FACT SEGMENTATION

Martin N. Greenfield

Minneapolis-Honeywell EDP Division

Wellesley Hills, Massachusetts

The FACT Compiler is Honeywell's English language narrative compiler used for commercial data processing applications. The program segments created by FACT have their position in memory dynamically relocated in order to make the most efficient use of the available core storage. The methods employed in this operation will be described as they are general in scope and of sufficient merit to find use in other applications.

Segmentation

Segmentation is the process of dividing a single program into pieces. This is done to permit the operation of programs that are too large to completely fit into memory. The pieces of the program are loaded only when needed. By having these segments time share areas of memory, the program may be executed.

The importance of segmentation has grown with the size of programs being produced. This has been accentuated by the popularity of compiler usage. It has become easier and as a result practical to write larger programs attacking larger problems. Divorcing the compiler user from machine considerations tends to have him create larger programs. The compiler tends to generate code that is more general and requires more space than the human created codes. The result is that every major compiler must give careful consideration to segmentation provisions.

Static and Dynamic Allocation

The customary practice of assigning memory in segmentation operations is static allocation. At compilation, assembly, or scheduling time, the area available to each segment is determined. The space is assigned and the segment always occupies the same locations whenever it is loaded. During execution, the flow of the program determines when each segment is loaded.

More recent developments have led to the usage of dynamic allocation of segments. At execution time both when a segment is to be loaded and where it is to go is determined. Allocation is determined by the monitor and can vary for subsequent loadings of the same segment.

Dynamic allocation requires a much more complicated loading and control routine than its static counterpart. More time is required to load a segment because it must have its contents conditioned to the positions it is to occupy. Inter segment communication is more involved because the positions of the segments are not predetermined.

Despite these penalties, the FACT Compiler design uses dynamic allocation for its object program segmentation primarily because it is so much more flexible and economical in its use of space. Applications such as data processing and information retrieval are characterized by handling large volumes of input having widely varying processing requirements. It is generally not known at the time a program is generated what the mix of processing segments and storage areas should be during each period of program execution. These requirements are dependent on the input. Only a data sensitive segmentation control can precisely determine what minimum portions of a routine must be present. Static allocation must compensate for this by keeping segments present for the worst case. The more flexible dynamic allocation conserves space by reacting only as needed. Segments that are unneeded are not kept in memory. Segments are loaded into what space is available at the time they are entered rather than requiring a specific area be designated for them. Because of this conservation of space, object code placing emphasis on execution speed at the expense of memory may be generated.

Characteristics of FACT Segmentation

Generally segments are composed of all of the instructions, constants and data storage required for their execution. In order to take the fullest advantage of the dynamic control, FACT segments are subdivided into smaller units. The smaller the unit, the greater can be the control over what must be present in memory.

Each major procedure, input edit procedure, report procedure, or internal or external file area is a segment. The advantage in separating segments from their file areas can best be clarified by example. A procedure may be operating on File A and File B. Upon terminating the operations on File B, the space used by File B may be made available to some other segment while operations continue on File A.

The mix of segments present during any period of execution is dictated by the flow of the program and is in turn sensitive to the processing requirements of the input. The storage occupied by a segment is released as soon as it is no longer required and this storage becomes available to any other segment.

Customarily, reloading a segment requires that it always be obtained from the secondary storage (program tape, disc, drum). In FACT, a check is made to determine whether a called segment that has been released is still intact in memory. This allows the operation to tend to release segments as soon as possible. If they are required a short time thereafter, they may be activated quickly without paying for a secondary storage access. Coupled with this feature is the tendency built into the control of keeping released segments intact as long as possible.

FACT segments are generated as relocatable blocks of code. Every segment must be operated on at the time it is loaded to make it executable. The location into which a segment is loaded is determined at the moment it is to be loaded. This location depends on the available configuration of space and will be described in more detail. Each segment is relocatable within memory. Segments are frequently moved and rearranged in memory in order to accomodate the loading of other segments.

The Segmenter

The routine that controls the segmentation operations is called the Segmenter. It is the initial segment of each program. It operates with either the Honeywell production monitor (Executive System) or checkout monitor (Program Test System) in activating, loading, releasing and relocating segments.

The Segmenter uses two tables. The Segment Control Table has an entry for each segment indicating its size, its base location (current address of the first word of the segment), its status, and its follower. The status can be active (A), inactive but intact in core (I), or out of core (O). The follower is the number of the adjacent segment in higher addressed core.

The Bank Control Table has an entry for each memory module (2048 words). The entry indicates the first and last segment numbers in the bank, the currently available inactive storage, the highest and lowest locations scheduled for usage in this bank, and the lowest location that has never been loaded. The highest and lowest scheduled locations are carried because the Honeywell 800 has multiprogramming capability. A scheduler defines the areas of memory available to each program. A FACT program must be scheduled for a consecutive set of registers. Other programs may be in parallel operation in the same bank but outside the scheduled area for the FACT program. Figure 1 shows both the Segment Control Table and the Bank Control Table.

The Segmenter normally relocates within the confines of a bank. No Segment is located such that it occupies portions of two banks. Because FACT segments tend to be small, this is not a restriction.

Segmenter Operation

By example using figures 1-6, typical Segmenter operations will be described.

Each FACT program initially loads the Segmenter as its first segment. After initialization, the first segment actually generated from source code is loaded. When other segments are addressed, the Segmenter will control their entry. Segments no longer needed will be released or deactivated by the Segmenter.

Following a short execution period, a typical memory layout and the corresponding tables in the Segmenter are pictured in Figure 1. The Executive System Monitor and the Segmenter are in bank 0. Active segments 3 and 5 are in bank one. Segments 1 and 6 in bank 0 and segment 8 in bank 1 were loaded and subsequently released. They are in an inactive status as indicated by the cross-hatching.

Examining the Segment Control Table entry for segment 5 shows that it starts in bank 1, location 200. It is 600 registers long, is in active status (A), and is followed in core by segment 8. The Bank Control Table shows that segment 0 (the Segmenter segment) is the first and segment 6 is the last segment in bank 0. An area from bank 0 location 500 to bank 1, location 1799 has been scheduled for this program. 1000 registers of inactive storage are available in bank 0. Location 1800 is the lowest location in bank 0 that has never had a segment loaded into it.

Activate Inactive Segment (Figure 2)

At this point if segment 1 is addressed, the Segmenter will discover from the Segment Control Table that is is intact in core. Segment 1 status is changed to active and the available storage in bank 0 is reduced by its length. Since this process is rapid and required no accessing secondary storage, it is attractive to release segments as soon as possible knowing that they may be reactivated rapidly.

Load Segments into Unused Storage (Figure 3)

When segments 9 and 10 are addressed, they must be loaded from secondary storage. The Segmenter will load them into areas of unused storage if possible so as to avoid destroying inactive segments. Preserving inactive segments will save accesses to secondary storage should they be required again. Adding the segment length and the field in the Bank Control Table designating the lowest location unloaded permits the Segmenter to determine that segments 9 and 10 will not fit into the end of bank 0, but will fit into the end of bank 1. The loader in the

monitor will search for the segments and load them where the Segmenter has specified. The Segmenter tables are then updated (base, status and follower fields of segments, last segment, unused storage, lowest location unloaded of bank 1).

Release Segment (Figure 4)

Releasing segment 5 has the Segmenter change the segment's status and add its length to the available inactive storage in the Bank Control Table. Segment 5 is still intact and may be reactivated easily if required again.

Load Segment into Opening (Figure 5)

Scanning the Bank Control Table will indicate that segment 7 will not fit at the end of any bank, but can fit into the inactive storage of bank 1. There is no inactive contiguous storage large enough to contain segment 7. The Segmenter must collect an adequate area by relocating the active segments in bank 1.

The relocation process can best be visualized by picturing the active segments in bank 1 as loosely strung beads. The gaps of string between the beads represent the inactive storage. The bead following the bead at the beginning of the bank is moved adjacent to it. The gap created below the moved bead is examined to see if the called segment will fit. If it will not, another bead is moved adjacent to the moved bead until an adequate gap is developed. The called segment will then be loaded into the gap.

To perform this operation it is necessary to know the sequence of segments in the bank. This is found starting with the first segment as indicated in the Bank Control Table (segment 3). In each Segment Control Word, the segment number that follows is shown in the follower field. Thus, segment 3's follower is segment 5, 5's follower is 8 and so forth. (This refers to the segment control table indicated in figure 4).

Scanning down the chain of segments, the last of the first group of active segments can be found. This segment is the head bead of the analogy, (segment 3) Following it will be a group of inactive segments (5 and 8). The bead to be first moved is the first active segment (segment 9) following the group of inactive segments . Segment 9 is moved next to segment 3. By doing this, inactive segment 5 is overlaid. Since segment 5 can no longer be directly activated, its status is changed to out of core (0). Segment 10 will also have to be moved before segment 7 will fit. After this, segment 7 is loaded into the bank. Segment 7 will overlay segment 8, so it is necessary to change segment 8's status to out of core (0).

All of the above operation seems to be quite involved and yet its operation is barely perceptible when watching it at the H-800 console.

Squishing (Figure 6)

Segmenter operations described so far have been confined to action within one bank. Should a segment be called that is larger than the inactive storage in any one bank, the inactive storage in all of the banks is accumulated in an attempt to fit the segment. To do this, the Segmenter employs a subroutine called the Squisher. This routine, whose name sounds like a fugitive from a science fiction movie, will coagulate the active segments into the lower addressed banks and locations available to the program. Where necessary segments will cross into other banks. Like all reputable science fiction creatures, the Squisher is destructive in that it destroys all of the inactive segments in memory. In return it collects all of the inactive space together into upper memory. This should enable the loading of the called segment that otherwise could not have fit.

Another use of the Squisher is in preparation for executing a sort. Sorts are amorphous by nature and operate faster if given more space. The Squisher operation gathers all available inactive space for the sort and informs the sort generator of the amount of space available.

Relocation, Communication, Linkage

The explanation thus far has been about when and why segments are relocated. The next sections will explain how the relocation is performed, how the migrant segments can successfully communicate with each other, and what the linkage is that brings the Segmenter into action.

Relocation

The execution of any code is sensitive to its location in storage. When segments are moved or loaded, their code must be adjusted to the new environment. When loading a routine from a secondary storage as tape, it is possible to have information on how to relocate each word delivered to the loader. This relocation information is used during loading and does not permanently reside in memory. Unavailability of relocation information for the memory to memory relocation of FACT requires another method of determining the relocation rules.

A Segment is generated as a block of code that must be moved together. When a segment is moved, the incremental address is the same for every word in the segment. The segments are constructed in the structured format shown in Figure 8. Each word of the segment that follows the same relocation rules is assembled into the same region. One word in the segment defines the length of each of these regions. It then becomes possible to have a relocation

routine operate on each of the regions and make the appropriate modifications.

Each segment is kept on the program tape though it were to be loaded into locations starting at zero. The relocation increment for a program being loaded is the intial loading address (or base). The memory to memory relocation subroutine is used to perform the relocation. Relocation data need not be kept on the program tape nor must the loader be concerned with this process.

Intersegment Communication

Segments must be able to operate on information in other segments, transfer information between segments, and transfer control to other segments. The mobility of FACT segments requires a communication device for these functions. This centers about a table called a Reference Region that is part of each segment.

The Reference Region is a table of addresses of locations in other segments that are addressed by the segment. These entries contain initially the number of the segment in one field and the location relative to address zero within that segment that is being addressed in another field. Whenever a segment is loaded or moved in memory, its relocation increment is used to update the Reference Region tables of all other segments of that program that are in memory. Whenever a segment is loaded, its Reference Region table is updated to reflect the positions of those segments that it addresses that are in memory. It is possible to rapidly scan and update the Reference Region tables by using the Segment Control Table. The segments may freely communicate using the information kept current in their Reference Regions.

The updatings of the Reference Region of segment 3 of the example used before is shown in Figure 7. Addressing between segments is always done using either indexed or indirect addressing features of the Honeywell 800. The entries in the Reference Region are arranged so they may be loaded into the special registers for this mode of addressing.

Linkage to the Segmenter

The Segmenter is dormant until called to load, activate, or release segments. When the Segmenter is active, the program it controls is kept dormant. (This need not be the case because of the parallel processing capability of the Honeywell 800). Other programs running in parallel do keep operating. It is necessary to stop the operation of the program calling the Segmenter because some of its segments in memory may be relocated, a segment may have to be loaded from secondary storage, and other calls to the Segmenter must be inhibited.

Calls to the Segmenter to release a segment occur at points in the code that can be defined a compilation time. The calling linkage is a simple direct entry similar to those used in entering monitors.

Linkage to activate or load a segment are quite different. The system is designed to allow segments to easily address each other and at the same time have the segments activated and released frequently. It would be most inefficient in time and storage if each instance of intersegment addressing had to require testing to determine whether the segment was active in memory.

The method employed relies on the fact that intersegment communication always uses indirect or index addressing that employs the words in the Reference Region table to load these special registers. Whenever a segment is inactive, all Reference Region words pertaining to it are forced to bad parity. When a segment loads the special register with the Reference Region word, the Honeywell 800 forces an automatic unprogrammed transfer of control to a specified location. The location transferred to is the start of the linkage to enter the Segmenter.

The hardware is such that no sequence counter or any other settings are disturbed by the unprogrammed transfer. After the Segmenter operation, it becomes very easy to restore control to the calling segment as none of its settings have been changed by the linkage. Thus, a hardware feature of the Honeywell 800 is used as an automatic test sensitive to the absence of an addressed segment.

Since a hardware error detection feature is being used in this linkage, the Segmenter must verify that this is a legitimate call and not a machine malfunction. The Segmenter reduces the probability of a mistaken call by verifying that the location of the word of bad parity resides in a Reference Region table and the Segment number referred to is inactive. The reliability of the H-800 memory further reduces the probability.

Operation with the Monitors

The Segmenter has been designed to work with both the Program Test System and the Executive System monitors.

The Program Test System, Honeywell's checkout system, allows dynamic or snapshot dumping of memory areas and tapes during the program operation. The dumping occurs at points where the programmer requests derail linkages to the monitor. The requested dumping is identified by the location of the derail. The mobility of the segments created

a problem that was handled by including the segment number in the derailing instruction. Use of the Segment Control Table allowed a routine to convert the derail location to a location relative to location zero. The particular derail could then be identified. The instructions dumped are converted via use of the Segment Control Table to locations relative to zero. Thus, the programmer need not be hindered by the actual position of the segment while it was being dumped.

The Executive System will be the primary monitor working with the Segmenter. Each FACT program is given an area of memory to operate in at scheduling time. The full power of the Executive System Monitor is available to the FACT programs. It is used to set restart points, restart individual programs, execute a full schedule of parallel operating routines, test the validity of the schedule, etc. Loading of segments is done by the Run Supervisor after receiving the information from the Segmenter. The search for a new segment on the Production Run Tape is always optimized to the proper direction.

Operating Experience

That described in this paper is not a proposed or theoretical system but one that has been in daily operation since August 1961. There have been embellishments to improve the Segmenter, but it has not changed in its basic operation.

In September 1961, the Segmenter was tied into the Executive System. Runs have been scheduled and executed involving combinations of two FACT programs and an assembly language program all operating in parallel. In that operation there are periods where five of the eight independent sequence counters in the H-800 are active. All eight of the sequence counters were used at some time during this operation.

Future Modifications

A number of ideas have been suggested to add to the flexibility and attractiveness of the Segmenter. Currently the Segmenter requires about 500 memory cells. There is a separate segmenter for each FACT program being run in parallel. A technique has been developed to have only one Segmenter for all FACT programs in operation. The FACT programs could have their segments intermingled in memory instead of having distinct areas as is now required. The most difficult part of this design would be the setting of restart points. The Segment Control Table would be employed to identify the areas occupied by the segments of the program for which a restart was being set.

To further conserve memory, the operations of the Segmenter would themselves be segmented.

The Segment Control Table is the source of information to the Segmenter as to the size of each segment. This table could have its entries changed by the program and enable specified segments to grow in size according to the dynamic needs of the routine.

The Segmenter makes no attempt to save the settings of a segment it is about to destroy. Such a segment could be dumped into secondary storage prior to its destruction. A choice could be made when the segment was again called as to having it loaded in its initial state or in the state it was in the last time it appeared in memory.

Acknowledgements

Acknowledgement should be given to the staffs of Computer Science Corporation, Inglewood, California, Philip Hankins and Company, Inc., Arlington, Massachusetts and the Honeywell FACT Group for their work in the design and implementation of that described in this paper.

SEGMENT CONTROL TABLE

SEG. NO.	STATUS*	LENGTH	FOLLOWER**	LOCATION
0	A	500	1	0,0500
1	I	500	6	0,1000
2	0	400		
3	A	200	5	1,0000
4	0	600		
5	A	600	8	1,0200
6	I	300	L	0,1500
7	0	900		
8	I	200	L	1,0800
9	0	300		
10	0	300		

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC	LOWEST LOC UNLOADED
0	0	6	1000	500	1999	1800
1	3	8	1000	000	1799	1000

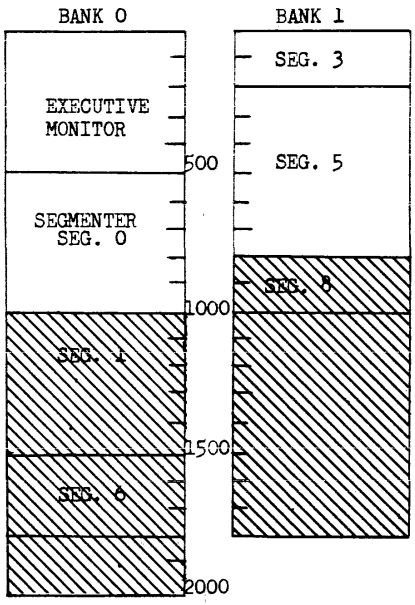


Fig. 1 STARTING CONDITION OF EXAMPLE

SEGMENT CONTROL TABLE (PARTIAL)

SEG. NO.	STATUS	LENGTH	FOLLOWER	LOCATION
0	A	500	1	0,0500
1	A	500	6	0,1000
2	0	400		
etc.				

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC.	LOWEST LOC UNLOADED
0	0	6	500	500	1999	1800
1	3	8	1000	000	1799	1000

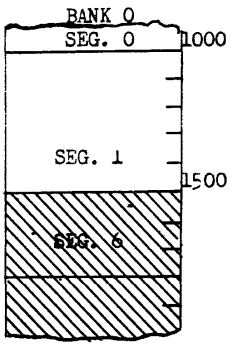


Fig. 2 ACTIVATE INACTIVE SEGMENT 1

* A = ACTIVE; I = INACTIVE, in HSM; 0 = OUT OF HSM
** L = LAST SEGMENT IN BANK

SEGMENT CONTROL TABLE

SEG. NO.	STATUS	LENGTH	FOLLOWER	LOCATION
.				
.				
8	I	200	9	1,0800
9	A	300	10	1,1000
10	A	300	L	1,1300

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC.	LOWEST LOC. UNLOADED
0	0	6	500	500	1999	1800
1	3	10	400	000	1799	1600

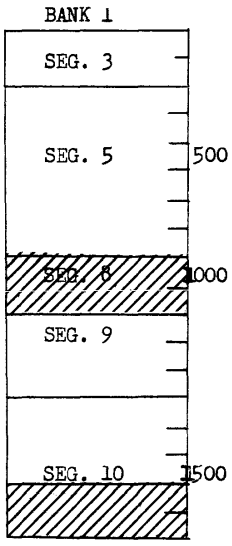


Fig. 3 LOAD SEGMENTS 9 and 10 INTO END OF STORAGE

SEGMENT CONTROL TABLE

SEG. NO.	STATUS	LENGTH	FOLLOWER	LOCATION
.				
.				
4	0	600		
5	I	600	8	1,0200
6	A	300	L	0,1500

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC.	LOWEST LOC. UNLOADED
0	0	6	500	0500	1999	1800
1	3	10	1000	0000	1799	1600

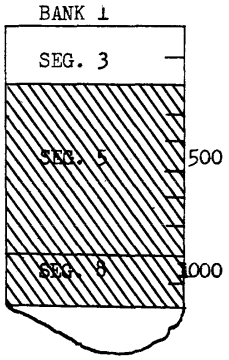


Fig. 4 RELEASE SEGMENT 5

SEGMENT CONTROL TABLE

SEG. NO.	STATUS	LENGTH	FOLLOWER	LOCATION
0	A	500	1	0,0500
1	A	500	6	0,1000
2	0	400		
3	A	200	9	1,0000
4	0	600		
5	0	600		
6	I	300	L	0,1500
7	A	900	L	1,0800
8	0	200		
9	A	300	10	1,0200
10	0	200	7	1,0500

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC.	LOWEST LOC. UNLOADED
0	0	6	500	500	1999	1800
1	3	7	100	000	1799	1700

FIG. 5 LOAD SEGMENT 7 INTO AN OPENING

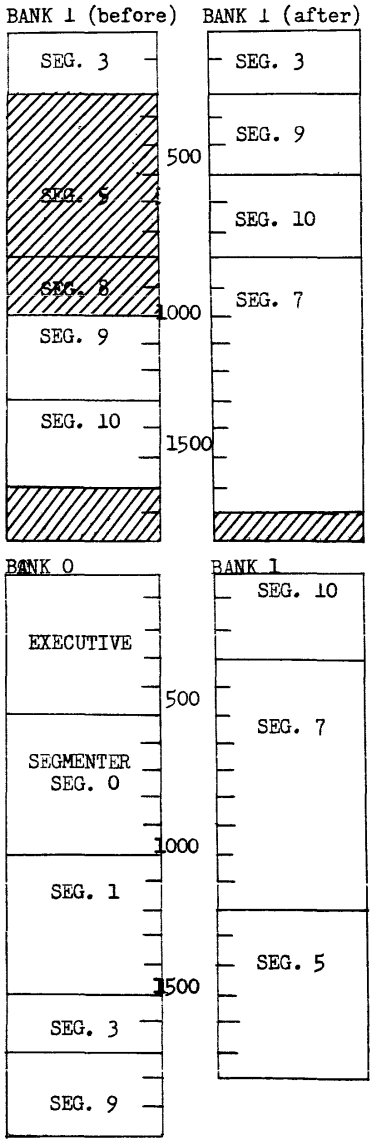
SEGMENT CONTROL TABLE

SEG. NO.	STATUS	LENGTH	FOLLOWER	LOCATION
0	A	500	1	0,0500
1	A	500	3	0,1000
2	0	400		
3	A	200	9	0,1500
4	0	600		
5	A	600	L	1,1100
6	0	300		
7	A	900	5	1,0200
8	0	200		
9	A	300	L	0,1700
10	A	200	7	1,0000

BANK CONTROL TABLE

BANK NO.	FIRST SEG.	LAST SEG.	UNUSED STORAGE	LOWEST LOC.	HIGHEST LOC.	LOWEST LOC. UNLOADED
0	0	9	000	500	1999	2000
1	10	5	000	000	1799	1800

FIG. 6 LOAD SEGMENT 5 AFTER SQUISHING



SEGMENT 3
REFERENCE REGION

TAG	REFERENCED SEGMENT	LOCATION	BAD PARITY
R1	SEG. 1	0,0210	x
R2	SEG. 5	1,0500	
R3	SEG. 9	0,0181	x
STARTING CONDITION			
R1	SEG. 1	0,1210	
R2	SEG. 5	1,0500	
R3	SEG. 9	0,0181	x
SEGMENT 1 ACTIVATED			
R1	SEG. 1	0,1210	
R2	SEG 5	1,0500	
R3	SEG. 9	1,1181	
SEGMENTS 9 and 10 LOADED			
R1	SEG. 1	0,1210	
R2	SEG. 5	0,0300	x
R3	SEG. 9	1,1181	
RELEASE SEGMENT 5			
R1	SEG. 1	0,1210	
R2	SEG. 5	0,0300	x
R3	SEG. 9	1,0381	
LOAD SEGMENT 7 (MOVE SEGMENT 9)			
R1	SEG. 1	0,1210	
R2	SEG. 5	1,1500	
R3	SEG. 9	0,1881	
LOAD SEGMENT 5 (MOVE SEGMENT 9)			

FIG. 7 CHANGES TO SEGMENT 3 REFERENCE REGION WORDS

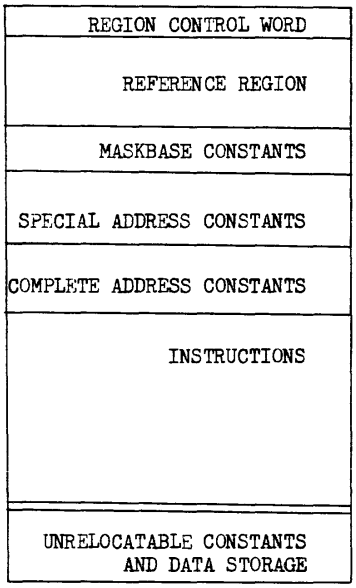


FIG. 8 TYPICAL FACT SEGMENT STRUCTURE