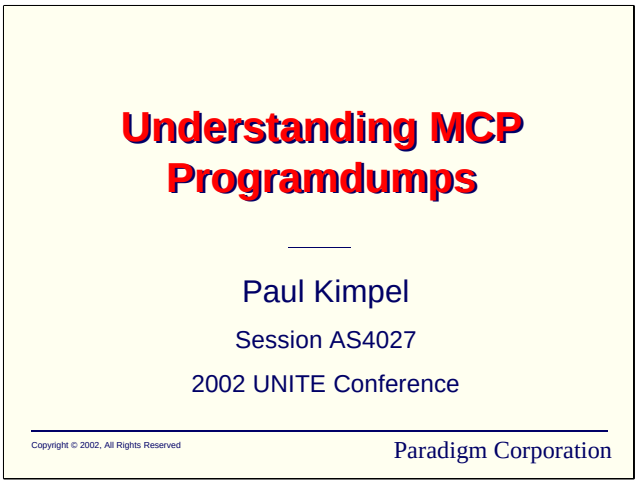


Understanding MCP Programdumps

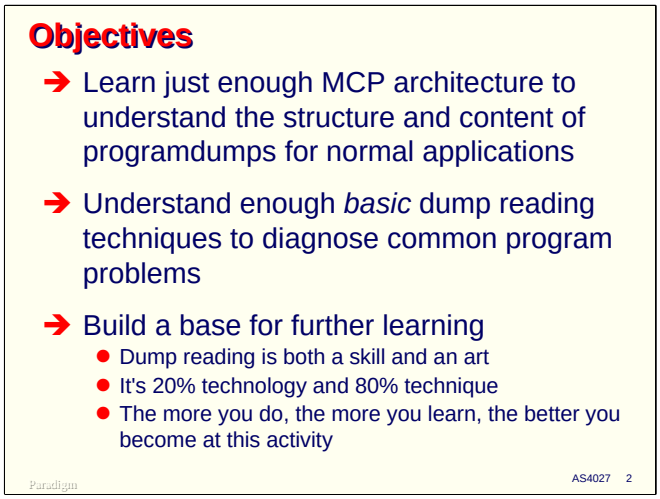
2002 UNITE Conference, Baltimore, Maryland.

Paul Kimpel

Paradigm Corporation
San Diego, California

http://www.digm.com

e-mail: paul.kimpel@digm.com



This presentation discusses programdumps and how they can be used under the Unisys ClearPath MCP.

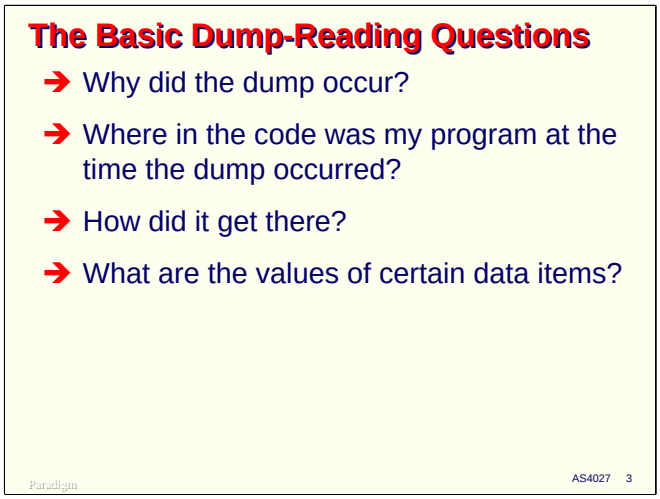
Memory dumps, dump reading, and problem diagnosis are large, complex subjects. Most of us who have been reading dumps and diagnosing program problems since the early days of the MCP architecture are still learning how. Since this presentation is limited in time, we need to set some objectives to get the most out of that time.

The primary goal of this presentation is to understand the basics of dump reading – enough so that you will be able to analyze programdumps and diagnose most common program problems.

A secondary goal is to understand just enough of the machine architecture used with MCP systems so that the structure and content of a programdump will make sense.

Finally, we want to build a base for further learning, so that as you read dumps, you'll get better at it and be able to take on the diagnosis of more sophisticated problems. Dump reading and problem diagnosis are not cut and dried tasks – they are both an art and a skill, and take time to develop. Like most things in computing, mastery of this area is at best 20% technology. The rest is technique, which you get only through practice and by working with others who are ahead of you on the learning curve.

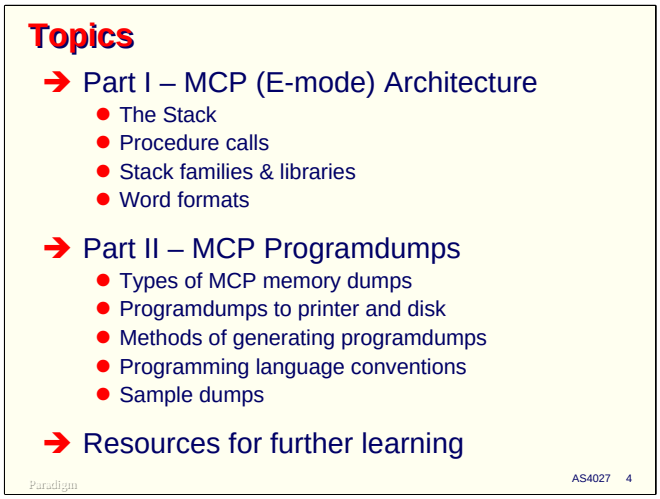
The more dump reading you do, the more you learn about both the technology of your programs and the MCP, and the better you become at diagnosing problems.



There are four main questions you should be asking when approaching the task of reading a dump. They are:

- Why did the dump occur? What event is responsible for the dump taking place? As we will see, there are a number of causes of dumps, including:
  - Program-requested snapshot
  - Operator-requested snapshot
  - Termination due to a fault of some sort
  - Termination due to MCP error checking
  - Termination by the operator or another program
- Where in the code was my program at the time the dump occurred? You often do not need a dump to answer this question, but it's usually important to know this in order the delve into the problem deeper.
- How did the program get to that point? What is the history of procedure calls or `PERFORM` statements which led the program to the routine where the dump occurred? In programs which simply branch around, this can be a very difficult, and often impossible, question to answer. For programs that are structured as a series of well-behaved subroutines, though, the dump will often tell you this directly.
- What are the values of certain data items when the dump occurred? This is often the most important type of information a dump can yield. The particular data items you need to examine depends entirely on the particular program you are looking at and why it has gone awry. Finding the locations of your program's variables in the dump is probably the most important thing we will talk about in this presentation, and due to the complexity of the MCP architecture – especially in the area of addressing memory – the one we will spend the most time preparing to answer.

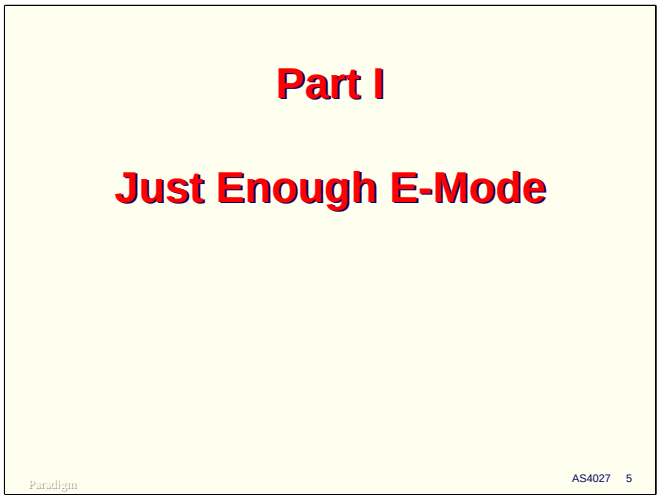
Try to keep these four questions in mind as we talk about the MCP architecture and begin to explore the structure of programdumps.



With that discussion of objectives, this presentation is divided into two main parts:

- In the first part, we will briefly introduce several features of the MCP architecture, also known as E-mode. We will talk about stacks, how procedure calls work, the role of stack frames and libraries, and discuss a few of the basic word formats used by the hardware.
- In the second part we will talk about programdumps themselves, and try to apply the topics from the architectural discussion to the art of reading a dump and diagnosing program problems. We will also talk about some of the differences between dumps for programs from different languages, and will look at simple sample dumps for Algol, COBOL-74, and COBOL-85.

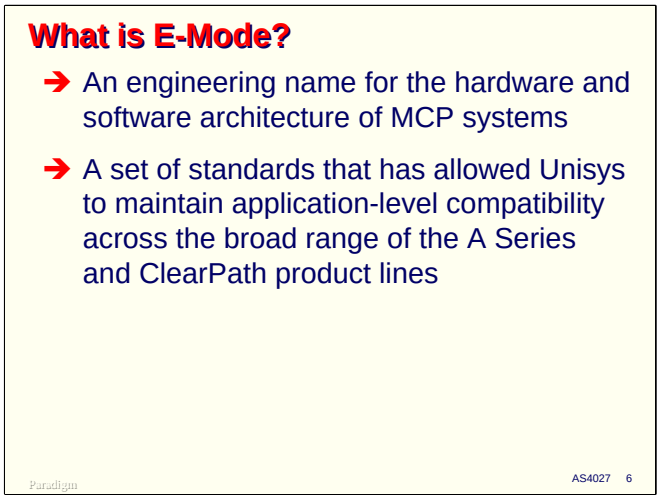
Along the way, I'll discuss some resources that you will be able to use to further your knowledge of the MCP architecture and dump reading.



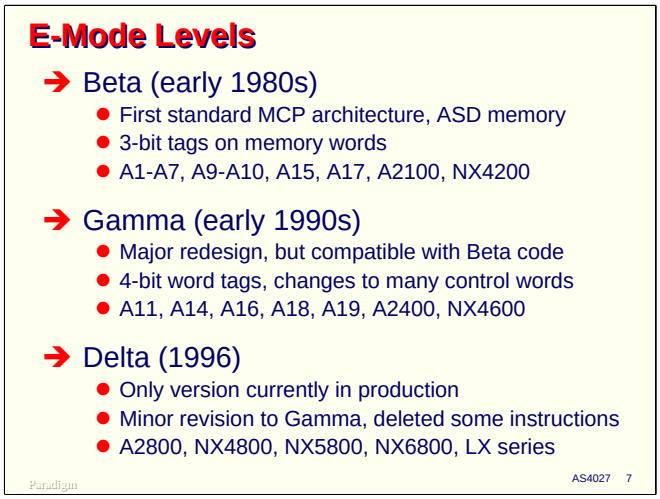
Let's start then with a discussion of just enough of the MCP's E-mode architecture to support basic dump reading tasks.

You should note that this is heavy stuff. The MCP architecture is by far the most complex one on the planet for a commercially available system, and may be the most complex one overall. It's brilliant and elegant, but takes some getting used to, so if you find this confusing, that's good – that means you're paying attention.

You need to know most of the following discussion to read Algol dumps. Less depth is required for most COBOL dumps, although the Algol-oriented issues are good background for all dump reading.



E-mode is simply an internal Unisys engineering term for the hardware and software architecture of the MCP. It also describes a set of internal standards that has allowed Unisys (and Burroughs before it) to maintain such good application-level compatibility across the broad range of A Series and ClearPath machines which have been produced over the last 20 years.



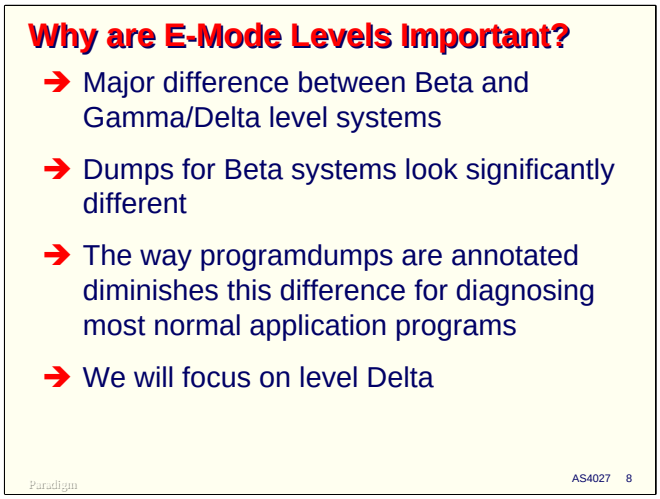
E-mode has gone through a number of revisions over time, and it's possible that it will go through more in the future. These versions are named using the letters of the Greek alphabet.

The first version of E-mode was "level Beta," which was first released in the early 1980s. This was the first standard MCP architecture. Prior to this, the B6000 and B7000 "Large Systems" were fairly compatible at an application program level, but were being designed and produced by separate internal organizations which tended to go somewhat their own way. As the product line and customer base grew, this was beginning to cause serious problems, so Burroughs attempted to pull the entire line together with the E-mode standard. The first machines to use this version were the A9 and A10, which also introduced the new memory addressing scheme, Actual Segment Descriptor. ASD memory allowed Burroughs to break out of the 6 MB addressing limit of the B6000/7000 series systems. The last systems produced using this level were the MicroA, A7, A2100, and NX4200.

Note that there never was anything which was called "level Alpha," but you could probably consider the B6000/7000 series to be that.

The second version of E-mode, level Gamma, was introduced in the early 1990s with the A11 and A16 systems. It was a major redesign of the architecture. The most notable difference was the expansion of word "tags" from three bits to four, changes to the instruction set, and extensive changes to many of the control words used internally by the hardware and MCP. We will talk more about word tags later in this section. The last machines produced using this level were the A2400 and ClearPath NX4600.

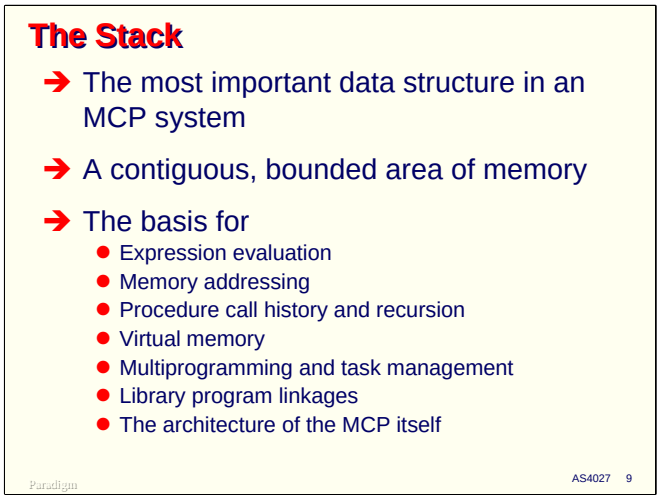
The current version of E-mode is level Delta. This was introduced in 1996 with the advent of the ClearPath NX4800. It is a less radical change to the architecture than the transition from levels Beta to Gamma, but the instruction set was changed in ways that made some programs compiled for Beta-level machines incompatible. This is why many of us had to undergo extensive recompilation of our application programs when we moved to NX4800 and NX5800 a few years ago. Level Delta is used on all of the machines currently in production, including the NX6800, the LX and CS series, and the new Libra models.



Why are these various levels of E-mode important? The answer is that because of changes to the control words and other architectural differences, dumps from machines running the various levels will look different. The differences between Beta machines and the others are very noticeable, while the differences between Gamma and Delta machines are much more subtle.

It turns out that the way that programdumps are formatted diminishes most of these differences, so it's not all that hard to switch from reading Beta dumps to the ones for the later versions. This issue is becoming less important over time, as the older machines are retired and the older versions of E-mode fall out of use.

For the remainder of this discussion, we will focus on E-mode level Delta, since that is the only one which is used with any of the currently supported machines.

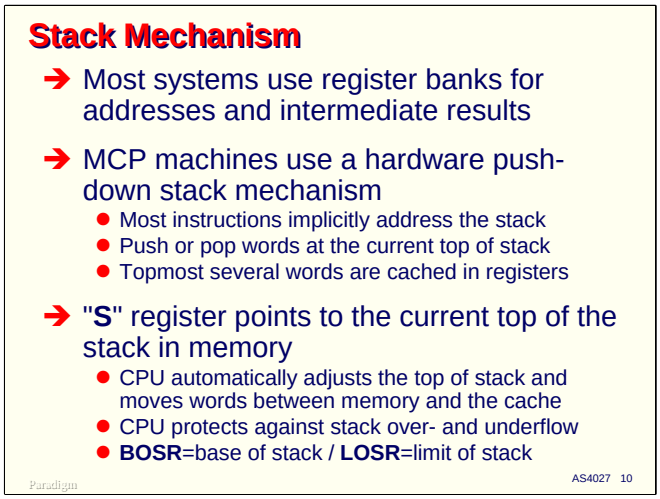


Most of our discussion on programdumps is going to be centered around something called *the stack*. The reason for this is that programdumps are nothing more than a picture of a program's stack, and optionally some related stacks.

Stacks are probably the most important data structure in an MCP system, so much so that the MCP machines are often referred to as "stack-oriented," although it would be more correct to call them "descriptor-oriented" or "capability-oriented" machines.

In its most basic sense, a stack is nothing more than a contiguous, bounded area of memory. Both the hardware and MCP software make use of this area in special ways, however, and the way that the stack works is actually the basis for most of the machine's characteristics and behavior, including:

- The way that arithmetic and other expressions are evaluated.
- The way that memory is addressed by the hardware.
- The way that procedure calls work, how the history (return linkages) for procedure calls are maintained, and the ability of the machine to do efficient recursion of procedures.
- The way that virtual memory works on the system, which is unlike any other on a commercially available platform.
- The way that multiprogramming, multiprocessing, and task management operate, and to a large degree, the very rich facilities we have for application-level multi-tasking.
- The way the library programs work and the reason they are so efficient.
- The architecture of the MCP itself.



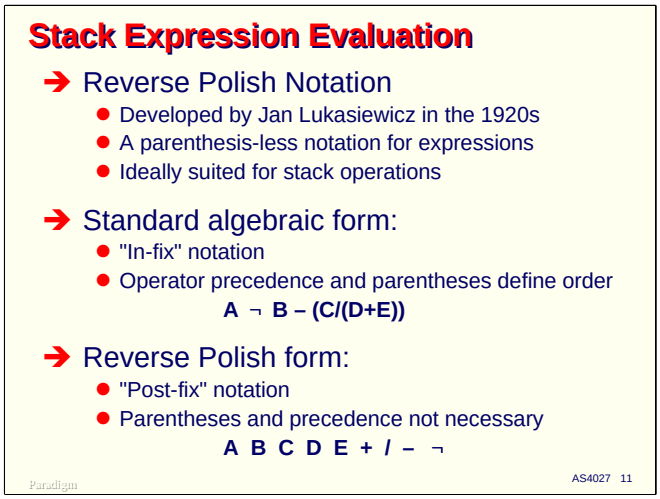
Most computer systems use register banks to manipulate addresses and store intermediate results during the evaluation of expressions. Many of these machines support stack operations, but these typically require that the instructions be coded a certain way, and that the user-level software actually do the stack manipulation itself.

MCP machines, on the other hand, are intrinsically stack oriented. The hardware supports a push-down, pop-up stack mechanism, and this mechanism automatically comes into play as instructions are executed. Most instructions do not explicitly reference the stack, and many instructions do not have addresses or other operands at all – they automatically use the words on the top of the stack.

Many instructions push words onto the stack or pop them off. The location in memory where these words are pushed or popped is called the *top of stack*. The location of the top moves as words are pushed onto the stack or popped off. Each processor has an address register called the S (for stack) register. This is the register that keeps track of the current location in memory for the top of stack. The processor automatically both stores or reads words from that location, and increments or decrements the S register to adjust the top of stack location, as required.

To improve efficiency and reduce traffic between the processors and memory, some number of words at the top of the stack are cached by the processor in internal registers. As these caching registers fill up or empty out, the processor automatically moves words between the registers and memory at the the top of stack location in memory.

Remember that the stack is a *bounded* are of memory. It has a finite length, and the processor also keeps track of where it is in the stack with respect to these bounds. There are two registers in the processor which do this, BOSR (pronounced BOW-zer), the Base of Stack Register and LOSR (pronounced LOW-zer), the Limit of Stack Register. If the S register exceeds the limits of these registers, your program will get a Stack Overflow or Stack Underflow interrupt and will be terminated. Stack overflows are not uncommon, but stack underflows are very rare, and usually indicate some serious hardware or MCP software problem.



The first part of stack operations to cover, and the easiest to understand, is how it supports the evaluation of expressions.

MCP systems use a technique called Reverse Polish Notation. This is a way of rewriting a traditional algebraic expression so that is requires neither parentheses nor operator precedence to evaluate it unambiguously. This form of expression is ideally suited to performing computations using a stack. It was developed by the Polish mathematician Jan Lukasiewicz in the 1920s.

If you take a simple arithmetic assignment statement and write it in the traditional algebraic form, you might have something like this:

A ¬ B – (C/(D+E))

where the arrow indicates the variable on the left-hand side "is assigned to" or "takes the value of" the expression on the right. This is called an *in-fix* notation because the operators are between the operands. In this case, the parentheses completely determine the order of operations, but in most languages, there is an implicit precedence among the operators that will determine a default ordering.

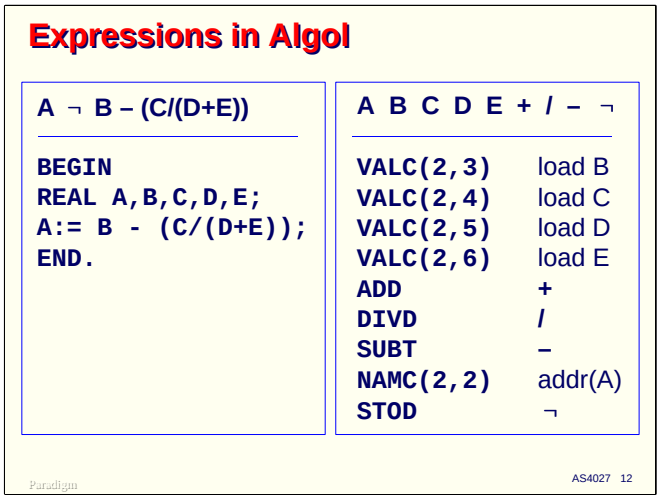
If we rewrite this in reverse Polish form, we get the following:

A B C D E + / – ¬

which is called a *post-fix* notation, because the operators follow the operands. (Actually, Lukasiewicz developed a "pre-fix" notation where the operators came first, which is called simply Polish notation – the idea of turning the expression around came later, hence the name *reverse* Polish.)

In this example, all of the operators came after all of the operands, but typically they will be intermixed. The way it works is that you scan along the list of operands until you find an operator. You then apply the operator to the two operands immediately to the operator's left, replacing all three tokens by the result. You continue scanning the expression, applying operators to the operands on its immediate left (some of which could be the result of earlier operators) until you reach the end. Consider the following algebraic expression and its reverse Polish counterpart:

A ¬ (C+D)/(E-F) – B **A C D + E F – / B – ¬**



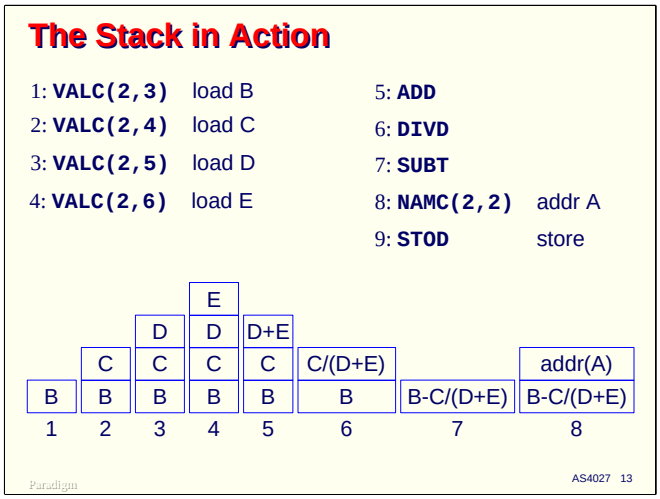
It is instructive to see how the principle of Polish notation translates to the way that MCP systems actually execute code. In the left-hand panel of the slide, the original algebraic expression has been translated into a minimal Algol program.

The right-hand panel shows the actual machine instructions which the MCP 7.0 Algol compiler generated when this program was compiled. Notice the one-to-one correspondence between the Polish notation and the instructions the compiler produced.

- The VALC (value call) is a load instruction – it reads a word from a memory location and pushes the value onto the stack. The numbers in parentheses are termed an *address couple*, which we will discuss shortly. There are four of these corresponding to the four operands from the right-hand side of the algebraic expression.
- The ADD instruction pops the top two words from the stack, adds their values, and pushes the resulting sum back on to the stack.
- The DIVD instruction then pops the top two words from the stack (one of which was just left there by the ADD instruction), generates the floating point quotient, and pushes the result back onto the stack.
- In a similar way, the SUBT instruction consumes the top two words of the stack and pushes its result back onto the stack.
- The NAMC (name call) instruction pushes the address of variable A onto the stack.
- The STOD (store destructive) instruction pops the top two words from the stack, one of which must be an address and the other a value. It stores the value at the indicated address. This instruction does not push anything back onto the stack, so at the end of evaluating the expression, the stack is at the same point it was when we began. The store is called "destructive" since it pops both of its operands from the stack. There is also a non-destructive store instruction (STON) which pops the address, but leaves the value on the stack. Non-destructive stores are used in expressions with embedded assignments, e.g., in Algol,

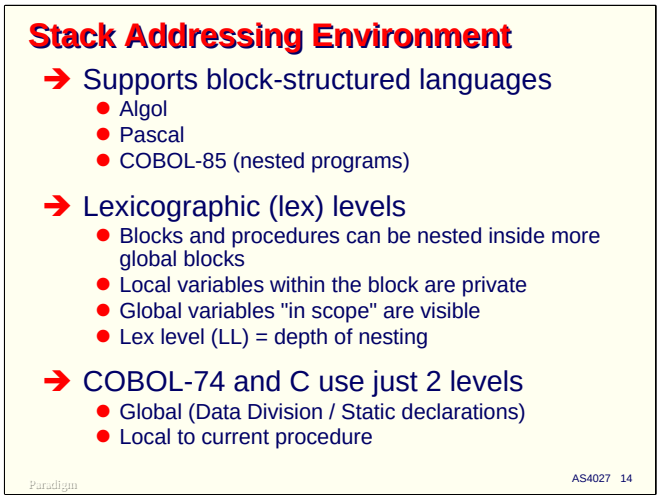
```
A:= B + (C:= D/E);
```

Note that in the reverse Polish expression the reference to A came first, but in the Algol code it came near the end. This is the result of a small optimization the compiler performs to minimize the amount of stack push-down that is required to complete execution of the expression.



We can "play computer" and watch what happens to the stack as each of the instructions on the prior slide is executed. On this slide, each of the instructions is numbered, and below them, are corresponding pictures of the stack after each instruction executes.

There was not room to show the result of step 9, the STOD instruction. After it executes, both entries that were present after step 8 have been popped, so the stack is back to the point it was before step 1.



The next facet of the stack to explore is how it is involved with addressing memory in the system. This is a complex subject, and there is actually much more to it than the brief introduction presented here. The scope of what follows is intended to be just enough so that you can make sense of a programdump.

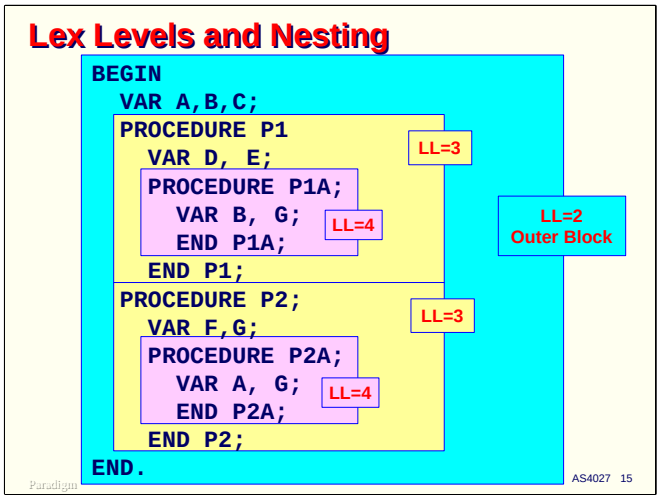
One of the hallmarks of the MCP architecture, dating back to the original Burroughs B5000 in 1962, is that it supports block structured languages such as Algol and Pascal. COBOL-85 is also considered a block-structured language, since you can have nested programs which act very much like the procedure blocks of Algol and Pascal.

The primary characteristic of block structured languages is that they are composed of, well, blocks. A block in both Algol and Pascal is defined by a BEGIN/END statement pair which contains at least one declaration. Blocks can stand by themselves, but they are most often used as the body of a procedure.

Blocks can be nested inside each other. The variables declared within a block are private to that block, and cannot be "seen" by more global blocks or those outside its nesting. Blocks nested within this one, however, can see that block's variables as globals. Therefore, the instructions in a block can see that block's variables, plus all of the variables in the more global blocks within which it is nested. The set of variables (and the set of blocks) which can be seen at any one time is called the *scope* of the block.

The depth of a block's nesting is called its *lexicographical level*, or more commonly, its "lex level." Lex level is abbreviated "LL."

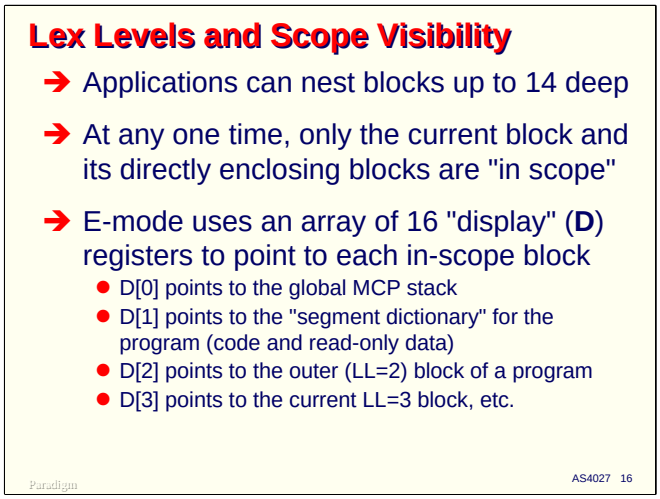
COBOL-74 and C programs normally use just two lex levels, one for the global data (Data Division in COBOL and global/static data in C), and one for the currently-executing procedure (which for COBOL-74 is the entire Procedure Division).



This slide shows an Algol- or Pascal-like program with several nested procedures. For reasons we will discuss shortly, the "outer block" (most global block) of a program is usually assigned lex level 2. Blocks nested within that are assigned progressively higher lex levels.

If you look at procedure P1A, it has as local variables B and G. It is nested within procedure P1 and the outer block, so these two blocks are within its scope. Code in P1A can see its local B and G variables, along with D and E from procedure P1, and A and C from the outer block. P1A cannot see variable B from the outer block, since its local B hides the more global B. P1A can also see procedure P2 (i.e., it could call P2), but it cannot see any of the declarations inside P2, including procedure P2A.

The way that block nesting, lex levels, and scope visibility work is very important to how variables are addressed in any computer system which implements block-structured languages. The MCP systems have an elegant mechanism for doing this, as shown on the next slide.



Programs for E-mode machines can nest their blocks up to 14 levels deep. As discussed on the prior slides, only the variables and procedures contained within the currently-executing procedure and its more global blocks are "in scope" at that point in time.

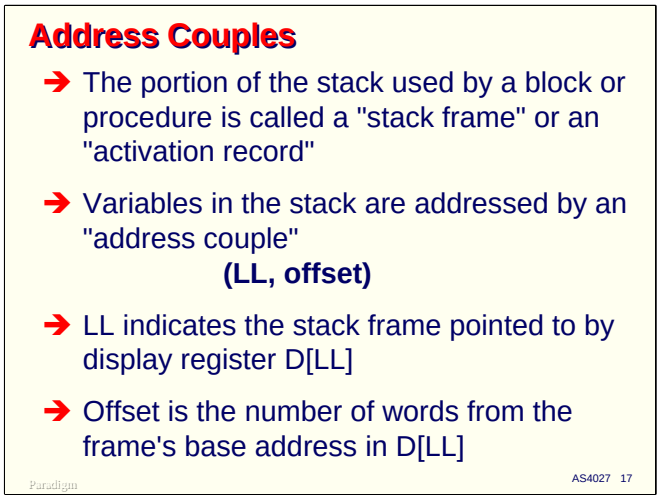
E-mode uses an array of 16 address registers, called "display" or "**D**" registers, to point to the parts of the stack for those blocks which are currently in scope. This array of registers is indexed by the lex level, so D[2] points to the outer block of a program, D[3] points to the next inner block that is active (if there is one, D[4] point to the block nested inside that, and so forth. It is possible for the outer block of a program to run at a higher lex level, and most languages provide compile-time options to control this, but it is not normally done. It is common , however, to compile object code modules for higher lex levels if you are doing binding.

Note that not all D registers are valid at any one point. The processor has another register, called LL, which stores the lex level of the procedure or block which is currently executing. Only D registers for that level and below are valid at that point in time. The addresses in the D registers for any higher lex levels are meaningless.

The D registers are numbered beginning with zero. D[0] points to the global stack for the MCP. D[1] points to a program's *segment dictionary*, a separate stack which is an addressing environment for the object code and read-only data associated with the program. Using a separate stack for the code and read-only data allows this data to be used reentrantly (i.e., by more than one copy of a program at a time), as we will see later in this discussion.

Since there are 16 D registers and the first two are reserved for the MCP and segment dictionary, that leaves 14 for user-level programs, hence the restriction on nesting blocks 14 levels deep.

The original B6700 and B7700 systems had 32 D registers, but experience with these machines showed that this was, for almost all practical applications, way more than was needed, so the number was reduced to 16 in the E-mode machines.



Many kinds of variables in programs are allocated directly in the stack. The portion of a stack that is used for a procedure's or block's variables is called its *stack frame* or *activation record*. When this stack frame is in the current scope, one of the D registers will point to its starting address in memory.

Variables for a procedure or block are addressed relative to the beginning of the stack frame using a concept called an *address couple*. This is written as two numbers in parentheses, thus:

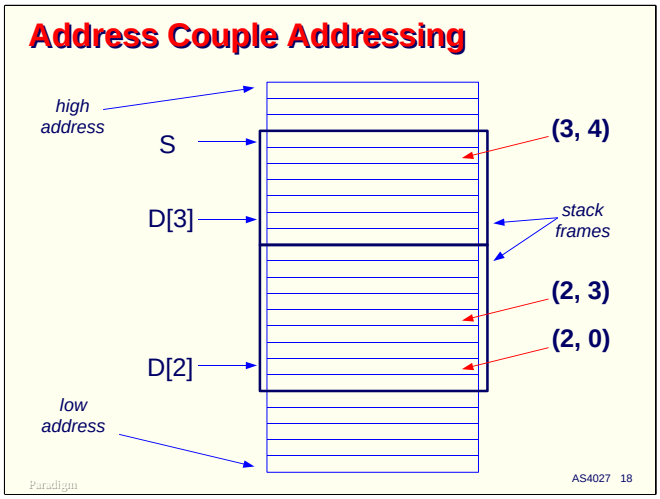
(*LL*, *offset*)

where *LL* is the lex level associated with the stack frame, and *offset* is the number of words (zero relative) from the start of the frame to the variable being referenced.

When the processor encounters an address couple, what actually happens is that it takes the address contained in the register D[LL] and adds the offset to it. This produces an absolute memory address, which is then used to reference physical memory. Therefore, instructions do not address memory directly, only indirectly through a base address for a stack frame in one of the D registers. This allows the stack frame to exist anywhere in memory (and move to somewhere else in memory) without having to change the object code at all. It is also one of the reasons why a 16-bit instruction like NAMC or VALC can access variables anywhere in the multi-gigabyte address space of a ClearPath MCP system.

There are a couple of ways that address couples are encoded as bit fields in instructions, but that is not a detail which need concern us for the purpose of basic dump reading skills. If you will remember from the example code shown for the reverse Polish expression evaluation earlier, the NAMC (name call) and VALC (value call) instructions had address couples associated with them. Therefore, these instructions are capable of addressing variables which are in the current scope of the procedure or block which is executing them.

The next slide shows a picture illustrating stack frames, D registers, and address couples.

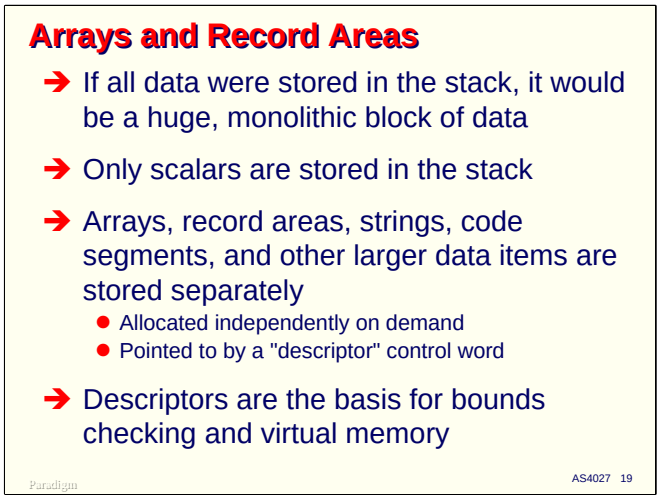


We will look at a number of schematic representations of a stack (or a portion of one) like the diagram above during this presentation. Stacks for MCP systems "grow" from low addresses to higher addresses in memory. This is the opposite of many other processor architectures which use stacks. Most others, including the Intel architectures, start their stacks at high addresses and grow towards lower ones.

The heavy rectangles show the areas of the stack occupied by two stack frames, one for LL=2 and another for LL=3. There could be many other such frames on this stack, and some of them could also be for LL 2 and 3, but since the D registers can only point to one frame per level at a time, these other frames would be *out of scope*. As we will see a little later, there is another way to address words in a stack if the frame is not in the current scope of the program.

The D registers point to the start of the stack frame. Actually, on Gamma and Delta level E-mode machines, the D registers point to the second word of the frame. Beta level systems pointed to the first word. There are a couple of hardware control words at the beginning of a frame, which we will discuss when we talk about procedure calls. In order to increase both the size and the number of stacks the system would support, some fields of these control words had to be expanded for the Gamma level architecture, but there was not enough room in these words to do so, so a third control word was added. By adding it below the first original control word, address couples did not have to change, which helped keep existing object code files compatible with the new architecture.

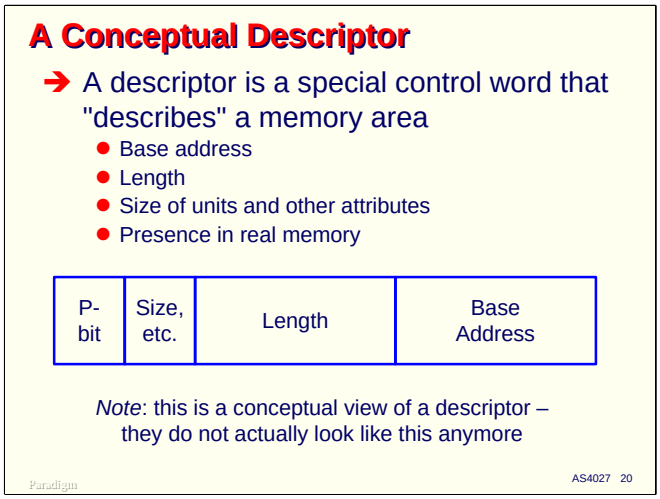
As you can see from the slide, address couple (2,0) references the same word that D[2] points to, while (2,3) references the word three higher in the stack. Similarly, address couple (3,4) references the word at offset 4 from the location pointed to by D[3].



If all of the data for a stack frame were stored in the stack, including arrays and record areas, stack frames would become quite large, and stacks themselves would tend to be huge, monolithic memory areas. This would make them difficult to allocate dynamically, and would make some features of Algol, such as array resizing, all but impossible.

On MCP systems, typically only variables which occupy one word (or in the case of double-precision variables, two adjacent words) are stored in the stack. Larger data structures, such as Algol arrays, and COBOL 01-level records, are stored elsewhere in memory. There is a special type of control word called a *descriptor* (specifically, a data descriptor) which is stored in the stack for the data structure, and which effectively points to the actual location of the data.

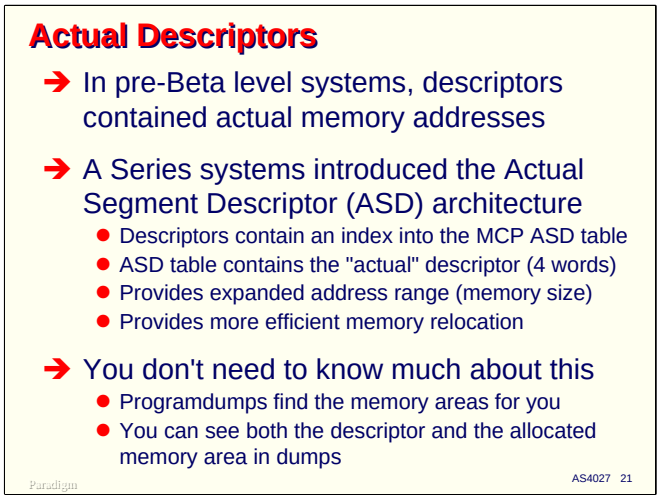
Descriptors are one of the most powerful and elegant aspects of the MCP architecture. Among other things, they allow memory for data structures to be allocated on demand, control the checking of area boundaries, and are the basis for virtual memory on the system. In order to understand programdumps, you have to know something about how descriptors work.



Conceptually, a descriptor "describes" a separately-allocated-memory area. This description includes

- The starting, or base, address of the area in memory.
- The length of the area.
- The size of the units for the area – single-precision words, double-precision words, 8-bit bytes, or 4-bit digits.
- Whether the area is writeable or read-only.
- Whether the area is actually present in memory. If it is not present, it could be because it has not been referenced yet, or because it has been rolled out due to virtual memory activity.

Descriptors on pre-Beta machines looked very much like the diagram on the slide. Implementation of the ASD architecture changed dramatically how descriptors physically work, and level Gamma E-mode changed this even further. Nonetheless, this conceptual picture of a descriptor is effectively how they work, and is essentially how programdumps represent them to you.

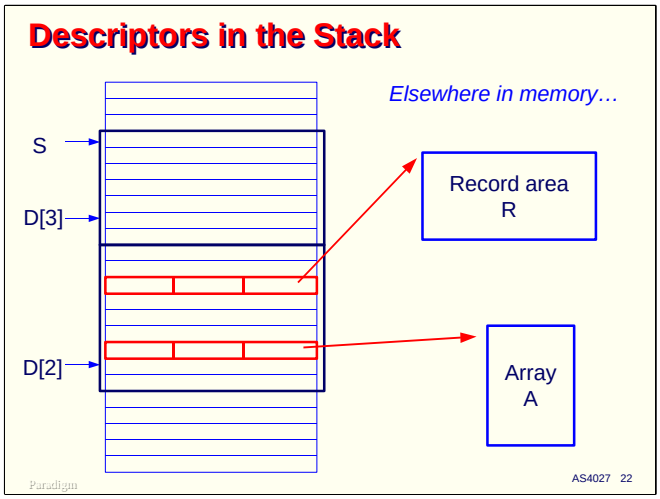


The way that descriptors work in E-mode machines is more complex than the conceptual picture presented on the prior slide, but there is a good reason for that. In pre-Beta machines, descriptors actually stored the physical address of the memory area. The field which held the address was 20 bits long, which allowed just over one million words, or 6 MB, to be addressed. In the mid 1960s, when the original B6500 was being designed, this seemed like more memory than anyone would ever need, but by the late 1970s, had proved to be far too little for large enterprise systems running data base systems and heavy on-line loads.

There were not enough bits in a descriptor word to make the address field larger, so the Burroughs engineers turned the original descriptor into an indirect pointer and stored the actual descriptors in a table controlled by the MCP. Instead of pointing directly to an area of memory, the address field in a descriptor now held an index into this table, called the Actual Segment Descriptor, or ASD table. Descriptors in this table are much larger – four words in length instead of one.

While there is some additional overhead to indirectly address a memory area through the ASD table, parts of the table can be cached in high-speed registers, and the use of a central repository for descriptors reduced some serious overhead problems in other areas of the virtual memory management scheme, particularly stack search. The net result was not only a significantly larger address space, but better overall system performance as well.

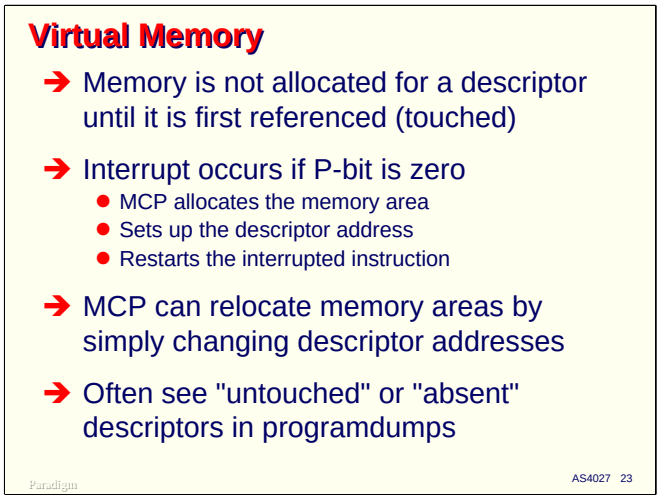
You don't actually need to know much about the ASD table or the format of descriptors in order to read a dump. Programdumps can show both the descriptor and the contents of the memory area it points to.



This slide shows how descriptors work in the stack. Each separately-allocated memory area has a descriptor, which is stored within the stack frame for the block or procedure in which it is declared. The descriptor points to the separate area, which is outside the area of memory allocated for the stack.

The MCP can move these areas around in memory to make room for other areas, or roll them out to disk if there is not enough physical memory to meet the needs of the currently running mix.

Descriptors are addressed using address couples, the same way that other variables are. A little later we will see how the contents of a separate memory area is accessed using the descriptor.



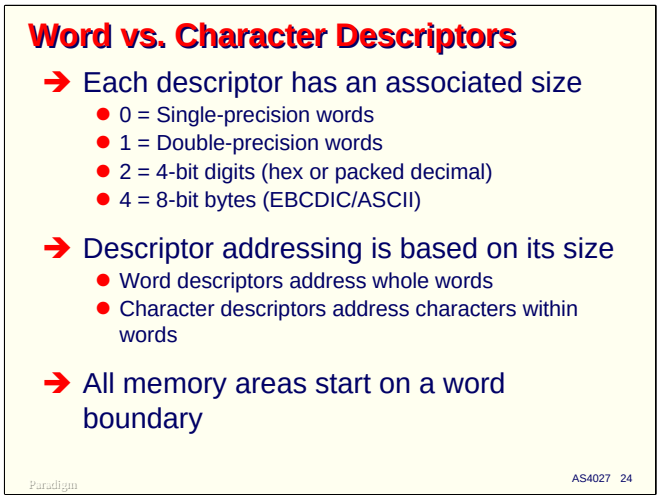
Using control words to keep track of data structures in memory has a number of wonderful advantages. The first is that memory for the data structure does not need to be allocated until the program actually needs it. If a particular data structure is never referenced during the execution of a program, the memory for its area will never be allocated, and only the space for the descriptor itself will be required. The process of first referencing a memory area and having it allocated is called *touching* the area or touching the descriptor.

Descriptors conceptually contain a bit which indicates whether the area is actually present in memory, called the *presence bit* or "P-bit." If a program attempts to address memory through a descriptor whose P-bit is off, a hardware interrupt occurs. Historically, we have referred to this as a "P-bit interrupt." On other systems this would typically be called a *page fault*.

When a P-bit interrupt occurs, the MCP allocates an area of memory of the appropriate size, and either initializes it to binary zeroes (if the area has not previously been allocated) or reads in from disk the contents of the area (if it had been previously rolled out due to virtual memory activity). It then fixes up the descriptor to point to the new area and restarts the instruction which triggered the interrupt.

Since the address of the memory area is available only through the descriptor, the MCP can relocate the area to another place in memory simply by changing the address in the descriptor – the rest of the program neither knows nor cares where the area physically resides in memory.

A descriptor which as not been referenced before is said to be *untouched*. A descriptor whose memory area has been rolled out to disk is said to be *absent* (but it's the area that's absent, not the descriptor). You will often see untouched descriptors in programdumps. With the large memory capacity of most currently installed systems, it is much more unusual to see an absent descriptor.

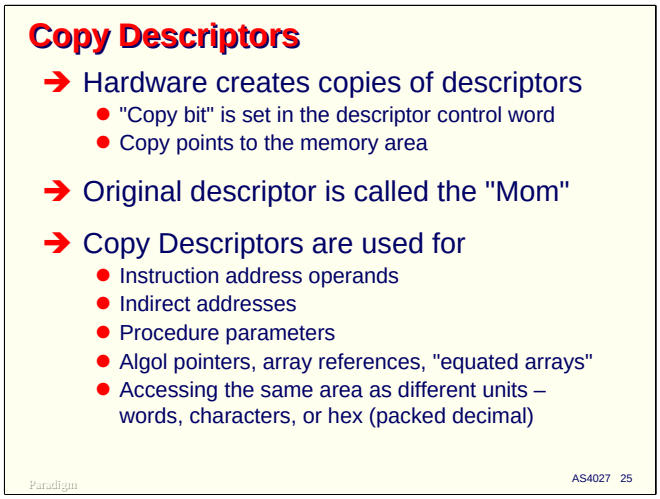


As mentioned earlier, descriptors have a unit size associated with them. This indicates whether the descriptor references a memory area on a word-oriented or a character-oriented basis. There are currently four unit sizes supported on the system:

- 0 indicates single-precision (48 bit) words.
- 1 indicates double-precision (dual 48 bit) words.
- 2 indicates four-bit characters (hexadecimal or packed decimal digits).
- 4 indicates eight-bit characters (typically EBCDIC, but also ASCII or one of the national character sets).

Character-oriented descriptors address whole words on word boundaries. Double-precision descriptors address pairs of adjacent words, but they are not required to start on even addresses. Character descriptors address characters within words. Physically, memory is word oriented, and all memory areas, regardless of their unit sizes, are allocated starting on a word boundary.

What about unit size 3? It used to exist. Those who were around in the 1970s will remember that the earlier B6000 and B7000 systems supported a six-bit character code called BCL (Burroughs Common Language), which provided compatibility with even earlier machines, such as the B5000, B5500, and B300. BCL support in the hardware was dropped more than 20 years ago, with the introduction of the Bx900 series systems.

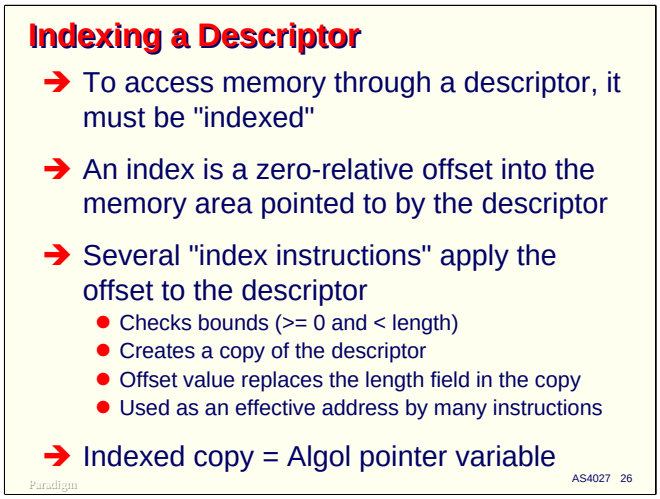


Another aspect of descriptors is that there can be copies. The original, or master, descriptor for a memory is called the *mom* (mother) descriptor. The hardware creates copies of descriptors in the process of determining actual memory addresses, and for some other reasons as well. There is a bit in the descriptor word, called not surprisingly, the *copy bit*. This is off for moms and on for copies.

Copy descriptors are used for a wide variety of purposes, including:

- Address operands for some instructions. Copy descriptors are one form of address which can be pushed onto the stack. We will talk more about these on the next slide.
- Indirect addresses. Copy descriptors effectively point to a memory area or one of the units contained within that area.
- Parameters to procedures. These are a form of indirect address.
- Algol pointers, array references, and "equated arrays." These are all represented by various forms of copy descriptors.
- Accessing an area originally allocated with one size of unit as another size. COBOL does this extensively when you have a memory area that contains a mixture of USAGE DISPLAY, USAGE COMP, or USAGE BINARY data items. The mom will describe the area in one unit size; copies of the mom will describe the area in other unit sizes.

Both moms and copies point directly to the memory area (actually, they point to the same ASD table entry, which in turn points to the memory area). Copies do not point to moms for allocated memory areas. Making a copy of an untouched mom causes the mom to be touched (on pre-Beta machines, making a copy of an untouched mom cause the copy to point to the mom without touching it – ugh).



Thus far, we have primarily seen descriptors used to address whole memory areas. Eventually, however, a program is going to need to address specific parts of the data structure contained within the memory area – words, characters, or perhaps strings of words or characters. Addressing into an area of memory is done through an operation called *indexing*.

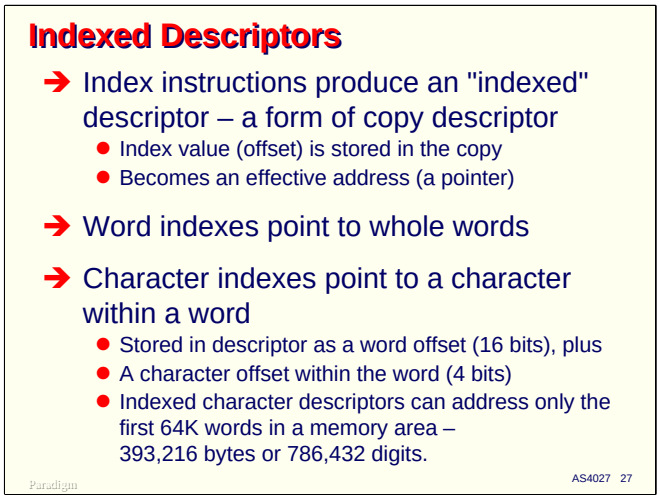
Indexing a descriptor involves applying an offset to its base address. The offset is always zero relative. For languages which use one-relative subscripts, or which allow the programmer to specify a lower bound, the compiler is responsible for generating code that will adjust the index to a zero-relative base. It typically does that by subtracting the lower bound from the index value supplied by the programmer. This is why arrays in Algol with a constant lower bound of zero are more efficient – the index is already zero relative, so nothing needs to be subtracted from it.

There are a number of operators in the E-mode instruction set which perform indexing operations. These instructions:

- Check the value of the zero-relative index against the bounds of the area. The index must be non-negative and less than the length of the memory area. This is the source of the ubiquitous "invalid index" interrupt that certainly every MCP programmer has had the pleasure of encountering.
- Create a copy of the descriptor. In the copy, they then replace the length field by the value of the index. This makes the *indexed copy* somewhat like an address couple – the descriptor effectively contains both the base address of the memory and the offset into it to the word or character being referenced. Many indexing instructions finish by leaving this indexed copy descriptor on the stack. Other indexing instructions (e.g., NXLV – index and load value, and NXLN – index and load name) virtually create this copy, but then use it to fetch the word it points to and push a copy of that word onto the stack.

The indexed copy descriptor is an effective address to one element of the memory area pointed to by the original descriptor. This effective address can be stored as a variable in the stack for later user. Algol pointers are nothing more than indexed copy descriptors.

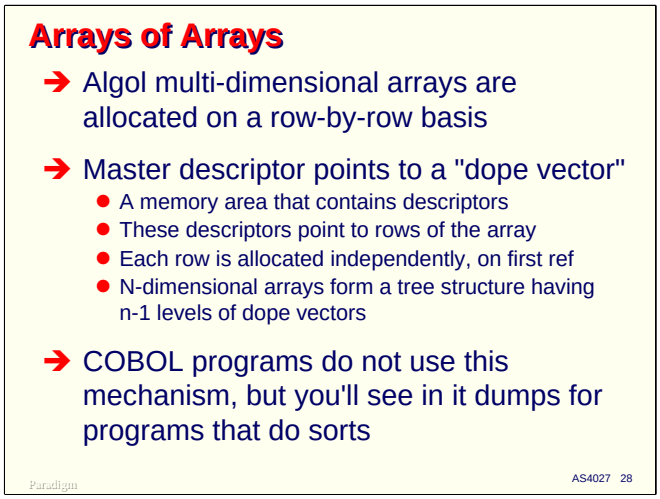
Some indexed copy descriptors have the ability to be *re-indexed*, i.e., to have another index value applied relative to the location their index value points to. This was not possible on earlier machines, since the index value replaces the length value in the descriptor, but with the ASD scheme, the length is retained in the actual descriptor within the MCP's ASD table, so the processor can still check that the re-indexing operation stays within the bounds of the allocated area of memory.



As discussed on the prior slide, an indexed copy descriptor effectively addresses one unit of its memory area. Indexed word descriptors point to a word on a word boundary. Indexed character descriptors point to an 8- or 4-bit character within a word.

The index field in a descriptor is the same size as the length field – 20 bits. For word descriptors, the index value is stored as a 20-bit word offset within the memory area, which means that programs can address within areas which are just over one million words long.

For character descriptors, however, the index value is stored in two parts – a 16-bit offset to a word within the memory area, and a 4-bit character offset within that word. This means that indexed character descriptors cannot address into areas larger than 2^{16} = 64K words (393,216 8-bit bytes or 786,432 4-bit digits).



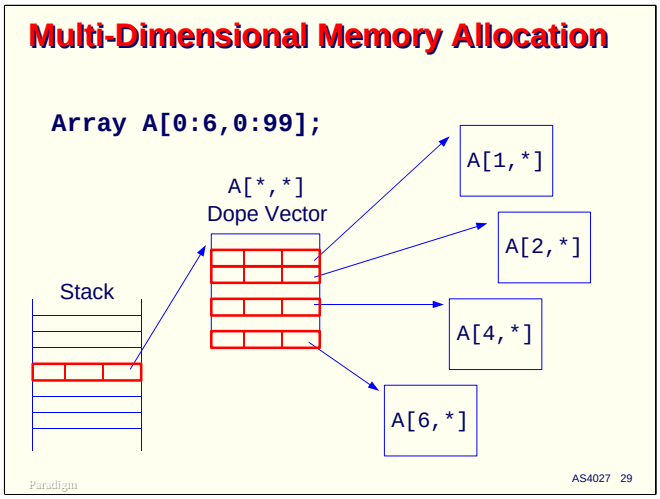
Up to this point in the discussion of data descriptors and separately-allocated memory areas, we have talked about single-dimension arrays, sometimes called a memory *vector*. Most programming languages support the concept of multiple-dimensioned structures, which require an index value to select an element from each dimension.

In some languages, such as COBOL, the multiple-dimensioned structure is simply mapped onto a contiguous area of memory. For example, a two-dimension table consists of rows and columns. In COBOL, such a structure has the rows placed head-to-tail within a record area. The compiler generates instructions to compute an effective offset into the area from the row and column subscripts. There is an operator in the E-mode instruction set which helps with this computation, OCRX (occurs index).

Algol, however, treats multiple-dimensioned arrays much differently. Each row in the last dimension is allocated as a separate memory area. The mom descriptor points to an intermediate area of memory, called a *dope vector*, which contains the descriptors for the rows of the next lower dimension. For a two-dimensional array, the mom will point to a dope vector that is the size of the first dimension. The dope vector will contain a descriptor for each row in the second dimension. For a three-dimensional array, the mom will point to a dope vector. Each descriptor in that dope vector will (potentially) point to another dope vector, whose descriptors in turn will point to the third dimension's rows.

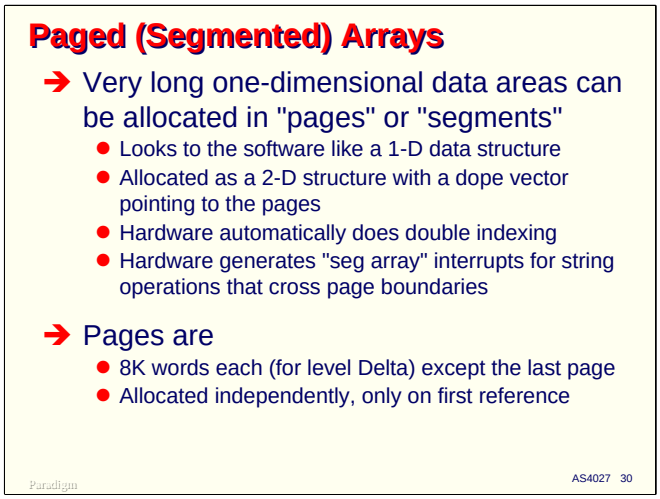
This approach creates a tree of dope vectors and "leaf" rows. An N-dimensional array will have N-1 levels of dope vectors. As with other separately-allocated data structures, only the rows which are referenced will be allocated, along with any dope vectors necessary to hold the descriptors that point to those rows.

There is no way to define data structures in COBOL which behave like this (except for paged arrays, which are discussed next), but it you do a sort in a COBOL program that calls input or output procedures, and you take a dump while the sort is in progress, you will usually see the stack frames for the MCP's sort procedure. These frames will contain descriptors for the multiple-dimensioned arrays used by the sort for buffers and work areas.



This slide shows a schematic representation of a two-dimensional array. The mom descriptor in the program's stack points to a dope vector. In this case the dope vector is seven words long (since the first dimension is declared with bounds 0:6).

As rows of the array are first referenced, areas of memory will be allocated for them, and corresponding descriptors will be placed in the respective elements of the dope vector to point to these areas. The MCP can relocate and overlay the individual row areas independently. In Algol, you can also resize the rows independently, so while the array was originally declared to have rows that are 100 words (0:99) each, you could end up with each row having a larger or smaller length. You can also resize the dope vector in Algol, to change the number of rows from its original value of seven to something else.



The final topic concerning descriptors and arrays is that of *paged arrays*, often referred to as *segmented arrays*. An unpaged array is referred to as a *long array*.

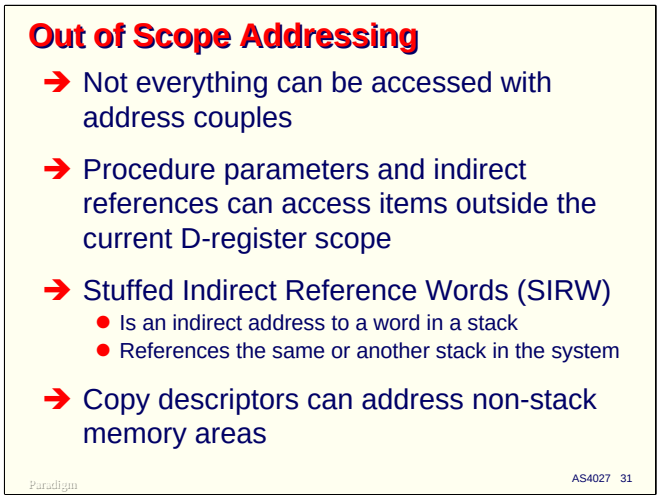
Single-dimension arrays (or the rows of the final dimension of a multiple-dimensioned array) which are very long can be difficult to allocate in memory, since the MCP must find a contiguous area of memory large enough to contain the entire row. Paged or segmented arrays allow the system to break up these very long memory areas into *pages* or *segments* which are shorter and more easily allocated.

Paged arrays are physically two-dimensional arrays which effectively look like a single-dimension array to the software. There is a bit in the descriptor that indicates whether the array is segmented. The software supplies a single index value to the indexing instructions, which break the value into two parts – a page number and an offset within the page. The MCP allocates a dope vector to hold descriptors for each of the pages. As with the two-dimensional structures, the individual pages are allocated only when first referenced. There is some overhead to the double indexing required for paged arrays, but in many situations this is offset by the lower impact of allocating smaller arrays, and the convenience of having a data structure whose physical memory requirements grow automatically only as new pages are first referenced.

Paged arrays are straightforward if all you are doing is fetching a single element from the array. String operations where multiple elements are accessed in sequence, however, are a problem when the operator reaches the end of a segment. In most cases, the E-mode processors are able to automatically jump to the next segment and continue the operation. Where this is not possible, a page boundary interrupt is generated and the MCP either finds a way to continue the operation, or it generates a fault.

Historically, having a string operation attempt to run off the end of an allocated area (in either direction) has been called a "segmented array error" or a "seg array error." It is nearly as common as "invalid index."

For E-mode level Delta, all rows of a paged array are 2^{13} = 8K words long, except the last page, which may be shorter. Array rows which are shorter than this are always allocated as unpaged, or long arrays. The default threshold at which the MCP starts allocating large arrays as paged ones is the page size, 8K words. You can modify this threshold with the SEGARRAYSTART ODT command.



Earlier we discussed how variables allocated in the stack can be directly accessed using address couples if they are in the scope of the currently-executing procedure. What about variables that are not in the current scope? Can they be addressed as well? The answer is yes, but not quite as efficiently.

The primary reason for addressing something out of scope is that it has been passed as a parameter to a procedure by name or reference. Algol supports Array Reference and Procedure Reference variables, which can access items which are outside the current scope. The MCP also creates out of scope references for certain types of entities, such as library entry points.

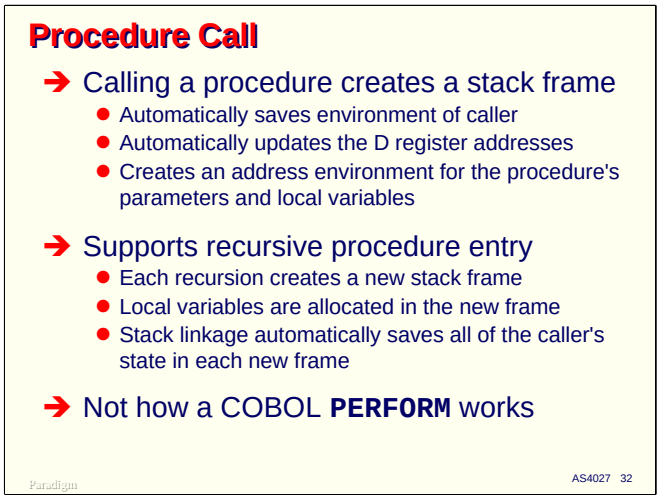
There are two ways such variables can be addressed in the E-mode programs (there is at least one more, but it is not available to user programs). The first is the delightfully named Stuffed Indirect Reference Word, or SIRW.

An SIRW is a control word which contains three main parts: the offset from the base of stack to the beginning of a stack frame, the offset from the base of that stack frame to the word being addressed, and the stack number (SNR, pronounced "sner") which contains the stack frame of interest. This means that an SIRW can not only address any word in your stack, it can potentially address any word in any other stack in the system (actually, its capability is even more general than that – a topic we will not address, so to speak, here). This address-anything-in-any-stack facility may sound insecure, but it's not – a user program can only generate SIRWs for variables which are at some point in scope. Any other SIRWs must be generated by the MCP for you, and it's mightily particular about what it allows you to have. You can think of an SIRW as simply an indirect address to something in a stack.

There used to be an "unstuffed" Indirect Reference Word (IRW) on the older machines, but it no longer exists. NAMC operators now always generate an SIRW. There is an instruction, STFF (called "stuff"), which formerly would cause an IRW to, literally, get stuffed,[1] but this instruction now acts as a no-op.

The second way that variables can be addressed out of scope is with an indexed or unindexed copy descriptor.

[1] I am not making this up.

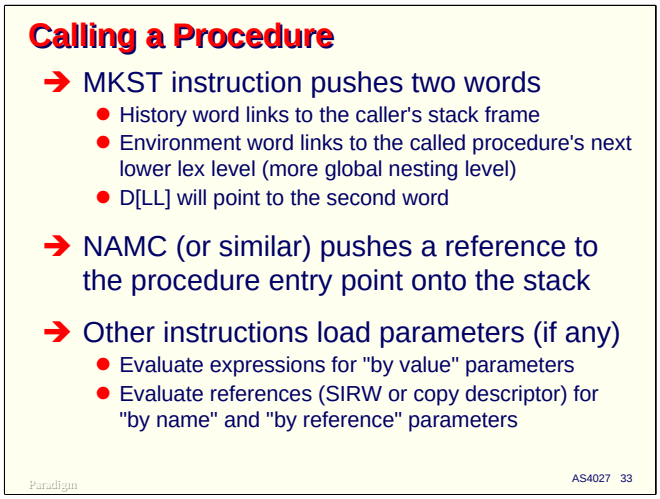


The next topic in our discussion of the E-mode architecture is that of procedure call.

When a procedure or subroutine (including typed procedures, or functions) is called, it creates a new stack frame. All of the caller's environment is automatically saved in the new stack frame by the hardware as part of the procedure call process. No special register saving or other state management is required by the software. The procedure call process also automatically reloads the D registers as necessary to point them to the set of stack frames that is in scope for the new procedure. Finally, the new stack frame creates the addressing environment for the procedure's parameters and local variables.

Creating a new stack frame for each procedure call automatically supports recursive procedure entry. Since all of the caller's state is saved in the new stack frame, and the frame contains the procedure's parameters and local variables, each recursion of a procedure gets its own set of variables allocated in the stack. The extent of recursion is limited only by the physical limitation on the size of a stack (64 K words for level Delta).

Note that the COBOL `PERFORM` does not use this procedure call mechanism. `PERFORM` does not need to be concerned with passing parameters and supporting local variables. It uses a much simpler mechanism that is managed by the COBOL program code itself. We will discuss this in Part II when we examine COBOL-74 and -85 programdumps.



From a software perspective, procedure call is a fairly straightforward process. From a hardware, perspective, however, it is extremely involved. The following steps occur:

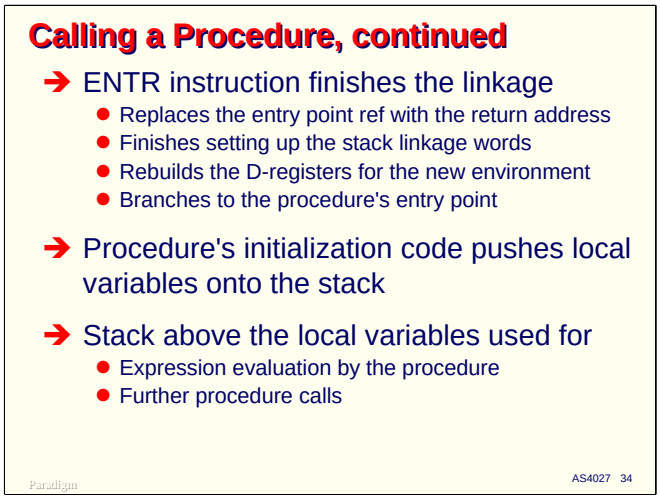
The software begins a procedure call by executing a MKST (mark stack) instruction, which pushes two control words into the stack. There are a couple of variants of this instruction, but they all end up accomplishing the same result.

- The first of the words pushed onto the stack is called the history control word, or HISW. It provides an offset within the stack to the base of the caller's stack frame. Thus the history control words effectively link the stack frames together.
- The second word is called the environment link word, or ENVW. MKST places a dummy value at this location; the real ENVW will be built later by the ENTR (enter) operator.

Next, the software uses a NAMC (name call) or equivalent operation to push a reference to the procedure's entry point on the stack. This reference must evaluate to a special control word called a Program Control Word, or PCW, which we will discuss later.

Following the reference to the procedure's entry point, the software must push any parameters onto the stack. For "call by value" parameters, a copy of the value of the parameter is pushed; for "call by name" or "call by reference" parameters, a reference to the actual parameter is pushed. This will normally be an SIRW, or a copy descriptor. If there are no parameters, then nothing is placed on the stack after the reference to the entry point.

The call to the procedure is now almost complete. The process is continued on the next slide.

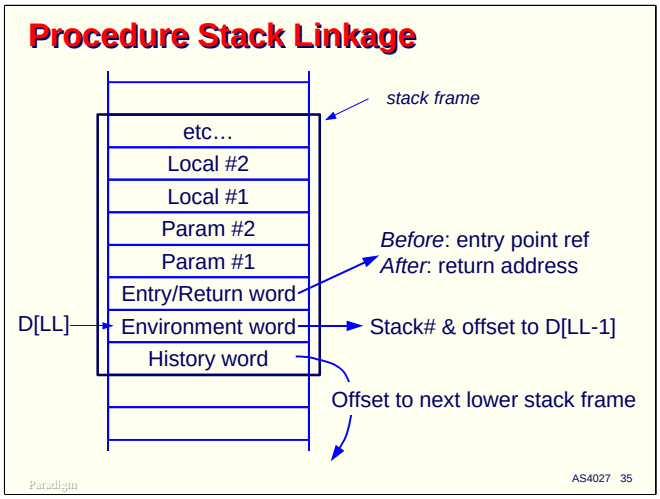


After pushing any parameters onto the stack, the software executes an ENTR (enter procedure) operator. This operator is one of the most complex in the instruction set. It does the following:

- The reference to the procedure's entry point is de-referenced to locate the PCW, which contains the code address of the procedure.
- The third word in the stack frame, which originally contained the reference to the PCW for the procedure's entry point, is replaced by a Return Control Word (RCW), which contains the code address after the ENTR instruction. This is the location where execution in the caller will be resumed after the procedure exits.
- The ENVW is completed and stored in the second word of the stack frame.
- The PCW contains the lex level at which the new procedure will run. This is used to update the LL and D registers as necessary to point to that procedure's scope.
- Finally, the processor branches to the code address of the procedure's entry point.

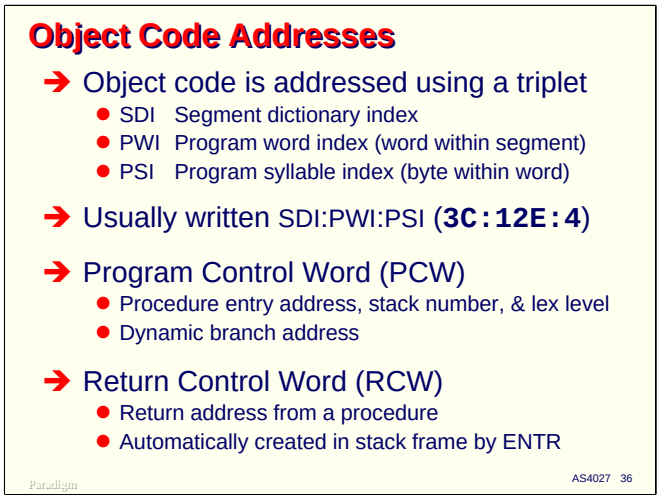
At this point, the procedure call itself is complete. The entry point of the procedure generally contains some initialization code that pushes words for the procedure's local variables onto the stack. Once that is complete, the body of the procedure executes. The stack above the local variables is used for expression evaluation, and for the stack frames of further procedure calls.

The complete process is illustrated on the next slide.



As described on the prior two slides, a procedure call proceeds as follows:

- A MKST instruction pushes the history control (HISW) and initial environment link (ENVW) words onto the stack.
- A NAMC or equivalent operator places a reference to the procedure's entry point (which must evaluate to a Program Control Word, or PCW) onto the stack.
- Additional instructions push parameters, if any, onto the stack.
- The ENTR instruction finishes creating the stack frame, builds the complete ENVW, updates the D registers, replaces the reference to the entry point PCW with a Return Control Word (RCW), and branches to the procedure.
- Initialization code in the procedure pushes words for any local variables onto the stack.
- Execution of the procedure continues, with stack space above the parameters and local variables used for expression evaluation and further procedure calls.



As part of the procedure call process, we referred to a procedure's "entry point address" or "object code address." Since object code is not stored in the stack, there is a different mechanism to address it.

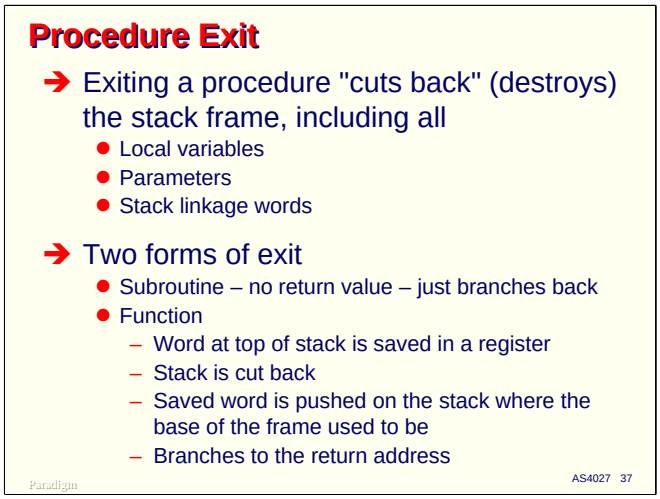
Instructions in object code are identified by a number triplet:

- The object code segment number, or Segment Dictionary Index (SDI).
- The word within that segment, or Program Word Index (PWI).
- The byte within that word, or Program Syllable Index (PSI).

These three numbers are typically written in hexadecimal with colons between the parts, SDI:PWI:PSI, e.g., `003C:012E:4`. You often see these triplets on compiler listings and in error messages generated by the system.

Object code addresses are contained in two related control words.

- The Program Control Word (PCW) is used to define the entry point address for procedures. It can also be used by the dynamic branch operators (e.g., DBUN) as a branch address.
- The Return Control Word (RCW) is built automatically by the ENTR operator as part of the procedure call process and stored in the stack frame. It has a layout very similar to the PCW, and describes the return address for a procedure call.



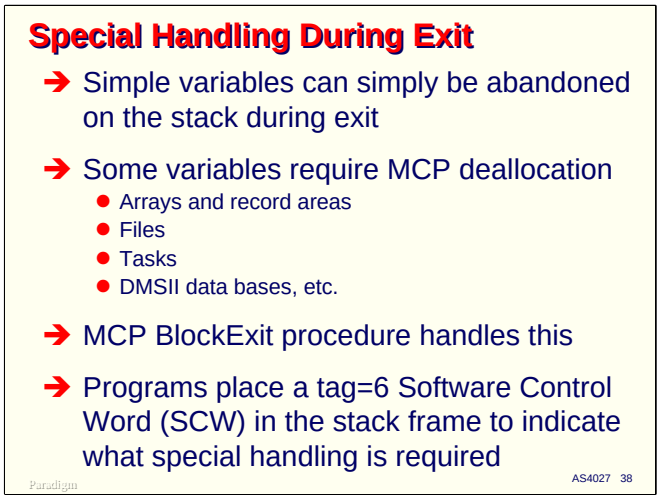
When a procedure exits, the hardware cuts back the stack to destroy the stack frame that had been used by the procedure. It knows how to do this because the link words at the base of the stack frame tell it where the prior stack frame starts in the stack, and have the information necessary to reload the D registers to reestablish the addressing scope of the caller's environment.

Cutting back the stack effectively deallocates everything in the stack frame – local variables, parameter words, and the three stack linkage words at the base of the frame. This is an efficient operation, since all the processor needs to do is change the settings in several registers. The words of the stack frame are not erased or cleared – they are simply abandoned on the stack to be overwritten by future push operations.

There are two forms of procedure exit. A subroutine exit simply cuts back the stack and branches to the return address in the RCW at the base of the stack frame.

A function exit (often called a *return*) is used when the procedure returns a value to the caller. It temporarily saves the value at the top of the exiting procedure's stack frame, cuts back the stack, then pushes the saved value back onto the stack before branching back to the caller. This leaves the returned value on the top of the stack for the caller to use in the evaluation of an expression. In this way, calling a function or typed procedure has a result just like pushing a value from memory onto the stack.

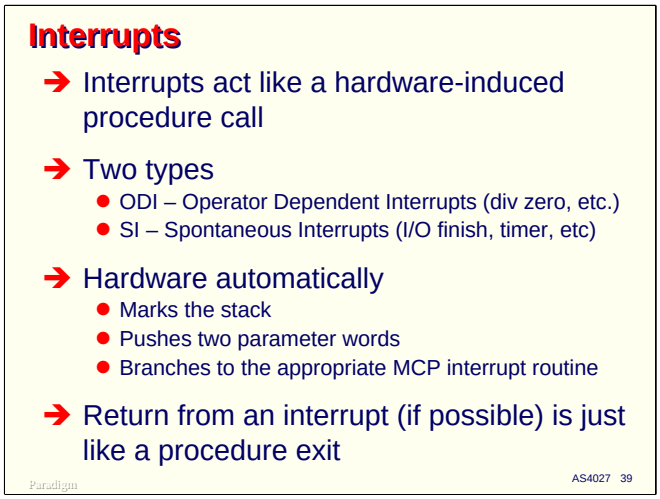
The distinction between calling a function and loading a value from memory onto the stack diminishes even further with a special type of procedure call termed an *accidental entry* or a *thunk*. This allows the caller to pass a function as a parameter when the called routine is expecting a call-by-value parameter or an indirect reference to a location in memory. Instead of an SIRW or indexed data descriptor to a data word, the caller passes an SIRW to a PCW which addresses the entry point of a procedure. When the called procedure references a parameter (typically through a VALC instruction), the hardware recognizes it is addressing a PCW instead of a data value, and automatically calls the function (which must have no parameters). The function leaves its result on the top of the stack, just as the VALC instruction would have done after loading it from memory.



As far as the hardware is concerned, everything in a stack frame can simply be abandoned on the stack when the procedure for that stack frame is exited. For scalar variables (integers, reals, doubles, booleans, etc.) and copy descriptors (array parameters and pointer variables) this is true.

Some local variables, however, require special handling by the MCP before their stack locations can be released. Local arrays declared by a procedure must be deallocated, local files must be closed and their File Information Block (FIB) deallocated, task variables must be handled specially, and so forth. These cases cannot be handled automatically by the hardware.

The MCP procedure BlockExit is designed to examine the contents of a stack frame and to gracefully dispose of the complex memory objects it finds there. The compilers place a call to this procedure at the end of the code for each procedure which could possibly allocate variables for such objects during its execution. The compilers also generate code to push a special control word onto the stack with information that will help BlockExit optimize this process. That word is called a Software Control Word (SCW). It has a tag of 6 and bit [47:1] set. You will often see SCWs in a dump near the top of stack frames.

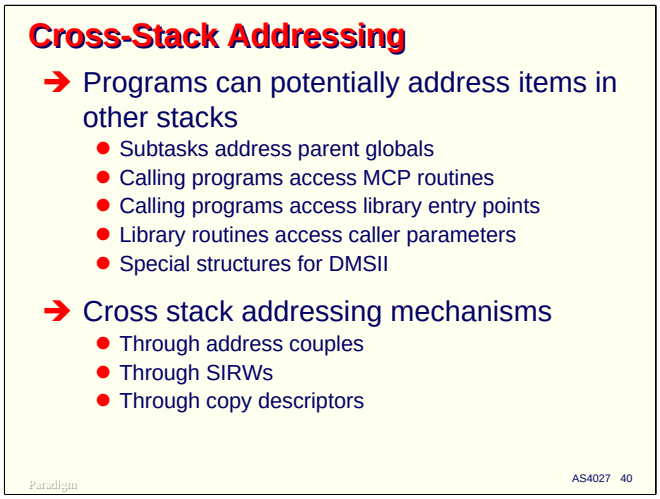


Interrupts on E-mode machines work in a novel way. They act like hardware-induced procedure call. There are two general types of interrupts:

- Operator Dependent Interrupts (ODI) are the result of executing some instruction. These include fault interrupts (divide by zero, invalid index, stack overflow, etc.), P-bit interrupts, and others which are directly or indirectly the result of an instruction.
- Spontaneous Interrupts (SI) are the result of some asynchronous process or unpredictable event occurring. Examples of these are I/O finish interrupts and timer interrupts.

In either case, when the interrupt is triggered, the hardware stops the current program's instruction flow, marks the stack, pushes two parameter words onto the stack, and branches to the appropriate interrupt handler procedure in the MCP.

In many cases, the program can be resumed after the interrupt is serviced. In such cases, the MCP simply exits from the interrupt handler procedure. This restarts the interrupted instruction, and the program continues executing.

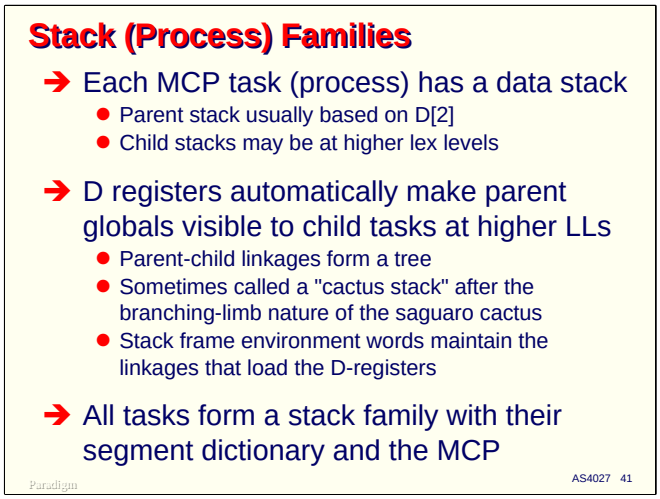


When we introduced the Stuffed Indirect Reference Word (SIRW), we saw that it contained a stack number in addition to the offset to a location within the stack. This is one of several ways that the system can allow a program running on one stack to address variables in other stacks on the system. There are a number of reasons why this might be done, including:

- Child subtasks can access variables in their parent stacks.
- Calling programs can access procedures in the MCP stack to perform services.
- Calling programs can access procedures in library stacks.
- Library routines (which run on the caller's stack) can access caller parameters and the globals in the library stack.
- DMSII makes extensive use of cross-stack addressing in the process of servicing data base requests.

There are three main mechanisms which can be used by user programs to address across stacks.

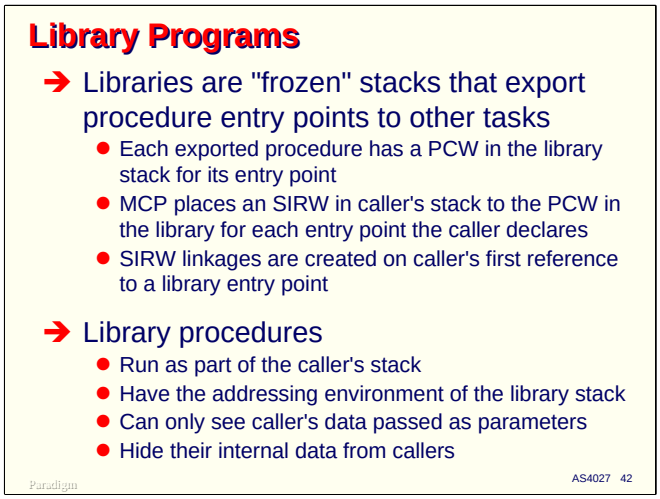
- Through address couples. If the running task has a parent whose global environment is within the running task's scope, then those lower lex levels will be directly visible to the running task, and it can access the globals through ordinary address couples.
- SIRWs provide an indirect address which can cross stack boundaries.
- Copy descriptors can be used to access one stack's memory areas from another stack, although the compilers and MCP go to great lengths to assure that copy descriptors are not allowed to exist after their memory area is deallocated.



In addition to addressing variables across stacks, stacks can be linked to form a tree-like structure called a *process family*. This tree-like structure is sometimes called a "cactus stack" after the manner in which the limbs of a saguaro cactus branch off from each other.

Each MCP task or process has a data stack associated with it. The stack for the parent task is usually based at LL=2. Child tasks may run at higher lex levels, depending on the programming language used and the way the child is initiated by the parent. The D registers in the processor automatically make parent global variables visible to child tasks running at higher lex levels. The environment words at the base of stack frames allow the hardware to automatically load the D registers when the MCP switches from one task (stack) to another, making all of the now-running stack's addressing scope visible to it.

All tasks form a stack family consisting of their data stack, their segment dictionary (D[1]) stack, and the MCP stack. We will see how this works when we discuss segment dictionaries shortly.

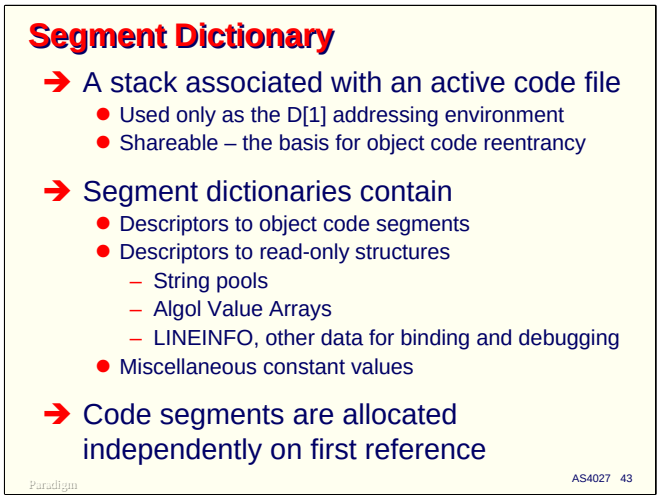


Library programs deserve special mention at this point, since they use the system's stack linking mechanism, but exist outside of the caller's stack family.

A library is a "frozen" stack. That is, once it initializes, it does not consume processor cycles – it exists in the system only as an addressing environment. Each library declares some of its procedures to be exported. This makes them visible to other programs in the system, and those programs can call the exported procedures of the library. Physically, this is accomplished by a mechanism called *library linkage*.

Calling programs access library entry points through special control words stored in the caller's stack. Before a program first references any of the exported library routines, the control words are set up to cause an interrupt. The MCP services the interrupt and replaces the initial control words with SIRWs which point to PCWs in the library's stack for the exported routines. Thereafter, the caller accesses the library's routines as an ordinary indirect reference through the SIRWs. Aside from the overhead to set up the linkage on the first reference, calls to library procedures are extremely efficient.

Library programs have an addressing relationship with calling programs only while one of the library's routines is being called. The library routine runs on the caller's stack (i.e., the stack frame for the called routine is built on the caller's stack), but the library routine's addressing scope is that of the library stack. The library can only see data from the caller which the caller passes as parameters. The caller cannot see the library globals, except for those entities which have been exported.

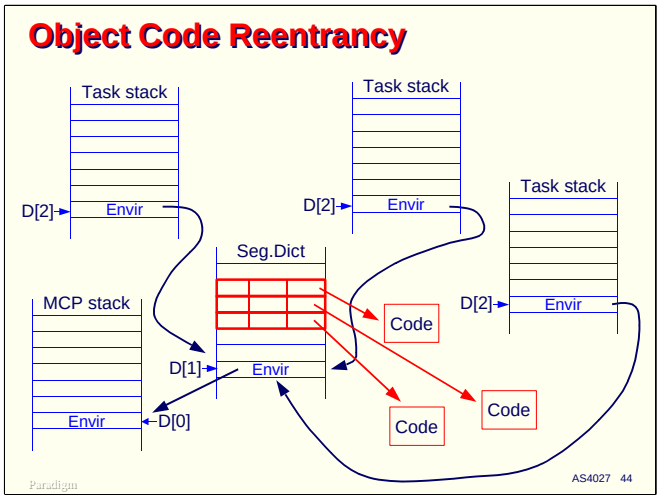


The final subject in our discussion of stacks is the *segment dictionary*. This is also called the *D[1] stack*, since it is always addressed at lex level 1. There is a segment dictionary associated with each active code file in the system. The dictionary is sharable among any number of data stacks, and forms the basis for object code reentrancy in MCP systems.

Segment dictionaries are another type of stack which exists only as an addressing environment. They are never used to execute instructions. They contain descriptors to the read-only objects for a program – object code, string pools, Algol value arrays, truthsets, translatetables, formats, etc. They also contain a descriptor for the LINEINFO tables that the compilers build. These tables cross reference object code addresses to source code line numbers. Some languages also place miscellaneous constant values in this stack. Since all of this data is read-only, there is no problem having multiple tasks share it simultaneously.

Segment dictionaries are built by compilers and stored in the object code file. The MCP automatically reads the segment dictionary from the code file when the first copy of a program begins executing, and sets up a stack for it. Since segment dictionaries for user programs are addressed at lex level 1, they are always in the global scope of their associated data stacks, which run at lex level 2 or higher.

The object code segments and other read-only data structures are allocated on first reference just like other arrays and record areas. If a particular code segment is not touched by a program during its execution, it will never be loaded into memory.



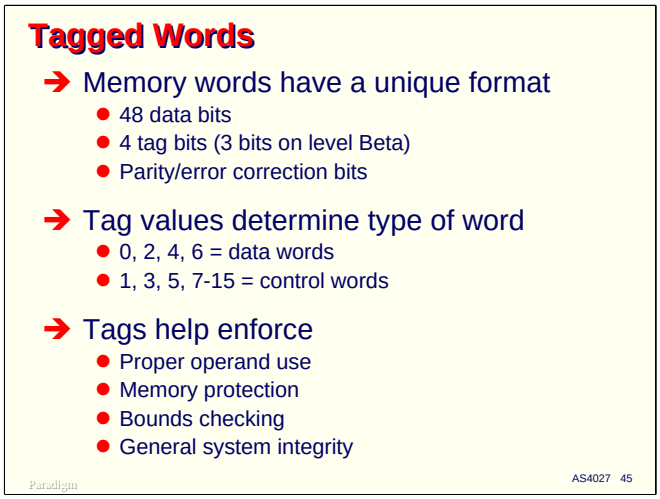
This diagram shows how segment dictionaries relate to their data stacks, and how they support reentrant use of object code.

When the first copy of a program, say `SYSTEM/DUMPALL` for example, enters the mix, the MCP sets up a memory area for its data stack, and a separate memory area for the segment dictionary. It then reads the segment dictionary information from the object code file, uses other information in the code file to set up the initial linkage in the data stack, and causes the D[2] register to point to the data stack and the D[1] register to point to the segment dictionary stack. It also sets up the stack linkage words to reflect this, and that the global environment for the segment dictionary is the MCP stack at lex level 0.

As the program executes, it touches code segments, which being absent, cause P-bit interrupts to occur. The MCP loads the corresponding code segments into memory from the code file as they are touched.

Now let us suppose that while the first copy of `DUMPALL` is running, someone else runs another copy of it. The MCP recognizes that the object code file is already in use (through a counter kept in the disk file header) and that the associated segment dictionary is already loaded into memory. It simply sets up a data stack for the second copy, and links it to the existing segment dictionary. The second copy then automatically references the same copy of the object code loaded into memory. The same thing occurs as a third and subsequent copies of the same program run simultaneously.

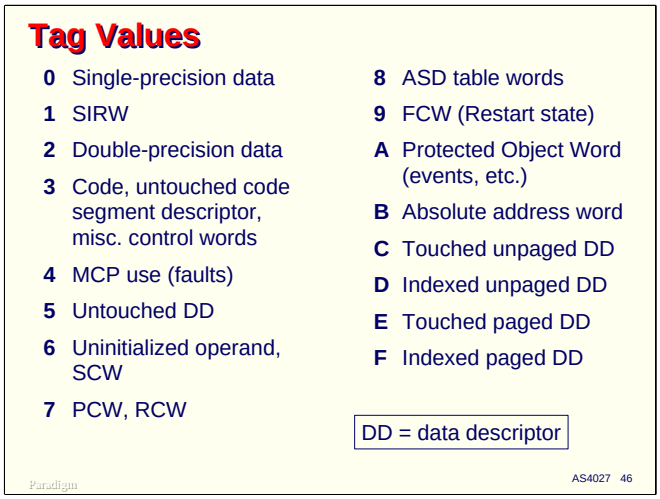
When the last copy of the program terminates, the MCP recognizes that there are no more data stacks using the segment dictionary, and it deallocates the D[1] stack, along with all of the read-only code and data segments that are present in memory at that point.



The final topic in our overview of the E-mode architecture concerns the format of data words. Words in memory have a unique structure on E-mode systems. They are composed of 48 data bits along with parity or error correction bits, but have four additional control bits (three on level Beta and earlier systems) called the *tag*.

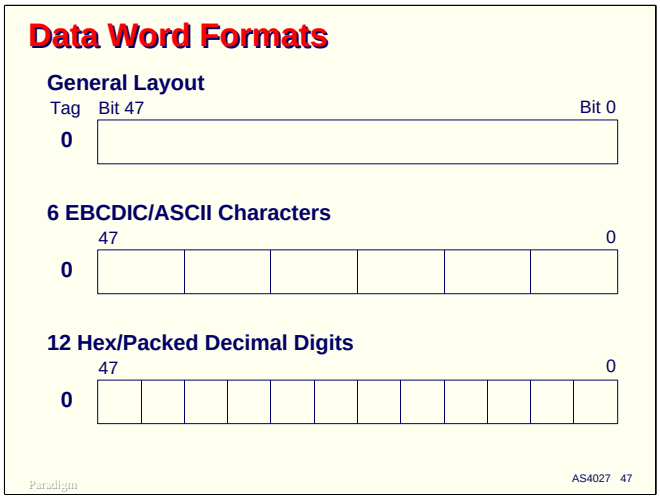
The value of the tag indicates to both the hardware and the MCP what kind of information is stored in the word. Values of 0, 2, 4, and 6 denote different types of data words, with tag=0 being by far the most common. The remaining tag values all denote various types of control words. Only words with certain types of tags are allowed to be used with most instructions, and the hardware rigorously checks this at run time. In some cases, operators work differently when used on words with different tags. If a tag value is inappropriate for the operator being executed, the hardware generally triggers an *invalid operand* (or "invalid op") interrupt.

Enforcement of the proper tag values helps enforce proper use of operands by the instruction set, along with memory protection, bounds checking, and overall security and integrity for the system. Tags are one of the reasons MCP systems do not suffer the same sensitivity to hacking as other systems – you can't just drop object code or addresses into memory and have the processor use that data as you intended. The words must have the proper tags, and except in certain well-defined and benign situations, user programs cannot set tags or create control words.



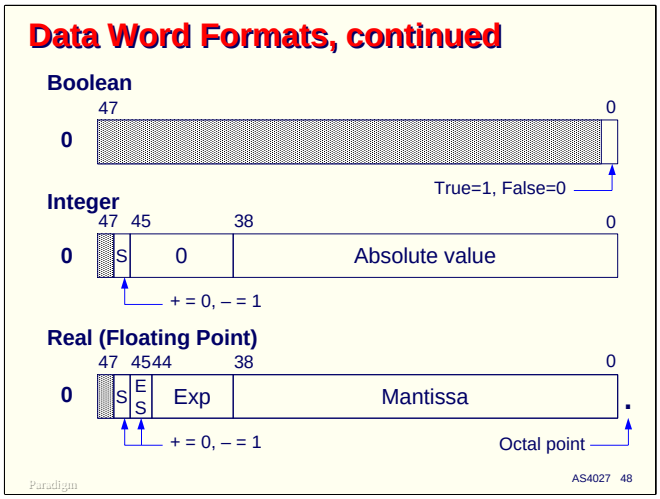
This slide shows how the sixteen values possible with a four bit tag field are used on level Delta systems.

- Tag=0 is the most common. It is used for most data words, including integer and floating point values, EBCDIC and packed-decimal strings, and Boolean values.
- Tag=1 is used for the Stuffed Indirect Reference Word (SIRW). It is also used for the Stack History Control Word (HISW) at the base of a stack frame.
- Tag=2 is used for double-precision words. These words normally appear in pairs.
- Tag=3 is used for object code, untouched code descriptors in segment dictionaries, and a number of miscellaneous control words.
- Tag=4 was something called a Step Index Word on B6000/7000 systems, but now is used primarily by the MCP to control software fault handlers (e.g., the ON <fault> statement in Algol).
- Tag=5 is used for untouched data descriptors.
- Tag=6 is an uninitialized operand. This is a write-only word. User-level programs can overwrite a word with this value, but attempting to read it causes an invalid operand interrupt. It is used as the initial value of Algol pointer variables. It is also used for the Software Control Word (SCW) that provides information to the MCP BlockExit procedure for deallocating complex memory objects.
- Tag=7 is used for object code addresses – the Program and Return Control Words (PCW, RCW).
- Tag=8 is used for words in the MCP ASD table.
- Tag=9 is pushed in the stack for some interrupts to give the processor additional information it needs to restart interrupted instructions.
- Tag=A is called a Protected Object Word (POW). It is used by a number of internal hardware and MCP facilities, including event variables.
- Tag=B is an Absolute Store Reference Word (ASRW). It allows the hardware to reference memory by an absolute word address. It does not appear in user programs.
- Tag=C is a touched, unpaged data descriptor.
- Tag=D is touched, indexed, unpaged data descriptor.
- Tag=E is a touched, paged data descriptor.



The next three slides show the formats associated with single- and double-precision data words. Detailed formats for the other tag values can be found in the *NX4800 and A 2800 Level Delta Architecture Support Reference Manual* (form 7012 6610-000), which is on the Product Information CD-ROM.

Single-precision data words have a tag of zero. The bits are numbered from 47 on the high end to 0 on the low end. The 48 data bits can hold six 8-bit characters (typically EBCDIC or ASCII), or twelve 4-bit hexadecimal or packed decimal digits.



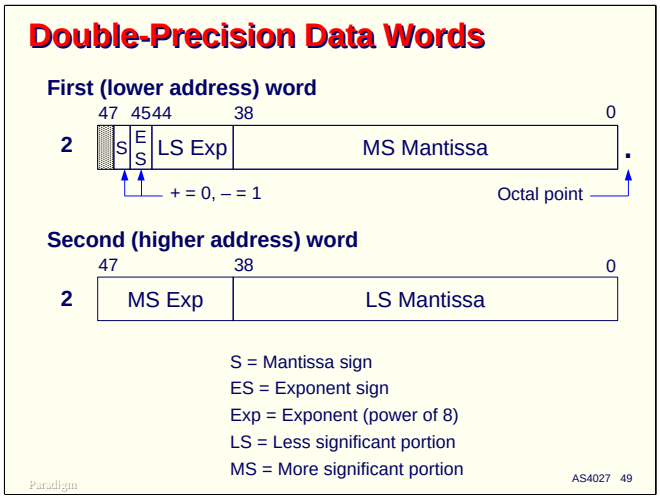
Single-precision words can also hold a Boolean value. For these words, the true or false value is determined solely from the low-order bit, bit 0. All of the other bits are ignored. Logical operators (e.g., AND, OR, NOT) operate on all 48 bits of the word, however.

E-mode machines (and all of their ancestors, back to the Burroughs B5000) use an unusual numeric representation. Integer and floating point values share the same word format, so that integers are essentially a subset of the floating point values.

Integer values are represented in the low-order 39 bits of the word. This value is stored as an absolute value, not in twos-complement like most other machines. Bit 46 is the sign bit. The number is non-negative if it is 0, and non-positive if it is 1. Plus and minus zero are both possible, but are numerically equivalent. Bits [45:7] must be zero in an integer value. Bit 47 is not used (it had significance on the B5000 and B5500, but tags took over this function starting with the B6500. The B5000/5500/5700 did not have tags).

Floating point values have the same layout, except that bits [45:7] are used for the exponent or scale factor of the mantissa (integer portion) of the value. Bit 45 is the exponent sign. Bits [44:6] are the exponent, which like the mantissa, is in signed-absolute value format. The exponent represents a power of eight. Each change in the exponent causes the mantissa to be shifted left or right three bits (a factor of eight). The decimal point (technically, the octal point) is to the right of bit 0.

This format explains why programmers must be careful manipulating bit masks or bit-packed words (such as COMS descriptors) with integer variables. If the word has non-zero bits in [45:7], the hardware is likely to normalize what it thinks is a floating-point number, with the result that the the bits in [38:39] tend to get shifted, often yielding disastrous consequences for the execution of the program.

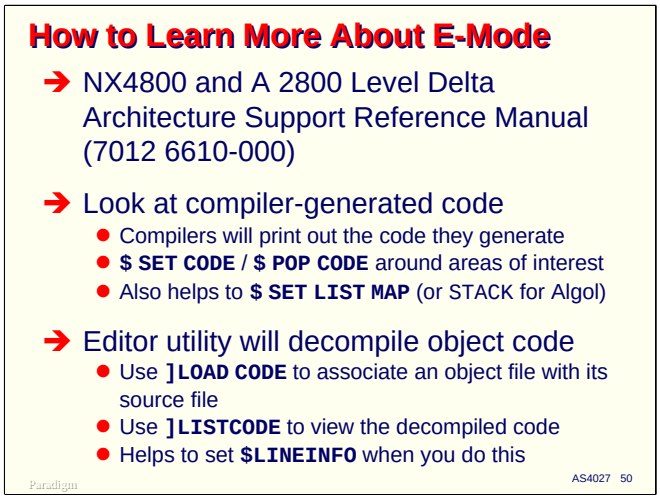


Double-precision words have tags of two and normally occur in pairs. The first word of the pair (at the lower memory address) has the same format as a single-precision floating point value. The second word contains extension fields for both the mantissa and exponent.

- The first word contains the more significant portion of the mantissa and the less significant portion of the exponent.
- The second word contains the less significant portion of the mantissa and the more significant portion of the exponent.

This arrangement was chosen so that a single-precision value could be converted to a double-precision one simply by appending a second word of zero bits.

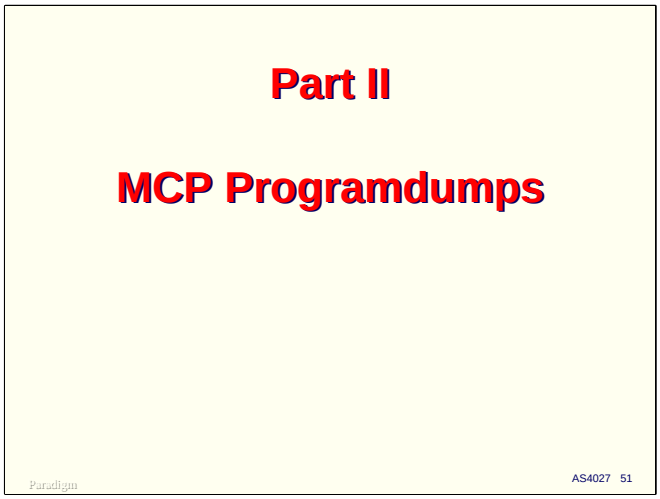
The decimal (octal) point is between the two portions of the mantissa. COBOL supports the concept of a double-precision integer. These items have their binary value right-justified in the mantissa of the second word, and an exponent value of +13 (base ten). This effectively shifts the octal point 13 octades to the right, to the point just to the right of bit 0 in the second word.



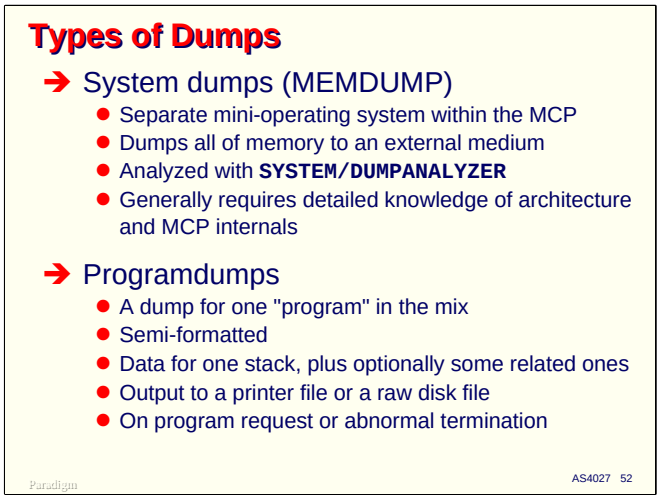
This completes our overview of E-mode in preparation for looking at programdumps. If you are interested in learning more about the architecture used with MCP systems, the *NX4800 and A 2800 Level Delta Architecture Support Reference Manual* (form 7012 6610-000) contains a fairly complete description of all of the word formats, the instruction set, and the manner in which addressing, procedure calls, and interrupts work. This is a highly technical document, and written in a somewhat abstract manner.

Another good way to learn more about how the architecture works is to look at the object code that is generated by the compilers. You can use the `$SET CODE` and `$POP CODE` compiler control options to list the generated code for a program. If you do not want to see the object code for the complete program (which can be quite voluminous) you can set and pop the `$CODE` option around just the statements or sequences of statements you are interested in. It helps to compile the program with the `$LIST` and `$MAP` (or `$STACK`) options set when you use the `$CODE` option.

If you have access to the system Editor utility (`OBJECT/ED`), it will display the object code in a pseudo-assembler format. To do this, run the editor against the source code for the program, and use its `]LOAD CODE` command to associate the corresponding object code file with the source. You can then use the `]LISTCODE` command to display the object code starting at a line number or a SDI:PSW:PSI address. In order to list code by line number, you must have set the `$LINEINFO` option when the program was compiled.

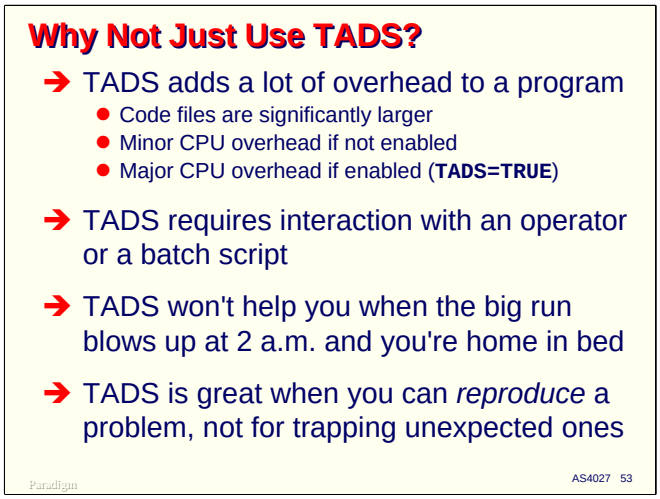


With that introduction to the E-mode architecture, we will now turn to the subject of programdumps themselves.



There are two main types of memory dumps you can obtain on MCP systems. The first is a full system dump, or `MEMDUMP`. This runs as a mini-operating system on the hardware, and copies all of memory to an external medium, typically a tape, a pre-allocated disk file (see the ODT `DN` command), or a CD-RW drive. This dump must be analyzed by the `SYSTEM/DUMPANALYZER` program, which is bundled with the rest of the standard software. Generally speaking, you must be very familiar with the machine architecture and internals of the MCP in order to read and understand these types of dumps.

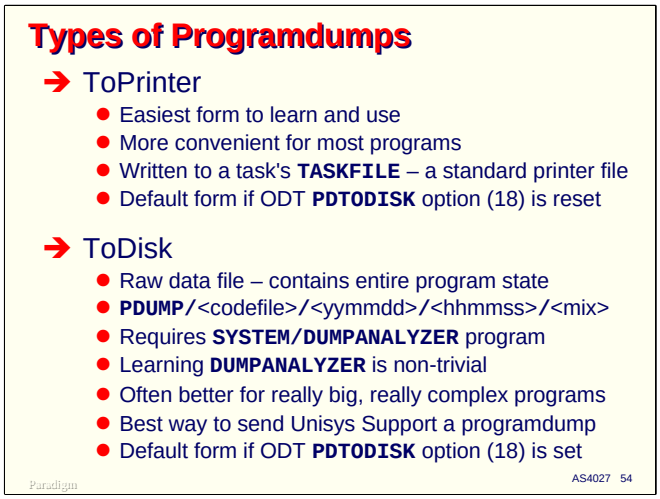
The second type of dump is the *programdump*. This is a dump of one "program" in the mix – one data stack and optionally some related stacks. This type of dump can be written either in a semi-formatted fashion to a printer file, or in a raw format to a disk file. Programdumps can be generated by the MCP on request by the program, on request by the system operator or a CANDE or MARC user, or automatically if the program terminates abnormally.



Some people eschew programdumps for one of the versions of the Test and Debug System, TADS. TADS is a wonderful tool (if you have paid for it – it's separately licensed), but it has its own set of limitations.

- TADS adds a lot of overhead to programs. The code files are significantly larger when compiled with the $TADS option. Even when running such programs in a non-debugging mode (TADS=FALSE), there is a small amount of overhead associated with each statement to evaluate the TADS statement triggers. When debugging is enabled (TADS=TRUE), the run-time overhead is significant. These factors generally make it undesirable to use the $TADS option when compiling programs for production use.
- TADS requires interaction with an operator, or a batch script, in order to run in debugging mode.
- For this last reason, TADS is not of much help when a production program blows up unexpectedly.

Therefore, TADS is great for reproducing and analyzing program behavior when you know you have a problem, but it's not much help in trapping an unexpected problem and giving you information about what was going on in the program when the problem occurred.



Programdumps come in two flavors – those written to a printer file in a semi-formatted manner, and those written to disk in raw form.

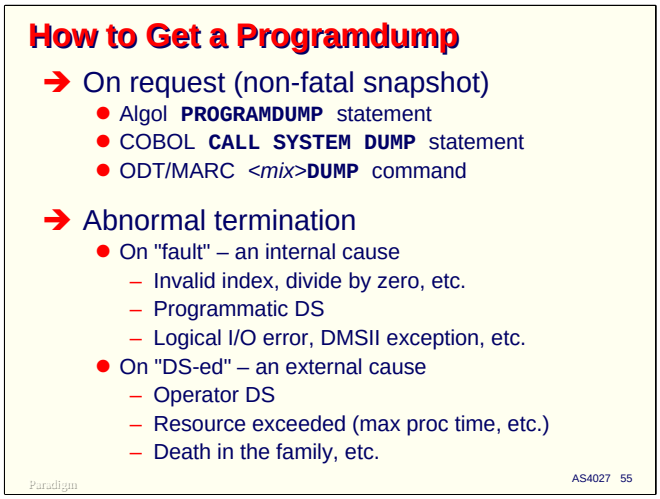
Programdumps to printer are usually the easiest to learn to read and to analyze. For most application programs, they are usually the most convenient form to use. Programdumps to printer are written to the TASKFILE. This is a file which is declared implicitly by the system for each task. It is (by default) a standard printer file. You can file equate the TASKFILE to give it custom printer attributes, or you can change the KIND attribute for the file. This is the default form of programdump if the ODT option PDTODISK (number 18) is reset, but you can override this default, as we will see shortly.

Programdumps to disk contain the entire program memory state. They are stored in a file under the directory PDUMP/=, with subordinate directory nodes consisting of the name of the code file, the date in YYMMDD format, the time of day in HHMMSS format, and the mix number of the program being dumped. These dump files will be stored under the usercode of the dumping task, and on the default family for the task, as determined by standard family substitution for the family DISK. You must use the SYSTEM/DUMPANALYZER program to analyze these dumps; learning to use this program is a non-trivial task. These dumps are generated by default if ODT option PDTODISK is set.

Programdumps to disk are often better for analyzing really big, really complex programs, since you don't have to print out the entire dump, and DUMPANALYZER can randomly access the objects in memory. It also has better formatting facilities for most of the system data structures than programdumps to printer provide. Programdumps to disk are usually the best way to send a dump to the Unisys Support group. One additional advantage is that they contain all of a program's memory state. With programdumps to printer, you must specify what portions of the memory state you want formatted on the dump.

If a programdump is written to disk, a brief summary is also written the the TASKFILE, indicating when the dump took place and the name of the PDUMP file.

For the remainder of this presentation, we will focus on programdumps written to printer files.

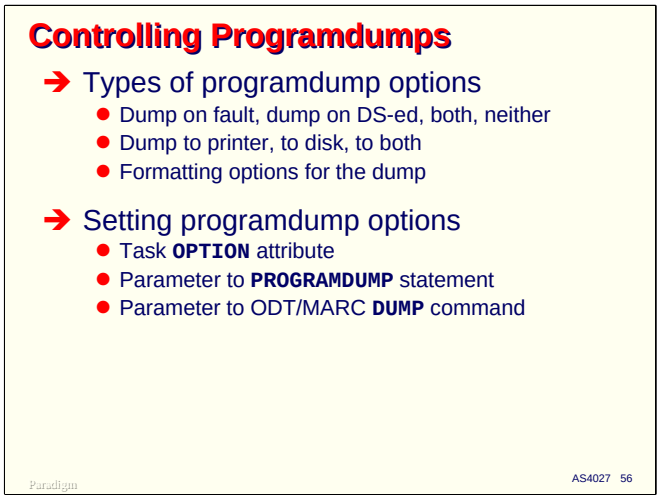


There are two basic ways to obtain a programdump. The first is simply to ask for it. Most languages allow you to call the MCP's ProgramDump procedure as an intrinsic. The dump occurs, and control returns to your program. It is, in effect, a snapshot dump, and you can request it as many times as you like during the execution of a program. In Algol, you use the `PROGRAMDUMP` statement. In COBOL-74 and -85 you use the `CALL SYSTEM DUMP` statement. See the *Task Management Programming Guide* (form 8600 0494) for snapshot facilities provided by other programming languages.

Users and system operators can also request a snapshot dump. On the ODT, you can enter a `DUMP` command along with one or more mix numbers. The next time the task is assigned to a processor, the dump will occur as if the program called the ProgramDump procedure. Make sure you enter at least one mix number with this command – a `DUMP` command without a mix number list activates the system-wide `MEMDUMP`. The snapshot `DUMP` command is also available in CANDE and MARC.

The second way to obtain a programdump is to have the system generate it for you if the program terminates abnormally. Abnormal terminations are divided into two classes, and you can request that a dump be generated for either class:

- *Faults* are terminations based on some cause that is internal to the program. These include the fault interrupts (divide by zero, invalid index, etc.), programmatic DS (setting the task status to -1 or `TERMINATED`), and software-detected errors such as `EOF NO LABEL` and DMSII exceptions.
- *DS* terminations are based on some cause external to the execution of the program. The most common of these is operator DS, but they can also be caused by exceeding a resource limit (maximum processor time, maximum lines printed, etc.), and task family issues such as "death in the family."

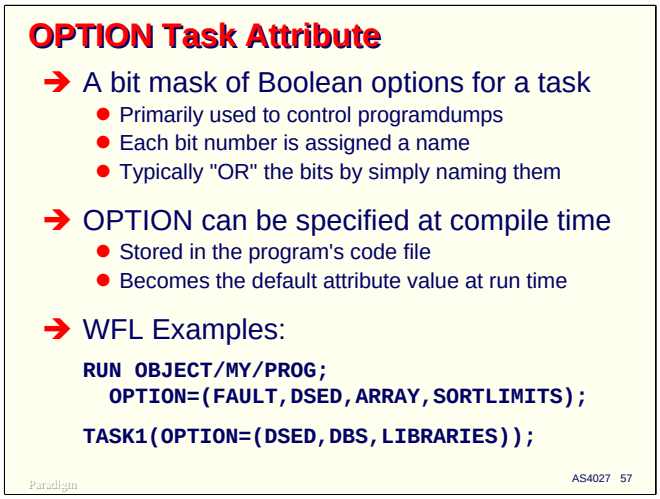


You can control three things about the way a programdump is generated:

- You can control the reason the dump is generated. In addition to requesting it programatically, you can specify whether you want a dump if your program terminates due to a fault (internal) cause, or due to a DS (external) cause, or both, or neither.
- You can specify that the dump is to be written to printer, or to disk, or both. If you do not specify this, the setting of ODT option 18, `PDTODISK`, determines the destination of the dump.
- You can specify a number of options that, for dumps to printer, control how the dump is formatted and what types of memory objects appear in it.

These programdump options can be specified in a number of places:

- Each task has an attribute called the `OPTION` word. This can be used to set a number of options for the task, including all of the ones for programdumps. We will discuss this attribute in more detail over the new few slides.
- The `PROGRAMDUMP` statement in Algol takes an optional parameter. This can consist of a list of keywords which specify programdump options, or it can be an arithmetic expression that yields a word value with the same bit layout as the `OPTION` attribute.
- The `DUMP` command for the ODT, MARC, or CANDE can optionally include the same list of dump options as can be specified for the `OPTION` attribute. If used, these will override those specified for `OPTION`.
- The `DS` command can also include the same list as the DUMP command. Thus, you can decide when you manually terminate a task whether you want a dump or want to suppress it.

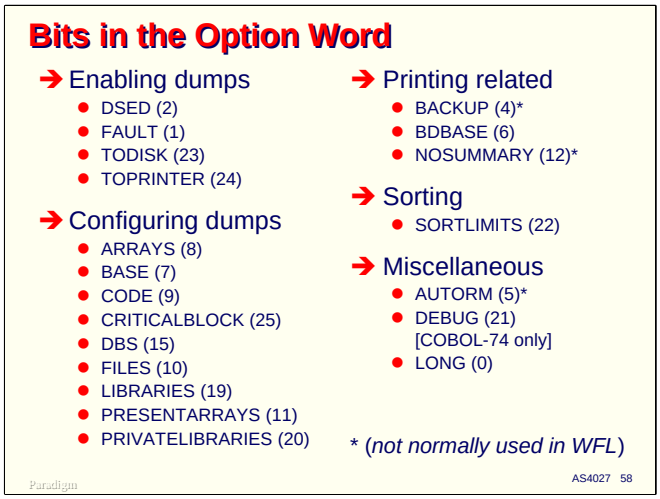


The `OPTION` task attribute is a bit mask of Boolean options for a task. You select the options by setting the appropriate bits in the attribute's value, which is a 48-bit word. `OPTION` is used primarily to control programdumps, but is has some other uses as well.

Each bit in the attribute's word has been assigned a name. You typically set the value of the attribute by simply specifying the bit names in a list. This effectively performs a logical "OR" of the bit values.

The `OPTION` attribute can be specified for a program at compile time. Like other task attributes assigned at compile time, its value is stored in the object code file, and becomes the default value for the attribute when the program is run. You can then override the default value using task equation at run time, or dynamically by task attribute assignment within the program. It's generally a good idea to set the `OPTION` attribute so that production programs will generate a programdump if they terminate unexpectedly – it costs nothing until the unexpected event actually occurs.

The slide shows two examples for setting bits in the `OPTION` attribute in WFL. Each option named in the list causes the corresponding bit to be set in the attribute's value. Similar syntax is available in CANDE and MARC for their `RUN` and `EXECUTE` statements.



This slide shows all of the bits defined for the `OPTION` task attribute as of MCP 7.0. The numbers in parentheses after the names indicate the bit number within the word to which they are assigned.

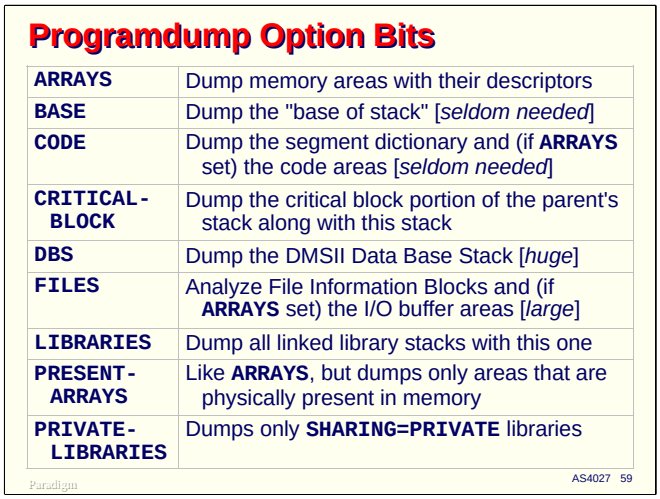
The first four bits control how a programdump is generated:

- **`DSED`** specifies that a programdump is to be generated if the program terminates due to a DS (external) cause.
- **`FAULT`** specifies that a program is to be generated if the program terminated due to a fault (internal) cause.
- **`TODISK`** specifies that the dump will be written in raw form to a `PDUMP/=` disk file.
- **`TOPRINTER`** specifies that the dump will be written in semi-formatted fashion to the task's `TASKFILE`.

You can specify any combination of these four options. Specifying `TODISK` or `TOPRINTER` will override the system-wide default determined by ODT option 18, `PDTODISK`. If both `TODISK` and `TOPRINTER` are specified, the dump will be written to both destinations.

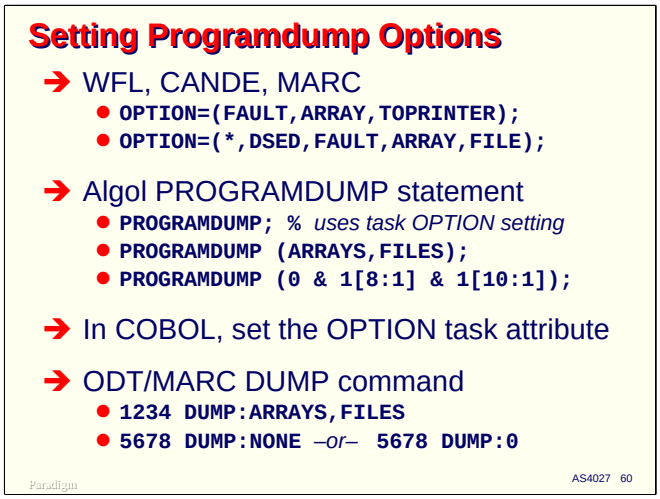
There are nine options for configuring dumps when they are sent to the printer. These are discussed in detail on the next slide.

The remaining options in the `OPTION` task attribute do not relate to programdumps. You can look in the *Task Attributes Programming Reference Manual* (form 8600 0502) to see how these other options are used.



The type and extent of information included for a programdump can be controlled using the following `OPTION` attribute settings:

- **`ARRAYS`** (or `ARRAY`) causes the memory areas pointed to by descriptors in the stack to be formatted on the dump. *You almost always want to set this option when you request a dump.*
- **`BASE`** causes the bottom-most words in the stack to be formatted on the dump. These contain MCP-related information for the task and typically tell you very little when analyzing normal application program problems.
- **`CODE`** causes the segment dictionary associated with the dumping stack to be formatted. This can be an interesting thing to look at if you are trying to learn more about the architecture of the system and how programs operate at run time, but it is seldom very useful when analyzing typical application problems.
- **`CRITICALBLOCK`** causes the "critical block" of a dependent child task to be dumped along with the child's stack. The critical block is that portion of the parent's stack which must remain active while the child is running. This includes the parent's outer block, plus any stack frames that contain globals or parameters referenced by the child task.
- **`DBS`** causes the DMSII Data Base Stack and Set Information Block (SIB) structures to be dumped if your program has a data base open. This generally produces a very large dump, and is not normally very useful unless you have a DMSII problem and know a lot about its internal structure.
- **`FILES`** (or `FILE`) causes the memory areas for File Information Blocks (FIBs) to be analyzed and included in the dump. If `ARRAYS` is set, the I/O buffers for the file will also be dumped. This is occasionally useful in analyzing an application problem, but it generates a rather large amount of output, and is most cases is better left unset.
- **`LIBRARIES`** causes the contents of frozen library stacks to which your program is linked to be included in the dump. This allows you to see the global variables in the library, which is often critical if you are trying to debug the library itself.
- **`PRESENTARRAYS`** (or `PRESENTARRAY`) is like the `ARRAYS` option, but causes only those memory areas which were actually resident in memory at the time of the dump to be included. If you also specify `ARRAYS`, it overrides `PRESENTARRYS`.
- **`PRIVATELIBRARIES`** is similar to the `LIBRARIES` option, but causes only libraries with the attribute `SHARING=PRIVATE` to be included in the dump.



As mentioned previously, options for programdumps can be set in a variety of ways. This slide shows some examples of the various methods.

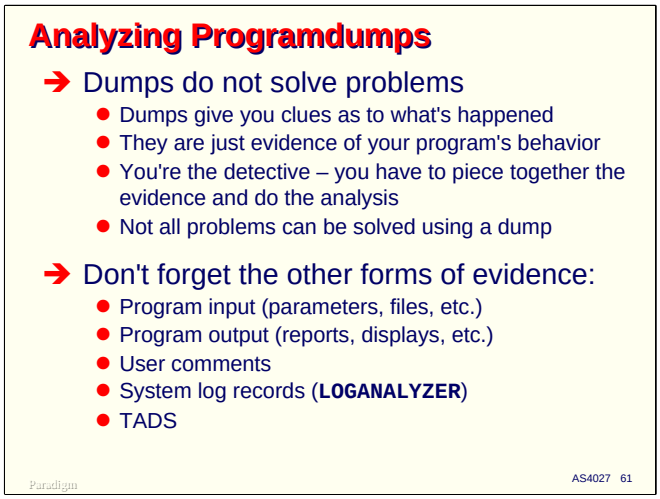
The `OPTION` attribute can be set in most environments which initiate programs. In WFL, MARC, and CANDE, the attribute can be specified as a modifier on a `RUN` or `EXECUTE` statement. In WFL, the attribute can also be assigned to a task variable. If you simply mention a list of option bit names, the resulting bit mask will replace any former contents in the `OPTION` word. If, however, you precede the list with an asterisk, as shown in the second example on the slide, the list of options you specify are "or-ed" with the existing bits in the `OPTION` word.

The Algol `PROGRAMDUMP` statement takes an optional parameter. If this parameter is not specified, the dump will use the current set of options in the task's `OPTION` word. When you specify the parameter, it can take one of two forms:

- The parameter can consist of a list of program dump option keywords, as you would specify them in WFL or CANDE. The compiler "ors" these together and generates a bit mask for the parameter.
- You can pass an arithmetic expression which sets the appropriate bits in a 48-bit word.

The `CALL SYSTEM DUMP` statement in either variant of COBOL does not accept a parameter, but you can set the task's `OPTION` word at run time using a `CHANGE ATTRIBUTE` statement.

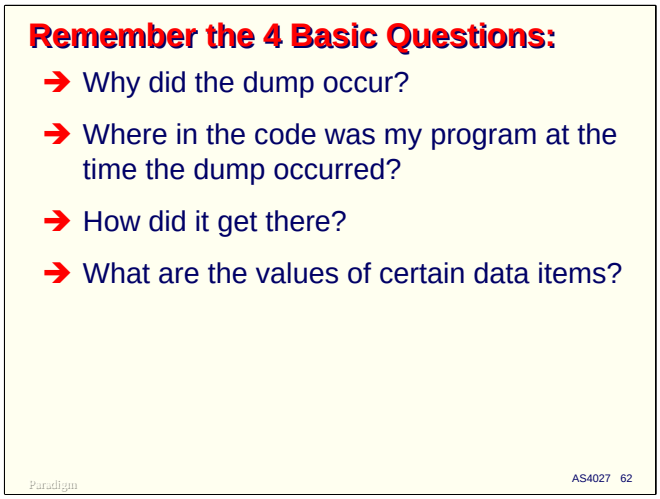
Finally, the ODT `DUMP` and `DS` commands available on the ODT and through MARC and CANDE can include a list of programdump options. These will override any options set in the task's `OPTION` word. If you want to suppress a dump when DS-ing a task, include an option of zero or the keyword `NONE` after the `DS` command.



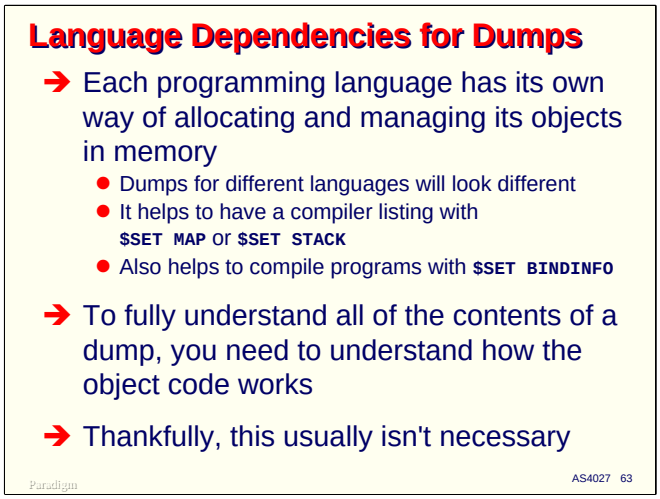
Before looking at some sample dumps, it's important to realize that dumps do not, of themselves, solve problems. They simply give you clues as to what is happing in the program at the point the dump was taken. They are just a form of evidence, and you must be the detective who pieces together the clues and analyzes them for meaning. It's also important to understand that not all problems can be solved by analyzing a dump.

Don't forget that MCP systems are capable of giving you a lot of evidence of a program's behavior other than a programdump. The other things that you will typically want to consider examining are:

- The input data to the program. Problems are often data dependent.
- The output from the program. This will often reveal that the program went awry at some point before the dump was taken.
- Comments from users and operations staff. They will have often noticed abnormalities in the behavior of a program that will suggest where the problem lies.
- The system log files are a rich source of information about program behavior, including especially file opens and closes, and linkages to library programs.
- Finally, TADS can be a critical resource if the procedural flow through the program is not clear, or if you need to examine changes to variables over time as the program runs.



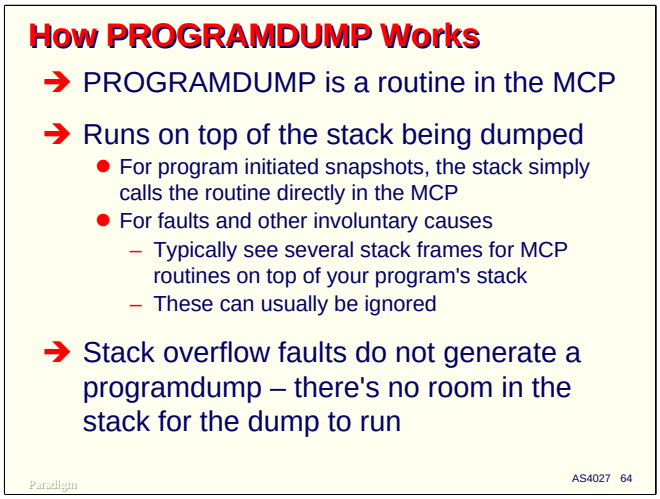
As we discussed in the beginning of this presentation, there are four basic questions that you typically need to answer when analyzing a dump. Try to keep these in mind as you approach the task of generating a dump and analyzing it.



Each programming language has its own way of constructing objects in memory at run time and allocating those objects in the stack. Therefore, dumps for programs from different languages will look significantly different.

It helps, especially when you are first learning to read dumps, to have a complete compilation listing available with the option $MAP or $STACK set. This will cause the compiler to print additional information on the listing, showing the stack locations it has assigned to variables and other objects in the program. Once you have more experience in reading dumps, you can often dispense with this extra output, especially for Algol and COBOL-74 programs.

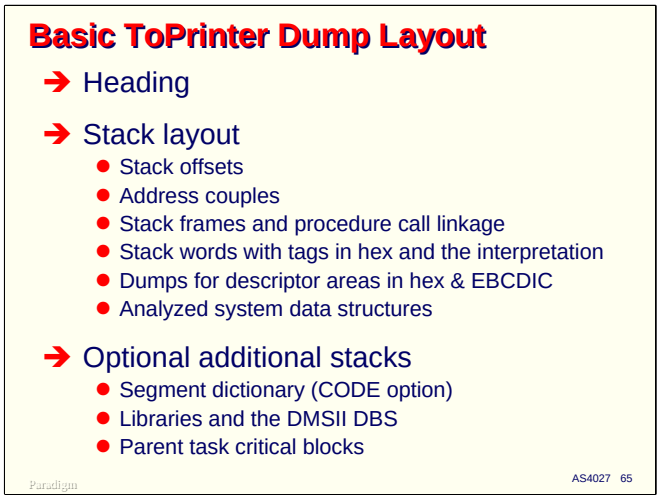
To really understand all that is in a dump, you need to understand how the compilers generate object code, and how that code operates at run time. Learning this in detail can be a geek's delight, but this depth of knowledge is not normally required to analyze dumps for most typical application program problems.



Programdumps are generated by the MCP procedure ProgramDump. This procedure runs on the stack that is being dumped, and simply examines the stack frames below it to format the dump.

If the dump is a snapshot being requested by your program, you will see the first stack frame for ProgramDump at the top of the stack, with your program's stack frames below that. If the dump is generated due to an abnormal termination, or by means of a `DUMP` command, you will typically see several stack frames for MCP routines at the top of the dump. These typically tell you very little, and can usually be ignored. Simply follow the stack frames backwards down the dump listing until you get to the first which for which the RCW in the frame indicates "`User segment @` …" or "`Stack xxxx segment @` …" The frames below that will generally be the ones of most interest to you.

If your program exceeds the area allocated for its stack, you will not get a programdump. The reason for this is that the ProgramDump procedure must run in your stack, and there is no longer any room for it to do so. You will need to find another way to analyze the problem that is causing the stack overflow.



A programdump formatted to a printer file is simply a picture of your program's stack at the time the dump was taken. There is a heading formatted at the beginning of the dump which identifies the hardware stack number, job/session number, and mix number of the program, along with its name and the date and time. Additional lines on the dump identify

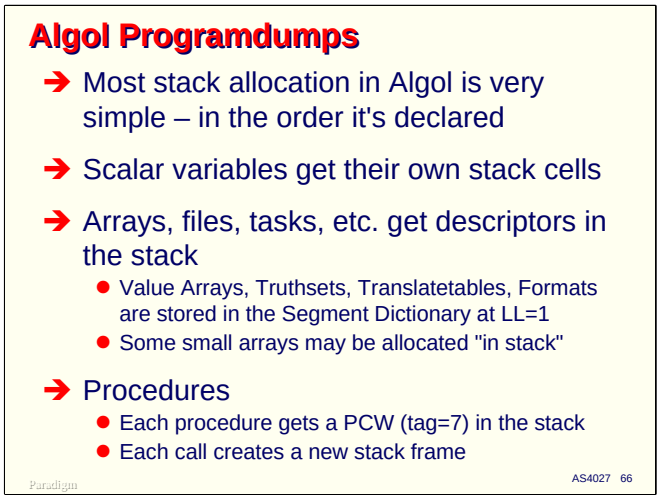
- The version of the MCP, the system serial number, and system hostname.
- Generic cause of the dump, along with the first part of the procedure call history. This is a series of object code addresses, in SDI:PWI:PSI format, with the latest procedure return address (the one where the dump occurred) listed first. If `$LINEINFO` was set when the program was compiled, line numbers will also be included.
- The task history word in hexadecimal. This is a bit-packed word which indicates the specific reason the program terminated. You can look in the *Task Attributes Programming Reference Manual* for the `HISTORY` attribute, which shows the layout of this word.
- Options specified for the programdump.

Following the heading is the dump of the stack. Along the left side is a four-digit hexadecimal number. This is the *stack offset*, and is the number of words from the bottom of the stack. Many control words refer to this offset value. Next to that is the address couple in *(LL, offset)* format that addresses that word within its stack frame.

Next to the stack offset and address couple is the stack word itself. It is formatted in three parts: a single hexadecimal digit for the tag, followed by the word's 48-bit value in two hex groups of six digits each. To the right of this, the programdump presents an analysis of the word's contents, based on its tag and position in the stack frame.

The programdump double-spaces the listing between stack frames, and shows an arrow with the relevant D register pointing to the base of each frame. The most useful word in the base of a frame is the one just above the D-register arrow. It should have a tag of 7 and be labeled an RCW – this shows the object code address to which the program would return when (if) it exits this frame. If you follow the stack frames backwards (towards the bottom of the stack), you will see the complete history of procedure calls that led to the point where the dump was taken.

Additional stacks, if present will be formatted after the main one.



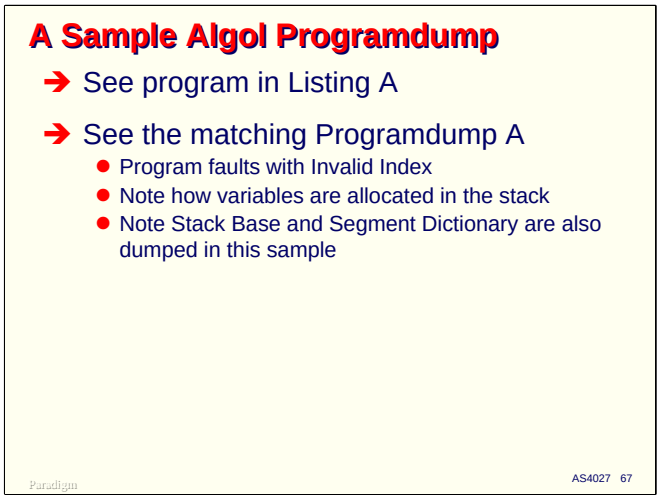
The Algol compilers allocate variables in the stack in a very straightforward way. Most variables are allocated in the order they are declared. Scalar variables such as integers, reals, Booleans, doubles, and complex variables are allocated directly in the stack.

Arrays, files, tasks, and other more complex objects get descriptors, which point to their separately allocated memory areas. Some very small arrays may be allocated "in stack" – they still get a descriptor, but the area for the array may be placed directly above the descriptor in the stack. If this type of array is resized, the original area in the stack is simply abandoned.

Read-only objects, such as Value Arrays, Truthsets, Translatetables, and Formats are allocated via descriptors in the segment dictionary. The data for these types of areas is stored as segments in the object code file, and loaded when they are first touched, just like code segments.

Procedures are allocated as PCWs in the stack. Entry points in calling programs to library procedures are initially allocated as a tag=5 word (an untouched data descriptor) with an invalid unit size (7). The remainder of the word contains information for the MCP to locate the library and entry point. When the calling program first tries to call the library procedure, referencing the invalid tag=5 word causes an interrupt. The MCP recognizes the invalid descriptor as an untouched entry point reference, builds the linkage to the library program, and replaces the invalid descriptor with an SIRW to the entry point's real PCW in the library stack. It then exits the interrupt, restarting the procedure call, which this time succeeds.

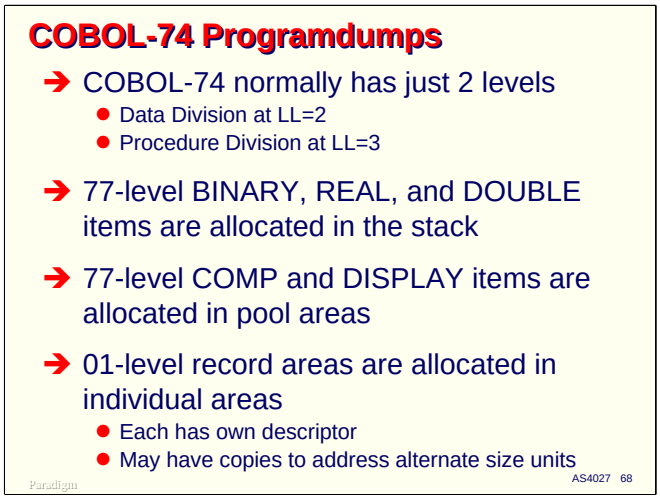
To learn how other types of objects are allocated by the Algol compiler, write a little program that uses the object, generate the object code listing with `$CODE`, and take a dump. See for yourself.



A sample Algol program is shown in the attached Listing A, with a corresponding dump A. This program does not do anything productive – it merely illustrates how a number of Algol variables look in the stack and how stack frames are built as procedures are called. The program was compiled with $MAP (the same as $STACK) set. You can see the address couples the compiler assigns to variables down the left side of the listing. For the outer block at LL=2, you should be able to find the corresponding stack cells in the dump. For higher lex levels, you need to find the matching stack frame first, then apply the address couple to find the variable in the stack.

This program faults with an invalid index interrupt, which is what generated the dump. The dump listing has been annotated by hand to highlight items of particular interest. As an exercise, try to find the following items:

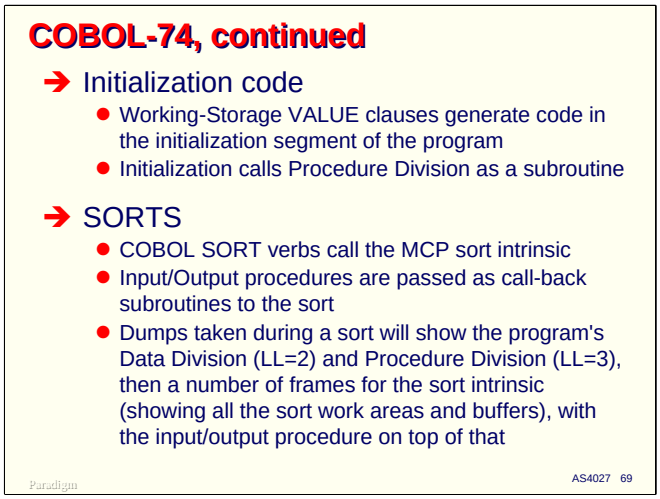
- Note the MCP stack frames at the top of the stack. You can follow these backward until you get to the stack frame with a line of asterisks – this is the stack frame for the interrupt handler, which is reporting a "boundary error" due to the invalid index. The RCW in this frame shows the address in the user program where the interrupt occurred.
- The point in the program where the interrupt occurred was indexing the array C by the integer variable X. C was passed as a parameter, and if you look in the dump at offset 0047, it shows a copy descriptor indicating the mom descriptor is at offset 0020 in this stack. Looking at offset 0020 shows it's the array BIG, which is declared with bounds 0:999999. Looking at the variable X (at offset 0048) shows that it has a value of 1000000, which is too big for BIG. Voilà.
- Looking at stack offsets 0018-001D, you will see the stack locations for a number of scalar variables. Note the two words with tags=2 for the double-precision variable Z.
- Immediately above Z at offset 001E is the descriptor for the FIB for file PRINT. Since the option FILES was not specified, this memory area is not analyzed.
- Immediately above that, at offset 001F, is a descriptor for the array A. It has bounds of 0:15, and the dump shows that it is 16 words long. The data for the array is formatted immediately below the descriptor. Note how the dump compresses sequences of identical words.
- Above the descriptor for A is the one for BIG. Note that this is a segmented (paged) array. The length of this array is reported as 1000000, but since it's paged, the mom actually points to a dope vector for the pages. You can see these as the first level of indentation. Two of the pages have been touched, and you see their areas formatted below them. A two-dimensional array would look very similar in a dump.



COBOL-74 programs normally have just two lex levels and two stack frames, one at LL=2 for the Data Division and one at LL=3 for the Procedure Division.

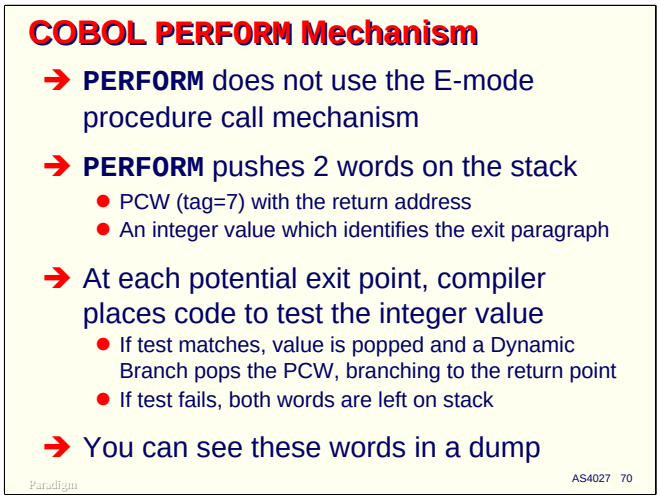
01- and 77-level BINARY, REAL, and DOUBLE elementary data items are allocated directly on the stack. 77-level COMP and DISPLAY elementary items are allocated in a pooled memory area pointed to by a descriptor. In some cases, this pool may be allocated "in stack," as the attached sample program shows.

01-level record areas are allocated as a descriptor on the stack. Each 01-level record area is therefore separately allocated. These are very easy to find in a programdump. COBOL-74 will frequently have multiple descriptors for the same area – the mom will describe the area with one size of unit (say, 8-bit bytes), and copy descriptors will describe the area as other units (typically 4-bit characters for packed decimal fields and 48-bit words). The copies are usually adjacent to the mom in the stack.



The Data Division in a COBOL-74 program acts like the outer block of an Algol program. The Data Division is actually executable – it contains instructions to build the stack for the program's variables, and if there are VALUE clauses, to set the variables to their initial values. At the end of the Data Division, it calls the Procedure Division as a parameter-less procedure.

If the program has sort input or output procedures, you will see the stack frames for the Data Division and Procedure Division towards the bottom of the stack, then a number of stack frames for the MCP's sort intrinsic, with a stack frame for the input/output procedure running on top of that. COBOL programs pass input/output procedures as a parameter to the sort, and the sort calls them to release or return records.



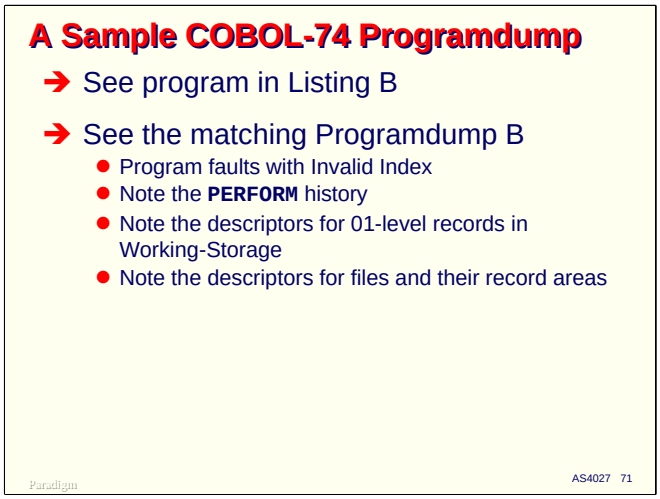
As we discussed in the first part of this presentation, the COBOL PERFORM mechanism does not use the E-mode procedure call mechanism, which was designed to support the semantics of Algol. COBOL PERFORMs are a lot simpler, and the COBOL compiler generates a much more efficient sequence of instructions to accomplish these primitive subroutine linkages.

When you code a PERFORM statement, the compiler pushes two words into the stack and simply branches to the label that begins the PERFORM range. The first word is a PCW containing the return address for the perform (i.e., the code address right after the branch). The second word is a small integer that identifies the terminating paragraph of the PERFORM range. At the end of each paragraph which ends a PERFORM range, the compiler generates code to test the value at the top of the stack against the number for that paragraph. If the values match, the top word is popped from the stack, and a dynamic branch instruction (DBUN) pops the PCW from the stack and branches to the return address. If the values do not match, both the small integer and the PCW remain on the stack.

This mechanism has two advantages:

- It allows PERFORMs to be recursive, as required by the COBOL standards.
- You can see the history of PERFORMs in the programdump for a stack (actually, what you see is the history of return addresses for PERFORMs yet to be exited).

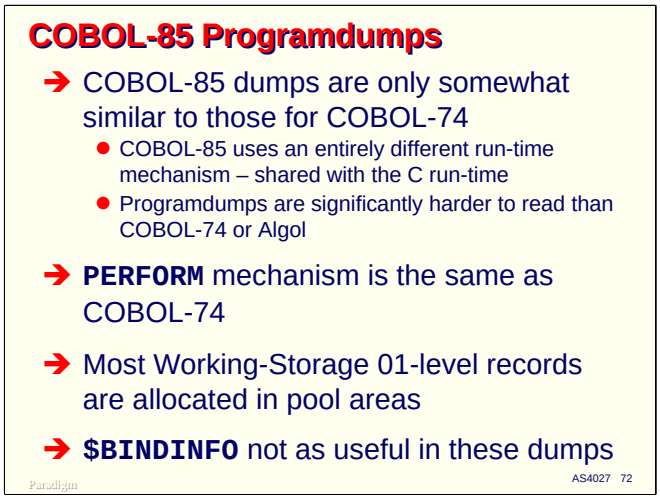
This mechanism has the potential disadvantage that if you branch out of a PERFORM, the exit mechanism can become confused, and it's possible to get a stack overflow.



A sample COBOL-74 program is shown in the attached listing B, with a corresponding programdump B. The compilation listing has $MAP set so you can see the address couple assignments made by the compiler.

This program also faults with an invalid index, due to stepping off the end of an occurring data item. The things to note particularly in this dump are:

- The stack frame for the MCP interrupt handler, which shows where in the program code the interrupt occurred.
- The PERFORM history in the LL=3 stack frame below the interrupt at stack offsets 0056-0059. You can see two sets of PERFORM words, each with its paragraph number and PCW return address. Unfortunately, the programdump does not translate the SDI:PWI:PSI object code address to a line number, so you will need a compilation listing to find these addresses in your program.
- The 01-level record areas for Working-Storage at stack offsets 0033-0039. COBOL-74 dumps are much easier to read if you set $BINDINFO when the program is compiled, as this causes the programdump to format the 01-level record name next to the descriptor for the area. With this option set you can usually read COBOL-74 dumps without requiring $MAP to be set during the compile.
- The 77-level elementary binary items at stack offsets 0020-0027, and the 77-level data pool's descriptor at offset 0028. Note that this is an "in stack" area, with the pool itself existing at stack offsets 002A-0031. This in-stack allocation is normally used only if the pool is very small.
- Descriptors for file FIBs and their 01-level record areas are at the bottom of the LL=2 stack frame. Note that the record area is adjacent to its FIB descriptor for the two files at offsets 001B-001C and 001E-001F.

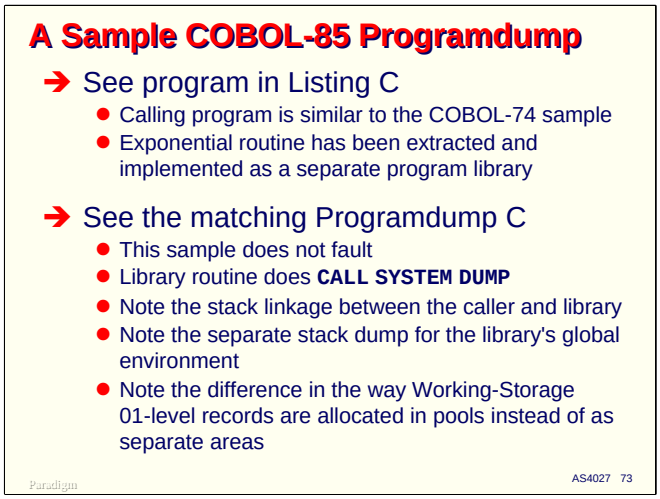


While with a little practice COBOL-74 dumps are usually quite easy to analyze, the situation is much more complex for COBOL-85. This language uses a completely different run-time mechanism, one that is also used with the C language. The `PERFORM` mechanism, however is identical to that of COBOL-74.

If you write a COBOL-85 program like a COBOL-74 program, you will see two stack frames, one at LL=2 for the Data Division and one at LL=3 for the Procedure Division. 77-level elementary items are allocated as they are for COBOL-74.

Most 01-level record areas are, unfortunately, not allocated as separate descriptors, but are instead grouped into pool areas. This makes their location in the pools almost impossible to determine without a compilation listing which has `$MAP` set. 01-level records passed as parameters appear always to be allocated with their own descriptor. Setting `$BINDINFO` for COBOL-85 programs is still a good idea, but the annotations this option enables on programdumps are not as useful as they are in COBOL-74.

If you write COBOL-85 programs with nested programs, the stack frames in the dump will look more like an Algol program. Calling a nested program is similar to calling an Algol procedure, and local variables for the nested program will be allocated in the nested program's stack frame (although the `$COMMON`, `$LOCALTEMP`, and `$OWN` compiler options can have an effect on this).



A sample COBOL-85 program is shown in the attached listing C, with a corresponding programdump C. The compilation listing has `$MAP` set so you can see the address couple assignments and offsets within data areas made by the compiler.

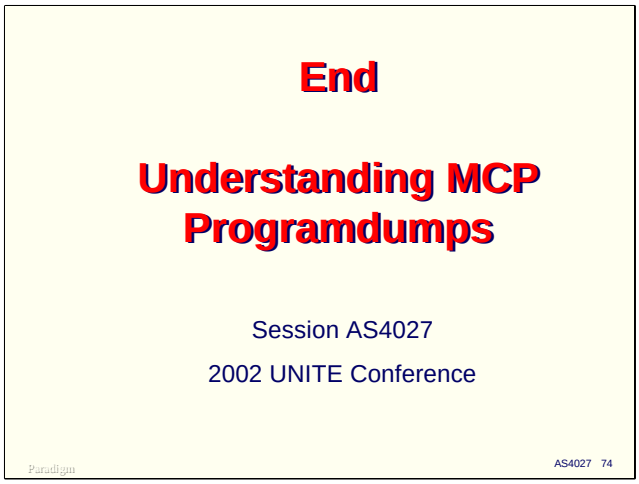
This program does not fault. It is essentially the same program as the COBOL-74 example, with one of its internal routines (which computes the natural logarithm function) extracted and implemented as a library program. The library program executes a `CALL SYSTEM DUMP` statement to generate a snapshot programdump. The dump was generated with the `ARRAYS` and `LIBRARIES` options, so you can see the global environment of the library formatted as a separate stack.

In the first stack dump, you can see the local environment for the library entry point procedure in the stack frame starting at offset 004F. Below that is the stack frame for the calling program's Procedure Division, starting at stack offset 003E. You can see the `PERFORM` history words at offset 004B-004E.

Below the stack frame for the caller's Procedure Division is the one for its Data Division at LL=2. In this you will see some formatted data structures for the library linkages. At offset 002B is the large pool area that holds the 01-level record areas for Working-Storage. The `$MAP` output on the compiler listing shows the offsets within this pool area to the individual data items.

At offset 001F, you will see a SIRW for the library entry point referenced by the calling program. Since this entry point has already been touched by the time the library procedure has been called, it has been fixed up to be a SIRW pointing into the library stack to the PCW for the procedure.

The library stack is dumped after the caller's stack, and has a similar structure. You can use the `$MAP` output on the library program's compilation listing to find the global variables used by the library.



This concludes the presentation on basic analysis of MCP programdumps.

Copies of this presentation and the annotated programdump listings are available at our web site:

`http://www.digm.com/UNITE/2002`

If you have any questions, you are welcome to contact me at the email address below.

http://www.digm.com

e-mail: paul.kimpel@digm.com