ISSN 0101-9201

CNPq

CBPF-CENTRO BRASILEIRO DE PESQUISAS FÍSICAS

Notas Técnicas

CBPF-NT-003/91

SISTEMA DE DIAGNÓSTICOS DO ACP-II

por

A. BRANCO, M. MIRANDA e B. SCHULZE

RIO DE JANEIRO
1991

RESUMO: Esta nota técnica apresenta um ambiente de diagnóstico desenvolvido para o sistema de processamento paralelo ACP-II. Este ambiente está definido para testar um nó de CPU sem sistema operacional através de uma segunda máquina com sistema operacional UNIX. Para isto, apresentamos as ferramentas de *software* para o ambiente *standalone* (nó sem Sistema Operacional) bem como a arquitetura proposta para o sistema de diagnóstico.

Neste trabalho são ressaltadas as seguintes características: *a*) a flexibilidade na geração e execução dos programas de diagnóstico, introduzida pela utilização de uma segunda máquina com sistema operacional (UNIX); *b*) a robustez do sistema com a unidirecionalidade do controle que garante a integridade da máquina com UNIX em relação a um mal funcionamento do nó em teste e *c*) uma estrutura com capacidade de interação entre os processos existentes, viabilizando a utilização de máquinas distintas, explorando os melhores recursos de cada uma para o melhor desempenho do sistema.

PALAVRAS-CHAVE: Sistema de diagnóstico, ACP-II, CPU RISC.

SUMÁRIO

I) Introdução

O trabalho de Física Experimental de Altas Energias requer uma capacidade computacional muito grande para tratar a grande quantidade de dados colhidos dos detectores durante os experimentos. Estes dados são tratados *off-line* em programas de reconstrução e análise e *on-line* em sistemas de aquisição de dados. Para apoiar o trabalho de aquisição, análise e simulação de eventos da Física Experimental de Altas Energias, partiu-se para o desenvolvimento de máquinas de processamento paralelo e de alto desempenho.

No histórico do LAFEX temos o trabalho realizado com o ACP-I, máquina de CPU's paralelas desenvolvida para funcionar utilizando um hospedeiro (*host*-μVax) que distribui as tarefas a cada módulo de CPU e serve de interface com o usuário. Com a evolução do sistema para o ACP-II, a máquina passou a possuir um sistema operacional independente (UNIX)[11], residente em cada módulo de CPU. Para a implantação do paralelismo foi desenvolvido um conjunto de rotinas denominadas de CPS (*Cooperative Processes Software*)[10] que possibilita a interação de processos de CPU's diferentes.

O trabalho descrito a seguir é o resultado atual de um estudo e desenvolvimento para um sistema de diagnóstico aplicado ao sistema de processamento paralelo ACP-II. O estudo está baseado em sistemas implementados em outras máquinas e em ferramentas de *software* existentes. A partir disto, foi possível definir a arquitetura utilizada neste sistema de diagnóstico, onde podemos ressaltar as seguintes características: *a*) a flexibilidade na geração e execução dos programas de diagnóstico, introduzida pela utilização de uma segunda máquina com sistema operacional (UNIX); *b*) a robustez do sistema com a unidirecionalidade do controle que garante a integridade da máquina com UNIX em relação a um mal funcionamento do nó em teste e *c*) uma estrutura com capacidade de interação entre os processos existentes, viabilizando a utilização de máquinas distintas, explorando os melhores recursos de cada uma para o melhor desempenho do sistema.

No trabalho, são abordados as ferramentas utilizadas e a implementação do sistema, contendo as informações necessárias para a implantação das rotinas de diagnóstico na máquina ACP-II.

Na Seção II, é descrito o ambiente de trabalho, com enfoque nas ferramentas necessárias para a montagem de uma plataforma de desenvolvimento.

Na Seção III, é definida a implementação do sistema de diagnóstico, com ênfase nos módulos individuais.

Na Seção IV é apresentada uma avaliação sobre o estado atual e a continuidade do trabalho.

II) O Ambiente de Trabalho

II.1) O Ambiente de Desenvolvimento: [28]

Chamamos de ambiente de desenvolvimento um ambiente mínimo de trabalho necessário para a geração e execução de um conjunto de programas de diagnóstico. O ambiente necessário é composto de:

1) Máquina de desenvolvimento - uma máquina que nos possibilite a edição, compilação e transferência do código executável para o sistema remoto em teste.
2) Máquina de teste - a máquina remota em teste onde são executados os programas desenvolvidos.

A máquina de desenvolvimento deve ter um ambiente mínimo de trabalho, que em nosso caso é fornecido pelo sistema operacional UNIX. Além do sistema operacional, é necessária uma ligação com a máquina remota, que em nosso caso é feita através de um barramento denominado Branch-Bus[1]. O sistema global pode ser visto na figura 1.

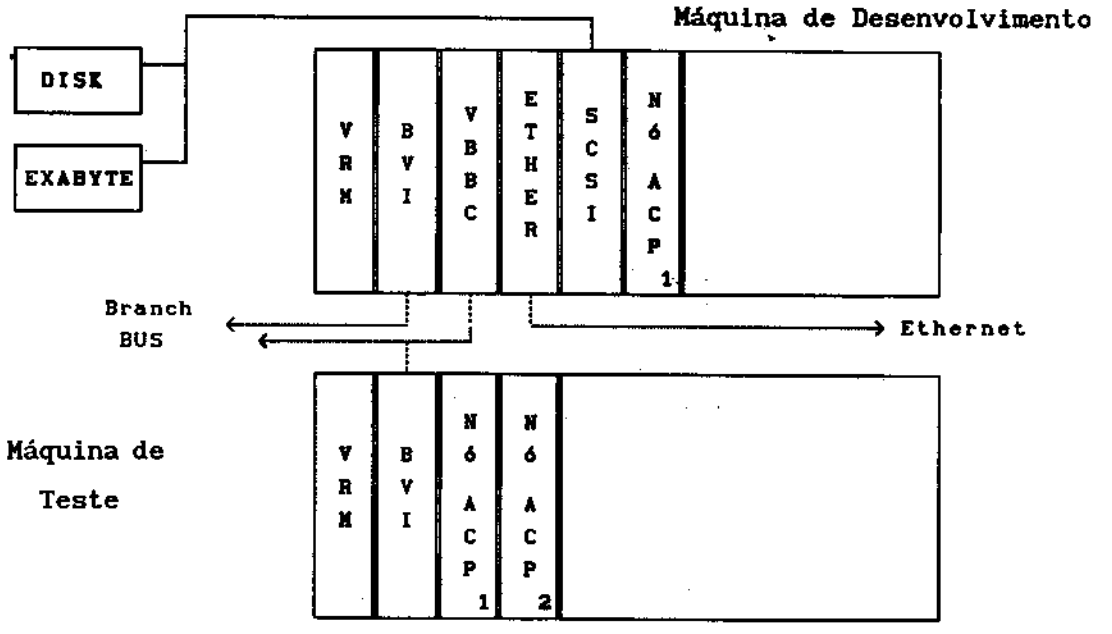


fig.: 1 - *O Ambiente de Desenvolvimento*

Os módulos utilizados são:

VRM - Módulo de arbitragem do barramento *VME*[2]

VBBC - Módulo controlador de acesso *VME/Branch-Bus*[4]

BVI - Módulo de interface entre *Branch-Bus* e *VME*[3]

SCSI - Módulo controlador do barramento SCSI (*Small Computer System Interface*)[17]

ETHER - Módulo controlador da interface Ethernet[14]

Nó ACP - Módulo de processamento. (CPU)[7]

Notamos que na configuração mostrada na figura 1, o controle do barramento VME só é possível pela máquina de desenvolvimento, portanto a máquina de teste fica sem possibilidade de interferir na máquina de desenvolvimento.

Na figura 2 mostramos o ambiente a nível de organização de *software*. Observamos que, para a geração de um programa *Standalone* precisamos de uma biblioteca específica (SAIO) e de uma compilação (SACC) que redirecione as chamadas para esta biblioteca. Para a comunicação com o sistema remoto, utilizamos um dispositivo especial denominado de *Branch-Bus Device Driver* (VBBC)[8], através do qual podemos comunicar o sistema de desenvolvimento com o sistema remoto. Já no sistema remoto ou de teste, precisamos basicamente da interface BVI e das rotinas mínimas da PROM MONITOR.

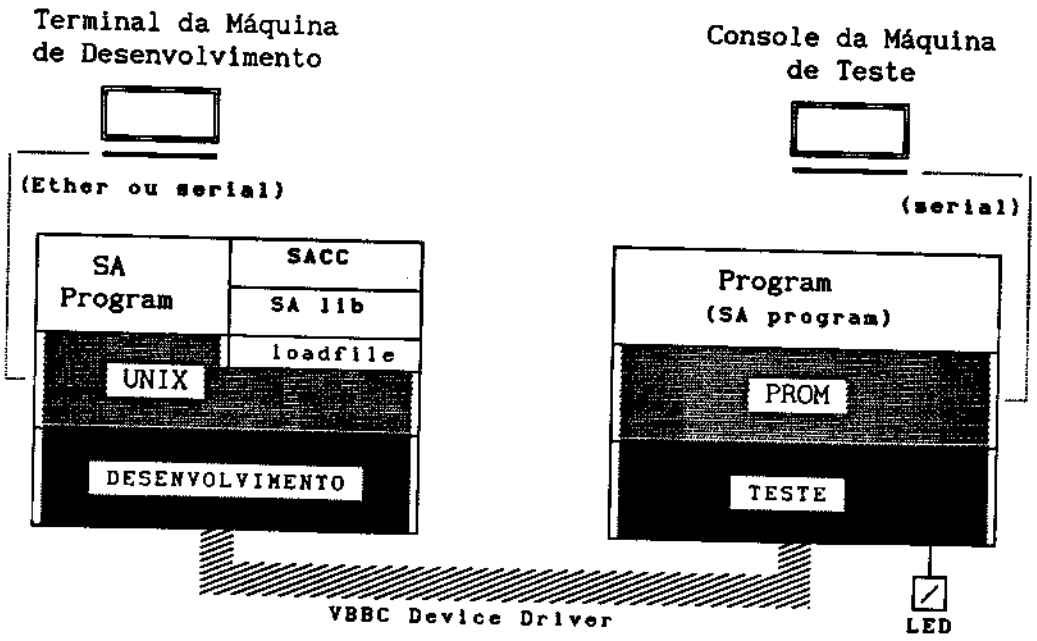


fig.: 2 - *O ambiente de trabalho*

II.2) A Biblioteca I/O Standalone (SAIO)[28]

Esta biblioteca resume um conjunto de funções de entrada/saída para os programas a serem executados em uma máquina sem sistema operacional. As chamadas desta biblioteca são decodificadas pela PROM MONITOR da CPU, onde estão definidas as funções básicas da máquina sem sistema operacional.

Como os programas a serem desenvolvidos são definidos para um ambiente restrito como o Standalone, também será restrita a biblioteca de funções I/O. A seguir temos as funções disponíveis e a descrição das mais utilizadas:

atob, badaddr, bstring, clear_cache, clearseg, close, execv, exit, flush_cache, getc, getenv, gets, inet, ioctl, isatty, lseek, nofault, nuxi, open, parser, printf, protocol, putc, read, sa_spl, scandevs, set_timer, setjmp, show_fault, showchar, stringchar, wbflush e write.

Comando	Descrição
atob	converte para binário uma string ASCII representada na base decimal, octal, hexadecimal ou binária.
gets	lê uma string do console.
printf	direciona saída para o console.
exit	finaliza o processo após fechar todos os arquivos abertos.

II.3) O Sistema de Compilação[28]

Para podermos trabalhar no ambiente *standalone*, foram implementados alguns comandos para compilação. Estes comandos estão preparados para compilar e concatenar (*link*) os programas com as bibliotecas específicas (no anexo A podemos ver a estrutura do diretório SA). A seguir temos a descrição dos comandos e logo abaixo a implementação em *Shell* do compilador SACC:

sacc - Utilizado para compilação de programas em C
saas - Utilizado para compilação de programas em linguagem assembly
sald - Utilizado para a carga e concatenação (*link*) com a biblioteca *saio*

```
#! /bin/sh
# sacc -- standalone compiler interface
#

        LIB=/usr/acpunix/src/SA/saio/ACP_SAIO_O
set -x
cc -I/usr/acpunix/src/SA/include/saio -I/usr/acpunix/src/SA/include/prom \
   -I/usr/acpunix/src/SA/include/sys -I/usr/include -N -T 80020000 \
   -L -L$LIB  -lsa "$@" -tr -h$LIB -B
```

II.4) O *Loadfile*

O programa *loadfile* tem como função permitir a transferência de um arquivo executável armazenado em disco para a memória de um nó remoto do ACP-2. Este tipo de transferência é normalmente feita para um nó remoto em que não foi carregado o UNIX e por isso, o programa a ser transferido deverá estar devidamente preparado para esta situação.

O *loadfile* está baseado nas rotinas do *VBBC - Device Driver*, que possibilita este tipo de transferência, inclusive o *driver* tem a capacidade de reconhecer se o nó remoto está ou não no mesmo bastidor(*crate*).

A sintaxe de utilização é a seguinte:

loadfile *file crate node* [-r] [-a *load_address*]

Aonde:

file é o arquivo a ser transferido.

crate é o número do bastidor destino.

node é o número do nó destino.

-r é opcional e define a execução do programa.

-a *load_address* é opcional e define um novo endereço de carga do programa e, se existir a opção -r, será também o endereço de execução.

obs: os valores numéricos serão assumidos em hexadecimal sem o prefixo 0x.

Ex: loadfile teste/arq1 01 90 -a 80010000 -r

II.5) A Memória do ACP-2

Os programas executados no sistema *standalone* devem respeitar a estrutura da memória definida no nó ACP-II. Esta estrutura está definida para um sistema mais completo, do qual interessam: a área reservada e a área para programas *standalone*. A seguir temos a descrição desta estrutura.[22]

fisico	kseg1	uso
0x1fc20000 to 0x1fc00000	0xbfc20000 to 0xbfc00000	prom text and read-only data (espaço da prom na placa de cpu)
(Top of RAM - 8k) ↓		sash and standalone program stack (8k reservados- buffer de mensagem do kernel)
↑ 0x00100000	↑ 0xa0100000	sash program text, data, and bss
↑ 0x00014000	↑ 0xa0014000	standalone program text, data, and bss (kernel é carregado aqui)
0x00013fff ↓	0xa0013fff ↓	dbgmon stack
↑ 0x00008000	↑ 0xa0008000	dbgmon text, data, and bss
0x00007fff ↓	0xa0007fff ↓	prom monitor stack
↑ 0x000027a0	↑ 0xa00027a0	prom monitor bss
0x0000279f to 0x00000500	0xa000279f to 0xa0000500	reserved area
0x000004ff to 0x00000400	0xa00004ff to 0xa0000400	restart block
0x000003ff to 0x00000080	0xa00003ff to 0xa0000080	general exception code
0x0000007f to 0x00000000	0xa000007f to 0xa0000000	utlbmiss exception code

II.5.1) A área reservada (acpboot area)[12]

A área reservada é um local de memória utilizada para comunicação entre as rotinas do sistema. Nela encontramos informações como o valor do relógio de tempo real e variáveis para inicialização do sistema operacional.

Listamos logo abaixo uma parte da definição da área reservada que pode ser encontrada em forma de *#include file* na biblioteca SA. Ressaltamos esta parte pois nela emcontramos três posições de memória muito importantes para o nosso sistema de diagnóstico:

```
ACPBOOT_UNIX_LOC        (0x00000690) - endereço inicial para execução
ACPBOOT_ACKNOWLEDGE_LOC (0x00000694) - indica execução iniciada
ACPBOOT_ENABLE_LOC      (0x00000698) - habilita execução
    obs.: O ambiente utiliza definições do UNIX
```

```
/* acpreset.h -- the reset locations in acpboot area */

#ifndef _SYS_ACPRESET_
#define _SYS_ACPRESET_  1

#define ACPBOOT_LOCATION                (PHYS_TO_K1(0x00000584))
#define ACPBOOT_CRATE_NUMBER            (PHYS_TO_K1(0x00000588))
#define ACPBOOT_MY_HOSTNAME_LOC         (PHYS_TO_K1(0x0000060C))
#define ACPBOOT_SERVER_HOSTNAME_LOC     (PHYS_TO_K1(0x0000064C))
#define ACPBOOT_HOW_TO_BOOT_LOC         (PHYS_TO_K1(0x0000068C))
#define ACPBOOT_UNIX_LOC                (PHYS_TO_K1(0x00000690))
#define ACPBOOT_ACKNOWLEDGE_LOC         (PHYS_TO_K1(0x00000694))
#define ACPBOOT_ENABLE_LOC              (PHYS_TO_K1(0x00000698))
#define ACPBOOT_SWAPSERVER_LOC          (PHYS_TO_K1(0x0000069C))
#define ACPBOOT_MEMTOP_LOC              (PHYS_TO_K1(0x000006A4))
#define ACPBOOT_STATISTICS_LOC          (PHYS_TO_K1(0x000006A8))
#define ACPBOOT_PROM_VERSION_LOC        (PHYS_TO_K1(0x000006B4))
#define ACPBOOT_END                     (PHYS_TO_K1(0x00000764))

#define RESET_ACKNOWLEDGE_VALUE         0xa1a1a1a1
#define RESET_ENABLE_VALUE              0xc0c0c0c0
#define GO_ENABLE_VALUE                 0xd0d0d0d0

#define BOOT_AS_SERVER                  0x53535353
#define BOOT_AS_CLIENT                  0x43434343
#define BOOT_ACCORDING_TO_SWITCH        0x78787878

#define ACPRESERVED_LOCATION            (PHYS_TO_K1(0x00000764))
#define ACPRESERVED_VCP_AREA_ADDR_LOC   (PHYS_TO_K1(0x00000764))
#define ACPRESERVED_VCP_ENABLED_LOC     (PHYS_TO_K1(0x00000768))
#define ACPRESERVED_END                 (PHYS_TO_K1(0x00000784))

#endif  _SYS_ACPRESET_
```

II.6) O Dispositivo VBBC[8]

É através do dispositivo VBBC que temos acesso ao nó remoto a partir do nó local. Na prática este acesso é feito por um comando disponível na definição do *driver* do dispositivo.

A definição da estrutura de comando (*bb_operation*) é a seguinte:

byte_count	tamanho da transferência
vme_address	endereço físico do VME (nó remoto)
virtual_address	endereço do *buffer virtual* (nó local)
write_enable	0= *vme_address -> virtual_address*
flags	*flags* de controle e opções
crate_id	identificação do bastidor (*crate*)
*status	ponteiro para área opcional de *status*

A chamada ao dispositivo para transferência é da seguinte forma:

```
vbbc_fd = open ("/dev/vbbc", O_RDWR );
stat = ioctl (vbbc_fd, BB_XFER, &bb_op);
close ( vbbc_fd );
```

onde:

ioctl	função de controle do VBBC device driver que retorna o status da operação
vbbc_fd	associação ao dispositivo VBBC
BB_XFER	definição para uma transferência comum
&bb_op	endereço da estrutura bb_operation

II.7) As Funções EXEFILE

Para enviarmos um código executável a um nó remoto, precisamos ler este arquivo e retirar informações como: o endereço de partida do texto executável, variáveis inicializadas e o próprio código executável. Para realizar estas funções temos disponível as funções do

grupo EXEFILE.

A função open_exefile tem como objetivo abrir um arquivo executável do sistema UMIPS e retornar o endereço de entrada do código.

A função read_exefile lê o arquivo e separa para um *buffer* na memória o código executável e as variáveis inicializadas.

A função close_exefile fecha o arquivo. A seguir temos um exemplo de utilização:

```
file = open_exefile ( arq, 13, &end );
read_exefile ( file, buffer, BUFLEN, &ladd );
close_exefile ( file );
```

III) Implementação

III.1) O Sistema de Diagnóstico

O diagrama da implementação do sistema de diagnóstico pode ser visto abaixo onde notamos a estrutura atual baseado nos seguintes programas:

Diags, *loadfile*, *write_block* e *read_block* - sendo executados no nó local.

Remoto - sendo executado no nó remoto.

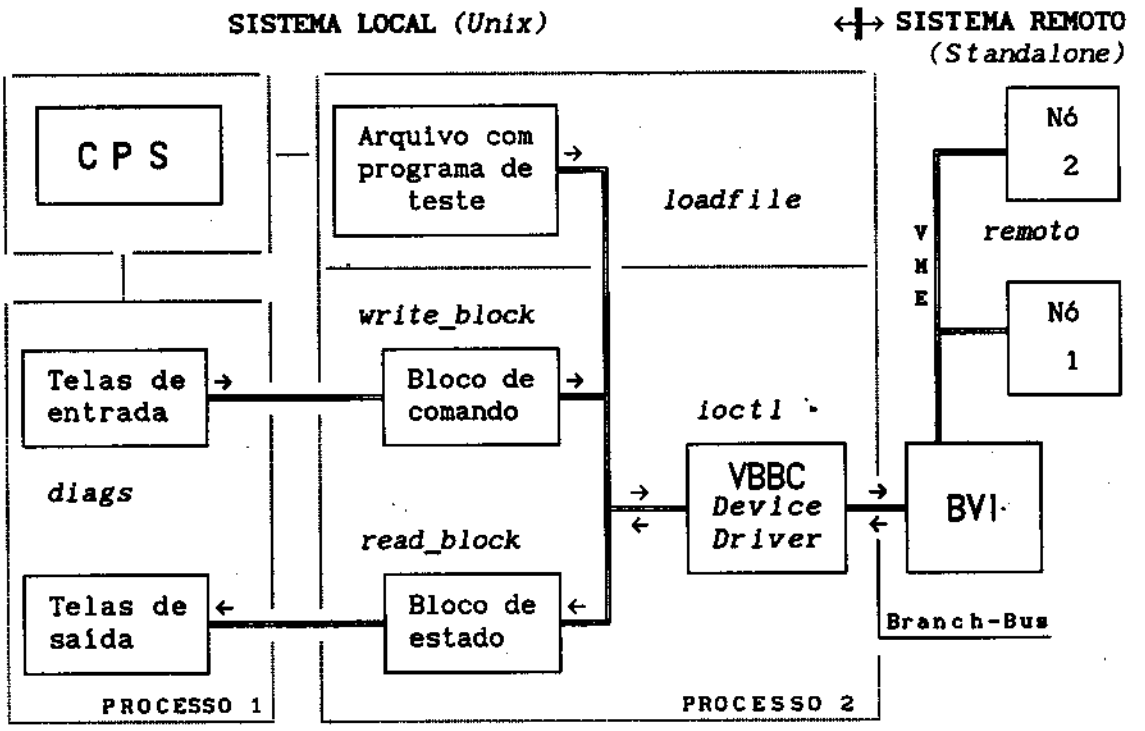


fig.: 3 - *O Sistema DIAGS*

III.2) O programa LOADFILE[1]

Tem a seguinte estrutura:

- verifica sintaxe da linha de comando
- abre o arquivo e devolve o endereço de partida
- abre o dispositivo *vbbc*
- lê o arquivo para um *buffer*
- escreve do *buffer* para o nó destino, via VBBC driver
- fecha arquivo
- envia estrutura da área de *boot*
- envia *reset enable*
- verifica *acknowledge*
- fecha o dispositivo VBBC

III.3) O protocolo de comunicação

Definimos uma área de memória do nó remoto para comunicação com o nó local. Esta área esta organizada em forma de estrutura e posicionada no endereço 0x00010000.

A seguir temos o formato da estrutura.

```
struct command_block {
   unsigned long command;
   unsigned long state;
   unsigned long debug;
   unsigned long errors;
   unsigned long parms[20];
};
```

Com isto podemos comandar e verificar o status de um programa *standalone* no nó remoto, através de um programa executando em cima do UNIX no nó local. Como o endereço da estrutura no nó remoto é fixa, podemos utilizar as rotinas do *VBBC device driver* para as transferências de dados.

[1] Baseado no bd.c (boot_diskless)

III.4) As funções de bloco

São as funções responsáveis para a escrita e leitura do bloco de comando do programa diags. A sintaxe da chamada e a estrutura estão descritas a seguir:

```
write_block ( pcb, crate_number, node_number);
read_block  ( pcb, crate_number, node_number);
```

- abre o dispositivo VBBC;
- escreve ou lê bloco de comando do endereço virtual 0x10000;
- fecha o dispositivo VBBC.

III.5) O Programa DIAGS

Tem como função interagir com o usuário do sistema, comandar o programa remoto e verificar o estado do mesmo. A seguir temos a descrição da estrutura:

- verifica a rotina a ser executada, inicializa as variáveis necessárias;
- lê o estado do nó remoto;
- envia parâmetros;
- envia comando;
- lê e imprime o estado do programa remoto.

III.6) O programa REMOTO

É a parte do sistema de diagnóstico que contém o código das rotinas de teste. Estas rotinas podem ser controladas pelo bloco de comando definido no item III.3. A seguir, temos a estrutura do programa:

- inicializa área de comunicação;
- fica em *loop* lendo comando até comando ser diferente de zero;
- desvia para rotina de comando.

IV) Conclusão

Atualmente, o Sistema Diags oferece uma plataforma para a implementação das rotinas de diagnóstico, com as seguintes rotinas implementadas: Teste de memória, escrita no console, escrita no display. Uma etapa seguinte é o estudo mais aprofundado do hardware do módulo ACP-II com fins de aperfeiçoamento e desenvolvimento de algumas rotinas mais especializadas para o diagnóstico.

No caso mais imediato estamos concentrando esforços no circuito de interrupção, pois a carência de rotinas eficazes torna a situação urgente.

Outra etapa é a implementação de um sistema mais confortável de interação com o usuário, sendo executado em uma máquina local e definido como processo 1 na figura 3. Com isto, teremos a possibilidade de produzir telas que facilitem o uso interativo em máquinas remotas com recursos gráficos mais potentes que os encontrados atualmente no sistema ACP-II. Uma alternativa deverá ser possível através da interação da máquina remota com o sistema ACP-II, através do CPS.

Observamos que a estrutura criada para o desenvolvimento e execução de programas de diagnósticos podem ser utilizadas para desenvolver outras aplicações *standalone*, por exemplo: sistema de aquisição de dados *on-line*, *bechmarks* do *hardware* do ACP-II e programas de depuração.

V) Agradecimentos

Gostaríamos de agradecer a nossos colegas do LAFEX e em particular ao nosso principal incentivador, Prof. Alberto Santoro.

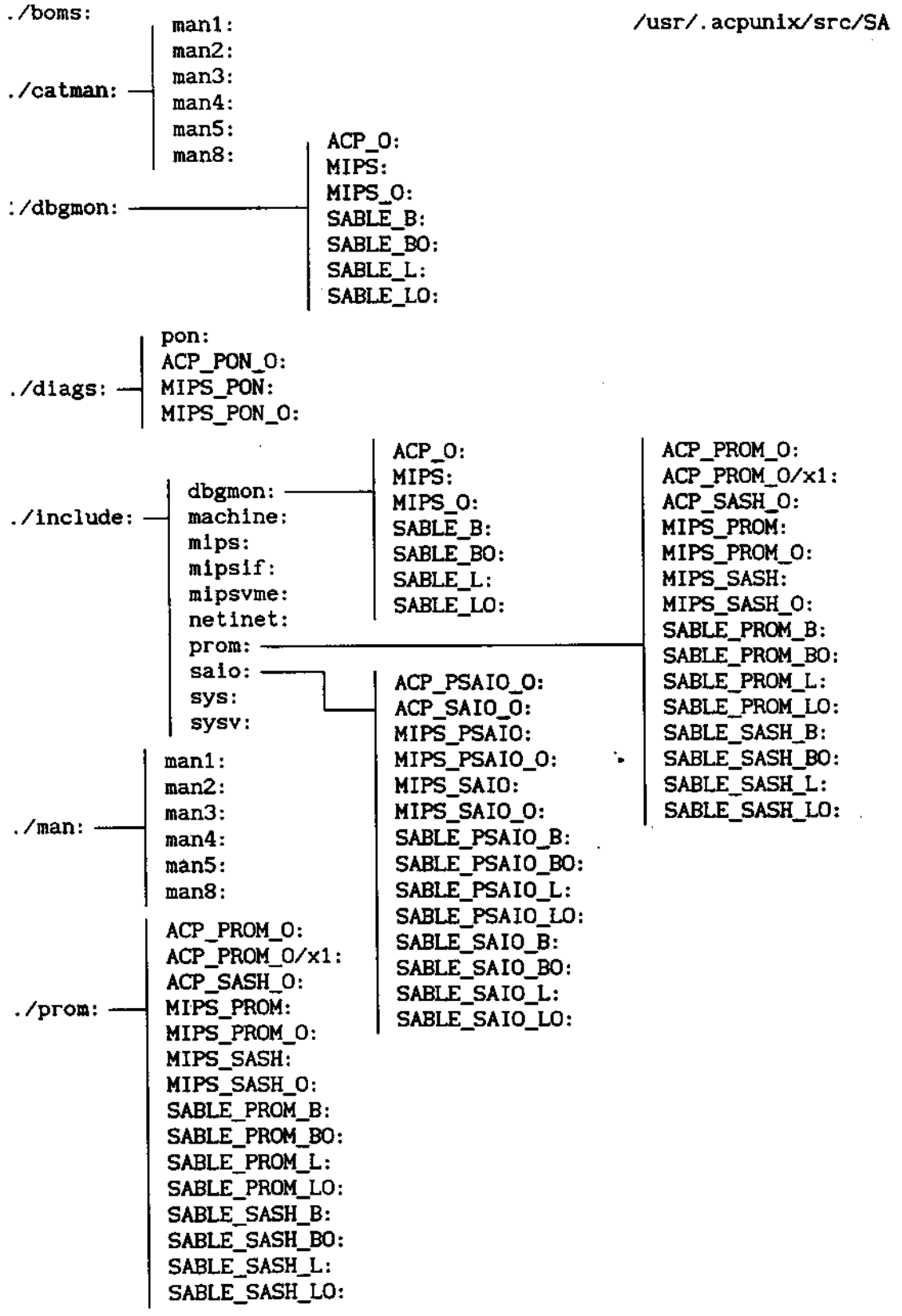
Um agradecimento especial à Raquel Schulze pelas sugestões e orientações às técnicas de redação do texto, e à Carla Barros por comentários técnicos de *hardware*.

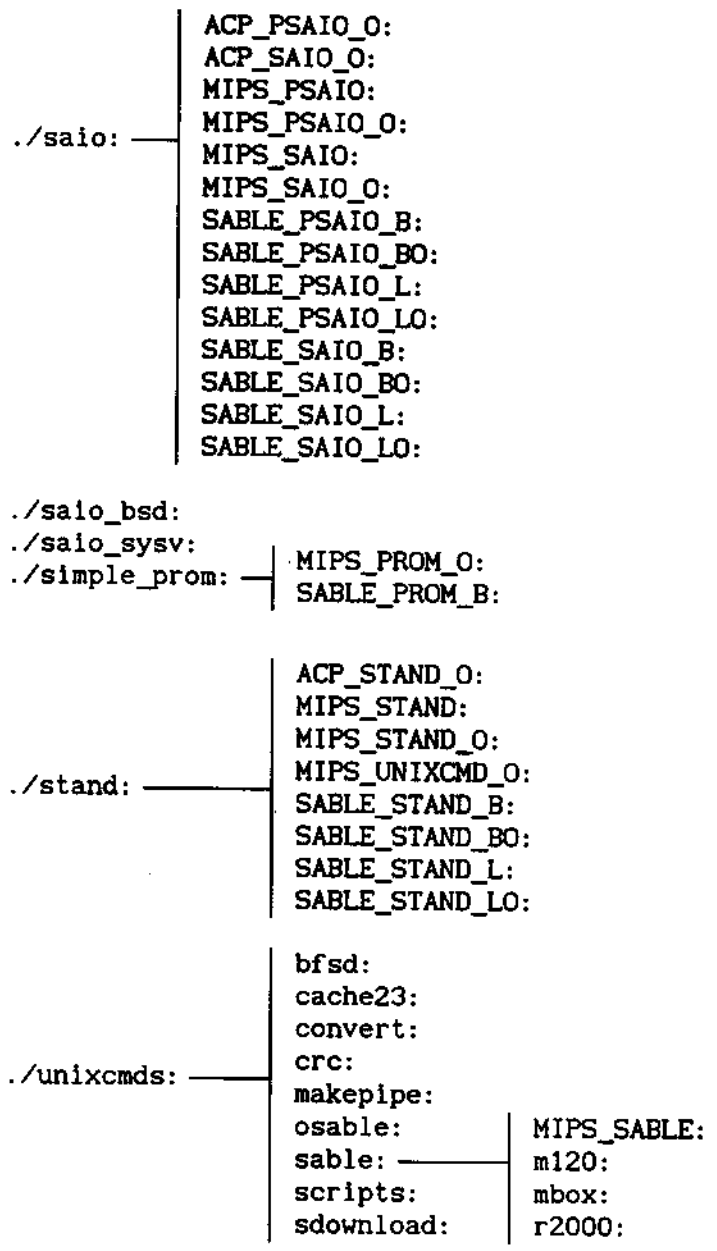
Este trabalho conta com o apoio da RHAE/CNPq.

VI) Anexo

ESTRUTURA DO DIRETÓRIO 'SA'

/usr/.acpunix/src/SA




./saio:
ACP_PSAIO_O:
ACP_SAIO_O:
MIPS_PSAIO:
MIPS_PSAIO_O:
MIPS_SAIO:
MIPS_SAIO_O:
SABLE_PSAIO_B:
SABLE_PSAIO_BO:
SABLE_PSAIO_L:
SABLE_PSAIO_LO:
SABLE_SAIO_B:
SABLE_SAIO_BO:
SABLE_SAIO_L:
SABLE_SAIO_LO:
./saio_bsd:
./saio_sysv:
./simple_prom:
MIPS_PROM_O:
SABLE_PROM_B:
./stand:
ACP_STAND_O:
MIPS_STAND:
MIPS_STAND_O:
MIPS_UNIXCMD_O:
SABLE_STAND_B:
SABLE_STAND_BO:
SABLE_STAND_L:
SABLE_STAND_LO:
./unixcmds:
bfsd:
cache23:
convert:
crc:
makepipe:
osable:
sable:
scripts:
sdownload:
MIPS_SABLE:
m120:
mbox:
r2000:

VII) Bibliografia

VII.1)Publicações e Manuais

[1]*Branch Bus Specification,* Fermilab - Advanced Computer Program, MArch/1986.

[2]*VMEbus Resource Module - VRM*, Fermilab - Advanced Computer Program, May/1986.

[3]*Branch Bus VMEbus Interface - BVI*, Fermilab - Advanced Computer Program, November/1987.

[4]*VME/Branch Bus Controller - VBBC*, Fermilab - Advanced Computer Program, September/1987 Revised April/1988.

[5]*VME Branch Bus Terminator Reference Manual - VBBT*, Omnibyte OB/ACP-VBBT, MAY/1988 No.2.

[6]*ACP/R3000 Processor Board Specification*, Fermilab - Advanced Computer Program, Rev.D, February/1989.

[7]*ACP/R3000 Module*, Fermilab - Advanced Computer Program - Universities Research Association, Inc., Version 1.0, Feburary/1989.

[8]*VBBC Device Driver, ACP 2nd Gen Software Note*, Fermilab, Univertsities Research Association, 5/1989.

[9]Miranda,M.S., *Tese de Mestrado "Um Conjunto de Ferramentas para Implementação de Processos Cooperativos"*, COPPE/UFRJ Eng.Sistemas, 4/1991.

[10]*ACP Cooperative Processes User's Manual*, Fermilab - Advanced Computer Program, Universities Research Association Inc., 3/1989.

[11]*ACP Cooperative Processes System Manual*, Fermilab - Advanced Computer Program - Universities Research Association, Inc., Version 1.1, 8/1990.

[12]Miranda,M.S., *Reserved Memory in ACP/R3000*, Fermilab - Advanced Computer Program - Universities Research Association, Inc.,1989

VII.2)Referências Bibliográficas

[13]The VMEbus Specification, by Motorola, PRINTEX Publishing Inc., Rev C.1 Oct.1985

[14]*Ethernet Node Processor ENP-10 Reference Guide* - Communication Machinery Corporation (CMC), Document No.6211000-05B, Nov.1988.

[15]*High Performance CMOS Data Book,* Performance Semiconductor Corporation, 1989.

[16]*High Performance CMOS Data Book Supplement 1989,* Integrated Device Technology.

[17]*Rimfire 3510 SCCI Host Bus Adaptor and Floppy Disk Controller Reference Manual,* Ciprico Inc., Publication Number 21017700, July/89.

[18]*EXB-8200 8MM Cartridge Tape Subsytem Compatibility Manual,* Exabyte Corporation, publication number 510002-004, 1988.

[19]*Assembly Language Programmer's Guide 02-00036,* MIPS Computer Systems, Inc. - 1988.

[20]*Language Programmer's Guide 02-00035,* MIPS Computer Systems, Inc. - 1988.

[21]*M/120 RISComputer System Technical Reference 02-00159,* MIPS Computer Systems, Inc. - 1988.

[22]*System Programmer's Guide 02-00037,* MIPS Computer Systems, Inc. - 1988.

[23]*The UNIX Programmer's Reference Manual 02-00010,* MIPS Computer Systems, Inc. - 1988.

[24]*The UNIX Programmer's Supplementary Documents 02-00028,* MIPS Computer Systems, Inc. - 1988.

[25]*The UNIX User's Manager Manual 02-00024,* MIPS Computer Systems, Inc. - 1988.

[26]*The UNIX User's Reference Manual 02-00027,* MIPS Computer Systems, Inc. - 1988.

[27]*The UNIX User's Supplementary Documents 02-00023,* MIPS Computer Systems, Inc. - 1988.

[28]Kane,G.;*MIPS R3000 RISC Architecture* -MIPS Computer Systems, Inc. - 1988.

[29]*System Programmer's Package Reference 02-00287,* MIPS Computer Systems, Inc. - 1988

ABSTRACT: This technical note describes a diagnostics system setup for the ACP-II parallel processing machine. This setup is defined for testing nodes not running under an operating system but using a second machine now running a UNIX-like operating system. For that we present the basic software tools for developing standalone programs as well as the architecture of the diagnostics system setup.

We want to distinguish some characteristics like: a) flexibility in generating and running the diagnostics code, introduced by the second node running UNIX; b) the oneway control of the UNIX machine over the test node which keeps the first machine asave of any malfunction of the node under test; c) a proposal of a setup where there is an interaction between the running processes, allowing the use of different machines with appropriate resources for each job.