The Format of BLITZ Object and Executable Files

Harry H. Porter III
Computer Science Department
Portland State University

Abstract

This document describes the format of "object files" and "executable files" as produced and used by the BLITZ tools.

Terminology and Files

The BLITZ assembler takes as input a single text file. This is called a source file. The source file should have a name ending in ".s". For example:

```
simple.s
```

The assembler produces a single file, known as an "object file". This file should have a name ending in ".o". For example:

```
simple.o
```

The object file is used as input to the BLITZ linker, which produces an "executable file". The linker will take one or more object files, and will produce a single executable file. The executable file will be named "a.out" by default, but it is common for the executable file to be renamed. Often the name of the executable file is the same as one of the original source files, after removing the ".o". For example:

```
simple
```

The executable file may then be loaded by an operating system and executed. Therefore, it will contain all that is necessary for executing the program.

The BLITZ assembler tool is called "asm" and the BLITZ linker tool is called "lddd". Another BLITZ tool, called "dumpObj", can be used to print out, in a human readable form, either object or executable files.

The Object File

The object file has the following format, which will be discussed in detail in the sections below. The file can be considered as series of fields; the length of the fields is given in the left-hand column here.

```
  bytes  description
  =====  ===========
    4    magic number "BLZo"  (in hex: 0x424C5A6F)
    4    contains_entry  (0 or 1)
    4    size of text segment chunk in bytes (M)
    4    size of data segment chunk in bytes (N)
    4    size of bss segment chunk in bytes
    M    text segment chunk
    4    separator "****"  (in hex: 0x2A2A2A2A)
    N    data segment chunk
    4    separator "****"  (in hex: 0x2A2A2A2A)

The following fields are repeated once for every symbol...
    4    symbol_number (1, 2, 3, ...)
    4    value
    4    relative_to
             0=imported
             1=text
             2=data
             3=bss
             4=absolute
    4    number_of_characters (L)
    L    character_string

After all symbols...
    4    zero to terminate (in hex: 0x00000000)
    4    separator "****"  (in hex: 0x2A2A2A2A)

The following fields are repeated once for every reloc_entry...
    4    type
           1 = 8-bits (loc=addr of the byte)
           2 = 16-bits (loc=addr of the half-word)
           3 = 24-bits (pc-rel, loc=addr of the instruction)
           4 = 32-bits (loc=addr of the word)
           5 = sethi (use hi 16 bits)
           6 = setlo (use lo 16 bits)
           7 = ldaddr (pc-rel, loc=addr of the instruction)
    4    location_to_update
    4    1=in_text, 2=in_data
    4    offset
    4    relative_to
    4    source_line_number

After all reloc_entries...
    4    zero to terminate (in hex: 0x00000000)
    4    separator "****"  (in hex: 0x2A2A2A2A)

The following fields are repeated once for every label...
    4    1=in_text, 2=in_data, 3=in_bss
    4    offset
    4    number_of_characters (K)
    K    character_string

After all labels...
    4    zero to terminate (in hex: 0x00000000)
    4    separator "****"  (in hex: 0x2A2A2A2A)
```

Magic Number

The first four bytes of the object file serve to identify it as a BLITZ object file. These bytes are the ASCII character codes for the letters "BLZo" (for BLitZ Object).

The magic number idea is not a surefire way to identify files, since there may be other files that happen to begin with these same four bytes, but it does provide a fairly good way for the linker to check that it is being given a meaningful file. Also, it allows a human looking at the file to guess what sort of data it contains. Although much of the file will contain bytes that are not interpretable as ASCII data, the four bytes of the magic number can be, so they should give the reader a clue about the file's nature.

Integers

All integers in the object file are stored as 32-bit signed values. They are stored in Big Endian order, i.e., the most significant byte will appear first.

Contains–Entry

The final executable program must have a single "entry point", which is the location of the first instruction to be executed; the program must begin execution somewhere. The entry point is indicated in a source file by the use of a special label, which must be spelled exactly as follows:

```
_entry:
```

This label must appear in exactly one of the source files. It identifies the location in memory for the operating system to branch to when the program is loaded and run.

The linker will combine several object files to produce an executable file. Exactly one of the object files must have been assembled from a source file containing a label "_entry". The object file containing the entry point will contain a 1 in this field; all other object files will contain 0.

The entry location is the first byte of the text segment in this object file.

Segments versus Segment Chunks

The final executable file will have a single text segment. Each object file has a text "chunk", i.e., a piece of the final text segment. The linker will concatenate all text chunks to produce the final text segment. When the linker concatenates text chunks, it will not insert any additional bytes. However, the final text segment will have its size rounded up to a multiple of the page size, by inserting additional padding bytes of zero after the last text chunk. Each text chunk will begin on a word boundary.

Likewise, each object file contains a single data segment chunk. The linker will concatenate these to form the final data segment in the executable file. Also each of the object files will describe a bss segment chunk and the linker will concatenate these to form the bss segment in the executable file.

The same rounding applies to the text, data, and bss segments and chunks. All chunks will be a multiple of four bytes long. The chunks will be concatenated with no intervening bytes to form a segment. The segment will then be rounded up to the next multiple of the page size.

When the linker concatenates the segment chunks to produce the complete segments, it will begin by ordering the files. The linker will then concatenate the chunks in this order. The object files will be used in the order they are given to the linker, with the exception that the object file containing the entry point will be moved to the front of the list. This has the effect of ensuring that the entry point, which is the first byte of the first text chunk, will be placed at the first byte of the complete text segment.

Sizes of the Segment Chunks

The next three words give the sizes of the segment chunks contained in the object file. These numbers need not be multiples of 4; each chunk will be rounded up to a multiple of 4 bytes. Even though the object file may contain a text or data chunk with an odd number of bytes, the linker will pad the chunk with 0 to 3 bytes to round it up to a multiple of 4 bytes.

The Text Segment Chunk

Next in the object file will be M bytes, where M is the size of the text segment chunk. Since M may not be a multiple of 4, the remaining integers in the file may not be "word aligned" in the file.

Separators

As an internal consistency check, there will be 4 bytes of "separator" data placed at the indicated points in the file. These four bytes are the ASCII character codes for the characters "****". That is, the separator word is 0x2A2A2A2A. If there is some inconsistency between the text or data segment sizes and the actual number of bytes provided, then these separators may help to catch the error. The linker will check that the separator characters appear correctly at the places in the object file where they are supposed to appear, and print error messages if not.

The Data Segment Chunk

Next in the object file will be N bytes, where N is the size of the data segment chunk.

Occasionally the data segment chunk will not used and will have a size of zero.

The BSS Segment Chunk

The bss segment will be initialized to contain all zeros. Therefore, there are no data bytes provided in the object file. Each object file contains a bss segment chunk, as indicated by the "bss segment chunk size", but no bytes will ever be provided since they are all implicitly zero.

Often the bss segment chunk in an object will not used and will therefore have a size of zero. Often none of the object files will use a bss chunk, so the entire bss segment will end up, after linking, with a size of zero.

Symbols in the Object File

A single program may originate from several source files. Each source file will be assembled into an object file. These object files will then be combined in the linking phase to produce a single executable file.

The program can be thought of a being composed of several "modules". Each module corresponds to a single source file. The linking process then combines the object files from each of the modules to produce the executable file.

Code in one module may refer to addresses, instructions, data, and values defined in other modules. As an example, module A may define a routine called "printf" and module B may call this routine. When module B is assembled, there is no information about where the "printf" routine will be located or even what module it will be in. As the linker processes all modules, it will modify the "call" instruction to fill in the final address of the "printf" routine, in a process called "relocation".

Symbols are used to share such things as the address of the "printf" routine across module boundaries. Module A would export the symbol "printf" and module B would import "printf".

Each symbol has a value. This value may be (1) the address of a location in the text segment, (2) the address of a location in the data segment, (3) the address of a location in the bss segment, or (4) an absolute value, which is not the address of any location.

In the first three cases, the actual value of the symbol cannot be determined until the linking phase, since we will not know the actual addresses of the chunks until the linking phase. The address of a location is subject to change by the linker; it is the linker that assigns addresses.

The Symbol List

The next section of the object file consists of a number of symbols. For example, an object file may contain 100 symbols. Each of the 100 symbols is represented in the object file with a "symbol entry", which will have information such as "symbol_number", "value", "relative_to", and the characters of the symbol.

Symbol Numbers

Each symbol in an object file is numbered sequentially, starting with 1. Since there is no symbol with number 0, a zero in this field is used as a flag to indicate the end of the list. Or to put it another way, the list of symbol entries will be followed by a word containing the integer zero, signaling the end of the list.

These symbol numbers are unique to a single object file. In other words, the numbers used in one file are in no way related to the numbers used in another object file. For example, the symbol "printf" my be assigned number 5 in one object file and number 8 in another file. These two object files might then be linked together later as part of a single program. The linker will identify the "printf" symbol in one module as the same "printf" symbol used in another module since they have the same spelling.

A symbol number is used to reference the symbol from other symbol entries and from relocation entries, but only within the same object file.

The first 4 symbols are predefined and have special meanings. These are:

```
Number    Symbol
======  =========
  1      .text
  2      .data
  3      .bss
  4      .absolute
```

Conceptually, the symbols ".text", ".data", and ".bss" name the addresses of the first byte in the corresponding segment chunks. The actual locations will not be known until the linker determines where to place the chunks when laying out the executable. The ".absolute" symbol is a constant of zero and is used to indicate a value that does not require relocation.

Symbol Values

Each symbol will have a 32-bit value.

In some cases, this value is known by the assembler; in other cases it is not. Consider the following piece of module B, which defines the symbols "cons", myvar", and "mysym" and which uses the symbol "printf" defined by module A.

```
        .import  printf
        .export  cons
        .export  myvar
        .export  mysym
        ...
        call     printf
        ...
cons    =        1234
mysym   =        printf+8
        .data
myvar:  .word    0
```

In this example, the symbol "cons" is known absolutely by the assembler. It has a value (namely 1234), and that value is absolute, not relative to any placement of the chunks by the linker. The "value" field of the symbol entry for "cons" will contain this value.

The value of the symbol "printf" was completely unknown by the assembler when it assembled module B. The symbol "printf" is imported from another module, and will be given a value by the linker. In the object file for module B, the "value" field for "printf" will simply be zero.

The symbol "myvar" names a location in one of the segment chunks in this module. In this case, the location is in the .data chunk in this module. So, we have some information about the value of "myvar" – that it has a certain offset from the beginning of the data segment chunk – but we are also missing some information, since we do not know the final location of that chunk. In this case, the "value" field will contain the offset from the beginning of the .data chunk. The linker will compute the final value after it places this chunk in memory.

In the case of "mysym", we know that the value will be 8 greater than the final value of "printf", which will be imported from another module. In this case, the "value" field will contain 8, indicating that the linker will need to add 8 to the value of "printf" to compute the value of "mysym".

The Relative–To Field

The “relative_to” field gives additional information about the symbol, indicating what its value is relative to.

In the case of a symbol like “cons”, the relative_to field will be 4, to indicate that it is an absolute value.

In the case of a symbol like “printf”, the relative_to field will be 0, to indicate that it is an imported symbol and that we have no information in this object file about its value.

In the case of a symbol like “myvar”, the relative_to field will be 2, to indicate that the value given is relative to the beginning of this data segment chunk. In other words, when the linker finally decides where to place this chunk, it should then add the “value” field to compute the actual address of “myvar”.

In the case of a symbol like “mysym”, the relative_to field will indicate some other symbol. In this case, it will be the symbol number of the symbol “printf”. The linker will then need to add 8 to whatever value “printf” is given to compute the value of “mysym”.

String Length and Characters

The next field will give the size in characters of the symbol. This will be followed by that many ASCII characters. Note that there will be no terminating ASCII “null” character, unlike the typical convention in “C”.

Relocation Entries

Next in the file is a list of “relocation entries”. Each relocation entry indicates that some data (in either the .text or .data segment) must be modified by the linker. This modification can only be done after the final values of all the symbols have all been computed by the linker. After computing the values of the symbols, the linker will run through all the relocation entries and, for each, update the indicated bytes in memory.

Each relocation entry has a “type”. There are 7 types, numbered from 1 through 7. There is no type 0. A zero in this field is used as a flag to terminate the list. The types are described later.

Location to Update

The field marked “1=in text, 2=in data” indicates whether the location to be updated is in the .text or .data segment chunk. The “location_to_update” field tells where in that segment chunk. For example, one entry may indicate that we need to update the 200-th byte in the data segment chunk in this object file.

All updates will go to locations within the same object file. In other words, a relocation entry will never refer to an address in another module. All of the “location_to_update” addresses are offsets relative to the beginning of either the data chunk or the text chunk in this object file.

Relocation Type

Each relocation entry instructs the linker to update a location in memory (given by the "location_to_update" field). That address in memory will be modified to contain the data indicated by the "offset" and "relative_to" fields.

There are several different relocation operations that can be performed. The "type" field tells which sort of operation the linker must do.

Offset and Relative–To

The "offset" field gives a 32-bit value. The "relative_to" field will contain the symbol number of a symbol in the symbol list in this object file.

The value is relative to the symbol given by the "relative_to" field. The new value is computed by linker by taking the value of the "relative_to" symbol and adding the value of the offset to it.

Type 1: 8-bits

In this relocation operation, the location_to_update points to a single byte. That byte will be modified to contain the value indicated by the offset/relative_to fields.

Type 2: 16-bits

In this relocation operation, the location_to_update points to a halfword (16 bits). That halfword will be modified to contain the value indicated by the offset/relative_to fields.

Type 3: 24-bits

In this relocation operation, the location_to_update points to a word (32 bits) containing an instruction. That instruction will be a call, jump, or branch instruction. These instructions contain a 24-bit field, which is PC-relative.

Normally, a jump (or call or branch) will be from one instruction to another location (the "target address") within the same module. When this is the case (the jump is from an instruction to a location in the same segment chunk) the assembler can fully determine the instruction and no relocation entry will be needed. The assembler will compute the jump offset and place it into the instruction. The linker may move the instruction and may move the target, but if these are in the same chunk, they will always be moved together so the relative offset can never be changed by the linker.

However, if the jump is from one module to another, or from one segment chunk (like .text) to another segment chunk (like .data) within the same module, the linker will be required to fill in the 24-bit displacement field in the instruction. The assembler cannot do it.

For a relocation entry of type 3, the location_to_update will point to the instruction (to the byte containing the op-code, not to the 24-bit offset field within the instruction). The offset/relative_to contains the target of the call, jump, or branch instruction. The linker will subtract the PC from the target address to compute the relative offset and will place this in the 24-bit field in the instruction.

Type 4: 32-bits

In this relocation operation, the location_to_update points to a word (32 bits). That word will be modified to contain the value indicated by the offset/relative_to fields.

Type 5: sethi

In this relocation operation, the location_to_update will point to the 16-bit field within a "sethi" instruction (not to the op-code byte). The offset/relative_to information will contain a value. The hi-order 16 bits of that value will be moved into the sethi instruction.

Type 6: setlo

In this relocation operation, the location_to_update will point to the 16-bit field within a "setlo" instruction (not to the op-code byte). The offset/relative_to information will contain a value. The lo-order 16 bits of that value will be moved into the setlo instruction.

Type 7: ldaddr

In this relocation operation, the location_to_update will point to a "ldaddr" instruction (to the op-code byte, not to the 16-bit field within the instruction). The offset/relative_to information will point to a target address. The 16-bit field in the "ldaddr" instruction is a PC-relative field; as such its value will depend on (1) the address of the ldaddr instruction and (2) the target address to be loaded. The linker will compute the value to be placed into the 16-bit field, much like it does for a type-3 relocation operation.

Uniqueness and Consistency of Symbols

Each symbol must be defined and given a value in exactly one module. A symbol that is imported but is never exported is in error. Likewise, a symbol that is exported by two or more modules is also in error. The linker will check this.

The Label List

The next section of the object file is a list of labels. These labels name locations in the text, data, and bss segment chunks in this module. The label information is mostly for documentation. For example, the labels can be printed out when instructions are disassembled from memory, giving the user a clearer understanding of the code being disassembled.

Labels need not be unique. The same label can be used in several modules, with different values in each module.

Often there is overlap between the list of symbols and the list of labels, within a single module. For example, assume module A contains the following

```
            .export  printf
    printf:
```

In this module, "printf" is both a symbol, since it is exported, and a label, since it labels some location of memory in this module. If "printf" had not been exported, then it would appear in the object file as a label only.

In the following example, "cons" appears as a symbol but not as a label:

```
          .export  cons
cons       =       1234
```

Label Offset

Each label entry has an “offset” field and an indication of which segment chunk it occurs in. The offset is relative to the beginning of the segment chunk.

Label Character String

Each label entry contains a count of characters, followed by that many ASCII characters. Note that there will be no terminating ASCII “null” character, unlike the typical convention in “C”.

The Executable File Format

An executable file contains a program that is ready to be loaded by the operating system and executed. It contains the bytes of a text segment (which the operating system may mark as read-only), the bytes of the data segment (which the operating system must treat as read-write, since they may be updated during execution), and information about the bss segment. The bytes of the bss segment will be initialized to zero by the operating system when the program is loaded, so there is no need to include any bytes in the executable file. Although the bss segment begins after being initialized to zero, the running program may change the bytes so this segment should be marked read-write.

The operating system will begin execution of the program by branching to the first byte of the text segment. In other words, the entry point will always be the first instruction in the text segment.

The executable file also contains some information about the symbols and labels used in the program. This information is not strictly necessary for execution, but may be useful when debugging a program.

The executable file has the following format, which will be discussed below.

```
  bytes  description
  =====  ===========
    4    magic number "BLZx"  (in hex: 0x424C5A78)
    4    size of the text segment (M, a multiple of page_size)
    4    size of the data segment (N, a multiple of page_size)
    4    size of the bss segment (a multiple of page_size)
    4    text_load_addr
    4    data_load_addr
    4    bss_load_addr
    4    separator "****"  (in hex: 0x2A2A2A2A)
    M    text segment bytes
    4    separator "****"  (in hex: 0x2A2A2A2A)
    N    data segment bytes
    4    separator "****"  (in hex: 0x2A2A2A2A)

The following fields are repeated once for every label...
    4    number_of_characters (K)
    4    value
    K    character_string
    4    separator "****"  (in hex: 0x2A2A2A2A)

After all labels...
    4    zero to terminate (in hex: 0x00000000)
    4    separator "****"  (in hex: 0x2A2A2A2A)
```

Magic Number

The first four bytes of the executable file serve to identify it as a BLITZ executable file. The bytes are the ASCII character codes for the letters “BLZx”.

Sizes of the Segments

The next three fields contain the sizes (in bytes) of the text, data, and bss segments. These sizes will always be multiples of the page size.

Load Addresses

The next three fields contain the addresses at which to load the segments. These tell the operating system where in memory to put the bytes of the text and data segments. They also tell the operating system where the bss segment begins; it is the operating system’s responsibility to initialize the bytes of the bss segment to zero before execution begins.

The Text Segment

Next in the object file will be M bytes, where M is the size of the text segment.

The Data Segment

Next in the object file will be N bytes, where N is the size of the data segment. Occasionally the data segment will not be used and will have a size of zero.

Symbols versus Labels

Symbols in the object files are processed by the linker and are involved in the relocation process. Each object file has a set of symbols and these sets are merged during linking.

Each object file also has a set of labels. The set of symbols in an object file is distinct from the set of labels in that file. Each object file has its own set of labels, and the labels sets from different files are distinct. The same string of characters may occur as a symbol and as a label in one or more object files; each label may or may not be related to other labels and/or symbols and may or may not have the same value.

In the executable file (the "a.out" file) there will be a single set of labels, which we will call the "execution labels" to prevent confusion with the labels from the object files. There are no symbols in the executable file.

The linker will create the set of executable labels by combining the sets of symbols and labels from the object files. Every symbol and every label in the object files will be placed into the executable file, but because the same symbol may appear in several object files and since several object files may contain distinct labels with the same spelling, the linker does not do a simple "set union" to create the set of execution labels.

The executable file will contain a list of all execution label. Each execution label consists of a string of characters and a value. Each entry in the execution label list begins with the number of characters. Since each execution label has at least one character, this number will never be zero. A zero will follow the last entry to indicate the end of the list.

Each execution label will have an associated 32-bit value and these values will be placed into the executable file.

The execution labels in the executable file come from two places: (1) the symbols in the object modules and (2) the labels in the object files. Each symbol will become an execution label, with the exact same spelling. During linking, each symbol is assigned a final, absolute value; this value will be used as the value of the execution label. The value of the symbol may have originally been a relative address within one of the segments, or it may have been an absolute value, but this information is not carried through to the executable file. Only the absolute value remains, not an indication of whether this symbol should be interpreted as the name of a memory address, or as a constant used in some other way.

Likewise during the linking phase, each label will be assigned an absolute address when the segments are given their final locations. Each label will be added to the set of execution labels; its value will be the final address of the label, as determined by the linker.

The names of the labels in the object files may conflict with the names of symbols and labels from other object files. For example, assume that several of the object files share an symbol called

```
myExample
```

One of the files must define and export this symbol; several other object files may import and use this symbol.

Also assume that several of the object files have labels with the exact same spelling, but neither import nor export it. For these files, "myExmple" is a completely local label. These local uses of

"myExample" will be completely unrelated to the symbol "myExample" used in other object files.

When the linker builds the executable file, it will modify the labels as necessary to ensure that all execution labels are unique. To achieve this, the linker will modify label names as necessary by adding an underscore and a number to guarantee uniqueness. For example, if the linker has already created an execution label with the name "myExample" and it encounters a completely unrelated label with the same spelling, it will modify the spelling of the label to

```
myExample_1
```

This will ensure that it does not conflict with the symbol called "myExample". If this is not sufficient for uniqueness, the linker will try the next sequential integer.

The DumpObj Tool

A BLITZ tool called "dumpObj" can be used to print out in human-readable form either an executable file or an object file. This tool will look at its input file (i.e., at its magic number) and will attempt to print out whatever sort of a file it is given.

The dumpObj tool will also perform some error checking on the file. If the file has obvious problems, error messages will be printed to identify the problems.