IDENTIFICATION

STINKING-DYNAL Relocatable Format

J. Pitts Jarvis III, Chris Reeve

27 March 1972

MOTIVATION

This document describes the file format that STINK and DYNAL load. It also sketches workings internal to the loader and describes symbol table formats in DDT.

REFERENCES

1. Samson, Peter, MIDAS, Artificial Intelligence Laboratory Memo 9Ø, (MAC-M-279), October 1968, SYS.Ø5.Ø1.

2. Daniels, Bruce, ATTACH -- A System Routine for Manipulation of Libraries, SYS.Ø7.Ø4.

HISTORY

STINK was originally written by Jack Holloway for the MIT Artificial Intelligence PDP-6 in 1965. It was modified by Chris Reeve of the Dynamic Modeling/Computer Graphics group over the period 1969 - 1972 for the special needs of our group. In February 1972, Pitts Jarvis added serveral new block types to accomodate the FAIL assembler and SAIL compiler which were written at the Stanford Artificial Intelligence Laboratory.

DYNAL, the dynamic loader, was written by Chris Reeve in late 1971 to dynamically load subroutines for a program running in the CARE environment.

STINK RELOCATABLE FORMAT

DYNAL and STINK expect to find files divided into blocks. Each block consists of three parts: a one word header, body, and a one word checksum. The header and checksum of each block are the same for all block types, whereas the format of the body varies.

The header word consists of five fields: end of file bit, loader conditional bits, type, count, and address. These fields are located in the header word as follows:

Table 1 -- Header Format

Field Bits	Name	Comments
4.9	EOF	1 implies End Of File, rest of block ignored
4.8 - 4.6	LCB	Loader Conditional Bits, see block type 1
4.5 - 3.8	TYPE	block type
3.7 - 3.1	COUNT	number of words in body
2.9 - 1.1	ADR	address, meaning depends on block type

The checksum is a one's complement checksum. The header and the body are checksummed together. The example illustrates how to compute the checksum. Suppose BP contains an AOBJN pointer to a block whose origin is BLOCK and whose length is LEN; hence

```
BP/ -LEN,,BLOCK.
```

The following incantation leaves the checksum of BLOCK in CKS.

```
START:  MOVEI   CKS,Ø               ; initial CKS
        PUSHJ   P,CKSIT             ; check sum one word
        AOBJN   BP,.-1              ; jump if more in block
        SETCA   CKS,                ; complement result
        JRST    4,                  ; halt

CKSIT:  JFCL    4,.+1               ; overflow
        ADD     CKS,(BP)            ; add word to checksum
        JFCL    4,[AOJA CKS,.+1]    ; add one to CKS if
                                    ;    carry Ø set
        POPJ    P,                  ; return
```

The checksum is computed by successively adding each word of the block to a running total and testing for overflow after each addition; a 1 is added to the running total for each overflow condition.

The exact format of the body varies with the block type. The following sections of this document describe the format of the body. There are also notes on symbol table formats for the translator, loader, and DDT. The block types as of the publication date of this document are listed here for reference. They are explained more fully below.

Table 2 -- STINK Block Types

TYPE	COMMENTS
Ø	illegal
1	loader command
2	code to be loaded at ADR (ADR not relocated)
3	code to be loaded at ADR (ADR relocated)
4	program name, last block of program
5	library search
6	COMMON (produced only by FORTRAN)
7	global parameter assignment
1Ø	local symbols
11	load time conditional
12	end load time conditional
13	local symbols to be half-killed
14	end of program (for libraries)
15	entries
16	external references
17	load if needed
2Ø	global symbols
21	fixups
22	polish fixups

STANDARD DATA

Block types 1 through 3, 7, and 2Ø through 22 have bodies with identical format. This format is called standard data. The body is divided up into subblocks each subblock having one word of code bits followed by a maximum of 12 (decimal) data words. Each subblock must use all its code bits unless it is the last. Associated with each data word are three or six code bits. The three high order code bits are associated with the first data word and so on until the code bits are exhausted. The code bits are decoded as shown in table 3. For a more complete description of the meaning of code (and extend) bits see the next two items.

Table 3 -- STINK Code Bits

Ø - do not relocate word
1 - relocate right half
2 - relocate left half
3 - relocate both halves
4 - global symbol (flags and squoze)
5 - minus global symbol
Note: flags specify destination of value, e.g., AC field, index field, etc.
6 - link
7 - extend (more information in next three bits)

The meaning of the extend bits are listed for reference in table 4.

Table 4 -- STINK Extend Bits

Ø - define symbol
1 - COMMON (produced only by FORTRAN)
2 - local to global recovery
3 - library request
4 - redefine symbol (same as define)
5 - global multiplied by n ($Ø \leq n < 1Ø24$)
6 - define symbol as "."
7 - illegal

As the loader reads a block, it accumulates a value (initially zero). The value of each data word, modified by its code, is added to the accumulated value. This process is continued until the loader encounters a finalize value mark (code less than 4). Each block of standard data may have one or more values. The block type dictates to the loader the manner in which these values are used.

CODES FOR STANDARD DATA

Code = 4 undefined symbol reference consisting of 4 flag bits and 32 bits of squoze.

The flags have the following meaning:

Table 5 -- Standard Data Global Symbols Flags

BIT	MEANING
4.9	Swapped or not swapped, 1 implies value should be swapped.
4.7 - 4.8	Destination of value Ø -- full word value 1 -- right half value 2 -- left half value 3 -- AC field value
4.6	Global or local, 1 implies global.

Code = 5 Same as 4 except negative of symbol's value is used.

Code = 6 Link request. This Code specification requires two data words; the first is a symbol (global or local depending on bit 4.6) and the second is a pointer to a chain of addresses in the code to be replaced by the value of this symbol. The chaining is done in the right half of the words. Bit 4.7 of the squoze word specifies whether or not to relocate the chain pointer.

Code = 7 More information in next code.

EXTEND CODE FOR STANDARD DATA

Extend Code = Ø — Define symbol -- a symbol and a value are supplied. The value is assigned to the symbol. If the symbol was previously defined and its old value differed from the new one, the loader will report an error. If already loaded code had references to this symbol, the value would be used to fix up the code.

Extend Code = 1 — Illegal to STINK or DYNAL.

Extend Code = 2 — Local to global recovery -- In a 1PASS assembly, the symbol which was presumed to be local was later declared global; this extend code specification causes the loader to modify its tables to reflect this condition.

Extend Code = 3 — Library request -- the supplied symbol is entered in the symbol table as undefined, forcing it to be "needed" in a library search block.

Extend Code = 4 — Redefine symbol -- same as define symbol except that the value of an already defined symbol may be changed without an error message.

Extend Code = 5 — Global multiplied by n ($Ø \leq n \leq 1Ø24$) -- the next global request (code bits 4 or 5) will be multiplied by n. N is the data

associated with this code.

Extend Code = 6 Define symbol as "." -- same as define symbol except that value is assumed to be the current location.

BLOCK TYPE 1 -- LOADER COMMAND

The body of block type 1 is "standard data". ADR is decoded as a loader command as follows:

Table 6 -- Loader Commands

Ø - global parameter assignment
1 - jump, specifies starting address
2 - global location assignment
3 - reset COMMON origin
4 - reset global relocation constant
5 - loader conditional on value
6 - set global offset
7 - .LOP, load time execution of specified machine instruction
8 - reset global offset

The notation LCn means loader command n.

LCØ: See loader block type 7.

LC1: Standard data is processed to produce one value, the starting address of the loaded program.

LC2: Standard data is processed to yield one value, the current loading location where STINK or DYNAL is set to this value.

LC3: Illegal to STINK or DYNAL.

LC4: Data is ignored. Current loading location is reset to its value prior to the last type 2 loader command.

LC5: Standard data is evaluated to yield a value which is tested against Ø. The test made is based on bits 4.8 - 4.6 in the block header as follows:

Ø -- not used
1 -- less than Ø
2 -- equal to Ø
3 -- less than or equal to Ø
4 -- not used
5 -- greater than or equal to Ø
6 -- not equal to Ø
7 -- greater than Ø

If the test succeeds, blocks between this one and the matching end load time conditional will be processed, otherwise, they will be ignored.

LC6: Standard data is evaluated to yield one value which is used as a global offset. This offset will be added into any evaluations of the current loading location.

LC7: Standard data is evaluated to yield three values. The first value is assumed to be PDP-1Ø operation code. The second value is assumed to be the contents of an accumulator. The third value is the contents of a location. The following instruction is executed by STINK or DYNAL:

<opcode><ac>,<loc>

The contents of <ac> after execution are assigned to global symbol .LVAL1 and those of <loc> are assigned to .LVAL2.

LC8: The body is ignored and the global offset is set to Ø.

BLOCK TYPE 2 -- Absolute Load

This block has standard data and starts loading the block into the absolute location specified by ADR.

BLOCK TYPE 3 -- Relocatable Load

This block has standard data and starts loading the block into the location ADR after ADR has been relocated.

BLOCK TYPE 4 -- Program Name

Only block types 13 and 14 may follow a program name block. The body of the block contains exactly one word, squoze and flags. All flags except bit 3 are ignored. If bit 3 is Ø the relocation is not reset; otherwise, the relocation is reset.

BLOCK TYPE 5 -- Library Search

The body of this block consists of a series of squoze symbols. This block must appear before any storage words have been loaded. The code bits define a series of conditions to determine if the following program is needed. A symbol is said to be satisfied if either flag bit 1 = Ø and the symbol has been previously seen, or flag bit 1 = 1 and the symbol has not been seen. If a symbol is satisfied, the following program is loaded, otherwise, the rest of the program is ignored. It is possible to

require that more than one symbol be satisfied before a program is loaded. If flag bit Ø is 1, and the associated symbol is satisfied, the loader will look at the next symbol and repeat the process until bit Ø is either Ø or a symbol is unsatisfied. If the symbol is unsatisfied, the program will not be loaded. Bit Ø = Ø marks the beginning of a series to be satisfied. There can be more than one series of symbols that must be satisfied before the program is loaded. If any one of the series of symbols is satisfied, the program is loaded. The flag bits 2 and 3 should be Ø and 1 respectively for all symbols.

BLOCK TYPE 6 -- COMMON

This block type is not guaranteed to work. It awaits the revival of FORTRAN.

BLOCK TYPE 7 -- Global Parameter Assignment

The first word of the block contains a symbol in squoze. Flag bit 4.6 is the global/local bit; the rest of the code bits are ignored. Following the symbol is standard data which is evaluated to yield one value which is assigned to the symbol.

BLOCK TYPE 1Ø -- Local Symbols

This is a block of local symbols. Each symbol is two words. The first word is squoze while the second is the value of the symbol. The flags are interpreted as follows:

require that more than one symbol be satisfied before a program is loaded. If flag bit Ø is 1, and the associated symbol is satisfied, the loader will look at the next symbol and repeat the process until bit Ø is either Ø or a symbol is unsatisfied. If the symbol is unsatisfied, the program will not be loaded. Bit Ø = Ø marks the beginning of a series to be satisfied. There can be more than one series of symbols that must be satisfied before the program is loaded. If any one of the series of symbols is satisfied, the program is loaded. The flag bits 2 and 3 should be Ø and 1 respectively for all symbols.

BLOCK TYPE 6 -- COMMON

This block type is not guaranteed to work. It awaits the revival of FORTRAN.

BLOCK TYPE 7 -- Global Parameter Assignment

The first word of the block contains a symbol in squoze. Flag bit 4.6 is the global/local bit; the rest of the code bits are ignored. Following the symbol is standard data which is evaluated to yield one value which is assigned to the symbol.

BLOCK TYPE 1Ø -- Local Symbols

This is a block of local symbols. Each symbol is two words. The first word is squoze while the second is the value of the symbol. The flags are interpreted as follows:

Table 7 -- Local Symbol Flags

Flag Bit	Meaning
Ø	half-kill symbol
1	relocation if flag bit 1=1, relocate left half
2	value if flag bit 2=1, relocate right half
3	block name

If the symbol is a block name, the value is the block level.

BLOCK TYPE 11 -- Load Time Conditional

This block has the same format as block 5 (library search) except portions of programs are conditionally loaded instead of whole programs. If the symbols in the block are satisfied everything following is loaded up to the next matching end load time conditional block (see below). If the symbols are not satisfied everything is ignored until the next matching end load time conditional block. Load time conditionals may be nested to any level.

BLOCK TYPE 12 -- End Load Time Conditional

This block has no body and is used to mark the end of a load time conditional.

BLOCK TYPE 13 -- Half-Kill a Block of Symbols

This block contains a list of local symbols to be half killed. The flags are ignored.

BLOCK TYPE 14 -- Block Generated by Library Creator

ATTACH creates this block type when it builds libraries. It

BLOCK TYPE 14 -- Block Generated by Library Creator

ATTACH creates this block type when it builds libraries. It is currently used only by DYNAL when loading libraries. This block type is used by DYNAL to separate the various programs in the library. This block is ignored by STINK.

BLOCK TYPE 15 -- Entries

This block must appear before any storage words are loaded; it contains a list of entries in the following program. If any of these entries have been seen, but not defined, the program will be loaded; otherwise, the program will be ignored. The entries are also used by ATTACH in buiding an SDAT for a library. This block is used only by DYNAL and is ignored by STINK.

BLOCK TYPE 16 -- External References

This block type contains a list of external references made by this program. If these references have not been seen by the loader, space is saved for them in the SDAT or USDAT.

BLOCK TYPE 17 -- Load if Needed

This block is the same as block 11 except the condition is on need rather than existence.

BLOCK TYPE 2Ø -- Global Symbols

The body has standard data. Data words in block type 2Ø come in pairs. The first word contains the squoze for the symbol. The flags indicate the type and to some extent control processing by the loader.

Flags = Ø4 Global symbol (internal).

Second word in value. Current symbol table is searched for an occurrence of this symbol. If none is found, the symbol is placed in the symbol table and any external requests (see below) involving it are resolved. If there is a previous occurrence, its value is checked. If the values are the same, no further action is taken. If the values are different, an error message is printed.

Flags = 6Ø Global request.

This construction is used when one program wishes to use a symbol defined by a different program. The first word is the squoze for the symbol whose value is described. The symbol table is searched for a global symbol by that name. If none is found, the symbol name and the second word of the pair which indicates the action to be taken, are placed in a table of unsatisfied requests until such a symbol is defined. When the definition of the requested symbol finally becomes available, the following action takes place. If bit Ø of the second word is Ø, then bits 18 - 35 are a pointer to a chain. The right half of the word pointed to is saved and then replaced by the right half of the value of the designated symbol. The saved right half is then used as a pointer to repeat this process. The process terminates whenever the pointer is Ø.

If the bit Ø of the second word is a 1, the process is slightly more complicated.

If bit 2 of the second word is a Ø, then bits 18 - 35 contain

a pointer to a word of code. The right half of the value of the requested symbol is added to either the right or left half of the indicated word. Note that this is a half-word add, and no chaining is done. The add will be to the right half if bit 1 = Ø and to the left half if bit 1 = 1.

If bit 2 is a 1 then the process is exactly the same as above except that the fixup is done to the symbol table. Bits 3 - 35 contain the right most 33 bits of the squoze for the symbol whose value is to be "fixed-up". This symbol must be the last symbol entered into the symbol table. The name is included only to serve as a check. Again the addition is done to the right or left half depending on bit 1. Since symbol table fixups are allowed, and since these fixups may be to global symbols (code bits = Ø4), the concept of a deferred internal is introduced. Deferred internals are global symbols which are going to have a symbol table fixup done to their values and hence should not be used to resolve global requests until after the fixups have been done.

Flags = 24 Indicate a global which will have the right half of its value fixed up.

Flags = 44 Indicate a left-half fixup will be done.

Flags = 64 Indicate both a left and a right-half fixup to be done.

Other than delaying satisfaction of global requests (and also the symbol table search for multiple definiton) these

symbols are treated exactly like globals (code bits = Ø4).

BLOCK TYPE 21 -- Fixups

The purpose of this block type is to allow a one-pass translator to reference symbols which are defined after their use. The block contains a series of data words each of which has a value in the right half and an address in the left half. The fixups are chained just as the global requests which have bit Ø a Ø in the second word (see block type 2Ø). If however, one of the data words is -1, then the next data word indicates a fixup to the left half of the word specified rather than the right half. Chaining is done through the left half in a manner analogous to that for right half chains.

BLOCK TYPE 22 -- Polish Fixups

Polish fixups allow complex fixups involving multiplication, shifting, etc., of relocatable or externally defined quantities. Each block of type 22 must contain one and only one polish fixup. The data words are considered to be a series of half-words. First the prefix polish for the expression appears, then a store operator, then the store address. The operators and operands are as follows:

Polish Operators	Comment
Ø	the next half word is an operand
1	the next two half words form a 36-bit operand (the first is bits Ø - 17, the second bits 18 - 35).
2	the next two half words give the squoze for a symbol whose value is to be used as an operand (a form of global request). The

	code bits should be Ø4.
3	add
4	subtract
5	multiply
6	divide
7	logical and
1Ø	logical or
11	left shift (LSH)
12	xor
13	one's complement (not)
14	two's complement (negate)

The store operators are as follows:

Store Operators	Comment
-1 (777777)	right half chained fixup (see block type 2Ø)
-2 (777776)	left half chained fixup
-3 (777775)	full word fixup. The entire word pointed to is replaced, chaining is done on the right half of the word pointed to.

All of these store operators use the half word which follows them as a pointer to the location to fixup.

A NOTE ABOUT SYMBOLS

Symbols like cats lead several lives. Unlike a cat, a symbol leads only three lives. These incarnations are translator output, internal loader format, and DDT format. The translator format has been described above.

Symbols are six or fewer characters encoded as squoze (alias radix 5Ø). A detailed description of the mechanics of coding squoze symbols exists in the MIDAS manual (ref 1). The radix 5Ø format allows the coding of six characters in 32 bits (hereafter called squoze) and leaves four bits left over for flags. These flags are the high order bits of the word (4.6 - 4.9) and are used at different times for different things.

INTERNAL LOADER SYMBOLS

At load time the symbols are segregated into two groups: locals and globals. The locals are immediately placed in DDT format (explained below) and are not processed further. The globals are passed into the global table as they are seen. Flag 4 of the symbol marks it as global. The symbol is followed by one or more words depending on whether it is defined or not. If the symbol is defined then flag bit 1 is 1 and only one word follows, the value. In all cases flag bit Ø is 1. (This is for historical reasons and we will dwell on it later). If the symbol is undefined then flag bit 1 = Ø and the symbol is followed by one or more references or requests.

If a global symbol is seen by the loader, but not yet defined, all references to the symbol must be saved so that all requests for the global can be satisfied at definition time. The format of references is described in table 8.

Table 8 -- Global Request Word

Bits	Comments
4.9	always Ø (this makes all references positive numbers)
4.8	polish request (reference address is fixup number)
4.7	links (references to symbol are on link list pointed to by reference address)
4.6	single word fixup, bits 3.1 - 4.5 further decoded
3.5 - 4.5	multiply symbol by this number (no multiplication if zero)
3.4	negate value
3.3	swap value
3.1 - 3.2	Ø - full word store 1 - store right half only 2 - store left half only 3 - AC field only
1.1 - 2.9	store location if 4.8 = Ø, polish fixup number if 4.8 = 1

If the polish bit (4.8) is on, the reference address is interpreted as a polish fixup number. This means that the loader found a polish fixup that referenced this symbol while undefined. When the symbol becomes defined, the polish fixup is looked up (it is kept in the same table) and the symbol value is substituted for its squoze in the fixup. A count of global requests is kept in each fixup; as symbols are defined the count is decremented. When this count reaches zero, the polish is evaluated and deleted from the global symbol table. Each polish fixup has a four word header followed by one or more tokens. The header is as follows:

Table 9 -- Symbol Table Polish Fixup Header Format

Word	Bits	Comments
word 1		
	4.6 - 4.9	always 44
	1.1 - 4.5	fixup number
word 2		number unsatisfied global requests
word 3		store operator
word 4		store address

Each token of the fixup is either one or two words. The first word of the token specifies whether another is to follow. The format of a token is described by table 1Ø.

Table 1Ø -- Polish Token

Word	Bits	Comments
word 1		
	4.9 - 2.2	zero
	2.1	high-order bit of value
	2.9 - 1.1	token type Ø => next word is squoze (2.1 = Ø) 1 => next word is value 3 - 14 => operator (only word of token)
word 2		
	4.9	zero
	4.8 - 1.1	arbitrary (either squoze or value)

It must be pointed out that polish fixups cannot be confused with global symbols because the fixup number should always be less than 5Ø*5Ø*5Ø*5Ø*5Ø; while bits 1.1 - 4.5 will always be greater than or equal to 5Ø*5Ø*5Ø*5Ø*5Ø for a symbol.

One other note must be made about the differences between STINK and DYNAL. STINK keeps its global symbols in one contiguous table while DYNAL puts global symbol table entries on a linked list. STINK uses bit 4.9 to separate entries in the global symbol table. The first word of each entry (squoze) has bit 4.9 turned on, the second word is ignored, if the third word has 4.9 turned on, it is the first word of the next entry in the table; if 4.9 is off, then all subsequent words with 4.9 off belong to this entry. The first word with 4.9 turned on is the first word of the next entry.

DDT SYMBOL TABLE FORMAT

Each entry in the DDT symbol table is two words long. The first word is the symbol name and the value follows. The flag bits are coded as follows:

Table 11 -- DDT Symbol Flags

Bit	Comment
4.9	half-killed
4.8	fully killed
4.7	local
4.6	global

If the flags are all zero, this is a block name. If the left half of the value is non-zero, then this is a program name; the left points to the bottom of the program and the right half points to the top of the program. If the left half of the value is zero, the value contains the level of this block.

Local symbols are entered into the symbol table in the order in which they are seen by the loader. DDT, however, expects to see them in the order shown below.

Non-Block Structure Program

lowest core location

program name
global
and
local
symbols
for that
program

Block Structure Program

Assume the block structure is as follows:

```
Begin b1 (same as program name)

     begin b2
     end b2

          begin b3
               begin b4
               end b4

          end b3

End b1
```

Then symbol table should have the following format.

```
program name  (b1)
block name     b2
symbols for block  b2
block name     b4
symbols for block  b4
block name     b3
symbols for block  b3
block name     b1
symbols for block  b1
```