Elate® Tool Programming Guide (C)

Version 1.26

Credits

The Elate® VPCODE 2 port and re-targeting of GCC was performed by DSS Distributed Systems Software, Inc., which also developed the vpcc/vpld support programs and the compiler related content in this documentation for VPCODE 2. The VPCODE 1 version of the compiler was written by NSG Network Software Group, Inc., a wholly-owned subsidiary of Open Text Corporation.

DSS Distributed Systems Software, Inc.
#103 - 7500 Abercrombie Dr.
Richmond, B.C., V6Y 3J9 Canada

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 native code of the target processor. Normally, translation into efficient 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 either 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. This is called *object-based* programming and is supported by Elate's unique usage of *Dynamic Binding* technology.

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

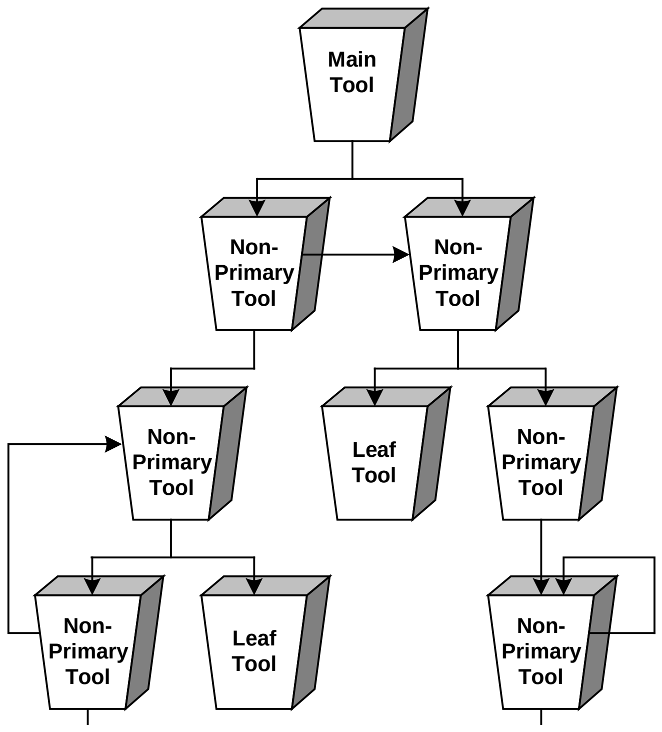

Figure 1-1

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 C. Please refer to the relevant Elate manuals for Elate Tool Programming in VP, C++ and Java™ technology.

2. Compiling 'Hello World'

This chapter covers how to compile and run a program. The example prints the words 'Hello World' to the screen. The Elate Operating System comes with a number of 'C' example programs, and these can be found in the directory *demo/example/c/*.

To compile the program *'hello.c'* type the following at the Elate command prompt:

```
$ vpcc demo/example/c/hello.c -o demo/example/c/world
```

The instruction to invoke the compiler is *vpcc.* The compiler is then given the name of the source program, including its full pathname, which it is to compile. Note that the shell will check for programs in */app/stdio* by default. Generally in Elate commands are typed while in the root directory, with paths given relative to the root. Compiling *must* take place from the root directory.

The option *-o* specifies the output tool name. If a destination path is not specified, the default location is '/'.

The line above invokes the compiler to compile the program *demo/example/c/hello.c,* telling it to output the tool to the same directory location and to name the tool 'world'. Therefore, if there were no errors, the compiler will have created a separate file with the name 'world', suffixed with *.00*. This is the executable file. To run this program, type at the command prompt:

```
$ demo/example/c/world
```

The words "Hello World" are then printed to the screen.

In the next chapter, the elements that make up the program *hello.c* are explained in detail.

3. Building and using tools with C

In the C source code it is possible to call any Elate tool, regardless of the language that was used to code that tool originally (e.g. C/C++ or VP). Additionally, compiled tools whether written in C or C++ can be called from VP source code. Note that in all cases source code is compiled to a tool in VP binary format.

The tool calling mechanism is a powerful facility for making Elate tool calls appear as regular functions. Because tools are not regular functions, some constraints exist.

Most Elate tools already have the calling mechanism enabled through existing header files. The Elate reference material clearly documents these C interfaces, if they already exist. It is only necessary to use the special declaration syntax explicitly for new tools that are to be called from within the C coded tool or application.

3.1 Declaring Tool Calls

Tool calls are declared using a modified version of the *__attribute__* mechanism of GNU C. The *__attribute__* mechanism differs from regular GNU C in that the attribute parameters are strings instead of identifiers. To declare a C function as a tool call you must specify the following:

- The full C declaration of the function, including the return type and parameters
- The assembler instruction to call the tool

For example, the declaration of the *lib/acos* tool is as follows:

```
extern double acos(double x) __attribute__(("qcall lib/acos"));
```

This indicates that a function call to *acos()* should be mapped to the tool invocation *"qcall lib/acos*", with the argument being placed in register *d0* and the result returned in that register.

The compiler determines the assignment of function arguments to registers, but in rare circumstances it is necessary to explicitly state which registers are used to pass the arguments. To override the default tool parameter passing convention, a tool call may look something like this:

```
extern int func(arg1, arg2,. . .,argn) __attribute__((
    "qcall */atool",
    "args reg1,reg2,. . .,regn"     /* MUST BE PROVIDED. */
));
```

This indicates that a function call should be mapped to the tool invocation *"qcall */atool",* with the first argument (left to right) being placed in register *reg1,* the second in *reg2* etc. The return value will be the default set by the tool.

Although rarely necessary, it is also possible to override the default return register:

```
extern int func(arg1, arg2,. . .,argn) __attribute__((
    "qcall */atool",
    "args reg1,reg2,. . .,regn"     /* MUST BE PROVIDED. */
    "return reg1"         /* MUST BE PROVIDED. */
));
```

Functions with variable arguments are supported, as in this declaration of the *printf()* tool:

```
extern int printf(const char *format, ...) __attribute__((
        "qcall lib/printf"
));
```

Once declared, a tool is called much like a regular C function. Note that the name of the tool is similar to the name of a C function in that it is in effect a function pointer. The example below illustrates this idea :

```
#include <stdio.h>
extern double sqrt(double x) __attribute__(("qcall lib/sqrt"));
extern double fabs(double x) __attribute__(("qcall lib/fabs"));

double (*arr[])(double) = {sqrt, fabs};

int main(void)
{
  double x, y;

  x = arr[0](144.0);            /* sqrt() */
  y = arr[1](-x);               /* fabs() */

  printf("Result should be 12: %f\n", y);
}
```

3.2 Creating qcallable tools in C

Elate already has many useful tools or functions that can be called from C programs, but of course application specific tools may still have to be created. To create a qcallable tool in C, the tool C source code must be compiled to VP byte codes, there should be a header file available to specify the qcall attribute and there must be a calling program.

To illustrate, a tool is coded in C that defines a function that, after compilation, is a qcallable tool. For the *sayhello.c* program to print to the screen, the Elate tool *lib/printf* must be called. A parameter *hello_string* has also been defined which will be printed by *lib/printf* when it is used in another program. The source of this tool can be found in *demo/example/c/sayhello.c* :

```
extern int printf(const char* format, ...) __attribute__((
"qcall lib/printf"
));

void sayhello(char *hello_string)
{
      printf ("%s\n", hello_string);
}
```

Before this tool can be used by our next program it must be compiled and a VP tool created. To compile *sayhello.c* the following is entered at the Elate shell command prompt:

```
$ vpcc demo/example/c/sayhello.c -o demo/example/c/sayhello -T
```

As before, the *-o* option tells the compiler the name that the tool is to be given. When calling this tool in another program, the name of the tool and its directory given at this point are the ones that must be specified at the qcall.

-*T* tells the compiler that it is a secondary tool. (See Chapter 8 on Compiler Options).

For a tool to be used in a program, the compiler must know where to find it. A header file is therefore created. A header file tells the compiler of any parameters or return results and the name of the tool to be called. The header file ensures that any functions or data defined is invoked in the way specified as opposed to the conventional C function calling convention.

In our example, the header is called *sayhello.h*, and the tool already coded and compiled is called with a qcall to *demo/example/c/sayhello*. Note that the function name *sayhello* is also defined:

```
extern void sayhello(char *) __attribute__((
     "qcall demo/example/c/sayhello"
  ));
```

When the header file *sayhello.h* is included in any program and a line of code which says *sayhello* is inserted, the compiler will know to emit *qcall demo/example/c/hello,(p0:-).* The example program below is called *testsayhello.c.*

```
#include <stdio.h>
#include "sayhello.h"

   int main(int argc, char *argv[])
   {
        sayhello("this is hello_string !");
   }
```

Below is another simple program, which in this case, calls a user-defined tool to perform a simple mathematical operation. The first program defines the function, the second is the header file and the third is the actual program calling the newly defined function.

demo/example/c/mathtest.c:

```
int mathtest(int x,int y)
{
      return((x*y)-1);
}
```

demo/example/c/mathtest.h:

```
extern int mathtest (int x,int y) __attribute__
((
      "qcall demo/example/c/mathtest"
));
```

demo/example/c/usemath.c:

```
#include <stdio.h>
#include "mathtest.h"

int main(int argc,char *argv[])
{
      int mathtest_result;
      printf("qcalling mathtest...\n");
      mathtest_result=mathtest(5,6);
      printf("mathtest(5,6) returns %d\n",mathtest_result);
}
```

After compiling both of the *.c* files:

```
$vpcc demo/example/c/mathtest.c  -T -o demo/example/c/mathtest
$vpcc demo/example/c/usemath.c -o demo/example/c/usemath -o
demo/example/c/usemath
```

the following is returned when the *demo/example/c/usemath* program is run:

```
--
qcalling mathtest...
mathtest(5,6) returns 29
--
```

The tool's entry point is determined by the first function linked in. In the case of a single C file, it will be the first function listed in that file.

3.3 Device Driver Interrupt Handlers

Interrupt handlers can be coded in C or C++ in the normal way. However, Elate must be made aware that it is an interrupt handler and the following code must be entered directly after any include files:

```
void <name_of_interrupt_handler>
(void*) __attribute__
((“interrupt_handler”)
);
```

4. The Elate Native C Compiler

The Elate Native C compiler is composed of several Elate programs, each of which is responsible for one stage in the compilation process. This chapter describes how these components work together to translate C code into Elate executables.

• vpcc	The front end that invokes programs to execute other compilation stages
• vpcpp	C pre-processor
• vpcc1	C to VP2 code translator
• vpas	Assembles compiler assembler output to object files
• vpld	Links object files into an Elate assembler file
• asm	Elate assembler (please refer to the Elate documentation)

4.1 Front End

To compile C source code, the Elate executable *vpcc* is invoked. This program is referred to as a *front end* because its task is to co-ordinate and simplify execution of the series of stages that make up the compilation. *vpcc* understands a set of command line arguments (see Chapter 8.1) and translates them, along with the names of C source files, into a series of invocations of the other components of the compiler.

Each file given on the command line must have a recognised suffix. For C this will be *.c* , and for C++ *.cxx , .cc , .cpp* or *.C*

4.2 Pre-processor

The pre-processor, *vpcpp*, is responsible for interpreting and acting upon all pre-processor directives in a given C source file. Pre-processor directives are those lines of a C program that begin with the hash (#) symbol. The output of the pre-processor is also a C source file, but one that lacks any pre-processor directives. Loosely speaking, the effect of running the pre-processor is to replace defined symbols with their values and to handle conditionally included sections of code. The pre-processor is a port of the standard GNU pre-processor.

4.3 C to Assembler Translator

vpcc1 converts a single C source file into a VP2 assembler file. This program assumes that if the C source file uses pre-processor directives or defined symbols, it has already been passed through the pre-processor. The translator is a customised version of the standard GNU C translator, re-targeted to generate Elate VP2 code and ported to run as an Elate program.

4.4 Object Files Generator

vpas processes a *.s* assembly file (as output by the compiler) to produce a *.o* object file suitable for the linker. This object file contains VP assembly instructions rather than binary code. It should be noted that *vpas* does not assemble VP or native code *.asm* files in the same way that *asm* does.

4.5 Linker

The Elate system relies on dynamic binding of procedures and does not provide for inter-tool binding of data (sharing of static or global data between tools). Therefore, the model of linking used by the compiler is quite different from the conventional one.

The linker operates on the *.o* object files output by *vpas*. The result of linking together a set of assembler files is that a single assembler file is processed by the Elate assembler, typically to create a complete program that is a single entry point tool. Global and static data for the program are then stored in the *Per Process Data Block (PPDB)*, a dynamically allocated block of memory. The exception is when an Elate tool is being generated which can then be called from within a VP2 coded program or application. Tools generated are treated the same as any other Elate tool and can be dynamically bound.

In addition to the compiled code, *vpld* generates prologue code, which does various kinds of initialisations when the program begins execution, and epilogue code that performs clean-up operations before the program terminates. By default, it also ensures that library support functions and definitions are included. A list of the command line options understood by *vpld* can be found in Chapter 8.2.

The linker also:

- Determines the total global data storage requirements of the program
- Establishes a mapping from global variable names to their offset in a global data storage area
- Maps variable names in accordance with their scope. A variable's scope might be global to the entire program or local to a particular file.
- Resolves all references to global data items such that they refer to the correct location in the global data storage area.
- Emits an initialisation routine that allocates the global data storage region and initialises it in accordance with the supplied initial values of the global variables.

The linker makes two passes over each input file. As each file is processed, the linker interprets the link-related pseudo ops and constructs a memory map of the PPDB. At the tail end of its output file, the linker emits a series of symbolic equates that define the mappings of global/static data into the PPDB. The symbols that are defined in this section are used within instructions that reference global data. For example, consider the following segment of C code:

```
int bar;

void foo(void)
{
      bar = 1;
}
```

This results in the compiler generating (amongst other things) the following code:

```
extern _bar,4,4,0

 [code removed]

cpy.i 1,r0
cpy.i r0,[gp+_bar]
```

The *.extern* pseudo op is fully explained in Chapter 8.3. Basically, it informs the linker that a global variable called *bar* has been declared and provides information about its size and alignment requirements. Notice that the variable is referred to in the generated code via the addressing mode [gp+_bar]. After its second pass is completed, the linker will have decided where the variable *bar* resides within the PPDB. If it decides to place *bar* at offset 8 in the PPDB, the linker would emit the following equate at the tail end of the linked assembler file:

```
_bar = 8
```

The generated code that references *bar* correctly identifies the location in the PPDB where bar resides. Pseudo ops such as the *.extern* in this example can also specify initial values for variables, and these initial value specifications are used by the linker to generate a series of instructions inside the *vp_init()* routine that initialises the values in the PPDB.

The operation of the linker is greatly simplified if it is generating a single tool without a control object. As tools have no global data, none of the processing for globals is required. The linker need only output the tool node definition and a *gos* into the first function encountered to set up proper entry into the tool.

4.5.1 Labels

Because input files are concatenated and the Elate assembler does not support variable/label scope in the way GCC expects, labels are mapped. To make identification of labels much easier, the compiler assists by delimiting them by special characters (currently exclamation marks). The compiler also emits pseudo ops to specify scope.

A label may have components appended to it to make it unique within a file and unique across all files. Additionally, a label that refers to an element of a structure has an encoding of its byte offset appended (*$$<offset>* or *$$m<offset>* for negative offsets).

The fact that a "$" cannot appear in a source code symbol but is valid (except as the first character) in an assembler label is taken advantage of and used to separate components.

Labels are case sensitive in C and C++. Nevertheless, because the assembler treats labels as case insensitive, the compiler adds a variable-length component to any label that is not completely lower case. This ensures that "foo", "Foo", and "FoO" in a C program are handled correctly.

Labels with global scope have a string prepended (currently the letter `x' followed by an underscore. This keeps them from conflicting with the assembler's reserved words (e.g., "tab"). Labels that are emitted by the linker as part of the program prologue or epilogue have the string "x_$$" prepended. This ensures that these labels will not interfere with any other kinds of labels.

4.6 Assembler

The Elate system assembler processes the output of the linker. The result is a single tool object, with the *F_MAIN* flag set for an executable tool.

4.7 Example

To help get a better perspective on how all these pieces fit together, consider the following command:

```
vpcc demo/example/c/main.c demo/example/c/proc.c -o main
```

This command compiles the two C source code files, *main.c* and *proc.c*, and produces as output *main.00*. This is done in the following steps:

Pre-processing
Vpcc runs the pre-processor (*vpcpp*) on each C file in turn, placing each result in an intermediate file. The name of the intermediate file is derived from the original source file's name. Normally, it is deleted by *vpcc* when it is no longer needed. Therefore, in this case where C files are being compiled, the pre-processed versions of the files will be stored in the temporary files *main.i* and *proc.i*.

Translation
The next phase of the compilation is the translation of the pre-processed C code into assembler source. *vpcc* runs the translator *(*vpcc1*)* on each of the intermediate pre-processed files. In this case, the result is two further intermediate files, *main.s* and *proc.s,* which contain Elate assembler source. After the intermediate pre-processed files have been translated, they are deleted.

Linking
Next, *vpcc* runs the linker (*vpld*), feeding it both of the intermediate files and the standard C library as arguments. The linker resolves all global data references, and produces a final assembler source file, *main.asm* Note that the file name used here is based on the argument given to the *-o* flag on the original command line. Once the intermediate assembler source files have been linked, they are deleted.

Assembly
Finally, *vpcc* runs the native Elate assembler on *main.asm*. The result is a main tool object *main.00*. If none of the steps encounters any errors, the program can be run.

4.7.1 Examples

```
vpcc demo/example/c/main.c
```

This command will cause the file *main.c* to be compiled, linked, and assembled. The resulting output will be in demo/example/c*/vpout.00*.

```
vpcc tmp/main.c
```

This command will cause the file *main.c* to be compiled, linked, and assembled. The resulting output will be in *tmp/vpout.00*.

```
vpcc -c tmp/main.c
```

This command will cause the file *main.c* to be compiled. The resulting assembler source file will be in *tmp/main.s*.

```
vpcc tmp/main.c -o tmp/bar
```

This command will cause the file *main.c* to be compiled, linked, and assembled. The resulting output will be found in *tmp/bar.00*.

```
vpcc tmp/main.c -ptest
```

This command will cause the file *main.c* to be compiled, linked, and assembled. The resulting output will be found in *test/vpout.00*.

4.7.2 Defaults

• Include path for the pre-processor:	`lang/cc/include`
• Library path for the linker:	`lang/cc/lib`
• Executable for the pre-processor:	`lang/cc/bin/vpcpp`
• Executable for the translator:	`lang/cc/bin/vpcc1`
• Executable for the linker:	`lang/cc/bin/vpld`
• Executable for the assembler:	`asm`
• The symbols taos and vpcode are defined by the pre-processor output file name:	`vpout.00`
• Run time support library:	`lang/cc/lib/libc.ism`

5. Compiler Configuration and Implementation Notes

This chapter describes those aspects of the Elate native port of the GNU C compiler that may be required by users of the compiler, particularly with respect to the interfacing of assembler code with the output of the compiler.

5.1 Data Formats

The following are the sizes that the C compiler associates with the primitive data types:

Type	Size (bits)
• char	8
• short	16
• int	32
• long	64
• float	32
• double	64
• long double	64
• pointer	32

By default, variables of type *char* are assumed to be signed.

5.2 Register Usage

VP registers are used as follows:

i0-i31	Integer register set
p0-p32	Pointer register set
l0-l31	Long register set
f0-f31	Float register set
d0-d31	Double register set
sp	Stack pointer
fp	Frame pointer (any available pointer register)
gp	Global data pointer

5.3 Function Call Conventions

The compiler adheres to the standard Elate system calling conventions, in that it defaults to a pass by value. Whether a structure is passed by value or by reference, the structure to be passed (a copy, or a pointer to the array itself) is placed on the stack to be dereferenced by the function. For *varargs/stdarg* functions, the unnamed arguments are placed on the stack.

5.3.1 Stack Usage

The stack is used for passing unnamed *varargs/stdarg* parameters and function parameters when the parameters overflow the number of available registers, and for storing local variables when the optimiser cannot allocate a register to store the variable for the lifetime of the function.

5.3.2 Debugger Support

The *-g* command line flag causes the compiler to emit stabs/dbx format debugging pseudo-ops.

5.3.3 Tracing Support

The compiler can generate code to trace the entry and the exit to every function. In addition, it can generate tracing output to indicate the call and the return from every Elate tool call. The *tracef()* tool is used to output the tracing information.

The *-mtrace-funcs* command line flag causes the function tracing output to be produced and the *-mtrace-tools* flag causes the tool call tracing output to be produced. The -*mtrace*-all flag enables all available kinds of tracing output.

5.3.4 Tool Generation Support

The compiler recognises that it is generating code for tools rather than normal C functions when it sees the *-felate-tool* command line flag. For backward compatibility, the *-ftaos-tool* command line flag is also observed.

5.3.5 Packed Alignment

By default, the compiler will align 64-bit data types (e.g., longs and doubles) to addresses that are evenly divisible by 8. The *-mpackalign* command line flag causes the compiler to align these types to addresses that are evenly divisible by 4 instead.

5.3.6 Error Messages

Any compiler error messages are saved in a file called *vpcc.log* in the same directory from which the compiler is run. If the file already exists, the previous contents are overwritten.

6. Troubleshooting

6.1 *exit()* and *taos_exit()*

When *exit()* is called by a program, run time support code may perform some clean-up routines before terminating. In the case of a C++ program, for example, destructors may be called. To terminate a program immediately, bypassing this clean-up, call *taos_exit()* instead.

6.2 *main()*

The *main()* function is called by the C runtime start up code with the assumption that it requires two parameters and returns an integer. Currently, if these two parameters are not declared for *main(),* the program will not run properly. Most include files make this declaration indirectly, but if not, it *must* be declared as follows:

```
    int main(int argc, char **argv)
```

6.3 Porting existing C code

To port existing applications to Elate, additional debugging/programming may be required. Much existing C source code assumes that the size of pointers, integers, and long integers are the same (usually 32 bits). This assumption is *invalid* in the Elate system. It is particularly important to be careful when passing a long parameter to a function. The compiler flag **-***Wconversion* can be helpful in locating these conversion problems.

Some C programs use integers and pointers in such a way that they are assumed to be practically identical. Whilst on many platforms this does not prevent a program from operating correctly, it may create problems in the Elate environment. Elate has specific files for pointers, integers, floats etc. and the compiler passes pointer parameters into pointer registers and integer (char, short, int) parameters into integer registers. A call to a function that was not previously declared with an ANSI-style function prototype or ANSI-style function parameter list may therefore not pass pointer parameters correctly. The following example illustrates the problem:

```
foo(arg)
unsigned short *arg;
{
...
}
...

foo(0);
```

The call to function *foo()* will pass the parameter as an integer rather than as a pointer and the value of *arg* will probably not be zero. When the compiler is able to detect these inconsistencies, it produces error messages. To avoid this problem, use an ANSI-style prototype for *foo()* or its parameter list:

```
int foo(unsigned short *arg);

int foo(unsigned short *arg)
{
...
}
```

Use of warning options, such as -*Wmissing-prototypes*, are encouraged.

Note also that old C (and ANSI C, in the absence of prototypes, and in variable-length argument lists) "widens" certain arguments when they are passed to functions. Of particular importance, a float is promoted to a double. Mixing the ANSI-style prototype declaration:

```
   extern int func(float);
```

with the old-style definition:

```
   int func(x) float x;
```

will generate incorrect code since the function will expect its argument to be a double.

6.4 Data Type Conversions

Under some very rare conditions, the compiler will generate incorrect code for some data type conversions. This is almost certainly a GCC bug that arises when it generates code for 64 bit long integers. The following is the only known example:

```
   f()
     {
       long l2;
       unsigned short us;
       unsigned long ul;
       short s2;

       ul = us = l2 = s2 = -1;
       /* ul is assigned an incorrect value. */
       return(ul);
     }
```

To work around this problem, rephrase the assignments as in the following example:

```
   l2 = s2 = -1;
   ul = us = l2;
```

It should also be noted that the Elate usage of 64 bit long integers might have an impact on the evaluation of certain benchmark figures. If other benchmarks make use of 32 bit integers any comparison between them and Elate benchmarks will not be valid. For further information on VP datatypes, please see the VP specification documentation.

6.5 *errno*

The linker handles the 4 byte global variable *errno*, which is declared in *errno.h* and is part of the Elate kernel. For this reason, programmers should not use a global variable called *errno* for any purpose other than its standard use for returning error codes from the system. If *errno* is declared to be a static or *sizeof(errno) != 4*, it is not handled specially.

7. Appendix - Writing in C++

It is possible for C++ code to call tools but the tool call declarations must be explicitly declared as C:

```
#ifdef C++
extern “C” {. . .
 qcall . . .
}
```

7.1 C++ to Assembler Translator

The C++ translator, which is called *vpcp1*, is similar to the C translator but processes C++ source code instead of C. It is a heavily customised version of the standard GNU C++ translator.

7.1.1 Exception Handling

It should be noted that C++ exception handling is not yet fully supported under inte**e**nt. In order to prevent the compiler generating exception handling code, it should be invoked as follows:

```
vpcc -fno-exceptions
```

7.1.2 Linker

Because the Elate environment relies on dynamic binding of procedures and does not provide for inter-tool binding of data, the model of linking used by the compiler is quite different from the conventional one. The linker operates on assembler files rather than binary object files. The result of linking together a set of assembler files is a single assembler file that is processed by the Elate assembler, typically to create a complete program that is a single entry point tool. The exception is when an Elate tool is being generated.

For further information on the linker (*vpld*) see Chapter 4.4.

8. Appendix – Compiler Options

8.1 *vpcc* Command Line Options

Below is a description of each of the command line options that *vpcc* understands. Each file given on the command line must have a recognised suffix; the suffix identifies the file type:

Suffix	**File Type**
.c	C source code
.cxx, .cc, .cpp, .C	C++ source code
.i	Preprocessor output of C source code
.ii	Preprocessor output of C++ source code
-	Standard input (Preprocessing only).
.h	Header file (Preprocessing only).
.s	Assembler source code (unlinked)
.asm	Assembler source code (linked)
.ism	Library assembler source code (unlinked). May use .s instead. Any other files are passed to the linker. These include:
.o	Object code (unlinked)
.a	Archive library, contains object code files.
.so	library description

Note that suffixes are recognised case sensitively (e.g., foo.Cxx and bar.CXX are not recognised, alpha.c is a C source code file, and baz.C is assumed to be a file containing C++ source code).

The dfa utility is called after assembly if -O is specified. The default action is to insert zaps and remove redundant tags and unreachable code. Only fatal errors are output unless -v –v (twice) is specified. If debugging is specified then a check is made for wrong parameters in qcalls, and only zapping is done - redundant tags are not removed.

The environment variable build.target may be set to the absolute name of a directory within which a build is to be made. Any compilation with this set must be within the directory or a subdirectory. Its chief effect is to ensure that any toolname generated is relative to the build directory rather than the root. It also sets the default include and loader search paths to first look relative to the build directory and then to the absolute location. The names of input files are not affected, if a path is absolute it will be relative to the root rather than the build target so it is normally best to specify relative names.

8.1.1 Options

Many options are standard GCC options and have the same meaning.

If a .asm file appears on the command line, most flags don't make sense since the file has already been linked and no other files can appear on the command line.

-aa	Normally any comments in the .s file are stripped before the final .asm file is output. This option causes vpas and vpld to keep any comments so they can be seen in the .o and .asm files.
-ansi	Disable any GNU features that might stop an ANSI conformant program compiling. The preprocessor defines __STRICT_ANSI__ and the ANSI standard include files will only include ANSI functions.
-asm	Suppress the assembling phase. Each source file is run through the preprocessor, translated, and linked with the others to produce vpout.asm or a .asm file named by the -o option.
-build <directory>	The directory is assumed to be the build target relative to which all absolute file names or directories are found or created. If not specified then the contents of the shell variable build.target are used instead. If neither are present then absolute paths are not altered. Default directories are searched relative to the

	build target directory first and then via the absolute name.
-B<path>	Use <path> as the base path for default include and library searches instead of the default.
-c	Suppress the linking phase. Each source file is run through the preprocessor, compiled, and assembled to produce an unlinked object file.. The resulting files have the same base name, but have suffix .o.
-cp<tool path>	Override the tool directory inferred from any -t or -o parameter. It is better to specify the full tool name using -t.
-cxx	Force C++ compilation. This flag is passed to the linker and is required when vpcc is given a set of .s files and therefore can't otherwise tell if it's dealing with C or C++.
-dD -dM -dN	Tell the preprocessor to output various details about macro definitions. Refer to the documentation on the standard GCC preprocessor.
-da -dv	Emit a VP2 specific RTL debugging dump with the file suffix .vp2. Refer to the documentation on the standard GCC debugging flags.
-D <symbol>[=<value>	Define preprocessor symbol. If no =<value> appears, then this has the effect of a #define <symbol> within each source file. If the =<value> appears, then the option has the effect of a #define <symbol> <value> within each source file.
-e <entry-point>	Use the specified procedure as the entry point for the linked tool. This will override any explicit or default entry point e.g. main. This must be used if -T is specified and there is more than one procedure as optimisation may reorder the procedures.
-E	Run only the preprocessor on the given C or C++ files. The output goes to standard output if -o is not specified. Standard input may be specified using -. If saving the output it is usual to name it the same as the input but with suffix .i (for C) or .ii (for C++), which can be compiled directly without preprocessing.
-f<name>	Enable special option <name>. These options are used for debugging the compiler and other unusual tasks. Please refer to Compiler Configuration and Implementation Notes for a complete description of the options.
-g	Generate directives for source code level debugging.
-gt	An alternative way to enable the -mtrace-funcs flag. This flag is also passed to the linker. Please refer to Compiler Configuration and Implementation Notes (included within the Elate build) for a complete description of the -mtrace-funcs flag.
-I<path>	Add the given directory <path> to the search path used by the preprocessor to locate include files.
-l<libname>	Include the shared library lib<libname>.so from the loader library list during the link stage.. If that does not exist the archive library lib<libname>.a is searched for instead.. (See the -L option below). The default entries include -lc, and if a C++ file is compiled or -cxx is specified the -lcp also. As such, to include a shared library, the following command line would be applicable: `vpcc -shared -o libFoo.so foo.o bar.o`
-L<directory>	Add the directory to the loader library list. The loader library list ends with /lang/cc/lib, or <basedir>/lib if -B is specified. For example, the command line for an application making use of a shared library would be: `vpcc -o test test.c -L/path.to/sharedlibs -lFoo` (This is where path.to/sharedlibs is the path to the new libFoo.so)
-m<name>	Enable special option <name>. These options are used for debugging the compiler and other unusual tasks. Please refer to Compiler Configuration and Implementation Notes for a complete description of the options.
-M<opt>	Cause the preprocessor to output dependency information. Use only with -E. -MM omits dependencies for system includes.
-n	Don't actually run any of the compiler components, but figure out what would need to be run. This is really only useful in combination with the -v flag.
-nostdinc	Don't include the standard include file paths when preprocessing.

-nostdinc++	Don't include the C++ specific include file paths when preprocessing.
-nostdlib	Don't include any standard libraries in the link.
-o<name>	Specify the base name for the final result of compilation. This flag has no effect if a .asm file is on the command line. Using the -o flag in a context where more than one output file would be generated by the command is not allowed.
-O -O0 -O1 -O2 -O3	Select the level of optimisation. Optimisation is disabled by default and explicitly using -O0. The lowest level of optimisation is enabled by -O (equivalent to -O1). The -O2 flag optimises for size and performance. The compilation will likely take more time to complete, especially if there are very large functions. The highest optimisation level is enabled by -O3. When optimisation has been enabled, the preprocessor symbol __OPTIMIZE__ is defined. Note: Zapping is currently disabled during compilation; the dfa utility is called after assembly to insert zaps if optimisation is specified.
-pedantic	Issue all the warnings required by ANSI standard C. This will warn about GNU extensions and some traditional C extensions that are not part of the standard.
-save-temps	Do not delete any intermediate files. Note Currently this option does not cause the intermediate files to be placed in reasonably named files in the current directory like GCC does. Use -v to see the names of the intermediate files generated.
-sysdev	The linker will produce a tool that may be used as a device driver in elate. The global data is accessed via the environment pointer __ep and all gos calls are turned into gose calls passing __ep. The -T option must not be specified.
-S	Finish after compiling to the unlinked assembler. The default is to name the output files the same as the input with the suffix replaced by .s.
-SS<num>	Set the minimum stack size of the tool to be <num>. The default size should almost always be large enough.
-T	Generate a single tool as the final output. In particular, this suppresses the output of a control object and alters the behaviour of the compiler and linker slightly. As a programmer's aid, it automatically defines the C preprocessor symbols __TAOS_TOOL__ and __ELATE_TOOL__. This causes the -ftaos-tool flag to be passed to the compiler.
-t<tool name>	Specify the full tool name to be used for the tool object. By default the tool name is derived from the output file specified by -o, the name is the terminal name less any .asm or .00 and the directory is the output file directory relative to the build target directory. The terminal name only may be specified if -cp is used to specify the directory.
-U<symbol>	Eliminate any preprocessor definition for <symbol> within each source file.
-v	Be verbose, displaying output indicating which programs are being executed and with what arguments. This flag can be repeated to increase the verbosity level.
-vp<num>	Generate version <num> of VP. At present, <num> must be 2, which is the default. This option is deprecated and may be removed.
-w	Suppress all warning messages.
-W<name>	All of the standard GCC warning flags are supported. For example, -Wall turns on all warning flags.
-Wa,<option>	Pass the <option> to vpas when producing unlinked object files.
-Wl,<option>	Pass the <option> to the linker vpld.

The option defaults for C are:

- The -fomit-frame-pointer flag is set. Use -fno-omit-frame-pointer to override it.
- Include path for the preprocessor: lang/cc/include
- Library path for the linker: lang/cc/lib
- Executable for the preprocessor: lang/cc/bin/vpcpp
- Executable for the translator: lang/cc/bin/vpcc1
- Executable for the unlinked assembler: app/stdio/vpas
- Executable for the linker: app/stdio/vpld
- Executable for the assembler: asm
- Executable for the dfa utility: dfa
- Output assembler name: vpout.asm
- Output tool name: vpout.00
- Run time support library: lang/cc/lib/libc.so
- The symbols __ELATE__ and __GNUC__ are predefined by the preprocessor.

The defaults for C++ are the same as for C except:

- Executable for the translator: lang/cc/bin/vpcp1
- Run time support libraries: lang/cc/lib/libcp.so and lang/cc/lib/libc.so
- The symbol __cplusplus is also defined for C++ source files.

Examples

```
vpcc tmp/main.c
```

This command will cause the file tmp/main.c to be compiled, linked, and assembled. The resulting output will be in vpout.00.

```
 vpcc -c tmp/main.c
```

This command will cause the file main.c to be compiled and assembled. The resulting unlinked object file will be in tmp/main.o.

```
 vpcc -o tmp/bar tmp/main.c
```

This command will cause the file main.c to be compiled, linked, and assembled. The resulting output will be found in tmp/bar.00.

```
 vpcc -cp test tmp/main.c
```

This command will cause the file main.c to be compiled, linked, and assembled. The resulting output will be found in test/vpout.00.

8.2 *vpld* Command Line Options

The following command line flags are recognised by *vpld*:

The linker is responsible for combining files of C/C++ object code and library descriptions into one VP assembler file that is ready to be assembled using asm. Each of the <file> arguments on the command line is an object file from the C/C++ assembler vpas. Programmers typically use vpcc to compile and link, rather than running vpld directly.

Options

-cp <ctlpath>	Specify the ctlprefix (and prefix for the toolname) to be <ctlpath>.
-cxx	Link using C++ conventions instead of those for C. This is typically unnecessary since the output of the C++ compiler is identified by a pseudo op within the file.
-g	Emit source code debugging information.
-gt	Generate tracing information in code generated by the linker.
-L <searchdir>	Add the directory <searchdir> to the list of directories searched for a shared library or archive as specified using -l <library>. The earlier entries are searched first.
-l <library>	Link using the shared library lib<library>.so or if that does not exist the archive lib<library>.a first found in the search path specified using -L <searchdir>.
-o <filename>	Send output to <filename> instead of the standard output.
-O <level>	Optimisation level 0..3 as for vpcc, Default is zero , -O with no parameer sets level to 1 This option has no effect currently but may in the future cause extra work to be done to improve the code output.
-SS <size>	Set the minimum stack size of the generated control object (default 262144 bytes) to <size>. While the linker insufficient stack may crash, so it is occasionally necessary to request a larger stack size.
-shared	Output a shared library, the output should refer to a .so file to create which will contain the shared library details for use in further links.
-split	Split by ent block into a directory. The main output will be a tool to initialise the data, and call the main entry point if it is not a shared library. The drectory will be called <toolname>.dir.
-sysdev	Output a tool that may be used as a device driver in elate. The global data is accessed via the environment pointer __ep and all gos calls are turned into gose calls paasing __ep.
-T	Output a single tool.
-t <toolname>	Set the name of the tool to <toolname>.
-v	Verbose. Repeat to increase amount of detail.
-version	Print the version of the linker and usage information and exit.
-vp<num>	Identify the version of VP as <num>. This defaults to 2.

8.3 *vpld* pseudo ops

8.3.1 Module Control

These pseudo ops must in general appear in a module header except for .linkonce, which may also appear in a module body. A complete object file must have .object at the start and .end at the end.

.als amount

The amount given by all of these pseudo ops is summed to help ensure that the program is given a reasonable stack allocation.

.constructor initializer

[C++ only] The initializer is added to the list of functions to be called at run time just before the main program is entered. They are called in reverse order to that in which the pseudo ops appear.

.cplusplus

This indicates that the file was generated by the C++ translator and the program requires appropriate libraries, initialization, and termination.

.destructor terminator

[C++ only] The terminator function is added to a list of functions to be called when the program exits. The destructor functions are called in the same order as declared by the pseudo ops.

.end

Every object file must end with this pseudo-op. It is used as a check when reading archive members to avoid reading beyond the end. See .module_start for an example of use.

.entry name, params

This names the entry point after any initialisation has been done and specifies the parameters. This allows main to omit the parameters or return void. One and only one entry point may be defined, except if a pure elate tool is being produced - see toolentry. The entry point name should also have .export specified to make it public. The standard parameters are p0 i0: i0 corresponding to

int main (int argc, char *argv[])

This can be defined as the entry point by:

```
          .export !.main!,0
          .entry !.main!,p0 i0:i0

          !.main!:
              .ent p0 i0:i0
```

.***header source-line***

The souce-line will be put by at the start of the assembler output file after any standard includes. The usual use would be to include additional assembler .include directives at the start. No quotes are necessary around source-line, it is simply the rest of the line.

.ident "string"

The string can be used to put identification information for the source file or compiler into the output tool. For example:

```
.ident "$GCC: (GNU) 2.8.1 $"
```

would identify the compiler used. The '$' characters at either end are necessary if the string is to be recognised and output by the ident tool.

.library

This module is optional, it is only included if required by a non-weak external.

.link_off
.link_on

The lines between these are copied by vpld exactly into the output assembler file except names within !...! have name mangling performed on them.

.linkonce link-key [,check]
.linkonce_end

During linking only one module's linkonce blocks for each link-key will be linked.

The declaration and initialisation or code for a symbol must either not be in a .linkonce block or else must share the same link-key. No locals declared in a .linkonce may be referenced outside. An .export or .extern within a linkonce block may be referenced outside the block and will be treated as an .import or an .extern without initialisation if the linkonce block is not included.

The link-key is not a symbol name - it does not conflict with other names and may be the same as a function or data area name. The check is an alphanumeric string, which if specified must the same for each particular key. It is enough to specify check once only in the header to get full checking as early as possible.

The functions and data declared in a linkonce block need not be marked with .weak to be removed, it is enough that they be part of a duplicate linkonce block.

If used a .linkonce will normally appear once in the module headers enclosing the linkonce declarations, and once in the module body enclosing the actual instances or initialisations.

.module_start count
.module_body count
.module_end

There may be more than one module in an object file. The declarations are grouped at the start and the bodies put at the end as in:

```
        .object
        .module_start 1
        .export !.abc!,i0:i0,1
        .module_start 2
        .extern !.def!,8,4,0,0
        .module_body 1
        !.abc!:
            .ent i0:i0
            ret
        .module_end
        .end
```

The count starts at 1 in each file and connects a body to its declaration. The linker reads the declarations in its first pass and the module bodies in the second pass.

The following pseudo ops must only occur in a .module_start block.

- .als .cplusplus.main.toolentry
- Declarations pseudo ops
- Interface pseudo ops

The following pseudo ops may occur in either a .module_start or .module_body block.

- .linkonce .linkonce_end
- .file

All other pseudo ops may only occur in a .module_body block.

.object

The magic string ".object\n" must be at the start of every .o or .so file or it will be rejected by the linker. See .module_start for an example.

.toolentry [name]

A go to the function name is put at the start of the output. This pseudo op may only be used once per file when linking a pure elate tool.., i.e. one without any initialisation or static data.

8.3.2 Declarations

These pseudo-ops may only occur in the .module_start part of an object file.

No locals declared in a .linkonce may be referenced outside. Any globals are treated as if they are

The params specification in the pseudo-ops below is used for cross-check purposes and is a unique compressed form of an ent only mentioning the highest register and with the registers in the order i,l,f,d,p. Thus, i2p0:- means three integer and one pointer register on input and none on output.

The xrefs in the declarations counts the number of references to the symbol other than in diagnostic records or calls. This is used to help arrange the PPDB for more efficient execution.

The calls in the declarations counts the number of calls using gos to the symbol.

.check name, check

The check is an alphanumeric string. Two instances of .check which specify the same name must also specify the same check.

.equate equate-name, basename, offset, xrefs

This declares that equate-name is a new name for the memory address that is the sum of symbol basename and offset. offset may be positive or negative. If basename is a function then so is equate-name and offset must be zero.

.extern name, size, alignment, initflag [,initstr], xrefs

This declares name to be a variable of length size bytes that requires its first byte to be on an address evenly divisible by alignment (if alignment is zero, there is no alignment requirement). The initflag is as follows:

0
No initialisation is supplied. A zeroed space in the PPDB will be allocated if no other .extern supplies an initialisation. The name is searched for in libraries if necessary but it is not an error if it is not found.
1
The variable's PPDB is initialised by a block copy instruction at start up time. The initial value is supplied in a .data block.
2
initstr is an assembler instruction that moves some value into p0. The value in the register is then moved into name. This is used to set up

```
                FILE *stdin, *stdout, *stderr;
```

3

The variable is a read-only const. vpas will have checked that any addresses are to other local read-only areas or labels.
4
initstr is the name of an offset from gp in the PROC struct. This is used to set up an external to errno.
5

initstr is used directly for name. This may be used to substitute an absolute number.
6
initstr is an assembler instruction that moves some value into p0. This is the address of name (initdlag 2 sets up a 4 byte pointer to the area).
7
No initialisation is supplied but the name is not searched for in libraries. A zeroed space is allocated if no other initialisation is found.

If initflag is orr'ed with 32 then it is a common area. There is no visible effect except that initflag 32|0 is equivalent to initflag 7.

If there is more than one .extern for the same symbol then the size and alignment of the symbol is the maximum of those specified.

An .extern in a duplicate .linkonce is treated as if its initflag is zero.

If initflag is non-zero in more than one .extern then a warning is output unless the symbol is marked .weak and only the first initialization encountered will be applied.

.export name, params, calls

This says name is a global function that is defined in this module.

The params and calls are as described in Declarations.

An .export in a duplicate .linkonce is treated like an .import.

.fdesc name, xrefs

The function name is referenced to get the address of the function. This may require an entry in the data area to hold the value.

.function name, [params], calls

The name specified is that of a local function.

The params and calls are as described in Declarations.

.import name, [params], calls

This says name is a global function that is not defined in this module, i.e it is an external function that needs to be linked up to.

The params and calls are as described in Declarations.

.static name, size, alignment, initflag, xrefs

This declares name to be a variable of length size bytes, that requires its first byte to be on an address evenly divisible by alignment (if alignment is zero, there is no alignment requirement). initflag may be 0,1 or 3 and the effect is as described for .extern. There is no implicit address ordering between successive .static variables.

.***syspath directory-path***

The directory-path is the path used for finding the tools specified in any following .system pseudo ops. This applies till a subsequent .syspath or the end of this module.

.system name, [params|], calls [, pathnoep, pathep]

The name refers to a system tool in the directory given in the last .syspath. It is treated as a .weak.export defined in this module so a user defined version takes precedence. They require no environment set up when called. This is used to set up the initial C library.

The params and calls are as described in Declarations.

The pathnoep and pathep specify the full path name to be used for the tool depending on if environment pointers are being used. They may be used to for instance supply two different versions of bsearch or qsort depending on the form of procedure pointer being handed over.

.weak name

The symbol name is marked with the weak property. This means that

If name is an .import (unsatisfied external reference) then it is not an error if name is not fixed up. It will not be searched for in a library but will be fixed up if found. A non-weak unsatisfied reference overrides this behaviour.

If name is an .export (global functions) then there may be more than one copy of name. A version which is not marked as weak takes precedence otherwise the first encountered is chosen.

If name is an .extern (global data) then more than one initialisation may be supplied and the first encountered will be chosen, a non-weak version takes precedence.

When linked in a shared library a weak .export or .extern will be referenced via an indirection so they can be linked instead to a version in the caller. In other cases references to extern data that are satisfied in a shared library will be fixed up to by an indirection in the main program.

8.3.3 Data Initialisation Pseudo Ops

Note that the data in .rodata blocks is considered preformatted whereas the initialisations in a .data block may be reordered.

.address name, offset ,dst-offset

This declares a data word containing the address of name+offset within the initialisation for a destination symbol in a .data block. The dst-offset states the displacement after the destination symbol where the address goes.

```
        .data
            .....
        !abc!:
           dc.i 1234
           .address !def!,12,4
```

This says !abc! is initialised to {1234, &def+12}

.align

This aligns the current address in a .data block to the next 8 byte boundary. The start of each symbol in a .datablock is determined by the alignment in its corresponding .static or .extern and a .align before the symbol is ignored.

.data

This introduces a block of static initialisations, that is data that sets the initial value of data declared with .static or .extern. This is ended by a .text or .rodata.

.retool ref-type,"tool path",dst-offset

This is similar to .address except the pointer is initialised as in

dc.p arh tool-path

ref-type is a tool reference type, 0 means a normal tool reference.

.rodata

This introduces a block of code related data, that is data that is read-only and not described by .static or .extern statements. This will typically contain jump tables for computed goto's. The only addresses allowed are to local .text or .data blocks, these are set up using dc.p !tag! rather than .address.

8.3.4 Code Pseudo Ops

.debug statement

The statement following a .debug is suppressed unless debugging is enabled. This enables vpas. to conditionally remove unnecessary statements, e.g. a go to an immediately following label.

.ent params
.entl params
.entd
.entdr
.entih

These are the entry points to functions and correspond directly to the VP assembler commands. The .ent and .entl will be converted into ente and entle commands if an environment pointer is required. Nothing special happens with the other forms but code after them will not be able to use __ep.

.***gos target, params***

This corresponds fairly directly to the VP assembler command gos. It may also become a gose or qcall.

If target is a pointer or expression and environment pointers are in use then target will be the address of a procedure descriptor. The following code will be emitted:

gose [target], [target+4], params

When target is a symbol, the code emitted depends on the symbol. For a target within a main application a gos is emitted, and when linking inside a shared library the following is emitted:

gose target, __ep, params

When linking to a .system symbol a qcall will be used.

When linking to a shared library or linking to an external from a shared library normally a gose will be used with a locally declared .fdesc.

Other forms e.g. ncall or a version using qcalle may be supported in the future

.link off
.link on
These two pseudo ops are used to bracket text that is copied, without any change whatsoever, to the output stream. This capability might be used, for example, to include hand-crafted code that requires no linking and contains no other pseudo ops. These directives may not be nested. The text is considered as in the code area.

.text

This starts a code block. The code blocks and .link blocks are concatenated in the output. It is possible that some or all initialisations from .data or .data blocks will also be inserted where they appear so the code generation should not assume they are necessarily out of line.

8.3.5 Diagnostics Pseudo Ops

.file ``filename"

This identifies the source code file corresponding to this assembler file. Used by the linker when producing error messages and translated to the assembler's ___file macro.

.line number

This identifies the source code line number corresponding to this point in the assembler file. It is translated to the assembler's ___line macro.

.stabs "string", type, access, desc, value
.stabn type, 0, desc, symbol
.stabd type, 0, desc

These DBX/stabs debugging pseudo ops are translated to the assembler's ___stabs, ___stabn, and ___stabd macros. These will become a compatibility feature when DWARF format is supported instead.

Both vpas and vpld inspect and update the stabs. The string field may specify a global data item and if so, the value field will be set by vpas to the global symbol. The access field may be updated to say how the symbol is to be located by a debugger. The details of type definitions in the string field are not inspected.

If the string field end with \\ it is assumed to be continued by the next stabs record. The "<name>:<type char> part must fit into the first stabs record when it is split.

Some information from .stabs is kept from a duplicate .linkonce block, in particular debug information for globals and type information - this is done by adjusting stabs records to replace local symbol names by a single space and change the .stabs to be that for a type definition.

The access field in the output ___stabs macros describes how value may be used to get the location of a symbol.

0	Standard action - vpld has not set the access specially.
1	value is an absolute value or location.
2	value is a tag in the code area for code or constant data.
3	value is a displacement in the data area
4	value is a displacement into the data area where there is a pointer to the required location
5	value is a displacement into the data area where there is a pointer to a function descriptor for a code location.

8.3.6 Interface Pseudo Ops

These pseudo-ops may only occur in the .module_start part of an object file.

.interface

This module describes the desired appearance of the output after linking. Thus, any .export describes a name that must be visible in the output. Any .weak applies to the names after linking. No actual code or data blocks should be supplied. By default any .global, .export, and .extern anywhere in the link will be visible unless .suppress is used.

.require library

This is used in a shared library description to say that it uses another shared library called library. If the using library is included in a link then library is also automatically included. Note that a name declared in library is not automatically available in the description of the using library unless it has a defining instance.

.retain name

This ensures name is kept visible in an output library even if .suppress has been specified. The name must be present in the linked module. No other properties are specified.

.shared library

This says the current module is a description of a shared library and that library is the tool name for the library.

.suppress

No visible names will be retained in the output library unless they are explicitly specified in a .library module.

.version version, revision

This sets the version and revision level when a shared library is output. When a shared library is loaded at run time a check is made that it has the same version and a greater or equal revision to the one the module expects.

8.3.7 Compatibility Pseudo Ops

The following pseudo-ops will not occur unless vpas uses compatibility pseudo ops.

.setaddr src-name, src-offset, dst-name, dst-offset

This pseudo op is used to initialise variables in the PPDB that point to absolute addresses. The address of src-name plus src-offset is copied into dst-name plus dst-offset.

.setref ref-type, "tool path", dst-name, dst-offset,

This pseudo op is used to initialise a variable in the PPDB that point to a function. The address of the tool is copied info dst-name plus dst-offset. This variation of the pseudo op is used to initialise a variable that is a pointer to a function.

9. Appendix – Elate Calling Mechanisms

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

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.

A tool may be :

- **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 (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 must be loaded and bound before it is available for use. If this is the case, on exit from the tool it may be abandoned by the kernel at any future time to free local memory space if required. Since tools are re-locatable, in the sense that they have absolutely no address dependent coding, it must be assumed that the kernel may relocate any tool at any time if no caller is referencing the tool non-virtually. Relocation, here refers to relocation within the same local memory space, not from processor to processor.

- **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.

e.g.	*qcall demo/mytool,(p0:i0)*	*; non-virtual*
	qcall demo/mytool,(p0:i0),VIRTUAL	*; virtual*
	qcall demo/mytool,(p0:i0),VIRTUAL+FIXUP	*; semi-virtual*

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)
```

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 '~' (tilde). This is an optimisation that tells Elate not to bother returning parameters on this call, thereby increasing speed and reducing stack overhead.

e.g. *qcall demo/example/mytool,(i0 : i~)*