Sysgen Reference Manual

Version 1.21

1. Overview

The entirety of the int**e**nt® and Elate® platforms are tailored around the needs of the application they are running, incorporating the precise minimum of tools needed to successfully run that application and no more. To facilitate the sophisticated process of determining what tools are required to successfully run a program, the sysgen utility (system image generation) is used to generate a bootable system image from a collection of tools and a specification of the user's requirements.

To go into more detail, sysgen takes an 'instruction file,' and follows the instructions contained within that file, to then make one or more system images. The **instruction file** specifies which tools to include, which device objects to start automatically, which processes to start automatically and some other attributes of the application. sysgen does not include a tool in the image unless it is specified in the instruction file or is required by a tool specified in the instruction file. This means that the image is the absolute minimum required to run the application as defined by the instruction file. All necessary tools are then loaded, translated, bound and written to an image file (or files).

2. Sysgen Options

2.1 Command Line Options

The options to sysgen may be specified individually (for example "-m -n -s") or grouped (for example, "-mns"). The exception to this is that where an option takes a parameter, the remainder of that command-line argument (if any) is taken to be the parameter to the option (if the remainder of the argument is empty, then the next command-line argument is used as the parameter to the option).

```
sysgen [options] <filename> [<filename> ...]
```

If <filename> is not specified, sysgen reads stdin for the name of the sysgen file to process. Multiple files can be specified at *stdin*.

Command Line Options:

-q	Quiet: Only print text if it is an error report.
-m	Map: Generate a map file, containing a list of the components written to the output image file, and their locations.
-v	Verbose: Write lots of diagnostic output.
-i	Include tools that are referenced as virtual or virtual+fixup into image.
-l	List: Print list of tools included/excluded in image.
-b	Enable embedding (currently off by default - though this may change in future releases).
-c	Change VIRTUAL+FIXUP *qcalls* into VIRTUAL *qcalls*.
-f	Option to allow multiple search paths to be specified.
-N	Change VIRTUAL qcalls into normal qcalls. Note that this modifies VP tools before translation. sysgen cannot modify the types of qcalls in native code.
-n	Change VIRTUAL+FIXUP *qcalls* into normal *qcalls*. Note that this modifies VP tools before translation. sysgen cannot modify the types of qcalls in native code.
-s	Strip Resource names from tools where possible.
-p	Only parse input files, do not produce any output or do any dependency searching. This is useful for checking syntax of instruction files.
-e	Emit toolnames before tool headers. This is useful for debugging purposes when toolname compression is enabled.
-D<NAME>[=<VALUE>]	This defines the pre-processor variable specified by NAME to have the value specified by VALUE. If VALUE is not specified, NAME is defined as a pre-processor variable with no value.
-t	Generate a symbol file for Elate debugger (see appendix one).
-f<path>	Option to allow multiple search paths to be specified. Each "-f" option specifies one additional path to search. This option may be specified multiple times. Paths are searched in the order in which they appear on the command-line. Note that if the "-f" option is not specified at all, the current directory is used as the default. If the *"-f"* option is specified at all, then the current directory must be explicitly specified using a "-f" option if it is desired in the search path.

2.2 Sysgen Instructions

The instruction file contains a list of tools, device driver objects and processes to be created at boot-time, tools to be called at boot-time, named-data-areas to be created, system parameters (such as scheduling parameters, memory layout specification and memory allocation methods), translator to use and valid tool extensions to be included in the output image file.

There are no restrictions on whitespace characters (including new lines) within or around instructions. Multiple instructions may be on the same line. If numbers are split across lines, the new-line character must be escaped using a backspace character. Strings may be split across multiple lines, in which case the resulting string will contain new-line characters unless the new-line characters in the instruction file are escaped.

Instruction files can be included during the processing of other instruction files. This allows other files to be created for specifying the instructions necessary to run an application, or to run on a particular processor type of platform. If the format of the instruction file is incorrect, then sysgen will generate a syntax error message.

Some sysgen instructions have global scope (such as *.trans* and *.toolext*), while others have a local scope (applying only to their containing memory object or scheduler parameter block). Multiple memory objects may be defined in a single file.

The following instructions are available in sysgen:

Instruction	**Description**
.call	Specify tool to be called at runtime
.emit_dependencies	Emit tool dependencies
.jtrans	JBC Translation specification
.map_region	Region mapping directive
.memory	Named memory area specification
.nda	Emit named data area
.obj	Specify device object to be started
.pad	Specify that padding bytes should be emitted to set the current memory area offset or address as specified
.path	Add Directory to Sysgen's Search Path
.priority	Specify scheduling parameters
.pii_opentool	Specify tool to be used to handle PII opentool requests
.scheduler	Specify scheduling parameters
.spawn	Specify process to be started at runtime
.strip	Tool stripping directive
.subblock	Specify a set of tools and NDAs to be emitted at the specified memory object offset or address. Other objects can be fitted around these
.system	Specify a command to be run at the time the sysgen file is parsed
.toolext	Tool extension specification
.trans	Translator specification

3. Syntactic Descriptions

3.1 Translator Specification - .trans

The *.trans* instruction specifies which translator should be used for converting VP tools into native tools. All tools placed in the output file must be native binary code, since the image has to be executable. This enables the images generated to be completely standalone, and able to run without a VP to native translator. As a consequence of this, if any VP tools are specified or indirectly referenced as a dependency, a translator must be specified, so that they can be converted to native code when sysgen executes.

The format of the instruction is

```
.trans ( <name> [, <flags>] [, emit] [, static|dynamic]);
```

NAME is the name of the translator to be used, flags is a numeric field indicating the flags word to be passed to the translator when it is called (both during sysgen operation and at run-time), and emit specifies that the translator tool should be emitted into the image, so it can be used at run time.

Translators and flags may be specified separately for static (sysgen-time) and dynamic (run-time) translation, by making use of the static and dynamic keywords. If static is specified, the translator and flags specified will be used for translation by sysgen. If dynamic is specified, the translator and flags specified will be used for translation at run-time for dynamically loaded tools.

Note that specifying static and dynamic translators is the only time that multiple *.trans* instructions are permissible.

If only one .trans instruction is present, then the presence or absence of the keywords static and dynamic is ignored, and the translator is used for both static and dynamic translation.

For example:

```
.trans ("sys/tr/i386", 0x00000002,emit);
```

The translator name specified should be the tool name of the translator. This should not include the name of the extension (.03 or similar). Translators are usually in the *"sys/tr/"* directory and generally have the same name as the processor whose native code they translate to, this being the i386 in the case of the example above.

3.2 JBC Translation specification - .jtrans

The *.jtrans* instruction specifies which translator should be used for converting Java™ bytecode classes into VP tools. After translation to VP, the classes will be translated into native code by the translator specified in the *.trans* instruction in exactly the same way as with normal Elate VP tools.

The format of the instruction is

```
.jtrans ( <name> [ , <flags> ] [ , emit ] [, static | dynamic]);
```

where "NAME" is the tool name of the jcode translator to be used, which should not include the extension (.03 or similar). Jcode translators are usually in the *sys/j/tr/* directory. Emit specifies that the translator tool is emitted into the sysgen image, so it can be used at run time. The "flags" parameter is simply a numeric field specifying the flag value passed to the Java™ Byte code translator when it is called (either at sysgen time or run time) to translate classes. Note that, at run time, the bit of the flags word that specifies whether to verify classes as they are translated is ignored in favour of the setting derived from the options of the jcode command used to start the Java^{TM} virtual machine.

If static is specified, the specified translator and flags is only used for translation at sysgen-time, allowing the translator and flags for run-time to be specified by using the dynamic keyword on another *.jtrans* instruction.

Note that the *.jtrans* instruction is only allowed to be specified more than once if it is being specified using the static and dynamic keywords.

3.3 Tool Extension Specification - .toolext

The *.toolext* instruction specifies which tool extensions should be used in dependency searching by sysgen. In order to perform its dependency search operation in the same manner the Elate kernel would when loading a tool, sysgen must search for native tools before VP tools. The order in which native tools are searched for is specified by using the **.***toolext* instruction.

The format of the instruction is

```
 .toolext ("<ext>" [, static|dynamic]);
```

This is where <ext> is the two digit extension (without the period) to search for. For example, *.toolext ("03"),* in the case of i386 processors.

Multiple tool extensions may be specified but each must be in a separate *.toolext* instruction. Tool extensions are used in the order in which they are specified, so that if sysgen fails to find a tool with the specified extension it will go to the next tool listed and attempt to load that. If none of the specified tool extensions can be found sysgen will attempt to load a VP tool and translate it using the specified translator.

Tool extensions may be specified separately for static (sysgen-time) and dynamic (run-time) loading of tools, by making use of the static and dynamic keywords. If static is specified, the tool extension specified will be used only by sysgen, and will not be passed through to the kernel for use in dynamic loading of tools. If dynamic is specified, the tool extension specified will not be used by sysgen, but will be used by the kernel for dynamically loaded tools. If neither static nor dynamic is specified, the tool extension will be used in both situations.

Example

```
.toolext ("03");
.toolext ("04");
.toolext ("14",static);
.toolext ("15",static);
.toolext ("24",dynamic);
```

n this example, the tool extensions *03*, *04*, *14* and *15* will be used by for static tool loading (tools put into the image by sysgen), and the tool extensions *03*, *04* and *24* will be used for dynamically loaded tools.

3.4 Named Memory Area Specification - .memory

The *.memory* instruction allows the user to define memory areas which can be used for emitting code and data, and also for run-time allocation.

The format of the instruction is

```
.memory ( name="<name>" , attr="<attr>"  [ , image="<image_name>" ] [
, manager="<new_tool>" ]  [ , base=<base_addr> ] [ , size=<size> ] );
```

- <name> is the name of the memory area.
- <attr> is a string describing the attributes of the memory area, which may contain the characters 'r' for read, 'w' for write, and 'x' for execute.
- <image_name> is the name of the image file to be created by sysgen for this memory area; if this is not specified, any initialised data for this memory area will be emitted into another memory area (the memory area into which the data is emitted can be controlled by specifying that the special sysgen NDA "SYSGEN_RELOCATES" should be emitted in a particular memory area). This will then be relocated into the correct location at boot time. This cannot be done if the memory area is a ROM area.
- <new_tool> is the name of the *_new* tool for the memory allocator class to use; if this is not specified, then the memory area is for use only at sysgen time, and is not usable for runtime memory allocation. This feature is often used for ROMs, where memory allocation cannot occur.
- <base_addr> and <size> are the base address and size of the memory area; specified in decimal or hexadecimal (with a preceding 0x). These may be omitted if they are to be filled in at boot time by PII specific software. If the base address is not specified, tools within the image will not be completely fixed up, and will require a fixup pass at boot time. This is usually part of the portable kernel initialisation, and will not require user intervention.

Objects such as tools, ndas, atoms, etc are emitted into memory areas. The instructions representing the objects to be emitted into the memory area should follow the memory area declaration and should be enclosed by a pair of braces *{..}.*

3.5 Region Mapping Directive - .map_region

Sysgen provides the facility to map particular usages of memory allocation into specified areas. For example, a memory area can be defined to be the default object for allocating stack memory. This can be overridden when processes are created, but if this has not been done, they will use the default region mappings set up during sysgen.

The *map_region* instruction sets attributes for the memory area containing the instruction. These attributes control the usage of the memory area, during sysgen operation and run-time. The format of the instruction is as follows:

```
.map_region (<type>);
```

<type> is one of the following values:

3.5.1 map_region (defaultstack);

The memory area containing this instruction will be used as the default location for allocating stack memory at runtime.

3.5.2 map_region (defaultdata);

The memory area containing this instruction will be used as the default location for allocating system data memory at run-time (using *sys/kn/mem/allocdef*). In addition, this memory object will
be set as the data memory object for all processes started by the *init* process, and will be inherited by all sub-processes unless another memory object is explicitly specified.

3.5.3 map_region (system);

The memory area containing this instruction will be used as the system data memory allocator at runtime.

3.5.4 map_region (defaultcode);

The memory area containing this instruction will be the memory object into which dynamically loaded tools are placed.

3.5.5 map_region (defaultmail);

The memory area containing this instruction is the memory area used for mail message allocation.

3.5.6 map_region (staticdata);

The memory area containing this instruction is used by sysgen to emit kernel data that needs to be writable (such as NDA lists, etc). This memory area must have a base address known at sysgen-time if the memory area into which the code is being emitted is at a known base address, and the code area is read-only.

3.5.7 map_region (staticdata_ro);

The memory area containing this instruction is used by sysgen to emit kernel data that does not need to be writable at run-time.

3.6 Tool Stripping Directive - .strip

The *.strip* instruction emits the specified tool into the image, with its tool headers and any trailing resource strings stripped off.

The syntax for the instruction is:

```
.strip ("<toolname>")
```

where toolname is the filename of the tool to be emitted. Unlike the *-s* option, which strips resources but leaves the tool header such that the tool can be called by a VIRTUAL or VIRTUAL+FIXUP call, the .strip instruction emits the tool without either header or resources.

3.7 Emit Named Data Area - .nda

Any program that has the correct name for a named data area can obtain a pointer to the required corresponding data, so that data can be shared between processes.

The *.nda* instruction facilitates this by allowing named data areas to be emitted into the sysgen image (and relocated if necessary). Tools within the image, and dynamically loaded tools can bind to these named data areas, or look them up dynamically at runtime.

```
.nda ("<name>");
.nda ("<name>", <size> );
.nda ("<name>", "<filename>" [, <size>] );
.nda ("<name>", contents = "<asm-source>" [, <size>] );
```

The first form of the instruction is only applicable to certain special NDAs which are automatically created by sysgen. Using this form of the instruction allows the user to specify where these special NDAs should be emitted. For further information please see the section on Special Named Data Areas.

The second and third form of the instruction is more useful for user data. The name of the NDA must be specified first. Any other string specified in the instruction is taken to be a file to use for initialising the NDA. Any number is taken to be the required size of the NDA. If only the filename is specified, the NDA will become the size of the file. If only the size is specified, the NDA will be the specified size, and will contain uninitialised data. If both are specified, the data from the file will be used to initialise the NDA, but the specified size will be used. If the specified size is larger than the size of the file, the entire file will be emitted into the NDA, and the remainder will be padded with zero bytes. If the specified size is smaller than the size of the file, only the specified amount of the file will be emitted to the NDA.

The fourth form of the instruction allows the user to specify assembler source code which is included as the contents of the NDA. Only data directives may be used here - code is not allowed. However, it is permissible to use assembler variables (simply by using their names exactly as in normal assembler source code), and sysgen variables (in the normal way, by using the $(NAME) form). The specified assembler source is simply assembled without any tool headers or other data being appended, and the resulting data is emitted into the NDA. If the size is specified, it affects the resulting NDA in exactly the same way as when a filename is specified. The file *'lang/asm/include/tao.inc'* is included by default for the assembly, but extra files may be included if required in the normal manner for assembler source code.

The assembler source code may be split across several lines if desired. For example:

```
.nda("TEMP",contents="
        .include 'some/include/file.inc'
        dc.i $12345678
");
```

3.8 Emit Tool Dependencies - .emit_dependencies

This instruction allows the user to specify the point in the image at which tool dependencies are emitted. If this instruction is not specified, all tool dependencies will be emitted at the end of the image. No parameters need to be specified as sysgen will automatically locate the relevant dependencies.

3.9 Specify Tool to be Called at Runtime - .call

This instruction allows the user to specify a tool to be called during the system boot sequence. This is usually used for kernel and PII initialisation purposes.

The syntax for the instruction is

```
    .call ("<toolname>");
```

The tool is called with no parameters and no return parameters are expected.

3.10 Specify Process to be Started at Runtime - .spawn

This instruction allows the user to specify processes to be started when the system boots.

The format of the instruction is as follows:

```
.spawn (tool="<toolname>" [, stack="<stackmemobj>"]
      [, data="<datamemobj>"] [, priority=<priority>]
      [, period=<period>] [, bcet=<bcet>] [, wcet=<wcet>]
      [, cmdline="<string>"] [, wait] [, local] [, stdin="<inname>"]
      [, stdout="<outname>"] [, stderr="<errname>"] );
```

where:

- <toolname> is the toolname (not the filename, i.e. no extension) of the process's main tool.

- ♦ <stackmemobj> and <datamemobj> are the names of the memory objects to be used for stack and data allocation respectively in the process. These only need to be specified if they are different from the default values.
- ♦ <priority> is the process's priority, defaulting to 128.
- ♦ <period> is the process's period, if it is periodic. <bcet> and <wcet> are the best case and worst case execution times respectively of the process. These three parameters are only relevant if the scheduling policy for the priority level at which the process is running makes use of them.
- ♦ <cmdstring> is the command line string which will be passed to the program being started. This will be split into individual arguments on whitespace boundaries.
- ♦ <inname> is the name of the file or device which will be used for the standard input file descriptor of the program.
- ♦ <outname> is the name of the file or device which will be used for the standard output file descriptor of the program.
- ♦ <errname> is the name of the file or device which will be used for the standard error file descriptor of the program.
- ♦ If <local> is specified, the program will be started with a local PID number (not network-unique).

A process which is created using .spawn will not necessarily execute immediately. Since the system *init* process runs at priority 16, normal processes will not run until it is finished. Processes started with a higher priority will run immediately. In addition if *wait* is specified the i*nit* process will wait for the child to finish before continuing.

3.11 Specify Device Driver to be Started at Runtime: .obj

This instruction allows the user to specify ncallable device objects to be started when the system boots.

The format of the instruction is as follows:

```
.obj( newtool="<newtool>", mountstr="<mountstr>", cmdline="<cmdline>"
);
```

where:

- <newtool> is the name of the *_new* tool for the device class.
- <mountstr> is the name which the device will have when it is mounted.
- <cmdline> is the command line string which will be passed to the device's *init* method.

3.12 Specify scheduling parameters: .scheduler

This instruction is specified outside of any memory area, and has no parameters. It simply marks the beginning of the scheduler parameter block. It should be immediately followed by an *"{"* open brace. The scheduler parameter block should be terminated by a *"}"* close brace. The format is:

```
.scheduler {
<scheduler parameters>
}
```

```
.scheduler {
    .priority (range=0-15, timeslice=false,
policy="sys/kn/sched/fixed");
    .priority (range=16-255, timeslice=true,
policy="sys/kn/sched/fixed");
}
```

3.13 Specify scheduler parameters for the specified range of priority levels - .priority

This instruction can only occur within a scheduler parameter block. Its syntax is as follows:

```
.priority (range=<range>, timeslice=<timeslice>,
policy="<policytool>");
```

where:

<range> is either a single priority level, or a range of the form *<low>-<high>*. A priority level is a number in the range 0-255.

<timeslice> specifies how timeslicing is to be performed. The value false causes timeslicing to be disabled for the specified priority level(s). The value true causes timeslicing to be enabled with a default timeslice length; this is appropriate for a platform that does not support a variable duty cycle timer. Any other value is taken to be the length of the timeslice for the specified priority level(s); this is equivalent to true on a platform with no variable duty cycle timer. If a timeslice period is specified, it is rounded up to the next higher multiple of the hardware timer's resolution. The timeslice period is specified as an integer value optionally followed by "ms" to indicate milliseconds, "us" to indicate microseconds, or "ns" to indicate nanoseconds. If none of these qualifiers is specified, nanoseconds is assumed.

Note that setting timeslice to false does not disable pre-emption, it simply disables round-robin timeslicing between processes at the same priority level. If a higher priority process becomes runnable due to an interrupt or some other condition occurring, it will run immediately.

<policytool> is the name of the scheduler policy tool to use for the specified priority levels.

Multiple *.priority* instructions may be specified within a .scheduler block. The ranges for these instructions must not overlap. There may be more than one scheduler policy tool specified by these instructions.

```
.scheduler {
    .priority (range=0-15, timeslice=false,
policy="sys/kn/sched/fixed");
    .priority (range=16-255, timeslice=true,
policy="sys/kn/sched/fixed");
}
```

3.14 Insert padding Bytes - .pad

The syntax for this instruction is

```
.pad ( offset=<offset> [ , fill=<fillbyte> ] );
```

It is used to insert padding bytes into the image such that the next object to be emitted into the image will be at a known address or offset, as specified. The value specified for offset is a 32 bit number which specifies either the offset from the beginning of the memory object (if the memory object is at an unknown base address) or the address to pad to (if the memory object is at a known base address). The "*fillbyte*" parameter can be optionally used to specify the byte value to use to fill the memory with (zero bytes will be used if this is not specified).

3.15 Add Directory to Sysgen's Search Path. - .path

```
.path ("<dirname>");
```

This instruction allows the user to add extra directories to the tool search path used by sysgen to find tools to go into the image.

Paths specified in this way are appended to the end of the initial search path produced by processing the command line, or the default initial search path consisting only of the current directory if no search paths are specified on the command-line.

3.16 .pii_opentool

```
.pii_opentool ("toolname");
```

This instuction specifies the tool to be used to handle PII opentool requests, and can only occur at the global scope.

It specified the tool which sysgen should call in order to generate tools when a tool-reference is encountered which begins with a "!" character. The specified tool is expected to follow the API documented in the PII design guide and API specifications.

```
.pii_opentool("sys/pii/somepii/s_opentool");
```

3.17 Specify a command to be run at the time the sysgen file is parsed - .system

```
.system ("<command>" [, <when>]);
```

This instruction allows the user to run commands when the sysgen file is parsed. This is useful, for example, for creating zip files or configuration files according to the sysgen options.

The <command> parameter is a string which is executed using the *system()* library function. Thus, it will be interpreted according to the underlying system shell's syntax rules.

The <when> parameter must be one of start , end and beforesize. It specifies when the command gets executed, as follows:

Value of when parameter	Point at which command is executed
start	At the point at which the instruction is parsed
end	After generating all images, map files and symbol files
beforesize	Immediately before determining sizes of objects in image

start system instructions are generally used to create files at the beginning of processing. Since the system command is executed as soon as the instruction is parsed, the result of the command can be used in the parsing of the file (for example, a file generated by a system command with its <when> parameter set to start can later be included using the *#include* directive).

end system instructions are most commonly used to clean up temporary files (zip filesystems, file lists, etc) after completion of the sysgen operation.

beforesize system instructions are useful for generating files to be contained in NDAs. Using this parameter, the command is executed at the last possible moment before the file's length is read and therefore the contents of the file must be effectively fixed.

Within the scope of the when parameter, the ordering of the system instructions is retained (i.e. the commands are executed in the order that they are specified in the sysgen instruction file, except where the when parameter overrides this).

If the when parameter is not specified, it defaults to start.

3.18 Groups Together Specified Objects. - .subblock

The syntax for this instruction is

```
.subblock (<addr>) {
    <object> ...
}
```

It is used to emit the set of objects enclosed within the set of braces after the *.subblock* instruction at the address or offset specified as a parameter to the instruction. These objects may be either tools or NDAs. Unlike the ".pad" instruction, other objects may be fitted around the *subblock* objects, so that the intervening space is not wasted.

3.19 Specify the name of file to which sysgen writes symbol-file information -.symbol_file

The syntax for this instruction is:

```
.symbol_file ("<filename>");
```

Description

This instruction should only occur at the global scope level (i.e., outside all memory blocks, subblocks or scheduler parameter blocks). The <filename> parameter is the name of the file to which the symbol-file information is written. No extensions are added to this filename by sysgen.

Example

```
.symbol_file ("sys/platform/win32/test.sym");
```

3.20 Others

All other lines which do not conform to any of the above syntaxes are taken to be tool names. These tools are translated and emitted into the sysgen image.

4. System Boot Description

The Elate system normally boots as follows:

On a hosted platform, a loader program exists. This is a program for the host OS, which is responsible for setting up the system ready for the Elate image to be started. It is also responsible for loading the Elate sysgen image into memory and calling its entry point. The host program generally also provides access to host OS services such as memory allocation, device access, etc.

On a non-hosted platform, either the image itself is directly booted when the machine is initialised or else a small loader program (such as a BIOS) loads the image and jumps to its entry point.

4.1.1 **Boot Tool**

Within the sysgen image, the platform-specific boot tool executes first (for further information on this please see the *Platform Isolation Interface Reference Manual.* This is contained within the release documentation in the *sys/pii/* directory). This may need to initialise processor states such as caches, page tables, system registers, etc. It must also set up a stack, and it must jump to the entry point of the next tool in the sysgen image. Normally the boot tool is stripped of its tool headers and resources, so a pointer to the header of the next tool is simply acquired by getting a pointer to the end of the boot tool. Since the boot tool has been stripped it must not perform any virtual calls, or virtual + fixup calls. If the sysgen image is completely fixed-up (the load address of the image was known at sysgen-time), it is possible to use normal tool references at this point. If the sysgen image is not completely fixed-up, no tool references at all may be used at this point, unless they are manually fixed up.

The next tool in the image should be called with a pointer to its tool header in the p0 register (see translator documentation for register passing conventions across calls). The BTF_FIXEDUP bit in the tool flags (at offset TL_FLAGS from the tool header) indicates whether or not the corresponding tool has already been fixed up. If this bit is set in the next tool in the image, then the value in the TL_CODE0 field of the tool header is a pointer to the entry point of the tool. If the BTF_FIXEDUP bit is not set, the value in the least significant 24 bits of TL_CODE0 is an offset from the tool header to the first entry point. At some point before the tool *sys/kn/init/p_start* is run (see later), a PROC_ structure must be set up (pointed to by GP, the globals pointer, which will contain process information such as the stack size, or the process id), and the stack checking variables within the PROC_ structure must also be set up.

The system may or may not be fixed up when the boot tool executes - this is known on a platform-specific basis. If the system is not fixed up, any references made by the boot tool will be relative to the entry point of the boot tool, taking the form of an offset. As a result of this these pointers should be manually fixed up before being used. Following this the boot tool should be stripped.

The following tools usually follow the boot tool in the image. They are called one after another in order, due to their location within the image.

4.1.2 **sys/kn/init/relocate**

This tool is portable (written in VP), and it relocates the data for any memory areas declared in the sysgen image which did not have an image file associated with them. Before this is done, no data or code within these areas can be accessed. When this tool is finished, it calls the entry point of the next tool in the image.

4.1.3 **sys/kn/init/makedata**

This tool is portable, and it fixes up any kernel lists which require fixing up. This includes the tool list, the NDA (Named Data Area) list and the NMA (Named Memory Area) list. When finished, it jumps to the entry point of the next tool in the image.

4.1.4 **sys/kn/init/tlfixup**

This tool is portable, and is used to fix up all tools that require fixing up. By this point, all fixup records in the tools in the image should either have been completely processed (by sysgen) or converted into absolute fixup records (also by sysgen), so that this fixup phase is simple and efficient. When finished, it jumps to the entry point of the next tool in the image.

4.1.5 **sys/kn/init/p_start**

This tool is portable. It processes a list of "run node records", which is constructed by sysgen to specify the number, type and order of things to be executed as the system boots. These include all *".call", ".spawn",* and *".obj"* instructions specified in the sysgen file. When finished, it loops, waiting for child processes to terminate, and reaping them when they do.

The boot sequence described above brings the system from a "just-booted" state into a state where VP is runnable, and where most of the system facilities are available. In addition, however, there are a number of kernel initialisation tools which are specified as *".call"* instructions in some of the kernel's sysgen instruction files (which are included by application or platform sysgen instruction files), and are therefore processed by *sys/kn/init/p_start:*

4.1.6 **sys/kn/init/makememobj**

This tool initialises and constructs all memory objects specified by the user as dynamically allocatable. Memory objects which are ROM, or which are RAM but do not have an allocator specified, are not processed.

4.1.7 **sys/kn/init/sched**

This tool initialises some data structures which are to be used by the scheduler later on.

4.1.8 **sys/kn/proc/pid/init**

This tool initialises the PID table (Process id), so that processes can be created.

4.1.9 **sys/kn/init/toolflush**

This tool installs an "out-of-memory" callback handler using the tool *"sys/kn/mem/addflush".* When memory is low, this routine is called, and it flushes all unreferenced tools from the tool-list.

These tools are usually processed before the user-specified tools. This is not strictly necessary, but there are some hard rules about ordering. For example, it is not possible to allocate memory before *"sys/kn/init/makememob*j" is called, it is not possible to start processes before *"sys/kn/proc/pid/init"* is called, and there should be no scheduler operations (either pre-emptive or co-operative) until after *"sys/kn/init/sched"* is called. For these reasons it is usually best to leave these tools where they are, so that they get executed before any others unless there is a good reason for doing otherwise.

5. Pre-Processor

The sysgen pre-processor operates on each line of input before it is processed by the main parser. It performs the following operations:

- Escaping newlines
- Removal of comments
- Operating on preprocessor variables
- Expansion of preprocessor variables
- Conditionally processing sections of the input
- Including files
- Printing messages to sysgen's stdout
- Aborting processing of instruction file

All pre-processor commands other than comments and variable expansions must be the first non-whitespace text on the line.

5.1 Escaping Newlines

If the last character on a line is a backslash, the backslash is removed and the contents of the next line are appended onto the current line before further processing.

5.2 Removal of comments

Comments in the style of C and C++ are supported. Any line whose first two non-whitespace characters are "//" will be ignored. Any text between the string "/*" and the corresponding "*/" will be ignored. Please note that nested "/**/" comments are properly processed. For example, the whole of the following string would be ignored:

```
/* This is a test comment /* This is an embedded comment */ This is another test comment */
```

5.3 Operating on pre-processor variables

The Sysgen pre-processor provides the facilities for variables to be defined, undefined, imported from the external environment, etc. The following directives are available:

#define VARIABLE
#define VARIABLE=VALUE

This directive sets the specified variable to the specified value (should one have been specified). If no value has been specified, the variable is simply defined to exist. However, if the variable has already been defined, its value will be replaced with the new value.

5.3.1 #undef VARIABLE

This directive removes any definition of the specified variable.

5.3.2 #import VARIABLE

This directive sets the specified variable to the value of the external environment variable of the same name. Only simple non-array environment variables are supported.

5.3.3 #asmimport INCLUDEFILE ASMVARIABLE [SYSGENVARIABLE]

This directive reads the value of the assembler variable ASMVARIABLE and sets a sysgen variable named either SYSGENVARIABLE (if it is specified) or ASMVARIABLE to the value of the assembler variable. The VP assembler itself it used to perform this operation, and the specified INCLUDEFILE is included. Only numeric variables (such as structure offsets etc.) can currently be imported in this manner.

5.4 Expansion of Pre-processor Variables

Any occurrence of the string *"$(name)"* in the input, where "name" is a pre-processor variable, will be replaced by the value of the variable "name". This is a textual substitution.

If "name" is not a valid pre-processor variable, an error will be raised. Pre-processor variables are expanded before any other pre-processor operations are done. Thus, pre-processor variables may be used in *#if* comparisons, and so on.

5.5 Conditionally processing sections of the input

5.5.1 #ifdef VARIABLE

This directive allows the following section of the instruction file to be processed if and only if the specified variable is defined. The following section of the instruction file is defined to be from the next line of the instruction file to the corresponding #else or #endif directive.

5.5.2 #ifndef VARIABLE

This directive allows the following section of the instruction file to be processed if and only if the specified variable is not defined.

5.5.3 #if CONDITION

This directive allows the following section of the instruction file to be processed if and only if the specified condition is true. The condition should be one of the following:

exist FILENAME	This condition evaluates to true if the specified file exists
!exist FILENAME	This condition evaluates to true if the specified file does not exist
VAR1 == VAR2	This condition evaluates to true if the two variables have equivalent values
VAR1 != VAR2	This condition evaluates to true if the two variables do not have equivalent values

For the comparison operations, if the variables are enclosed in double quotes, the comparison is done as a string comparison. Otherwise, a numeric comparison is done (the number format is not important, as it is the value, which is being compared). Valid number formats are: octal (indicated by a leading 0), hexadecimal (indicated by a leading $ or 0x) or decimal.

5.5.4 #else

This directive allows the following section of the instruction file to be processed if and only if the previous block was not processed due to its conditional not being satisfied. The following section of the instruction file is defined to be from the next line of the instruction file to the corresponding *#endif* directive.

5.5.5 #elseif CONDITION

This directive allows the following section of the instruction file to be processed if and only if the specified condition is true and the previous block was not processed due to its conditional not being satisfied. The following section of the instruction file is defined to be from the next line of the instruction file to the corresponding *#else* or *#endif* directive.

5.5.6 #endif

This directive terminates a conditional section, started by a *#if, #ifdef, #ifndef #else* or *#elseif* directive.

5.6 Including files

5.6.1 #include "FILENAME"

This directive inserts the contents of the specified file at the current point in the input.

5.7 Printing messages to sysgen's stdout

5.7.1 #print STRING

This directive prints the specified string to stdout.

5.7.2 #dumpenv

This directive prints the contents of the pre-processor variable table to stdout.

5.8 Aborting processing of instruction file

5.8.1 #error STRING

This directive prints the specified string to stderr, and aborts processing of the current instruction file.

6. Special Named Data Areas

There are four NDAs which have special significance in the images generated by sysgen. The contents of these NDAs are automatically generated by sysgen.

6.1 SYSGEN_RUNNODES

This NDA contains records identifying the operations to be performed when the system boots. These include records specifying device objects to be created, tools to be called and processes to be created. The type and internal format of the records in this NDA are identified by the first word in the record. The valid values for these are as follows:

- 0 = Terminator. This record type specifies that there are no more run records in the NDA.

- 1 = Self-relative call. The next word in the NDA contains an offset from the start of the record to the entry point of a tool to be called at boot-time. These records are 8 bytes in length, including the type field.

- 2 = Absolute call. The next word in the NDA contains the address of the entry point of a tool to be called at boot-time. These records are 8 bytes in length, including the type field.

- 3 = Create process, and do not wait for completion. Immediately following this type field is an Elate "spawn" structure and a "Process Control Block" structure, describing a process that is to be created at boot-time. The "spawn" structure is described in "*The Reference Manual For The Elate Realtime Kernel*". It has a variable size, which is stored in the first word of the structure (immediately after the type field). The "Process Control Block" structure has a fixed size. The *init* process does not wait for this process to complete (or even to run at all) before continuing.

- 4 = Device object, self-relative constructor. The next word in the NDA contains an offset from the start of the record to the entry point of a tool which is the constructor (the *_new* tool) for a device to be created at boot-time. The next word is the size of this record. Following this is the number of command-line argument strings to be passed to the device constructor. Then there is an array of offsets from the start of this record to the command-line strings. This array is terminated with a NULL offset. Immediately after this array is the name of the mount-point at which this device should be mounted if it is successfully initialised. After this, there are the device's command-line strings (referenced by the array earlier). At the end of these strings, there is padding to take the end of this record to an aligned address (if necessary).

  The constructor tool is called at boot-time with the standard conventions for device object constructors. If successful, the *init* method of the returned object will be called, with the specified command-line strings. If the device initialisation was successful, the object is then mounted at the specified mount point.

- 5 = Device object, absolute constructor. This record type is identical to type 4, except that the next word in the NDA contains a pointer to the device object constructor tool rather than an offset, and the array of command-line strings contains pointers to the strings rather than offsets.

- 6 = Create process, wait for completion. The structure of this record is identical to the structure of type 3 records. The init process waits for this process to complete before continuing.

The SYSGEN_RUNNODES NDA is processed at boot-time by the kernel initialisation tool *"sys/kn/init/p_start".*

6.2 SYSGEN_RELOCATES

This NDA contains records indicating which areas of memory (if any) should be relocated at boot-time. The first word in the record is the size of the area to relocate. The second word is the destination address for the block, and immediately following this is the data to be relocated. There may be multiple records of this type in this NDA. The records are terminated by one whose size field is zero.

The SYSGEN_RELOCATES NDA is processed at boot-time by the kernel tool *"sys/kn/init/relocate".*

6.3 SYSGEN_FIXUPFLAGS

This NDA is 4 bytes long, and it contains flags indicating which of the kernel data structures have been fixed up.

If bit 0 is set, the tool list pointers are self-relative. If bit 1 is set, the NDA list pointers and the data pointers in the NDA list nodes are relative to the start of the NDA list node containing them. If bit 2 is set, the atom list pointers are self-relative. If bit 3 is set, the NMA list pointers and the pointers to the manager tools within the NMA list nodes are relative to the start of the NMA list node containing them.

If any of these bits are set, the kernel tool *"sys/kn/init/makedata"* fixes up the corresponding lists at boot-time.

6.4 SYSGEN_TOOLLDR

This NDA contains information specified by the user in the sysgen instruction file, to be used during the execution of the system for dynamic tool loading and translation.

The first thing in this NDA is the name of the translator to be used for dynamic translation from VP to native code. Immediately following this is a 32 bit word containing the flags to be passed to this translator (note that this may be unaligned). Following the flags word is a string containing the list of tool extensions to use when searching for native tools. Each tool extension consists of

two characters (there is no nul character terminating these strings). If multiple tool extensions are valid (as is usually the case), their strings are concatenated together. For example, on a system where the tool extensions ".15", ".14", ".04", ".03" and ".00" are valid, the string would be

"1514040300". Tools are searched for in the order of the extensions in this string. So, for example, ".15" tools are loaded in preference to ".00" tools.

Following the tool extension string there may be padding bytes to make the size of this NDA 4 byte aligned if necessary.

7. Endianness

The image file produced by sysgen is little endian, and should be converted to big endian format for big endian target processors.

The program to do the conversion is *dd*. Here is an example of its use :

```
dd {conv=swab,swas} < sys/platform/mpc800/ads/molecule.img >
sys/platform/mpc800/ads/molly.img
```

8. Example Code

Shown below are several example sysgen instruction files. These are given as a demonstration of sysgen's facilities, and do not correspond directly to any particular system.

8.1 Example 1

```
//***************************************************
//Test sysgen instruction file for PC Standalone system
//Normal boot – Already in RAM prior to starting
//***************************************************
.trans ("sys/tr/i386");
.toolext ("15");
.toolext ("14");
.toolext ("04");
.toolext ("03");

.scheduler
{
       .priority  (range=0-255, timeslice=true, policy="sys/kn/sched/fixed");
}

.memory  (name="RAM", attr="rwx", base=$0, manager="sys/kn/mem/nstd/_new",
image="sys/platform/pcstand/k15stpc.img")
{
.map_region (defaultstack);
.map_region (defaultdata);
.map_region (system);
.map_region (defaultcode);
.map_region (defaultmail);
.map_region (staticdata);

#include sys/pii/pcstand/boot.sys
sys/kn/init/relocate
sys/kn/init/makedata
sys/kn/init/tlfixup
sys/kn/init/p_start
.call ("sys/pii/pcstand/meminit");
.call ("sys/kn/init/makememobj");
.call ("sys/kn/init/sched");
.call ("sys/kn/proc/pid/init");
.call ("sys/kn/init/toolflush");
#include sys/kn/ramdata.sys
#include sys/pii/pcstand/data.sys
#include sys/pii/pcstand/init.sys
#include sys/pii/pcstand/tools.sys

.obj  (newtool="dev/timer/pc/_new",mountstr="/device/timer",cmdline="timer -r6");

.obj  (newtool="dev/display/pcvga/_new",mountstr="/device/display",cmdline="display -
p");
.spawn  (tool="app/stdio/hello",cmdline="hello");
}
```

Each part of this file is now discussed in more detail:

```
.trans ("sys/tr/i386");
.toolext ("15");
```

In this portion of the sys file, the i386 translator is specified to translate VP byte codes into native code, while, the following tool extensions are required to be recognised by the tool loader:

- 15 = Pentium
- 14 = 486

- 04 = 386 with 387 maths co-processor
- 03 = 386

A .00 extension (VP byte codes) is always understood and is assumed to be at the bottom of this list. Note that the order of the tool extension list is important as this represents the priority of loading should a tool exist in different formats. For example if hello.15 and *hello.00* are both present on disk, then the .15 tool will be loaded in preference to the .00. There should always be a VP version of a tool to ensure portability, but occasionally native tools are coded for greater speed or where specific chip specific functionality is required.

```
.scheduler
{
.priority  (range=0-255, timeslice=true,
      policy="sys/kn/sched/fixed");
}
```

The *.scheduler* and *.priority* instruction allows the system builder to specify the type of scheduling algorithm to be used in the target. Various predefined schedulers are available (see '*The Kernel Reference Manual*' for more information) as well as the ability to specify a custom, user-defined scheduler. In this particular example, no scheduler has been selected (instruction is commented out).

```
.memory  (name="RAM", attr="rwx", base=$0,
manager="sys/kn/mem/nstd/_new",
image="sys/platform/pcstand/k15stpc.img")
```

The *.memory* instruction defines a memory area (it is possible to have more than one). Various attributes are initialised, for example, read/write/execute permissions, base address, memory manager used, as well as the name of the memory area. The image="..." parameter specifies the file into which the data and code in this memory area will be placed. The user is then expected to put this in memory at the appropriate address (0 in this case).

Once the memory area is defined the area can be split up into one or more regions, with the *.map_region* instruction.

```
.map_region (defaultstack);
.map_region (defaultdata);
.map_region (system);
.map_region (defaultcode);
.map_region (defaultmail);
.map_region (staticdata);
```

The *.map_region* (defaultstack); instruction states that the memory area containing this instruction will be used as the default location for allocating stack memory at runtime. Essentially, the above series of instructions specifies which memory area is to be used to provide memory services for one of the following purposes:

- stack – process stack space
- data – memory space for allocating data
- system – memory space for the Elate system itself
- code – memory space for storing dynamically loaded tools
- mail – memory space allocated for mail messages
- static – Used by libraries to allocate data memory which may need to be shared with other processes, and which must exist after the allocating process terminates.

```
#include sys/pii/pcstand/boot.sys
```

The *#include* instruction includes instructions from another sys file. In this case it includes the following instructions :

```
.strip ("sys/pii/pcstand/boot");
.nda ("PLATFORM_DATA",16);
```

The *.strip* instruction strips the header and trailer from the specified tool and emits the tool to the sysgen image. In this case the tool concerned is the boot tool. The boot tool is responsible for booting the Elate system into a runnable state. This includes setting up certain processor specific memory areas and making any initial memory block available. A named data area (NDA) is also created by the *.nda* instruction. This is memory in the kernel data area that can be referred to by a name rather than a pointer and in this case the memory is used by the boot tool to store the memory base address and size. In this example the NDA is 16 bytes and is called PLATFORM_DATA.

```
sys/kn/init/relocate
sys/kn/init/makedata
sys/kn/init/tlfixup
sys/kn/init/p_start
```

This is a list of special kernel initialisation tools that must be included in the system image.

```
.call ("sys/pii/pcstand/meminit");
.call ("sys/kn/init/makememobj");
.call ("sys/kn/init/sched");
.call ("sys/kn/proc/pid/init");
.call ("sys/kn/init/toolflush");
```

The *.call* instruction allows the user to specify a tool to be called during the system boot sequence. This is usually used for kernel and PII initialisation purposes, as is the case in this example.

```
#include sys/kn/ramdata.sys
#include sys/pii/pcstand/data.sys
#include sys/pii/pcstand/init.sys
#include sys/pii/pcstand/tools.sys
```

These instructions include further instructions from the specified sys files. In this example important kernel and PII tools are included.

```
.obj(newtool="dev/timer/pc/_new",mountstr="/device/timer",cmdline="ti
mer -r6");
```

The *.obj* instruction allows the user to specify ncallable device objects to be started when the system boots. In this example the timer device driver is not started, as no timeslicing is required.

```
.obj(newtool="dev/display/pcvga/_new",mountstr="/device/display",cmdl
ine="display -p");
```

This instruction would have started the VGA display device driver, but is commented out. The example application spawned by the next instruction uses *ktrace* (a character based tracing mechanism). Note that the *ktrace* tool is very low level and does not require a display device driver as it communicates with the underlying system via *sys/pii/odata*.

```
.spawn  (tool="app/stdio/hello",cmdline="hello");
```

This instruction spawns (causes to execute) the specified tool, in this case *app/stdio/hello*. The *app/stdio/hello* tool itself has just one instruction:

```
        ktrace “hello world”
}
```

8.2 Example 2

The following file shows a simple instruction file which defines a single memory area called "RAM". The size and base address of this memory area are undefined at sysgen-time, and are filled-in by the platform-specific boot tools at the beginning of the image.

```
.trans ("sys/tr/i386");
.toolext ("03");

.scheduler
{
        .priority  (range=0-255, timeslice=true, policy="sys/kn/sched/fixed");
}

.memory (name="RAM", attr="rwx", image="ram.img", manager="sys/kn/mem/nstd/_new")
{
        .map_region (defaultstack);
        .map_region (defaultdata);
        .map_region (system);
        .map_region (defaultcode);
        .map_region (defaultmail);
        .map_region (staticdata);

        #include sys/pii/doshost/boot.sys
        #include sys/kn/raminit.sys
        #include sys/pii/doshost/init.sys
        #include sys/pii/doshost/tools.sys

        // Device drivers here....
        .obj (newtool="dev/timer/pc/_new",mountstr="/device/timer", cmdline="timer");

        // Applications here....
        .spawn (tool="sys/kn/proc/tst/simple", stack="RAM", data="RAM");
        .spawn (tool="sys/kn/proc/tst/simple", stack="RAM", data="RAM");
        .spawn (tool="sys/kn/int/tst/waitint");
}
```

8.3 Example 3

The following file shows an instruction file which defines two memory areas called "RAM" and "BOOTROM". The scheduler is configured so that priority levels 0 - 15 do not timeslice, but levels 16 - 255 do timeslice.

```
.trans ("sys/tr/i386");
.toolext ("03");

.scheduler
{
        .priority  (range=0-15, timeslice=false, policy="sys/kn/sched/fixed");
        .priority  (range=16-255, timeslice=true, policy="sys/kn/sched/fixed");
}

.memory (name="RAM", attr="rwx", image="ram.img", manager="sys/kn/mem/nstd/_new",
        size=$00400000, base=$00100000)
{
        .map_region (defaultstack)
        .map_region (defaultdata)
        .map_region (system)
        .map_region (defaultcode)

        #include sys/pii/doshost/boot.sys
        #include sys/kn/raminit.sys
        #include sys/pii/doshost/init.sys
        #include sys/pii/doshost/tools.sys

        // Device drivers
        .obj (newtool="dev/fs/doshost/_new", mountstr="/", cmdline="fs");

        ;Applications....
        .spawn (tool="sys/kn/proc/tst/simple", stack="RAM", data="RAM");
        .spawn (tool="sys/kn/proc/tst/simple", stack="RAM", data="RAM");
        .spawn (tool="sys/kn/int/tst/waitint");
}

.memory (name="BOOTROM", attr="rx", image="bootrom.img", base=$F0000, size=$10000)
{
        // This is an example of a set of tools which might be required
        // to boot the system:

        // Switch to protected mode
        .strip ("sys/pii/platform/toprot.03");


        // Load the RAM image into RAM. This may copy from ROM into
        // RAM, perhaps uncompress, perhaps load from a filesystem, etc.
        .strip ("sys/pii/platform/loadram.03");



        // Jumps to the start of the RAM image, where
        // the main Elate system boot occurs
        .strip ("sys/pii/platform/jumpram.03");

        // Ensure that the bootstrap code is at the end of the ROM.
        // Using a subblock means that any dependency tools are fitted
        // around the subblock to allow better space utilisation than
        // if a pad instruction was used
        .subblock (0xfff0) {
                /*
                This is where the processor's execution starts.
                This tool should be less than 16 bytes, and should jump
                to the start of this ROM, where the rest of the system
                initialisation will take place
                */
                .strip ("sys/pii/platform/sysboot.03");
        }
}
```

8.4 Example 4

The following file shows an instruction file which defines two memory areas called "RAM" and "ROM". This demonstrates a typical ROM-based system. All of the code is stored in the ROM.

The RAM area does not specify an image file, since there is physically no way to initialise this memory at power-up. Instead, any initialised data for the RAM memory area is emitted into the ROM memory. This data (or code, if required) will be automatically relocated to the correct address in RAM during the kernel boot sequence.

This example also loads several device drivers, including a filesystem which accesses a filesystem image stored in ROM (actually a ZIP file), allowing file access and dynamic tool loading without requiring a disk drive.

```
.trans ("sys/tr/i386");
.toolext ("03");

.scheduler
{
        .priority  (range=0-255, timeslice=false, policy="sys/kn/sched/fixed");
}


.memory (name="RAM", attr="wrx", manager="sys/kn/mem/std/_new", base=0x00100000,
        size=0x00100000)
{
        .map_region (defaultstack);
        .map_region (defaultdata);
        .map_region (system);
        .map_region (defaultcode);
        .map_region (defaultmail);
        .map_region (staticdata);
        .nda ("SYSGEN_RUNNODES");
        #include sys/kn/ramdata.sys
}

.memory (name="ROM", attr="rx", image="rom.img", base=0x00000000, size=0x00100000)
{
        #include sys/platform/pcstand/boot.sys
        #include sys/kn/rominit.sys
        #include sys/platform/pcstand/init.sys
        #include sys/platform/pcstand/tools.sys

        .obj  (newtool="dev/timer/pc/_new",mountstr="/device/timer",cmdline="timer");
        .obj  (newtool="dev/fs/zip/_new",mountstr="/",cmdline="zip -zzipfs.nda");
        .obj
(newtool="dev/tool/elate/_new",mountstr="/device/loader",cmdline="tlldr");

        .emit_dependencies
        .nda ("zipfs.nda","zipfs.zip");
}
```

9. Appendix - Elate Symbol File Structure

9.1 Introduction

The symbol file generated by sysgen is essentially a machine-readable version of the MAP file generated by sysgen. It contains information about the tools, atoms and NDAs in an image, and the memory layout of the target system.

The symbol file also contains debug information for tools containing it, which is not generated in the MAP file. This allows the symbol file to be used by the Elate debugger to provide source-level debugging facilities without requiring the debug information to be sent across the host-target link, which is often comparatively slow. The symbol file also contains enough information to allow creation of other symbol table formats, or executable formats, such as ELF.

9.2 Structures

The symbol file has a small file header, with the following structure:

Size	Description
unsigned char [8]	Magic number (Hexadecimal bytes: 1B F8 01 8D D7 FD A4 9D)
int32	Structure version number of symbol file contents (current version = 1)
int32	Elate processor type number of the executable code in the corresponding image

Following this header, there are a variable number of sections of different types. Each header has a small header, defined as follows:

Size	Description
char [4]	Magic number, determining the section type
int32	Size of section
int32	Flags indicating section parameters

The following section header magic numbers are currently defined:

"MEML": Specifies the memory layout of the target system
"TLST": Contains the addresses, sizes and debug information for all tools in the image
"ATOM": Describes the atoms which are statically defined in the image
"NDAS": Defines the NDA names defined in the image and their corresponding addresses

New section types may be defined in future versions of this specification. If a section is found whose magic number is unrecognised, the section should be ignored.

The size stored in the section header contains the file offset between the beginning of adjacent section headers. This chain of section headers is terminated by a header whose magic number contains four zero bytes.

There may be padding after the useful data in each section, so the only reliable way to find the start of the next section is to use the file offsets given in the headers. For every type of section it should be possible to unambiguously determine the end of the valid data, although this does not necessary also correspond to the start of the next section.

9.3 Memory Layout Section

This section of the symbol file describes the memory layout of the target system, as defined in the sysgen instruction file. The flags in the header for this section have the following definitions:

Bit Number	Description
0-31	Undefined (set to 0)

After the section header, there is an int32 containing the number of memory records in this section. The first memory record follows immediately. Memory records have the following structure:

Size	Description
int32	Base address of memory area (0xffffffff if the base address is unknown)
int32	Size of memory area (0x00000000 if the size is unknown)
int32	Attributes of memory area (see below)
char []	Nul-terminated name given to memory area in sysgen instruction file
char []	Nul-terminated name of associated image file (string may be empty if there is no image file for the memory area)

The attribute field of the memory record structure has the following bit definitions:

Bit Number	Description
0	Bit is set if memory area is readable
1	Bit is set if memory area is writable
2	Bit is set if memory area is executable
3-31	Undefined (set to 0)

9.4 Tool List Section

This section contains information about all (non stripped) tools in the image. The flags in the header for this section have the following definitions:

Bit Number	Description
0	Section contains absolute addresses if this bit is clear, otherwise the addresses in the section are relative to the load address of the image (unknown until load time)
1-31	Undefined (set to 0)

After the section header there is an int32 containing the number of tool records in this section. The first tool record follows immediately. Tool records have the following structure:

Size	Description
int32	Address of tool in target memory (offset relative to image load address if bit 0 is set in symbol file header flags)
int32	Size of tool
int32	Address of tool entry point in target memory (offset relative to image load address if bit 0 is set in symbol file header flags)
int32	Size of code
char []	Nul-terminated toolname
int32	Size of debug data (may be 0)
[]	Copy of debug information from tool after translation, as written into image (may have 0 length if tool had no debug information)

The format of the data in the debug information part of the record is as specified in *docn/struct/debug.html*. These nodes contain all of the debug information from each tool in the image which had any, from the NULL word indicating the start of the debug info up to and including the debug signature word at the end.

9.5 Atom Section

This section contains information about all statically created atoms defined in the image. The flags in the header for this section have the following definitions:

Bit Number	Description
0-31	Undefined (set to 0)

After the section header there is an int32 containing the number of atom records in the section. The first atom record follows immediately. Atom records have the following structure:

Size	Description
int32	Atom value
char []	Nul-terminated atom string

9.6 Nda Section

This section contains information about all NDAs in the image. The flags in the header for this section have the following definitions:

Bit Number	Description
0	Section contains absolute addresses if this bit is clear, otherwise the addresses in the section are relative to the load address of the image (unknown until load time)
1-31	Undefined (set to 0)

After the section header there is an int32 containing the number of NDA records in the section. The first NDA record follows immediately. NDA records have the following structure:

Size	Description
int32	Pointer to data area associated with name
char []	NDA name

For more information, please see the file *elate/app/sysgen/symbol.html.*