NATIONAL RADIO ASTRONOMY OBSERVATORY

Charlottesville, Virginia

March 6, 1975

VLA COMPUTER MEMORANDUM #119

IMPLEMENTATION CODING STANDARDS FOR THE ASYNCHRONOUS SOFTWARE DEVELOPMENT PROJECT

D. L. Ehnebuske

This memorandum presents the standards for coding in SAIL which have been adopted by the software development group for the VLA asynchronous software development project. This standard will be used until extensive experience with coding in SAIL has been gained by the group members. This document will be subject to revision then.

Currently, the coding standard discussion forms chapter 3 of volume four (the manuals part) of the asynchronous computer project book.

4.3 Asynchronous Implementation Coding Standards *****
Version 1

4.3.1 Introduction *****

4.3.1.1 SAIL and Notation *****

The main implementation language used by the asynchronous software development group is SAIL. SAIL is ALGOL-60 extended and was developed by the Stanford Artificial Intelligence Laboratory at Stanford Unviersity, Stanford, CA. 94305. The notation used for syntax in this chapter of the Project Book is taken from the SAIL User Manual by Kurt R. VanLehn, NTIS Document AD-765 353, to which the reader is referred.

The purpose of this chapter is to establish the coding standard to be used by the asynchronous software development group in coding SAIL. Like all of the Project Book, this chapter will evolve as the ideas in it are refined but unlike many parts of the Book this one will do so only in fairly major steps in order that the coding remain of fixed style.

4.3.1.2 Justification for Having Standards *****

The question of whether the asynchronous software development group should have any coding standards at all is one that ought not be dismissed without some consideration. The major reasons for NOT having one are:

* All of the people in the programming group have been programming for sometime and have developed their own programming styles (idiosyncracies?) and the change to a uniform and somewhat foreign style is likely to slow the writing of the code -- we have a LOT of code to write and not much time to write it in.

* Restrictions in the logical layout of code will sometimes mean that coding will have to be written in the "Standard" way when there exist more computationally efficient ways to express the same thing.

* It's a pain.

These considerations must however be weighed against the potential gains of using a standard. The major items here are:

* The "Public Property" nature of the code for the project entailed by our decision to use the Chief-programmer-team scheme of Baker et al, means that we will all have to be able to read and understand each other's code. A uniform standard would certainly help in this area.

* The longevity of the project will require people, including the original authors, to successfully modify parts of the code. Standard ways of doing and writing things would surely be of value to them.

* Finally, we desire to produce a package that is a piece

of quality workmanship, functionally complete and aesthetically pleasing. Uniform coding practices is certainlly an aid to this end.

4.3.1.3 What this Chapter Contains and What It Doesn't *****

This chapter of the Project Book contains the standards for how ode is written -- its physical and logical layout. Physical layout is ow code appears on the page, where it is indented, where not, how names are decided upon, where commentary is to be placed and the like. Logical ayout is concerned with how the code is broken up into modules, what ind of execution flow control structures are used, how errors are handled etc.

The standards are not concerned with how the code will be aintained, modified, stored or made into libraries. It does not contain specifications for testing code, or for evaluating execution efficiency. t does not even consider whether the code will perform any function, uch less what was desired.

4.3.1.4 How the Standards Were Developed *****

These standards were developed by combining considerations in three areas. First, the already well developed coding practices of the rogramming-type group members. In this category many of the usages of arious members were culled to extract those that seem particularly clear, sometimes merging two styles. Second, discussions with individual roup members concerning what things they felt were important to a good oding standard were considered. Much of this input was in response to a preliminary version of this document. Third, a good deal of thinking about SAIL and about what other programming projects have felt were mportant to coding standards was stirred in.

The result is a blend that, it is hoped, will aid the development nd maintenance of the project's code while not disrupting our ndividual styles to too great a degree.

4.3.1.5 How the Standards Are Presented *****

For this report, the standards have been divided into three parts: The first section deals with the logical layout of programs. The second s concerned with the physical layout. The third is a reference sheet hich summarises the standard but presents no justification for the items listed. It is this third section to which we will refer during the ctual coding.

4.3.2 Logical Layout *****

4.3.2.1 Modularization *****

4.3.2.1.1 Introduction and the Concept of Strong and Weak Modules *****

The first question we must address is simply: what is a module? We adopt the answer proposed by Stevens et al. (Structured design, IBM SYSJ o. 2 1974 p. 116). "The term module is used to refer to a set of one or ore contiguous program statements having a name by which other parts of the system can invoke it and preferably having its own distinct set of ariable names".

The reason for defining such a thing is that it has been shown that programs which are divided into multiple modules (as opposed to a program consisting of one huge module) are easier to maintain, design, and understand. It has also been shown that there are ways of dividing a programming task into modules which maximize the advantages of modularization and ways, which when used, do very little.

Recently, there has been quite a debate in the literature over which modularization techniques are best and from this debate there have emerged several valuable concepts. The most important two appear to be module strength and length. Module strength is measured in terms called "Binding", "Coupling", and "Information Hiding". Length is measured in lines of code. A "good" modularization yields modules of optimum length, which hide a lot of information, are highly bound, and minimally coupled.

4.3.2.1.2 Information Hiding *****

A module is said to hide information if by its construction and documentation the users of that module need not know anything about HOW the module is implemented. Obviously, the users need to know what the module does and how to call it up. A common example of a module which hides information is a sine routine. In order to use a sine routine the user need only know what it does, what its name is and how to call it. How is the routine implemented? Does it use fixed point or floating point? Does it do a polynomial fit? The user doesn't know or care.

The advantage of routines which hide information from their users goes beyond those apparent to the user of the routines. It is also of use to the routine implementer. If it becomes desireable to change a routine (for instance to speed it up or change the method of attack for aesthetic reasons) this may be undertaken without fear of disrupting the routine's callers -- for they know nothing about the insides, which leads us to the concept of coupling.

4.3.2.1.3 Coupling *****

Coupling is a concept which deals specifically with the connections between modules. A connection is any reference made in one module to something defined in another module. Things typically referenced in another module include data items (passed parameters, global variables) and control items (entry points and returns from subfunctions). Coupling is measured in two ways. First, and most obviously, by the number of connections. Second by the quality of the connections. A connection is said to be strong if it is to a control item or to a data item global to both modules, weak if to a passed data item. The strength of a data connection is also affected by the way in which the writer of the calling routine views the contents of the passed data. If he thinks of it as telling the called routine what to do (a control item) the connection is stronger than it would be if it were just data (a data item).

Let us examine what it is that coupling measures. First, low coupling means that it should be easy to understand the functioning of a given module without too many references to other modules (if the principle of information hiding is adhered to as well one need only understand what the other modules do and not refer to their internal working at all). Second, if the system has errors in it the low coupling

eans that the number of paths along which errors can propagate is very imited, easing the task of bug extermination. In sum, coupling is a measure of inter-module complexity; the lower the coupling the simpler he inter-module relationships.

At this point let us consider the effect of global data areas (common or control blocks) on coupling. Every module sharing a global rea is invariably coupled (strongly) to every other through every data tem in that global data area. It doesn't matter whether the routine in question actually uses each data item or not since any routine sharing he global data area CAN change anything in the area it just might. In ystems with lots of modules and large global areas the coupling is enormous. Enormous coupling nearly always leads to the "Mad Bomber Syndrome" in which a seemingly healthy module is bombed by the Mad omber through the global data area, but who is it?

4.3.2.1.4 Binding *****

Coupling is a concept concerned with the connections between elements in different modules. Binding is concerned with the connections etween elements of the same module. While it is desirable to reduce oupling a case for quite the opposite can be made for binding. Binding can be thought of as a measure of the cohesiveness of a module.

Stevens, Meyers, and Constantine list six ways in which elements of a module may be bound:

* Coincidental
* Logical
* Temporal
* Communicational
* Sequential
* Functional

hese types of binding are in order from the weakest to the strongest.

Coincidental binding occurs when a module is just a hodge-podge of ode. Yes, there are coincidentally bound modules believe-it-or-not. An xample is a module that does nothing other than eliminate duplicate coding in other modules and is usually a sign of "shop worn" software. Logical binding occurs when the code in a module has some logical ohesiveness. Many "general purpose" routines are logically bound. For instance "disk-tty I/O" routines are probably logically bound. Temporal bound code is code that is not only logically bound but also related in ime. Good examples of temporally bound code come from the housekeeping" genre (initialization, termination & c.). Ideally temporally bound code could be executed all at once more or less. ommunicationally bound code is code that does a whole bunch of things o the same data. For instance, writing the U-V data in the data base and on an archive tape. Sequential binding as its name implies, comes about when a module does several things which, because of the nature of he problem, must be done one after the other. Examples include routines which read this and update that, which calculate X and output it to the TTY. The highest classification of binding is functional binding. In unctional binding all of the code is dedicated to performing one unction. Mathematical functions are the first examples of functional binding that come to mind but there are many more: read a line of text rom the specified terminal, Convert the text string to reverse polish,

Search for record X on device Y.

Further discussion of binding categories may be found in Stevens et al.

As one can see from the descriptions of binding categories, the point of the concept of binding is to provide a handle for estimating the cohesiveness of a module. The higher the category that a module may logically be assigned to the more highly bound is that module.

4.3.2.1.5 Length *****

The last parameter that we will consider which affects the way that conceptualized systems are broken into modules is that of length. A good case can be made for the idea that there is an optimum length, measured in lines of code, that modules ought to have. Estimates of this optimum have been made for the last several years and the concensus now lies at somewhere in the range 50-200 and there has been a gradual shift toward the lower end of this range.

What are the considerations involved in these estimates? They are primarily the answers to the questions: "How many lines of code can you write and still hold the whole thing clearly in mind?" and "How many lines of code can programmers write in one go without making a mess of it?" and "How many lines of code are there in modules of 'successful' software systems?" Answers to these questions usually fall in the range given above and as was pointed out have been falling of late (according to one source the answer given in the mid-50's was on the order of 1000). Are we getting stupider or are we learning what Dijkstra called "humble programming"?

4.3.2.1.6 Putting It All Together *****

What does an "ideal" modularization look like? The easy answer is that all modules composing the system should be functionally bound, minimally coupled to each other, hide information and be about 100 lines long. As far as practical guidelines go the only really useful thing that tells us is the 100 lines part. In order to make use of the other concepts in the actual coding effort it seems to make sense to try to abstract a few heuristics, some do's and don'ts. Obviously, these heuristics are no substitute for understanding the concepts but they should be directly applicable.

Question: What does the module do?

If the answer takes a lot of space just to get a good idea across the module is probably pretty poorly bound and ought to be rethought.

Question: What is the minimum I need to tell the module in order for it to work?

Question: What information does the module have access to?

If the answer to these questions is not the same something could probably be done to un-couple the module from the rest of the system.

Question: What information can the module mess up that it doesn't need to change?

If there is something that the module can write on that it doesn't need to, think about passing the information to the module by value instead of by name.

Question: Does the user send my module anything that tells me what to do?

If the answer is yes the module is probably either not functionally sound since it does more than one thing or it is coupled more closely than need be by the control item being passed in as data.

Question: What does the user need to know about the module in order to make effective, error free use of it?

If the answer to this question even hints at how the module works inside there's some information the module is not hiding. Ideally, the answer should be: what the module does, what its name is, what its parameters are, where its results come out, and (if applicable) what the error return flag values mean.

The last and hardest question is:

Question: Has the way I've organized the internal workings of my module made it essentially impossible to write "good" modules in the next level down (which is, of course, stubbed)?

The purpose of asking this question is to force us to look ahead to the next level of modularization to avoid some of the pitfalls before we actually get there. Naturally, the lower levels will be a bit fuzzier but much of the technique of modularization is easiest if the general outline is thought through ahead of time. It isn't very good practice to say "Oh we'll make all that into a module in the next level down," unless what that module is to do and what data it needs to do it to is clear already.

4.3.2.2 Control of Execution Flow *****

4.3.2.2.1 Recommended Structures *****

Dijkstra and others have considered the questions of what flow control structures are necessary for writing programs and which that aren't actually necessary are convenient. They have also asked which control structures are clear and easy to understand and which obscure and confusing. These considerations have led to the major part of the collection of techniques known as structured programming.

Fortunately, there is a set of control structures which meet the desired criteria. All of these structures are among those available in the SAIL language and it was primarily for this reason that the major implementation language used in the asynchronous software development project is SAIL.

The control structures found to be necessary for writing programs fall into two groups either of which is sufficient to write any program. Only one of these groups is used in structured programming because the other contains confusing (error prone) structures. The control structures minimally required for structured programming are generally called the If-then-else and the While constructs.

The While and If-then-else constructs while sufficient to program any flow control required are not the only constructs that are clear and useful. In particular, they provide no means for modularization. So we also will find frequent utility in what are generally known as the Call and Return constructs. There are two more constructs which, while they can always be written as a composite of While and If-then-else, occur so frequently in programming practice that they appear in many languages. These are the so called Incremental-while (For or Iterative-do) and the Case-of. The two constructs have, in addition to frequency of encounter, the additional merit of being clearer in their meaning than the corresponding translation into the more basic types.

Translating the common names for the above control structures into the ALGOL BNF definitions (used in SAIL) so that they may be cross referenced with the User Manual we have:

Table 4.1 BNF Names of "Good" Control Structures

Common name(s)		ALGOL BNF Name
If-then-else		<conditional_statement>
While		<while_statement>
Incremental-while	or	<for_statement>
For	or	
Iterative-do		
Case-of		<case_statement>
Call		<procedure_statement>
Return		<return_statement>

Unfortunately the correspondence between the commonly used terms and the SAIL syntax definition of the above constructs is not exact in every case. In particular, the <while_statement>, the <for_statement> and the <case_statement> have a wider range of optional forms than is usual.

The <while_statement> and the <for_statement> have the NEEDNEXT forms and the <case_statement> has a form without the <numbered_state_list>. None of these forms are included in the definition of their common counterparts.

4.3.2.2.2 Other Control Structures *****

In addition to the control structures discussed in the previous section, SAIL has a large number of other control structures available. The question which naturally arises is: what should our attitude toward their use be? The answer is that we should use only those that are in keeping with the ideas of structured programming. That is they should be clear in their meaning and not introduce a high rate of system bugs.

In this light, what can we say about the other control structures in SAIL? In order to eliminate any of them on the basis of the proposition that it should not lead to high error rates, one needs fairly extensive experience in using it. In the case of most of the ax candidates we don't have very much knowledge of what to expect. There is one exception however -- the <go_to_statement>. The <go_to_statement> has been attacked in so many studies that it seems fair to eliminate it on the basis of being highly prone to errors.

The only other justification for eliminating a structure is lack of larity. The obvious problem here is that clarity has a good deal to do with training and personality. Perhaps if we regularly use the remaining structures in SAIL we will become adept at understanding what they mean nd consider them all extremely clear. In fact, there are those whose tility and meaning are clear at first sight. Since the asynchronous software development group will be working on the programs for quite a hile and will no doubt have to train a considerable number of new embers during this time, the criterion for saying that a given control construct is clear should be that it be readily understood without ubstantial experience in SAIL coding.

Adopting the idea that a construct is clear if it can be easily understood by programming types without experience in SAILing seems to ead to the conclusion that the <done_statement> is reasonably clear but hat the <next_statement> is not. It also points to the conclusion that the NEEDNEXT forms of the <while_statement> and <for_statement> as well s the form of the <case_statement> which lacks the numbered_state_list> are lacking in clarity.

As a practical matter, there will no doubt arise instances during he coding of the data reduction system when the use of the features of SAIL that lack clarity could be of great value and of course we ought not close our eyes to them. The above discussion is set down as a guide o make our work easier not more heroic. When such a situation arises ne ought to ask, before using one of the less clear control structures, "Is there no easy way to use the clear constructs to do the same thing?"

4.3.2.3 Inter-module Error Handling *****

So far we have discussed two of the major considerations in the ogical layout of programs. The first is the criteria that should be used to divide large systems into managably small chunks called modules. The second is concerned with the structures that are used to control xecution flow, primarily within a module. In this last section we will iscuss the problem of the way in which modules and their callers should behave in cases where a called module discovers that the data passed to t by its caller precludes calculating a proper result. Such occurences re an unfortunate result of having to deal with the real world -- Murphy's Law or some derivative of it insures that we are in deep trouble if we don't think about what to do when.

When defining how we should handle errors, as is true of other aspects of implementation technique, there is a long term advantage to igh uniformity; it avoids our having to spend time trying to figure out hat we have done when, months later, we have to understand why things don't work. Inter-module error handling may be thought of as comprising wo distinct though interrelated parts: the responsibilities of the aller of modules and the responsibilities of the called module. Since all modules except the top and bottom level are both callers and callees the authors of modules generally will have to deal with both concepts at nce. Nonetheless, for the purposes of discussion we will deal with each part separately.

.3.2.3.1 Responsibilities of the Called Module to Its Caller *****

The main thing that a module should do for its callers is to carry ut its function in a repeatable manner -- post haste. If unable to do

s[illegible] it must tell its caller why it was unable to do so in terms that are sensible to the caller. In addition, it must not destroy any data that the caller gave it to work with unless it can complete its function w[illegible]thout error. The reason for the dictum against passing back half d[illegible]gested data on an error return (a very hard one for sure) is that, since the caller has no idea how the callee has been implemented (the m[illegible]dules are uncoupled), the caller can make no sense out of the data. H[illegible]nce it has no way to recover from the error. On the other hand if the callee has not changed the data and if, at the same time, the callee has told the caller what was wrong the caller may be able to fix the problem a[illegible]d try again.

Consider a concrete example: a module which takes as a parameter a f[illegible]oating point number and replaces it with the square root of the n[illegible]mber. Obviously, it is unable to find the square root of a negative number. If we calculate, by some time consuming process, a number and then require the square root of it we may call the root finding routine, i[illegible] the square root routine gets a negative number and then ruins it in the process of telling us that it is having trouble we've lost our number and must recalculate it (if we are able). In contrast, if the s[illegible]ot routine merely told us that we gave it a negative number, we might b[illegible] able to fix it up and try again. While this example is trivial, the same applies in a more forceful way to more complex routines.

Summing up then, a module must behave in the following ways:

A) It must carry out its function when ever possible.
B) It must not alter any data passed to it if it is not able to complete without error.
C) It must, on detecting an error condition, tell its caller what is wrong in a way that is meaningful to its caller.
D) It must not die horribly -- control must return to the caller.

4.3.2.3.2 Responsibilities of the Caller to Its Callee *****

The caller-callee interface is bidirectional and although the majority of the responsibility for its maintenance lies with the callee, the caller too has its share for there are things that the callee cannot p[illegible]rotect itself against.

The caller must know what the callee is supposed to do. The subordinate module can hardly be to blame if its caller expects it to do s[illegible]omething other than what it is designed (and documented) to do. Next, the caller must understand the subordinate's interface. The callee cannot (under most languages) protect itself very effectively against c[illegible]alls made through an incorrect interface. Lastly, the caller must interrogate and understand the meaning of any error flags that its subordinate may pass back.

The utility of the above responsibilities should be clear in light of the discussion about what a callee should do for its caller. They are b[illegible]asically just those that the callee has little or no control over.

4.3.3 Physical Layout of Programs *****

4.3.3.1 Overview *****

We have so far considered what standards should be applied to the logical layout of code, that is how control should flow within and between modules; what structures should be used to control the flow and how information is to be pushed around in a system of interacting modules. In this section we will consider the other aspect of writing code -- the physical layout of the code on the page(s).

The reasons for considering physical layout are closely parallel with those for logical layout (namely to reduce the rate at which errors are introduced while coding or modifying and to make the coded programs intelligible) but the emphasis of our considerations of physical coding standards will have a slightly different tone. Physical layout is mainly concerned with the readability of programs as they appear in a printed listing, not so much with the structure of the programs. All that a program is is already encoded in the statement of that program to the compiler. Physical layout is for people only. It is intended to be redundant information which leads a person reading the program to understand the logical structure of the program by looking at how the program is laid out on the page. The idea is to force the viewer to see the structure of the forest by planting the trees in a meaningful pattern.

In the subject of physical layout, since the information presented is totally redundant, there is a considerably wider range of successful methods for organization than for logical layout. Indeed, what programmers consider the essence of their programming styles seems more closely bound up in the physical layout of programs than in the logical layout. Thus any attempt at standardization is more likely to run afoul of the programmers involved when considering the physical side of the question than when considering its logical aspects. Nonetheless, in a large system such as the one we are building, the need for standardization of the presentation of code is very great. Also, since there is a large range of successful methods that could be used for standardization the choice of one over the others is largely a matter of taste. Here we have chosen one of the many possible. The choice of the overall method was made considering the various ways in which the programming types in the asynchronous software development group already tend to do things. The choice of the details was made ex cathedra.

4.3.3.2 Commentary *****

Perhaps the least controversial subject under the "Physical Layout" heading is commentary. Perhaps this is because all programmers believe that commentary is a good thing that they really ought to do more of. Be that as it may here is the standard.

At the start of a module there should be sufficient commentary, separated from the text of the program, to explain:

1) What version this program is e.g. a version number.
2) What the module does e.g. a functional description.
3) What the program's interface is. That is the commentary should state how to call the module, what its parameters are, what global data references are made and what kind of data is returned. If the routine has DCL files they should be mentioned here.
4) What the error indicators are and what they mean

in terms that someone with a functional knowledge of the routine only can understand,
5) Which sub-functions are called on in the module,
6) What macro files are required for compilation of this routine.

Next, separated from the above and from the text of the module below, should be an overview of how the module is implemented indicating what algorithms are used and where they may be found, what the overall structure of the flow in the program is, what the major variables of the program represent, any helpful hints about the implementation that one should be aware of and the like.

Following the garp above is the main text of the program. It is recommended that we name each of the main blocks of the module. If the block is a major section of the module there should be a short accompanying comment stating what the block is going to do. Something on the order of "loop around getting a new estimate on the square root until the error is known to be less than 0" is probably sufficient.

During "straight" code inside a block commentary should be attached only to code that is not clear in its own right. Avoid saying exactly what the code says. The reasoning being that comments should be stuck in as guides where the reader might get the wrong idea. A <conditional_statement> is a place where readers tend to go the wrong way and so is a good place to think about leaving a signpost.

4.3.3.3 Indentation *****

In addition to commentary there are two more devices which we may use to aid the reader in understanding how a module works. They are indentation and paging. In this section we will set down the standard for indentation and in the next (and last) we will consider paging.

Program modules roughly consist of commentary (which is ignored by the compiler) and code (which is processed by the compiler). Code is usually thought to be divided into statements, each statement being a part of independently digestable "chunk" of program. In many languages, ALGOL included, the code statements can be further divided into the categories of declaratory and executable statements. Declaratory statements tell the compiler about the symbols that are to be manipulated, while the executable statements tell the compiler how to carry out the desired manipulation. Taken together, the two types of statements form a complete description of the module interpretable both by man and machine. But while the machine pays attention to each and every symbol people in general do not. The purpose of indentation is to arrange the statements of the program on the page so that people will see what is important for them to see at each level of detail.

For people the most difficult part of understanding comes in the executable portion of the code and it is with the executable statements of a program that indentation is concerned.

Key to the understanding of when a series of statements should be indented further than those around it is the idea of nesting level. The first level of nesting consists of those statements in the program that are always executed. The next level consists of those statements that are executed conditionally on a single test. The third level of nesting

comprises all of the statements that are conditional on a test which is itself part of the second level nesting. And so on.

The idea is to indent all of the statements at a given nesting level the same amount, each level being indented further than those with lower level numbers. To fix the whole thing we assume that the first nesting level is not indented at all. While that is the basic idea, there are usually minor flourishes added to it as a sort of "fine tuning" to further improve readability. Since the flourishes are, to a considerable degree, a matter of taste and a rather small perterbation are that those that are presented here are incorporated without further justification.

The indentation from a given level to the next deeper is a a matter of taste. Four or five spaces is reasonable. What ever you choose, be consistent.

The first statement of a given nesting level appears on a new line indented to indicate its nesting level. A statement may not appear on a line that is indented to a depth incorrect for its nesting level.

A BEGIN - END block marks the boundaries of a nesting level. The code between the BEGIN and the END is at a level one deeper than it would have been if the BEGIN - END were not there. The BEGIN is not a part of the nesting level that it defines but the END is.

A <conditional_statement> marks the beginning of a nesting level. The <statement>'s that follow the THEN and the ELSE (if it appears) together with the THEN or ELSE constitute the whole of the nesting level. Exception: if the conditional is short enough to to fit on a line it may be so placed.

A <case_statement> defines a series of nesting levels of equal depth, one for each case of the variable. The numbers of the <numbered_state_list> are a part of the level each defines.

<for_statement>'s and <while_statement>'s both mark the start of a nesting level consisting of the <statement> which follows them.

These definitions are pretty horrible to read. For examples see the Appendix 'Standards Reference Sheets'.

4.3.3.4 Paging *****

The last subject we will consider is that of paging. Compared with the preceeding mess paging is a dream. The basic idea is that a single idea should not span pages of the listing but that there should not be a whole bunch of ideas on any single page without separating them from each other by some blank lines.

Despite the paper shortage we are not lacking in paper to print our listings on. In fact if through proper paging we can save having to recompile code by having it understood properly in the first place we may even save paper.

4.3.4 Appendix: Standards Reference Sheets *****

4.3.4.1 Modularization Standards *****

A module should have the following characteristics:

1) It should be highly bound. That is it should do one and only one function and it should do all of that function. The main question to ask is "What does this proposed module do?" The answer should be short. If the answer MUST be long and complicated something is wrong.

2) It should be loosely coupled to the rest of the system. That is it should have available to it exactly that information required by it and no more. If it does not need to change a piece of information it should not be able to change that information. This point is against common or external data shared by routines. It is against passing information intended to control a subroutine.

3) It should be about 50 - 200 lines in length, tending to the shorter end of the range.

4) It should "hide information" from its user. That is the user should not have to know anything about how the module works inside to reliably use it.

Consequences of the above:

1) Parameters should be passed to a module by its caller in preference to using common/external data whenever possible.

2) Parameters should be passed by VALUE not by REFERENCE.

3) Functions should be used in preference to subroutines to pass back single values.

4.3.4.2 Standard Control Structures *****

The following control structures in SAIL are the only reccomended ones:

1) <conditional_statement> -- all forms

2) <for_statement> -- except NEEDNEXT FOR

3) <while_statement> -- except NEEDNEXT WHILE

4) <case_statement> -- <numbered_state_list> form preferred

5) <procedure_statement> -- all forms

6) <return_statement> -- all forms

7) <done_statement> -- form without <block_name> preferred

4.3.4.3 Standards for Inter-module Error Handling *****

Modules are recommended to follow the following conventions

or error handling:

As Caller

1) Understand what the module you are going to call is supposed to do.

2) Understand the callee module's interface.

3) When the module returns, check the error indicator.

As Callee

1) No matter what DO NOT blow up; control MUST return to the caller.

2) Until it is clear that you can complete without error DO NOT change anything that is passed to you except the error indicator(s).

3) The possible settings of the error indicators should be meaningful to the user of a black box.

4.3.4.4 Commentary Standards *****

At the head of a module there are two sections of commentary equired. They should be separated from each other and from the text of the program. If of substantial length, the separation may be by a page.

The two sections of commentary at the head of the program module contain:

First section

Information describing the module as a black box.

amely: 1) The version number of the module.
2) A functional description of the module.

3) The interface to the module. Parameters passed, their types, parameters modified, what .DCL files are to be used in invoking the module etc.

4) What the possible settings of the error indicators mean.

5) What macro, .DCL, etc. files are required for compilation of this module.

6) What routines are directly invoked by this module.

Second section

Information describing the implementation of the module.

Namely: 1) An overview of the logic of the module including references to published algorithms, project book

definitions etc.

2) A description of what the major variables of the module represent.

3) An exposition of anything else that you feel likely to help someone trying to understand the module.

Major blocks should be named. There should be a short comment indicating what the block is to do.

During "straight" code commentary should be used sparingly. The presence of commentary should alert the reader that the author thinks that here is a place that it is easy to misunderstand. Do not drown the reader in commentary or he will stop reading it and miss something important that you have to say.

<conditional_statement>'s have a higher tendency to be misunderstood than most others (in general).

4,3,4,5 Indentation Standards *****

Indentation from one level to the next deeper level is a matter of taste. When you have chosen be consistent. Four or five spaces is a reasonable number.

A statement may not appear on a line indented to indicate a level incorrect for that statement. Note that this restriction does not preclude putting more than one statement on a line.

The major "definer of levels" is the BEGIN - END block. The appearence of a BEGIN - END block makes all of the code after (but not including) the BEGIN through the END one level deeper than it would otherwise have been.

```
begin "grutley"
      blah blah;
      blah blah; blah;
      nange;
      thrunge;
      end "grutley";
```

Note that there is no distinction made between BEGIN - END blocks that declare variables and those that do not; all BEGIN - END blocks make a new level.

A <conditional_statement> makes zero, one or two new new levels. The number of levels depends upon the complexity of the statement and upon whether or not the ELSE clause of the <conditional_statement> appears. If the ELSE appears, the conditional makes two levels of equal depth. If the conditional is short enough to fit on one line, it makes no levels. Otherwise it makes one. The level(s) consist of the <statement> construct which follows the THEN or ELSE and includes the THEN or ELSE.

```
if gigantic!boolean!expression
      then statement!of!the!then
      else statement!of!the!else;
```

```
if other!boolean
      then begin
            stmt1;
            stmt2;
            end
      else stmt3;

if expr then stmt1;

if expr2
      then stmt2
      else stmt3;
```

A <case_statement> defines a series of nesting levels of equal epth, one for each case of the variable. The bracketed numbers of the <numbered_state_list> form a part of the level that they head. The form f the case statement without the <numbered_state_list> is permissible ut not preferred.

```
case argument of begin
      [0] stmt1;
      [1] begin
            stmt2;
            stmt3;
      end;
      [2] stmt4;
end;
```

The <for_statement> and the <while_statement> behave, with respect to indentation, analogous to the <conditional_statement>, without an LSE clause. That is they define a new level consisting of the the statement> which follows them. The DO in them forms the first part of the new level just like the THEN or ELSE in the conditional.

```
while monio!booleanlexpression
      do begin
            stmt1;
            stmt2;
            stmt3;
      end;

while true
      do stmt;
blah; blah;

for i_1 step 1 until tenzillion
      do begin
            stmt1;
            stmt2;
      end;
```

4.3.4.6 Paging Standards *****

The idea behind paging is to separate distinct thoughts physically n the same page by blank lines but to try to keep a single idea all on one page.

Thus, the required commentary at the head of a module should be set ff from the text of the module and should itself be broken into its two distinct parts. If the commentary is long enough (like more than 1/2 page, say) do a page up after it. Following the declaratory statements f the main line, internal procedures are the most usual things to ppear next. Since each represents a complete thought set each off from the main line code and from its fellows.