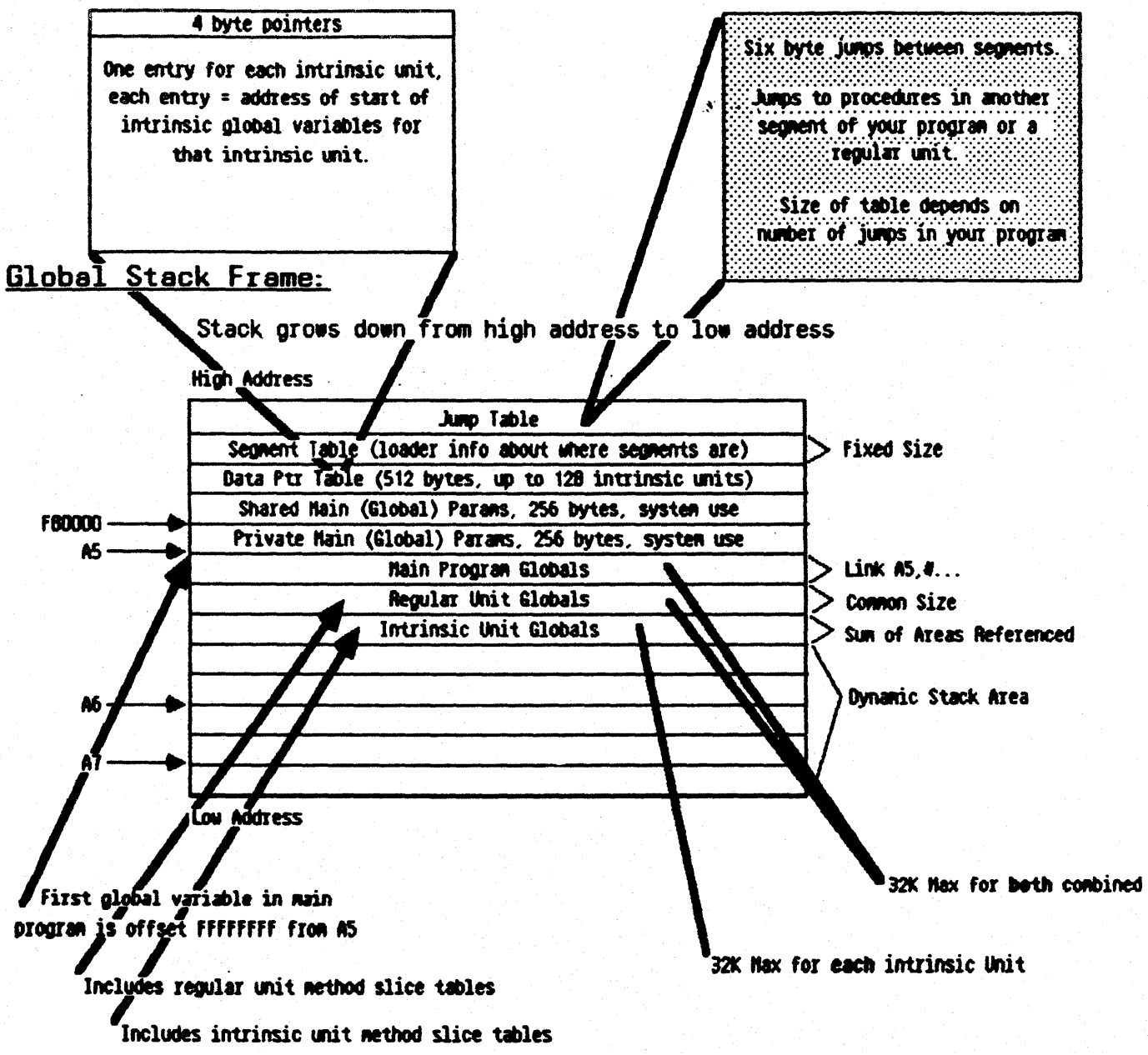
More on Debugging

Lisa Stack Frame Information

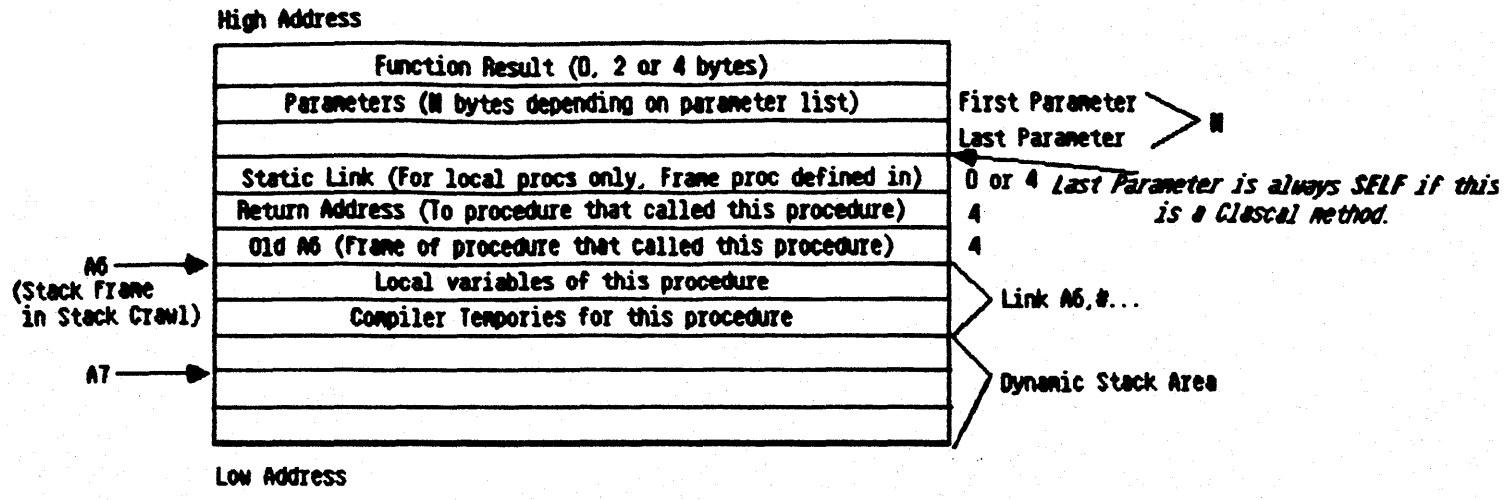
Register Usage:

D0 - D2 and A0 - A1 Can be used as user temporaries by your procedure
D0 - D3 and A0 - A2 Used for compiler temporaries
D4 - D7 and A3 - A4 Compiler uses for locals and pointers
A5 Pointer to global stack frame (for main program)
A6 Pointer to current local stack frame (current procedure)
A7 Pointer to the top of stack (Supervisor if domain = 0 otherwise user)

Global Stack Frame:



Local Stack Frame (usual case):

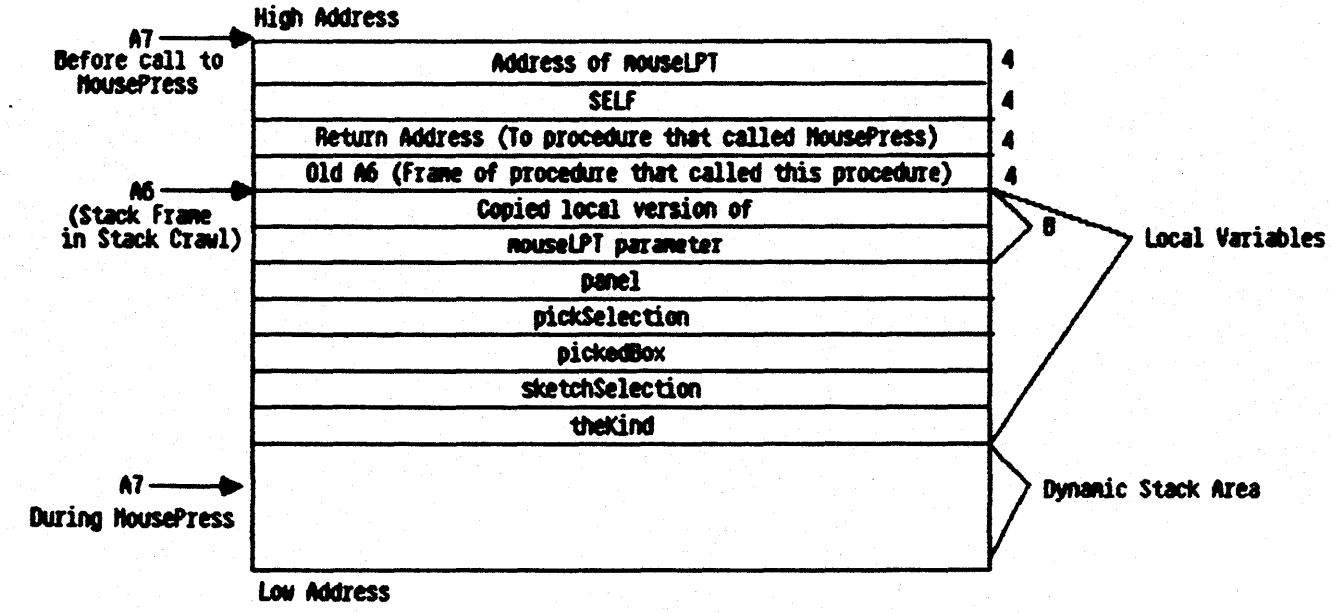


Parameter Information:

Info passed for Parametric Procedures and Functions:

Address of procedure body	4
Static Link, value = 0 if this is not a local procedure	4

Stack Frame of TSamView.MousePress:



Local Variables Assigned to Registers in TSamView.MousePress

panel = A3
sketchSelection = D6
theKind = D7
pointer to UABC globals = A4
pickSelection = D5

Using LisaBug in the ToolKit — Clascal Environment

Moving Around Stack Frames

You can use the following information about stack frames and register usage while debugging in the ToolKit environment. The Global stack frame is the stack frame for the main program. There is one global stack frame for each process. All other procedures, functions and methods have a unique local stack frame created each time they are called. Local stack frames for methods always have the handle SELF as the last parameter. Local stack frames for procedures and functions that are not at the outer most level will contain a Static link pointing to the stack frame of the procedure they were defined within. This allows that local procedure to access variables defined in it's parent procedure. When a local procedure is passed as a procedure parameter, this static link is also passed. Clascal methods, procedures and functions declared in the interface of a unit and procedures declared in the outer most level of the main program never contain a static link. They don't have to since their parent procedure is the main program whose stack frame is always referenced by the A5 register.

Finding Where You Are Within a Clascal Method - Comparing Source to Object Code

Here is an example of a Clascal Method.

```
PROCEDURE TSamView.MousePress(mouseLPt: LPoint);
 VAR  panel:              TPanel;
      pickSelection:      TPickSelection;
      pickedBox:          TBox;
      sketchSelection:    TSketchSelection;
      theKind:            INTEGER;
 BEGIN
      {$IFC fTrace}BP(10);{$ENDC}
           .
           .
           .
      {$IFC fTrace}EP;{$ENDC}
 END;
```

A method is similar to a procedure or function call. The runtime environment for the method includes a stack frame containing information used in that particular call of the method. The stack frame contains a place to store all the local variables and parameters to the method. It also contains information about the procedure that called the method. This is the caller's return address and the location of the caller's stack frame. You can use this stack frame information to look around on the stack and inspect the local variables of all methods in the call chain. The local stack frame of TSamView.MousePress can be found on page two of this section. Notice where the particular parameters and local variables for this method ended up in the stack frame. To understand what code is generated for each method source statement, you can use compiler options to create a listing of your source along with the assembly

language format of the code generated. The use of these compiler options is documented in the 3.0 Workshop manuals. After you study a few examples of this while looking at the actual code with LisaBug, you will find it is not that hard to figure out what line of code you are executing within a particular method.

Using IL (((AHandle))) to See What Handle You Have or If You Have a Handle

You can find out if a certain address is a handle, or if you have a handle to the correct thing. In LisaBug, you can continue to add levels of indirection to IL of that address until you get to the CREATE (NEW) method of some class. You will get there if you have a handle then the name of that class will be listed by LisaBug. You can do this with an address you key in or with a Register. A handle points to a master pointer which points to an object, the first field of the object points to the method table for that object. The first entry in that table points to the CREATE (NEW) method.

What It Probably Means When You Call a Method You Did Not Expect to Call

If you see an object or method in your stack crawl that makes no sense, you may have performed a coercion, assigning one handle to another, and ended up with an object of a different type than you intended to have. This indicates a bug in the logic of your program that can be confusing to find while debugging. If you have range checking turned on {$R+} these errors will be caught by the system when they happen except when a coercion is done by passing a handle as the actual parameter to a formal VAR parameter.

Domain Hassles and the NV Bug

By entering LisaBug through the ToolKit debugger, you are forced to enter while executing code in the domain of your application. This means the symbols you want to see, the methods of your application, will be available in LisaBug. If you enter LisaBug in Domain 0, you will notice that it does not know about your symbols. The ToolKit debugger saves you from this type of hassle except for the NV Bug. This is a bug in LisaBug that occurs when one of your procedures contains the characters 'NV' next to each other in that order and this combination is also on a word boundary. LisaBug suddenly thinks it has found a link instruction, or something, and for some strange reason, this makes LisaBug forget all the rest of the symbols in that segment! If you ever enter LisaBug from the ToolKit debugger and it doesn't know about your program symbols, that is probably the problem. The solution is to rename the procedure containing the NV.

Strategy When You Crash and Fall into LisaBug

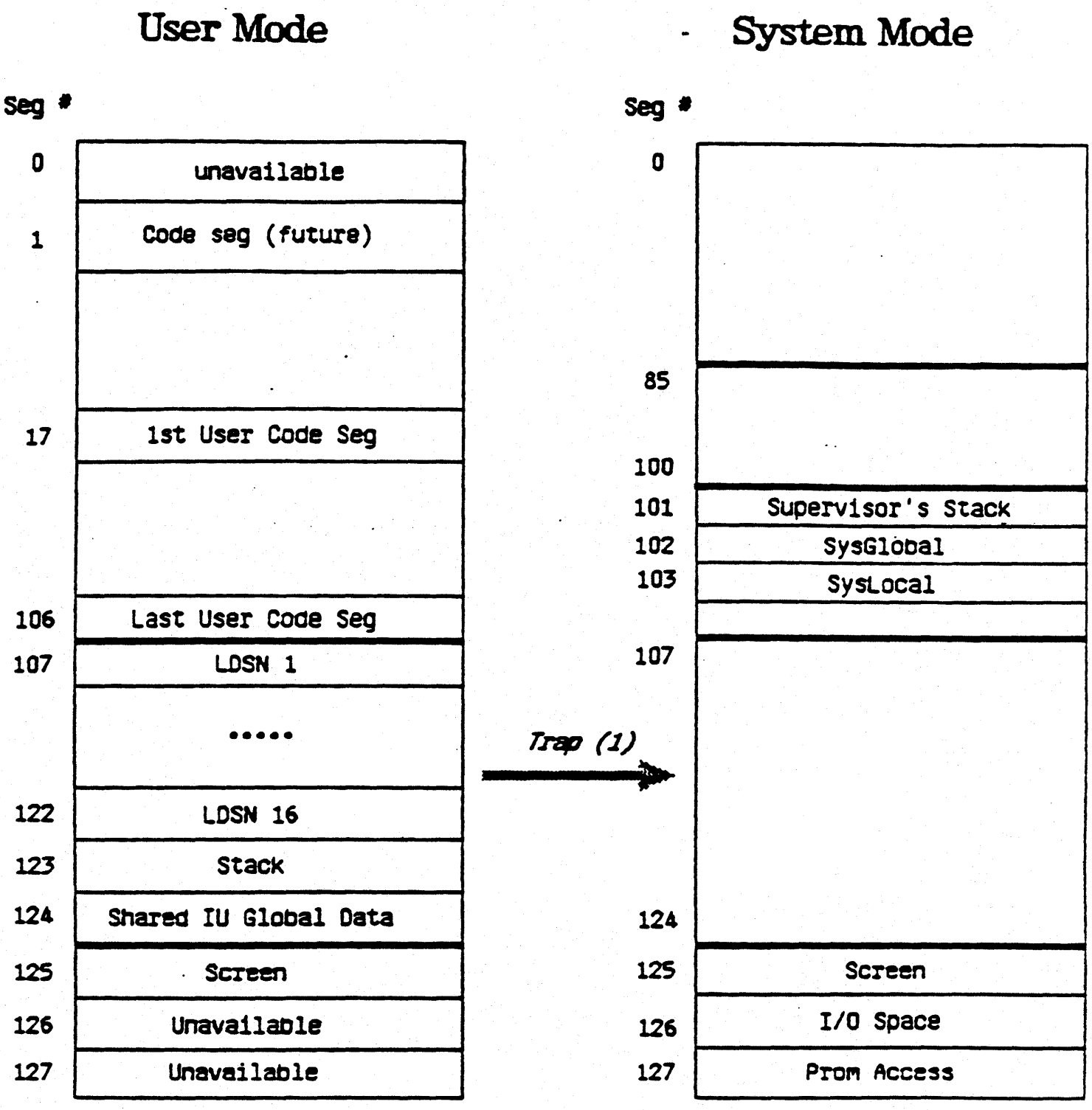
First do a Stack Crawl to get an idea of where you are. You can also do an IL of PC-20 to look at the code surrounding the crash. If you have an address error or something and you need to figure what source statement you are executing in your method, look through the code using IL while looking at a

source listing of the method. A paper listing helps at this point! If you need to find out the values of any variables, you can access them as offsets from the various stack frames found in stack crawl. Finding out the values of variables in an object can be done using InspectObject within the ToolKit debugger or by poking around in memory with Lisabug using the information presented in this document about the format of things. Further documentation on LisaBug commands can be found in the Workshop Manual.

TidBits of Info

The top two digits of any address divided by 2 gives you the Lisa Segment number. LDSN 1 = D6 & D7 (Seg 107), LDSN 2 = D8 (Seg 108)... F0 = the Clipboard segment (Seg 120). DA = the start of your document (Seg 109).

Process Address Space Layout



Hardware Segmentation: 24 bit address - 7 bit seg# and 17 bit offset