Elate® Tool Programming Guide (VP)

Version 1.22

1. Introduction

Elate® is a truly portable operating system. The underlying model is of a Virtual Machine or Processor to which all Elate programs are written. Its unique translation technology takes the Virtual Processor byte code and translates it into the native code of the target processor. Normally, translation into native code only takes place when loaded from store, (e.g. disk or network). The translator knows which processor it is running on and can generate the appropriate code. Programs for Elate can currently be written in the assembler language of this virtual processor (VP code), or in C, C++ or the Java™ language.

Elate is designed so that all application programs are written as small sections of code called *tools*, which are executable pieces of code. The whole of Elate, including the kernel and all its functions, is programmed this way.

An application is a tree structure of a number of dependent tools, as shown in Figure 1.1.

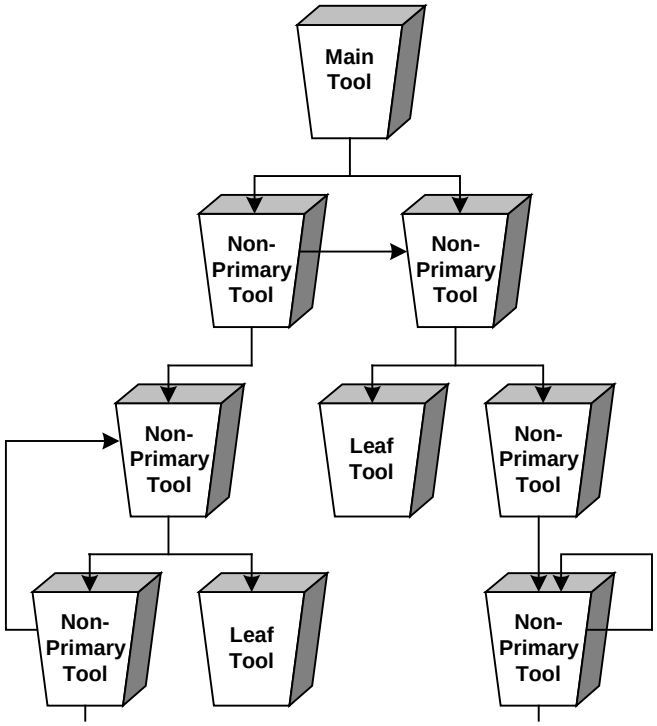

Figure 1-1

Tools are *re-entrant* and *multi-threaded*, being discarded only if they are no longer referenced and only if the memory is required. This ensures that Elate is also extremely memory efficient.

Elate already comes with over 6,500 tools coded and ready for use. These tools are typically less than 1K in size. They include kernel functions, ANSI C Libraries and many others. All are available to the development programmer on the development platform.

This document covers the main aspects of tool programming when writing in assembler, whether VP, the language of the Virtual Processor, or native. Please refer to the relevant Elate manuals for Programming in C, C++ and Java™, and the appropriate Native Code manual provided by the processor manufacturer.

2. Assembling a Program or Tool

To run an application or to call a tool, it must have first been assembled. This section of the manual covers how to assemble and run a 'hello world' program. The Elate Operating System comes with a number of example programs, and these can be found in the directory *demo/example/*, as can the source file of 'Hello World'.

All Elate source files must be suffixed with *.asm* to denote that they are source programs. When they are assembled to .00 byte code files they are unbound and in template form until they are loaded.

To assemble the program 'hello.asm' type the following at the Elate command prompt while in the root directory:

```
$asm -v demo/example/hello
```

In the above example command line, the verbose option has been used (-v) to provide additional diagnostic information. It is recommended that developers new to intent and Elate use this option.
If there were no errors, the assembler will have created a separate file with the name *hello*, suffixed with *.00*, containing VP byte codes. To run this program, type at the command prompt, in the root directory :

```
$demo/example/hello
```

The words "Hello World" are printed to the screen.
Note that the full pathname is required. The suffix is optional for asm files, whereas no .00 suffix should be specified when running a tool, as shown in the example above.

Below is the source code of 'hello.asm'. Every element required for programming a tool is in this program and is explained in the rest of this manual.

```
.include 'tao'
tool 'demo/example/hello',VP,TF_MAIN,1024,0
      ent - : -
      printf "hello world\n"
      ret
toolend
.end  ; the end of source directive is entirely optional
```

It is instructive to look at the disassembly of this code. This can be achieved by:

```
$ dis demo/example/hello.00 | less
```

Note that the disassembly is piped to the utility *less* to allow scrolling of the output.

2.1. The structure of an Application Source File

An application source file consists of any include file(s), the primary or standalone executable tool and any non-primary tools. Non-primary tools coded in the same source file are held separately when assembled and they need not be located in the same directory as the primary tool. On assembly, the tools will be suffixed with .00 for tools coded in VP.

2.2. The Include File

Include files define the equates (constants) and macros needed for an application program. An include file's default extension is *.inc*. All system include files can be found in the default include file location: *lang/asm/include/.*

tao.inc defines standard system items for VP programs and is the most commonly used, non-application specific, include file.

To specify an include file the *.include* directive is used with the name of the include file placed directly after it within inverted commas. The assembler automatically prefixes this location if no path name is specified and suffixes *.inc* unless another extension or "." is specified.

```
.include 'tao'
```

Application specific include files may be absolute, that is prefixed with a "/". File names with no path ("*tao*") are sought in the system include directory */lang/asm/include/.* File names with a path ("myinclude/foobar") are absolute (i.e. taken relative to the root directory) whether they have a leading "/" or not.

```
.include '/demo/example/foobar'
```

Application specific include files may also be relative, in which case the assembler searches from the current directory location. As a rule, the assembler should normally be run from the root directory. This is not strictly necessary, but is merely suggested as a good practice.

```
.include 'myinclude/foobar'
```

2.3. The Primary Tool

As described in the introduction, all application programs consist of one or more tools. The primary or main tool is the tool that is able to be started as a lightweight process and it is this process which executes non-primary tools. The main tool itself becomes a lightweight process when it is spawned by the following:

- *qcall sys/kn/proc/create*
- *qcall sys/kn/proc/start*

The primary tool defines the stack size and global variable space, secondary tools are simply subroutines which are linked together dynamically at runtime.

2.3.1. Setting up a Tool

Tools are referenced by name; this is defined after any include files or defined global variables, but before the *ent* directive and code for that tool. As well as defining the tool name, it is also necessary to define the Assembler Language, the Tool Type, the Stack Size and the Global Variable Size. These are all specified on the same line, separated only by commas.

The Tool Name using the *tool* macro

Positioned after the *tool* macro, which is placed within single quotes, is the name by which the tool is to be referenced, *<pathname>/<name>*. It should be noted that tool names are case sensitive. If two tools are given the same name, differing only by case, they are two distinct and different files.

The Assembler Language
Tools must define the assembler language in which they are written. This can either be VP code or native (e.g. I386) and this must be defined for each tool.

The Tool Flags
Each application or program has one tool defined as the main or executable tool. This is indicated by using *TF_MAIN*. Once the program has been assembled, the name of the main tool is then the executable file.

The Stack Size
The stack size (in bytes) which is to be allocated. This is application dependent but a minimum of 8192 bytes is recommended. Should the stack size be insufficient it can be changed by the programmer. Also, although 8192 bytes is the suggested initial stack size, Elate has some support for dynamically extending the stack at run time if it turns out not to be big enough.

The Global Variable Size
The size of the global variables space (in bytes) which must be allocated for each instance of the application. Zero is permitted The global variable space can be the size of a structure defined before the *tool* macro.

Example uses of *tool* macro for primary tools:

```
tool 'demo/example/hello',VP,TF_MAIN,8192,0
```

```
tool 'demo/example/hello',VP,TF_MAIN,16384,16
```

```
tool 'demo/example/hello',VP,TF_MAIN,65536,GLOBALS_SIZE
```

The *toolend* macro
When all the code for the tool has been written (after the final *ret* in the code, and any data which may follow it) the end of the region started by the tool macro must be closed by the *toolend* macro. As it is possible to have more than one tool in a source file, the programmer has the option to make the end of the file more explicit by the use of an *.end* directive to conclude the file. Note that this is entirely optional as the assembler will finish processing on reaching the end of the source file.

```
      ret
toolend
.end  ; end of source directive - optional
```

2.3.2. Accessing the Command Line Input Parameters

In certain circumstances, it is necessary for a tool or program to access the command line input parameters. This is achieved by the use of the tool *lib/argcargv.* This tool returns C-style argc and argv parameters. It is standard practice for all primary tools to have a call to *lib/argcargv*, although not always required.

The *lib/argcargv* API requires no inputs but outputs are a pointer to the block of argument pointers to the parameters (*argv*), which are fragments of the original command line string, and an integer (argc) which is the number of parameters in the argv block. Parameter 0 is always the name of the program.

```
qcall lib/argcargv,(-:p0 i0)
```

2.3.3. Tidying up before closing a tool

Once all the code for the tool has been written and before any data, it is advisable to carry out some general housekeeping procedures. Elate provides a tool for this called *lib/exit*, which automatically performs these tasks. This tool has the same functionality as the C language *exit()* function.

lib/exit automatically releases any allocated memory blocks, as well as automatically closing any files opened with *lib/fopen*. The standard I/O buffers are flushed and the *stdin*, *stdout*, *stderr* channels are also closed.

lib/exit takes an integer to return to the shell, which is typically zero to indicate success. There are no outputs from *lib/exit*. Although it is not obligatory to make a qcall to *lib/exit*, it is strongly recommended that a call be made before closing the tool with *ret* and the *toolend* macro.

In order to make a successful return status to the shell, i0 is cleared before the *qcall* to *lib/exit*.

```
clr i0
qcall lib/exit,(i0:-)
```

This can also be done more simply as :

```
qcall lib/exit,(0:-)
```

2.4. Non-primary tools

Non-primary tools, that is, tools that are called from other tools, can be coded in the same source file as the primary tool. The parameters *TF_MAIN*, stack size and global variables are not defined. A tool file will be created, suffixed with 00 because it is coded in VP code. In all other ways, coding of non-primary tools is the same as a main or primary tool. Note the equated constant “tool 'foo', VP” is used to denote a non-primary tool.

```
tool 'demo/example/hello2'
```

3. Structure of the VP Processor

3.1. Register Files

VP code was specifically designed to liberate the programmer from the restrictions that a limited number of registers can impose. Therefore, each tool or sub-routine has its own set of five register files. Each register file contains a specific type of data as can be seen in Table 3.1 below.

The programmer specifies at the beginning of a tool or subroutine, by use of the '*ent*' directive, the total number of register inputs and outputs required by the piece of code being written. This is done for every tool or subroutine, effectively making an infinite number of registers available to the developer programmer.

• i0	integer 32 bit
• l0	long 64 bit
• f0	float 32 bit
• d0	double 64 bit
• p0	pointer 32 bit

Table 3-1

3.1.1. Integer Registers

Integer registers may contain integer, bytes, characters etc. and they start at i0. The default for integer registers is 32 bits. All VP implementations support 32 bit integers. Elate assumes 32-bit integer on all instructions unless qualified otherwise. (See Chapter 5.1
The basics)

There is also a special purpose register, *si*, the signature integer. When making an ncall to a method of an object, this special integer register is used. However, it should never be used by the application programmer (see the manual *'Object Based Programming with Elate'* for more information on making *ncall*s).

3.1.2. Pointer Registers

Pointer registers are nominally 32 bits, like integer registers. All VP implementations support 32 bit pointers. In addition to the general purpose pointer registers there are also 4 special pointer file registers. Special registers are used in any place where ordinary registers are used. Care must be taken as other instructions may alter them (see Chapter 10). These special registers are tabled below. Examples of the use of these registers are given later.

sp stack pointer
gp global pointer
lp link pointer
pp parameter pointer

3.2. The *ent* directive

The number of registers and the type required are specified at the beginning of a non-primary tool or routine. All tools and routines must have an entry directive as their first instruction before any other code. The standard entry point is *ent.*

Parameters are passed by any registers in each register file, with parameters appearing in low number registers in the respective files. Register files are local to a routine and unless they are explicitly

passed to the called routine, they will not be visible in that routine and will be preserved when control is returned to the caller.

Parameters passed in and out must be specified on the entry instruction. Primary tools with the *TF_MAIN* bit set have an *ent* directive, which does not have any registers for input nor for output as the routine is self contained. All other tools must specify exactly which registers are to be used. All registers used must be contiguous starting from 0. The parameters are separated by a ':' with the inputs first and the outputs second.

A single '-' denotes no parameters, so a primary tool *ent* directive would be:

```
ent -:-
```

A non-primary tool *ent* directive might be:

```
ent p0 p1 i0 i1:i0
```

Local registers need not be specified, as this is taken care of automatically by the assembler. No other instructions should be placed before the *ent* directive.

A further three entry directives are available and are only to be used in the special circumstances outlined below:

entl	leaf tools and subroutines only (ones that call no other tools or subroutines)
entih	interrupt handlers only
entd	defaultmethod in object programming only

4. VP Instructions

4.1. The basics

Instruction parameters can take three specific forms: constant, register or expression. Instructions are encoded as a single byte followed by parameters. One of the most commonly used instructions is *cpy*, which is used for both register and memory access as well as moving constants into registers or memory.

The different qualifiers for instructions of this kind are shown in Table 4.1 (byte, 16 bit short etc). A qualifier is only required in cases where a register other than 32 bit integers is required, and the assembler is unable to automatically determine which form of register is required. It is also important to ensure that data in memory is appropriately aligned. Note that this also applies to structures. The correct alignment for each data type, with examples of how qualifiers are used, are in Table 4.2.

.b =	byte
.s =	16 bit short
.i =	32 bit integer
.l =	64 bit long integer
.f =	32 bit float
.d =	64 bit double float
.p =	32 bit pointer

Table 4-1

`cpy [p0],i0`	alignment: 4 bytes
`cpy.i [p0],i0`	alignment: 4 bytes
`cpy.b [p0],i0`	alignment: 1 byte
`cpy.s [p0],i0`	alignment: 2 bytes
`cpy.l [p0],l0`	alignment: 8 bytes
`cpy.f [p0],f0`	alignment: 4 bytes
`cpy.d [p0],d0`	alignment: 8 bytes
`cpy.p [p0],p2`	alignment: 4 bytes

Table 4-2

To copy the contents of p0 into p1:

```
cpy.p p0,p1
```

To copy the contents in the memory address pointed to by p0 into i0:

```
cpy [p0],i0
```

To copy the contents of i0 to the contents of the memory location pointed to by p1:

```
cpy i0,[p1]
```

To copy 42 into i0:

```
cpy 42,i0
```

To copy 99 into the memory location pointed to by the pointer p0:

```
cpy 99,[p0]
```

There is also a non-aligned version of the cpy command. To see how this works examine the following code, assuming p1 and p0 are integer aligned:

```
cpy [p0],[p1+1]
```

This statement would cause Elate to execute a hard-coded breakpoint when running the checking translator version of the build under the control of *ebug* (and possibly a system crash when using the non checking translator version of the build). This is because the destination is not integer aligned; it may be the programmer actually wanted to do the following:

```
cpy [p0],[p1+4]
```

This code would run perfectly. If a non-aligned copy were actually required then the correct code would be:

```
cpy.ni [p0],[p1+1]
```

Because there is a requirement that longs are long-aligned, and there is also a *cpy.nl* instruction to manipulate unaligned longs. Both the *cpy.ni* and *cpy.nl* operators are likely to decrease efficiency.

4.2. Expressions

More complex operations are generated by expressions. Expressions may contain expressions within themselves. Most VP operations are performed by expressions that are copied using the *cpy* instruction to a destination register. By using an expression on the left, much more complicated operations can be performed:

To add 6 to i1 and to place the answer in i3:

```
cpy (i1 add 6),i3
```

It is important to note once again that expressions themselves can have other expressions within them. Therefore, to multiply i2 by 6 and then add to i1, placing the answer in i3, would be coded as follows:

```
cpy (i1 add(i2 mul 6)),i3
```

The assembler allows for the following arithmetical symbols within expressions:

+	add
-	subtract
*	multiply
/	divide

The precedence of the operators in expressions is now as in the C language.

For more detailed information on expressions please refer to the VP reference documentation.

Macros are provided for some mathematical operations such as *add*:

```
add 6, i0
```

The above line generates the following code:

```
cpy (i0 add 6),i0
```

4.3. qcall, go, gos and ncall

qcall, go, gos and *ncall* are four ways of transferring execution to another section of code.

4.3.1. qcall

The *qcall* macro takes the name of a tool as a parameter, followed by the input and output registers. The programmer specifies appropriate registers for each individual application or routine. Local registers need not have been specified at the beginning of the tool.

```
qcall lib/argcargv,(-:p0 i0)
```

The tool is loaded and bound when the application referencing it is loaded. It will remain available in local memory for at least as long as the referencing application is in memory. It will not be relocated in memory whilst the caller is present.

Please note that toolnames are not filenames. Although the mapping from toolname to filename is simple, in that the '/' is used as a directory seperator by the tool-loader, toolnames are simply long names. This is very similar to the manner in which the Java language makes use of '.' as a separator, which is converted to '/' for loading from disk). The effect of this is that toolnames are absolute within the Elate filetree, and that tools cannot be moved around within the Elate filetree.

A qcall has the following features:

It may be non-virtual

This means that the tool was loaded and bound when the object referencing it was loaded. It will remain available in local memory at least as long as the referencing object is in memory. It will not be relocated in memory whilst the caller is present.

It may be virtual

This means that the required tool need not be in local memory at the time it is required. If the tool is not available in local memory the tool is loaded and bound before it is available for use. On exit from the tool its memory space may be deallocated by the kernel in order to free local memory space. While the tool is referenced it will remain in memory, however it cannot be assumed that this will be the case when the tool is no longer referenced.

It may be semi-virtual (*VIRTUAL+FIXUP*)

This means that, in the same way as a virtual tool, it is only loaded on the first call to it, and not when the caller is loaded, but is then treated as non-virtual. i.e. it remains in memory until no longer referenced by the calling tool.

It may be embeddable

This means that the entire body of the subtool is inserted in-line. Please note that embedding should be used sparingly. It is normally used to qcall a tool where a native version exists that is simply a single instruction. For more information about this command please see sys/tr/embed.html with the Elate release.

e.g.

```
  qcall demo/mytool,(p0:i0)                    ;normal == non virtual
  qcall demo/mytool,(p0:i0),RF_VIRTUAL         ;virtual
  qcall demo/mytool,(p0:i0),RF_VIRTUAL+RF_FIXUP ;virtual+fixup
  qcall demo/mytool,(p0:i0),RF_EMBED           ;embed
```

4.3.2. go & gos

go transfers execution unconditionally to a label **within the same *ent* block**, so no register passing is required.

```
go next_routine
```

gos transfers execution to a label placed just before **the start of a different *ent* block**. In this case, it is necessary to specify the registers to be passed.

```
gos sub_procedure,(inputs:outputs)
```

4.3.3. ncall

This is used to call a named method of a class. The input and output registers must be specified.

```
ncall p0,drink,(p0:i0)
```

Please note that sometimes the return data from a tool, subroutine or ncall is not required. This can be identified by using the zap operator '~' (tilde). This is an optimisation that tells Elate not to bother returning parameters on this call, thereby increasing speed and saving stack space.
e.g.

```
      qcall demo/example/mytool,( i0 : i~)
```

4.4. Labels or Tags

Labels in VP are special tags, which identify a location. Non-data tags are referenced by *go* and *gos* instructions:

```
gos go_subprocedure
```

Data tags are referenced by *cpy*.

```
cpy [data_label], i0
```

Tags are specified with a trailing colon ':' Data tags are labels in the data section, which is introduced with the .data directive data.

```
str_label:  dc.b “some bytes”,0
            .align                  ; makes sure of word align
data_label: dc.i 12,34,56
```

Note the use of the *.align* directive to ensure that data is correctly word aligned.

4.5. Structures and Memory Allocation

The VP assembler supports structures in order to simplify defining and accessing variables. Structures can be used to define the format of:

- Global variables
- Local variables
- Allocated memory blocks

A structure definition does not reserve memory space anywhere but only defines the names, offsets and size of a structure. When some memory of the given size is allocated, the fields can be referred to by their names, rather than by offset.

There is a balance between having ‘packed’ structures and aligned integers (which may provide better performance). When the structure has been ‘packed’ (without any padding to ensure integers are integer aligned), *cpy.ni* and *cpy.nl* should be used (with consequent decreases in performance on some target platforms).

Please note that, it is necessary to use nint32 (or other corresponding directive) to define an int32 at an unaligned offset, as using int32 (or other corresponding directive) in cases where the next offset is

unaligned will generate an error. All of int16/int32/int64/float32/float64 have an "n" prefixed version. For further information on this, please *see 'The Elate Assembler Reference Manual'* (*app/asm/ref.html*).

In the case of a mixed structure, any longs and doubles should precede integers and floats. These should be grouped together, by decreasing size (anything that is 8 aligned going before anything that is four aligned, and so on). This is because the start memory blocks, i.e. *malloc,* are initially long aligned, and then proceed onwards.

4.5.1. Global Variables

A global structure is defined before the tool macro to allow the variables to be managed by the kernel. We define the *structure* to begin at offset 0 (blank). The variables required are listed, specifying the type and the name. *size* names the current structure offset.

```
structure
      int32 NAMEANY1_A
      int32 NAMEANY1_B
      int32 NAMEANY1_C
size NAMEANY1_SIZE
```

In the tool definition the size of the structure would be NAMEANY1_SIZE.

```
tool 'demo/example/hello',VP,TF_MAIN,8192,NAMEANY1_SIZE
```

The named global variables can now be accessed by using *gp,* the globals pointer.

```
…
cpy 1,[gp+NAMEANY1_A]
cpy 2,[gp+NAMEANY1_B]
clr [gp+NAMEANY1_C]
…
```

4.5.2. Local Variables

The structure is defined inside the tool if it is a local structure. We define the *structure* to begin at offset 0 (blank) and list the variables required specifying the type and the name. *size* names the current structure offset.

```
structure
      int32 NAMEANY2_A
      int32 NAMEANY2_B
      int32 NAMEANY2_C
size NAMEANY2_SIZE
```

The structure can then be allocated memory from the stack inside the tool :

```
allocstruct NAMEANY2_SIZE         ; allocate NAMEANY2_SIZE bytes on stack
```

The structure is accessed by using the stack pointer, *sp*. However, if more than one structure is to be allocated on the stack, you must preserve the stack pointer and use standard pointer registers instead. To allocate a structure using a pointer register:

```
allocstruct NAMEANY2_SIZE,p0
allocstruct NAMEANYOTHER_SIZE,p1
```

To free space allocated on the stack by allocstruct use freestruct n, where n is the number of bytes to free. Note that allocstruct/freestruct pairs must be matched and nested. The exception to this is at the end of sub-routines, where it is possible to leave out the freestruct instruction – this is because ret frees up any memory allocated on the stack. It is also possible to use library functions such as *lib/malloc* and *lib/free* to allocate memory.

4.5.3. Allocated memory blocks

Memory blocks can be allocated using library functions such as *lib/malloc* or the underlying kernel functions such as *sys/kn/mem/allocdata.*

By way of example, consider a simple buffer, declared as a global structure :

```
structure
      pointer BUF_START
      int32 BUF_SZ
size BUF
```

As the above structure is global, there is no need to specifically allocate memory for the structure. It is necessary to allocate memory for the buffer itself and its size may well be decided at run time.
To specify the size of the buffer, for example 1k :

```
cpy  1024,[gp+BUF_SZ]
```

To allocate memory for the buffer :

```
cpy [gp+BUF_SZ],i0
qcall sys/kn/mem/allocdata,(i0 : p0 i0)
```

The returned pointer is then stored in the global variable for later use :

```
cpy.p  p0,[gp+BUF_START]
```

When the buffer is no longer required, the memory can be deallocated thus :

```
cpy [gp+BUF_START],p0
qcall sys/kn/mem/free,(p0 : -)
```

Alternatively, the memory could have been allocated using lib/malloc :

```
qcall lib/malloc,(i0 : p0)
```

Freeing the memory block after calling *lib/malloc* is done by making a *qcall* to *lib/free.* However, advantage can be made of the call to *lib/exit* at the end of the tool (see Chapter 3.2.3) and allowing this to free the C library allocated memory blocks instead.

4.5.4. Bitmap structures

It is possible to create bit maps or bit field structures in VP. These make use of the *dbitstart* and *dbit* assembler macros. dbitstart indicates the start of the structure and dbit defines the name of the actual bit for ease of reference in the code. For example:

```
dbitstart    ; mask,bit
      dbit FLAG0,BFLAG0
      dbit FLAG1,BFLAG1
      dbit FLAG2,BFLAG2
      dbit FLAG3,BFLAG3
      dbit FLAG4,BFLAG4
      dbit FLAG5,BFLAG5
      dbit FLAG6,BFLAG6
      dbit FLAG7,BFLAG7
```

In the above example the values associated with the mask names would be as follows:

Name	Value

FLAG0	1
FLAG1	2
FLAG2	4
FLAG3	8
FLAG4	16
FLAG5	32
FLAG6	64
FLAG7	128

And for the bit names:

Name	Value
BFLAG0	0
BFLAG1	1
BFLAG2	2
BFLAG3	3
BFLAG4	4
BFLAG5	5
BFLAG6	6
BFLAG7	7

This aids considerably when performing bitwise operations on data.

4.6. Program Control

4.6.1. Loops

There are several types of loop construct in VP. They are implemented as macros. In the following notes the [.x] indicates the data type that is being tested e.g. .p for pointer. For integer .i is optional.

while – endwhile

```
while[.x] <condition is true>
      :
endwhile
```

Note that a test occurs before entering the loop so the code inside the loop may or may not run.

for – next

```
for  [<register>|<constant>,]<register>
      :
next <register>
```

The *for* macro basically sets up a label at the start of the loop. If applicable it will also copy a constant or the contents of another register into the counter register. Note that <*register*> can be integer or long. The *next* macro will decrement the counter register contents and jump to the label set up by the *for* macro, as long as the contents of the counter register is greater than zero.

repeat-until

```
repeat
      :
until[.x] <condition>
```

Note that the body of the repeat loop is always entered at least once.

loop-endloop

```
loop
```

```
      if i1=1
            break;
      endif
      :
      breakif i0 = 0
endloop
```

This forms an unconditional loop. To exit the loop you must use a *break* or *breakif* statement.

As well as the *break* statement it is also possible to control loop iteration through use of the *continue* statement. This will cause the loop to jump to the next iteration of the loop, but does not necessarily terminate the loop.

4.6.2. Conditional code

To support conditional code execution VP has the following constructs. As before [.x] is the data type being tested and for integer this is optional.

if-elseif-else-endif

```
if[.x] <condition>
      :
elseif[.x] <condition>
      :
else
      :
endif
```

switch-whencase-otherwise-case-endswitch

The basic syntax is :

```
switch
      whencase <condition1>
      whencase <condition2>
      otherwise
            :
            break
      case <condition1>
            :
            break
      case <condition2>
            :
            break
endswitch
```

An example, *switch.asm*, is included in the /demo/example directory of the int**e**nt release.

bool macro

The *bool* macro provides a convenient way of performing a conditional jump.
The basic syntax is :

```
      bool <condition>,label
```

For example :

```
      bool i0=0,exit
```

4.7. Coding Macros

VP allows the programmer to develop a range of macros to enhance programming productivity. The general format of a macro is :

```
.macro <macroname>
      ; body of macro
.endm
```

An example of a trivial macro is shown below :

```
.macro mymacro
      .check %n = 3
      clr %3
      add 3,%1
      add 3,%2
      add %1,%3
      add %2,%3
.endm
```

The macro could then be called from VP code in the following manner :

```
mymacro i0,i1,i2
```

In this case 3 would be added to i0 and i1. These two registers are then added together with the result placed in i2.

Note the use of the *.check* directive. This checks the calls to the macro at assembly time and validates the number of parameters passed. Should a call have an incorrect number of parameters, in the above example, an assembler error message will be displayed.

4.8. Defines

It is possible to designate labels to registers using the *defbegin-defend* construct. This can make code more readable in certain circumstances. The syntax is demonstrated by the following example :

```
defbegin 0
      defp my_ptr
      defi an_integer
      defi a_byte
      ;
      ; code using the above register names
      ;
defend
```

In the above examples the label *my_ptr* is allocated to p0, *an_integer* is allocated to i0 and *a_byte* is allocated to i1.

Code can then manipulate the defined labels as if they were registers e.g. *cpy 2,an_integer*. On the *defend* the registers will correctly be deallocated and can then be used if required in the normal way.

It is also possible to have nested *def* blocks. Note the use of 0 after the *defbegin*. This indicates that this is the outer *def* block, or top level. This can provide an extra safeguard in complex programs as if the programmer tries to define another outer block, with *defbegin* 0, the assembler will give an error.

Nested blocks arise where you want to use some registers temporarily and would like the assembler to deallocate the registers so that they can be used elsewhere in the program. The following example illustrates this point :

```
defbegin 0
      defi nVar1
```

```
      defp pVar2
      :
      if nVar1=1
            ; I need some new registers just for this section of code
            defbegin
                  defi nLoc1
                  defp pLoc2
                  ; use these variables just in here
                  cpy –1,nLoc1
                  :
            defend
            ; defend will make sure that
            ; the registers allocated to nLoc1 and
            ; pLoc2 are now made available
            ;for use in the rest of the program.
      endif
      :
      :
defend
```

4.9. Tracing

To assist with debugging VP programs several tracing facilities are available.

Trace device
The trace device is used for troubleshooting application programs. It is typically used for displaying debugging information and error messages. In order to write messages to the trace device, the programmer can use the *tracef* macro, discussed below.

By default the trace device is the screen but can be redefined to be a file using the shell command *ftrace* <*filename*>. For example, *ftrace myfile.txt* would cause trace output to go to the file *myfile.txt* rather than the screen. In order to set the trace device back to the screen again simply type *ftrace*.

Tracef
This macro enables messages and data to be written to the trace device. The syntax is identical to *printf*, for example VP source can contain a statement such as :

```
      tracef “error code : %d\n”,err_code
```

With *tracef* the characters are printed directly to the trace device, without buffering. The *tracef* macro has code that waits for an acknowledgement from the trace device, to say that the characters have been printed by the trace device. Note that this behaviour is blocking in nature. This is unlike *printf* where characters are printed to a buffer that is flushed only when the buffer is full or when the buffer is flushed specifically by use of a library function. Note that *printf* is non-blocking in nature. *printf* is thus not ideal for debug purposes as the next instruction after a *printf* may cause the program to crash, however the characters that *printf* was to print may not yet have printed (flushed from buffer), making tracing of execution with *printf* very difficult. *Tracef* was designed to circumvent this problem. Both *printf* and *tracef* will work in a multi-processor environment where the trace device (for *ftrace*) or *stdout* (for *printf*) are on a different processor to the one executing the trace statement.

ktrace.log
This is the trace device for the kernel developer. If you are developing low-level code and your system should crash at any point, looking at *ktrace.log*, which can be found in the root directory, may provide useful information.

5. Using macros

As seen earlier the VP programmer can develop macros to aid productivity as well as readability of code. It has also been indicated that many of the programming structures, for instance, while *endwhile* are in fact implemented as macros. There are also a large number of predefined macros to assist in common programming tasks, for example :

- error handling
- linked list manipulation
- general purpose e.g. multi-byte to wide characters

5.1. Error checking macros

These include macros such as :

- *boolerrno*
- *boolnoterrno*
- *breakiferrno*
- *iferrno*
- *errorf*

The first four are examples of macros that test for *errnos*. These macros can be used in conjunction with routines that return an *errno* (a value in the range -128 to -1) when some error condition has been identified within that routine.

By way of example consider a function that configures a small text buffer. It will return a pointer to that buffer assuming that memory could be allocated and internal initialisation could be completed. If the routine is successful it returns a valid pointer, but if not it returns a value in the range –1 to –128 in the register, say p0. The error checking code would be as follows :

```
:
qcall demo/example/makebuff,(i0:p0)
boolerrno p0,buff_fail
:
```

boolerrno checks the value in p0, and if it is in the range –1 to –128 it causes a jump to the label *buff_fail*. As can be seen from the list above there are variations on this basic macro.

Another useful macro is *errorf*. This macro allows a formatted output of data along the lines of *printf* or *tracef*. However, its output is to the file *error.log*, which is located in the root of the Elate directory structure. It is useful for logging error codes and general post-mortem debugging.

```
iferrno i0
      errorf "Error %d occured.\n",i0
endif
```

5.2. Linked list macros

A common programming task is to manipulate a number of objects in a linked list data structure. Elate has a number of macros already coded to reduce the effort required to handle linked lists.

These are given here :

- *initlist* -- initialises linked list
- *addhead* -- add node to head of list
- *addtail* -- add node to tail of list
- *addnode* -- addnode after node specified
- *addnodeb* -- add node before node specified

- *succ* -- get next node ptr
- *succnode* -- get next node ptr jump to label if at end of list
- *pred* -- get ptr to prev node
- *prednode* -- get ptr to prev node jump to label if at head
- *remhead* -- remove node at head of list jump to label if list empty
- *remove* -- remove specified node from list

In order to use these macros the programmer has to understand the data structures that they manipulate. There are three main structures :

- List header -- defined in *equs.inc*
- List node -- defined in *equs.inc*
- User List node -- user defined

The list header is defined as :

```
      structure
            pointer LH_HEAD
            pointer LH_TAIL
            pointer LH_TAILPRED
      size  LH_SIZE
```

The list node is defined as :

```
      structure
            pointer LN_SUCC
            pointer LN_PRED
      size  LN_SIZE
```

The user defined node is application specific but an example is :

```
      structure
            struct NDHDR,LN_SIZE    ; name, size
            int32  APPDATA
      size  ND_SIZE
```

Note that the basic list node structure has been embedded in the user node structure to permit list manipulation.

Before using the list macros, it is necessary to allocate memory for the list head :

```
      cpy LH_SIZE,i0
      qcall lib/malloc,(i0:p0)
      ; insert NULL error checking code
```

We can then initialise the list with initlist p0. This provides us with an empty list. To add items we allocate memory for the nodes to add and then use the appropriate macro, for example :

```
      cpy ND_SIZE,i0
      qcall lib/malloc,(i0:p1)
      ; insert NULL error checking code
      cpy 1234,[p1+APPDATA]
```

At this point the node has been created and initialised. To add it to the list :

```
      addhead p0,p1,p2
```

p0 points to the list, p1 points at the node to add and p2 is a scratch register (used by the macro internally). The macros to remove nodes are used in a similar fashion.

To walk along the list the macros *succ*, *succnode*, *pred*, *prednode* are provided. To illustrate their use consider *succnode*. This macro is given three parameters, a pointer register to receive the pointer to the next node in the list, a pointer register that points to the current node and finally a label. The label provides a useful exception handling facility as succnode causes execution to jump to this label should the current node be at the tail of the list (in which case there is not a 'next node').
For example :

```
      :
      succnode p2,p1,tail_list
      :
tail_list :
      printf "There isn't a next node – you are at the list end.\n"
      go exit
```

The current node is in p1 and the next node comes back in p2. If p1 points to the current tail of the list then a jump to *tail_list* occurs.

5.3. General purpose macros

These consist of macros to carry out such tasks as:

- conversion from multi-byte to wide characters e.g. *mbtowc*
- input/output e.g. *fprintf*

For further details, please refer to the *'VP Reference Manual'*.

6. Using Libraries

VP programmers have at their disposal a powerful array of built in library functions. Currently Elate has a complete library of ANSI C functions, as well as a significant part of the POSIX specification. The library functions are called using a *qcall* as described earlier in this manual.

These library functions are fully documented in the Elate build documentation. Note that this documents ANSI and POSIX functions.

By way of example, the following code demonstrates the use of some POSIX directory functions.

```
      :
      cpy.p dirname,p0
      qcall lib/opendir,(p0:p1)
      if.p p1=NULL
            printf “Failed to open %s\n”,dirname
            go exit
      endif
      :
      loop
      qcall lib/readdir,(p1:p2)
            if.p p2=NULL
                  printf “End of listing.\n”
                  break
            endif
            add DE_NAME,p2
            printf “-- %s\n”,p2
      endloop
      :
      qcall lib/closedir,(p1:i0)
```

Note that library functions are implemented as tools in VP and are thus called in the same way as user defined tools. The library tools have been developed in VP for maximum portability and performance.

7. Native Processor Coding

The information in this section is not normally needed by programmers writing only for the Elate Virtual Machine. It is intended for use by programmers writing device drivers, porting the kernel or writing optimised native code sections.

7.1. Changes to an Application Source File

In order to be able to write a tool in native code a few simple changes are required.
The example below would be required for writing in i386:

```
.include 'tao'
```

7.1.1. Changing the tool definition

The language in which the tool is going to be written must be specified, after the *tool* macro and the tool name, in place of VP.

```
tool 'demo/example/hello',i386
```

No further changes are required and the rest of the tool is coded in the language for the specific processor.

When assembled, a tool file is created with a suffix suitable to the target processor. Table 7-1 lists the processors currently available with their appropriate suffixes.

7.2. Tool Selection, VP or native

At load time Elate's unique technology, *dynamic binding*, searches the tree of tool dependencies. Elate selects all the tools defined, only taking VP (00) tools if a tool with the same processor number as the target processor is not available. It is therefore strongly recommended that an equivalent VP tool is created for each native tool, to ensure portability. Please note that this section is subject to rapid alterations.

- 00 VP
- 01 n/a
- 02 PSC-1000
- 03 i86 common
- 04 i386
- 05 SH-4
- 06 Thumb
- 07 n/a
- 08 arm6
- 09 Arm6f
- 10 SA110
- 11 R4000_L
- 12 R4000_B
- 13 I386f
- 14 I486
- 15 Pentium
- 16 PPC_B
- 17 SH3
- 18 n/a
- 19 PPC_L
- 20 n/a
- 21 R4100_L
- 22 R4100_B

- 23 PentMMX
- 24 Pent II
- 25 V850
- 26 CF52xx
- 27 V850e

Table 7-1

In the above table, n/a generally refers to a processor port that is commercially sensitive.

8. Special Registers

8.1. *sp* - stack pointer

The stack pointer points to the lowest address containing valid data. If the developer programmer wishes to manipulate the stack directly, it should be noted that it is a downward growing stack. (See Figure 8-1.) It is always aligned to a CPU-specific alignment and this alignment is automatically preserved whenever it is adjusted.

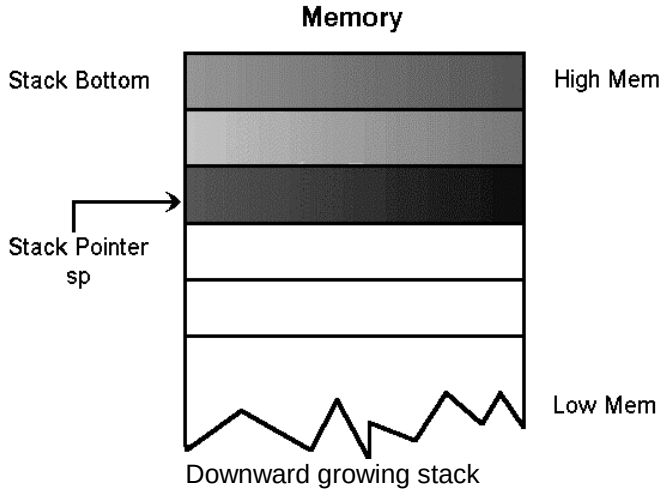

Figure 8-1

8.2. *pp* - parameter pointer

The parameter pointer is a special read only register, which is set by routine entry code to the stack pointer of the caller before the call occurred. It can be used for passing parameters via the stack. Note that the standard parameter passing convention uses registers, not stack, in the vast majority of cases. Usually this is not implemented as a physical register, rather it is a known offset from *sp*. Therefore, *pp* equals the value of *sp* immediately before a *go-sub*.

```
. . .
      structure
      int32 param1
      int32 param2
      int32 param3
      size ppdemo_size

      allocstruct ppdemo_size, p0
      cpy 1,[p0+param1]
      cpy 2,[p0+param2]
      cpy 3,[p0+param3]
      gos ppdemo,(-:i0)
      tracef "result = %d",i0
      freestruct ppdemo_size
. . .
ppdemo:
ent -:i0
cpy [pp+param1],i1
cpy [pp+param2],i2
cpy [pp+param3],i0
add i1,i2
mul i2,i0
ret
```

8.3. *lp* - link pointer (return jump address)

The link pointer contains the return address which the routine should jump to after tidying up the stack in a *ret*. This register may be read and written, however changing it to point elsewhere will cause *ret* to return to a different location.

8.4. *gp* - global pointer

The global pointer (gp) points to an area of thread-wide memory. The size of the user definable area is specified in the main tool. (See Chapter 4.5.1 Global Variables). As the global pointer is set up by the kernel it should never be modified by any other code. This is because gp points to an area that contains both the user data whose size is specified in the main tool, and some data the kernel needs to maintain the thread

9. Planned Changes

Future releases of Elate are intended to use a new version of the assembler syntax. This improved version allows the programmer to deal in abstract data types, in a similar fashion to C or Pascal. The assembler keeps track of these types and enforces type-equivalence over assignment. Other features, listed below, have been added to support the typing.

9.1. Type Checking

The major change to the assembler is the addition of strong typing. This is essentially the rigorous enforcement of type rules with no exceptions. All types are known at assembly time. The advantage of this is that strong typing catches more errors at assembly time than weak typing, resulting in fewer run-time exceptions

Where an incorrect type is used an error will be thrown. Registers may be assigned names using *regdef*. This command replaces the defines (defi, defp, etc.) previously used.

It should be noted that the new assembler syntax automatically generates the correct kind of load for instructions such as *cpy*. When using the new assembler, it is therefore unnecessary to specify a qualifier.

9.2. Types

The *typedef* instruction (borrowed from C) introduces user types to the assembled module. Here is an example:

```
typedef int32 int
```

This is built into the system include files, so a new type *int* is available to all assembler programs. Now we can define a new type:

```
typedef int typeone
```

This introduces a new type *typeone* based on *int.* Note, at this point we have no variables, registers or other storage locations associated with *typeone*, just a type. If we define a new type, also based on *int*:

```
typedef int newtype
```

the new type will *not* be assignment-compatible with a variable of type *typeone* despite their being defined on the same base type. This is different from C, and follows to the philosophy that naming a new type implies some conceptual difference from other types.

9.3. Register Names

The *regdef* instruction lets the programmer tie a new (user-defined) type to a register. For example:

```
regdef typeone examplereg
```

Now *examplereg* is a register to which the programmer can assign a value of '*typeone*.'

9.4. Records

In future releases the structure syntax is to be replaced by a new *record* syntax. Records create a separate namespace for their members. While the members of a structure are individual entities within the assembler, a record and its members are tied together as a single conceptual unit.

```
record widgettype
      int A
```

```
      typeone value
      short C
endrecord
```

The record above is an abstract entity, as was the ‘structure’..
Some attributes are allowed on the record definition, and on the definition of its individual members. These are *extends, unaligned*, *align*, and a constant repeat count (in square brackets).

During use, a pointer register will generally be declared to point to a record of the appropriate type. This is done using the *regdef* command:

9.5. Regdef

```
regdef widgettype [widget]
```

Which would reserve a VP pointer register to be used for accessing the above record. Note the use of square brackets to denote a reference: this should be read *widgettype* is the type of [widget] (note, again, the C connection). Somewhere, then, the programmer would assign an address to the register before referencing the record’s members.

```
cpy widget_address, widget
```

or perhaps

```
qcall lib/malloc, (sizeof (widgettype) : widget)
```

(sizeof is an assembler function added for the strict typing)
These variables would then be accessed as below:

```
cpy 1, [widget.A]
cpy examplereg, [widget.value]
```

Note that with the new syntax the assembler would automatically substitute ‘*cpy.s*’ for ‘*cpy*’ when used with the short variable C. There is therefore no need for the programmer to remember to add the appropriate qualifier for to this kind of command.

9.6. Casts

Casts are a construct to specify that an expression's value should be converted to a different type. These follow the C style. Two points worth noting are: they *always* succeed at the strict typing level, and they *never* generate any type conversion code. If the resulting VP code attempts to copy an *int* to a *float*, this would fail at the basic VP assembler level.

```
cpy (typeone)numnewtypes, examplereg
```

```
cpy (ftypeone)(i2f numnewteypes), fexamplereg
```

As can be seen in the second example, it is necessary to explicitly include any type conversions that require some action at the VP level.

9.7. Extended *ent-ret* Syntax

The syntax for the *ent-ret* combination provided by Tao’s new assembler allows comma-separated lists of arbitrary expressions, and type checking of returned arguments against those expected by the *ent* line. It can also specify its input arguments in a manner similar to the *regdef* instruction, and these arguments may then be used within the code. This prevents register input/output clashes, as there are no identifiers associated with the output arguments.

When using the new syntax of *ent*, it is necessary to use the corresponding typed version of *ret*. The arguments are type-checked against the output types specified on the *ent* line. It should be noted that it is not possible to assume any evaluation order.

There is also a new directive *entend* which denotes the end of an ent block. This makes it easier for the assembler to keep track of scopes, and introduces a marker for the programmer, whereas before an *ent* block was terminated by the next *ent* directive or the end of the tool, etc.

A non-primary tool *ent* directive might therefore be:

```
ent (string a, char [b], int c, int d : int, string, char[])
```

Note the use of parentheses. The corresponding *ret* instruction is of the form:

```
ret (-1, a, b)
```

In the above example, a and b would have to be p0 and p1, because the parameter passing regime demands it. The ret instruction would simply copy –1 into i0 and return.

```
ret (-1, (string)b, (char [])a)
```

This is casting the return types to stop the assembler objecting, and requires the swapping of p0 and p1 before the ret instruction is issued. The assembler would automatically generate the code for this swap.

9.8. Other Builtin Functions

To make the extended syntax more useable, the following assembler functions have been added: *sizeof, typeof, untype*. Here are some examples:

```
qcall lib/malloc, (sizeof (widgettype) : widget)
```

```
.if sizeof (int) == 4         ;it always will!
```

```
.if typeof (%1) != 0          ;is %1 a variable of user-type
```

```
.if typeof (%1) == typeof (widgettype)
```

The result of typeof (Variable) is 0 for all base VP types (l,l,f,d,p) and a unique handle for all user types.

```
cpy.s [untype (widget) + 2], nutsandbolts       ;remove user typing
```

10. Source Code Syntax

The following is the syntax of standard Elate VP commands, which should appear in all normal VP source code files. This is an incomplete list.

10.1. Include Files

```
.include '<filename>'
```

10.2. Tools

Primary Tool

```
tool '<toolname>',VP,TF_MAIN,<stacksize>,<globalssize>
      ent - : -
      …
      ret
toolend
```

Non-primary Tool (assuming VP)

```
tool '<toolname>'
      ent <inputs>:<outputs>
      …
      ret
toolend
```

10.3. Source File End Marker (optional)

```
.end
```

10.4. Calling Mechanisms

tool call (quick call)

```
      qcall <toolname>,(inputs:outputs)[,VIRTUAL[+FIXUP]]
```

method call (named call)

```
      ncall p0,<methodname>,(inputs:outputs)
```

10.5. Structures

Global Variables

```
structure
…
size <GLOBALSTRUCTURENAME>
```

Local Variables

```
structure
…
size <LOCALSTRUCTURENAME >
```

11. More Example Code

Refer to **.asm* in the *demo/example* directory of the Elate build.