Contents

CHIPMUNK 1

All programs described in this report are implemented as CHIPMUNK programs and extensions. This appendix will give a short description of the language and the framework of CHIPMUNK , version 1.4 . With this information the user will be able to read as well as write programs in CHIPMUNK .

After a short introduction to calling CHIPMUNK , macro's and variables, all the built-in functions and their syntax will be described. Then a practical guide to programming and adding extensions will be given.

1.1 Introduction

CHIPMUNK is a language for writing short programs to perform application specific hardware manipulation tasks. It consists of a command-interpreter and a facility for loading and saving programs. For historical reasons the programs will be called macro's.

CHIPMUNK can be called in two ways:

- in batch mode. The -b option and a command (for example the macro that is to be executed) must be supplied. After the macro is executed CHIPMUNK exits with the return code of the macro;

```
% chipmunk -b echo Hello user
Hello user
% _
```

- in interactive mode. An initial macro could be supplied, if not, `init.mac` will be called. After the initial macro is executed CHIPMUNK prints its prompt (> _) and waits for user input.

  ```
  % chipmunk
  Autoloading init.mac.
  This is Chipmunk version 1.3.2 .
  This is THE init file.... running on linux
  > _
  ```

At the prompt the user can enter one of the commands described below or call a macro. If a macro is called for the first time it is automatically loaded (autoload) from a file with extention `.mac`.

1.2 Variables

CHIPMUNK provides variables which can be used to hold a numeric value or a string. Their value can be used in future commands.

A variable can be set in three ways:

1. by specifying its value using `set`;
2. by copying it from environment (`copyenv`);
3. implicitly by commands that need to return more than one value.

A reference to a variable is made by prefixing its name with `$` or `$$` (see section 1.4.1 for the difference). For example:

```
> set ERRORS 5
> copyenv HOME
> echo $ERRORS $HOME
5 /home/stout
> _
```

Apart from the variables the user can define, there are some predefined, internal variables:

- **STATUS**: contains the return value of the last executed command. Cannot be set by user;
- **BASE**: memory access base address. BASE offsets all memory accesses implicitly;
- **IOBASE**: I/O access base address. IOBASE offsets all I/O accesses implicitly;

- **VERBOSE**: controls how eloquent commands will be. Useful range from -1 (fully dumb) to 2 (full). Set to 1 by default or to another value by the init file;

- **TRACE**: if set to 1 shows all macro lines processed for debugging purposes. A value of 2 shows all skipped lines as well. A value of -1 shows the lines after full argument evaluation. Set to 0 by default;

- **CHECK**: if set, checks syntax of macro's. This done by ignoring almost all commands after they are parsed.

1.3 Macro's

A CHIPMUNK macro is a sequence of command lines which can be written to and read from disk as a text file.

After definition a macro is just like a chipmunk builtin command: it can be called from the prompt or from within an other macro[1]. The arguments of the macro are available within the called macro as `$1,$2...` . The number of recursion levels is limited to 100 (5 for TurboC). A macro can return a return code.

Macro's can be defined in three ways:

1. by typing the macro lines interactively using `macdef`. It is however not possible to edit previously edited lines;

2. by preparing the macro file (`name.mac`) with your favorite editor and call the macro (which implicitly loads the macro file);

3. indirectly by including the macro definition in another macro. All lines from `macro <name>` up to `endmac` are read literally as the macro <name>. This allows writing of utility libraries that don't clutter a whole directory by the volume of macros.

After definition a macro resides in memory. Use `shomac` to list their names and definitions. Explicit redefinition implies erasure of the previous definition. To enable implicit re-declaration, a macro must be removed using `macdel`.

Within a macro some commands behave slightly different compared to a call from the command line, usually less verbose. Some commands (for example `while`, `if`, `macro`) can only be used from within macro's (the list of commands elaborates on this).

Example:

```
> macdef greet
macdef> copyenv USER
macdef> echo Hello $USER
macdef>
```

[1]Or from within itself. Recursive macros are allowed, but be aware that recursion depth may be limited

```
> greet
Hello stout
> _
```

1.4 Built-in Functions

There are three types of commands available:

- Basic CHIPMUNK commands. These are not application specific;
- Macro control commands. These commands perform flow control for macro's and cannot be used at the prompt;
- Optional commands. These are application specific. Some must be explicitly included others are automatically include for my convenience. For the MCCD-accelerator project the following are included:
  - Memory and I/O operations for manipulating the hardware;
  - Implementations of several test algorithms;
  - Terminal emulation for communication with the i960 processor;
  - MCCD-software interfacing.

1.4.1 Command Line Processing

A command line basically has the following form:

```
command [argument ...]
```

Each command requires a specific set of arguments. An argument could be alphanumeric or a numeric expression. CHIPMUNK expects commands and arguments to be separated by one or more spaces. If an argument contains spaces it must be enclosed in quotes.

When a command line is evaluated all references to variables are substituted by their value. A reference to a variable is made by prefixing its name with `$` or `$$`. The variable's value is treated as a single argument in the first case (`$`), or as a new part of the command line, possibly resulting in more than one argument, in the second case (`$$`).

```
> set testvar "hi there"
> macdef testmac
macdef> echo arg 1 is: $1
macdef> echo arg 2 is: $2
macdef>
> testmac $testvar
arg 1 is: hi there
```

```
arg 2 is:
> testmac $$testvar
arg 1 is: hi
arg 2 is: there
>
```

Expressions are build from (binary and unary) operators (see table 1.1) and numeric values according to three rules:

expression = expression binary_operator expression

expression = unary_operator expression

expression = numeric_value

Any expression can be enclosed by parentheses. A numeric value is treated as a hexadecimal value or, with `#` prefix as a decimal value (note that this is not an operator as opposed to hex to BCD conversion which is an operator).

First unary operators are evaluated from right to left. Then binary operators are evaluated from left to right. Note that this evaluation order is different from most other computer languages which specify an order of precedence and provide operand grouping. Use parentheses to specify the order of evaluation. The command `set` takes the assignment operator (as its second argument).

A few things to note:

- CHIPMUNK is case sensitive;
- operators and special characters can be escaped by preceding it with the escape character `\`, so '`echo \$STATUS \+ \#10`' will print '`$STATUS + #10`' instead of a value;
- all text within two equal quotes (one of ", ' or') is treated as a single argument;
- each command can be preceded by white space;
- when the first non-space character on a line is `#`, that line is a comment;
- for debugging purposes, a completely evaluated line is shown when TRACE is set to -1;
- By setting the variable CHECK to a positive value a syntax check is performed on every macro by to perform syntax check on a macro: each line (regardless flow control stuff) is evaluated (but not executed) when CHECK is set;
- the maximum number of arguments is limited (currently 28 arguments).

<table>
<tr><th colspan="2">unary operators:</th></tr>
<tr><td><code>!</code></td><td>logical not</td></tr>
<tr><td><code>~</code></td><td>bitwise not</td></tr>
<tr><td><code>@</code></td><td>indirection 32-bit access</td></tr>
<tr><td><code>@s</code></td><td>indirection 16-bit access</td></tr>
<tr><td><code>@b</code></td><td>indirection 8-bit access</td></tr>
<tr><td><code>%d</code></td><td>hexadecimal to BCD conversion</td></tr>
<tr><th colspan="2">binary operators:</th></tr>
<tr><td><code>+ -</code></td><td>additive</td></tr>
<tr><td><code>* / %</code></td><td>multiplicative</td></tr>
<tr><td><code>&</code></td><td>bitwise and</td></tr>
<tr><td><code>|</code></td><td>bitwise or</td></tr>
<tr><td><code>^</code></td><td>bitwise exclusive or</td></tr>
<tr><td><code><< >></code></td><td>left and right shift</td></tr>
<tr><td><code>&&</code></td><td>logical and</td></tr>
<tr><td><code>||</code></td><td>logical or</td></tr>
<tr><td><code>== !=</code></td><td>equality / inequality</td></tr>
<tr><td><code>=s=</code></td><td>string equality</td></tr>
<tr><td><code>< ></code></td><td>relational</td></tr>
<tr><td><code><= >=</code></td><td>relational</td></tr>
<tr><th colspan="2">assignment operators:</th></tr>
<tr><td><code>++ --</code></td><td>increment and decrement</td></tr>
<tr><td><code>+= -=</code></td><td>additive</td></tr>
<tr><td><code>*= /= %=</code></td><td>multiplicative</td></tr>
<tr><td><code>~~</code></td><td>bitwise complement</td></tr>
<tr><td><code>|= ^= &=</code></td><td>bitwise or, xor, and</td></tr>
<tr><td><code><<= >>=</code></td><td>left and right shift</td></tr>
<tr><td><code>||= &&=</code></td><td>logical or, and</td></tr>
</table>

Table 1.1: CHIPMUNK operators

1.4.2 Command Descriptions

All sources for the commands can be found in the supplement.

The syntax descriptions of the commands contain the following labels:

- **cmd**: a builtin command;
- **macname**: name of a macro;
- **name**: name of a variable;
- **expr**: expression;
- **str**: alphanumeric argument;
- command specific.

Optional arguments are enclosed in square brackets (`[...]`).

1.4.3 Basic CHIPMUNK Commands

The source for the basic CHIPMUNK commands can be found in the supplement, section `control.c`.

copyenv

description : set a variable from the programs environment
usage : copyenv name
returns : 1 if name not found
remarks : if not found, variable is not created

echo

description : echo the arguments
usage : echo [options] [str] [...]
returns : number of chars printed
remarks : options include -n for no line feed. No output if VERBOSE < 0.

env

description : print all the variables and
usage : env [var ...]
returns : 0
remarks : takes wildcard. Completion on variable names

exit or quit

description : quit the program
usage : exit or quit
returns : CHIPMUNK returns `$STATUS`

help, ?
description : print help about the commands
usage : help [cmd]
returns : number of commands shown
remarks : takes wildcard. Completion on command names

putc
description : echo characters specified hexadecimal
usage : putc [chr ...]
returns : number of chars printed
remarks : no output if VERBOSE < 0.

set
description : set a variable
usage : set name [asignment operator] [expr]
returns : 0
remarks : creates new variable if name doesn't exists. Completion on variable names

syst
description : execute a command on the host system
usage : syst [system command]
returns : command return value
remarks : completion on filenames

timer
description : sample CPU time usage
usage : timer [reset]
returns : CPU time usage since last reset in system clock ticks

unset
description : remove a variable
usage : unset name [name]
returns : 0
remarks : bombs if name doesn't exists. Completion on variable names

usleep
description : sleep for a few microseconds
usage : usleep value
returns : 0
remarks : is busy waiting on some systems

ver
description : print version info about the program
usage : ver
returns : major version number to facilitate version checking.

1.4.4 Memory Operations

The source for the Memory operations can be found in the supplement, section `memops.c`.

and, ands, andb
description : loc is anded with the given mask
usage : and[s,b] loc mask
returns : 0
remarks : access word size is 32-bits without suffix, 16-bits with suffix s and 8 bits with suffix b

dump
description : dump a memory area to screen
usage : dump loc [len]
returns : 0
remarks : default for len is 128 locations (8 lines)

fill
description : Fill memory with a constant.
usage : fill loc len value [...]
returns : 0
remarks : the constant can be one or more words long

free
description : perform a free for previous malloc
usage : free ptr
returns : result
remarks : BASE is not added

malloc
description : perform a malloc to allow dry-running/memory-testing
usage : malloc len
returns : ptr
remarks : BASE is not added

memcpy

description : copy a memory area to another place.
usage : memcpy src len dest
returns : 0

memmap

description : perform the memmap system call to get access to physical memory
usage : memmap file len offset
returns : result

or, ors, orb

description : loc is or-ed with the given mask
usage : or[s,b] loc mask
returns : 0
remarks : access word size is 32-bits without suffix, 16-bits with suffix s and 8 bits with suffix b

poke, pokes, pokeb

description : poke a value into a memory location
usage : poke[s,b](s/b) loc value
returns : 0
remarks : access word size is 32-bits without suffix, 16-bits with suffix s or 8-bits with suffix b

xor, xors, xorb

description : loc is xor-ed with the given mask
usage : xor[s,b] loc mask
returns : 0
remarks : access word size is 32-bits without suffix, 16-bits with suffix s and 8 bits with suffix b

1.4.5 IO Operations

The source for the IO operations can be found in the supplement, section `io_ops.c`. Off course these are only available on machines where the architecture supports IO addressing (currently PCs).

inb

description : input a 8 bit byte from IO addressing space
usage : inb loc
returns : value

inw

description : input a 16 bit word from IO addressing space
usage : inw loc
returns : value

ioperm

description : acquire permission to address IO space
usage : ioperm loc len
returns : results

outb

description : output a 8 bit byte in IO addressing space
usage : outb loc value
returns : 0

outw

description : output a 16 bit word in IO addressing space
usage : outw loc value
returns : 0

1.4.6 File Operations

The source for the file operations can be found in the supplement, file `fileops.c`.

load

description : load a file into memory
usage : load filename loc
returns : 1 on IO error (file not found etc.)
remarks : memory will be filled up to the length of the file. Completion on filenames

log

description : log session to a file
usage : log filename
returns :
remarks : do not use: will not be update anymore

save
description : save from memory into a file
usage : save filename loc len
returns : 1 on IO error (directory not found etc.)
remarks : completion on filenames

1.4.7 Macro Control Commands

The source for the Macro control commands can be found in the supplement, section `control.c`.

ask
description : echo args and returns keycode
usage : ask [str ...]
returns : keycode
remarks : the key is converted to lowercase

else
description : use with if
usage : else
returns : return value of previous command

endif
description : end of if statement
usage : endif
returns : return value of previous command

endmac
description : end of macro definition within macro
usage : endmac
returns : 0

endwhile
description : end of while statement
usage : endwhile
returns : return value of previous command

if
description : jump to else or endif if expr is false
usage : if expr
returns : return value of previous command

mabort

description : abort all macro's
usage : mabort
returns : 0
remarks : reports which macro is aborted (plus line number)

macdef

description : define a macro
usage : macdef macname
returns : 0
remarks : quick and dirty way to define (short) macro's interactively. It is not possible to edit previous lines. An empty line quits macdef.
If the macro macname already exists the previous version is deleted

macdel

description : delete a macro
usage : macdel macname
returns : 0
remarks : use this command if a macro should be reloaded. Completion on macro names

macro

description : define a macro within a macro
usage : macro macname
returns : 0
remarks : macro's defined this way do not require an extra file. Conditional macro definition can be implemented this way.
If a macro name already exists the previous version is deleted

return

description : return from macro with status value
usage : return [value]
returns : n.a.
remarks : if the return from macro is done otherwise, its return value is not defined (although it currently is usually 0)

savmac

description : save macro to file
usage : savmac macname
returns : 1 when macname does not exist
remarks : completion on macro names

shift

description : shift macro arguments
usage : savmac
returns : 1 if there are still arguments left
remarks : argument 1 is lost

shomac

description : show the defined macro's
usage : shomac [macname]
returns : 0
remarks : takes wildcard. Completion on macro names

tstenv

description : test or if args exist as chipmunk variables
usage : tstenv [name ...]
returns : 1 when *all* args exist in env
remarks : takes wildcard. Completion on variable names

1.4.8 Test Implementations

The sources for the Test implementations can be found in the supplement, sections `memtest.c` and `tests.c`.

addrtest

description : address line test
usage : addrtest [-r]ref [-a]amask [options]

-r<ref>	reference adress
-a<amask>	address line toggle mask
-d<dmask>	data mask (ffffffff)
-w<n>	word size (32)
-t<n>	threshold (min. bits diff.)(16)
-p<n>	number of passes (1)

returns : word with faulty address lines
remarks : described in **??**, page **??**.
Displays help when arguments are omitted

calcrc16

description : compute CRC16 of specified location(s)
usage : usage: calcrc16 [-r]ref [-n]len [options]

-r<ref>	reference address
-n<len>	length (bites -a and -c)
-s<n>	stride (bites -a and -c)
-a<amask>	address mask (bites -n and -s)
-c<cmask>	count mask (bites -n and -s)
-d<value>	data mask (0xffffffff)
-w<n>	word size 8, 16 or 32 (32)

returns : 16-bit crc
remarks : make sure to specify either -n OR -a and -c

crashtest

description : performs vital MCCD-access
usage : crashtest address [message]
address target address
message message to print (ERR: AT crashed ...)
returns : 0
remarks : described in **??** on page **??**

datatest

description : detect data line SAF & BF
usage : datatest [-r]ref [options]

-r<ref>	reference address
-c<cmask>	count mask (0)
-p<n>	number of passes (1)
-w<n>	word size (32)
-d<dmask>	data mask (ffffffff)
-s	use walking one test set(off)

returns : number of faulty locations
remarks : described in **??** on page **??**.
Displays help when arguments are omitted

dlbftest

description : data line bf test
usage : dlbftest [-r]ref [<options>]

-r<ref>	reference address
-a<amask>	address mask (0)
-c<cmask>	count mask (0)
-w<n>	word size 8, 16 or 32 (32)
-d<dmask>	data mask (ffffffff)
-t<n>	threshold (50)
-p<n>	number of passes (1)

returns : number of errors
remarks : described in **??** on page **??**.
Displays help when arguments are omitted

dlsatest

description : data line stuck at test

usage : dlsatest [-r]ref [options]

-r<ref>	reference address
-a<amask>	address mask (0)
-c<cmask>	count mask (0)
-w<n>	word size 8, 16 or 32 (32)
-d<dmask>	data mask (ffffffff)
-t<n>	threshold (99)
-p<n>	number of passes (1)

returns : number of errors

remarks : described in **??** on page **??**.
Displays help when arguments are omitted

ghosttest

description : ghost image test

usage : readghost [-r]ref [-x]xmask [options]

-r<ref>	reference address
-x<xmask>	don't care address lines
-a<amask>	addressing address lines (0)
-d<dmask>	data mask (ffffffff)
-i<val>	initial value to expect (none)
-w<n>	word size 8, 16 or 32(32)
-v	write random value (off)
-t<n>	threshold 0..31 max. bits diff. (0)
-p<n>	number of passes (1)

returns : percentage misses.

remarks : described in **??** (page **??**).
Displays help when arguments are omitted

memtest

description : perform specified memory test.
usage : memtest [-r]base [-n]length [options]

-r<base>	base address
	-n<length>number of locations
-a<n>	select algorithm used:
	0. SMATA;
	1. Mats+;
	2. March C-;
	3. March B (default);
	4. March G.
-w<n>	word size n=8,16,32 (32)
-o	random address order (off)
-c<cmask>	counting address order (off)
-b	bit-fault detection (off)
-d<n>	data mask (0xffffffff)
-u<n>	us sleep for delay (100000)
-s<n>	stride n (unit=byte)(0x4)
-p<n>	pass n times (1)
-e<n>	stop after n errors occurred (0)

returns : number of errors
remarks : described in **??**, page **??**.
Displays help when arguments are omitted

pmt

description : pseudo memory test: generates regular pattern
usage : pmt base len stride n
returns : -1 on error
remarks : Performs simple, symmetric march-test n times:
$\{\Downarrow (r0, w1); \Uparrow (r1, w0)\}$

read

description : robust read, returns most read word
usage : read [-r]ref [options]

-r<ref>	reference address
-a<amask>	address mask (0)
-c<cmask>	count mask (0)
-n<n>	length
-s<n>	stride (32)
-w<n>	word size 8, 16 or 32 (32)
-p<n>	number of passes (1)

returns : data read (ave.)
remarks : specify either -n and -s OR -c and -a.
Displays help when arguments are omitted

setfaults

description : set simulated faults
usage : setfaults [-v|-V]
returns : 1 if not in DEBUG mode
remarks : specify `-v` to see what faults are currently simulated, `-V` for all faults and their description

tver

description : print version info about test parts
usage : tver
returns : 0

visit

description : perform read accesses
usage : visit [-r]ref [options]

-r<ref>	reference address
-a<amask>	address mask (0)
-c<cmask>	count mask (0)
-w<n>	word size 8, 16 or 32 (32)
-p<n>	number of passes (1)

returns : number of visits
remarks : displays help when arguments are omitted

write

description : write on specified locations
usage : read [-r]ref [options]

-r<ref>	reference address
-d<value>	data to write
-a<amask>	address mask (0)
-c<cmask>	count mask (0)
-n<n>	length
-s<n>	stride (32)
-w<n>	word size 8, 16 or 32 (32)
-p<n>	number of passes (1)

returns : number of locations
remarks : specify either -n and -s OR -c and -a.
Displays help when arguments are omitted

1.4.9 MCCD Specific

The source for the MCCD specific commands can be found in the file `mccd-interface.c`. See [**?**] for a description of these commands.

send
description : send a string to the board

term
description : go into terminal emulation mode

mccd_init
description : call mccd_init in interface

mccd_mem_init
description : call mccd_mem_init in interface

start_grabber
description : start grabbing....

stop_grabber
description : stop grabbing....

start_960
description : start the 960 doing a contour

get_contour
description : get a contour, and print it

reset_g335
description : reset the G335

send
description : send a string to the board

stop_grabber
description : stop grabber

showcost
description : showcost

showscanm
description : showscanm

showpath
description : showpath

copyframe
description **:** copyframe

showmodelc
description **:** wshowmodelc

showcontour
description **:** showcontour

showcontouri
description **:** showcontouri

showcontours
description **:** showcontours

showrespc
description **:** showrespc

st
description **:** st

gvi1
description **:** gvi1

gvi3
description **:** gvi3

gvi5
description **:** gvi5

gvi7
description **:** gvi7

gvi9
description **:** gvi9

calcost
description **:** calcost

findpath
description **:** findpath

tracon
description **:** tracon

area_nc
description : area_nc

interp
description : interp

smocon
description : smocon

genmod
description : genmod

1.5 Adding Commands

Extra commands can be added in three steps:

1. write an implementation of your command in c;
2. add the command name and the function to be called in `commands.h`;
3. make a new CHIPMUNK executable.

1.5.1 Implement A New Command

You implementation must be in the programming language C. It should include the following:

- inclusion of `config.h`, `chipmunk.h` and `commands.h`;
- a function with a prototype like:
  `int funcname(int argc,char **argv);`

The following example shows how to deal with variables:

```
/*
  NAME extra.c

  add extra commands
  */

#include <stdio.h>
#include <stdlib.h>

/* mandatory includes */
#include "config.h"
```

```
#include "chipmunk.h"
#include "commands.h"

/* prototype should be like this */
int newcommand (int argc,char **argv){
  char *cargv[3];
  int yeah;
  /* use arguments */
  printf("First argument : %s \n",argv[1]);

  /* load variables */
  yeah = getintvar("yeah");

  /* store variables */
  cargv[0] = "set";
  cargv[1] = "yeah";
  cargv[2]= (char *)MY_MALLOC(9);
  sprintf(cargv[2],"%x",yeah+1);
  set (3,cargv);
  free (cargv[2]);

  return (0);
}
```

Check out other sources for comprehensive examples.

1.5.2 Add New Commands

To use a command, CHIPMUNK must know its name, how to call it and what help lines to print. The include file `commands.h` is used for installing commands. Just include the following lines in commands.h:

```
#ifdef INCL_EXTRA
D("newcmd", newcommand, "A new command","[str]")
#endif
```

The new command is invoked with `newcmd` which calls the `newcommand` function. Also a short description is included as well as a description of the calling parameters. The latter two are displayed by help. Note that no semicolon is allowed at the end of the line.

In this example the new command is included only when `INCL_EXTRA` is defined.

1.5.3 Make CHIPMUNK Executable

To make a new CHIPMUNK executable all you have to do is invoke `make` possibly with some additional arguments. The following example shows a UNIX script which makes CHIPMUNK including `test.o` and `extra.o`:

```
#!/bin/sh
make "CONFIG_OPTIONS=-DINCLUDE_TEST -DINCL_EXTRA" \
     "EXTRA_OBJS=tests.o extra.o"
```

The Chipmunk manual

R.D.L. Stout
(with small modifications by) R.E. Wolff
Department of Computer Architecture and Digital Technique
Faculty of Electrical Engineering
Delft University of Technology

March 23, 1995