Reference Manual for the Sysbuild Utility

Version 1.3

1. Overview

At its more primary level, the int**e**nt® environment is built upon concepts of stream-lining, efficiency and compactness. The system is tailored to the needs of whatever application it is running, so as to guarantee that only those tools and resources which are actually required by the application are included. During run time, this policy is pursued through the use of dynamic binding, which can be used to ensure that tools are only brought into memory when needed.

At design time, furthermore, int**e**nt uses a number of mechanisms to ensure that the system image from which the application is to be run contains the minimum of tools which are needed to successfully run the application.

For the creation of such system images, int**e**nt provides two image generation utilities, the System Generation Utility (Sysgen), and sysbuild. This document is designed to facilitate the use of the sysbuild utility. A detailed account of sysgen may be found in the *Sysgen Reference Manual*.

The sysbuild utility is given a configuration file, which contains details of tools and data files which are required by the application, and information on how the program should be run. The sysbuild utility then calls upon the underlying Sysgen functionality. All tools specified in the applications sysbuild file are then loaded, translated, bound and written to an int**e**nt system image. This image file is a bootable int**e**nt image which can be downloaded to the target hardware. It contains only tools specified in the original application sysbuild file, and their dependent tools.

2. Invoking Sysbuild

Sysbuild may be invoked to create a bootable intent image by using the following command line:

```
sysbuild [options] <platform> <appsysfile>
```

Here *<platform>* signifies the platform for which the image is to be created, e.g. win32.

The parameter *<appsysfile>* specifies the application's sysbuild file. This file describes the application in terms of the tools and data files required, and contains details on how the program should be run. (The format of such files is explored in greater depth in the section "Creation of an application system configuration file" later in this document.) This parameter can be given as an absolute path to the file. If no path is given, sysbuild will look for the file in the *sys/platform* and *sys/platform/<platform>* directories. All application sysbuild files must have the extension .sys, but this does not need to be specified on the command line.

Thus, for example, either of the following command lines can be used to invoke the sysbuild utility to create a bootable system image for a win32 platform using specifications from an application sysbuild file *sys/platform/win32/filename.sys* (where *filename.sys* is simply an example name):

```
sysbuild win32 sys/platform/win32/filename
```

or

```
sysbuild win32 filename
```

2.1. Command options

A range of command options are available. Some of these are specific to use with the *sysbuild* command. Most, however, may also be used when invoking sysgen. All sysgen options are also supported by sysbuild, although in the case of *–m*, *-n* and *–s* all three options are set as a default with sysbuild. Sysbuild options may appear in any order on the command line, but must be specified separately.

2.1.1. Sysbuild specific options

-debug
: Sets options to produce an image for debugging purposes. This provides the same functionality as the *-nos* and *-DWANT_CHECKING_TRANSLATOR* options (see below).

-DWANT_DEBUGFS
: This option will give debug output for filesystem access. This can be very useful when building a new application file to ensure all required tools and data files have been included.

-DWANT_APP_AUTOSTART
: This option allows the user to specify that an example application will be started when the image loads, rather than needing to be started from the AVE menu in the default fashion. All of the example application sysbuild files support this option.

2.1.2. Sysgen options

The following options are all sysgen options, but are also supported by sysbuild. Sysgen options *-n*, *-m* and *-s* are on by default.

The options to sysbuild must be specified individually (for example, "*-m -n -s*"). Options that take a parameter, such as *–f* or *-D*, are treated in a different fashion. In such cases, the remainder of that command line argument, if there is one, 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.

-b
: Enables embedding (currently off by default - though this may change in future releases).

-c
: Changes VIRTUAL+FIXUP *qcall*s into VIRTUAL *qcall*s.

-f <path>
: This option allows 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. 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 an *-f* option if it is one of the desired search paths.

-i
: Includes tools which are referenced as virtual or virtual+fixup into the image.

-l
: This option prints a list of the tools included in/excluded from the image.

-N
: Changes VIRTUAL *qcall*s into normal *qcall*s. It should be noted that this modifies VP tools before translation. The underlying sysgen functionality cannot modify the types of *qcall*s in native code.

-p
: This option specifies that sysbuild should only parse input files, and should not produce any output or perform any dependency searching. This can be useful for checking the syntax of instruction files.

-q
: This option enables quiet mode. In this mode, text is only printed if it is an error report.

-t
: Generates a symbol file for the int**e**nt debugger. The symbol file 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, allowing the symbol file to be used by the int**e**nt debugger to provide source-level debugging facilities without the slow process of sending the debug information across the host-target link. Further information concerning the symbol file can be found in the document on Symbol File Structure in the file *app/sysgen/symbol.html* in the build.

-v
: This option enables verbose mode. In this mode, considerable diagnostic output should be expected.

-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.

2.1.3. -m, -n and -s

The following options are passed through to sysgen.

-m
: Generates a map file. This file contains a list of the components written to the output image file, and their locations.

-n
: Changes VIRTUAL+FIXUP *qcall*s into normal *qcall*s. This option acts to modify VP tools before translation, and cannot be used to alter native tools.

-s
: Strips resource names from tools where possible.

However, when used with the *sysbuild* command, *-m*, *-n* and *–s* are enabled by default. As a consequence, the options *–nom*, *-non* and *–nos* are provided for use with *sysbuild*, to disable each of these options.

-nom

Stops the generation of a map file (disables the *–m* option).

-non

Turns off the *–n* option.

-nos

Disables the *–s* option so that resource names are not stripped from tools. This has the same result as the use of the *–debug* option.

3. Creation of application system configuration files

An application's system configuration file contains details of tools and data files which are required by the application, and information on how the program should be run. The *sysbuild* command calls upon the sysgen program to create the required system image. It is the sysgen utility that is responsible for parsing the application system configuration file. Further information on the command syntax can be found in the 'Syntactic Descriptions' chapter in the *Sysgen Reference Manual*.

Application system configuration files for use with the *sysbuild* command all make use of the same basic format. The file is divided into a sequence of five 'stages,' which are included at different points in the image building process.

SETUP	This stage is used to set up the configuration required for the specific application. If it is a PJava application, for example, then it will need to define WANT_PJAVA so that the PJava libraries are available.
DEPENDS	This stage is used to set up any optional parts of the application's configuration depending on the platform. This stage is not usually needed.
ROMTOOLS	This stage contains a list of the tools that are required to run the application. This will usually be divided into two lists, a list of executables and a list of data files.
RAMTOOLS	This stage contains a list of tools that must be stored in the platform's Read Write memory area. This section is also usually not needed.
APPS	This stage contains the information required to start the application.

Each of these is described in more detail below.

3.1. The SETUP stage

The 'setup' stage is used to set up the configuration required for the specific application. It is always enclosed by the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)
```

The comment on the second line is optional, but has been included to make the files more readable.

All setup commands consist of a *#define* or *#redefine* statement that may be conditional. To set a value the following format may be used:

```
#redefine <value to set>
```

To set it to a specific string use:

```
#redefine <value to set><string>
```

Values can also be set to other values, or combinations of other values. For example:

```
#redefine AWD_OPTIONS $(AWD_DEVICE) $(AWD_OPTIONS)
```

It would be possible to use the *#define* command instead of *#redefine* in the examples above, but only if the application or platform has not already set the value. If the value has already been set in this fashion, the *#define* command will fail.

In some cases values should only be set if they are not currently set. In the example below, the *#ifndef* command is used to establish whether this is the case. If check proves that the value is not set, then the value is defined by the *#define* command. It is safe to use the *#define* command here, since it follows a check to verify that the value is not already defined.

```
#ifndef IMAGE_FILE
   #define IMAGE_FILE example
#endif
```

The values that can be defined in this section are listed in the build document *sys/platform/sysbuildapp.html*. More details of these values may be found in the *Sysbuild Reference Guide*, which can be found in the build in *sys/platform/sysbuildref.html*.

3.2. The DEPENDS stage

The ‘depends’ stage is used to set up any optional parts of the application's configuration, depending on the platform. This stage is optional, and is not usually required in an application system configuration file. It is not used to indicate those features of the intent system and platform upon which the application depends.

All commands in this stage must be enclosed between the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_DEPENDS)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_DEPENDS)
```

As in the case of the last section, the comment in the latter line is designed to make the code more readable.

Any of the values defined in the ‘setup’ stage can be used in this stage. Optional parts of the application can be conditional on any of the values defined in the ‘setup’ stage.

3.3. The ROMTOOLS stage

This stage contains a list of the tools which are required to run the application, and which can be run from ROM. Most required tools will be capable of running from ROM. There will usually be two lists, a list of executables and a list of data files.

All commands in this stage must be enclosed between the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)
```

To include an VP tool, one should simply specify the toolname as in the example below.

```
path/to/tool/toolname
```

The specified tool and all tools that it relies on will be loaded into the image. The tool’s extension can be given if it is important to specify which version is to be used. However, in general this approach is not recommended.

To include a Java class file, specify the file as in the example below.

```
my/java/tool.class
```

In the case of Java classes, the .class extension must be given.

The ‘depends’ stage should also contain a list of data files that the application requires. These are included in a slightly different way. To include the file *images/image.gif* use the following command:

```
.system("echo \
images/image.gif \
>> $(CFILENAME));
```

More than one file may be included, as in the example below.

```
.system("echo \
images/image1.gif \
images/image2.gif \
>> $(UFILENAME));
```

It should be noted that either UFILENAME or CFILENAME can be used to define data files. Data files followed by CFILENAME will be compressed if a suitable filesystem is used, whereas files followed by UFILENAME will remain uncompressed. It is also possible in this stage to specify which tools should be dynamically translated. However tools that these depend upon will not automatically be loaded into the image, and will also need to be specified. If data files are required for this application, then the value WANT_FILESYSTEM should have been set in the 'setup' stage.

Application configuration files that need to be modified should also be specified in this stage. In this case, the 'setup' section should contain WANT_RWFS so that a Read/Write filesystem is available to store the modified files.

This stage can have conditional stages so that some files (tools or data) are only included if a certain value is defined, for example if there is an optional part of the application that relies on a ppp dialup connection you could have:

```
#ifdef WANT_PPP
   list/of/tools
   and/data/files
   for/ppp/option
#endif
```

3.4. The RAMTOOLS stage

This stage contains a list of tools and data that must be stored in the platform's Read/Write memory area. It is unusual for tools to be included in RAM, and most of them are run from ROM. This stage is usually not needed.

All commands in this stage must be enclosed between the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_RAMTOOLS)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_RAMTOOLS)
```

3.5. The APPS stage

This stage contains the information required to start the application.

All commands in this stage must be enclosed between the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_APPS)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_APPS)
```

In the following example, the command line is being used to run a Java application *my/java/app.class*.

```
.spawn(tool="app/stdio/jcode",cmdline="jcode my/java/app.class");
```

Further information on running Java programs may be found in *Using Java Technology with intent*. More general information on starting applications may be found in the *Sysgen Reference Manual*.

This stage can also use conditional statements to start selectively different applications or parts of applications.

4. Creation of platform system configuration files

This section describes how to create new platform system configuration files for use with the *sysbuild* shell command.

The platform system configuration information is split into two, or sometimes three, files.

- *platform.sys* – This is the platform system configuration file, and contains all information about the platform except for details of devices.
- *devices.sys* - This is the devices system configuration file, and defines the devices that the platform supports.
- *sysbuild* - This third file is optional. It takes the form of a shell script, which is designed to process the image file after it has been created.

Further details of each of these files is provided in this section.

4.1. Platform system configuration file

The platform system configuration file is split up into several stages, which are included at various points of the image building process.

SETUP	This stage is used to set up the configuration required for the specific platform. This may include setting some AVE parameters, or defining the serial port configuration for the Sejin keyboard etc.
DEPENDS	This stage is used to set up any optional parts of the platform configuration depending on what is required by the applications and other areas.
MEMORY	This stage defines the memory layout for the platform. It defines the platform boot tools, memory regions etc.

Each of these stages is described in more detail below.

4.1.1. The SETUP stage

This stage is used to set up the configuration required for the platform. Values following the format WANT_xxxx will normally be defined by the application and int**e**nt system configuration files.

The ‘setup’ stage is enclosed by the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)
```

In this stage, *#define* and *#redefine* can be used to set values in the same fashion as for the ‘setup’ stage for an application system configuration file.

The values that can be defined in this section are listed in the build document *sys/platform/sysbuildapp.html*. More details of these values may be found in the *Sysbuild Reference Guide*, which can be found in the build in *sys/platform/sysbuildref.html*.

4.1.2. The DEPENDS stage

This stage sets up any optional parts of the platform configuration according to the requirements of the applications, etc. The ‘depends’ stage is enclosed by the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_DEPENDS)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_DEPENDS)
```

This stage will be comprised of conditional statements, of the form "if this then that". For example if the platform does not have a keyboard, but it is usual to connect a Sejin keyboard to a serial port, then the following will be needed:

```
#ifdef WANT_KEYRAW
    #redefine WANT_SEJIN
    #redefine WANT_SEJIN_KEYRAW
#endif
```

Besides setting up application-specific aspects of the platform configuration, the ‘depends’ stage usually converts any generic WANT_xxxx settings to settings appropriate to the specific platform. For example, the WANT_TCPIP setting may indicate that the dial up devices PPP, AWD, etc. are required for this platform.

The values that can be used in this stage are listed in the build document *sys/platform/sysbuildplat.html*.

4.1.3. The MEMORY stage

This stage is used to define the platform's memory layout. It also contains various other platform specific information. All commands for this stage must exist in between the following two lines:

```
#if $(BUILD_STAGE) == $(BUILD_STAGE_MEMORY)
#endif // $(BUILD_STAGE) == $(BUILD_STAGE_MEMORY)
```

This stage also has other structural requirements and will generally look as follows:

```
// TRANSLATORS

// translator names and flags
.trans ("$(VPTRANS)");
.jtrans ("$(JCTRANS)",<jcode translator flags>);

// tool extentions
.toolext ("XX");

#ifdef WANT_LINK_SLAVE
   .chipnum(-1);
#endif

// SCHEDULER

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

// IMAGE FILE

// ROM system
.memory (name="ROM",attr="rx", \
         image="sys/platform/<platform>/$(IMAGE_FILE).img", \
         manager="sys/kn/mem/gwc/_new")
{
   // PLATFORM BOOT TOOLS

   // KERNEL ROM TOOLS

   #include sys/kn/rominit.sys

   // PII

   // RUNTIME INIT

   .map_region (staticdata_ro);
   .nda ("SYSGEN_RUNNODES");
   .emit_dependencies;
   // APPLICATIONS AND DEVICES

   #redefine BUILD_STAGE $(BUILD_STAGE_DEVICES)
   #include sys/platform/<platform>/devices.sys

   #redefine BUILD_STAGE $(BUILD_STAGE_ROMTOOLS)
   #include sys/platform/elate.sysinc

   #redefine BUILD_STAGE $(BUILD_STAGE_APPS)
   #include sys/platform/elate.sysinc
}

// RAM system
.memory (name="RAM",attr="rwx")
{

   // PLATFORM SPECIFIC NDAS

   // KERNEL STATICS

   #include sys/kn/ramdata.sys

   // MEMORY OBJECTS

   // system memory
   .map_region (system);

   // tool memory
   .map_region (defaultcode);

   // statics r/w memory
   .map_region (staticdata);

   // stack memory
   .map_region (defaultstack);

   // mail memory
   .map_region (defaultmail);

   // shared memory
   .map_region (defaultdata);

   #redefine BUILD_STAGE $(BUILD_STAGE_RAMTOOLS)
   #include sys/platform/elate.sysinc
}

// READ ONLY FILESYSTEM

#include sys/platform/buildrofs.sysinc
```

It should be noted that in the example above there are two memory regions, ROM and RAM. Not all platforms will require two separate regions in this fashion. Information on filling out the above example can be found in the sysgen documentation in the build.

The "PLATFORM BOOT TOOLS" section of this stage should contain the tools required to boot the platform. These are usually a mixture of native and VP binaries.

The "PII" section of this stage should contain all tools required by the Platform Isolation Interface. This again will be a mixture of native and VP binaries.

4.2. Devices system configuration file

This file defines which device drivers the platform supports. This will consist of:

- Platform specific device drivers
- Generic Elate® device drivers

4.2.1. Platform specific device drivers

Platform specific devices are defined using the *.obj* sysgen command. These should be wrapped in ifdef/endif directives in the following manner:

```
#ifdef WANT_ETHERNET
   .obj(newtool="/dev/network/myplat/_new", \
        mountstr="/device/network",cmdline="myplat",local);
#endif
```

For most platform specific devices, the *local* option should be used as in the example above. The exceptions are keyboard, keyraw, pointer, TCP/IP and display devices. In such cases, instead of local the option used should be *$(<device>_LOCAL_REMOTE)*, where *<device>* is *KEYBOARD*, *KEYRAW*, etc, depending upon the device being defined. Thus a platform specific keyraw device might be defined as follows:

```
#ifdef WANT_KEYRAW
   .obj(newtool="/dev/keyboard/myplat/_new", \
mountstr="/device/keyboard",cmdline="myplat",$(KEYRAW_LOCAL_REMOTE));
#endif
```

The *<device>_LOCAL_REMOTE* values are defined in the generic system configuration files for use with the link driver.

If the platform has a socket device then it should be mounted on *$(INET_MOUNT).* Any platform specific Read/Write filesystem should be mounted on *$(PLATFSROOT*). These are both set up by the generic platform configuration files.

4.2.2. Elate generic devices

To include an Elate generic device use the following syntax :

```
#redefine $(DEVICE_STAGE)=$(DEVICE_STAGE_XXXX)
#include sys/platform/elate.sysinc
```

The following is a list of the available generic devices:

DEVICE_STAGE_TOOLLOADER	DEVICE_STAGE_AVE
DEVICE_STAGE_NULL	DEVICE_STAGE_LINK_MASTER
DEVICE_STAGE_TRACE	DEVICE_STAGE_LINK_SLAVE
DEVICE_STAGE_ERROR	DEVICE_STAGE_ELATE_TCPIP
DEVICE_STAGE_PIPE	DEVICE_STAGE_SEJIN
DEVICE_STAGE_RAMFS	DEVICE_STAGE_SEJIN_KEYRAW
DEVICE_STAGE_ZIP_ROFS	DEVICE_STAGE_SEJIN_POINTER
DEVICE_STAGE_EFS_ROFS	DEVICE_STAGE_AWD
DEVICE_STAGE_ELATE_ROFS	DEVICE_STAGE_SCRIPT
DEVICE_STAGE_MERGEFS	DEVICE_STAGE_PPP
DEVICE_STAGE_DEBUGFS	DEVICE_STAGE_DEBUGSTUB
DEVICE_STAGE_KEYBOARD	DEVICE_STAGE_DEVZIP_ROFS
DEVICE_STAGE_ADDMEM	DEVICE_STAGE_MOUNT_ZIP
DEVICE_STAGE_CONSOLE	

The values above are described in more detail in the *Sysbuild Reference Guide* which can be found in the build in *sys/platform/sysbuildref*.

The devices system configuration files should only include devices that are suited to the specific platform. For this reason, it may in some cases by necessary for a platform to have a generic device set up in a special and platform-specific way.

4.3. Post sysbuild script

This optional file, *sys/platform/<platform>/sysbuild*, is a shell script that is executed after the image has been created. This file does not exist for every platform.

This script can be used to transform the image before downloading. For example, it maybe used so that byte swapping, Srecord creation, and so forth, occur automatically.

The first command line option will be the name of the image created, for example *avedemos*. It does not contain the path or the file extension. The remaining command line options will be the options given to sysgen for image creation.

5. Example Code

Shown below are several example application system configuration files.

5.1. Example 1

The file below relates to a number of demo applications which can easily be accessed from intent's audio visual environment. This file does not contain a 'depends' or a 'ramtools' file.

```
//Setup Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)

     // sysgen image filename
     #ifndef IMAGE_FILE
          #define IMAGE_FILE avedemos
     #endif

     #redefine WANT_AVE
     #redefine WANT_TTF_FONT_ENGINE
     #redefine WANT_FILESYSTEM

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)


//ROMTOOLS Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)

     //Data Files Required
     .system ("echo \
     images/cradle.flm \
     images/cface96.gif \
     images/rings.cpm \
     fonts/curlz.ttf \
     fonts/glowworm.ttf \
     fonts/mmincho.ttf \
     app/start/ave/avedemos.scr \
     app/start/ave/demos/blend.scr \
     app/start/ave/demos/boing.scr \
     app/start/ave/demos/clock.scr \
     app/start/ave/demos/filters.scr \
     app/start/ave/demos/fonts.scr \
     app/start/ave/demos/freeball.scr \
     app/start/ave/demos/gadgets.scr \
     app/start/ave/demos/images.scr \
     images/boing1.gif \
     images/boing2.gif \
     images/boing3.gif \
     images/boing4.gif \
     images/boing5.gif \
     images/boing6.gif \
     images/boing7.gif \
     images/boing8.gif \
     images/boing9.gif \
     images/boing10.gif \
     images/boing11.gif \
     images/boing12.gif \
     images/boingm.cpm \
     images/boings.cpm \
     >>$(UFILENAME)");

     //Tools required
     demo/ave/rbuttons
     demo/ave/lbuttons
     demo/ave/lcheckboxes
     demo/ave/lists
     demo/ave/textfields
     demo/ave/textareas
     demo/ave/choices
     demo/ave/menus
     demo/ave/filedialogs
     demo/ave/viewimg
     demo/ave/viewflm
     demo/ave/blend
     demo/ave/clock
     demo/ave/font
     demo/ave/rgbmask
     demo/ave/vgrad
     demo/ave/boing
     demo/ave/freeball
     demo/ave/table

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)


//APPS stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_APPS)

     #ifdef WANT_APP_AUTOSTART
          .spawn(tool="demo/ave/rbuttons",cmdline="rbuttons");
          .spawn(tool="demo/ave/lbuttons",cmdline="lbuttons");
          .spawn(tool="demo/ave/lcheckboxes",cmdline="lcheckboxes");
          .spawn(tool="demo/ave/lists",cmdline="lists");
          .spawn(tool="demo/ave/textfields",cmdline="textfields");
          .spawn(tool="demo/ave/textareas",cmdline="textareas");
          .spawn(tool="demo/ave/choices",cmdline="choices");
          .spawn(tool="demo/ave/menus",cmdline="menus");
          .spawn(tool="demo/ave/filedialogs",cmdline="filedialogs");
          .spawn(tool="demo/ave/viewimg",cmdline="viewimg -t0
/images/rings.cpm");
          .spawn(tool="demo/ave/viewflm",cmdline="viewflm -t0 -d100
/images/cradle.flm");
          .spawn(tool="demo/ave/blend",cmdline="blend");
          .spawn(tool="demo/ave/clock",cmdline="clock");
          .spawn(tool="demo/ave/font",cmdline="font -s*1 -nMMincho -p24");
          .spawn(tool="demo/ave/font",cmdline="font -s*4 -nMMincho -p24");
          .spawn(tool="demo/ave/font",cmdline="font -s*3 -nMMincho -p48");
          .spawn(tool="demo/ave/font",cmdline="font -s*2 -nMMincho -p72");
          .spawn(tool="demo/ave/font",cmdline="font -nGlowworm -p24");
          .spawn(tool="demo/ave/font",cmdline="font -nCurlz -p24");
          .spawn(tool="demo/ave/font",cmdline="font -p24 -i");
          .spawn(tool="demo/ave/rgbmask",cmdline="rgbmask");
          .spawn(tool="demo/ave/vgrad",cmdline="vgrad");
          .spawn(tool="demo/ave/boing",cmdline="boing");
          .spawn(tool="demo/ave/freeball",cmdline="freeball");
     #endif

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_APPS)
```

5.2. Example 2

This file relates to a number of Java-based games, for each of which a different set of tools and data files need to be specified. The ‘romtools’ stage for this file is divided into the lists of tools and data files required for each of the games Dugout, Warp and Iceblox. This file does not include a ‘depends’ stage or a ‘ramtools’ stage.

```
//Setup Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)

      // sysgen image filename
      #ifndef IMAGE_FILE
            #define IMAGE_FILE hornellgames
      #endif

      #redefine WANT_FILESYSTEM

      #redefine WANT_MINIMAL_PJAVA

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)


//ROMTOOLS Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)

      //Dugout

      //Data Files Required
      .system ("echo \
      hornell/dugout/dugout.gif \
      hornell/dugout/dugout0.au \
      hornell/dugout/dugout1.au \
      hornell/dugout/dugout2.au \
      hornell/dugout/dugout.html \
      app/start/hornell/dugout.scr \
      >>$(UFILENAME)");

      //Tools required
      .system ("echo \
      hornell/dugout/dugout.class \
      >>$(CFILENAME)");


      //Warp

      //Data Files Required
      .system ("echo \
      hornell/warp/warp.html \
      hornell/warp/warp0.gif \
      hornell/warp/warp1.gif \
      hornell/warp/warp2.gif \
      hornell/warp/warp3.gif \
      hornell/warp/warp4.gif \
      hornell/warp/warp5.gif \
      hornell/warp/warp6.gif \
      hornell/warp/warp7.gif \
      hornell/warp/warp8.gif \
      hornell/warp/warp9.gif \
      hornell/warp/warpsnd0.au \
      hornell/warp/warpsnd1.au \
      hornell/warp/warpsnd2.au \
      app/start/hornell/warp.scr \
      >>$(UFILENAME)");

      //Tools required
      .system ("echo \
      hornell/warp/warp.class \
      >>$(CFILENAME)");


      //IceBlox

      //Data Files Required
      .system ("echo \
      hornell/iceblox/iceblox.html \
      hornell/iceblox/iceblox.gif \
      app/start/hornell/iceblox.scr \
      >>$(UFILENAME)");

      //Tools required
      .system ("echo \
      hornell/iceblox/iceblox.class \
      >>$(CFILENAME)");


#endif // $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)


//APPS Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_APPS)

      #ifdef WANT_APP_AUTOSTART
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify
com.tao_group.applet.ElateAppletViewer
hornell/dugout/dugout.html",stdin="device/null",stdout="device/null",stderr
="device/error");
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify
com.tao_group.applet.ElateAppletViewer
hornell/iceblox/iceblox.html",stdin="device/null",stdout="device/null",stde
rr="device/error");
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify
com.tao_group.applet.ElateAppletViewer
hornell/warp/warp.html",stdin="device/null",stdout="device/null",stderr="de
vice/error");
      #endif

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_APPS)
```

5.3. Example 3

The following application system configuration file relates to two Java-technology-based demonstration programs, Molecule Viewer and Wireframe. As can be seen below, the ‘romtools’ stage of the file is divided between the lists of tools and data files for Molecule Viewer, and those for Wireframe. Once again, this file does not include a ‘depends’ stage or a ‘ramtools’ stage.

```
//SETUP Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)

      // sysgen image filename
      #ifndef IMAGE_FILE
            #define IMAGE_FILE jdkdemos
      #endif

      #redefine WANT_FILESYSTEM

      #redefine WANT_MINIMAL_PJAVA

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_SETUP)
```

```
//ROMTOOLS Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)

      //Molecule Viewer

      //Data Files Required
      .system ("echo \
      sun/applets/MoleculeViewer/Matrix3D.java \
      sun/applets/MoleculeViewer/XYZApp.java \
      sun/applets/MoleculeViewer/example1.html \
      sun/applets/MoleculeViewer/example2.html \
      sun/applets/MoleculeViewer/example3.html \
      sun/applets/MoleculeViewer/index.html \
      sun/applets/MoleculeViewer/models/benzene.xyz \
      sun/applets/MoleculeViewer/models/buckminsterfullerine.xyz \
      sun/applets/MoleculeViewer/models/HyaluronicAcid.xyz \
      sun/applets/MoleculeViewer/models/cyclohexane.xyz \
      sun/applets/MoleculeViewer/models/ethane.xyz \
      sun/applets/MoleculeViewer/models/water.xyz \
      app/start/sun/applets/moleculeviewer.scr \
      >>$(UFILENAME)");

      //Tools required
      .system ("echo \
      sun/applets/MoleculeViewer/Matrix3D.class \
      sun/applets/MoleculeViewer/XYZChemModel.class \
      sun/applets/MoleculeViewer/XYZApp.class \
      sun/applets/MoleculeViewer/Atom.class \
      >>$(CFILENAME)");

      //WireFrame

      //Data Files Required
      .system ("echo \
      sun/applets/WireFrame/Matrix3D.java \
      sun/applets/WireFrame/ThreeD.java \
      sun/applets/WireFrame/example1.html \
      sun/applets/WireFrame/example2.html \
      sun/applets/WireFrame/example3.html \
      sun/applets/WireFrame/example4.html \
      sun/applets/WireFrame/index.html \
      sun/applets/WireFrame/models/cube.obj \
      sun/applets/WireFrame/models/dinasaur.obj \
      sun/applets/WireFrame/models/hughes_500.obj \
      sun/applets/WireFrame/models/knoxS.obj \
      app/start/sun/applets/wireframe.scr \
      >>$(UFILENAME)");

      //Tools required
      .system ("echo \
      sun/applets/WireFrame/Matrix3D.class \
      sun/applets/WireFrame/FileFormatException.class \
      sun/applets/WireFrame/Model3D.class \
      sun/applets/WireFrame/ThreeD.class \
      >>$(CFILENAME)");

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_ROMTOOLS)
```

```
//APPS Stage

#if $(BUILD_STAGE) == $(BUILD_STAGE_APPS)

      #ifdef WANT_APP_AUTOSTART
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify 
com.tao_group.applet.ElateAppletViewer 
sun/applets/MoleculeViewer/example1.html",stdin="device/null",stdout="devic
e/null",stderr="device/error");
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify 
com.tao_group.applet.ElateAppletViewer 
sun/applets/MoleculeViewer/example2.html",stdin="device/null",stdout="devic
e/null",stderr="device/error");
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify 
com.tao_group.applet.ElateAppletViewer 
sun/applets/WireFrame/example2.html",stdin="device/null",stdout="device/nul
l",stderr="device/error");
            .spawn(tool="app/stdio/jcode",cmdline="jcode -noverify 
com.tao_group.applet.ElateAppletViewer 
sun/applets/WireFrame/example3.html",stdin="device/null",stdout="device/nul
l",stderr="device/error");
      #endif

#endif // $(BUILD_STAGE) == $(BUILD_STAGE_APPS)
```