Inter-Office Memorandum

To	File	Date	November 11, 1982
From	Roy Levin	Location	PARC/ CSL
Subject	MakeBoot		

XEROX

Release as [Indigo]<Cedar>Documentation>MakeBoot.press
Draft [Indigo]<PreCedar>MakeBoot>MakeBoot.tioga
Last edited by Levin on November 11, 1982 11:12 am

MakeBoot is a program that constructs a boot file suitable for installation on a Pilot logical volume. The version of MakeBoot described in this document runs in the Cedar environment. MakeBoot subsumes the Rubicon version of StartPilot, which has similar functionality but executes in the Alto/Mesa environment. MakeBoot as described here is a functional extension of the version available on [Igor]<PreCascade> and is suitable for constructing boot files containing Cedar programs.

Introduction

A boot file is essentially a "virtual execution environment": it consists of a memory image containing a number of BCDs that have been loaded but not started. The simple view of MakeBoot is that it takes a collection of specified BCDs, constructs a memory image, and writes it out as a boot file. In practice, however, MakeBoot requires more information than just the names of the BCDs to be included in the memory image; this information is contained in a text file (or files) called the *command input*. The command input includes a number of parameters that affect the way the boot file is constructed.

The memory image built by MakeBoot is loaded into memory by a simple loader called the *germ*, which then transfers control to the main body of a module named in the command input. Naturally, there is a certain amount of delicate initialization required to transform the loaded memory image into a running Pilot environment. The details of this initialization include configuration of memory, which, of course, includes dealing with the contents of the initial memory image (boot file) that the germ has loaded. For example, certain portions of the code in the boot file need to be initially resident in main memory, since they will be needed before Pilot's machinery for swapping from disk has been set up. The germ and the Pilot initialization code have to cooperate with MakeBoot to insure that memory residency, and other properties, are preserved during the delicate initialization phase. To do this, they share information contained in the boot file, information which is constructed from the BCDs that make up the memory image and the command input. MakeBoot relies almost entirely on its input for this information; very little is implicitly assumed.

The descriptions below assume certain knowledge of the structure of BCDs, which we summarize here. A .bcd file, which is the output of the compiler, binder, or packager, consists of three related parts: a descriptive portion called the "binary configuration description" (BCD), a (possibly empty) collection of code segments, and a (possibly empty) collection of symbol segments. The descriptive portion includes

tables that describe a hierarchy of configurations and the modules they contain, the interfaces imported and exported by the root configuration of the hierarchy, and the packaging of the code and global frames that comprise the modules in the hierarchy. The BCD also describes the location of the code segments and their associated symbols, each of which may be either in the same file with the descriptive portion or in a different one. (There are other, subsidiary tables in the BCD that are not used by MakeBoot.)

Command Input

This section describes the syntax and semantics of the command input. Many of the possible specifications should be used only upon the advice and counsel of a wizard, since they may cause extremely strange behavior in the initialization or subsequent execution of the boot file. Items of command input that fall in this category have their semantic descriptions preceded by (*).

Syntax

MakeBoot treats its command input as a linear text stream, which may include white space and comments in the same form and places as Mesa source code. In the syntax equations that follow, white space is not significant and a "|" character denotes metalinguistic alternatives. Words in italics are metalinguistic variables (non-terminals), all other characters appear literally in the command input. (No definitions appear for *identifier* or non-terminals beginning with *decimalNumber*; these are presumed to be implicitly understood. Note that, as with all Mesa-related descriptions, the case of alphabetic characters within identifiers is significant.)

```
input              ::= inputItem | input inputItem
inputItem          ::=  gftItem |
                        mdsBaseItem | codeBaseItem |
                        pdaBaseItem | pdaPagesItem |
                        framePagesItem | frameWeightItem |
                        nProcessesItem |
                        svItem | svSizeItem |
                        wartItem | noTrapItem |
                        residentItem | resDescItem | initiallyInItem
gftItem            ::=  GFT : decimalNumberIn[1..1024] ;
mdsBaseItem        ::=  MDSBASE : pageNumber ;
codeBaseItem       ::=  CODEBASE : pageNumber ;
pdaBaseItem        ::=  PDABASE : pageNumber ;
pdaPagesItem       ::=  PDAPAGES : decimalNumberIn[1..256] ;
framePagesItem     ::=  FRAMEPAGES : decimalNumberIn[0..256) ;
frameWeightItem    ::=  frameSpec : fsi , decimalNumber fsiChain ;
frameSpec          ::=  FRAMEWEIGHT | FRAMECOUNT ;
fsiChain           ::=  empty | , fsi
nProcessesItem     ::=  PROCESSES : decimalNumberIn[1..1024] ;
svItem             ::=  STATEVECTORCOUNT : priority, decimalNumber ;
svSizeItem         ::=  STATEVECTORSIZE : decimalNumber ;
wartItem           ::=  WART : wartModule ;
noTrapItem         ::=  NOTRAP : moduleList ;
residentItem       ::=  RESIDENT : generalList ;
resDescItem        ::=  RESIDENTDESCRIPTOR : generalList ;
```

```
initiallyInItem     ::=  IN : generalList ;
generalList         ::=  listItem | generalList , listItem
listItem            ::=  configPart | CODE [ configPartList ] |
                         GLOBALFRAME [ configPartList ] |
                         SPACE [ nameList ] | CODEPACK [ nameList ] |
                         FRAME [ nameList ] | FRAMEPACK [ nameList ] |
                         BCD [ nameList ]
configPartList      ::=  configPart | configPartList , configPart
configPart          ::=  module | configName [ moduleList ]
moduleList          ::=  module | moduleList , module
module              ::=  moduleName | moduleName . instanceName
nameList            ::=  name | nameList , name
configName          ::=  name
moduleName          ::=  name
instanceName        ::=  name
name                ::=  identifier
pageNumber          ::=  decimalNumberIn[0..65536)
fsi                 ::=  decimalNumberIn[0..256)
priority            ::=  decimalNumberIn[0..8)
decimalNumber       ::=  decimalNumberIn[0..65536)
wartModule          ::=  moduleName | configName [ moduleName ]
empty               ::=
```

Semantics

gftItem ::= GFT : *decimalNumberIn[1..1024]* ;

The argument specifies the number of entries to be allocated to the global frame table. Default value: 512.

mdsBaseItem ::= MDSBASE : *pageNumber* ;

(*) The argument specifies the starting virtual page number of the MDS. (The architecture requires that this value be 0 mod 256.) Default value: 512.

codeBaseItem ::= CODEBASE : *pageNumber* ;

(*) The argument specifies the first virtual page number that MakeBoot will use to load non-MDS data and code (exclusive of data with other architectural constraints, such as the process data area). Default value: 768.

pdaBaseItem ::= PDABASE : *pageNumber* ;

(*) Obsolete. The process data area always begins at virtual page 256.

pdaPagesItem ::= PDAPAGES : *decimalNumberIn[1..256]* ;

The argument specifies the number of pages of memory to allocate for the process data area. Since the process data area contains state vectors and process state blocks, the value supplied here interacts with the values for the state vector counts and number of processes supplied elsewhere. Sufficient PDA space will always be allocated to accommodate the state vector and process counts; if the PDAPAGES parameter is large enough to permit additional pages to be allocated, they will be used for process state blocks. Default value: 1.

framePagesItem ::= FRAMEPAGES : *decimalNumberIn[0..256)* ;

(*) The argument specifies the number of pages of MDS to be allocated to the initial frame heap. See the discussion of *frameWeightItem*s, below, before attempting to change this specification. Default value: 10.

frameWeightItem ::= *frameSpec* : *fsi , decimalNumber fsiChain* ;

(*) The distribution of frame sizes in the initial frame heap is controlled by a sequence of specifications, introduced by either FRAMEWEIGHT or FRAMECOUNT (the two are synonyms). The *decimalNumber* is the minimum number of frames of the given *fsi* to be allocated in the initial frame heap. If, after allocating these frames, the specified number of FRAMEPAGES has not been consumed, MakeBoot allocates the remaining space by interpreting the *decimalNumber*s as relative weights for the *fsi*s. If present, the *fsiChain* determines the strategy to be used by the frame allocator at runtime when no frames are available for the given *fsi*. If *fsiChain* <= *fsi*, an allocation fault will occur. If *fsiChain* > *fsi*, the allocator attempt to allocate a frame with frame size index *fsiChain*. Default values:

FSI	Weight	FsiChain
0	9	1
1	13	2
2	9	3
3	8	4
4	7	5
5	6	6
6	4	7
7	2	8
8	2	9
9	1	10
10	1	11
11	1	0
12	1	13
13	1	14
14	1	15
15	1	16
16	1	17
17	1	18
18	1	0
others	0	0

nProcessesItem ::= PROCESSES : *decimalNumberIn[1..1024]* ;

The argument specifies the number of processes to be created in the initial process data area. (Actually, this is the minimum number of processes; see the description of PDAPAGES.) Default value: 50.

svItem ::= STATEVECTORCOUNT : *priority, decimalNumber* ;

(*) The distribution of state vectors across priority levels is controlled by a sequence of STATEVECTORCOUNT entries. Default values:

Priority	Count
0	1
1	1
2	2
3	1
4	1
5	3
6	1
7	1

svSizeItem ::= STATEVECTORSIZE : *decimalNumber* ;

(*) The argument specifies the size in words of the state vectors allocated in the process data area. Default: 16.

wartItem ::= WART : *wartModule* ;

(*) The argument specifies the module that is to receive the initial transfer of control from the germ after the boot file has been loaded into memory. There is no default value.

residentItem ::= RESIDENT : *generalList* ;
resDescItem ::= RESIDENTDESCRIPTOR : *generalList* ;
initiallyInItem ::= IN : *generalList* ;

These specifications control the swapping characteristics of the portions of the boot file that come from the input BCDs. Each such portion can be viewed as a *segment*, that is, a sequence of consecutive pages in the boot file. All segments occupy virtual memory, but these specifications affect the existence and lifetime of real memory associated with them. A segment that is specified to be RESIDENT will be present in real memory when the boot file is loaded by the germ, and Pilot initialization will assume that it is to remain in memory indefinitely. That is, it will not be possible to swap the segment out of main memory. A segment that is specified to be IN will also be present in real memory when the boot file is loaded, but will be eligible for swapping after initialization is complete. A segment that is specified to be RESIDENTDESCRIPTOR will not necessarily be present in real memory at load time, but the information necessary to swap it in will be made resident. The naming of particular segments in the input is accomplished by the following *listItem* s.

configPart
CODE [*configPartList*]

These synonymous specifications identify unpackaged code segments within the input BCDs. Note from the syntax equations that a *configPart* may include a single level of configuration name to resolve ambiguities in module naming; however, configuration names must be unambiguous across the entire set of input BCDs. Alternatively, a module naming ambiguity (but not a configuration naming ambiguity) may be resolved by the instance names supplied in the C/Mesa input when the relevant BCD was bound. If a *configPart* is of the form *configName* [ALL], all unpackaged code segments within the indicated configuration will be matched by this specification.

Note on PACKed code: If a module name appearing in this specification corresponds to a module whose code is PACKed (by the Binder) with the code of other modules, the specification applies to the entire code segment, not just the portion that belongs to the named module. If other modules whose code is part of the same segment have conflicting specifications in the MakeBoot command input, the result is undefined. Note that this applies only to code *packed* by the Binder; for *packaged* code segments produced by the Packager, see the description of the CODEPACK specification, below.

GLOBALFRAME [*configPartList*]

This specification identifies unpackaged global frames, and is in all ways analogous to the CODE[ ] specification above.

SPACE [*nameList*]
CODEPACK [*nameList*]

These synonymous specifications identify packaged code within the input BCDs. The names in the *nameList* are CODE PACKs in the Packager's terminology. (The CODEPACK form is generally preferrable to the SPACE form, to avoid any confusion with Pilot's notion of spaces.) No name qualification facilities are provided, so code pack names must be unambiguous across all input BCDs. If, instead of the *nameList*, the reserved word ALL appears within the brackets, all code packs in all input BCDs will be matched by this specification.

`FRAME [ nameList ]`
`FRAMEPACK [ nameList ]`

These synonymous specifications identify packaged global frames within the input BCDs. The names in the *nameList* are FRAME PACKs in the Packager's terminology. No name qualification facilities are provided, so frame pack names must be unambiguous across all input BCDs. If, instead of the *nameList*, the reserved word `ALL` appears within the brackets, all frame packs in all input BCDs will be matched by this specification.

`BCD [ nameList ]`

This specification identifies the descriptive portions of the input BCDs. The names in the *nameList* are configurations (but not necessarily top-level ones) within the input BCDs. If, instead of the *nameList*, the reserved word `ALL` appears within the brackets, all input BCDs will be matched by this specification.

`noTrapItem ::= NOTRAP : moduleList ;`

(*) The modules in this specification will have their initial code traps suppressed, implying that explicit arrangements must be made for starting them properly.

MakeBoot Operation

MakeBoot is invoked from the executive by loading MakeBoot.bcd and supplying a command line. The syntax of the command line resembles that of the compiler and binder:

`[bootFile: f1, loadMap: f2] _ input[key1: value1, ..., keyn: valuen]/switches`

The only mandatory part of the command line is `input`, which names a BCD (default extension `.bcd`). The `bootFile` specification supplies the file name for the boot file (default extension `.boot` or `.germ`, as appropriate); the `loadMap` specification supplies the file name for the load map file (default extension, `.loadmap`). Defaulting occurs in the usual way if either or both of these specifications are omitted: f_1 is used as the root of names if it is present, otherwise `input` is used. As is customary, the order of the keyword parameters is irrelevant. The following keywords are recognized within the brackets following `input`:

`Parm: commandInputFileName`

The indicated file (default extension `.bootmesa`) contains command input. All command input files are read and parsed before MakeBoot processes any input BCDs, including `inputFile`. The command input may be distributed arbitrarily among command input files, except that a single *inputItem* may not span a file boundary. MakeBoot effectively concatenates all command input into a single stream in the order that they appear as parameters on the command line. In general, it is a good idea to have one command input file associated with each input BCD file (unless, of course, a particular input BCD requires no command input). Also, it is generally meaningless to have no command input files at all, since MakeBoot cannot provide reasonable defaults for everything.

`BCD: bcdFileName`

The indicated file (default extension `.bcd`) is to be loaded as part of the memory image. There is no particular semantic distinction between BCDs specified as keyword parameters and the `inputFile`; the syntactic distinction on the command line is essentially historical. The BCDs are loaded in the order that they appear on the command line.

`nProcesses: decimalNumberIn[1..1024]`

This value of this parameter overrides the `PROCESSES` specification in the command input.

`gftLength: decimalNumberIn[1..1024]`

This value of this parameter overrides the `GFT` specification in the command input.

MakeBoot recognizes several switches, all of which are intended for wizards only:

D	include vast amounts of debugging output in the loadmap file
E	in addition to the normal output boot or germ file, write an ether-bootable version with the same file name and extension either `.pb` or `.eg` (boot file or germ)
G	write the output in germ file format instead of boot file format
H	print numeric values in the loadmap in hexadecimal
!	call the debugger before doing anything

MakeBoot Limitations, Restrictions, Subtleties, and "Gotchas"

Building boot files is a tricky business, and the usual effect of making a mistake in the command input is that the boot file simply won't initialize. This, in turn, usually means you can't get to the debugger, so tracking down these mistakes is difficult. This section, while not strictly about MakeBoot, combines collected wisdom, experience, and folklore about building boot files that may help you avoid some of the pitfalls and long debugging sessions. Be sure to read it thoroughly before trying to use MakeBoot.

1) Limitations in Pilot initialization currently require that anything that is `RESIDENTDESCRIPTOR` must also be `IN`. In general, if you are building a boot file based on UtilityPilot and your `RESIDENTDESCRIPTOR` and `IN` specifications are not identical, you better know what you are doing. An uncaught signal in the vicinity of PilotControl.CreateParent (usually the signal comes from a few levels deeper and is likely to be Space.Error[invalidParameters] in UtilitySpaceImpl.CreateInternal) is a strong hint that you don't know what you're doing.

2) `RESIDENT` means resident, and Pilot will not let you apply SpecialSpace.MakeSwappable to something that was specified as `RESIDENT`. This restriction may be lifted in the future, but don't count on it.

3) MakeBoot has no analogue of the runtime loader's /L switch for code links; all input is implicitly loaded as though /L were specified. This is almost always the right thing, but can lead to subtle binding problems if frame links were logically needed (they almost never are).

4) Having noted that code links are almost always good, we should hasten to add that `RESIDENT` code and unresolved code links don't get along. If some code segment in your boot file has code links, some of which are unresolved after the boot file has been made, the world will cave in if you attempt to load (using the runtime loader) a BCD that resolves one or more of those links. There is no good reason why this should be so, it's just a limitation of the present Pilot virtual memory implementation. You will know this has bitten you when, during loading, you get the uncaught signal CachedRegionImplB.Bug[makeWritableButNotSwappable].

5) While we're on the subject of code links, you should understand that code links are actually written to disk with the (otherwise readonly) code. This is not a problem for BCDs outside the boot file, because the runtime loader clears the link area when the BCD is subsequently (re)loaded. However, for (swappable) code in the boot file, code links are not cleaned up at boot time. This means that if the links have been resolved by dynamic loading, they will be garbage upon restart. Furthermore, Runtime.IsBound applied to such a link will give the wrong answer. This may be fixed sometime in the future, but in the meantime, beware. (To encourage you to think carefully about such links, MakeBoot will type out a warning message about each module that has code links and unbound imports. Each such import will be reported as `InterfaceName[nnn]`, where `nnn` is the "interface item number", as in the Binder. Consult a wizard if you don't know how to interpret such things.)

6) As you have doubtless noticed, MakeBoot's command input syntax is rather idiosyncratic and doesn't fit cleanly with Mesa or C/Mesa. The reasons are entirely historical, and compatibility requirements (and inertia) hinder change. Not only is the syntax a bit strange, but the parser

is less than wonderful, so syntax error messages may be a little cryptic. However, the source position where the parser got confused (which appears in square brackets in most error messages) is rarely far from the actual trouble spot. Hunt around, and look carefully at the syntax equations if you are stumped.

7) MakeBoot's global frame allocation algorithm is almost the same as the one employed by the runtime loader. If an input BCD is a single module, MakeBoot will attempt to allocate its global frame in the frame heap. If this fails, or if the input BCD consists of more than one module, its global frames are allocated in segments consisting of an integral number of MDS pages. Each frame pack has its own segment, and there is one segment for all unpackaged global frames. All of this is identical to the runtime loader's behavior under similar circumstances. However, if the frame segment is specified to be `RESIDENT`, MakeBoot will add any left-over space in the last page of the segment to the frame heap, breaking it up according to the frame weight ratios in the command input. This is because MakeBoot assumes that the frame segment will always be resident (see note 2 above); consequently it is safe to add the unused space to the (resident) frame heap. The runtime loader never does this.

8) As observed above, unpackaged global frames of a single (top-level) configuration are always bundled together in a single segment. This implies that a GLOBALFRAME[ ] specification for one of them will apply to all. If conflicting GLOBALFRAME[ ] specifications are present in the command input, the result is undefined.