The research reported in this paper was sponsored by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation.

TECH MEMO *a working paper*

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406
Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/550/00

AUTHOR Jeff Barnett for L.H.
L. Hawkinson

TECHNICAL Jeff Barnett
J. Barnett

RELEASE [illegible]
C. Weissman, SDC
[illegible]
D. Anschultz, III
for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 29 PAGES

(Page 2 is blank)

LISP 2 Internal Storage Conventions

ABSTRACT

This document describes storage allocation conventions for the LISP 2 system proposed for the IBM S/360 computer. Core storage is to be partitioned into a large number of zones, each of which is classified according to the types of structural units that may occur within it. The configuration of each of the standard zones and its occupant units are described in detail. A scheme for the software paging of binary program space is outlined.

TABLE OF CONTENTS

		Page
Section 1.	General Organization of Core Storage	5
1.1	Zones	5
1.2	Layout of Core Storage	5
1.3	The Quantized Core Map	5
1.4	Space Selection	6
1.5	Class Definition and Space Creation	7
2.	Program Zones	8
2.1	Constraints and Purposes	8
2.1.1	Fixed Program Space	9
2.1.2	Storage Map Space	9
2.1.3	Pushdown Stack Space	10
2.1.4	Binary Program Space	10
2.2	Program Structure Spaces	10
3.	Fixed Units	18
4.	Field Unit and Array Zones	18
5.	Type Representation	18
5.1	Type Encoding	18
5.2	HCONS	19
6.	Software Paging of Binary Program Space	21
6.1	Binary and Secondary Program Space (BPS, SPS)	22
6.2	User Control Facilities	23
6.2.1	LAP	24
6.2.2	EXCISE	24
6.3	Automatic Task Recognition	25
6.4	Remaining Issues	29
Fig. 1	Fixed Program Space	11
Fig. 2	Pushdown Stack Space for the 360	14
Fig. 3	Binary Program Space	16
Fig. 4	Possible Design of BPI for the 360	17
Fig. 5	Automatic Task Recognition	26

1. GENERAL ORGANIZATION OF CORE STORAGE

1.1 ZONES

The directly addressable core storage available to the LISP 2 system is partitioned into a set of zones. Each zone belongs to a particular class according to the types of structural units permitted within it. Each zone, furthermore, is divided into two distinct areas. The active area contains well-defined units whose types are among those allowed by the class. The free area is that part of the zone not in current use, and available for the building of new structures. The words of core storage that constitute a zone are always contiguous by address.

1.2 LAYOUT OF CORE STORAGE

The storage map for a LISP 2 job concerns the partitioning of available core storage into zones. In preparation for the demands of tasking and/or paging in a time-shared environment, the flexibility of the storage map must be enhanced considerably for the 360 LISP 2 system as compared with its Q-32 counterpart. There are three principal amendments to the Q-32 scheme through which the desired degree of flexibility can be achieved: First, no restrictions should be placed upon the order in which zones appear in core storage; second, any number of zones of a particular class should be able to exist simultaneously and non-contiguously in core--for example, there may be two distinct list node zones used by different portions of a program; third, as mentioned previously, it should be relatively easy for a programmer to define a new zone. The remainder of this section is devoted to the impact of these three amendments on the problems of storage allocation.

1.3 THE QUANTIZED CORE MAP

Available core storage is partitioned into zones in such a way that the boundaries between adjacent zones are all multiples of some constant power of two (defined by the variable QUANTUM). This obviously means that each zone contains a number of bytes that is a multiple of QUANTUM. Thus, one may say that core storage is quantized, and one may call the set of bytes (beginning at address J*QUANTUM and extending through address (J+1)*QUANTUM-1) the Jth quantum of core storage.

The quantized core map is simply a table whose Jth entry describes some of the properties of the Jth quantum. In particular, the Jth entry contains a reference to the zone descriptor Z of the zone to which the Jth quantum belongs. The zone descriptor Z is used to elicit more information on the contents of the Jth quantum and its relationship to neighboring quanta.

Given a core address A, the quantum into which A points is easily determined by dividing the difference of A and QORG (origin of lowest-addressed question) by QUANTUM, and ignoring any remainder (this computation may be performed by a shift instruction). Thus, for example, one may find the class manager of the zone into which an address A points by the following three steps: (1) find the quantum J to which A points; (2) use J to index the quantized storage map and obtain the zone descriptor Z; and (3) access component MANAGER (Z), which yields the class manager for the zone into which A points. The quantized core map is used extensively in the type determination primitives and garbage collector, where the type of unit to which an arbitrary core address refers must be computed efficiently.

The value of QUANTUM chosen for an implementation depends not only on the machine, but on the available core storage ar well. For the 360, 2^{10}, 2^{11}, or 2^{12} would seem to be an appropriate choice.

There are two variables, SYSORG and SYSEND, which indicate--respectively--the lower and upper bound addresses for core storage available to a particular LISP 2 job. All zones that contain units known to the LISP 2 system must be located between SYSORG and SYSEND (whose values are multiples of QUANTUM). Any zone whose boundaries fall outside this interval is termed a virtual zone. Since pointers into a virtual zone cannot refer to actual structural units, they may be used to represent absolute, unstructured quantities. Three general format address intervals have been so used in Q-32 LISP 2: [0, 1] for Booleans; [200000_8, 377777_8] for small, unsigned integers; and [400000_8, 777777_8] for small, signed integers. Similar conventions will be adopted for the 360. Virtual zone quanta must be included in the quantized core map except for that which would contain address zero.

1.4 SPACE SELECTION

At any time, there may exist in core any number of zones of a particular class. However, allocation functions for that class do not want to be given extra arguments specifying in which zone they are to build their structural units. There exists a dilemma: one wants functions which can build units in any zone of a given class, yet one does not want to specify explicitly which zone they are to use every time they are called. (Who, for example, would be willing to supply a third argument to CONS?)

To resolve this dilemma, a selection mechanism for zones must be used. At any given time for each class, one zone of that class is said to be selected, and all allocation functions for that class build units in that zone.

A zone selection is made in a program by calling the function SPACESELECT in section LISP; SPACESELECT takes one argument, the name of a zone, and selects it for that class to which it is known to belong.

A possibility arising from the allowed multiplicity of zones of a particular class is that of creating in lieu of garbage collection a new zone of that class in an otherwise empty portion of core storage. The advantage of this scheme, if core is apportioned among zones in such a way as to leave a minimum of free area in each, is that new zones of a particular class can be created on demand as free areas of other zones if that class becomes exhausted. Of course, when all empty space is gone, a garbage collection must be performed, but the need for garbage collection is eliminated in the situation where some zone is depleted while another zone contains a large free area. One possible disadvantage of such a scheme is that core may become fragmented unless that part of garbage collection which deals with reallocation of structural units is capable of reassigning and moving units from one zone to another. Methods for accomplishing such inter-zone transfers are understood and appear to be feasible.

The zone selection mechanism defined here is patterned after that used successfully in Q-32 LISP 1.5 and LISP 2 for input/output device selection.

1.5 CLASS DEFINITION AND SPACE CREATION

For LISP 2 to be a truly flexible and efficient system, it is desirable to provide the user with facilities for defining a new class of zones and for creating instances of it. For example, if the user wishes to work extensively with SLIP cells or a particular type of field unit, he should be able to devote one or more zones exclusively to these units in order to achieve maximum efficiency in storing and maintaining them. Even if these valid user demands were to be ignored, there still remains the problem of defining within and for the system itself the multitude (approximately a score) of standard zone classes. The need for a formal mechanism for defining classes and creating zones cannot be denied.

Fortunately, the organization of storage in terms of zone descriptors, zone managers, unit descriptors, etc. provides a sound framework for the required facilities. To define a new class of zones it is necessary to create and appropriately initialize a manager for it. A function CLASSDEFINE will be available to perform this service; it will take as parameters the initialization for all components of the manager. Of course, many of these initializations are primitive functionals for use in garbage collection and type determination, and although what is expected of each of them can be stated quite simply, it is still system programming and beyond the domain of the average user. Thus, standard sets or subsets of parameters for the common cases will have to be provided in much the same way standard parameters are provided for opening input/output files. Alternatively, or additionally, it may be possible to generate the appropriate parameters from declarations made by the user for those units that can occur in the zone.

Once a class has been defined, a zone of that class may be created by expropriating an area of core (most likely from empty space), creating a proper zone descriptor for it, and updating the quantized core map. The function SPACECREATE will take as arguments the initialization parameters for the zone to be created, and will perform the required modifications of and additions to the storage map.

2. PROGRAM ZONES

The program zones for LISP 2 are those that are involved in function calling sequence linkages. This group includes fixed program space, storage map space, binary program space, and pushdown stack space. These four, though decidedly dissimilar in character, are so interrelated in the mechanics of a function call that they must be treated together.

2.1 CONSTRAINTS AND PURPOSES

The principal motivation in the design of program structure is that of providing a workable and efficient function-to-function calling sequence. A solution is judged workable not only by its ability to effect a call, but also from the viewpoint of backtracing and garbage collecting the pushdown stacks. Any measure of efficiency of a particular solution must balance both time and space considerations.

The constraints on a function-to-function calling sequence (i.e., those features of the system that must function properly within any proposed scheme) are the following:

1. All calls must be recursive.

2. If, upon entrance to a function, either of the pushdown stacks is found to be insufficient, it must be possible to garbage collect at that point in time.

3. It must be possible, upon entrance to a function, to unwind the stacks with fluid variable-restoration to any higher level, and to continue processing in the function to which one has unwound.

4. It must be possible during garbage collection to seek out and update all reference pointers on the symbolic stack (the most difficult problem is that of correcting return addresses when binary programs move).

The two most important efficiency considerations are:

1. The in-line code required to perform and parameterize a call should be minimized (subject to the constraints stated above).

2. Base registers should be allocated so as to minimize the time and code required to load base addresses, but the number of general registers so allocated should also be minimized.

Before setting down the detailed configurations of the four program zones, it is useful to describe their functions in a general way.

2.1.1 Fixed Program Space

Fixed program space acts as the anchor for LISP 2 as a system. It is always to be found at a fixed location in core, and its contents are always available for reference. Among other things, fixed structure space may contain:

1. the system sign-on code

2. function-to-function linkage routines

3. special-purpose code sequences to accomplish simple tasks (such as fluid bindings) that require too much space to appear in-line

4. quantized core maps

5. base address tables for fixed unit zones (identifier space, quote cell space, etc.)

6. commonly occurring, unstructured constants

2.1.2 Storage Map Space

Storage map space is used only for storage management descriptors. The system descriptors, zone descriptors, zone managers, and possibly a part or all of the quantized core map are found here. The justification for this space is based on the need for treating these descriptors in a special way during garbage collection--they cannot be allowed to wander about core as can other structural units. Storage map space is a uniform field unit zone; thus all of the various types of descriptors appearing within it must be of the same size.

2.1.3 Pushdown Stack Space

The configuration of pushdown stack space has a great impact upon that part of the system devoted to the production of executable code (the compiler and assembler). This is due to the fact that pushdown stacks must be designed to conform to the constraints on calling sequences stated earlier in this section, and the compiler and assembler must in turn conform to the conventions for using the stacks. A pushdown stack is divided into intervals, each of which provides space in which an active function may store its arguments, return information, lexical variable values, and partial results.

2.1.4 Binary Program Space

Binary program space contains executable code for LISP 2 functions. Each function in the zone is represented by a structure which is termed a binary program image (BPI) or, more precisely, a resident binary image (RBI) to distinguish it from BPI's in secondary storage. A BPI is produced by the LISP 2 assembler (LAP) from its symbolic arguments, which are usually generated by the compiler but which may be supplied independently by the user. LAP is not a general-purpose assembler: its features are specifically designed to interface with the LISP 2 compiler, garbage collector, storage conventions for fixed structures, and the pushdown stacks. (LAP is described fully in TM-3417/400/00.)

Any BPI which is generated by LAP must be re-entrant and dynamically relocatable. To satisfy these two requirements, all scratch space needed by a function must be obtained from the pushdown stacks or general registers (accumulators), and any reference from a BPI to a word within or without it must be available for updating by the garbage collector.

2.2 PROGRAM STRUCTURE SPACES

The fixed program space for the 360 is illustrated in Figure 1. It anchors the LISP 2 system because general register 15 is at all times kept equal to the fixed program origin (FPO) and thus any part of the space within the interval [FPO, FPO+2^{12}] is always immediately accessible. Fixed program space can not be said to contain structural units as can be seen in Figure 1; however there are several distinct labeled areas serving the following purposes:

FNLNK	Function-to-function linkage code must be located at FPO, i.e., register 15 must always point to it
LINKAGES	Other functional and ROUTINE link and return code (FMCALL, RTCALL, RETURN, etc.)
OUTLINE	Special-purpose code sequences; calls from BPI's for reasons of efficiency (FLBIND, FLREST, SETLOC, etc.)

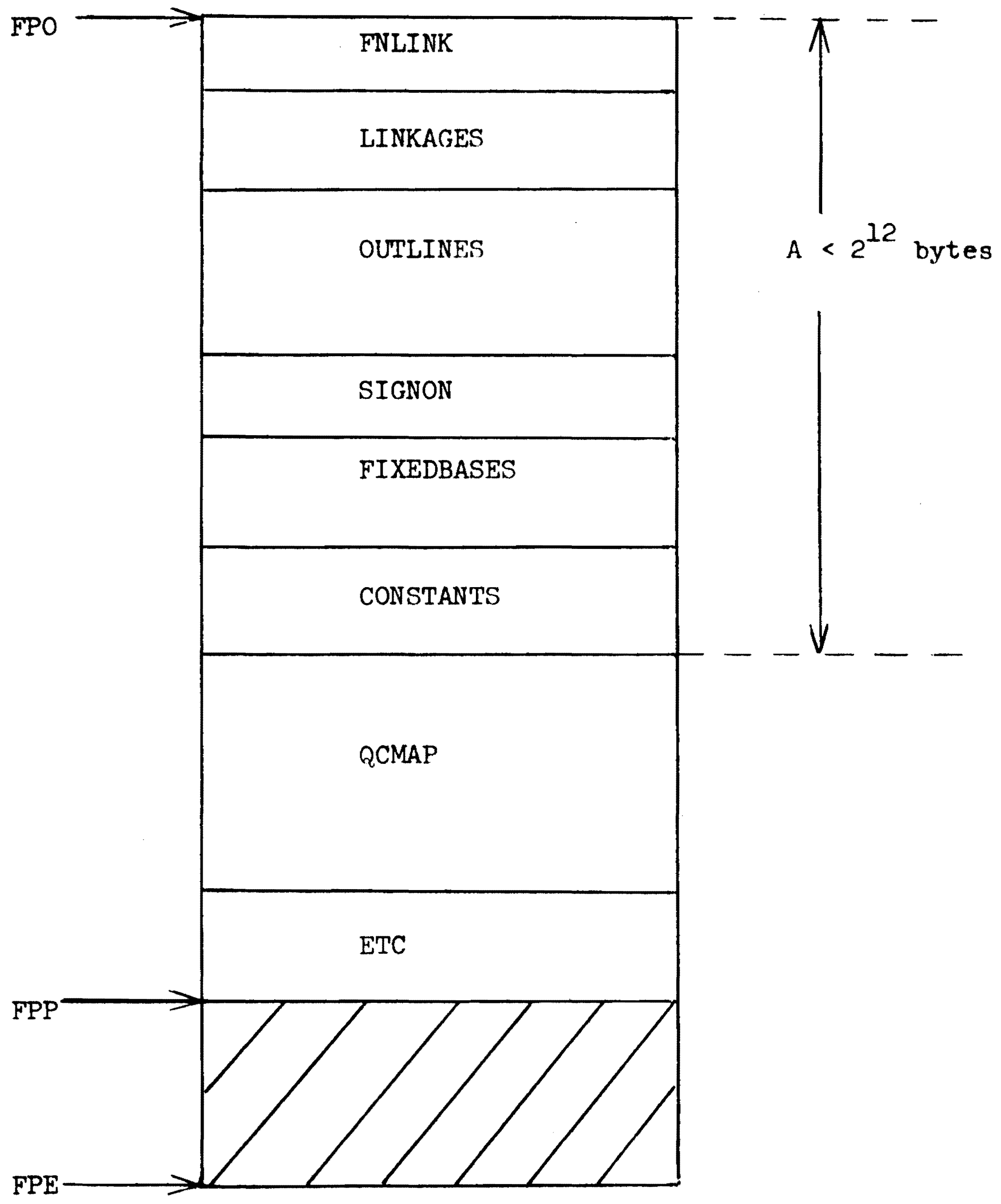


Figure 1. Fixed Program Space

SIGNON	Sign-on code for LISP 2 system with monitor linkage; also rescue error restart according to monitor capabilities
FIXEDBASES	A set of base address tables for fixed structure zones (see Section 3), loads from which are generated by LAP on certain references to fixed units (see the LAP document); each table contains some number (1-16) of full words, the kth element of which contains the base address of the kth quantum of the class of space to which it applies; the entry names for these tables are: CHBASE, IDBASE, QTBASE, FLBASE, OWBASE, ENBASE, and FNBASE; these base address tables must be updated whenever fixed unit zones are created, destroyed, or moved
CONSTANTS	Commonly occurring absolute field constants, e.g., integers $\{n, -n, 2^n, 2^{n-1}, -2^n, -2^{m-1}, 2^{12}n, 10^n\}$; reals $\{n, -n, 10^n, -10^n\}$; virtual integers $\{n, -n\}$; etc. for reasonable, small, integral values of n
QCMAP	The quantized core map (see Section 1.3)
ETC	Other useful things

Data and code are generated into fixed program space by LAP when an origin is given in its LISTING argument; when this is done, entries may be defined for future reference. All names given in the above descriptions of labeled areas are entry names.

One possible configuration of pushdown stack space for the 360 is illustrated in Figure 2. Two active areas, the absolute stack and the reference stack, may be observed. Each of the two stacks consists of a sequence of called function intervals; for each called function interval on the absolute stack there is a corresponding interval on the reference stack. Such a pair of called function intervals constitutes a pushdown stack unit. The stacks expand away from the stack-space boundaries PDO and PDE toward one another as the recursive depth of called functions increases. As each function is entered, a new interval comes into existence for it on both the reference and absolute stacks; when that function is exited, the intervals are removed.

A snapshot cross-section of each of the two stacks is shown in Figure 2 at a moment during the execution of function g, which has previously been called from function f and which is in the process of calling function h. The called function intervals for g appear in full in the figure, and with no loss of generality, the substructure of the intervals can be described in terms of that shown for g.

An interval on the absolute stack can only contain "absolute" fields and is allowed to have "holes," i.e., fields whose values are unknown and irrelevant. Conversely, a reference stack interval can only contain "reference" fields and can have no "holes," except for the portion A, the significance of which is defined below (return-block and register-block). In either case, a pointer to any field on either stack is legitimate for general format, but the format of the field it refers to cannot always be determined.

A called function interval on the absolute stack (such an interval for g is covered by ΔASP_f in Figure 2) consists of:

1. scratch space for the calling function (absolute quantities)

2. stack-transmitted absolute arguments which have been evaluated by the calling function prior to calling a sub-function

A called function interval on the reference stack (that for g is covered by ΔSP_f in Figure 2) consists of:

1. scratch space for the calling function (reference quantities)

2. stack-transmitted symbolic arguments evaluated by the calling function prior to calling a sub-function

3. a register-block

4. a return-block

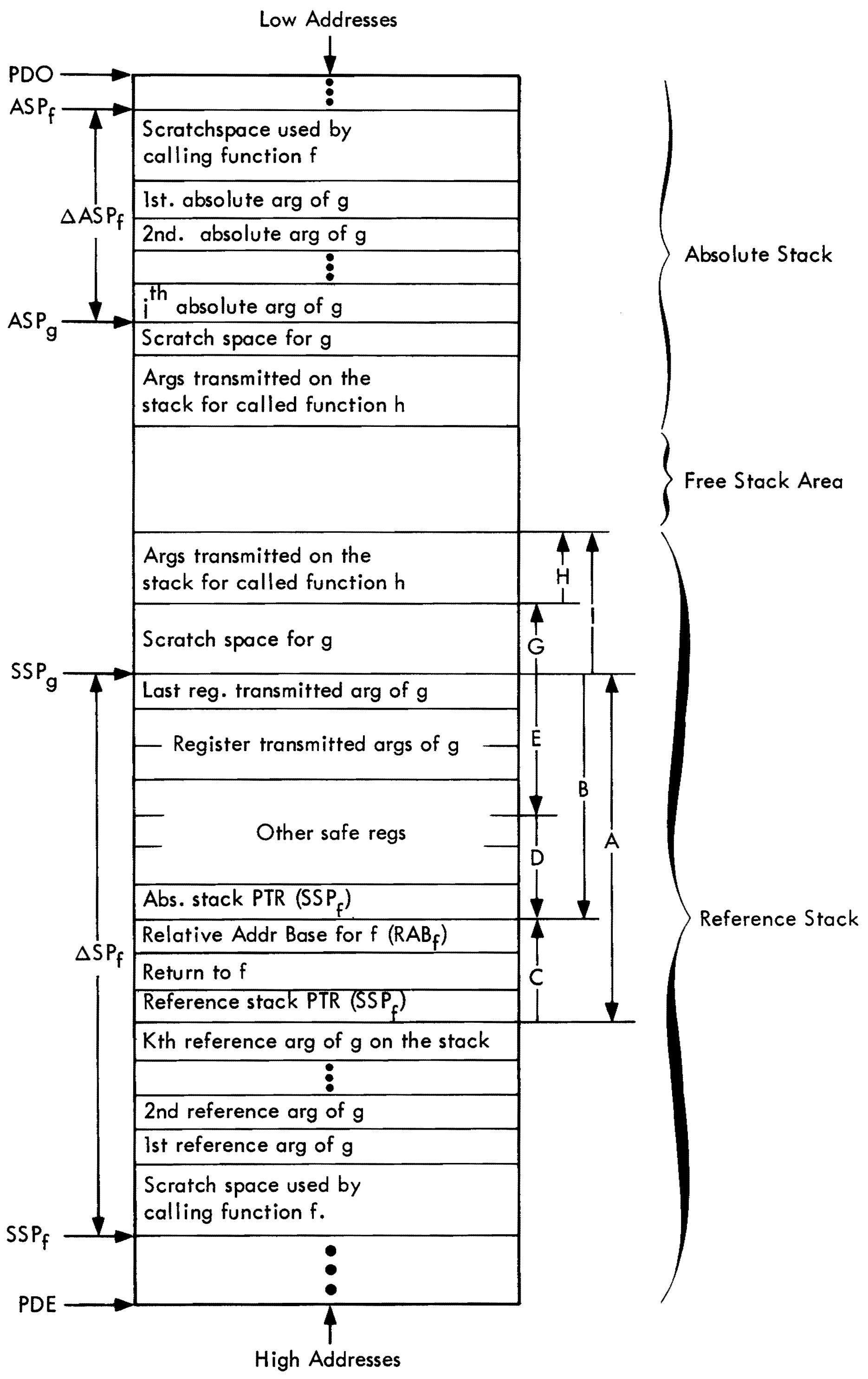

Low Addresses
PDO
ASP_f
Scratchspace used by calling function f
1st. absolute arg of g
2nd. absolute arg of g
ΔASP_f
i^{th} absolute arg of g
ASP_g
Scratch space for g
Args transmitted on the stack for called function h
Absolute Stack
Free Stack Area
Args transmitted on the stack for called function h
H
I
Scratch space for g
G
SSP_g
Last reg. transmitted arg of g
Register transmitted args of g
E
B
A
Other safe regs
D
Abs. stack PTR (SSP_f)
Relative Addr Base for f (RAB_f)
C
Return to f
Reference stack PTR (SSP_f)
ΔSP_f
Reference Stack
Kth reference arg of g on the stack
2nd reference arg of g
1st reference arg of g
Scratch space used by calling function f.
SSP_f
PDE
High Addresses

Figure 2. Pushdown Stack Space For The 360

The return-block (C) for the 360 consists of:

1. previous absolute stack pointer (ASP_f)
2. binary program base address for calling function (RAB_f)
3. return location in calling function (RET_f) (with implicit call parameters)
4. previous reference stack pointer (SSP_f) (the address of this element in the previous return-block).

The register-block (B) is made up of two parts:

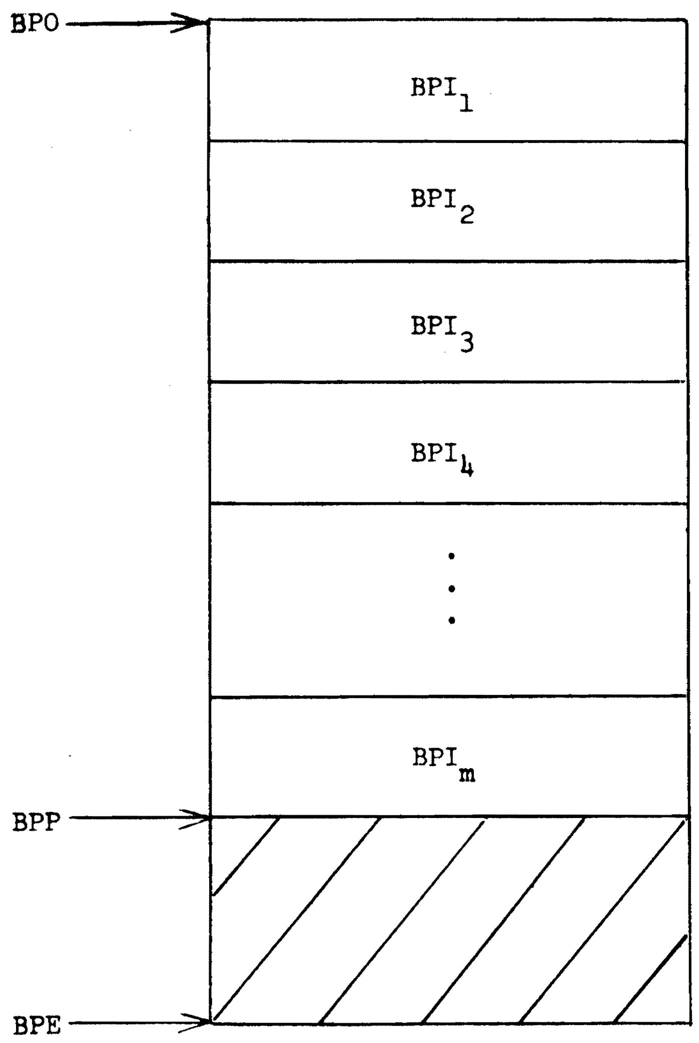
1. register-transmitted arguments for the function (E)
2. contents of registers which are saved by function g upon entry and restored upon exit.

The sum E+D = B is an implementation constant reflecting the number of registers allotted to the compiler less those allocated for scratch and communication. The arguments of the function that are transmitted on the stack are distributed in sequences on the reference stack (E) and/or on the absolute stack. In most cases on the S/360, real arguments are transmitted in the floating-point registers.

In each 360 calling sequence there is a parameter called the MAP which establishes for each constituent of the register block whether that constituent is a reference field or not. This MAP is used by the garbage collector in its marking scan of the reference stack. In scanning from the current SSP toward PDE, it looks only at the reference fields and can distinguish between general, functional, locative, and state restoration formats.

The state restoration format is a special field format that can appear only on the reference stack; it is used to hold, among other things, a set of saved variable bindings that are to be later restored. It requires a special format on the stack so that it can be recognized by the stack-unwind portion of the TRY-EXIT feature, which must do the restoration.

A diagram of binary program space is presented in Figure 3; it is seen to contain units called binary program images (BPI's). A possible configuration of a binary program image for the 360 is illustrated in Figure 4. The complexity of the division of a BPI into code sections results entirely from the base addressing scheme of the machine; no jump can be made to a location more than 2^{12} bytes past a given base register. Only those functions, therefore, that assemble into a BPI of more than 1024 words require more than one code section.



BPI_k are binary program images (they are of diverse dimensions)

Figure 3. Binary Program Space

Figure 4 shows the formats for binary program images in the 360. A fuller description of a BPI, together with the calling sequence that binds it to other BPI's through its function descriptor, can be found in the LAP document (TM-3417/400/00).

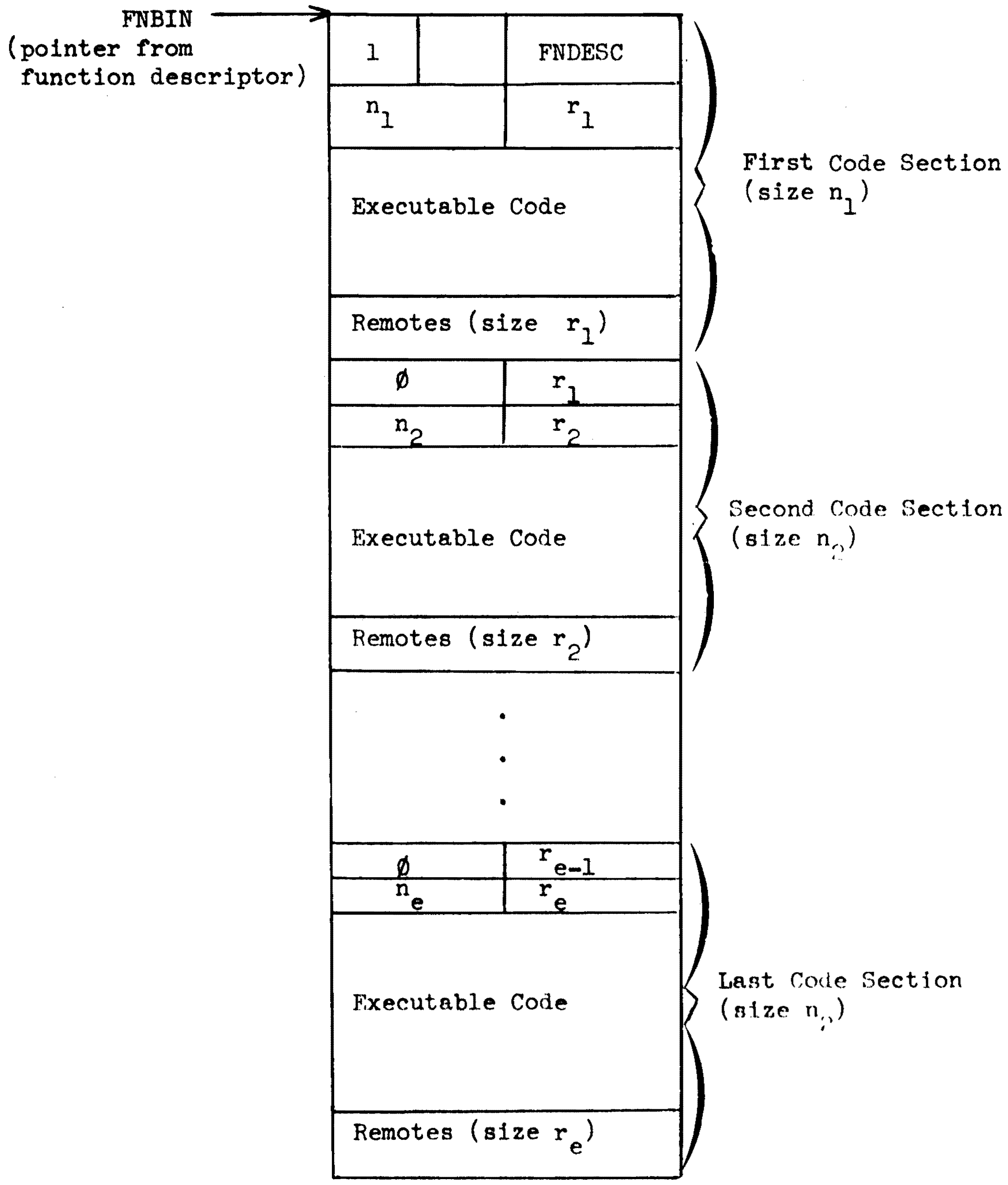


Figure 4. Possible Design of BPI for the 360

3. FIXED UNITS

Fixed units are defined in LISP 2 to be those structural units that can be referred to directly from a binary program image (BPI). Included in this group are identifiers, quote cells, free variable structures, and function descriptors.

The fact that fixed structures are referred to from BPI's imposes two requirements on each such structure: (1) that it be located in a fixed position with respect to some base address, and (2) that it contain a count of the number of references to it from all active BPI's. These requirements result from the decision to exempt BPI's from garbage collector processing except when it is necessary to update relative addresses within them.

The fixed location and reference count problems are compounded on the 360 by the desire to make references to fixed units as efficient as possible. This may be accomplished by minimizing base register loading, which in turn demands that those fields of a unit that are most commonly referenced be packed densely within a page. The best apparent solution to this problem is to separate each fixed structure into two parts. One part contains commonly referenced fields, and its core address is the site address of the structure. The other part contains all other fields; its location must be obtainable from the side address by a simple transformation. This solution has been adopted for the 360 with some trepidation; the problems of storage allocation and maintenance arising from it are manifold.

4. FIELD UNIT AND ARRAY ZONES

Field unit and array zones contain most of the structures commonly manipulated by a user program. Field unit zones come in three styles--pure,uniform, and mixed--whereas array zones come in only the "detailed header" style.

5. TYPE REPRESENTATION

Fields that contain type information have by now appeared in a multitude of structural units: among these are free variables, function descriptors, and unit descriptors. In describing these, the type-representing fields have merely been specified to be in general format. The problem of type encoding remains to be discussed; this is done below.

5.1 TYPE ENCODING

Within the Q-32 LISP 2 system, there exists a rather elaborate scheme for encoding the declaration and type information associated with a structural unit. When such a unit is created by the functions CREATE or MAKEFREE, the argument representing its type is transformed from symbolic IL format into a sequence of 6-bit bytes. This byte sequence is then stored in the newly generated unit in one of several ways, depending upon its length. Later, when the functions STYPE or FTYPE are called upon to retrieve the type, the inverse transformation from byte sequence to S-expression is made.

The justification for the Q-32 encoding scheme rests solely upon the consideration that a byte sequence will require less storage space on the average than would the equivalent S-expression. Against this purported advantage must be weighed the obvious disadvantages. Encoding and decoding byte sequences is a time-consuming operation, especially for decoding, where often much structure is generated for transient usage (such as type checking, argument classification, and field format determination within the primitives and garbage collector). A substantial amount of obscure code in the system is devoted to encoding and decoding. The encoding scheme is clumsy and difficult to extend; this limitation is especially serious in view of the proposed addition of format type and table definition features to LISP 2, which will expand considerably the syntax of types. To meet the requirements of the expected expansion of this syntax would require a complete redesign of the byte-encoding scheme.

The first step toward solving the difficulties of type encoding seems to dictate abandonment of byte sequences in favor of something more flexible--for instance, S-expressions. The second step is to find a way to keep storage requirements for the more flexible format at a tolerable level. Is this possible? First of all, it can be observed that, out of a large sample of declaration data, there will be large subsets, all elements of which are of the same type. Furthermore, if one considers not simply the complete declarations, but rather all the dotted pairs of which they are ultimately composed, it is intuitively clear that the number of distinct pairs is low in proportion to the size of the sample as a whole. Thus, the answer to the question seems to be that indeed, storage requirements for declaration data in S-expression format can be kept within reason if, whenever the sub-parts of two types are equal, they in fact are represented uniquely.

5.2 HCONS

A feasible scheme for realizing unique internal representation of data has been described by J. McCarthy and is termed "hash-CONSing". In order to illustrate this term, it is useful to define an H-expression in terms of a function HCONS:

1. Any identifier or Boolean is an H-expression
2. If H1 and H2 are H-expressions, then HCONS (H1, H2) is an H-expression
3. If H1 and H2 are H-expressions, then EQUALN (H1, H2) $\rightarrow$ EQN (H1, H2)
4. CAR (HCONS (H1, H2)) = H1 and CDR (HCONS (H1, H2)) = H2
5. NOT ATOM (HCONS (H1, H2))

Clearly, an H-expression, as defined above, differs from a general S-expression in two respects: it is internally unique (property 3 above), and it allows only identifiers and Booleans as atomic elements. (Other, less restrictive definitions are possible within LISP 2.) In addition, implementation considerations impose the constraint that the CAR and CDR fields of a node created by HCONS may not be altered. Fortunately, these restrictions on H-expressions do not hinder their use in representing declaration data. Types in IL presently include as atoms only identifiers, small integers, and NIL. Also, once a type has been copied from S-expression to H-expression, it need never be transformed the other way, since in all practical later use it is indistinguishable from an S-expression (properties 4 and 5). In addition to their usefulness for representing types, H-expressions can be of great value for many user applications.

Implementation of the function HCONS requires a hash to be performed on its two arguments in order to select a small subset of all existing hashed nodes (the structures created by HCONS) to be searched for one whose CAR and CDR fields are equal (EQN) to the first and second arguments, respectively. If such a node is encountered in the bucket, it is already guaranteed to be unique and a pointer to it is returned as the value of HCONS. If none such is found, a new node can be added to the subset with full confidence that it too will be unique.

The specific problem in a LISP primitive HCONS is in representing the partitioning of hashed nodes into subsets, termed <u>buckets</u>, consistent with the hash computation. Two possibilities come immediately to mind: the easier but more space-consuming is to keep (unhashed) lists of hashed nodes for each bucket; the less space-consuming but more complex solution is to divide the space containing hashed nodes into a number of areas equal to the number of buckets determined by the hashing algorithm, with an escape mechanism for an area overflow. A good compromise seems to involve representing a hashed node by three address-size fields: the CAR field, the CDR field, and a field to link the node to another one in the same bucket. This can be done easily on a machine such as the 360 at a cost of 1-1/2 words per node, provided a special hashed node zone is defined. This appears to be a tolerable price to pay for hashed nodes in view of the limited use anticipated for them. Heavier use of hash-CONSing than that now anticipated could justify a more elaborate and efficient scheme.

As a summary and conclusion to the foregoing discussion of type encoding for LISP 2, the following points can be made:

1. The syntax of declaration information will expand substantially with the coming of format and type definition features; increased subspecification in declarations will extend to structural units as well as to variable types.

2. To meet these needs, type encoding in terms of byte sequences must be abandoned in favor of S-expressions, or something similar, which can be processed more quickly and flexibly.

3. Hash-CONSing into H-expressions preserves all the advantages of an S-expression format for declarations while reducing the storage requirements by a large factor.

4. A new class of zones will have to be defined for unique nodes. These unique values must be compatible with other list nodes in the layout of the CAR and CDR fields. A reference into a zone of this class must be deemed non-atomic by ATOM.

6. SOFTWARE PAGING OF BINARY PROGRAM SPACE

LISP 2 systems, unlike LISP 1.5, are characterized by large memory requirements, to the extent that the total space requirement for BPI's alone exceeds available core storage. This growth of binary program space is a natural side effect of (1) the expected growth of the LISP 2 language; (2) the increased sophistication of the compiler and the resident general storage management facilities; (3) the addition of a number of program modules to the LISP 2 system, such as the syntax-directed facility, the pattern-matching facility, and the interactive edit/debug facility; (4) the traditional growth of program modules in the expected LISP 2 application areas.

The following paragraphs outline a proposed software paging scheme for the dynamic management of binary program space. The scheme assumes a fast random-access secondary storage device is available for the storage of binary programs. The scheme assigns memory residence to those binary programs that are immediately relevant to the current process, and depends on heuristic methods for automatic task recognition and automatic measurement of the relevancy of particular programs to the current task. The scheme operates dynamically, is largely independent of the automatic general storage reclamation provided by the garbage collector, and proceeds with little user interaction. A set of user-oriented commands provides substantial control of the status of binary programs at the points where user control is essential.

Such a scheme for the management of binary program space is motivated by the following considerations:

- A small fraction of the BPI's are necessary to process the current task;

- An increase in the amount of memory space available for structural units will result in an increase in the mean time between garbage collections, and a corresponding decrease in the overhead for general storage reclamation;

- A general-purpose program chaining mechanism will benefit the programmer in large-scale application areas;
- Binary program space should be modular, in the sense that the user always has easy access to the parts of the system relevant to his current program, and is not penalized for the parts of the system he does not use;
- Experience gained by simulating paging in our restricted sense will lead to some insight into the organization of binary program space most desirable in a hardware paging environment. Functions that are frequently part of the same task should be stored together on secondary storage in task clusters.

6.1 BINARY AND SECONDARY PROGRAM SPACE (BPS, SPS)

The compiled code for a LISP 2 function or procedure is called a binary image or a binary program. Some characteristics of binary programs are that (1) they occupy less space than the IL representation of the function or procedure they represent, and (2) they may be processed an order of magnitude faster than the corresponding IL representation. However, the essential characteristic of the binary program for the management of BPS is that (3) binary programs are a read-only data form. That is, once the compiled code for a function has been assembled, it may not be changed except through recompilation and re-assembly of a modified version of the original, with the current version being excised.

A BPI that is in memory is called a resident binary image (RBI). The primary characteristics of an RBI are that (1) it is executable, and may be used in the current task directly if it is required, and (2) it is completely relocatable, in the sense that its location in memory may be varied dynamically without affecting the outcome of the current task. The memory space reserved for holding RBI's is called binary program space (BPS). Because an RBI is relocatable, binary program space can be compacted whenever a particular RBI is deleted, and the memory space occupied by the RBI can then be used for other RBI's or temporary data structures.

Secondary program space (SPS) is an area on a random access storage device, and is used for storage of binary programs. A binary program that is stored in SPS is called a stored binary image (SBI). The characteristics of an SBI are that (1) it is not executable, so that if it is required in the current task, then it must be loaded into memory; (2) the address of the SBI is available to the resident binary program management facility, so that the SBI can be made resident by loading it from the random access SPS; (3) the SBI is written onto secondary storage once, under user control, when the SBI is first compiled, so that SPS is treated as a read-only device except for the initial recording and later reclamation; (4) the SBI is read into memory whenever it is not resident and is required by the current task.

There must be a garbage collector for SPS, though it may be completely independent of LISP 2. A LISP 2 dependent garbage collector would require

- The physical address of each SBI
- The garbage collection of SPS would involve copying secondary storage into secondary storage using the information held in the function descriptors, and updating the addresses of the SBI's in SPS. When an SBI is deleted from SPS, the function descriptor must be retained until the SPS address has been used by the secondary garbage collector.

A LISP independent garbage collector for SPS would require:

- A symbolic address assigned to each SBI at the time of its creation
- The garbage collector may reallocate SPS without changing the symbolic address, and so without affecting LISP.

6.2 USER CONTROL FACILITIES

The management of binary program space depends on the following control facilities:

- COMPILE and LAP, to produce in memory a relocatable binary image of a function.
- EXCISE, to remove all traces of a binary image.
- STOREBI, to store a copy of the binary image of a function on secondary storage, so that it may be reloaded when necessary.
- EXCISEBI, to remove the copy of the binary image from secondary storage.
- DELETEBI, to remove a binary image from memory when a copy exists on secondary storage, in order to reallocate the memory space occupied by the binary image.
- LOADBI, to generate in memory a binary image of a function from the copy on secondary storage. This is implicit whenever a function is called after the binary image was discarded with DELETEBI.

These facilities are described in greater detail later. The essential idea is that COMPILE and LAP are necessary to generate the binary image of a function, but they are not useful for regenerating the binary image, which would be necessary if the user had excised the binary image and then wanted to activate it.

By storing the binary image on secondary storage, the user attains a higher degree of freedom. The binary image in core may be regenerated by loading a copy from secondary storage. When the current core copy of the binary image is not actively involved in the current task, it may be deleted. Note that DELETEBI is merely a temporary removal from residence, whereas EXCISE is a complete purge.

The user may delete binary images under program control. This facility should be useful whenever the binary program requirements of a task can be anticipated. For the more general case of the absence of user control, a heuristic method for automatically deleting and reloading binary images is proposed. The method seeks to optimize the use of binary program space relative to dynamic storage requirements and relative to the cost of deleting and subsequently reloading binary images. The method is designed primarily to supplement the user's application of the DELETEBI and LOADBI facilities in accordance with current task requirements.

6.2.1 LAP

This is the interface between the compiler and the internal storage conventions. It is the final step in transforming symbolic code into executable code. The effect of LAP is to transform the S-expression output (LAP code) into a relocatable binary image (RBI) in memory. The symbolic references to structures and function descriptors in fixed space become explicit references. The count cells in the referenced fixed units are incremented, and a function descriptor for the RBI is created. All accesses to the RBI are accomplished via the function descriptor. This assembly process may be termed linking.

6.2.2 EXCISE

The effect of EXCISE is to destroy the effects of LAP and of STOREBI: the binary images of a function in memory and on secondary storage are made inaccessible to the user, and the memory space and secondary storage space reserved for the binary image are released for general re-allocation. An EXCISE is implicit before a function is assembled. If there is no binary image of a function, then the EXCISE has no effect. An EXCISEBI is implicit before an EXCISE.

If the function cannot be excised, then the value of EXCISE (fn) is NIL. A function cannot be excised if it is active on the pushdown stack. The detailed effects of the EXCISE are:

1. The function descriptor is made into an error trap so that the message FUNCTION EXCISED can be printed if the user attempts to activate the binary image before the function is recompiled.

2. For each reference from the binary image to a fixed unit, the corresponding count field is decremented.

The count field is a number associated with any fixed unit to indicate the number of references from BPS. References to fixed units (identifiers, quote cells, free variables, and function descriptors) are determined by examining the code itself or a bit map of it stored with the RBI. The core space occupied by a fixed unit is reclaimable if the reference count is zero and if there are no references to it from active list structure or from the pushdown stack.

After an EXCISE, a function may be restored only by re-assembling, which usually involves recompiling. Thus EXCISE is an expensive space-maker if the user wishes to run the function at some later time. The EXCISE is of limited usefulness, providing benefits only when the user's specific problem area is not serviced by the resident library functions, and when the user recompiled functions. It does not directly solve the problem of making space when LISP 2 is in the depths of recursion.

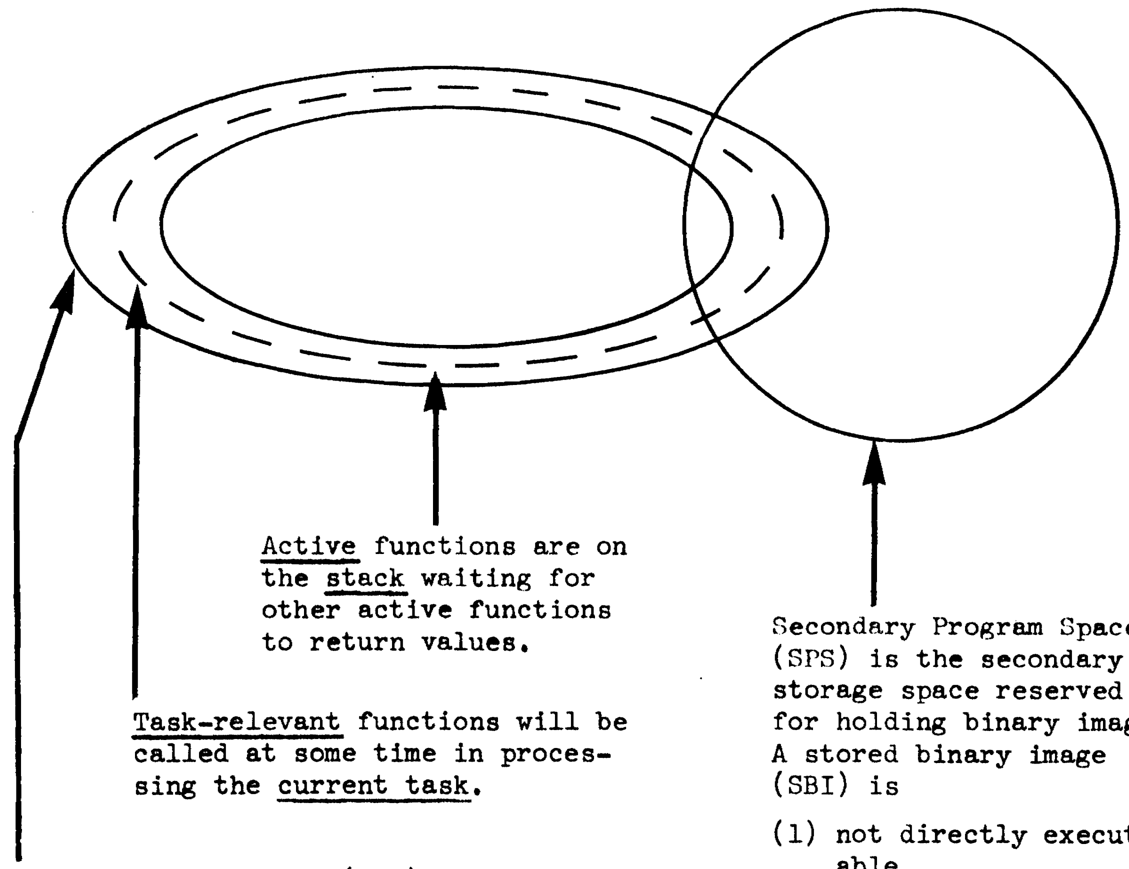
MAKE A LOADABLE BINARY IMAGE, (STOREBI _function_ _name_)

The RBI of the function is written onto secondary storage, and a field of the function descriptor is set to the address of the RBI in secondary storage. The field is zero if the RBI has not been written onto secondary storage. The copy of the binary image in secondary storage is termed a loadable binary image, or SBI. There is never more than one SBI for a given function.

Once an SBI has been created for a function, the RBI is redundant to the extent that it may be restored from secondary storage by a RELOAD operation. The SBI includes the relocation information normally associated with the RBI. When the RBI is not involved in the current task, the memory locations occupied by the RBI may be reclaimed by the garbage collector. When an RBI is deleted, the function descriptor must be replaced by a trap to a routine to load the RBI and re-activate the function descriptor.

6.3 AUTOMATIC TASK RECOGNITION

Included here are definitions of the concepts of _task_ and _task relevance_. The relation of these and other concepts in the management of binary program space is summarized in Figure 5.



Active functions are on the stack waiting for other active functions to return values.

Task-relevant functions will be called at some time in processing the current task.

Binary Program Space (BPS) is the memory space reserved for holding binary images. A resident binary image (RBI) is

(1) directly executable

(2) completely relocatable.

Secondary Program Space (SPS) is the secondary storage space reserved for holding binary images. A stored binary image (SBI) is

(1) not directly executable

(2) loadable into BPS if it is not resident and if it is required by the current task

(3) deletable from BPS whenever it is inactive.

Figure 5. Automatic Task Recognition

A task is the computation that occurs during the time between an input to the supervisor and the time the supervisor is ready for another input. A task is characterized by the set of functions called before the pushdown stack returns to the supervisory level. We refer to tasks as disjoint or independent if they involve disparate sets of functions. A supervisory level or dispatch level is a level of the stack corresponding to the portion of the stack reserved for the local variables of a supervisor. A supervisor or dispatcher is a binary program from which substantially disjoint tasks may be initiated. The possibility of several supervisory levels is a slight complication of the concept of task. We will think of the deepest supervisory level as the current supervisor, and any task initiated from that level will be called the current task.

A sequence of inputs to the LISP 2 supervisor will ordinarily include several disjoint tasks, and often the binary programs involved in all the tasks cannot reside in memory throughout the processing of the inputs. It is important to anticipate the binary program requirements of each task so as to reduce the overhead due to loading and unloading binary programs for the different tasks. It is also desirable to react to task requirements automatically, to avoid burdening the user with "housekeeping" details.

A task may be anticipated simply by listing the functions to be used in processing the task. This is burdensome, error-prone, and difficult to do in general. An alternative is to consider the set of functions referenced by a given function, together with all the functions referenced by those functions, etc. The advantage of this alternative is that the set of referenced functions can be derived automatically from an initial function. The disadvantage is that it overestimates the requirements of the task: too much memory space is allocated for functions that are not subsequently involved in the task.

The proposed solution is based on the following observations. Functions that belong to the same section are written and compiled together, and they tend to be involved in similar tasks. Conversely, functions in different sections tend to belong to disjoint tasks. Of course, these observations do not hold for section LISP, which contains miscellaneous functions of general utility.

The primary advantage of identifying tasks with sections is that it capitalizes on the existing LISP 2 structure to solve two important problems associated with task anticipation:

1. The user has a simple, natural method for describing the set functions associated with a particular task. He accomplishes this by organizing his function definitions into functionally connected groups called sections.

2. Automatic task recognition can be performed efficiently by measuring the percentage occupancy of the pushdown stack for each section.

Associated with each function descriptor is a task relevance index (TRI). The TRI is a real number in the range 0 to 1 which heuristically measures the frequency with which the resident binary image (RBI) for a function is necessary to process the current task. The TRI varies dynamically. It is operationally defined by a reward-and-punishment scheme as follows:

1. The TRI is initialized to 5.

2. After each census of BPS, the TRI for a function

   a. is rewarded if the binary image of the function is active,

   $$TRI \leftarrow \theta * TRI + (1 - \theta)$$

   b. is punished if the binary image of the function is resident but inactive,

   $$TRI \ \theta * TRI$$

   c. is ignored if the RBI is not in memory.

This scheme has the following properties:

1. The TRI converges to the percentage of times the TRI was rewarded relative to the number of times the TRI was either rewarded or punished.

2. A smaller value of θ produces faster reaction of the TRI, whereas a larger value produces a slower, more stable convergence.

A census is a survey of the binary images in the system for the purposes of determining their status with respect to activity, deletability and task relevance. A binary image is active if there is a return address to the RBI on the pushdown stack. A binary image is deletable if it is not active, and if there is a corresponding SBI on the secondary storage. The task relevance is a combination of the task relevance index and the activity of functions belonging to the same section.

The purpose of the census is to recover a predetermined amount of memory space by deleting, if possible, the irrelevant RBI's from BPS. A census occurs when a "growing pain" is invoked by the general storage reclaimer for the purposes of increasing the amount of free storage. A census also occurs when a function required by the current task is not resident in BPS. This latter condition is recognized because the function descriptor for the function contains a trap to the LOADBI routine.

The concept of task relevance is sufficiently flexible so that a threshold may be set before the census, such that all deletable RBI's below the threshold in relevance will be deleted. A problem that is not well understood as of this writing is what to do if an insufficient amount of BPS is released by the census. One alternative is to lower the threshold, but proper design of the threshold setting should eliminate this alternative. If all the resident binary programs are strictly not deletable, then the required space can be obtained only by invoking the general storage reclaimer, which may result in a growing pain.

The heuristics associated with the management of binary program space are directed entirely at selecting which of the deletable RBI's should actually be deleted. If all the functions to be used by a task are in memory at the start of the task, then there will be no overhead due to loading of SBI's from SPS, provided a census is not invoked either by the general storage reclaimer or inadvertently by the user through the application of the LOADBI function. The heuristics seek to retain as residents those RBI's that have a good likelihood of being used again before the next census. Also, the heuristics seek to expel those residents that are unlikely to be used again in the near future. One of our basic assumptions is that recent activity is a reasonable measure of the immediate future activity. And one of our basic goals is to minimize the memory requirements of binary programs.

6.4 REMAINING ISSUES

Many of the details of the software paging scheme have been left unclear, simply because little experience with the scheme has been obtained. One of the major questions to be settled through experimentation is the computation of the task relevance of a given function.

The section name can be related to task relevance by measuring the percentage occupancy of the stack by the functions in the section. It may be better to take a weighted percentage, such as the sum of the TRI's of active functions in the section. Alternately, it may be desirable to keep a section index, to vary dynamically, and to measure the recent overall activity of the section, or the cohesion of the section, etc.

Other variables have been proposed as relevant to the question of whether or not a deletable RBI should be deleted. These miscellaneous variables include: (1) the size of the RBI, because it is cheaper per instruction to load large functions and release the space; (2) the frequency of loading; (3) the time of the last load; (4) the frequency of calls of the function during the current task; (5) the static call count, or the number of references to the RBI from BPS; (6) the number of calling sequences in the RBI; (7) etc.

It is likely that the accuracy of the task relevance index (TRI) can be improved by setting up a "reward trap" after a census for the deletable RBI's. The purpose of the trap would be to reward the TRI if the RBI was used before the next census.