a CNA CAMAC address it performs no CAMAC access at all !.

The returned encoded value is required in **`cfsa`** access function.

Formal C syntax definition:

```
int cdreg(ref_camadd, L, C, N, A)
    unsigned long *ref_camadd;
    unsigned short L;
    unsigned short C;
    unsigned short N;
    unsigned short A;
```

Syntax of a call:

```
err = cdreg(ref_camadd, L, C, N, A);
```

Where:

`ref_camadd` = add of the returned encoded value

`L` = CAMAC Loop number (for the moment only 1 loop supported)

`C` = Crate number in the loop (in the range [1..63]

`N` = Slot number in the crate.

`A` = Subaddress in the CAMAC module

gcamfunc ; Encode a CAMAC function

This function encode a CNAF CAMAC address/function, it performs no CAMAC access at all !.

The returned encoded value is required in multiple access function.

Formal C syntax definition:

```
int gcamfunc(ref_camfunc, L, C, N, A, F)
    unsigned long *ref_camfunc;
    unsigned short L;
    unsigned short C;
    unsigned short N;
    unsigned short A;
    unsigned short F;
```

Syntax of a call:

```
err = gcamfunc(ref_camfunc, L, C, N, A, F);
```

Where:

`ref_camfunc` = add of the returned encoded value

`L` = CAMAC Loop number (for the moment only 1 loop supported)

`C` = Crate number in the loop (in the range [1..63]

`N` = Slot number in the crate.

`A` = Subaddress in the CAMAC module

`F` = Function number

Single CAMAC access routine :

cfsa : Read or Write CAMAC access

The way of access : read/write depends on the F value, see CAMAC reference manual.

Formal C syntax definition:

```
int cfsa(F, encoded-value, ref_data, QX)
```

```
unsigned short F;
unsigned long encoded_value;
unsigned long *ref_data;
unsigned short Q;
```

Syntax of a call:

```
err = cfsa(F, encoded_value, ref_data, Q);
```

Where:

F = CAMAC function to perform at given encoded CAMAC address

encoded_value = CAMAC encoded address

ref_data = address of data for the read/write CAMAC access.

QX = Q and X response of required CAMAC access

bit 15 = Q response and the sign of QX: QX < 0 Q response, QX>= 0 no Q response

bit 14 = X response.

Multiple CAMAC access routines:

These functions use array of encoded CAMAC function returned by gcamfunc.

pmcami : Block CAMAC function

This function perform a sequence of CAMAC function given in an array.

Formal C syntax definition:

```
int pmcami(size,data_ar,camfunc_ar,retry,coco_ar, ref_coco)
    int size;
    int *data_ar;
    long *camfunc_ar;
    int retry;
    int *coco_ar;
    int *ref_coco;
```

Syntax of a call:

```
err = pmcami(size,data_ar,camfunc_ar,retry,coco_ar,ref_coco);
```

Where:

size = number of camac function to perform in camacfunc array

data_ar = address of data array for corresponding camac function

camacfunc_ar = address of camac function array to be performed

the encoded function value must be set up with **gcamfunc**

retry = mode of CAMAC [illegible] required:

> 0 max retry [illegible] Q false, check Q response

= 0 no retry, check Q response

< 0 no retry, ignore Q response

coco_ar = array of corresponding completion code returned

element = 0 no error, 170 no Xresponse, 169 no Qresponse

ref_coco = Global completion code

=0 no error occured, 68 one error at least.

mcamt : Repetitive CAMAC function

This function perform several time the same CAMAC function using an array for the data.

Formal C syntax definition:

```
int mcamt(size,data_ar,camfunc,retry, ref_perf, ref_coco)
    int size;
    int *data_ar;
    long camfunc;
    int retry;
    int *ref_perf;
    int *ref_coco;
```

Syntax of a call:

```
err = mcamt(size,data_ar,camfunc,retry,ref_perf,ref_coco);
```

Where:

`size` = number of repetition for the CAMAC function.

`data_ar` = address of data array for the successive execution of the camac function .

`camacfunc` = camac function to be performed.

The encoded function values must be set up with **gcamfunc**

`retry` = mode of CAMAC access checking required:

> 0 max retry on each action while Q false, check Q response

= 0 no retry, check Q response

< 0 no retry, ignore Q response

`ref_perf` = number of action successfully performed

`ref_coco` = array of corresponding completion code returned

element = 0 no error, 170 no Xresponse, 169 no Qresponse.

gpevtconnect, gpevtdisconnect : Synchronisation routine

These facilities are not specific for synchronization with CAMAC LAM events, they intend to provide DSC programs with general purpose means to get connected with external events such as: CAMAC LAM, trigger from Front panel interrupt module (ICV196), timing line events (PLS module).

gpevtconnect : Ask connection with a CAMAC LAM

To get synchronized with CAMAC LAM the program must first ask to get connected to. A call to gpevtconnect function for each of the expected CAMAC LAM must be done.

A CAMAC action 1 or 2 single CAMAC access function can be associated to the call in order to let them performed by the driver as soon as corresponding LAM occurs.

Formal C syntax definition:

```
int gpevtconnect(type,evt_val, ref_EvtDescr)
    int type;
    int evt_val;
        long *evt_descr
```

Syntax of a call:

```
synchro_device_id = gpevtconnect(type, 0, ref_EvtDescr);
```

Where:

type = 1

evt_val = ignored parameter for CAMAC synchronisation

ref_EvtDescr = long ref_EvtDescr[5] can be redefine as an union structure pointer:

```
union U_EvtDescr {
    long element[5];
    struct {
    short C, N;
    long F1_add, F1_data;
    long F2_add, F2_data;
    } atom;
};
```

Where:

ref_EvtDescr -> atom.C = Crate of the LAM to connect

ref_EvtDescr -> atom.N = Slot address of the LAM to connect

ref_EvtDescr -> atom.F1_add and F1_data =

if F1_add >= 0 gives the encoded CAMAC function to be first performed by the driver at LAM processing

ref_EvtDescr -> atom.F2_add and F2_data =

if >= 0 gives the encoded CAMAC function to be secondely performed by the driver at LAM processing

synchro_device = returned value giving the device file identifier to be used when the program want to get synchronized either by means of the system file **select** command or in a specific request on this events source by means of the **read** system file command which returns as well in the associated buffer the event identifying the LAM source.

gpevtdisconnect : Ask disconnection from a CAMAC LAM

To get rid of a previously connected event source the program must ask to disconnection of this source of CAMAC LAM.

Formal C syntax definition:

```
int gpevtdisconnect(type,evt_val, ref_EvtDescr)
    int type;
        long *evt_descr
```

Syntax of a call:

```
synchro_device_id = gpevtconnect(type, 0, ref_EvtDescr);
```

Where:

`type = 1`

`ref_EvtDescr` = same as for gpevtconnect with only the 2 first elemnts used:

`ref_EvtDescr -> atom.C` = Crate of LAM source to disconnect

`ref_EvtDescr -> atom.N` = Slot address of the source of the LAM.

The other arguments are ignored.

select, read : Getting synchronised with CAMAC LAM event

After connection established with the expected CAMAC LAM sources the program has to get synchronized with event and has to identify from which source LAM occurred.

For that purpose, the returned device file identifier from gpevtconnect call, the same for all the connection the program performs, allows the program to wait for this events, according to its needs, either by the `select` or by the `read`.

Events data structure:

These events are pushed in the corresponding device file and can be read by the program, the data structure of the event is as follows:

```
struct sdvmeT_Event {
                short count;
                unsigned char C, N;
                };
```

Where:

`count` = serial number of [illegible] connected

`C` = crate where the LAM occurred

`N` = Slot in the crate where LAm occurred.

Getting synchronized by the sel[illegible] the standard UNIX way, after return from the `select` the program must [illegible] from is coming the event

If the device file is the one of the [illegible] source, to know which out of all the connected LAM source, generated a LAM the program must perform the reading of the device which returns in the buffer the data identifying the source : see below getting synchronized by read.

Getting synchronized by the read :

Reading from the device file descriptor given back on gpevtconnect is waiting for CAMAC LAM to occur or getting in the buffer associated to the read the event data identifying the source of event.

This source depends on the kind of device generating the event, in case of sdvme module the structure of CAMAC LAM events is described above in the paragraph **Event data structure.**

Formal C syntax definition: `int read( synchro_device_id, buffer,byte_count);`

```
int synchro_device_id;
char *buffer;
int byte-count;
```

Syntax of a call:

```
err = read( synchro_device_id, buuffer, byte_count);
```

Where:

`err` = see Lynx O.S. reference manual for usage of read

`synchro_device_id` = Device file descriptor index given by the `gpevt-connect` call (always the same for all connect the program perform)

`buffer` = buffer to get event descriptor: minimum size required = `size of(sdvmeT_Event)` i.e. = 4 bytes. if the buffer is bigger it can receive as many already received event as the buffer can completely contents. the actual maximum event awaiting a read for a device is 8. When device ring buffer is full a next event will purge all the ring buffer and a special event to warn the program is generated with all field set to 1:

special event in case of purge: `Count= (-1) C= $ff N= $ff`

`byte_count` = see Lynx O.S. reference manual for usage of read

The NODAL CAMAC functions interface:

The NODAL interpreter got extension of its function to provide NODAL user with the CAMAC driver facilities. The on line documentation for that purpose is available by typing under a nodal session the man followed of the function name. The whole NODAL man pages can be printed out from a Console station as any system man pages.

The NODAL function available are:

GCAMAD to encode CAMAC address

SCAM to perform a single camac accesses

CAMDR to perform a sequence of CAMAC accesses

MCAMT to perform a CAMAC block access

SEE
the NODAL man pages manual
PS/CO Note 91-0020
by F. Perriollat,G. Cuisinier, A. Gagnaire
OR
under any console station
ask the man facility to display on line
documentation of any NODAL function:
>man *function-name*

Serial Camac VME specifications summaries:

hardware:

- VME board single 6U.
- VME board, addressing short I/O, data port size 16 bits
- Register generated single Camac access, conservative mode , PIO only
- Bit and byte serial mode, 5, 2.5, 1 and 0.5 MHz selectable
- Stacking of serial demand message in a FIFO (1024 memory)
- Reply and demand messages generates maskable interrupt.

Setting of jumpers:

Base address of the module:

It is adjusted by the jumpers: [JA11 .. JA15] corresponding to the address lines [A11 .. A15] giving 32 different selectable base address for the module (Jumper JAx present means line address Ax is zero):

[JA15, JA14, JA13, JA12, JA11] = [0, 0, 0, 0, 0] for base address $0

" " = [0, 0, 0, 0, 1] " " $800

... ...

" " = [1, 1, 1, 1, 1] " " $f800

Address modifier supported by the module:

It is fully determined according to the setting of the JAM jumper, whose value is 0 when present:

JAM = [0] stand AM = $2d for supervisor data access only.

JAM = [1] stand AM = $29 0r $2d for supervisor/non-privileged data access

Driver installation :

After hardware selection of appropriated setting of the VME module the installation of software driver requires some informations corresponding to the hardware setting:

- **VME base address of the module,**
- **MC68153 interrupt vector** : this vector is given to CPU at Demand interrupt to generate the interrupt in the CPU, therefore this value should be chosen carefully in order not to collide an already allocated vector interrupt in the CPU
- **MC68153 interrupt level** : this value defines the level of the CPU triggered when an demand interrupt is generated.

The installation can be made by calling the installation program with the following syntax:

```
>sdvmeinstall -A<base address> -V<vector> -L<interrupt level>
```

with the constraints: vector value between 64 and 255

Level value 1 and 7

exemple:

```
>sdvmeinstall -A0 -V160 -L2
```

SDVME Driver system interface :

For standard usage this level is hidden by a library interface .

This interface follows the standard way of Unix for programming I/O serviced by a driver.

Device file: (associated LynxOS minor devices)

the installation processing create devices file to provide user with system resources necessary to invoke driver services, there are 3 class of this resources:

- **/dev/sdvmelRW** : system resource to call Camac access services, shared by all simultaneous user.
- **/dev/sdvmelsurv** : system resource to get synchronized with Power failure and spurious interrupt of the loop, non possible to be shared, the survey task should own this ressource.
- **/dev/sdvmel01** to **sdvmel16** : system resource to get synchronised with LAM in the loop, these resources are non sharable, they are like handle which enable the owner to call services to ask for synchronisation with LAM and to get information on incoming LAM.

File system interface:

The driver direct services interface is provided by the file system interface c.f. Lynx O.S. user's manual:

- **open** on the minor device name: to get a device descriptor to call file system function associated to the driver link to the device.
- **ioctl** : to perform a special service of driver.
- **select** : to wait for the condition of the open file descriptor and other to change.
- **read** : to read bytes from the open file descriptor.
- **close** : to get ridd of the file descriptor from a previous open.

The file definition **<sdvme.h>** must be included in the program

Camac access:

The common device file **"/dev/sdvmelRW"**must be open to get an appropriate device descriptor index for the ioctl function performing Camac access request.

example:

```
int fid_access;
...
fid_access = open("/dev/sdvmelRW", O__RDWR, 0755);
```

Single Camac access call : (SDVME_scam)

Invoked by means of the ioctl function = SDVME_scam passing an argument defined defined by the following C structure provided by the header file < sdvme.h> :

```
struct sdvmeT_scam {
 long       CamFun
 long       data;
 unsigned short      QX_response;
 unsigned short      Reg_status;
};
```

Where the caller puts parameters of call, and gets back results:

- Reg_cde = (RO.) Encoded camac function according to sdvme command register structure. See sdvme hardware manual : this is the value to put in the sdvme command register.
- data =(RW.) Data Read/Write according to camc function
- QX_response = (WO.) bit[15] is Qresponse of camac function call

  bit[14] is X response of camac function call
- Reg_status= (WO.) Hardware status of sdvme module, for further in vestigation unexpected no X response oc cured. The 14 right most bits of the short are the corresponding bits of status register of sdvme module, see in sd vme hardware manual specification of status register.

example of call:

```
struct sdvmeT_scam arg_scam;
...
cc= ioctl (fid_access, SDVME_scam, &arg_scam);
```

Driver return specific error in errno when error returned by the call:

EFAULT Wrong parameter pointer (out of range or protected)

EACCESS This file id does not stand this funstion

EINVAL Illegal parameter found in arg.

Camac access sequence call: (SDVME_pmcami)

Invoked by means of the ioctl function = SDVME_pmcami passing an argument defined by the following C structure provided by the header file < sdvme.h> :

```
struct sdvmeT_pmcami {
 int sz;
 long   *dval;
 long   *AFNC;
 int    retry;
 int    *QX;
 int    *coco;
};
```

Where the caller puts parameters of call, and gets back results:

- sz = number of single camac command in the table
- dval= table of data for each of the camac function of the sequence Read/Write/Test to be executed
- AFNC= table of the Camac command sequence encoded according to sdvme command
- retry = if > 0 number of retry on each Camac command while no Q response

  if = 0 no retry, check

  if <0 no retry , ignore Q response
- QX= completion array: QX[i] = 0 if ok, 170 if no X response, 169 if no Q response.
- coco = global completion code: 0 if no error, value of error occuring at stop.

The driver returns a specific error in errno when error returned by the call:

EFAULT Wrong parameter pointer (out of range or protected)

EACCESS This file id does not stand this funstion

EINVAL Illegal parameter found in arg.

example of call:

```
struct sdvmeT_pmcami arg_pmcami;
...
cc= ioctl (fid_access, SDVME_pmcami, &arg_pmcami);
```

Repetitive Camac action call : (SDVME_mcamt)

Invoked by means of the `ioctl function= SDVME_mcamt` passing an argument defined by the following C structure provided by the header file < sdvme.h> :

```
struct sdvmeT_mcamt {
        int sz;
        long   *dval;
        long   AFNC;
        int retry;
        int  *perf;
        int  *coco;
       };
```

Where the caller puts parameters of call, and gets back results:

- `sz` = number of time the camac action will be repeated.
- `dval` = table of data for each Camac action.
- `AFNC`= Camac function encoded according to sdvme command to be repeated.
- `retry` = meaning according to value:
  - if > 0 number of retry,on each Camac command while no Q response
  - if = 0 no retry, check Q response
  - if <0 no retry , ignore Q response
- `perf` = number of action performed
- `coco` = global completion code: 0 if no error, value of error occuring at stop.

The driver returns a specific error in `errno` when error returned by the call:

EFAULT Wrong parameter pointer (out of range or protected)

EACCESS This file id does not stand this funstion

EINVAL Illegal parameter found in arg

example of call:

```
struct sdvmeT_mcamt
...
cc= ioctl (fid_access, SDVME_mcamt, &arg_mcamt);
```

Connection to a Camac LAM:

The system resources to invoke this services are non sharable, the device file available are:

/dev/sdvmelsurv is the unique resource to get automatically connected with spurious LAM or power failure LAM and to get synchronised with these incoming LAM. This device file is dedicated to the task surveying the Camac loop.

/dev/sdvmel01 to **/dev/sdvmel16** are the resources to ask connection with LAM in the loop, and get synchronised with these incoming LAMs.

To invoke the associated services the resource must be opened in order to get a device descriptor identificator required for the further file system calls:

example:

```
int fid_synchro;
...
fid_synchro = open("/dev/sdvmel01", O__RDONLY, 0755);
```

Request for connection with a LAM of a Camac Slot :

Invoked by means of the ioctl function = **SDVME_connect** passing an argument defined by the following **C** structure provided by the header file < sdvme.h> :

```
struct sdvmeT_connect {
        short C, N;
        long F1_CamFun, F1_Data;
        long F2_CamFun, F2_data;
        };
```

Where the caller puts parameters of call, and gets back results:

- C = Crate number of the LAM source to connect.
- N = Slot number in the crate of the LAM source to connect.

The following parameters allow the caller to let perform by the Interrupt service routine of the driver, a camac action, up to 2 actions, as soon as the LAM occured. This is used to disable the LAM source.

- F1_CamFun = Camac function 1 to be performed if non zero by the driver when receiving the LAM
- F1_Data = associated data to Camac function specified by F1_CamFun
- F2_CamFun = Camac function 2 to be performed if non zero and if F1_CamFun non zero as well, by the driver when receiving the LAM
- F2_Data = associated data to Camac function specified by F2_CamFun

The driver returns a specific error in errno when error returned by the call:

EACCESS The file id does not start the connection

ENODEV The file id does not [illegible] frequency between installation program and actual [illegible]

EWOULDBLOCK driver fatal error (internal table corruption detected)

EINVAL Illegal parameter found in arg.

EISCONN Try to connect to a LAM already in use by another user.

example of call:

```
struct sdvmeT_connect arg_connect;
```

```
...
cc= ioctl (fid_access, SDVME_connect, &arg_connect);
```

Request for disconnection : (SDVME_disconnect)

Invoked by means of the ioctl function = **SDVME_disconnect** passing an argument defined by the following C structure provided by the header file < sdvme.h> :

```
struct sdvmeT_connect {
        short C, N;
        long F1_CamacFun, F1_Data;
        long F2_CamacFun, F2_data;
        };
```

Where the caller puts parameters of call, and gets back results:

- C = Crate number,of the LAm source to disconnect.
- N = Slot number in the crate of the Lam source to disconnect.

Other parameters = non significant because ignored .

The driver returns a specific error in errno when error returned by the call:

EACCESS The file id does not stand this cunstion

ENODEV The file id does not exist for the driver (discrepancy beetween installation program and actual configuration of the driver)

EWOULDBLOCK driver fatal error (internal table corruption detected)

EINVAL Illegal parameter found in arg.

EISCONN Disconnect refused: either such connection or no owner of the connection.

example of call:

```
struct sdvmeT_connect arg_connect;
...
cc= ioctl (fid_access, SDVME_disconnect, &arg_connect);
```

Getting synchronised with a LAM : (select, read)

Once the program get connected it can wait for incoming LAM from the specified camac module.

If the program wants to wait for incoming event from different devices it has to use the standard Unix way: the **select** function which tell it which device got an event.

If the program uses only the camac events source it can wait by **read** on the device:

- one or more events arrived, the read returns the event in the buffer
- nothing arrived, the read blocked on wait until one event comes or time out occured

The buffer must be big enough to receive a complete 4 bytes event, the buffer will receive as many complete events, he can get, as there are in the device.

The design of an event is given by the following C structure:

```
struct sdvmeT_Event {
                        short    flag;
                        unsigned char C;
                        unsigned char N;
} ;
```

Where :

Flag = serial number of corresponding LAM since the connect was done.

C = Crate number of the corresponding LAM source.

N = Slot number of the corresponding LAM source.

```
/*      %( example of C syntax usage:  */
        ...
#include <sdvme.h>
        ...
        int fid_access;
        int fid_synchro;
        struct sdvmeT_connect arg_connect;
        struct sdvmeT_Event Event;
        int cc;
        ...
        /* open the channel to perform the camac access and
           the channel to [illegible] communication with the LAM
           from the module */
        fid_access = [illegible](DWR,0755);
        ...
        fid_synchro = open("/dev/sdvme101",O_RDONLY,0755);
        ...
        /* ask to be connected on the incoming LAM from the
           Camac modules  */
```

```
                cc = ioctl(fid_synchro, SDVME_connect, &arg_connect);
                ...
                /* ask to get synchronised:
                   the read will wait for LAM to come and
                   give  back the first event in the buffer */
                cc = read(fid_synchro, &Event, sizeof(Event));
                ...
                /* Then Process the received event */
                ...
/*              )& */
```

Miscellanious functions :

Control of driver behaviour :

single ioctl functions , to control the behaviour of the driver:

IOCTL whose argument is ignored, the supported function code are:

SDVME_nowait	if no event in device the read will not wait, it return 0 byte
SDVME_wait	default option, make the read wait if no event in device

Request to get information from a Slot:

Invoked by means of the ioctl function = **SDVME_ SlotInfo** passing an argument defined by the following **C** structure provided by the header file < sdvme.h> :

```
struct sdvmeT_SlotInfo {
        short C, N;
        int   Owner;
        int   count;
        long F1_CamacFun,F1_Data,F2_CamacFun,F2_data;
        };
```

Where the caller puts parameters of call, and gets back results:

C = Crate number of the Slot.

N = Slot number in the crate.

Owner = Index of device if connected, -1 if not connected

Count = number of LAM having occured from this Slot.

F1_CamFun = Camac function 1 to be performed if non zero by the driver when receiving the LAM

F1_Data = associated data to Camac function specified by F1_CamFun

F2_CamFun = Camac function 2 to be performed if non zero and if F1_CamFun non zero as well, by the driver whenreceiving the LAM

F2_Data = associated data to Camac function specified by F2_CamFun

Driver returns specific error in errno when error returned by the call:

EACCESS The file id does not stand this cunstion

ENODEV The file id does not exist for the driver (discripancy beetween installation program and actual configuration of the driver)

EWOULDBLOCK driver fatal error (internal table corruption detected)

EINVAL Illegal parameter found in arg

EISCONN Try to connect to a LAM already in use by anotheruser.

example of call:

```
struct sdvmeT_
...
cc= ioctl (fid_access, SDVME_connect, &arg_connect);
```

11

TSVME404 - GPIB interface driver

Author: Nicolas de Metz-Noblat

Introduction

The TSVME404 is a simple VME board allowing the connection of a GPIB (IEEE 488) bus. This board has no local processor, but is delivered with 16Kb RAM and 16Kb ROM with a simple firmware. The GPIB access is done through a TMS9914A chip.

This driver was developped without using the local firmware, neither using the local RAM, because we wanted a multi-user interface and dont want to rely on an external untested RAM.

By definition of the GPIB protocol, two main modes of operation can arise: either this board acts as the bus controller, or it acts as a slave board. This is specified when installing the driver, but if the jumpers allows it, it is possible to dynamically exchange the controller status with another computer on the same bus.

As on any Unix system, access to the driver is achieved via the open, read, write, ioctl, select and close system call. On open, a special file (usually /dev/gpib) whose major device number correspond to the driver is opened. In order to simplify the usage of the GPIB from user's programs and to allow concurrent usage of the bus by different tasks, several modes of operation are supported:

In controller mode, the minor device zero correspond to the master device, i.e. allows an access to the whole bus functionnalities, but for practical reasons, this kind of open is restricted to a single task (only one task can be woken up by SRQ and interpret the result of a parallel poll). Several tasks can open different devices (whose just minor device number differs) at the same time and be woken up by the driver on SRQ of the uniq instrument to which they are related. by convention, the minor device is equal to the device number+1 (and usually the special file name is /dev/gpib.xx, where xx is the station number on th bus).

In slave mode, either a single task can be blocked in wait on the master device, or several tasks can wait for different subaddresses.

Hardware settings summary

Following jumpers can be set on the board (see reference manual):

The S1 jumper allows the choice to force the board to be system controller or slave, or to leave the choice to the software ("auto"). The prefered switch setting is this last position, i.e. the software controlled one.

The S2 jumper enables selection of the type of output for the interface circuit of the GPIB data lines. In "T.S." position, the buffer outputs are Three-State type, except during "parallel poll" cycles in which they automatically switch to an open-collector type. In the "C.O. position, the buffer outputs are of open-collector type.

The S3 jumper(s) allows to choice the software interrupt level. This will affect the overall system response time and should be choosen carefully.

The S4 jumpers choice the board address and must be coherent with the initialisation software.

The S5 jumper allows to restrict the board access to supervisor mode, and since this is a true driver, the prefered position is the "Sup" position.

The five front-panel switches allow the choice of the board GPIB address on the bus.

Driver initialisation

When installing the driver, it does require the following informations:

- VME base address of the device (see jumpers S4).

- MC68030 interrupt vector: This number should be choosen carefully in order to avoid conflicts with other boards. Preferred default value is 120.

- Initial state as controller or slave GPIB station.

The driver is normally installed by the **dynaminst** program. This one does require some informations found in the file **/etc/insttab**.

Normal usage:

normal user will just use the driver as follow:

```
#include <t404.h>                    /* special tsvme404 ioctl definitions */
int dev;                             /* buffered device */
dev = open("/dev/gpib.21","r");      /* talk with station 21 */
write(dev., buf, len);               /* send command to instrument */
read(dev, buf, sizeof(buf));         /* read instrument response */
close(dev);
```

Software specialist usage:

In order to have full access to the gpib functionnalities,

```
#include <t404.h>                    /* special tsvme404 ioctl definitions */
int dev;                             /* buffered device */
```

```
dev = open("/dev/gpib","r");          /* reserve the whole bus */
ioctl(dev, T404_LISTEN, 21);          /* set station 21 as listener */
write(dev, buf, len);                 /* send command to instrument */
ioctl(dev, T404_UNLISTEN, 21);        /* stop any listener */
n = ioctl(dev, T404_SPOLL, 21)        /* poll station 21 */
ioctl(dev, T404_TALK, 21);            /* set station 21 as talker */
read(dev, buf, sizeof(buf));          /* Read instrument response */
ioctl(dev, T404_UNTLK, 21);           /* stop any talker */
close(dev);
```

Ioctl special function codes:

General ioctl calls:

T404_DEFEOC define end of transfer conditions. The int parameter is interpreted as follow:

- the MSB byte is a flag to generate EOI on the last byte of each write command.

- the two LSB bytes define the End of Line character(s):
  - 0xffff means end only on EOI or byte count,
  - 0xyyff means stop on reception of char yy
  - 0xyyzz means stop on reception of char yy followed by char zz (usually 0x0d0a).

T404_BUS_STATUS read the status of the HPIB bus lines.

T404_TMS9914 allows to send special commands to the tms9914 chip.

ioctl functions restricted to controller mode:

The following ioctl functions are available to modes, but restricted to the opened device if not on the master device.

T404_CLEAR Device clear. This command is either a general device clear command if applied on the master device with parameter -1, or a selected device clear of the specified instrument.

T404_LOCAL Set specified instrument into local mode.

T404_REMOTE Set specified instrument into remote control mode.

T404_LOCAL_LOCKOUT Lockout the Local/Reset command of the instrument.

T404_SIGNAL Define the signal to be sent on SRQ from the opened instrument.

T404_SPOLL Serial poll opened instrument

ioctl function restricted to the master device:

T404_LISTEN put specified instrument in listener mode.

T404_TALK put specified instrument in talker mode.

T404_SEND_SECOND send a secundary device address.

T404_STANDBY Let the requested operation execute. (this is normally used once a listner and a talker have been selected and we are not interested in listening to the transfer).

T404_UNLISTEN Stop any listener.

T404_UNTALK Stop any talker.

T404_IFC Generate InterFace Clear signal.

T404_PASS_CONTROL Pass control to another controller.

T404_GET Send Group Execute Trigger command.

T404_PPCONF Configure parallel poll on device

T404_PPUNC Unconfigure parallel poll

T404_PPDIS Disable parallel poll

T404_PPOLL Execute a parallel poll

ioctl functions restricted to slave device:

T404_SRQ Request for service

T404_STSPOL Prepare Serial Poll response

T404_STPPOL Prepare Parallel Poll Response

T404_READ_SECUND Fetch specified secundary device address

Waiting for SRQ from a device.

A task driving an instrument that can generate an SRQ has to open this HPIB device and to issue an ioctl(d,T404_SIGNAL,signum), where signum is a signal that will be generated to the requesting if the polling of this instrument has the bit 6 on.

Any instrument that can generate SRQ should be serviced by a task that will handle it.

Usage example: HP5335A universal counter.

```
/* ========================================================== */
/* HP5335-A universal Frequency counter test program          */
/* ========================================================== */
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <signal.h>
#include <fcntl.h>

#include <t404.h>

#define ASK(t,f,v) {printf(t); fflush(stdout) ; scanf(f, v);}
#define ASKOK() {printf("Ok ?"); fflush(stdout); getchar();}
#define ERR(l,t) {perror(t); exit(l);}
#define SYSERR (-1)

#define SIGSRQ SIGUSR1          /* Our signal for SRQ */
int   hp5335;                   /* Open device number */
int   station;                  /* HPIB station number for 5335-A */
char  device[80];               /* Special file name */
char  dummy[80];                /* Dummy response buffer */
int   cr;
int   service_request;

/*************************************************************/
/* SRQ signal handler                                        */
/*************************************************************/
static sig_srq ()
{ printf ("SRQ arrived\n");
  ++service_request;
}
/*************************************************************/
/* Write command to hp5335                                   */
/*************************************************************/
write_command (txt)
char *txt;
{ char  txt1[128];              /* Intermediate buffer */
  int   len;
  sprintf (txt1, "%s\r\n", txt);
  len = strlen (txt1);
  if ((cr = write (hp5335, txt1,
}
/*************************************************************/
/* Test 2 : **** LOCAL/RESET check                           */
/*************************************************************/
test_2 () {
  if ((cr = ioctl (hp53
    ERR (2, "hp5335 Sel-
  if ((cr = ioctl (hp5335,
    ERR (2, "Remote");
  write_command ("IN");                 /* initialize */
  printf ("\n");
  printf ("Press LOCAL/RESET on 5335A front panel and verify\n");
  printf ("that REMOTE LED goes out.\n");
  ASKOK ();
}
```

```
/*************************************************************/
/* Test 3 : **** LOCAL LOCKOUT check                  ****  */
/*************************************************************/
test_3 () {
  if ((cr = ioctl (hp5335, T404_REMOTE, 0)) == SYSERR)
    ERR (2, "Remote");
  if ((cr = ioctl (hp5335, T404_LOCAL_LOCKOUT, 0)) == SYSERR)
    ERR (2, "Local Lockout");
  printf ("Press LOCAL/RESET on 5335A front panel and verify\n");
  printf ("that REMOTE LED stays ON.\n\n");
  ASKOK ();
  if ((cr = ioctl (hp5335, T404_LOCAL, 0)) == SYSERR)
    ERR (2, "hp5335 Local mode");
}
/*************************************************************/
/* Test 4 : **** 'WAIT' & 'SRQ' mode test             ****  */
/*************************************************************/
test_4 () {
  int   (*sig_old) ();       /* saved alarm signal handler */
  int   i;
  char  buf[1024];
  printf ("Verify that following LEDs are lit for 5 measures:\n");
  printf ("    - TALK\n    - LISTEN\n    - SRQ\n    - REMOTE\n");
  ASKOK ();
  write_command ("IN,GA.02");   /* Initialize, Gate adjust */
  sig_old = signal (SIGSRQ, sig_srq);
  service_request = 0;
  if ((cr = ioctl (hp5335, T404_SIGNAL, SIGSRQ)) == SYSERR)
    ERR (2, "Enabling SRQ requests");
  /* Wait to Send mode on, Enable SRQ */
  write_command ("WA1,SR1");
  sleep (1);
  for (i = 0; i < 5; i++) {
    if ((cr = read (hp5335, buf, sizeof (buf))) < 0)
      ERR (2, "reading");
    buf[cr] = 0;
    printf ("      %s\n", buf);
    if ( i<4) {
      write_command ("RE");       /* reset for new measurement */
      sleep (1);
    }
  }
  write_command ("SR0");
  ASKOK ();
}
/*************************************************************/
/* Test 5 : **** Teach - Lea                          ****  */
/*************************************************************/
test_5 () {
  int   i,l;
  char  buf[512], buf1[512];
  printf ("Verify that 53                                  n");
  if ((cr = ioctl (hp53                                    ERR)
    ERR (2, "Remote");
  /* select function 1, set scale value 11222448800 */
  write_command ("FU1,MS1122448800");
  ASKOK ();
  printf ("Reading current setting\n");
  write_command ("PQ");
  if ((cr = ioctl (hp5335, T404_DEFEOC, -1)) == SYSERR)
    ERR (2, "Define end of transfer conditions");
```

```
  for (;;) {
    if ((l = read (hp5335, buf, 30)) < 0)
      ERR (2, "reading");
    if ((buf[23] == 17) && (buf[24] == 34) && (buf[25] == 68)
        && (buf[27] == 0) && (buf[28] == 0))
      break;
  }
  printf ("Verify that 5335A displays 100. 000 00 -9 with S\n");
  printf ("and function PER A LED's ON and GATE LED flashing\n");
  write_command ("FU9");
  ASKOK ();
  printf ("Rewriting setting\n");
  buf1[0]='P';
  buf1[1]='B';
  for (i=0;i<l;i++)
   buf1[i+2]=buf[i];
  if ((cr = write (hp5335, buf1, l+2)) < 0)
      ERR (2, "writing");
  printf ("Verify that 5335A displays: '11. 224 49 +15 Hz'.\n");
  ASKOK ();
}
/*************************************************************/
/* Main program                                              */
/*************************************************************/
main (argc, argv)
int   argc;
char **argv;
{ int   i;
  printf ("************************************\n");
  printf ("* 5335-A HPIB verification program *\n");
  printf ("************************************\n\n");
  for (;;) {
    ASK ("Enter HP 5335-A station number: ", "%d", &station);
    if ((station >= 0) && (station < 31))break;
    printf ("Illegal HPIB station number !\n");
  };
  sprintf (device, "/dev/hpib.%d", station);
  if ((hp5335 = open (device, O_RDWR, 0)) < 0) ERR(2,device)
  printf ("Connect Time Base Out from rear panel to Input A.\n");
  printf ("Set 'GATE ADJ' to CCW and Channel A & B input to:\n");
  printf (" Preset, 50 Ohms, X1, DC and positive slope\n");
  ASKOK ();
  printf ("Test 2 : **** LOCAL/RESET check        ****\n");
  test_2 ();
  printf ("Test 3 : **** LOCAL LOCKOUT check      ****\n");
  test_3 ();
  printf ("Test 4 : **** [illegible]\n");
  test_4 ();
  printf ("Test 5 : **** Teach - [illegible] test  ****\n");
  test_5 ();
}
```

12

ICV196VME - ITX interface driver

Authors : Friedtjof Berlin
Alain gagnaire

Introduction

This ICV196 VME module contains 96 I/O lines, arranged as 12 ports of 8 lines. The lines 0 to 15 (ports 0 and 1) are used for external interrupts and these are the lines which are supported by the driver.

The driver supports up to 4 of the icv196 modules, each containing 16 interrupt lines and permits a user program to connect to one or more of the 4*16 interrupt lines provided by the modules. Which lines are available depends of course on how many modules are physically installed.

The additional I/O ports contained on the icv196 module (port2 - port11) are not supported by the driver, with one exception: the direction of these ports has to be set by an **ioctl()** function on the desired module (see chapter:"Ioctl special function codes").

Before using the driver, it must be installed. The program icv196vmeinstall, which is provided with the driver, performs the installation (see chapter "Installing the driver").

On the DSC's, the driver object code, its installation program and a testprogram "testicv" is situated in the directory /usr/local/drivers/icv196. The directory /u/dscps/icv196vme contains the source code for all programs concerning the driver.

Driver interface functionality

The following is a very brief description of the calls executed in an application program to interact with the driver. For more details, see chapter "Calling the driver from a user program" and the program examples given.

Similar as for other drivers developped under LynxOS at PS, the library functions **gpevtconnect()** and **gpevtdisconnect()** are used for connecting to and disconnecting from an icv196 interrupt line. The [illegible] standardize the driver interface.

In short the following calls access the driver from a user program :

1) **gpevtconnect()** : connects to an interrupt line on the desired module

2) **gpevtdisconnect()** : disconnects from an interrupt line on the desired module

3) **read()** : waits for an interrupt from an icv196 module and returns information on the event (e. g. which line was the source of the interrupt) when it arrives. If more than one event has occcurred since the last **read()**, information on all the events (up to 8) is returned.

4) **select()** : This call is used to wait for interrupts which may come from either one of the 4 icv196 modules in the driver **or** from another source. If, in this case, it was detected that the icv196 module gave the interrupt, a **read()** call has to be executed to find which line was the source of the interrupt.

5) **ioctl()** : reads/sets parameters in the driver.

6) **open()** : this call is used for connecting to the driver if **no interrupt synchronisation** is wanted. The open is performed on the so called "service handle" which allows reading different status information from the driver and setting parameters in the driver via the **ioctl()** call.

Hardware settings summary

The icv196 module contains the following jumpers (see reference manual):

ST 1....ST 4: define the address offset of the module. The jumpers define bits ?? to ?? of the adddress offset.

ST 5: sets the board access mode; supervisor mode or general access.

ST 6: timer output on/off (the timer is not supported by the driver).

For further information on the jumpers, see the icv196 hardware manual.

Installing the driver

After having initialised the hardware settings on the icv196 module, the following data must be provided to the installation program for the driver, **icv196vmeinstall**:

. **VME base address** of the module.

. **MC68153 (CPU) interrupt vector**: this vector is given to the CPU at an interrupt demand to generate the interrupt in the CPU. It should not already be used by an other interrupt source!

. **MC68153 interrupt level**: this value defines the level of the interrupt triggered by the icv196 module.

As there can be up to 4 icv196 modules supported by the driver, the index mod given in the installation syntax below can have the values A..D representing module nr. 0..3.

The installation can be performed by calling the installation program with the following syntax:

>icv196vmeinstall -<mod>O<base address> -<mod>V<vector> -<mod>L<interrupt level>

with the constraints: 64 < vector < 255.

1 < interrupt level < 6.

example:

>icv196vmeinstall -A0500000 -AV128 -AL2

This installs module 0 with the following parameters:

Address offset: 0x0500000 (hexadecimal)
Interrupt vector: 128 (decimal)
Interrupt level: 2 (decimal)

Calling the driver from a user program

Headers/links:

In the source file of the user program, the following header files must be included:

1) /u/dscps/rtfclty/gpsychrolib.h

2) /u/dscps/icv196vme.h

In the makefile, it must be linked with the file:

/u/dscps/rtfclty/gpsynchrolib.o

The following is a closer description of the calls used in a user program to interact with the driver.

gpevtconnect(): Connect to an interrupt line

This call connects the user program to the given module and line and enables interrupts on that line.

Formal C syntax definition:

```
int gpevtconnect(type, evtval, ref_dat)
                              int type;
                              long evtval;
                              struct icv196T_UserConnect *ref_dat;
```

Syntax of a call:

```
synchro_device_id = gpevtconnect(type, 0, ref_dat)
```

where:

```
type = 2                          /*for icv196vmedriver    */
evtval = 0;                       /*ignored in this context*/

ref_dat = struct icv196T_UserConnect {
                                    unsigned char module;/*mod. number */
                                    unsigned char line;  /*line number */
                                    short mode;/*explained under read()*/
                                  }
```

If synchro_device_id <= 0, the call failed.

gpevtdisconnect: Disconnect from an interrupt line

This call disconnects the user program from a previously connected interrupt line.

Formal C syntax definition:

```
int gpevtdisconnect(type, evtval, ref_dat)
                              int type;
                              long evtval;
                              struct icv196T_UserConnect *ref_dat;
```

Syntax of a call:

```
retval = gpevtdisconnect(type, 0, ref_dat)
```

where:

```
type = 2     /*for icv196vmedriver    */
evtval = 0;  /*ignored in this context*/

ref_dat = struct icv196T_UserConnect {
                                    char module;/*module number  */
                                    char line;  /*line number    */
                                    short mode; /*ignored        */
                                  }
```

If filedescriptor < 0, the call failed.

select, read : Getting synchronised with an external interrupt

After the connections are established to the desired interrupt lines, the program can synchronize with events occurring on these lines.

For that purpose, the returned device file descriptor from the **gpevtconnect()** call allows the program to wait for events by calling **select()** or **read()**. The file descriptor returned by gpevtconnect is always the same for one program, even when **gpevtconnect()** is called several times to connect to several interrupt lines.

As mentioned in the introduction, **select()** is used to wait for interrupts which may come from **either** one of the 4 icv196 modules in the driver **or** from another source. If, in this case, it was detected that a icv196 module gave the interrupt, a **read()** call has to be executed to find which line was the source of the interrupt.

Getting synchronized by select(): this is the standard UNIX/LynxOS way to synchronize with external events. After **select()**, **read()** gives the events having occurred (the active interrupt lines) in the icv196vme module, if any. For details on **read()** see below.

The **select()** call is described in the LynxOS manual under system calls.

Getting synchronized by read():

Reading from the device file descriptor given back on **gpevtconnect()** means reading the events which have occurred since the last **read()** call. This means that the specified buffer in the **read()** call will be filled with one or more structures as described below (Event data structure).

Formal C syntax definition:

```
int read( synchro_device_id, buffer,byte_count);
          int synchro_device_id;
          char *buffer;
          int byte_count;
```

Syntax of a call:

```
err = read( synchro_device_id, buffer, byte_count);
```

Where:

err = if **read()** failed. -1 is returned

synchro_device_id = Device file descriptor given by the gpevtconnect() call (always the same for all connect the program performs)

buffer = buffer to ge[illegible] memory size required [illegible] of(icv196T_UserEvent) [illegible] gives space for one event, maximum is 8. If 8 events have occurred after a read(), and no new read() has been executed, the 9 th event will purge the driver internal buffer, and a special event to warn the program is generated with all fields set to 1:

special event in case of purge: count= (-1) module= $ff line= $ff

byte_count = gives the number of bytes read.

The cumulative mode:

When calling gpevtconnect(), the parameter mode in the structure icv196T_UserConnect has two possible values; 0 = non-cumulative mode and 1 = cumulative mode. The mode chosen determines how to interprete the parameter count in the structure icv196T_UserEvent described below.

The non-cumulative mode means that count indicates the number of events having occurred since the interrupt line was enabled.
The cumulative mode means that count indicates the number of events having occurred since the last `read()` call.

In this way, using the cumulative mode, it can be tested if a program has missed interrupts since the last `read()`. It also prevents the internal event buffer in the driver to fill up with events of the same kind. This may be useful if it's not neccessary to read all the events of this kind, and the driver is delivering more events than the connnected task(s) can read.

Event data structure:

The events which can be read through the `read()` call have the following structure:

```
struct icv196T_UserEvent {
                          short count;
                          unsigned char module, line;
                          };
```

Where:

`count` = event counter (see expl. above)

`module` = module where the interrupt occured

`line` = line in the module where the interrupt occurred.

ioctl : Reading/setting parameters in the driver

The **ioctl()** system call provides an interface to the parameters in the driver which can be read or set from a user program.

Examples on how to execute `ioctl()` calls on the icv196 driver can be found in the file /u/dscps/icv196vme/testicv.c on the workstations in the ITX control network

Formal C syntax definition

```
ioctl( synchro_device_id, request, arg);
            int synchro_device_id;
            int request;
            char *arg;
```

Syntax of a call:

```
err = ioctl( synchro_device_id, request, arg);
```

Where:

`err` = if read() failed, -1 is returned, success = 0

`synchro_device_id` = Device file descriptor index given by the gpevt-connect call (always the same for all connect the program performs)

`request` = code for driver ioctl action

`arg` = pointer to a structure containing data to be read/written. This structure may be different for different values of request (see below).

Ioctl special function codes:

The ioctl special function codes below replace the parameter `request` in the `ioctl()` call.

For the description of the data structures used to transmit the data to and from the driver, see the end of this section. These structures are all defined in the file "icv196vme.h".

`ICVVME_getmoduleinfo`

Function: get general information on the installed module

Variable to transmit data:

```
struct icv196T_ModuleInfo arg[4];/*array of 4 module info blocks*/
```

`ICVVME_connect`

Function: connect to an interrupt line on a given module. Normally, this call is not used by a program, as it is integrated in the `gpevtconnect()` call described above.

Variable to transmit data:

struct icv196T_connect `arg;`

`ICVVME_disconnect`

Function: disconnect from an interrupt line on a given module. Normally, this call is not used by a program, as it is integrated in the `gpevtdisconnect()` call described above.

Variable to transmit data:

```
struct icv196T_connect arg;
```

`ICVVME_dflag`

Function: Toggles (sets/resets) debug flag to allow monitoring of messages from the driver on serial channel 3 of the DSC

`ICVVME_nowait`

Function: sets non-blocking read, meaning that the read() call returns immediately whether data has been read or not.

`ICVVME_wait`

Function: sets blocking read, meaning that the read() call returns after the data has been has arrived or on timeout.

ICVVME_setupTO

Function: sets timeout for read(). The value should be given in milliseconds. Default: 6000 ms. The old timeoutvalue is returned.

Variable to transmit data:

```
int *arg;
```

ICVVME_intcount

Function: reads the interrupt counters for all lines in the module given in the argument.

Variable to transmit data:

```
struct icv196T_Service arg;/* the interrupt counter values are  */
                           /* returned in arg.data[0..15]       */
```

ICVVME_setreenable

Function: When the reenableflag is set, the interrupt line is automatically reenabled after an interrupt to be ready for the next event.

Variable to transmit data:

```
struct icv196T_connect arg;
```

ICVVME_clearreenable

Function: When the reenableflag is cleared, the interrupt line is **not** automatically reenabled after an interrupt. In this mode, the line must be enabled with ioctl(..., ICVVME_enable, ...) after each interrupt to be ready for the next event.

Variable to transmit data:

```
struct icv196T_connect arg;
```

ICVVME_enable

Function: Enables an interrupt line.

Variable to transmit data:

```
struct icv196T_connect arg;
```

ICVVME_disable

Function: Disables an interrupt line

Variable to transmit data:

```
struct icv196T_connect
```

ICVVME_readio

Function: The I/O ports contained on the icv196 module (port2 - port11) which are not used as interrupt lines, can be written to and read from like standard I/O lines. Library routines are available; Chapter 9 in this manual: VME - addressing facilities library or /u/dscps/icv196vmefclty. The direction of these ports can only be set by the

ioctl function ICVVME_setio. In order to read back the status of the ports (input/output), ioctl (...,ICVVME_readio,....) is called.

Variable to transmit data:

```
struct icv196T_Service arg; /* arg.data[0] contains the bit    */
                            /* pattern corresponding to the I/O*/
                            /* port status.  Bit 2 = port 2,   */
                          /* etc...   input = 0, output = 1   */
```

Note: line number ignored as argument for this call, as the call is not line specific.

ICVVME_setio

Function: Set I/O port direction (see above).

Variable to transmit data:

```
struct icv196T_Service arg;/* arg.data[0]:I/O port to be set    */
                           /* arg.data[1]:direction of the port */
                           /*   0 = input,     1 = output       */
```

Note: line number ignored as argument for this call, as the call is not line specific.

ICVVME_intenmask

Function: read the interrupt enable mask of a icv196 module.

Variable to transmit data:

```
struct icv196T_Service arg; /*arg.data[0] contains the bit      */
                            /*pattern corresponding to the     */
                            /*interrupt enable mask.           */
                            /*Bit 0 = line 0, etc...           */
                          /*enabled = 1, disabled = 0          */
```

Note: line number ignored as argument for this call, as the call is not line specific.

ICVVME_reenflags

Function: read the reenable flags for all the interrupt lines on a icv196 module.

Variable to transmit data:

```
struct icv196T_Service arg; /* the reenable flags are returned */
                           /* in arg.data[0..15]               */
                            /* 0 = not set, 1 = set            */
```

Note: line number ignored as argument for this call, as the call is not line specific.

ICVVME_gethandleinfo

Function: Get information on a user handle. 1) The process id of the process using the handle 2) The connected interrupt line

Variable to transmit data:

```
struct icv196T_HandleInfo  arg;
```

Structures defined in icv196vme.h for transmitting data via `ioctl()`:

```
struct icv196T_UserConnect {  unsigned char module;
                              unsigned char line;
```

```
                                short    mode;
                              };
struct icv196T_UserEvent {    short    count;
                              unsigned char module;
                              unsigned char line;
                            };
struct icv196T_UserLine { unsigned char group; /*module number */
                          unsigned char index; /*line number   */
                        };

struct icv196T_Service {
                           unsigned char module;
                           unsigned long data[ICV_IndexNb];
                        };

struct icv196T_ModuleParam{
          unsigned long base; /*offfset in VME address space   */
          unsigned long size; /* size of used VME space        */
          unsigned char vector[icv_LineNb]; /* interrupt vect. */
                                  /* for each interrupt line */
          unsigned char level[icv_LineNb];/* interrupt level   */
                                /* for  each interrupt line*/
                        };

struct icv196T_HandleLines {
                   int pid; /* id of process using the handle */
                   struct icv196T_UserLine lines[ICV_LogLineNb];
                            };

struct icv196T_HandleInfo {
          struct icv196T_HandleLines handle[ICVVME_MaxChan];
                           }

struct icv196T_ModuleInfo {
          int ModuleFlag;
          struct icv196T_ModuleParam ModuleInfo;
                           };
```

open : Drivers access exclusive for reading/setting parameters

The open() call is used for connecting to the driver if an **interrupt synchronisation** is wanted. The open is performed with the "service handle" which allows reading different status information from the driver parameters in the driver via the `ioctl()` call.

Syntax of a call:

```
 static char path[] = "/dev/icv196service"       serv-
ice_filedesc = open(path,O_RDONLY);
```

program example: Synchronizing with events

The following program gives an example on how to use the driver for synchronising with events. The task connects to two different interrupt lines and waits for interrupt on these two lines. Depending on which line gives the interrupt, different parts of the program is executed.

```
#include <icv196vme.h>

#define SIZE 32            #define INT_ISR 0          #define
INT_RT 1          extern int gpevtconnect();          extern  int
gpevtdisconnect();

main()
{
    struct icv196T_UserConnect connct;
    int i, j, k;
    int type;
    int retval;
    int evtval;
    int synchro_fd;
    int byte_count;
    char buff[SIZE];
    struct icv196T_UserEvent *event;

    connct.module = 0;          /* module numbers: 0 to 3      */
    connct.line = INT_ISR;      /* line numbers: 0 to 15       */
    connct.mode = 1;            /* cumulative mode             */
    type = 2;                   /* type = 2 for icv196         */
    evtval = 0;                 /* no initialization of event */

    /* connect to line 0 */
    if ((synchro_fd = gpevtconnect(type, evtval, &connct))<0){
       perror("could not connect to line 0\n");
       exit(1);
    }
    connct.line = INT_RT;

    /* connect to line 1 *
    if ((synchro_fd = gpevt nne t (type, evtval, &connct))<0){
       perror("could not connect to line 1\n");
       exit(1);
    }

    for (i =0; i < 1000; i++
        for (j=0; j < SIZE; j++) buff[j] = 0;/* clear buffer */
            byte_count = SIZE;/* read max 8 events     */
           if ((retval = read (synchro_fd, buff, byte_count))
                < 0) {
               perror("could not read\n");
               continue;
           }
```

```
            event = (icv196T_UserEvent *) buff;
            if byte_count == 0      {
                    perror("timeout on read\n");
            continue;
        }
        for (;event->count != 0;event++)
        { /* treat ALL events in the buffer !*/

            if (event->line == INT_RT)
            {
             /*  TASK RT  */
                          .
                          .
                          .
            }
            if (event->line == INT_ISR)
            {
             /*  TASK ISR  */
                          .
                          .
                          .
            }
         }      /* end event buffer treatment      */
      }         /* end for loop waiting for events */
      /* disconnect from line 0 */
    if ((retval = gpevtdisconnect(type,evtval,&connct)) < 0) {
          perror("could not disconnect from line 0\n");
          exit(1);
      }
      connct.line = INT_RT;
      /* disconnect from line 1 */
     if ((retval = gpevtdisconnect(type,evtval,&connct)) < 0){
         perror("could not disconnect from line 0\n");
         exit(1);
      }
}             /* end main                          */
```

FPIPLSVME - PLS Telegram and FPI driver

Authors: Alain Gagnaire (software),
Claude dehavay (hardware)

Introduction:

The FPIPLSVME module has 2 functions :

- To receive, decode and memorize the PLS (Program Line Sequence) pulse train (maximum 1023 bits).
- To generate up to 8 different VME interrupts corresponding to the 8 trigger sources: the trigger pulse of the PLS telegram, and the 7 external lines associated to the 7 plugs of the front panel.

A full specification of the module is given by the reference manual of the VME board provided by the designers of the module:

PLS Receiver and Front Panel Interrupt VME module 80401CO

by Claude Dehavay (Cern PS-CO). PS/CO/Note 91-xxx

The software interface of the driver facilities is proposed at 2 levels:

- The direct access to the driver interface based on UNIX i/o system call. This is not recommended because it makes the user program dependent on the drivers implementation and non portable.
- The library interface which provides global functions hiding to the programmer the UNIX system call interface and the driver specific interface. This interface is available for C program and NODAL program.

Driver interface functionality:

The driver supports up to 4 fpiplsvme modules, and allows an application program to be connect to one or more of the 4*8 interrupt lines provided by the modules and to read the 4 telegrams coming from the 4 different PLS sources. The number of available lines and telegrams depends of course on how many modules are physically installed in the DSC crate.

These functions are provided via the user interface of the standard Unix file system

The access to the Pls telegram : (Read Library)

- Reading from one of the module [illegible] telegram

Connection to an interrupt : [illegible]

- Connect function command : in order to get further an event associated with a trigger of a given line from one of the fpiplsvme module. When the trigger occurred the driver will put an event,in the ring buffer of the requesting device. The event is a sequence of 4 bytes, if we represent this event by : byte [3,2,1,0] the structure of the event is:

bytes [3,2]= 1 word = number of trigger occurrences since the connection was made.

byte [1] = fpiplsvme module index ([0 ... 3]) of the trigger source.

byte [0] = Line index of the trigger ([0...7]) source.

N.B.: the connect is multiple, several device (up to 8) can get the event associated to the same trigger source.

- Disconnect function command : to get rid of a previous trigger source connection. The in coming trigger are no longer associated to the device.

Synchronisation with a trigger :

Performed with a call to the select or read function. The read will be used as well to get information on source of the incoming trigger (previously connected) .

Synchronisation with a trigger (select, read file system call):

- Select : if the device descriptor of the fpiplsvme driver, dedicated for synchronization is given in the list of a select, and if a connect was previously made on it,for one or more trigger source, the select will return when one of this trigger occurred. The knowledge of the trigger source is acquired by reading the incoming events from the ring buffer of the corresponding device descriptor.
- Read : to get synchronized with a trigger source and to read the ring buffer in order to know the trigger source. When the ring buffer is empty the call is blocking during a laps of time, after which a time out is returned. The associated buffer must be tailored to receive at least one full event i.e.: minimum 4 bytes; the buffer is fed only with an integer number of events.

FPIPLSVME Driver interface library: (fpiplslib.o , gpsynchrolib.o)

Introduction:

The library hides to the programmer the UNIX system call of the driver direct interface. The user may want to have a minimum knowledge of the system resources involved, for that purpose he has to read the next chapters describing the direct interface.

In a few words, the user has to know that this interface is based on UNIX i/o system call. To access the driver facilities, the library has to open a device file, the currently opened device file identifier is stored in the global context of the library (local to the running process) . There are two different devices involved :

- one for reading the PLS telegram. This resource is unique and shared by the different user processes, a semaphore is used to make exclusive the access to the hardware.

  For reading the PLS telegram, the library opens the device file dedicated for that purpose whose name is : **/dev/fpiplsRTgm**.

- one for the synchronization, open at the first library call. This resource is exclusive (one device resource for each different user process)

  For the synchronization request on the type `Evtsrce_fpi`, the library tries to get a free and exclusive device file, out of the set of resources dedicated for that purpose whose names are : **/dev/fpiplsFpi02** to **/dev/fpiplsFpi08** .

  For the special PLS task a dedicated device is reserved, this is : **/dev/fpiplsPls**. The selection of this resource is done at the connect using the type `Evtsrce_pls`.

N.B. : using library access makes program independent of the driver implementation and source code portable.

How to use the library :

In the program source file include the header file associated to these libraries:

```
#include  <fpiplslib.h>
#include  <gpsynchrolib.h>
```

The Makefile building the user program must include the following lines

- The general path to dsc library is defined in the Makefile as :

```
# general path to lead to the dsc library:
ROOT= /u
```

- To work with the access routine, define the library path:

```
# define path to load camaclib.o
# camaclib.o full path
FPIPLSLIB= $(ROOT) [illegible]
```

- To work with the synchronization routines, define the library path:

```
#define path to [illegible]
RTLIB= $([illegible]
```

- To tell the loader to reference [illegible], include the `$(FPIPLSLIB)` and/or `$(RTLIB)` in the compilation/link command line .

Services routines :

fpiSetTO :

Set the time out attached to a synchronization device.

Formal C syntax definition:

```
int fpiSetTO(fdid, ref_val)
    int fdid;
    int *ref_val;
```

Syntax of a call:

```
err = fpiSetTO(fdid, ref_val );
```

Where:

fdid = file descriptor index returned at the connection call.

ref_val = at call : new time out for the synchronization(in 1/100 s.)

at return : previous time out value.

PlsReadTgm :

To read the pls telegram from a module.

Formal C syntax definition:

```
int PlsReadTgm(module, ref_buffer, buffer_size)
    int module;
    short *ref_buffer;
    int buffer_size;
```

Syntax of a call:

```
err = PlsReadTgm(module, ref_buffer, buffer_size);
```

Where:

module = number of the module to read the telegram.

ref_buffer = address of a buffer, to receive the telegram in it.

buffer_size = actual size of buffer. This size must be as big as a full telegram, i.e. 32 short.

gpevtconnect, gpevtdisconnect : Synchronisation routine

These facilities are not specific for synchronization with FPIPLS events, they intend to provide DSC programs with general purpose means to get connected with external events such as: CAMAC LAM, trigger from Front panel interrupt module (ICV196), timing line events (PLS module).

gpevtconnect : Ask connection with a TRIGGER

To get synchronised with a Front panel the program must first ask to get connected to. A call to gpevtconnect function for each of the expected trigger must be done.

Formal C syntax definition:

```
int gpevtconnect(type,evt_val, ref_EvtDescr)
    int type;
    int evt_val;
        long *evt_descr
```

Syntax of a call:

```
synchro_device_id = gpevtconnect(type, 0, ref_EvtDescr);
```

Where:

type = Evtsrce_fpi the library will use a device in the pool fpiplsFpi

type = Evtsrce_pls the library will use the device : /dev/fpiplsPl

evt_val = ignored at this moment.

ref_EvtDescr = long ref_EvtDescr[5] can be redefine as an union structure pointer:

```
union U_EvtDescr {
    long element;
    struct {
            char group;
            char index;
            word mode;
    } atom;
};
```

Where:

ref_EvtDescr -> atom.group = module ([0..3]) of line to connect

ref_EvtDescr -> atom.index = line number ([0..7]) to connect

ref_EvtDescr -> atom.mode =

0 the default value is used.

1 cumulative mode, it means that only one event is generated for a trigger source. The event will notify the number of actual triggers occurred since

2 à la queue leuleu , for each trigger an event is generated in the ring buffer of the device.

The default is cumulative for the device fpiplsPls, 'à la queue leuleu' for the other.

synchro_device = returned value giving the device file identifier to be used

- when the program wants to get synchronized either by means of the system file **select** command or in a specific request on this events source by means of the **read** system file command which returns as well in the associated buffer the event identifying the trigger source.

- in the service routines : `fpiSetTO`.

gpevtdisconnect : Ask disconnection from a TRIGGER

To get rid of a previously connected event source the program must ask to be disconnected from this trigger.

Formal C syntax definition:

```
int gpevtdisconnect(type, ref_EvtDescr)
    int type;
    long *evt_descr
```

Syntax of a call:

```
synchro_device_id = gpevtdisconnect(type, ref_EvtDescr);
```

Where:

`type` = `Evtsrce_fpi` the library will use a device in the pool fpiplsFpi

`type` = `Evtsrce_pls` the library will use the device : `/dev/fpiplsPl`

`ref_EvtDescr` = same as for gpevtconnect

select, read : Getting synchronised with TRIGGER event

After connections have been established with the expected trigger sources, the program has to be synchronised with the events associated to the lines and has to identify from which source they are coming from.

For that purpose, the returned device file identifier from gpevtconnect call, which is the same for all the connection the program performs, allows the program to wait for these events either by the **select** or by the **read**, according to its needs.

Events data structure:

These events are pushed in the corresponding device file and can be read by the program, the data structure of the event is as follows:

```
struct fpiplsT_Event {
                short count;
                unsigned char group, index;
                };
```

Where:

`count` = according to requested mode at connection:

- cumulative mode : gives the number of trigger re[illegible]

- 'a la queue leu leu mode : gives the serial number of trigger since connection.

`group` = module number of the associated trigger .

`line` = line index of trigger in the group..

Getting synchronised by a select call: this is the standard UNIX way, after return from the `select` the program must identify which device the event is coming from.

If the device file is the one of the TRIGGER event source, to know which out of all the connected TRIGGER source generated an interrupt, the program must perform the reading of the device which returns in the buffer the data identifying the source : see below getting synchronised by read.

Getting synchronised by a read call :

Reading from the device file descriptor given back on gpevtconnect is waiting for trigger to occur or getting in the buffer associated to the read the event data identifying the source of event.

This source depends on the kind of device generating the event, in case of the fpiplsvme module the structure of trigger events is described above in the paragraph **Event data structure.**

Formal C syntax definition:

```
int read(synchro_device_id, buffer,byte_count);
          int synchro_device_id;
          char *buffer;
          int byte-count;
```

Syntax of a call:

```
err = read( synchro_device_id, buffer, byte_count);
```

Where:

`err` = see Lynx O.S. reference manual for usage of read

`synchro_device_id` = Device file descriptor index given by the `gpevtconnect` call (always the same for all connect the program performs)

`buffer` = buffer to get event descriptor: minimum size required = `sizeof(sdvmeT_Event)` i.e. = 4 bytes. if the buffer is bigger it can receive as many already received event as the buffer can completely contents. The actual maximum number of event awaiting a read for a device is 8. When the device ring buffer is full, the arrival of one more event will purge all the ring buffer and generate special event to warn the program :

special event in case of purge: `Count= (-1) C= $ff N= $ff`

`byte_count` = see Lynx O.S. reference manual for usage of read

The NODAL FPIPLS functions interface:

The NODAL interpreter has now new functions to provide NODAL user with the FPIPLS driver facilities. The on line documentation for that purpose is available by typing under a nodal session the man followed of the function name. The whole NODAL man pages can be printed out from a Console station as any system man pages.

The available NODAL functions are:

FPICNCT , PLSCNCT to get connected

FPIDCNCT, PLSDCNCT to get rid of a previous connection

PLSRTGM to read the pls telegram from a module

SEE

the NODAL man pages manual

PS/CO Note 91-0020

by F. Perriollat,G. Cuisinier, A. Gagnaire

OR

under any console station

ask the man facility to display on line

documentation of any NODAL function:

>man *function-name*

FPIPLSVME specifications summaries:

hardware:

- VME board single 6U.
- VME board, addressing short I/O, data port size 16 bits

Setting of jumpers:

Base address of the module:

It is adjusted by the jumpers: SW1..SW4 corresponding to the address lines [A15 .. A12] giving 16 different base addresses for the module (Jumper SWx present means line address Ax is zero):

[SW1, SW2, SW3, SW4] = [0, 0, 0, 0, 0] for base address $0

" " = [0, 0, 0, 0, 1] " " $8000

... ...

" " = [1, 1, 1, 1, 1] " " $f000

Address modifier supported by the module:

Driver installation :

After hardware selection of appropriated setting of the VME module, the installation of software driver requires some informations corresponding to the hardware setting:

- **VME base address of the module,**
- **MC68153 interrupt vector** : this vector is given to CPU at Demand interrupt to generate the interrupt in the CPU, therefore this value should be chosen carefully in order not to collide an already allocated vector interrupt in the CPU
- **MC68153 interrupt level** : this value defines the level of the CPU triggered when a demand interrupt is generated.

The installation can be made by calling the installation program with the following syntax:

```
>fpiplsvmeinstall \
<-XO<base address> -XV<vector [64..255] > -XL<int. level[1..5]>..>
```

where `X = A,B,C or D` and several sets of declaration in sequence according to the actual module in the configuration.

module declared as `A` gets the module number ,named also group number, `0`,

module `B` group number `1`,

module `C` group number `2`,

module `D` group number `3` .

exemple:

```
>fpiplsvmeinstall -C0e000 -[illegible] -A[illegible] -A[illegible]
```

corresponding to a configuration with [illegible] module group 0 and module group 2

FPIPLSVME Driver system interface :

For standard usage this level is hidden by a library interface .

This interface follows the Unix standard way for programming I/O devices serviced by a driver.

Device file: (associated LynxOS minor devices)

The device installation create devices file to provide user with system resources needed to invoke driver services. There are 3 classes of these resources:

- **/dev/fpiplsRTgm**: system resource to read the PLS telegram, shared by all simultaneous users.
- **/dev/fpiplsPls** : system resource dedicated for the PLS task and to be synchronised with trigger.
- **/dev/fpiplsFpi02** to **fpiplsFpi08** : system resource to be synchronised with triggers, these resources are non sharable, they are like handle which enable the owner to call services for synchronisation with trigger , to get informations on incoming triggers.

File system interface:

The driver direct services interface is provided with the file system interface c.f. Lynx O.S. user's manual:

- **open** : to get a device descriptor to call file system function associated to the driver.
- **ioctl** : to call a special service of driver.
- **select** : to wait for the condition of the open file descriptor and other to change.
- **read** : to read bytes from the open file descriptor.
- **close** : to get rid of a previously open file descriptor.

The files <fpiplsvme.h> and <fpiplsvmeP.h> provide some definitions.

Before to invoke the services and the synchronization functions of the driver, a device file "**/dev/fpiplsXXXXX**"must be open to get an appropriate device descriptor index for the ioctl function performing the request.

with XXXXX = Fpi02 .. Fpi08 or Pls

for reading of the telegram, the device file "**/dev/fpiplsRtgm**" must be open first:

example:

```
int fid_access;
...
fid_access = open(" [illegible] ", [illegible]);
```

Set Time Out function call :

Invoked by means of the ioctl functio[illegible]

```
cc = ioctl (fid_access, FPIPLS_setupTo, &val)
```

With the argument of ioctl as follows:

```
int val;
```

Where the call routine puts parameters of call, and gets back results:

- val = at call, new value to set up the time out value in 1/100 s.

at return, to give back the previous value of time out setting.

The driver returns a specific error in errno when error returned by the call:

EFAULT Wrong parameter pointer (out of range or protected)

EACCESS This file id does not stand this function

PlsReadTgm :

The reading of the pls telegram from a module is like read from a standard file .

The system resource for that purpose is the dedicated device **/dev/fpiplsRTgm**

To select the module whose telegram has to be read, the **seek** must be use to position the reader on the target module:

exemple :

```
                    ...
          short  buff[(0x400 >> 1)];
          int cc;
          int m=0;
          int fid_access;
                    ...
          fid_access = open( "/dev/fpiplsRtgm", O_RDONLY, 0755);
                    ...
 /* select the module whose telegram is to be read */
          cc = lseek(fid_access, m, 0);
                    ...
/* read the telegram from the selected module */
          cc = read(fid_access, buff, sizeof(buff));
                    ...
```

Connection to a trigger :

The system resources for invoking these services are non sharable, the device files available are:

/dev/fpiplsPls is the unique resource dedicated for the special PLS task to be synchronised with the PLS telegram trigger.

/dev/fpiplsFpi02 to **fpiplsFpi08** are in the pool of resource available for being synchronised with external triggers.

Before to invoke the associated services, an open must be performed in order to provide a device descriptor identifier which is required for further file system calls:

example:

```
int fid_synchro;
...
fid_synchro = open("/dev/fpiplsFpi03", O_RDONLY, 0755);
```

Request for connection with a trigger:

Performed with a call to ioctl with the function code = FPIVME_connect and passing an argument defined by the following C structure (c.f. : in the header file < fpiplslib.h>) :

```
struct T_FpiplsArg {
                        struct fpiT_LineUserAdd Add;
                        short mode;
                        };
```

with the structure fpiT_LineUserAdd defined by:

```
struct fpiT_LineUserAdd {
                            char group;
                            char index;
                          }
```

Where the call routine puts parameters of call:

- group = value [0..3] giving the module number of line to connect.
- index = value [0..7] giving the line number to connect.
- mode =

  0 the default value is used

  1 cumulative mode, it means that only one event is generated for a trigger source. The event will notify the user on how many actual trigger occurred since last read

  2 'à la queue leuleu' for which a trigger an event is generated in the ring buffer of the

  The default is cumulative mode for device fpiplsPls, 'a la queue leuleu' for the other.

Getting synchronised with a trigger : (select, read)

Once the program is connected, it can wait for incoming trigger from the specified **fpiplsvme** module.

If the program must wait for incoming event from different devices, it has to use the standard Unix way: the **select** function which tells it, from which device he received an event.

If the program uses only the trigger events source, it can wait and receive the event in one call to the **read** from the device:

- one or more events arrived, the read returns the event in the buffer
- nothing arrived, the read blocked on wait until one event comes or time out occurs.

The buffer must be big enough to receive a complete 4 bytes event, the buffer will receive as many complete events, present in the devices, as there is enough room in the buffer.

The design of an event is given by the following C structure:

```
struct fpiT_Event {
                        short    count;
                        unsigned char group;
                        unsigned char index;
} ;
```

Where :

- count = according to requested mode at connection:
  - cumulative mode : give the number of trigger received since previous read.
  - 'à la queue leuleu' mode : gives the serial number of trigger since connection has been established.
- group = module number of the associated trigger .
- line = line index of trigger in the group..

```
/*        %( example of C syntax usage:  */
          ...
#include <gpsynchrolib.h.h>
          ...
          int fid_access;
          int fid_synchro;
          struct  T_fpiplsA
          struct fpiT_Event
          int cc;
          ...
          /* open the channel to perform the Camac access and
              the channel to get synchronisation with the LAM
              from the module accessed  */
          fid_access  = open("/dev/fpiplsFpi03",O_RDONLY,0755);
```

```
...
fid_synchro = open("/dev/fpiplsRTgm",O_RDONLY,0755);
...
/* ask to be connected to the incoming trigger from the
   fpiplsvme module */
cc = ioctl(fid_synchro, FPIVME_connect, &arg_connect);
...
/* ask to be synchronised:
   the read will wait for trigger to come and
   to return the first event in the buffer */
cc = read(fid_synchro, &Event, sizeof(Event));
...
/* Then process the received event */
...
/*      )% */
```

PS Division

ABIE	Habtamu	PS/PO
ADORNI	Valerio	PS/CO
AMENDOLA	Giandomenico	PS/HI
ARRUAT	Michel	PS/OP
ASSOR	Jean-Luc	PS/CO
BENINCASA	Gianpaolo	PS/CO
BERLIN	Fridtjof	PS/CO
BOBBIO	Piero	PS/CO
BOUCHE	Jean-Marc	PS/CO
BOUCHERON	Jean	PS/RF
CENDRE	Jean-Claude	PS/OP
CHERIX	Etienne	PS/OP
CHOHAN	Vinod	PS/AR
CROTEAU	Pascal	PS/CO
CUISINIER	Gerard	PS/CO
CUPERUS	Jan	PS/CO
CYVOCT	Georges	PS/OP
DAEMS	Gilbert	PS/CO
DE METZ-NOBLAT	Nicolas	PS/CO
DEHAVAY	Claude	PS/CO
DELOOSE	Ivan	PS/OP
DI MAIO	Franck	PS/CO
DUPUY	Bruno	PS/OP
ELYN	Jean-Michel	PS/OP
EVANS	John	PS/RF
FALK	Elisabeth	PS/BD
FERNIER	Pascal	PS/OP
FRAMMERY	Bertrand	PS/OP
FUCHS	Joachim	PS/OP
GAGNAIRE	Alain	PS/CO
GELATO	Gianni	PS/BD
GIUDICI	Francois	PS/CO
GUEUGNON	Daniel	PS/OP
HEINZE	Wolfgang	PS/CO
HOH	Roger	PS/OP
LELAIZANT	Monique	PS/CO
LEWIS	Julian	PS/CO
LUSTIG	Hans-Dieter	PS/CO
MALMEDAL	Terje	PS/OP
MANGEOT	Bernard	PS/OP
MERARD	Lucette	PS/CO
METRAL	Gabriel	PS/OP
MULDER	Hendrik	PS/OP
MUNKOE	Leif	PS/CO
NONGLATON	J-Michel	PS/OP
OVALLE	Ernesto	PS/OP
PACE	Alberto	PS/OP
PASINELLI	Sergio	PS/OP
PERRIOLLAT	Fabien	PS/CO
PETTERSSON	Thomas	PS/AR
PRIESTNALL	Kevin	PS/OP
RAICH	Ulrich	PS/CO
RENAUD	Yves	PS/OP
ROSENSTEDT	Anker	PS/CO
RUETTE	Michel	PS/OP
SERRE	Christian	PS/CO
SHEPHERD	Peter	PS/CO
SICARD	Claude-Henri	PS/CO
TANKE	Eugene	PS/HI
TEDESCO	Jean Simon	PS/BD
TERRIER	Jean-Pierre	PS/RF
VANDORPE	Bernard	PS/OP
VICENTE	Victor	PS/OP
WILDNER-MALANDAIN	Elena	PS/OP
ZELEPOUKINE	Seriozha	PS/CO

Pour Information:

HÜBNER Kurt	PS
SIMON daniel	PS

\+ PS GROUP LEADERS

SL Division:

ANDRE	Airy	SL/CO
BLAND	Alastair	SL/CO
BURNS	Alan	SL/BI
CHARRUE	Pierre	SL/CO
CHRISTIANSEN	Hans Peter	SL/CO
GAREYTE	Claire	SL/CO
GHINET	François	SL/CO
KISSLER	Karl-Heinz	SL/CO
KOSTRO	Krzysztof	SL/CO
RAUSCH	Raymond	SL/CO

Division ECP:

PETERSEN Jorgen	ECP/DS