SDS 940 Lectures

Butler W. Lampson

This is a transcript of the lectures that I gave to the FFL Systems Group at White, Weld & Co in 1966. The lectures were taped, transcribed by a secretary, and proofread by me at that time. A copy of the resulting manuscript was scanned by Bob Frankston (a member of the group) in 2004, except that pages 83-123 and 151-153 were missing. I ran this scan through an OCR program in June 2004. A different copy of the manuscript, but a complete one, was scanned by Steve Bellovin in December 2011 and the missing pages OCRed by me. The following text is the result of these two OCRs. It has been edited and proofread, but not with great care. I added the table of contents, and a few notes [enclosed in brackets].

The "working document" frequently referred to is the Time Sharing System Reference Manual, Project Genie Document R-21. The October 1968 version of this document is available online, and it appears to be pretty similar to the version that existed when these lectures were given. A later version produced by SDS is also online, and it is quite a bit different.

I am trying to track down the source code, or at least the listings. The lectures frequently refer to the details of this code.

Contents

To: FFL Systems Group Date: June 1, 1966 - July 1,1966

From: Butler W. Lampson Version: 1

Introduction

The, system comes in three major pieces; namely the monitor, the exec and the subsystems. The monitor is the guts of the system. It is mostly resident in core and it eventually takes care of all interactions between the programs and the outside world: all input-output, scheduling, swapping, handling the drum or disc, handling teletypes etc., etc., etc. The exec is mostly non-resident, i.e., swapped; it runs as a user program. Its function is to provide what you might call second level control. Namely, it takes care of implementing the command language by which the user controls the system; it takes care of file naming, which allows the user to give symbolic names to his files and reference them afterwards, and it provides a number of special services such as floating-point, input-output, string handling operations and one thing and another. This we will discuss perhaps in some detail. The third component of the system is the subsystems. A subsystem is just some language or convenient facility which has been programmed and inserted in the system in such a way that you can get at it by giving its name as a command to the executive. A subsystem is regarded as a completely independent program, not closely connected to any part of the system proper. It's basically just like a user program. The only difference between a subsystem and a user program is that the name of the subsystem is in an exec table together with information about the location of the subsystem and its starting address. If a user types in its name the exec will find it in the table, bring it into user core and start it up at the specified address. After that the user is talking to it just as though it was one of his own programs. The subsystems include the debugging system, the assembler, the editor, Fortran, Cal, Lisp, Snobol, etc., etc., etc. What we are going to be mostly concerned with is the monitor in this discussion, because the monitor is the most complicated and the most essential part of the system. The executive is much simpler, it interacts less with parts of itself, and it also is much easier to change since it runs as a user program. The errors made in changing the executive are likely to be less disastrous than errors made in changing the monitor.

That is the overall structure of the system. And as I say we're going to be talking mostly about the monitor. We won't be talking about subsystems at all except maybe to make a list of them.

Hardware

The next thing we want to do is discuss all the hardware changes to the 940, everything which is different from what it says in the 930 manual. These changes come more or less under the following headings:

modifications to the instructions

to the interrupt system

to the memory addressing and bussing

and finally, various new I/O devices

We'll take these more or less systematically in order. As far as instructions are concerned there are the following things. Two modes have been added to the standard 930. When the system is in *normal mode* it is operating like a standard 930 and there are no changes at all. By doing an EOM you can put it into *system mode*. In system mode it still works like a standard 930 except that some of the things that we'll discuss in the memory system are enabled as well as a couple of new instructions which we'll also discuss in a minute. You can get into user mode by doing a transfer with the sign bit of the instruction set. This way you get from system mode to user mode. In user mode there are large number of instructions that are called privileged. This means that in user mode they cannot be executed. If a privileged instruction is executed in user mode, it causes a *trap*. A trap is a forced transfer to a particular location in lower core, 40 or 42, something like that. There are four traps, each one with a unique location. At this location the system will put a transfer to a little subroutine that will do something about the trap. Namely, it will type out a little message "You made an error" or something like that. Or whatever you want it to do. Exactly what the system does do is something we will cover in more detail later on when we are discussing the relevant parts of the monitor. What the hardware does when it sees a privileged instruction trying to be executed in user mode is to force a transfer into this fixed location. The privileged instructions are all input/output instructions and anything that might halt the machine (which includes not only the explicit halt instructions but a whole lot of other instructions that are illegal for one reason or another). There are a lot of undefined opcodes in the 930.

Then, there are a couple of new instructions. There's an instruction for clearing interrupts called BRI, which is exactly equivalent to branch indirect except for its effect on the interrupt system. In the normal 930 any branch indirect clears an interrupt, in fact, clears one interrupt level, so that if you have three interrupts hanging and you are sitting in the top priority one you do a BRU indirect which clears the top priority interrupt and also leaves you sitting back in the routine which is processing the interrupt of next lower priority. See figure 1. Because of the fact that the system sometimes wants to execute BRU indirect without clearing an interrupt, we've added a special instruction (BRI); which specifically serves to clear an interrupt and otherwise acts like a BRU indirect. BRU indirect then no longer clears interrupts. This is effective only in system mode, of course. Another change: the EAX instruction (effective address to index) has been modified. This instruction in the 930 computes its own effective address and puts it in the bottom 14 bits of the X register. Since EAX may be indexed or indirectly addressed, and in the latter case the indirect word may again be indexed or indirectly addressed, the computation involved may be non-trivial.

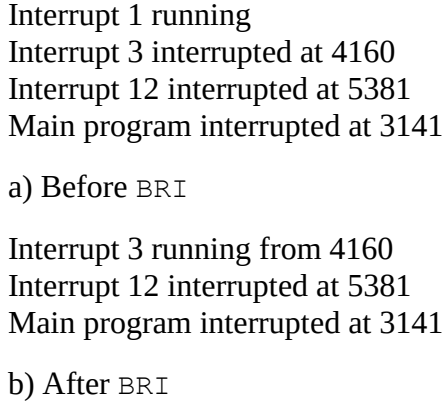

Fig. 1: Action of BRI

Example: if X contains 1212 and memory is

```
1000     LDA* 1400, 2
2612     BRU  (12 + 12 + 12 = 1224)
```

then EAX* 1000 will put 01224 in the bottom 14 bits of X. This is a very important instruction.

It has been slightly modified in *system mode* so that in addition to setting the address part of the index register to the effective address, it will set the sign bit of the index register zero or one, depending on whether the effective address was relabeled, was mapped or not. The mapping is simply determined by whether the sign bit is turned on or not, in system mode. If the sign bit of the instruction is turned on mapping starts right away. If it's not, the first level will be done without mapping but if you go indirect you may again have the possibility that the sign bit is on, in which case mapping will start at that point. So, again, it's not too easy to tell whether the final address is mapped or not. Consequently it's very nice to be able to find out without explicitly running down the chain looking at all those sign bits, whether the address is mapped or not. And in order to accommodate this, in system mode EAX changes the top bit of the index register to zero or one depending on whether the final effective address is or is not mapped. The reason for this is that the way you normally use EAX is that a link has been put into location 0, a subroutine link essentially, and what you do is you say EAX* 0. And what this does is it goes out to 0 and in 0 there will be a link that was left by a POP. (See Fig. 2) Everyone knows about POPs?

Relabeled	2216	POP ABC,2
Absolute	0	ZRO* ABC,4
	in POP routine	EAX* 0
	old contents of X	200
	.resulting in X	ABC + 200,4

Fig, 2: Action of EAX in system mode

What the EAX is going to do is to go back and pick up the effective address of this POP, namely ABC, 2, with the intention of using that as an argument. You do the EAX first, and that gives you the effective address in the X register and then in order to pick up the argument word all you have to do is say LDA 0, 2. In order to pick up the second word of the argument you say LDA 1, 2 and so on. The difficulty is that if you are in system mode and the address was mapped, then you must not say 0, 2 but you must turn a sign bit on, and say 0, 6; otherwise you won't get any mapping. So you have to know whether you have mapping or not so you can decide whether to say 0, 2 or 0, 6. And this is the reason for this little gimmick.

In some cases you don't have to worry about this problem, in particular, if the only thing a POP needs is the first word addressed: ABC, 2 itself and not any subsequent words. Suppose you want to get that word into the A register. Just say LDA* 0, and that sends you indirect to 0, indirect to 2216 and off to ABC, 2 and gets it to you right away. The mapping problem is taken care of automatically, because when you go to 0 mapping is turned on because the sign bit is set.

All right, so if you want to use a single word then you can get it automatically with LDA indirect, because the mapping will be turned on automatically by the convention that it gets turned on when you go indirect through a word that has a sign bit set. But if you need, say, ABC+1, 2 (for instance if you were doing a floating point POP and were addressing a floating point number which takes up two words), then you want to get the effective address in the X register so that you can do LDA 0,2 and LDA 1,2 and get both words. But in that case you need to know about the mapping. The only way you can avoid worrying about the mapping is to just assume that the POP came out of user mode. Then you always turn the sign bit on in the instruction. Or you assume it came out of system mode and you never turn it on, But if you want to have a POP that works both ways then you need this gimmick. The chances are good that you will be adding some SYSPOPs. Basically the reason for this is that, for instance, when you get some-

thing all set up and want to make a disk reference, the way you will probably do this is by executing a SYSPOP. Why? In order to make that disk reference you have to look up in a table, which is going to be permanently resident in core. It probably will be nice not to have that table in the user's own relabeling. The only time you need to reference that table is in this one case when you are actually ready to make a disk reference. So this means that probably the way you want to reference that table is by having a SYSPOP, so that when you are all set to reference it you put the relevant things in the A and B registers or whatever and execute the SYSPOP. The SYSPOP then, running in system mode, can reference the table by changing the relabeling or in some other way.

Now, what else is there about instructions? Oh yes, overflow instructions. The two basic input-output instructions on the 930 are called EOM (which stands for Energize Output M. Why M, I don't know) and SKS. What EOM does is essentially to send signals to all the I/O devices and other miscellaneous hardware. Somebody is supposed to recognize the signal and say "That's for me," and turn itself on. SKS stands for Skip on Signal. It looks at lines coming in from the outside world and the address field selects one of these lines and skips or doesn't skip depending on whether the line is on or off, up or down. Normally, of course, these instructions are privileged. However, in the normal 930 these instructions have one non-privileged use. Namely, they are used to turn off and to test the overflow indicator. It is necessary for the user to be able to do this. So, originally what we did is we had a whole lot of complicated logic to say EOM is privileged but if it is an overflow EOM it's all right, it's not privileged. Well, this was very messy. So in order to straighten out the situation we added a new instruction, opcode 22. I guess it doesn't have a mnemonic of its own. It can have three or four different addresses corresponding to the three or four different overflow instructions that used to be handled by EOM and SKS. This instruction I guess even works in normal mode, although there is no particular reason to use it in normal mode. As a result of this change we have been able to make all EOMs and SKSs privileged.

The next thing is the interrupt system. The normal 930 has a priority interrupt scheme and perhaps it would be helpful it I describe it very briefly since most people will never get to the section of the manual in which the priority interrupt scheme is described. Priorities work like this. You buy them in bunches of two, and you can buy lots, so there are lots of priority interrupts. They are numbered: 1, 2, 3 up to 872 which is the most you can get. The idea is this. Each priority interrupt has a line from the outside world attached to it. And each priority interrupt also has an address in memory, starting at 201, so the first one has address 201 and the second one 202, etc. Signals can come in on these lines. And if a signal comes in—suppose just one signal comes in, on line 4—the effect is that at the end of the execution of the current instruction, whatever it is, the normal flow of execution will be stopped and the computer will instead be forced to execute the instruction in location 204 since this is the fourth interrupt. This instruction will normally be a BRM which will store the location counter and transfer off to some subroutine which is designed to process the interrupt. When the interrupt is processed you return from the subroutine with the BRI instruction that I discussed earlier, which will clear the interrupt. This much is common to all interrupt schemes. The priority thing comes in the following way. Suppose that while I'm processing interrupt number 4 a signal comes in on interrupt number 2.

Now 2 is regarded as being of higher priority than 4, and what this means is that I want to execute 2 right away regardless of whether I'm executing 4 or not. So 2, being higher priority than 4, is able to interrupt 4, so that even though I am processing an interrupt on level 4 I am forced to execute the BRM at 202. And at this point, when I am in the routine specified by the BRM in 202, I have two interrupts going. Interrupt 4, which is not executing, just hangs. When I exit from 2 with a BRI I go back and start executing 4 again.

Q Does it start all over again?

A No. It starts wherever it was. Say you execute the instruction at 202, which may be BRM 1202. What happens is that it stores the location counter in 1202 and goes to 1202+1, where there will be instructions that save the central registers: STA, STB, STX. After the routine at 1202 has run, it will restore everything with LDA LDB LDX, and then it's going to say okay now I'm finished, so how do I exit? I exit by saying BRI 1202. Now BRI is exactly like BRU indirect. Remember? So what it does is to transfer off to the location specified in 1202, but it is exactly what was in the program counter when we executed the BRM 1202. It was exactly the location of the instruction which was about to be executed when the interrupt came along and said, "No stop, don't do that!" So the effect is that I go back and continue executing whatever I was doing when the interrupt came along without any disturbance at all, provided, of course, the interrupt routine saves the central registers, It has to do that. The only thing the BRI does is to clear the highest priority interrupt. So I come back now from interrupt level 2 to interrupt level 4, which is restarted automatically. Eventually 4 gets finished, and then its BRI sends me back to whatever main non-interrupt program was executing.

Q Did you say that the central registers are not preserved?

A Not automatically, but every interrupt routine of course should preserve them, and if it doesn't horrible things will happen.

Q It is never interrupted during the storing?

A It might be, it doesn't matter. Suppose it's interrupted after, doing STA. All right, that means some other routine will go off to do its operation, which presumably will preserve the central registers since it also has storing and loading instructions in it. When it comes back this one will just proceed with its storing and there's no disturbance. Okay? An interrupt routine is supposed to be completely invisible to whatever it is interrupting, and if it's not that means it wasn't written right.

Now suppose on the other hand that I was in interrupt level 2 and interrupt 4 came along. Since 4 is lower priority than 2 I do not want to stop executing 2 when 4 arrives. What happens is that 4 just sits there and hangs. It's an interrupt which is unfulfilled,—until 2 is finished. As soon as 2 is finished, 4, which has been hanging there, immediately gets in, and starts to execute.

Q How many things can we have hanging?

A As many as you want. You can have them all hanging, waiting for number 1, okay? Naturally, because of this priority scheme, you want to make a great effort to keep the high level interrupts short because they hang all the lower ones while they are going. It may be worthwhile to point out that above these priority interrupts there are several built-in interrupts in the computer, the ones relating to the TMCCs (2 for each), the clock interrupt and power fail. They are really in the priority scheme in the same way, differing only in their numbering and the fact that they come with specific devices. You can think of them as just being higher priority interrupts above 200.

Q The higher priority ones are in the middle there some place?

A Power fail is highest. Then comes the clock. Its location is 75. Then comes the W-buffer at 31 and 33, then other TMCCs and DACCs, then the regular priority interrupts.

Q There's another priority in the channel?

A That's right, each TMCC has two interrupts. The channels are higher than any of these 200s although you can get this rewired. You can make it go any way you want.

This is what you buy on an ordinary 930. There is one other little goodie you can get which allows you to say for each one of these interrupts whether you want anyone to pay any attention when the interrupt arrives or not. If you are paying attention the interrupts are said to be armed, and if you are not pay-

ing attention they are said to be disarmed. We use that feature in a couple of places. In addition, the whole interrupt system can be enabled or disabled. If it is disabled, any interrupt that comes in just sits there and waits. Disabled is sort of equivalent to saying, "Some really high priority thing has come along that is more important than anybody else." When you reenable the interrupts, any that are waiting immediately start trying to get in and the highest priority one does.

Q Is there any possibility of losing an interrupt?

A No.

Q Suppose I made a `201` and is it possible `204` could get in twice?

A Yes, that is possible. If you have an interrupt coming up frequently, yes. Definitely.

Q Is this a normal occurrence?

A Never. That's an absolute disaster. You arrange that device which generates interrupts so that it doesn't generate them at such a high frequency. You can build a device so that it never generates an interrupt more often than every 5 milliseconds.

Q What if you disable the interrupts for 5 milliseconds?

A Then you're a fool, and you deserve to lose! You may disable for a short period of time because if an interrupt comes in during this time you may get badly confused. Particular reasons for disabling, which will show up when you look at the system itself, are that you may be operating (outside of any interrupt routine, in the main program) and you may want to fiddle a table that also gets fiddled by an interrupt routine. If you get interrupted in the middle of the fiddling process the interrupt routine will get very confused. So what you do is to turn interrupts off for the fiddling time. It must not be very long or you are likely to get yourself into serious trouble.

Q This means that a routine can't cause the same interrupt it is processing?

A External devices cause interrupts. The program is not an external device. A program is a program. It cannot generate an interrupt.

Q But an interrupt routine may have I/O instructions in it?

A It depends on what the interrupt is. It may, yes. But the routine can't cause an interrupt. The only thing that the handling routine can do is fire the device up again. If the device is constructed so that immediately when it's fired up it generates the same interrupt again, probably there will be a mess. If you have just fired the device up, there is no point in its causing an interrupt. You already know that you fired up. The significance of an interrupt is to tell you something you didn't know.

Q In effect, the interrupt disarms itself?

A No it doesn't disarm itself. I don't understand why you think that it should.

Q You said if something is mechanically wrong it could be hung up in that interrupt for hours. Why?

A Because the interrupt keeps coming through.

Q Why would the interrupt keep coming through?

A Because it's being generated each time. Each time you leave the handling routine you get the same interrupt again.

Q Why do you get an interrupt? What generates that interrupt?

A If the device is malfunctioning so that it has its interrupt line all the time, then you can get into trouble. You can get into trouble if you take a soldering iron and put it on top of the B register! If the hardware is not working then you're in trouble.

Q How is that handled?

A When the hardware is not working you call a service man to fix it.

Q How do you find out what's wrong?

A You have a diagnostic routine to run around and check everything. And this is, by the way, one of the significant problems of this system, because the diagnostic routines are not adequate.

We have modified the interrupt system slightly, in that it used to possible to shut out interrupts when you were sitting in certain kinds of loop, particularly if you were sitting in a loop of the form BRU* or BRX*. Also, something of the following form:

```
A LDA* B
B ZRO* A
```

could shut out interrupts. If the LDA is executed, then in following the indirection we go from A to B, from B back to A, from A to B and it's an infinite loop. This is obviously an error, but any user can w rite this.

All three of these conditions in the 930 will tie up the interrupt system. All these things go so fast that they never allow the machine to get into a state in which an interrupt can occur. And this is obviously disastrous, since it means that the user, by making an indirect loop, which is not at all impossible, can hang the whole system forever. Once he starts there is no way to get out of it because there is no way to get an interrupt in to stop it. So that's very, very bad and the way we correct this situation is by jiggling the insides of the machine so that there is no sequence of two instructions which don't allow an interrupt in the middle, with one exception, which is EOM. Between every two instruction executions in the 940 it is possible to accept an interrupt. This means that the longest amount of time you could have to wait before an interrupt gets in is the longest amount of time for the longest executing instruction, which happens to be divide, which is 10 cycles, or 17.5 microseconds. So much for that.

Q Is time-slicing handled with interrupts?

A As we go through the system properly we will see all these interrupts and how they are handled one at a time, but it might be helpful to make a list, of the important ones anyway: there's the clock, the W-buffer (which has two interrupts), the teletype interface, power fail, and the drum. There is also a link to a subsidiary machine, the PDP-5. This is driven through the DMC and if you drive displays through the DMC you will get an interrupt similar to this one. That's all.

Q What about tapes, printer, etc.?

A They all go through the W-buffer.

Besides interrupts there are also traps. Traps are caused by illegal (privileged) instructions, memory violations (which we will discuss below), read only violations and something called the user mode trap. Those are the four traps. We will discuss them all in detail later on. Overflow is not a trap. Overflow just occurs and sets an indicator which you have to look at. That's an obsolete way of handling overflow, but that's the way it's handled in the 930, the 930 being more or less an obsolete machine.

Memory hardware

We have quite a lot of things associated with the memory and will start with the things in the bussing hardware, which are less important. A 940 system always has at least two memory boxes. A memory box is some number of words of memory (4000, 8000, or 16000). Its significant property is that it operates independently of all other memory boxes. This means you can have memory references going on in two separate memory boxes at the same time. The CPU can't have memory references going in two separate boxes because it needs to finish the first reference before it can start the second one. But you could have one box being run by the CPU and one run by the channel, for instance. For convenience, we'll talk about a 32K machine which has two 16K memory boxes. Normally the way things are arranged is with addresses zero to 37777 in one box and 40000 to 77777 in the other.

We have modified this arrangement in the following way: all even addresses are in one box and all odd addresses are in the other box. This is called interleaving. The reason for this is the following: when you run I/O at very high speed, and particularly when you run our core speed drum, it transfers information in blocks and this means that if the drum starts to run out of this box, starts at location 4000 and wants to transfer from 4000 to 8000 say, it will take every memory cycle if it's running at core speed, which means that nobody else will be able to get a cycle if you're using the non-interleaved scheme.

When the drum starts to run out of one box it will lock the CPU out of this box because it has priority and has to get every cycle. It locks the CPU out of this box for 2000 memory cycles while it does the transfer. And that's very bad, because it means the system can't run at all, no interrupts can be processed, nothing. The CPU is completely hung. If you have interleaving, on the other hand, the drum will take one reference out of the first box, then one out of the second, then one out of the first, etc. , which means that the CPU can run at 50% of its normal speed at least. Probably it will be better than that. Interleaving is very simple to implement. Suppose we have a short address, six bits, a 64 word memory (See Fig. 3). You have address lines coming to the memory, the low order bit on the right. As the address lines come out of the computer they go into the memory. The usual way to arrange it is that they go straight into the memory. All you have to do to get interleaving is to interchange the low order and high order lines. This means that it will be even or odd that decides what the first bit is going to be on the lines coming into the memory. So where before it was the high order bit which decided which module you get, now it's the low order bit, which decides which module you are going to get. So it's completely trivial.

The second change in the memory itself is a little more subtle. Suppose we have the two modules and two devices going. We have a CPU, which is generating memory references in some random way to the two modules, and we also have the channels, say the RAD channel, which is also generating memory references. Now the RAD is a synchronous device and every four memory cycles it generates a word while it's running. If the word isn't disposed of by the time the next word comes along, a disaster has occurred, because the word is lost. The RAD has a buffer which sits there holding the word, and somewhere else the next word is being assembled. When the next word is finished being assembled it comes into the buffer, because there is no room for it anywhere else, and this means the first word has to be emptied out of the buffer by the time the second word comes along. The easiest way to handle this problem is to say that the RAD has to have priority sometimes, so we'll give it top priority all the time. When it asks for this memory module we'll give it this memory module and during the first cycle I guarantee the word will be out of the RAD buffer as fast as possible. Now this has the following disadvantage: if the CPU is asking for this box at the same time as the RAD, the CPU will get hung for one cycle while the RAD is looking at the box. This is interference. In this particular case, with two modules and the RAD getting one cycle out of four, there will be interference 1/8th of the time, so you get 12% degrada-

tion in CPU performance, simply due to the fact that 12% of the time when the CPU wants a memory cycle it has to wait

However, this analysis ignores an important point, namely, that when the RAD generates a memory request, it can wait for three cycles because another word is not going to come along for four memory cycles. So when it says, "I want memory," what it really means is, "I've got to have one of the next four cycles, but I don't care which one it is, just as long as I have one of those four cycles." Each device using the memory sends it address lines and data lines, and a priority line or request line which says, "I want the memory". Built in somewhere is a gadget that settles ties on the basis of built-in priorities for the various lines. What you do is to make the priority line for the RAD a little bit more complicated: you use two lines instead of one. This means that the RAD can say either, "I want the memory with high priority," or, "I want it with low priority". High priority means mine is higher than anybody else and low priority means mine is lower than lots of people, anyway, lower than the CPU. So what the RAD does is to ask for the memory with low priority on the first three cycles. If the CPU wants it the RAD has to wait. On the fourth cycle it asks for high priority, because there's another word coming along pretty soon. If the CPU still wants the same module after three cycles, it will be hung on the fourth. In fact, the CPU almost never references the same module four times, because instructions, for one thing, go in alternating modules because of the interleaving. Furthermore, if you take random instructions and random addresses, it is 50-50 that the address will be in a different module than the instruction. So the chance of the CPU wanting the same module four times in a row is less than 1%. This means that you effectively reduce the RADs interference with the CPU from 12% to zero at the cost of putting in this very simple priority device.

Q How does the RAD buffering really work?

A There's a buffer which holds the assembled word and this is the buffer that is connected directly with the memory. Then there is another buffer in which the RAD is assembling a word, six bits at a time, as it comes off the disk. In this buffer there are control signals telling where to put the next six bits, and when the fourth set of six bits is coming along that is the critical time. Probably, you can take one of those control signals right off and use it for the priority lines. And this a $500 modification. Mel invented this, he told Schmidt about it, and Schmidt asked SDS about it. They said they didn't know anything about it and Schmidt went through the roof. So he made a tremendous row and got SDS to put it in.

So much for the modifications to the memory. The other thing about the memory is the relabeling, and that's complicated. We will discuss it in some detail. Built into the system are 8 six bit registers which we usually draw like this. (see fig. 4) These are 48 flip flops. Each one of these six bit registers, which we number zero to seven, is divided up into a five bit field and a one bit field, called A and R. When the computer addresses memory, it does so by generating a 14 bit address field. It goes through all kinds of gyrations: direct and indirect addressing, indexing, anything it wants, to generate these 14 bits. When it's all finished it has the 14 bits which are the effective address and it ships them out to the memory

In between is the relabeling box, or the map. These terms are synonymous. The function of the map to make some kind of transformation on these 14 address lines and ship out some new address lines to the memory. In fact, there are 14 lines coming in here, 14 bits of address, enough for 16K. We have 65K of real core, so 16 lines have to come out of the map. Now the question is, how does this map work? It takes these 14 bits that come in and does some transformation and ships out 16 bits. It could be any transformation. You could run the bits through a random number generator and you could produce 16 new bits but that probably wouldn't be good. You could add 50 to the address. That wouldn't be too good either because you could have done that in the computer already.

Q Is the 930 memory extension register involved?

A Memory extension has nothing to do with this scheme and is never used. What this box does is the following: it takes a 14 bit address and divides it up into eleven bits and three. The latter is called the page number P and the former the word address W. Pages are called blocks too. So eleven bits are shipped off to the side for the moment and we don't worry about them. The top three bits of P give us some number from zero to 7. We use this number to select one of the 8 relabeling registers over here. We take the A field out of the appropriate register. Now we are going to make a new address which we are going to get by keeping the W field and gluing on the front of it A, which is the five bit field we get from the relabeling register. The situation we have is a transformation yielding 11 bits of W field and 5 bits of A field, or a 16 bit address. We got these 5 bits of A field by using the 3 bit P field to select one of 8 relabeling registers. So we now have a 16 bit address, and this is the address we feed to the memory.

The significance of this is that there are 8 pages in the memory that the user can address, and we can make an arbitrary transformation, we can say that each individual one of these 8 pages is going to some page of the physical memory. In physical memory there are 32 pages of 2K each. In addressable memory, virtual memory we call it, there are 8 pages. These 8 registers define the relationship between the 8 pages of virtual memory and some combination of pages of real memory.

Now, there are a couple of little points. If the A field is zero and the R field is 1, then that is regarded as an error. The hardware generates a memory trap. That's one of the four traps explained above. The second kind of trap was for memory violations. And the way you get a memory violation trap is to go through a relabeling register with 40 in it. That's an error.

The R field is used in the following way. When the CPU ships out the address, it also ships a signal which says, "This is a load", or, "This is a store". And if it is a store and the R bit is on, that causes a read only trap. Thus, the R bit is turned on in each page which you want to protect from being changed. Whenever a relabeled memory reference for a store is generated and the R bit in the corresponding relabeling register is on, that causes a read only trap, which again goes off to a special place in the system. So this means that when you set up the relabeling, you can define for each virtual page whether it is protected from storing.

Note that real page zero cannot be made read-only, since 40 is interpreted as an error at all times. This particular convention is at variance with all the rest of the rules. So the number 40 is interpreted in a special way. Zero is interpreted in the normal way, as referring to real page zero.

For the pages of virtual memory that are unassigned the monitor will put 40 in the corresponding relabeling register. There is no reason why you should have 16,000 words of memory assigned to you if your program doesn't use 16,000 words. It's a big waste of real memory. You should have only as much memory assigned to you as you need. So what the system does is to assign as much memory as it thinks you need (exactly what this is will be discussed later). It will put 40 in all the other relabeling registers.

If the user addresses a word in a page which has 40 in its relabeling register, there will be a trap generated by the hardware. The trap goes off to a specific routine in the system, and this routine will try to figure out what to do, whether that really is an error or whether it should assign you some more memory.

Q If I put a zero in one of the relabeling registers is or is not that an error?

A No, that means that virtual page will correspond to real page zero.

Q I want to protect users from violating the monitor?

A Then I just never put zero in the user's relabeling.

Q Is the user free to change it on his own?

A He can change his pseudo relabeling, and when he does we check to see that he is doing it legally. When we discuss pseudo-relabeling, which we will do in great detail, you will see exactly what the restrictions are and how he can change this relabeling and what the relationship is between pseudo-relabeling and real relabeling. It is completely impossible for the user to ever get real block zero in his real relabeling.

Q How does he alter the monitor?

A Only by making a very special system call which causes very special things to happen.

A How is relabeling set up?

A This machine has two instructions called `POT` and `PIN` (parallel output and parallel input). If you say `POT OUTLOC` it finds location `OUTLOC` and takes the 24 bits from there and puts them out on 24 lines and that's all it does. What happens to the 24 lines is someone else's responsibility. `PIN` likewise looks at the 24 lines and sticks them in the memory word addressed. What happens to these 24 lines after a `POT` is determined by what you do immediately before the `POT`, and what you normally do immediately before the `POT` is an `EOM`. So before the `POT` you specify the destination with an `EOM`. The `EOM` will have an address field which will be interpreted by the hardware to set some flip flops, which will decide what happens to the information provided by the `POT`.

In particular there are two `EOM`s for setting relabeling register one and relabeling register two (relabeling register one being the first four of the six bit relabeling registers and relabeling register two being the second four). The way you set the relabeling registers is to say `EOM 20400B`; `POT RR1`; `EOM 21000B`; `POT RR2`, and the contents of `RR1` and `RR2` will be used to set the relabeling. There is no way of reading the relabeling registers; you are just supposed to remember what you put in them.

Q You said if you put `40` in a relabeling register you got a memory violation when you address it. How did you get that?

A When you do this `POT`, you are outputting a certain memory word which you address. So you put into that memory word whatever you want to have in the relabeling registers.

Q This is done by the system.

A Yes. The system sometimes sets up the relabeling with `40` in some of the relabeling registers.

Q Why?

A Because it doesn't want you to have 16K of memory. If you have a 4K program it's ridiculous to assign you 16K of memory because this takes up 16K of valuable cores.

Q Just clear up one point. You never give the `EOM` and `POT` instructions to set the relabeling registers?

A Who's "you"?

Q The user.

A The user can't give an `EOM`—it's privileged. Only the system sets the hardware relabeling.

The next question is we have all this glorious machinery, but when is it used? The answer is this: all memory references generated by a program running in user mode are relabeled. Addresses generated by a program running in system mode are not relabeled. However, if the instruction has the sign bit set then the address normally will be relabeled. Furthermore, if the instruction goes indirect through a word that has the sign bit set addressing will become relabeled from that point on. In the assembly language set-

ting of the sign bit is indicated by a `,4`: `LDA 0,4`. The thing following the comma is the number which tells what is supposed to go in the top three bits of the instruction word. `0,2` means indexing, `0,4` relabeling (in system mode only). You could write `0,6`, which would be both. You can also set the third bit with `0,1`, but that is the `POP` bit and you normally don't set it this way, but with the opcode.

Q There was something about the sign bit indicating relocation in the 930 manual?

A The significance of that is that the standard 930 loader interprets that top bit as determining whether or not to relocate the address. It's strictly a software function. We do not use it that way.

This is the explanation of how you get relabeling in system mode. The sign bit of the instruction is set, or the sign bit of some word in the indirect chain is set and soon as you see the sign bit on you switch over to relabeled addressing. In the indirect chain you continue to relabel all the addresses after this point.

Q Why isn't it possible for the system, the assembler, to set that bit for you?

A The assembler doesn't know what you mean. If I am writing a system mode program, there is a definite difference between writing `ABC` and writing `ABC,4`. And the assembler has no way of figuring out which one I meant.

Q The assembler could do something like this: when the program exceeds the page then all references—

A That has nothing to do with it. The significance of the bit is whether the system wants to look in the user's memory. For instance, when a `SYSPOP` is executed and you come out to the part of the system that is executing the `SYSPOP`, you will have to look back into the user's memory for the address of the argument of the `SYSPOP`. This will be done by invoking relabeling. That's why we have this rule about indirection and sign bits which we discussed earlier when I was talking about `EAX`. When you do a `SYSPOP` the hardware sets location zero to contain the address of the `SYSPOP` and turns the indirect bit on. If the call came out of user mode it turns the sign bit on also. The same thing is true, by the way, of `BRM`. When `BRM` stores that link in system mode (for instance after an interrupt) it stores one or zero in the sign bit depending on whether it came out of user mode or not. So there is always this convention that an address which is relabeled has the sign bit on; a full word which contains a relabeled address has the sign bit on. This means that whenever I indirect through that word I automatically get relabeling activated without saying specifically that I want it. This is very nice. You don't have to think in the program about whether relabeling is being activated or not.

Q Do I have to set the sign bit in user mode?

A In user mode *everything* is *always* relabeled. No, it would be disastrous if the user addressed anything absolute.

Q What happens if I set the sign bit in user mode? Does it matter?

A No, it has no effect in a regular instruction. If the `POP` bit is set, then it becomes a `SYSPOP`. In user mode if you execute a `POP` with the sign bit on, that is the signal that says "This is a `SYSPOP`, go to system mode". Normally if you execute a `POP` in user mode everything is done in the normal way: you put the link in user location zero and you transfer to user location `100` plus the address of the `POP`. If you put the sign bit on, this is a signal that says `SYSPOP`, put the link in absolute location zero and transfer to absolute location `100` plus the address. In those 64 words the system will put 64 links, 64 addresses which branch off to the 64 routines which are handling the 64 `SYSPOP`s. From there on it will do whatever it sees fit.

So the sign bit has two functions. The first thing it's used for is to indicate relabeling in system mode. The second thing it's used for is to indicate a SYSPOP in user mode. The third thing this bit is used for, which is really an adjunct of the first thing, is this: if I'm in system mode and I do any kind of transfer to a relabeled address, I automatically go into user mode. Suppose the user has executed a SYSPOP, which means that the link has been put into location 0. I start executing the SYSPOP, the SYSPOP goes cruising along and executes and finally it finishes and wants to return. The way a POP normally returns is with BRR 0 which means transfer to the address given in 0 plus one, which is exactly what you want. So the SYSPOP does BRR 0. Now what happens? The BRR goes off to 0 and it says "I'm supposed to go to this address plus one", so it figures that out. Then it looks at the sign bit and it says, "Is that address relabeled or not?"

If it's not relabeled, if the SYSPOP was called out of system mode, it just goes off to the absolute address that was specified. If the SYSPOP was called out of user mode, then it will go back to user mode automatically, because a transfer to a relabeled address sends you back into user mode. This means that the SYSPOP returns immediately into user mode without any disturbance, with having to say explicitly to the hardware, "I want to go back to user mode." Everything is taken care of by this bit being on or off. A system mode routine can also go into user mode explicitly, as it were, by writing BRU *+1,4.

Q Where does the POP find the sign bit?

A The sign bit can be anywhere along the chain of words you go through to get the address. So that, in particular, it can be in the instruction itself. For instance, I can do BRU ABC,4. The sign bit means that I will take ABC as relabeled and when I transfer to ABC I will go into user mode. If I do BRR 0 and 0 contains an address with the sign bit set, then the 0 will be taken as an absolute 0 since I'm in system mode and I go to absolute 0. BRR is similar to indirect addressing—so it works the same as BRU indirect. The only difference from BRU indirect is that it adds one to the address. So I go to this word, (0) which is absolute, and I say "Aha, I am still running down the chain", so I have to add one to the address in 0. Now I have to look to see if the next step is to be relabeled or not. The sign bit in 0 being on means that I'm now relabeled.

Q So the address that you go to has the sign bit?

A The word *containing* the address has the sign bit. I called a SYSPOP. It's definitely not necessary for the word I return to have the sign bit on. That has nothing to do with it. The word that has the sign bit on is absolute.

Q You have your BRR indirect or whatever you have. It picks up the address and where the address is is where the sign bit is to be on?

A That depends. If it goes indirect the sign bit is anywhere in the addressing chain. It could be set in the instruction. That is not the way it usually is, but it doesn't matter how it usually is. The point is to understand what the hardware does. It is not helpful to try to understand what it usually does, because then if it does something unusual you'll foul up. What the hardware does is to look at the sign bits of all the words involved in the addressing chain, starting with the instruction and working down as many levels as is indicated by whatever the instruction says and whatever indirect bits may be on. Anytime during this chain when it finds the sign bit set, it switches to relabeled mode. When it's all finished if it winds up in relabeled mode and if it's a transfer instruction, then it transfers to user mode at the time it does the transfer.

Q How does it know when it's a transfer?

A Because if it's BRU it says transfer. The transfer instructions are BRU, BRM, BRI, BRR.

To summarize the action of relabeling, there are 8 six bit relabeling registers, customarily drawn as in fig. 4. When I generate a relabeled address and the hardware ships it out to the memory, it ships it out along with a line that says relabel this before you send it to the memory. That means it goes through the relabeling box. What the relabeling box does is to separate the top 3 bits from the bottom 11 bits—11 plus 3 equals 14—take the 3 bits and use them to look up one of the 8 six bit relabeling registers. It takes the bottom 5 bits of those 6 bits and glues them on the front of the 11 bits, getting a 16 bit address. And this is the address it uses to reference absolute, real memory. This means that these 8 registers can be regarded as defining a map. They are explicit tabular definition of the map from the integers 0 through 7 to the integers 0 through 63. In this way each of the user's virtual pages which he can address with 16K of address space is associated with one of the 32 pages which exist in the real memory. This association is completely arbitrary. The fact that virtual page 0 is associated with real page 23 has no connection with the fact that virtual page 1 is associated with real page 14 and that virtual page 2 is said to be illegal because `40` is in the relabeling register. Each one of the 8 pages operates completely independently of all the others.

Q You've got 5 there, 5 bits gives you 32 pages, each 2K long, so you have 32 chunks of 2K each and 2K address portion is the 11 bits, the real page number itself is 5 bits, the page number is the 3 bits which give you 16K?

A Of virtual memory, right. Which is translated into 65 K of real memory.

Q The map is set up on a dynamic basis by the system itself?

A Correct. When the system finds a particular user it figures out from its tables which blocks of memory it has put the user's program in and sets up the relabeling so that the user will see what he expects to see, regardless of how much convolution has been going on in the memory since the last time the user was around. You see, horrible things can happen, the user gets dismissed, he gets swapped out, lots of people come in and out and a long time later he comes in and he is just not anywhere like the same place in memory that he was in before.

Q Also the program can be non-contiguous?

A Right. Obviously, there is no relationship between one page and the next.

Q It is non-contiguous as far as pages are concerned but it is also non-contiguous as far as memory banks is concerned?

A The memory banks are interleaved. Nobody sees that except way, way down in the hardware.

It is even possible to give two virtual pages to the same real page. You can do anything. What I'm trying to explain is the hardware, and you should understand it well enough so that you can understand all the things that can be done with it regardless of whether they are reasonable or not. It's hard to tell beforehand which ones are reasonable and which ones aren't. If you understand the hardware then you will be able to figure out any application regardless of whether it initially seems reasonable or not. If you try to understand it on the basis of what you want to do now, then you are just going to get confused when you find somebody doing something that is not what you originally had in mind. The hardware doesn't care what you want to do. It says, "You give me certain bits and I'll do certain things." It's very well defined and not too complicated and the most important thing is to understand exactly what it is the hardware is doing in terms of bits received and bits transmitted.

Q How long does relabeling take?

A It takes 20 nanoseconds but that's slack. It's 20 nanoseconds being wasted in the 930. In fact, there's a lot more than 20 nanoseconds being wasted in the 930 at the moment. Relabeling is a trivial opera-

tion, a table lookup, and you can do it very fast. There is no logic required and you are not comparing anything with anything else, or doing anything the least bit complex.

Q How long does it take to set the register, about 7 microseconds?

A To do a `POT` takes 3 cycles, so the `EOM` plus `POT` takes 4 cycles.

There is one more thing to be brought up which is a minor extension of what we already have. In addition to these 8 relabeling registers there are 2 more which are regarded as being part of another full word in which the other 2 registers don't exist. This is called the monitor relabeling, the monitor map. The significance is the following: normally when the monitor addresses memory, if it has not been relabeled, it gets absolute memory. If the monitor addresses location `4720` it will get absolute location `4720` in the real absolute memory. We have put in the following little gimmick, however: if the monitor addresses something in the top 2 pages of its absolute addressing space, that is, if it addresses anything above `30000`, it will be mapped automatically even though relabeling wasn't specified, and it's mapped through these additional registers. The reason for that is that the monitor can refer to memory which is neither absolute nor part of the user's current relabeling without having to change the user's current relabeling. For instance, the way these registers are used is that the top block is always defined to be the temporary storage block for the current user, which means the monitor can get at the information in the temporary storage block for the current user without having to do any relabeling twiddling. You can't do that absolute, because that block is swapped like everything else. The way we used to handle this problem is that the monitor would have to explicitly change the relabeling. We had to get a word which has the relabeling for the temporary storage block, `POT` it out into one of the two regular relabeling words, and then fix that back up before going back to the user. But that was most unsatisfactory and this is much better.

Q Is that the zero page in the user's core?

A No, the temporary storage block is something the user himself never sees. It's block in which the system keeps the information about the user which it needs, as well as the drum buffers. The user never has access to that block. It's a disaster if he gets access to it. It's almost as bad as getting access to the system proper because all kinds of pointers and things are there—the system, for instance, keeps in that block information about what the user's relabeling is, and a lot of other information like that. That block is sacred.

Q Another thing I don't understand. You say the top 2 pages of the monitor itself are mapped, above `30000`. If the monitor operates like a user, how does it have more than 16K?

A `30000` octal. Addresses are always in octal. Sizes of things are often in decimal but addresses are always in octal.

Q Will you tell us exactly what goes into that monitor map?

A I'm talking about the hardware. You can put anything in it you want. These are two registers which operate exactly the same way as the 8 user mode relabeling registers. The only thing is that what invokes them is a little bit different. In particular, the way the hardware works, of course, it that it looks at an address coming out from the monitor. The address is again divided up 3 and 11. We want to map the top 2 pages. That means that we want to say, "If these 2 bits are one, then we are in the top 2 pages. Use the next bit to select a monitor relabeling register." Instead of using all 3 bits to select one of 8 registers, the way it does in user mode, it only activates relabeling if the top two bits are ones, and uses the third bit to select one of the two registers.

Q What about the extension registers?

A They're not useful. They're much too difficult to handle and they can only address 32K and generally they're not—they're just very inflexible and confusing and not at all nice.

By the way, the monitor registers don't have read only capability. That's not a fact of any great significance, because the monitor, which is the only person using them, is supposed to know whether it should clobber the system or not.

There's one more thing to cover under modifications to the hardware. There were four traps which we listed earlier, and we discussed three of them—illegal instruction, memory violation, and read only traps. There is one more trap, which is called the user mode trap, and its significance is that if it is on (it can be turned on or off by an `EOM`), whenever a transition occurs to user mode by way of a branch to a relabeled address (that's the only way you can get a transition to user mode) the branch will be suppressed and the trap will occur instead.

Q What was that again?

A The user mode trap can be turned on or off, and when it's on whenever a transition occurs into user mode, which happens by way of a branch to a relabeled address, the transfer will be suppressed and the trap will occur instead.

Q What is it for?

A If the clock interrupt occurs which says the user should be dismissed and you are in the middle of a `SYSPOP` of some kind you can't dismiss him right away because you may leave the system in a bad state. You have to wait until the system is finished and goes back to user mode. In user mode you can dismiss him at any time, but you can't necessarily dismiss him in system mode because the system may be doing something, and the clock interrupt routine doesn't particularly know what the system is doing. What this means is that when the clock interrupts and figures out that it should dismiss the user, it looks to see whether it came out of system mode or not, which it can do so by looking at the sign bit of the address stored by the `BRM`. If it came out of user mode it dismisses him right away and if it came out of system mode it turns on the user mode trap. This means that when the system finishes whatever it was doing and goes back into user mode, a trap will occur and at that point the user will be dismissed. This sounds trivial, but it actually turned out to be quite important. The way we were handling this problem before is that the clock would set a flag, and practically every single exit from the system into user mode would check this flag, which was a damn nuisance, somewhat wasteful of time and very inconvenient.

Q Does this guarantee a certain number of microseconds of processing time?

A That is not the idea. The system can continue in system mode as long as it wants. Unless there is a bug in it, it is controlled and it doesn't stay in system mode for very long.

That covers the modifications fairly exhaustively. These things are all described in the paper which is referenced in the outline. Some of them aren't; the user mode trap and the monitor map were added since this paper was written, but most of the things are described and many of them in some detail.

We ought to conclude by describing the unusual pieces of hardware that are attached to the system. Interrupt arming we already discussed. The clock is very simple, it's a device which 60 times a second generates an interrupt, a very high priority interrupt, higher than anything except power fail. 60 times a second you get this interrupt. What you do with it is up to you.

Q Doesn't the system keep track of it?

A The system keeps track of lots of things. It increments various counters, and decides whether it should dismiss people and one thing and another. But we will do that all in detail when we look at the system proper. We are looking at the hardware at the moment.

Q There's only one clock?

A There's one clock. Right. It generates an interrupt 60 times a second. Every time an interrupt occurs the system does something. Usually it just increments or decrements a counter. Under some circumstances it will do more things.

There's the teletype interface, which is a box called the CTE 11 which is provided by SDS. Its characteristics are roughly the following: a lot of wires come into it. Something is on the end of these wires and this something has to have certain properties; it sends things down these wires in the certain way. If it sends 10,000 volts down the wires it will blow up the teletype interface, so it's got to do something reasonable! It doesn't have to be a teletype; it just has to be something that looks like a teletype. On the other side is the computer. There are the following connections: 24 lines, the POT-PIN lines, and there's an interrupt line, or two interrupt lines. Whenever a character comes from the teletype, the teletype interface generates an interrupt. The computer in processing this interrupt does a PIN which reads 24 bits from the teletype, from the POT-PIN lines. These 24 bits contain two pieces of information: the number of the channel on which the character was input and the character itself, 8 bits o character.

Q There are eleven bits per character?

A Three bits are not part of the code, but are used for synchronization. Eight bits are the code.

On output the CPU does a POT out to the teletype in which it specifies a teletype number and a character to be output. This character goes and sits in a buffer in the interface, and the interface does what's necessary to output it. When the output is finished, after 100 milliseconds, there is a little timer, which causes the interface to generate an output interrupt. The CPU can do a PIN and find out which teletype is responsible for the output interrupt. The significance of the output interrupt is "I've finished outputting that character. You can give me another character if you want". In this way the CPU can read input from the teletype interface and send output to the teletype interface. In both cases the teletype number as well as the character involved is sent.

There are two interrupts which the teletype system generates, one to say "I've got a character ready for you to read", and the other to say, "I finished outputting what you told me to output and I'm ready to output something else." Individual channels on the teletype interface operate, from the computer's point of view, complete independently. Any interrupt may come from any of the channels and the fact that output is going on in another one. Those are the essential characteristics of the teletype interface hardware.

Q Do you have an interrupt for each teletype?

A No, two interrupts in all. It's because there is only one interrupt for input and one for output that you have to provide the teletype number in the POT-PIN operation. This is obviously better because with 64 teletypes it is ridiculous to have 128 interrupts.

Q Why is it that it works on a character basis than on a double character basis, for instance?

A Because you don't want to work that way. You want everything character by character. That's one reason. Another reason is that if you work on a double character basis, you have to twice as much buffering in the interface. The buffering represents about 2/3rds of the cost of the teletype interface, and it would be expensive to have more. If the interface has to collect—for instance, if you use the

double character scheme for output—you ship two characters instead of one to the interface—the interface has to have storage for those two characters instead of for just one.

Q Doesn't it have 24 bits for the 24 lines?

A Oh, no that's quite different, there's no 24 flip flops there. The way these lines are set up is this: a lot of them are always zero, the ones in the middle. The 8 lines for the character come from a character buffer, and the lines for the tty number come from some unary to binary decoder which sees line such and such and encodes it into binary. When you do the `POT` you send out 24 lines, 24 bits. When you `PIN` you read in 24 bits. Those 24 bits contain a character over on the left-hand side, the teletype number on the right-hand side and zeros in the middle.

That's the hardware. What you can do as a user is very involved and is one of the things we will spend quite a bit of time on. There's a lot of machinery associated with making the teletypes work the way the user wants them to work.

Char	0 0 0 0 0 0 0 0	tty #
0-7	8-17	18-23
`PIN`/`POT` Lines		

There's one more piece of significant hardware and that is the drum or RAD and I will probably call it drum most of the time. This is primarily a swapping device. It operates at half core speed, full core speed, or one-quarter speed depending on what it is. The RAD operates at quarter core speed, and it runs over a special direct access to memory channel (`DACC`). Normally it is run with blocks of reasonable size, either 2000 words at a clip or 256 words at a clip, depending on whether it is being used for swapping or for file storage. Physically I believe the RAD can operate on units as small as 64 words. It is basically like any other I/O device with one exception. The channel has a little gimmick in it called the channel map. Memory is divided up into 2000 word pages, so that in the user's relabeled memory absolute address `13777` may be right next to absolute `50000`, because the pages are completely independent. If the user says, "I want output from `3700` to `4300`, one 256 word block, this is normally read or written from one 256 word block on the RAD. The RAD channel works on absolute memory addresses, not on relabeled addresses, and this means a serious problem here, because this 256 word block is not a single block in absolute memory. If we start at `13700` and end at `50277` there is no way to tell the channel that we really want `13700` to `13777` and `50000` to `50277`. The way the situation is corrected is the channel has a small map in it which essentially allows it to say, "When I cross the page boundary, I can switch from absolute addressing to addressing through the map, which allows me to get a different absolute page."

The scheduler

The scheduler is what decides when programs get run, when they don't get run, what happens when they have to wait for I/O, what happens if they compute for too long, etc. The relevant section of the working document is section 2. The relevant section of the listing is the huge package called `SPAC`. `PAC` stands for "program active", and this section of the listings contains a large amount of code. It contains the scheduler code at the beginning where it says "Scheduler" in the comment, and then it contains a lot of code associated with forks and a lot of code associated with the swapper, which we will have occasion to discuss later.

The idea is the following: there are these things in the system called processes. They are sometimes also called forks. In fact, they are always called forks in the working document. I will probably call them both. A process is a collection of information which suffices to allow a program to be executed. Looking at it in another way it is an entity which can execute instructions. The way a process is defined is this: suppose a process has been set running by some mystical process and we want to stop it. A process is defined by all the things we must save when we stop it if we are to start it up again in such a way that it

will not know it has been stopped. What are these things? First of all, there is the map. Then there are the central registers, the A, B & X registers and the program counter. Besides that there is some kind of status information which says why we stopped it and when we should restart it. In actual fact there are some other things that are used to define a process besides these which have to do with other features of the system. These will be discussed later on.

All this information for each process is kept in a table called the PAC table or program active table, which you will find in the working document immediately following page 2-1.

Q Is there one of these for every active process?

A There is one PAC table entry for every process, every active program, every fork, every whatever you want to call it.

And it is this entry which essentially defines the program. If you have one of these you have a process and you can run it. If you don't have one of these you have nothing. That's what an active program is and these active programs are the things with which the scheduler deals. Each one of them, as I have said, is defined by a PAC table entry and is identified to the scheduler by its index in the PAC table, which is a negative number called PACPTR, a number which you will find occurring throughout the system. The significance of PACPTR is the following: that there is a bunch of symbols in the system called PX, PL, PB, etc. , etc. which you find running down the left hand side of the page 2A, and if you write something like this

```
LDA PL, 2
```

and you have PACPTR in X, you will then get PL word for that particular program, for the program identified by PACPTR.

Now the meaning of this from the point of view of the PAC table is the following: if the PAC table is a section of core, PL, PB, etc. are symbols whose values are addresses at the end of this section, and PACPTR is always negative. The first program in PAC is the one in the earliest core locations and has the biggest negative PACPTR. The smallest negative pack pointer is -12, since a PAC entry has 12 words. The size of PAC is a parameter; it is defined in MSYMS. Why the PACPTR is negative I'm not exactly sure, but we are now irrevocably committed to its being negative because there are lots of places where we do SKN.

PACPTR is the handle on the program, and whatever you do with this active program, whether it's running or whether it's sitting on some queue somewhere waiting for this or that, it is always identified by this number. If you ever get an entry in the PAC table which has no PACPTR pointing to it, then that entry is lost for good; you'll never find it again.

The next question is the following: suppose we have an active program—suppose we have lots of active programs—we have the following obvious problem. We can't run them all at once, since we have only one CPU, so we have to run them one at a time. This means that first of all we have to be able to start them up and stop them. Secondly, we have to have some sort of rule for deciding which one we run next.

Q Where is the PACPTR kept?

A It depends. It's just a number; you can put it anywhere. It may sit in some table or be in some queue entry, or anything. There is a word in the system called PACPTR which holds the PAC address of the program currently running.

Q How do you get it?

A Who is "you"?

Q The user.

A The user never sees a PACPTR. He has nothing to do with that. Remember, he doesn't know that there's time-sharing going on. He just has this big computer to himself.

Q This is set up in the beginning?

A When the monitor establishes a process it finds an empty slot in the PAC table for it. Maybe I should explain about that. There is this thing called the free program list, FPLST, which is essentially a list chained through all the empty entries in the PAC table. When you need a new entry in the PAC table, you pick one off the front of FPLST. And in the very beginning of the world the monitor sets up FPLST by putting all the entries in the PAC table on it. When you need an entry from the PAC table you take one from FPLST, and when one becomes obsolete you put it back on FPLST.

Q How do you know which number is the number for PACPTR.

A For what? Which number? For what PACPTR? That depends on which program it is for. Suppose a program is dismissed because it's waiting for teletype 64, Then presumably somehow associated with teletype 64 there will be a word which contains the PACPTR of the program that's waiting for teletype 64. When teletype 64 comes along and says "O. K., it's time to go", you can find in that location the PACPTR for the appropriate program. Exactly how these things are organized is the function of the various parts of the system. You'll see lots of places where PACPTRs show up. I can take PACPTR and put it in cell 103 and then I can come back later, take it out of cell 103 and do something with it.

Q But it's usually kept in one place?

A No, it's kept in lots of different places depending on what's going on. The program will get dismissed for all kinds of different things, and PACPTR winds up in all kinds of different places.

Q Including the program?

A The program itself never sees its own PACPTR. It doesn't know about that, remember?

Q Are there quite a number of these things, of PAC slots?

A It's a system parameter how many there are. Look in MDBG and MSYMS.

We now proceed to discuss the operation of the scheduler. First, activation and dismissal. First of all, if a program is running how can it get dismissed? A dismissal occurs as the result of one of the following things: You might be waiting for I/O. For instance, you ask for a character from a teletype and the character is not there yet because the guy went for coffee. So there's no character. Well, it's ridiculous to leave you sitting there waiting for the character tying up the CPU while the guy gets his coffee. You get dismissed and eventually when the character arrives you get reactivated. Or alternatively, maybe you have 350 characters to type out. If the teletype output buffer is only 30 characters long, you ship 30 characters out one at a time to the teletype. Then you come along with the 31st character and there isn't any room for it. So you have to be dismissed until there is more room. Again, you could request dismissal explicitly, usually on some specified condition like time or waiting for another process, or some other thing like that. Finally, you could compute too long; that's called quantum overflow. Those are the important things.

So basically there are three ways to get dismissed: you could be dismissed if you asked for I/O that's not ready, because of your explicit request, or because you computed for a long time and the computer

decided it was somebody else's turn. Of these three kinds of dismissal, two are produced by your explicit request to the system. The other is produced essentially by an interrupt (the clock), and is involuntary.

Q What is the amount of time for which a program gets to run?

A It is defined in a very complicated way which I am about to explain.

Q How long can a program wait for a character, say, before it gets dismissed?

A If it asks for the character and the character is not there it gets dismissed immediately. You don't sit there waiting for the character. That's ridiculous because the time involved in getting the character is very long compared to the time involved in the CPU. At least 100 milliseconds, even if things are going full blast.

You've gotten yourself dismissed. Now the next thing is how you get reactivated. That is controlled by a word of the PAC table called PTEST. The exact details of how we get PTEST set up will be discussed in a minute, but the significance of PTEST is the following: it contains essentially two pieces of information, the activation condition and an address of some kind. The activation condition happens to be kept in the opcode bits of the word, and the address is kept in the address field. This is not essential.

Q How do we reactivate? What do I have to do, look at PTEST?

A Somebody is going to look at PTEST to decide about reactivation. Now the exact way in which we do that we will get to in a minute, but I want to explain how activation works first.

When the program is dismissed the thing that dismisses it will know why it's being dismissed. Maybe the teletype routine is dismissing it because it asked for a character and none is there. Or maybe the drum routine is dismissing it because it asked for drum I/O and there are already six other people waiting for the drum so it's down in some queue somewhere waiting for the drum. Or maybe it's being dismissed waiting for 3 o'clock. The routine to process dismissals waiting for 3 o'clock knows it's until 3 o'clock that it's supposed to be waiting. In any event, whoever is doing the dismissing knows why the thing is supposed to be dismissed. If the dismissal is because of the quantum overflow there is no particular condition for reactivation. If the computer gets around to the process, it should be reactivated.

There are lots and lots of dismissal conditions. There are dozens of different things which can cause dismissal, many different kinds of I/O conditions and lots and lots of specific requests. The idea is the following: you do not want the scheduler to be aware of all these details of different possible conditions for reactivation. If it is, it has to know far too much about other parts of the system. We would like to have the scheduler decoupled from other parts of the system as much as possible. In other words, we would like to have a mechanism by which the scheduler really only knows that it's supposed to go to some word and look at bit 23. If bit 23 is on then program is activated, otherwise not.

Somebody else is responsible for how bit 23 gets turned on. If it's the teletype dismissal, the teletype should be responsible for figuring out that "I should turn bit 23 on", the scheduler is only responsible for looking at bit 23. In fact, we don't even have the ability to look at bit 23; we have much more simple-minded things than that. The activation conditions are listed on page 2-4 of the working documents—and you see that there are not very many. The most commonly used ones are 0, 1, 2, and 3, which essentially say "Go and look at the word in the address over here and find out whether it's >0, ≤ 0, ≥ 0 etc.

For instance, the simplest possible way you could handle a dismissal would be the following: you dismiss the program until a word is ≤ 0 and set the word to +1. Then when it's time to reactivate the program the interrupt that gets it reactivated sets the word to -1. Then the next time the scheduler comes along and test the activation it will say, "Aha, time to reactivate the guy".

Is this mechanism clear to everyone? This is quite important. The significance of it is that scheduler which decides to activate is decoupled completely from the routine which decides whether activation is permissible. Communication is achieved only through contents of certain memory cells. There are some activation conditions that are sort of funny and do have a little bit of relationship to the real world, like number 3. which is word < teletype early warning, early warning being a system parameter which says essentially how many characters you should allow to pile up in the teletype input buffer before activating a program. We'll discuss that in more detail when we get to the teletype logic. And number 4, which refers to time delay, has an explicit reference to the clock. Number 6, which is sort of a funny one, word equals address of word plus 2, results from the conventions used in the I/O part of the system. All the I/O buffers have the form of 2 header words followed by the data words. The buffer is in some sort of initial state when the first header word points to the first data word, i. e. , points to the second word after itself. That's the reason for condition 6.

Q Why don't you show what happens in a typical I/O operation?

A Okay, suppose I am a program executing a `TCI`. I go bouncing off to the teletype routine and the teletype routine looks to see if there are any characters. Now there's a word somewhere that says how many characters there are in the teletype buffer, It looks at that word, that word is zero. No characters. So the teletype routine sets up the activation condition, and the activation condition is this: there is a word in the teletype table which essentially says "It's time to run this guy now, because a break character has come in" when it is positive; every break character increments it by 1. So when the teletype routine recognizes that it is time to dismiss the guy it sets the word to -1, and it sets up an activation condition which says activate on word non-negative. This means that every time the scheduler tests this condition it's going to ask "Is this word > 0? " and the answer is going to be "No", as long as no break character comes in on the teletype. When the teletype interrupt routine is fired up by the arrival of a character and it observes that this character is a break character, it will increment the word.

Q This allows you to completely decouple the scheduler from all the routines that might cause an interrupt in the program?

A Exactly and this goes both ways. The scheduler doesn't have to know about them and they don't have to know about the scheduler. All the teletype interrupt routine knows is that if a break character arrives there's this magic word it's supposed to increment. All the scheduler knows is that there's this magic word it's supposed to look at to see whether it is negative or not. That's the extent of the communication.

This, by the way, is a technique used throughout the system and explains many things which you otherwise might find extremely puzzling. The system is not organized on the basis of a subroutine which calls 6 second level subroutines each of which calls 6 third level subroutines etc. Very rarely do you call subroutines to more than two levels. Communication is always accomplished through tables in memory, and what happens is that one routine will set up a table and then it will go away. Later on some interrupt will come along and fire some other routine which will look at that table and on the basis of what it sees it will decide something. This is the philosophy on which the system is written. It is not the way most programs are written and is a little bit difficult to understand.

Q What's a break character?

A A break character is a character which essentially says that the program is supposed to pay attention to this character right away. We'll discuss that in more detail when we get to it.

So enough on activation and dismissal. Now, the question is how to organize all these programs? Say we have three programs that are dismissed that are waiting to run. How do we organize them?

Somehow the scheduler will have to be able to find them so it can look at their PTEST words. The stupidest way to do it would be to say "The scheduler will scan the PAC table, starting at the beginning and going to the end, and it will check the PTEST at each slot". This is how we did it for a while and this system has the virtue of simplicity; there's nothing to get out of order. There's really a lot to be said for having a big table and just scanning it, because you don't have lists and queues and little pointers and things and there's so many fewer little pointers that can be pointing to the wrong place. If you just have this simple scan there's no way for the scheduler to get into a loop, whereas if you have a list and the list becomes circular the scheduler will go round and round and round and never gets out. And that, in fact, has happened.

Unfortunately, this scheme is basically not satisfactory in spite of the advantage of simplicity. First of all, it incorporates no mechanism for priority, because the position of the program in the PAC table is fixed and consequently there is no way of saying "This program is more important" by looking at it first. The second objection is the PAC table is very large and many of the slots are empty or are devoted to programs which are of no interest to the scheduler for. one reason or another. Consequently you waste a lot of time looking at things.

So we established a somewhat more ingenious scheme which involves having three scheduler queues. This mechanism is described rather compactly on page 2-5. The queues are called QTI, QIO and QQE. QTI stands for teletype queue. QIO stands for input-output queue, and QQE is for quantum exceeded, meaning you computed too long, you got time sliced. Whenever a program is dismissed, it is put on one of these queues, and the queues are organized in the following way: Each one has one word associated with it called the header word, and then the queue entries, which are PAC table entries, chained through PNEXT. There will be some programs on the teletype queue, and there will be some programs on the I/O queue and there will be some programs on the quantum exceeded queue. The first word of the header for next queue, which in turn points to the first program, which points to the second program, which points to 0 [this description seems wrong]. And so it goes. Now we can define the priorities of the programs. There are three priorities corresponding to the three queues. QTI has the highest priority, QIO is the next highest priority, and QQE the lowest priority. You can always add more queues if you like. That's not very hard.

Q For instance, if a guy comes out of QQE and executes and then is hung up on QTI? There's no facility to scan QTI and give this guy higher priority than someone already on QTI.

A No, there is not at the moment. However, there is a standard routine for putting people on queues and you could modify this routine quite easily. The routines called QPUT and QGET are responsible for putting people on queues.

Q All they do is splice pointers?

A That's right. At the moment there is no mechanism in the scheduler to establish priority.

Q To establish a new queue all you do is establish a new header word?

A That's right. And you have to find the place in the system where the headers are initialized. After that everything should take care of itself.

Q You can establish a new queue just like that?

A You have to change the system to establish a new queue, but the point is that is not very hard. You cannot establish a queue at any time, definitely not. What I'm doing is explaining the mechanism which is used and pointing out that it's not very difficult to add queues.

So this is the queue mechanism and it's obvious how people dismissed for the various conditions go on the various queues. The algorithm the scheduler uses is the following: when it decides that it's time to look for a new program to run, and how it decides that is something we haven't discussed, but when it decides that it starts off on the teletype queue and goes cruising down the teletype queue looking for someone to be activated. If it finds somebody to activate it takes him, since he is the highest priority guy. If it doesn't find anybody, it goes right on, cruising down QIO looking for somebody to be activated. If it finds somebody there it activates him, and if it doesn't find anybody it goes right on cruising down QQE looking for somebody to activate. And if it doesn't find anybody at all it goes right back to the beginning and starts over again because it has nothing better to do.

Q After a process has been run, do you run the next process in the queue?

A No, because in the meantime somebody earlier in the queue structure may have become activatable.

When you put people on the queues you normally put them on the end, although there are a few places where you put people on the beginning on a queue. So this is the queue mechanism. Now we should conclude by discussing the timing, which controls how people get dismissed and reactivated. You may be dismissed for I/O; that happens because you made an I/O request and the I/O's not ready. Then you get put on the appropriate I/O queue, whether QTI or QIO, and you are then dismissed and the scheduler goes on to somebody else.

The mechanism by which it is handled, by the way, is almost always the following: suppose you do a TCI and the character is not ready. It dismisses you, leaving in the entries of your PAC slot the contents of the central registers, and in particular leaving in the entry for the program counter, the address of the TCI instruction. When you get reactivated the scheduler doesn't know that you were doing teletype input; that's not in the picture at all. All it does is to set up your map and gets your memory into core, and then it transfers to the specified location. So the way in which you finally get the character is that the TCI instruction is executed all over again from the beginning. And this is normally the mechanism that's used.

This has serious implication for what TCI can do, for what the routine that implements TCI can do. In particular it cannot do anything irrevocable before it figures out whether or not to dismiss the program, because that irrevocable thing is going to be done a second time when the instruction is reexecuted. If the TCI increments some counter, for instance, and then decides to dismiss the guy, it had better decrement the counter, because the net effect on the guy who is dismissed is as though the TCI was never executed. It is as though there had been an instruction that said "check" sitting immediately in front of the TCI and that it was the check instruction that caused the dismissal, because when the guy's reactivated you come back and you redo the TCI routine from the very beginning. No information is available which says that you already did it once and got dismissed. That information is completely lost. Is this clear? It is very important, if you add dismissal conditions, that you understand this mechanism.

There are some cases in which this restriction is intolerable, and for those cases there is a special gimmick called the "phantom user" which is given the responsibility for doing things where you can't just go back and start at the beginning. It is very horrible.

Q You're worrying about doing something twice. Why can't you naturally conclude one thing and then start where you left off?

A Because the difficulty is preserving information that tells where to start. If you get half way through the teletype input routine and decide, "Okay, it's time to dismiss the guy, but there are 16 temporary storage locations that I've used that I want saved," then somewhere there has to be space for saving those 16 temporary storage locations, because somebody else is going to come along and use the teletype routine while you are dismissed. And furthermore, it's different for each routine that can cause

dismissal. The storage allocation becomes very messy. Also slow. Furthermore it's very important to handle things so that all the dismissal conditions are treated in exactly the same way, because otherwise the scheduler will have to take different action for each dismissal condition when it re-activates the program, finding the temporary storage locations that were saved here and restoring them. Each time I added a new condition I would have to add more machinery and it would get out of control before you had a quarter of a system done. Hence the rule that normally when you dismiss a guy, you dismiss him in such a way that when he is reactivated the instruction that caused the dismissal will be re-executed. This has implications about what the routine processing this instruction can do. Say, for instance, it might be convenient in the implementation to increment some counter that says how many characters have been taken out, or something like that, before you actually find out that there are no characters there. It's hard to know without looking at the implementation what problems may arise. However, it's important to be aware of the fact that when you do implement the TCI routine you don't have complete freedom in how you do it because of the possibility that you may have to dismiss the guy. If you do dismiss the guy you must make sure that everything is clean.

Note that all of this is of course invisible to the user. As far as he is concerned, each instruction is executed exactly once.

Q Is the TCI routine very long?

A A couple of hundred instructions. It has about six different special cases it has to look at.

Q Why don't you make it reentrant?

A Then you'd have to remember the program counter of the location of the TCI instruction. That would be one more thing to remember.

Q Why are you discussing this in so much detail if the user never has to worry about it. Why do we care about the system problems?

A I realize you don't care about our problems in the TCI instruction, but if you want to make an addition to the system which involves dismissing somebody, you better be aware of what the situation is, because otherwise your addition is not going to work. Worse, it's going to not work in a very obscure way. The whole point of these lectures is for you to understand the system, so that if you want to make some change you can make it. Otherwise you could just read the working document. That tells you all the external things you need to know.

The next thing to discuss is the quantum overflow machinery. The details are quite complex, but the idea is essentially the following: every process has associated with it two quanta called the short quantum and the long quantum. These two numbers are both parameters. In fart, they are not only parameters in the system, but the long quantum ran be different for different users. There is a little table of 16 possible long quanta. If you look in the PAC table picture, you will see that there is a word called PQU which contains a field called QUTAB, a 4 bit number which tells which long quantum is being used. You could use this for the short quantum also. We haven't chosen to do that but maybe we will. At the moment there is only one short quantum and it's set at 8 clock cycles. This choice very much depends on what kind of performance you want from the system.

This arrangement gives you quite a bit of flexibility. You could change the long quantum on the basis of a user request, or you can change it on the basis of a system calculation involving, say, the size of the user. If he is really big, it's a lot of work to bring him in, so he should run for longer once we do bring him in. There are lots of different things you can do which you can figure out for yourselves.

The two numbers are used like this. Suppose that I have a user who is going to compute for many hours. There is no possibility that he is getting dismissed for I/O or anything, he is just going to com-

pute. I fire him up and he is guaranteed that he will be allowed to run for the time given by his short quantum. He will definitely be allowed that much time, independently of anything that happens in the outside world except for somebody coming along with a hammer and smashing the computer. After that his life becomes considerably more problematical. In particular, if anybody who is waiting for I/O becomes activatible at any time after the guy's short quantum has run out, he will be dismissed right away, on the next clock cycle. If nobody waiting for I/O has become ready to run, he will be allowed to continue running until his long quantum runs out. He always gets to run for a short quantum.

Q After he's in a short quantum the second time around...

A He doesn't get into a short quantum the second time around. He runs for the short quantum and then he gets into the long quantum. At any time during the long quantum he can be dismissed as the result of somebody being ready to run who was on QTI or QIO. If that situation doesn't occur, he will be allowed to continue running until the long quantum runs out. At this point he will be dismissed if there is anybody waiting for any reason at all, in particular if someone else is waiting to compute.

The implication of all this is the following: Suppose you have a system with no I/O going on at all, but lots of people computing. Each person will be allowed to run for a time equal to his long quantum. Then he will be dismissed in favor of the next person. There will be a fair round robin, with the length of the time slice for each person being given by the long quantum.

Now, if a person gets dismissed for I/O you save the amount of the long quantum that's left in the QR field of PQU. When he is reactivated he will continue to use up this long quantum. Eventually he will be dismissed for quantum overflow and the users waiting to compute will get a chance. The exact mechanism is sort of confusing. We will look at it in detail when we go over the code.

The following obvious question arises: how is it that we go about testing whether anyone is waiting for I/O after a short quantum has run out. We can't do it by running through the scheduler each time a clock interrupt occurs; that's too slow. Instead, we have this word called ACTR (which by the way is described in the working document). ACTR stands for activation counter. It is normally set to minus one. Whenever any interrupt routine comes along which makes somebody activatible, it increments ACTR. This means that the only thing that the clock interrupt has to do is to test ACTR. If it is still negative, there is no chance of activating anybody who is waiting for I/O because the only way that somebody waiting for I/O can become ready to run is for some interrupt routine to come along and make him ready to run. So that's the mechanism, and it is described in gory detail towards the end of Section 2.

Before we start talking about the PAC code, maybe I'll say a few words about the organization of the system. It comes in packages, each package being represented in the listings which you have by separate bunches of pieces of paper stapled together and identified by a name beginning with S which you find at the top of each page of the listing. There are several packages which consist of code: PAC which is the scheduler, fork structure, swapper etc.; W, which contains the W-buffer logic; TTY, which contains the teletype logic; DRM, which contains the drum I/O; and SIO, which contains the routines to handle the general user interface for I/O in the system. Then there is a package called MSYMS which contains the constants and temporary storage locations and tables for the monitor, and there is a package called MDBG which contains a large number of flags and OPDs and macro definitions which are common to all the packages of the system. You will notice in the middle of page 8 of MDBG there is in solitary splendor the word FREEZE. The meaning of that is that assembling MDBG will cause the assembler to put itself into a state in which you can assemble the other packages of the system with the assembler thinking that each one includes the portion of MDBG before the FREEZE. What the FREEZE does is to convince the assembler that all the symbols and macros that have already been defined are permanently built in. After having done this FREEZE we use the same copy of the assembler, without allowing it to reinitialize itself, to

assemble all the other packages. Thus they will all be assembled with the same collection of OPDs, constants, etc. which we put in the 8 pages of MDBG which contains all system flags and parameters, together with a collection of standard macros. So you'll find throughout references to various things that show up in MDGB. You also find reference to lots of things in MSYMS, which are always storage locations of one kind or another. Things in MDBG are mostly flags and operation definitions and things like that. MSYMS isn't used like MDBG. It is assembled by itself and it uses the external linking facilities of the loader. MDGB is different. It has all these parameters and things which are going to be used to control the assembly and to define table sizes and things like that, so you can't handle them with external linking.

Another thing which should be explained is that these programs are listed on a 1401 which has 48 characters. We have used the overprinting feature to generate the missing 15 characters in various obscure ways. In particular semicolon is generated by a comma overprinted with a minus sign. Semicolon, by the way, in the assembler is exactly equivalent to carriage return.

We now begin looking at PAC. On page 2 there is a comment: trapped feature. Now TRP is a macro which is defined in MDBG, and what it does is to make all those symbols equivalent to the symbol TRAP, which is the address of a location which causes the system to generate an illegal instruction trap. The reason that these are trapped features is that these are all the features that aren't implemented. There are various tables which point to these symbols, which are supposed to be the beginning of routines to handle these features. The tables can be set up as though the features were there, and if we simply equate all these symbols to TRAP, we can trap all the unavailable features. As we implement them, we can take the appropriate symbols out of the argument list of TRP without any change in the main tables.

The heart of the scheduler is the code starting at POPX which is in the middle of page 3, so we will look at that first and we will then go back and look at other things. The scheduler starts there and it goes on right into the middle of Page 5. POPX is a routine which is the standard exit for system program operators. It restores the central registers from locations called SS01, SS02 and SS03, which are the standard places in which the central registers are put by system programmed operators. In particular, the operator called BRS, which is the sort of all-purpose operator whose address is used to indicate lots of different routines, always puts the central registers in these locations. If you add BRSs to the system, which you probably will, that's something you should bear in mind. POPX then goes to XPOP, which is the standard exit for SYSPOPs after the central registers have been fixed up. If the SYSPOP wants to change some of the central registers it can just set them up and then go to XPOP. Once you get to this point you assume that the address in the user program which you are supposed to return to is in zero, which is where it will normally be after the POP has been executed. If zero gets clobbered the routine that comes back to POPX or XPOP is responsible for setting it back up to this user program address before coming to one of these locations.

TIME is the short quantum count, and if it is not negative the short quantum has not run out, and you definitely exit with BRR 0, which takes you right back to the user program because the link for the user program is in 0, as I just explained. If TIME has gone negative, then the short quantum has run out, so we look to see if the long quantum has run out by saying SKN TTIME, TTIME being the word containing the long quantum. If the long quantum has run out we do SKN 0 which tests whether we came out of user mode or not. If we did come out of user mode then the sign bit of the link in 0 will be set. If TTIME has gone negative, in other words, if the long quantum is used up, then we test to see whether we came out of user mode or system mode. If we came out of system mode we return, (since a system mode program cannot be dismissed); if we came out of user mode we know that quantum overflow has definitely occurred.

If, on the other hand, the long quantum has not run out we test ACTR. If ACTR is negative, then we know that even though we are in the long quantum, no I/O interrupt has occurred, and therefore we return. If ACTR is not negative we know that some interrupt has occurred and that we should dismiss the guy unless we came out of system mode. If we came out of user mode, then we come down where quantum overflow occurred, and we know that we are supposed to dismiss the guy. So we save the central registers in SS01, SS02 and SS03. We then come to the location called SQO. Anytime you go there the guy you were running is going to be dismissed for quantum overflow. There are two ways to get there. One is from XPOP or POPX which we have just described. The other way to get there is by an explicit transfer to SQO, which is done by BRS 45. The user can execute a BRS, namely, BRS 45, to simulate quantum overflow, to get himself dismissed as though his quantum ran out. This is what he does when, for instance, he wants to check some external condition but not tie up the CPU by testing all the time. He says, "Dismiss me on quantum overflow," which means that after everyone also gets a chance to run I'll come back in and test the condition again.

So we have come to the point where we are being dismissed on quantum overflow. We increment 0, the reason being that what we are going to do now is to go into the main body of the scheduler, which expects to find in 0, the address to which it will return control. Since we just came out of a POP we want to return control to the location after the POP. This means we have to increment the link. If we were in the POP we would return with BRI, which automatically does the incrementing. Now we are going into the main body of the scheduler, which is going to return exactly to the address in 0, not to that plus one. In order to make the return go to that plus one, we have to increment.

Then we set TTIME to -1, which indicates that the long quantum has definitely overflowed. There's two ways that could happen. The long quantum might not really have overflowed, because we might have gotten dismissed on ACTR, but if we do get dismissed on ACTR we are going to pretend that the long quantum did overflow. There are times when we preserve the long quantum, namely, if a guy is getting dismissed on I/O we keep the long quantum around. If we didn't do that he could devise the following strategy: compute for an amount of time just less than a long quantum and get himself dismissed for some trivial I/O operation, which would get him on a high priority queue. When that I/O is finished he could come back, and if we reset the long quantum at that time then he would never get dismissed for quantum overflow. Whenever the guy's dismissed for quantum overflow (on to QQE) he gets to start over again.

Then we fall down to PACQE which is where we go to dismiss somebody on to QQE. There are some places in the system that transfer directly to PACQE having already taken care of all the things that we have now done. There we do LDB PACDMB; LDX =QQE. In other words, we put into the B register a dismissal condition and into the X register the address of a queue, namely the quantum overflow queue. Then we go to POPDMS, which is a location that dismisses somebody with the dismissal condition specified in B onto the queue specified in X. PACDMB is defined at the very beginning of PAC on page 1 in about the sixth line. It is defined to be ZRO *,* meaning the current location. In other words, it is a number with zero in the op field, and the address is the word itself. Activation condition zero says "Activate on word > 0," which means that if we use PACDMB as an activation condition we will always activate. Therefore this is the proper activation condition to use for somebody being dismissed on quantum overflow, because we want to always activate him.

The significance of POPDMS, which is not exactly explained by the comment, is that if we get to there we're going to dismiss the program which we are currently running with the PTEST word specified in B and onto the queue specified in X. There are several places in the system which come directly to POPDMS. In fact, I think all the places that dismiss programs come to POPDMS after setting things up

properly. PACLVL is a temporary storage location. What we do is to pick up PACPTR, which tells which program we are running. We put that into A. We now have PACPTR in A, the dismissal condition in B, and the queue in X. Now we call QPUT, which is a separate routine which is responsible for putting a program onto a queue. It is at the end of PAC, and I don't think we will look at it in detail because it is not particularly deep.

We have now put the program on the queue, and we are guaranteed that someday the scheduler will come around and pick it up again. Now before we start up the next program we have to make sure that everything is set up properly in the PAC slot, the central registers and everything else. Thus when the scheduler does come around to reactivate this program everything will be all right. The program counter we expect to find in 0, which is why we have this LDA 0 at POPINS. We mask out in significant bits of this, and we store it in PL,2, namely, in the program counter location for this particular PAC slot. The reason for the ETR and ADM is that if you look in the PAC table picture you will see that there are 6 bits in PL which are used for something else, and we have to protect those 6 bits. We mask them out with this extract and put them back in. Then we save the central registers by picking up SS01, SS02 and SS03 and storing them away in PA,2; PB,2; and PX,2 at the top of page 4.

Now we deal with the quantum. This is to make sure that in the QR field of PQU we set up the proper number for the quantum remaining. So what we do is to pick up PQU and shift it enough to get the QUTAB field in the A register. In other words, this allows us to get the index in the table which will tell us how big the long quantum should be. Then we pick up TTIME and we ask if it's bigger than -1. If there is any long quantum remaining we keep what's remaining, otherwise we pick up a fresh long quantum from the table. Then we pick up PACPTR again and we do the necessary fiddling to put this long quantum into the QR field PQU, which is accomplished with the XMA, ETR and ADM.

Now we are just about ready to start to try to find another program. We have just about finished putting away the old program. The only thing we have still to do is to change TJOB, which is a cell used for accounting. Every time a clock interrupt occurs one of the things we do is MIN *TJOB. This location is a counter of some kind, an elapsed time counter. We are incrementing somebody's elapsed time counter. When a program is running, TJOB is set to point to the elapsed time counter of the user who owns the program. When no program is running, TJOB is set to increment the elapsed time counter for the system. What we are doing now is to arrange that if a clock interrupt occurs while we are in the scheduler we will charge the system for it. This means we have a rather crude accounting scheme in which we charge people for elapsed time essentially by charging a user 1/60th of a second if his program happens to be around when the clock interrupt occurs. It's obvious that over a period of one second this isn't going to be very accurate. But over a period of an hour it's going to be pretty good, because all the inequity is going to even itself out. Sometimes you get charged while the system is actually doing somebody else's interrupt routine, but other times it's doing your interrupt routine and somebody else is charged, so that it evens itself out pretty well.

Q If we are processing when this happens then we are charged for it but if we're in any dormant state...

A We're not, because TJOB is set to somebody else. However, I would like to point out that it's possible that the work that is actually being done is not work for you, because it may be some interrupt routine. In that case you get charged anyway, but it's fair because some other time somebody else will get charged for work that is set for you.

Q We change TJOB every time we change programs?

A Exactly, but not for interrupt routines.

Q Is there a header block or something set up some place which contains information relative to a job?

A Yes, there are lots of things. We'll get to that too. The tables in the system are essentially indexed in three different ways, by PACPTR for tables which contain information which is different for different processes; by teletype number for tables which differ for different teletypes; and by job for tables which differ for different jobs. The teletype tables and the job tables are quite closely tied together and we'll look at them in detail when we look at the teletype logic.

The last thing we do before leaving this program is a BRM DPT. This is a routine to make sure that everything connected with the drum I/O is all right before we switch processes. It is in the drum package.

Now we come to PACGO. This means, "Everything is all right, I'm supposed to find somebody to run." The assumption is that by the time you got to PACGO everything from the last process is cleared up. There are a couple of ways to get to PACGO other than going through all this logic for dismissing the previous guy.

We are going to keep in the cell called PACLVL the number of the queue that we are working on now, and that initially is QTI. The code that follows is going to run down QTI and when it finds the end it is going to know from PACLVL which queue it's working on. Thus it's going to be able to go to the next queue. During the scheduling process we are going to keep in PACPTR the address of the process for which we are currently trying to decide whether it is okay to activate it or not. This PACPTR is going to move down the various processes which occur on each queue. It starts out pointing to the header word.

The main loop starts at PEST and it goes down to the comment "activate program". If we get down to PACACT it means we have found an activatible program. If we don't get down to PACACT then we will just run around in the loop indefinitely. We pick up PACPTR, i.e. the program that we have been looking at, and now we are going to pick up the next program in the queue. The pointer to the next program in the queue's stored in a word called PNEXT in the PAC slot. We look to see if PNEXT is negative. If it's not negative then we have come to the end, since normally PNEXT is a (negative) PACPTR. We go to handle new queues at PACNXQ, which is at the bottom of the page. If we have a new PAC slot we save the old PACPTR in a word called PPREV, and we put the new PAC slot in PACPTR. Now we pick up PTEST for the new PAC slot and we get its activation condition into X. We use the activation condition to index a list called CACLST (which stands for complete activation condition list) which essentially contains a bunch of transfers off to the little routines which test the various activation conditions. We put the word that we get out of CACLST into T.

Each one of these little activation condition routines expects to find certain things in the central registers when it is called: PACPTR in X and the word addressed by PTEST in A. The activation condition routines are on page 5. CACLST is about the fifth line on page 5. It refers to activation routines 0, 1, 2, 3, 4 and 9. If the activation number is 5 we will always activate and we go to PACACT. If it is 6, 7 and 8 we never activate since we go to PEST which is the scheduler loop. Most of the CAC routines defined in a rather obscure way with this macro called CACR which is defined in MDBG The result is always to go to PACACT or PEST, depending on whether the activation condition is satisfied or not.

If we get to PACACT we are going to have to actually activate a program. The code to do this runs from PACACT to PACNXQ. Note that the instruction just before PACNXQ is BRU* 0. This is the instruction that starts up the program again, if we get that far, but very horrible things can happen before that, as we will now see in detail.

We pick up PACPTR and we go to this routine called PGET which is responsible for getting the program into core. This is actually a call on the swapper. You will find the routine on page 7. You can as-

sume that after PGET has been called the guy has been brought into core and can ignore the fact that the swapper may have gone through horrible gyrations to get him into core.

PGET skips if successful. If it's not successful that's because you have a drum error, in which case we abandon the attempt to run this guy and go back to PEST to try to run somebody else. If we are successful in bringing this guy in, we change his activation condition to 7 with an address of 1, which indicates the guy is now running in case anybody should happen to look at the activation condition.

Q Do you give any indication that a drum error has occurred.

A The assumption is that the drum error will be corrected later. If the drum error is irretrievable, then the guy will never get in, but there's this little counter that counts drum errors, and it will start counting up rapidly. We can look at it and try to figure out what's going on.

We save the old value of the activation condition in FK04, which we don't have to worry about for the moment, and we then have to take the program off the queue which it was on, because it is now about to be run and must not stay in the queue structure. If it does stay in the queue structure then as we keep on dismissing it more and more copies of it will get into the queue structure and eventually the thing will clog up. It is taken off the queue by the routine called QGET, which expects to be called with the address of the previous thing in the queue in the X register. We saved that address away in PPREV, you may recall, back at PEST. The address of the thing itself should be in the A register.

Now we pick up PACPTR again and get the quantum word. Remember, we put the quantum word away back on the previous page. Now we undo what we did there and put QR in TTIME. That's the long quantum word. We now set up the short quantum word, which is always NSQ, the current standard value for the short quantum. That is a parameter which occurs in MDBG at the bottom of page 3.

Q You always give the user a full short quantum.

A Right. This is on the theory that if we went to the trouble of bringing him in, he should be allowed to run for at least a short quantum; otherwise it's just too expensive to bring him in.

So we set up TIME, then we set up UMTF, which is a magic cell which says essentially whether user mode trap is enabled or not. Then we pick up T. What's in T is what was left over from PEST, namely, the address of the word in CACLST which was determined by the activation condition. If you look at CACLST on the next page you will see that most of the opcodes are zero. But some of them are non-zero, in particular the one that says 1 PACACT. What we are going to do now is transfer on the opcode field of the CACLST word. We add =ACTLST+1, and transfer depending on whether the opcode field of the word in CACLST was zero or one. The significance of this is that if the word was one that meant the activation condition really said, "An interrupt occurred for this process." And for that we have to do a very special thing. If anything else happened we go to PACSRT, which is just the next instruction on page 4 after the branch off. We are allowing with this device for the possibility of having some funny kinds of activation conditions which cause the program to be activated in a different way. At the moment the only one that's in there is this one for interrupts. Now we pick up the program counter and store it in 0, we restore the overflow bit, pick up the central registers from PA, PE and PX, and branch through 0.

Q Have you any idea how long it—do you have any statistics on the amount of processing time associated with activating the programs.

A It's less than 5%. Definitely. Except when swap time begins to swallow things. This is very bad. The thing that's slow is the swapper, which has to decode the relabeling. At the moment that's not done efficiently at all. We hope to improve that, but that's much the slowest thing.

Q All the housekeeping and all the scheduling and all that business is less than 5% with the exception of the swapping.

A No, the whole thing is still less than 5%. But most of that overhead is the map computation. In other words, when the swapper writes somebody out it gets his pseudo relabeling and has to figure out what the absolute relabeling is and what should be read onto the drum and set up the drum command list.

Q All this is included in the 5%?

A Yes, definitely. The scheduler is negligible.

Now the only thing that remains is PACNXQ. This was where we went when we ran off the end of a queue. What we do is pick up PNEXT, and test to see whether it is QQE or not. What PACNXQ does is the following: all it does is check to see whether the queue that it is going on to is QQE or not. If it is, we reset ACTR, that magic counter that tells us whether we have somebody to run on the higher queues. Then we go back to PACSCN, which picks up a new queue and goes cruising on. That is the scheduler. You have now seen it.

Q Are interrupts disabled here?

A No. An interrupt routine never tampers with the scheduler.

Q You get interrupts in this code?

A Yes. The only thing the interrupt routine can possibly do that the scheduler cares about is to change some word that's being looked at by the activation routine. If the interrupt occurs before the activation routine is called, then, that's fine. If it occurs after, that's too bad, but we'll get it the next time around. In any case it doesn't make any difference.

There's one more piece, the piece that actually causes program interrupts. Maybe I should explain briefly about program interrupts at this point so we can understand this piece, as its a fairly simple idea. You will find it described in section 4 of the working document. Every process has software implemented interrupts associated with it and there are 20 of them. When one of these interrupts occurs, the process is forced to execute an SBRM * 200 + interrupt number. PACINT is doing the SBRM *.

The next topic is going to be the memory allocation system and the swapper, and we will operate on amore general level than we did yesterday. We will not look at code. Before we go into this I want to say a few words about this thing called the "phantom user." This code you will find starting on page 5 of SPAC . There are places within the system where you are sitting, for example, in an interrupt routine and you discover you have to do something very complex. It may even be something which requires a swap. You cannot just sit there in an interrupt routine while the swap takes place, because the swap will take a long time. It may take several hundred milliseconds if there are other people waiting to be swapped. So this means that the interrupt routine has to have some way of putting some information somewhere which says "do something as soon as you can but not with such absolute top priority as an interrupt routine." Is this idea more or less clear?

For example, one of the things that started it all off was the handling of the rubout button on the teletype—a panic condition which causes all kinds of terrible gyrations in the system: information has to be read from one part of memory into another and all kinds of other things have to happen. Furthermore, the person that they are happening to is probably not in core when the rubout button is pushed, because anybody could be in core and there's no reason why this particular person should be. This means that the reaction to the rubout button is a rather complicated one which is not related to the program which is currently being run and, furthermore, which is too complicated to do in an interrupt routine. Of course,

there are lots of actions which are not related to the program that's currently being run. In fact, everything that's done in interrupt routines is like that. Most of these, however, are fairly short and are just handled by interrupt routines right away. For instance, when a teletype character comes in, a small amount of processing has to be done which is completely disconnected from whatever happens to be going on the instant the teletype character

This works fine for anything that can be handled in less than a millisecond, but for anything complicated it works very badly because you can't just sit there in the interrupt routine tying up everything else. Hence there has to be some mechanism for getting system functions performed with high priority but not in the interrupt routine. Furthermore, it seems desirable not to try to perform these functions within the framework of the scheduler.

In theory what you could do is say, "Okay, let's make a new process to execute this rubout, and let's give it a PAC slot and put it on some queue and eventually the scheduler will get around to running it." This isn't such a bad idea; in fact we considered it very seriously. The major objections to this idea are two: first of all, you use up a whole PAC slot for this and have much more space than you really need, because a PAC slot has room in it for all kinds of information that's completely irrelevant to just getting the simple job of processing a rubout done. Secondly, this involves going through all the rigmarole of the scheduler, which is pretty complicated and time consuming, and this is something you'd like to do quickly and efficiently.

In order to avoid these problems we set up this thing called the phantom user. The phantom user consists of one queue of things to do and this queue consists of a number of small entries. We'll discuss the form of an entry in somewhat more detail when we start talking about the W buffer logic. Each one essentially specifies an activation condition and someplace to go and one or two parameters. There is a permanent entry on the front of QTI for the phantom user, so that he is run every time the scheduler is entered. If there are any entries on his queue, he is supposed to run down it and get all the processing that is specified by those entries done.

Perhaps it would be helpful if I gave a couple of fairly specific examples of just what's involved here. One example is the rubout example I gave you before: the rubout arrives, the teletype input interrupt routine observes that it is a rubout—it tests for that explicitly—and having found that it's a rubout it generates this phantom user queue entry which says "This is a rubout. It came in on teletype such-and-such." (that's the parameter) The entry also contains the address of the rubout processing routine. The activation condition is "always activate" since you can process the rubout right away. Then the teletype input interrupt routine goes away. Now the next time the scheduler is entered it will observe that the phantom user has something to do and it will start it up.

The phantom user code is at PUSCN, at the bottom of page 5. The first thing it will do is to pick up this entry corresponding to the rubout and go to the rubout processing routine. That routine computes and figures out what to do, and it may have to read in some of the user's memory in order to leave some information for him about what happened. When it has to do that it will generate a new phantom user entry, the function of which is essentially to say "I'm waiting for a swap to finish, a swap which is going to bring in this memory on which the rubout processing routine is going to do some work, and when the swap is finished fire me up again so that I can do what I wanted to do with that memory." So then the rubout routine goes away again, and more time passes. When the swap is finished, the rubout routine is reactivated. Then it finishes the computation and exits. So what the phantom user really is then is a scheduler in miniature. You will notice that it really is true that we could have handled this problem by just making a new process and running it through the regular scheduler; the reasons for not doing that are the ones that I outlined.

Another example is the following: suppose that we have a W buffer (a TMCC) which has the card reader and printer on it. Now the channel is such that you can only run one device at a time. Suppose then there are two people that want to use it—one wants to use the card reader and the second wants to print. There has to be some mechanism for sharing the W-buffer. Several mechanisms have been tried in the system, one after another, but the one we have settled on at the moment is essentially the following: we proceed as though there were lots of W buffers and no possibility of conflicts, until we get up to the critical point where we are actually about to activate the channel. Then we test to see whether the channel is available. If it is not available—(by this time the system has committed itself irrevocably—it cannot use the scheme which it normally uses of going back and dismisses the guy until channel is ready and then re-executing the instruction which caused everything to start)—we make a phantom user entry. This entry has an activation condition which says "Wait for the W buffer to be ready", and the address that it wants to go to is exactly the place where the W buffer is about to be fired up. The parameter essentially says what device is involved. So as usual there is an activation condition, there's a starting address and there's this one parameter in the phantom user. Now sometimes W buffer requests will wait only a little while. For instance, if a card is being read the W buffer will be ready within 100 milliseconds or less.

On the other hand, if you're waiting for a tape rewind it may take several minutes, and at frequent intervals during that several minutes six or seven instructions will be executed which cause the system to check for the tape rewind to be finished (this is necessary because a completed rewind does not generate an interrupt)

Q The phantom user is primarily associated with I/O problems?

A Almost entirely with I/O problems. There may be teletype problems, there may be drum problems, there may be problems with other I/O devices. One of the things the phantom user is used for a lot is the thing I indicated above. Namely, if you get deeply involved in some process in the system and you discover you need to swap, you just have to sort of hang there and wait for the swap. Rather than tying the whole system up while the swap is going on, you put this phantom user request on so you get yourself restarted, when the swap is finished.

Memory

It is to be hoped that everyone understands about hardware relabeling. Otherwise life is going to be confusing. We are directing our attention to Section 5 of the working document. Section 5 is rather compact and makes it rather less clear than it might be what's going on, so we will talk at considerable length.

The idea of relabeling is the following: there are many places in the system where it is necessary to have some record of what the state of the user's memory is, and what that really means is, what memory will be in the hardware relabeling when I run this user. But even that does not state the whole question for the following reason: what will be in the hardware relabeling essentially tells me what real physical blocks of real core memory will be assigned to the user when I run it. That's not what I really want to know. What I really want to know is what will be in those blocks, because at the time when I look at the record the user may not be in core. The only place where any record of what was in his memory exists is on the drum. At the time when he is on the drum I can specify what's in his memory by pointing to certain drum blocks and say "Look, page zero is supposed to have the contents of that 2000 word sector of the drum in it and page 5 is supposed to have the contents of this other 2000 word sector on the drum."

I understand intuitively that there is some meaning attached to this statement, "Look, here's some of this guy's memory." There's something there that does not change, and from the user's point of view he

really doesn't know about swapping and he really doesn't understand about pseudo relabeling; he just knows that he's got this 16,000 words of memory and he expects it to still be there 2 milliseconds from now or 2 seconds or 2 hours from now. So somewhere, even though all these fantastic gyrations are going on, and words have been flying back and forth between the drum and the memory at 500,000 per second, there has to be some way of getting a handle on memory and saying, "Look, that block in the user's memory is the one I want and I want to be able to identify with some number that's still going to be the same number an hour from now."

So this is the problem. There is 2000 words of the user's memory which exists in some form somewhere, but the form in which it exists and the physical location of it will be changing from one second to the next. However, most parts of the system are not really interested in the physical location of the thing or anything about it except that it has some individual existence of its own. They only need some way to refer to it. And that's the function of pseudo relabeling, to give you some way to refer to each piece of the user's memory. Because we have this paging hardware, the most reasonable units to use would seem to be 2,000 word pages of user memory. We could, of course, divide the memory up into any size units we wanted. We could, for instance, say we will refer to memory only in 16,000 word units. This is not satisfactory for a number of reasons. First of all, users don't always have 16,000 words of memory. Secondly, the 16,000 words may not be all in one place, they may be scattered around, some of them may be on the drum, some may be in core. When it is in core it may be all scrambled, because logically it exists in 8 independent 2,000 word pages and they can be scattered anywhere we want because we have this arbitrary mapping defined by the hardware relabeling. Furthermore, from the point of view of utility we would like to have a scheme that's more general than just being able to refer to 16,000 words of memory. Since we do have this hardware facility for breaking the memory up into pieces it seems that the user can make some use of the ability to manipulate his memory in independent pages of some reasonable size, like 2,000 words, and that he would prefer not to be constrained to just thinking of his memory as a 16,000 word monolith.

Somewhere in the world there's a table called the pseudo memory table (PMT). It essentially defines the user's memory. It has entries which are numbered starting at 1. Each entry in the pseudo memory table specifies one 2K block, where by "specifies" I mean it contains a pointer to some place where that block is now located. This is the sole connection between all parts of the system that are not directly concerned with memory allocation and the real world. The PAC table, for instance, contains the user's map. That means it contains two words of pseudo relabeling, 8 numbers which refer to 8 entries in this table. This table is the only connection between that PAC table map, which after all is intended to define the user's program, or to serve as part of the definition, and the real world. Without this table I have no way of getting from the PAC table's statement of what the user's memory is to the real bits that the user actually put there at some previous time.

Each entry of this table contains several status bits and either a core block number or a drum address. There's a bit that says whether it's in core or on the drum, another bit that says whether it's read only or not, a bit that says whether it's executive or not, and one more bit about which we will defer discussion for the moment.

What you've gotten so far is a very systems oriented view of this thing. The user will not think of pseudo relabeling in this way, but this is the way it really is, if you like. This is really the guts of it, and if you understand this it will be easier to see all the different ways in which the mechanism can be used. So there is this table which defines the user's memory. Its length is a system parameter. Each slot contains either 0 or a word which consists of certain bits describing the block and a pointer which says where it is. At times which are essentially arbitrary with respect to anything which the rest of the system does, the swapper may come along and pitch the block out of core onto the drum. It will change the bit that says whether it's in core or on the drum and set up the drum address. Or it may come along and

switch the block from the drum back into core, in which case it will set up the core block number and change the bit back, to indicate core rather than the drum. The swapper acts independently of everybody else, and the only way in which other people have of knowing what the swapper has done, is to come and look at the core/drum bit in PMT. The only part of the system that ever fiddles this PMT entry is the swapper.

So the function of this table is to provide the rest of the system with minimal handles on memory. You notice, by the way, that PMT does not at all define what the user's map is going to be at any particular moment. In order to define that you must have pseudo relabeling registers which specify for each of the 8 blocks of the user's virtual memory which of the blocks defined in PMT is to be used.

Q Where is PMT.

A It's in system core.

Q Is it associated with each process?

A No, with each user. There may be many processes belonging to the same user and using the same PMT.

Q It is always kept in core?

A Right.

Q Who owns a PMT?

A Each user who is currently entered, each job. Each job the system is running, each entity which it knows about for accounting purposes, has a PMT. This job may have lots of processes all working on the same PMT.

How can we put together real memory that we can run a program inside of? Well, to do that we have to specify relabeling registers. Now when we discussed the hardware relabeling we pointed out that the essential function of these 8 relabeling registers was to specify some combination of 8 blocks of real core, out of the 32 that are available, which are going to be in the virtual memory of a running program at some particular time. That we could, for instance, say, "Okay, we're going to put real block 23 into virtual block zero, we are going to put real block 13 into virtual block one, we're going to put real block 23 again into virtual block two and leave the others blank. The pseudo relabeling registers are exactly the same thing except that instead of referring to real core they refer, of course, to entries in PMT. This means that there is a considerable gap between a set of pseudo relabeling registers and anything which can run on the physical hardware. There is a rather complicated part of the system called the swapper which is responsible for producing from a set of pseudo relabeling registers some real relabeling registers that we can actually put into the hardware relabeling. So it's really a long way from this 8 bytes of pseudo relabeling to anything can actually be run, and a lot of involved code has to be executed to make that happen. The situation we have now is that there's some process which has some memory, and each block of that memory exists as an entry in PMT.

Suppose the scheduler has decided that it's going to run the process that has this particular map. What it will do then is to call the swapper, delivering to it this map, and it will ask the swapper, "Please set up some real relabeling registers which will allow this process to run." What is involved in doing this? First of all, you have to make sure that all this memory is in core. If it isn't in core the process can't run because it can't actually physically address the memory unless it's there. So the first thing the swapper must do is to look, for each one of these bytes, at the core/drum bits. If the block is on the drum, it must find an empty core block to put it into and must set up a drum read command which will cause the block to be read from the drum into the empty core block. It must change the bit from drum to core and

set up the core block number in the PMT entry. It's then known what core block the memory is actually physically present in. This is called swapping.

To get this empty core block it may have to evict somebody else, which it will do by finding his PMT entry, selecting a space on the drum to write the block out into and again changing the bits appropriately so that a record is left of where the evicted block was put on the drum.

When the swapper is finished with this job it is known that all of the blocks of memory in the map are in core. That means that each PMT entry specified by a pseudo relabeling byte has an address number which tells us where in core the block is. We can now proceed to step 2, which is to construct real relabeling which contains the real core block number which corresponds to each one of the PMT indexes in the pseudo-relabeling. At any given time, the swapper keeps around real relabeling essentially for one process, namely, the one that is currently running. That's not quite true, but almost true.

So when you call the swapper it does all these horrible things that are required in order to get the memory into core and construct the real relabeling. Then it puts that real relabeling in two fixed registers where the system can always find it. Furthermore, it POTs this real relabeling out to the real relabeling registers, the hardware registers. This means that when the swapper returns to the scheduler the scheduler knows the relabeling has been properly set up so that this user can run, and if it now sets up the central registers correctly and transfers to the location specified in the PAC table entry for this process, the process will then proceed to run and will, if everything has gone properly, be unaware of the fact that it has been dismissed.

Q How can you be sure when you go to pseudo relabeling registers and cheek the PMT to find out whether a block is core that it will still be in core when you have read in the missing blocks?

A That has to do with the more detailed mechanisms of the swapping process itself which I am getting to.

Q What's the maximum amount of memory a job can have?

A A job can have as many processes as it wants. The processes are completely independent of the memory. If you insist on having an independent 16K for each process then you can have no more processes than there is room for in PMT, which at the very most would be 4.

Q That is the limitation on the number of processes?

A If they are all to have a full 16K of completely independent memory—the limitations are actually more stringent than that, as we will see a little later, but I'll explain exactly how it works.

Q Does the swapper always read 2K blocks?

A Yes because the bottom 11 bits the relabeling doesn't tamper with at all; they could be anything. From the point of view of the system one page is 2000 words, but whether or not you use all those words is up to you.

We are now going to discuss in a little more detail the precise mechanism involved in making swapping take place the way you want it to. Most of the stuff will be basic and will not change. Suppose now that you're the swapper and you were just given some pseudo relabeling and you are going to see to it that this memory gets into core, which is, after all, the primary function of the swapper. Now you as the swapper have the following resources at your disposal: the portion of the drum reserved for swapping, core minus the part used by the system (which is probably 8K), the PMT tables, a table called RMT, (which stands for real memory table), and finally a table called RMC which is the lock table. These last two tables are the same size as the amount of available real memory. Each one of them contains one entry for each block of real memory. RMT tells you what each block of real memory is, at this moment, as-

signed to, i.e., it tells you what user and what PMT entry for that user is currently occupying this block of real memory. This means that the swapper can look at RMT and tell exactly what is being done with each block of real memory at the moment. RMC is a table of counters, one for each block of real memory. Each word of RMC contains a number which tells how many different things have attached this real block.

To explain what this means I'm going to have to tell you exactly when this counter is incremented and decremented. It is incremented once for each pseudo relabeling byte which refers to this real block and belongs to a process which is currently regarded as being entirely in memory. There may be several processes at one time which the system thinks of as being entirely in memory. Once the process is read in the system keeps track of how long it remains completely resident—the system remembers the process until it has to write some of this memory out, at which time it essentially regards the memory as being pretty much lost. Of course, the PMT word will still show the memory in core, so that it will not be read in again if it is needed, but there will be nothing to keep the swapper from deciding to abandon it. So the system remembers about processes which are entirely in memory. The exact decision about what to remember is one which I will discuss in a minute, those parts which are not changing too much. This means, you notice, that one process may cause the counter to be incremented more than once. In particular, a pseudo relabeling register might read 06061400,06000000; in other words it might refer to block six 3 times. This means that if this process is entirely in memory the RMC word corresponding to block six will be incremented 3 times. Correspondingly, the location will be decremented for each byte of pseudo relabeling belonging to a process which ceases to be completely in memory. This means that when the swapper brings the process in it increments the RMC word for all the bytes involved and when it throws it away it decrements those words.

Q It throws away only one page in the process?

A It releases all the pages. If the page hasn't actually been clobbered, the record of who it belongs to will still be in RMT; but at the time when it gives up its hold on that process, the swapper decrements all the counters. If it didn't do that, the problem of figuring out when to increment them and when to decrement them would become hopelessly complex.

In addition to this mechanism one of these RMC words can be incremented by any part of the system which wants to make sure that a block of memory does not get swapped out. For example, when drum I/O is taking place the drum I/O itself takes place independently of the process that started it up, which gets dismissed while the I/O is taking place and in fact may be mostly swapped out. However, if I/O is taking place from the user's core instead of through a buffer, which it sometimes is, this means that the core which is being read into or written out of obviously can't be swapped out, because if it is then there is no way for the drum to get at it. Therefore the drum I/O routines will increment the RMC word for each block of memory which they are actually using for input-output. When they are finished using it they will decrement it.

Another thing that could happen is that a user, if he is properly authorized, can specify that he wants a particular block to remain resident. What happens in this case is that the RMC word is incremented. When he says "I don't want it anymore," it is decremented.

The significance of this is very simple. Any RMC word which is negative corresponds to a block which doesn't have to be in memory. Any which is positive corresponds to a block which somebody expects is still in memory. Before you throw it out in the swapper you had better make sure that whoever expects it to be in memory stops expecting it. Zero corresponds to one use of the block, and one corresponds to two uses so that the RMC word will be conveniently negative when nobody is using the block. It's easy to test for negative with SKN.

So now suppose that I'm the swapper and that I come along and discover, by looking at pseudo relabeling registers I've been given, that I need three blocks of memory. What I do is to go running down RMC looking to see if there's three blocks that aren't in use (i.e. RMC word negative). I pick off unused blocks as I come to them and set up drum commands to read into these unassigned blocks. Maybe I find three blocks by the time I get to the end and maybe not. If I have found three blocks then all is well. I fire the drum up and it starts to read, and I'm finished. After a while the drum will finish and the memory I wanted will be in core. Now this may not happen; there may not be enough unused blocks. What this means is that I have to go around through the processes that I think of as being resident in memory and abandon some of them.

I don't want to abandon just any one. For example, the swap may be initiated sort of as an afterthought just to keep the drum going while some other process is running. In that case the process that's running obviously should not be thrown away. But under normal circumstances, with a reasonable amount of core, there will be several processes resident at one time. Some have just finished running and some are just about to run and one is really supposed to be run. I will go through the list of processes that are available and find one that seems satisfactory. Then I will go through the pseudo relabeling, decrementing RMC words for all the bytes. Now I can make another scan through RMC.

Q How do you decide which process to abandon?

A That is a complicated, tricky question which is not really decided. But the idea on which we are currently working is the following: the way the swapper will function is that it will swap somebody in and then it will look ahead and on the basis of what it sees in the queues it will set up swaps for as many other people as it can find room for. Then it will run the guy that it brought in first, so that while he's running some other people will be coming in. This is what you might call look-ahead. It will try to read in people that are going to be the next people to run, while it is running the guy that it has now. The pool of available space in real memory will consist of those processes which I have finished running but haven't yet thrown away. I keep them around in case one comes up for running again. Then I can run it right away without throwing it out. So the situation will be this: there will be a buffer with pointers to the current processes, the oldest one still around, and the one most recently read.

See Figure 6

Pointer 2 refers to processes that I have run that I'm now throwing away. Therefore when I need more space I pick it off the process pointed to by this pointer and move it up by 1. When I catch up to pointer 1, I have to stop reading because this is the process that's currently running. When I read a new process in, I move pointer 3 and put the process in at that point. This is very schematic, and of course this diagram really isn't like this because the ordering of things is very much confused by the priority logic of the scheduler, but this is the basic idea. You try to look ahead and you throw away processes from behind.

Q Will you have a table of processes in various queues in the monitor someplace?

A Well, you have those queues themselves. The swapper maintains its own tables. The question of whether a process is in memory is really independent of its relationship to the queues. So the swapper maintains its own tables of processes that are in memory and the way it decides who to tack on for new things is to go and look at the scheduler queues.

This is the basic idea of how the swapper logic works. You notice that this lock table is a very important component of the whole thing. What it means is that some other part of the system can reach out and say "lock that block" without having to know too much about the internal workings of the swapper. If the W buffer, for instance, wants to do I/O directly out of some user memory all it has to know is that

it's block 16, and it goes and looks at word 16 of RMC and increments it by one. From then on the swapper will be prevented from using that block until the W buffer decrements it. This of course imposes the danger of forgetting to decrement the counter. In this case the block will get lost. This swapper will come to the delusion that it can never use it.

Q Why does this incrementing lock the block?

A Because it guarantees that the counter will never go negative, so that the swapper will never choose to write it out.

Another possibility, suppose there are two processes both using the same block. This means that the block will not get swapped out until both processes have been abandoned. In the case of something like DDT where there are maybe six or eight processes using, it becomes extremely unlikely that the DDT blocks will ever go negative, so they will never get swapped out at all, which is what you want because there's a lot of people using DDT and it's very desirable to keep it around.

So this is a mechanism that serves two purposes: one is that it makes it easy for other parts of the system to force blocks to become resident without having to get involved in the intricacies of the swapper; the other is that to some extent it makes it likely that blocks that you want to keep around will stay around; that is because if a block is used a great deal it will be in the pseudo-relabeling of many processes and this means that it will tend to be incremented most of the time. Only the case where all the processes become inactive will it ever get down to minus one.

Now, there is one mechanism which I mentioned above and which is described in detail in the working document: it allows you to say "I want this block to become resident." That block's RMC word will be incremented. You have to release it explicitly. This mechanism is dangerous, since if too many blocks become permanently resident the system will become choked up.

I will now proceed to introduce a further complication which is described in the working document. In addition to PMT there is another table called the shared memory table (SMT). Its function is essentially to keep track of blocks of memory that are used by more than one user. If you have several processes belonging to the same user, they share the same PMT and no special machinery is required for them to share memory. If the several processes belong to different users they have different PMTs and some new mechanism is necessary if they are to communicate. This is the SMT table, which has essentially the same format as PMT. It is divided into two parts; the entries from 1 up to NCMEM-1 (at the moment NCMEM is equal to 48, but that is just a parameter) are one class, and the ones above that are another class. The words have exactly the same format, but the mechanism for getting at them is quite different. A pseudo-relabeling byte is interpreted in the following way (which is different from what I said earlier). You look to see if it is bigger than NCMEM or not. If it is not then it does not refer to PMT at all. Instead it indicates a word of SMT, one of the words between 1 and NCMEM-1. If it is bigger than NCMEM then it refers to PMT. We see that every user thus has available to him NCMEM pages of permanent shared memory. This means that the maximum amount of memory available to the user is 64-NCMEM pages. You don't have to make NCMEM equal to 48. You can make it anything or whatever you want, within certain restrictions.

Q Suppose you made it 0.

A You had better not make it 0 because that will foul our system up something awful. You can make it 10 and get away with it. The reason you can't make it 0 is this: that the function of this part of SMT is to contain the pointers for various blocks of memory which are important to the system, namely for all of the swappable portions of the system proper and for the important subsystems like DDT, QED and some others. Of course, these blocks are read-only.

Q What you do here is set the value of NCMEM to a smaller or a larger value depending on whether or not there is a large amount of interaction between programs and system.

A That's right. I want NCMEM to be as large as possible subject to how much memory I insist on having available in PMT, because there are lots of advantages to having NCMEM large. Besides allowing me to share things conveniently it cuts down on the size of all the PMTs. You only have one SMT, but you have lots of PMTs—one for each user. There is no sense in allocating a lot of space for them if you don't really need it.

Q The fact that NCMEM is set at 48 means that I have two processes with independent 16 K memory.

A Then you can only have two. Right. Why you would want two processes with entirely independent memories is not clear to me.

Q The reason for wanting to have more than one process is because you want to have a large program, a 64 K program, and you want to operate this in time sharing as a complete independent program unto itself. It doesn't use anything else, it's a real big large program.

A It will be one process in that case. Remember the memory doesn't have to be assigned to any one process at any one time. It is possible to have blocks of memory which are in PMT but not in anyone's pseudo-relabeling.

Q I don't quite get you?

A Well, you remember them and you put them back in your pseudo-relabeling when you want them. We'll discuss that in more detail later on.

So we fill up the space in SMT below NCMEM with as many of the common subsystems as we have room for. The other common subsystems are handled in a slightly different way. Notice that this provides an automatic mechanism for sharing NCMEM-1 blocks of memory. No matter how many people want to use DDT, there only has to be one physical block for the drum and there only has to be one entry for DDT in SMT (per block). There may be subsystems which won't fit in this space, in fact there almost certainly will be as the system expands. We still want to be able to share them: if there's a re-entrant sub-system and two people are using it, it is ridiculous to have two copies of it.

So you have to have some way of addressing more of SMT than the first NCMEM entries. The answer to that is very simple: we go back to PMT and remember that a PMT entry contains either the absolute drum address or the absolute core address of a block. There is a third possibility: the PMT entry may also contain a pointer to SMT. And then it is possible to have two PMTs which contain pointers to the same word of SMT, two different users, two different jobs pointing to the same shared block. That block could be either a subsystem or some data block that both are working on.

Q Are there two ways of getting to the SMT table?

A That's right. Anything below NCMEM can be gotten at directly by simply setting the pseudo-relabeling byte in the correct way. Anything above NCMEM has to be gotten at indirectly. The reason for having all this elaboration in the thing is simply because of the restricted size of a pseudo-relabeling byte.

So much for SMT and so much for pseudo-relabeling and swapping. Except that we now have to discuss acquisition and releasing, and we have to discuss this question of re-entrancy. I think we'll take them up more or less in that order.

So far we have been talking about memory from the point of view of the system. The system has all these processes and they all have maps and they are all running like crazy. It has to figure out who to run

and just how to find his memory. Now we want to talk about this whole business from the point of view of the user. Now he is not interested in how a swapper works, but he wants to have some flexible method of controlling the amount of memory he gets. Also, if he is a novice he does not want to think about this at all, he wants this whole thing to be completely automatic if he doesn't choose to use any machinery.

Before we were discussing the system running full blast and two processes with pseudo-relabeling coming in and out. The obvious question is "How did it get that way?" And the answer to that question is more or less like this: When a user comes in and enters the system there's a special little routine in the monitor which initializes all the tables belonging to this user and makes one process for him which contains in its map the blocks corresponding to the executive and one block which is called the temporary storage block. Each user, each job has a single temporary storage block which is used to hold various things the system needs to know about this user, plus temporary storage which the various executive routines need while they are working for him, plus drum buffers. Those three things fill up the temporary storage block. The reason they do is that you assign enough drum buffers to fill it out. So by magical processes at the beginning the guy is started off with this single process which has a map consisting of the executive and one temporary storage block. Everything that happens from then on happens through mechanisms which are equally available to the executive and to the user. It's only this first initial thing that's magical. So that's how the first pseudo-relabeling gets established.

After that various things can happen. The executive can read in his program and fire it up, start up sub-systems, and one thing and another. One thing that can happen is this. The executive maintains for each user a pair of registers called the user's program re-labelling. It is just one pair of pseudo-relabelling registers which contains the memory which the executive and DDT think is the user's program at the moment. It starts out cleared to 0. If this is all zeros, it doesn't do us much good. How do we get something other than zero in there? Well, the user can ask the executive to read a program into this space from someplace where it's been saved. If he does that, the exec will grab enough memory to hold the program with the mechanism I'll describe in a minute. Now the user says (at the teletype) "Start me up at location 423." So the exec does so, giving the program the map which was constructed when it was read in, and the program starts to run. After a while it becomes ambitious and decides that one block is not enough and that it should have more.

Q The program decides this.

A The program decides this. Right.

The program is now running, the executive out of the picture. So what does it do? It wants more memory. Well, how does it get it? Remember, we don't want the novice to have to worry about all this elaborate machinery. We don't want him to have to call the system and say to him give me another block, explicitly. He thinks that he has all this memory, 16 K, and that's all there is to it. Fine. So, thinking that he has it, he addresses a little bit. He says `LDA 20004`. Well, what happens. `20004` arrives at the re-labelling hardware, which finds 0 for that page. PANIC. It generates a trap. So you go to location 43 or wherever it is the trap goes to. Now 43 contains a `BRM` to a system routine for processing memory traps. The system has to figure out what to do about this memory trap.

The most idiot thing you could do is to say that memory traps are a disaster. I will type out a message, and abort. And sometimes it does that, but it does a lot more sophisticated things than that too, because there are a lot of cases in which a memory trap is not to be regarded as a disaster, but as a signal that the user has asked you to do something, even if he's asking implicitly rather than explicitly by calling on the system. So what's the rule? The rule is this: the user process may be in one of three modes as regards memory acquisition. It may not be allowed to access any more memory at all. This means that a trap is a disaster. In this case the exec generates a memory panic and what happens to the memory panic

is something we will discuss when we go into forks. But that's a catastrophe. Now the second mode, which we might call the normal mode, is that the process *is* allowed to access more memory, and if it does it is to be given additional memory. What happens in this case is that the executive memory trap routine calls on the swapper and says "give me a new block". The swapper runs around its tables and if necessary writes some block out and comes back and says "Here, you have block 43". Then the memory acquisition routine runs around in PMT looking for an empty slot, that isn't being used, [say slot 14] and it plugs in block 43 in this empty slot. Then it starts this little loop that goes through block 43 and clears that to zero. Next it finds the user's pseudo-relabeling and puts 14 into virtual block 4, which is the one that corresponds to address 20004. It used to be zero, because it used to be unassigned. The user now has block 4 assigned, and it happens to be assigned to PMT entry 14. We can now go back and re-execute the instruction that caused the trap, and it will execute successfully this time, because the block is now there. The user can do this with all the blocks. He can, for instance, do LDA 30000 to get another block, and so on. It takes exactly eight instructions to grab 16 K of memory. Of course you've got to pay for the 16 K: it's going to take longer to swap you, and you'll probably get charged for it.

Q Once you get that memory it is swapped every time?

A Definitely. It's important to understand there is nothing in the system that is completely free. You can address 16 K, but you pay for it. This means it is to your advantage not to use 16 K. Actually for the simple-minded user who doesn't understand this mechanism, the memory allocation will normally be more or less optimal. He'll start with a little program and if his tables do grow larger he'll get more memory. If it doesn't grow larger, he'll stick with a small memory. Of course if the tables do grow larger, then get smaller again, he'll have to understand something about the mechanism in order to take advantage of that fact.

We'll start today by explaining the mechanisms for twiddling memory from the user's point of view. The user has for each process that is currently running, a pair of pseudo-relabeling registers. There are two operations which the user can use to read and set the pseudo-relabeling. He can do BRS 43, which reads the pseudo-relabeling into A and B; and he can do BRS 44 which takes the contents of the A and B registers and makes them the new pseudo-relabeling. Now, when BRS 44 is executed the system will check to make sure that what the guy is asking for makes sense, namely that he doesn't put any numbers in any of these bytes which refer in blocks that are unassigned in PMT, and also that he doesn't try to refer to any blocks which for some reason or other the system won't let him refer to. For example, he will in general not be allowed to refer to his own TS block. If he puts a number in one of these bytes corresponding to his own TS block, the BRS 44 will indicate an error, because he is not allowed to address his own TS block. It is reserved by the system for its own purposes, and if he changes it he can cause a lot of trouble.

Q How is the error indicated?

A It generates an illegal instruction trap. That is normally the way errors manifest themselves.

By the way, speaking of errors, in looking through the system code you will often find places where it does BRU TRAP. This is what the system does when it detects an error: going to TRAP has exactly the same effect as executing an illegal instruction at the location specified by the contents of absolute location 0. The reason for this is that what normally happens is that the guy executed a SYSPOP of some kind which left the link in 0. Then he goes off to the system, which discovers his error and just transfers to TRAP. If it does this without ever having changed 0 then 0 will contain exactly the location of the SYSPOP which caused the error. So there are many places throughout the listing where you will find the system will check for something or other, and when it doesn't find what it likes it goes transferring off to TRAP. The system when it goes to TRAP does exactly the same thing as it would have done if hardware

had seen an illegal instruction, because the hardware trap routine—the routine which the system puts at the hardware location where legal instruction traps go—does exactly the same thing: it goes straight to TRAP after putting the link into zero.

So there are these two operations by which the user can read and set his pseudo-relabeling. When he sets his pseudo-relabeling he changes his map completely. Suppose he starts out with a map in which he has blocks 3, 4, 5 assigned and the rest unassigned (relabeling registers 03040500,0). Suppose he's sitting at location 4500, that's in block one. Now suppose at location 4500 we have a BRS 44 and the contents of the A and B registers at that time are 06070800,0. When this BRS 44 is executed, the map at that moment will change from containing blocks 3, 4, and 5 in the first three pages to containing blocks 6, 7, and 8. When the system returns from the BRS 44 it will return to location 4501, which will now be in an entirely different piece of program from the one we had before, because location 4501 is the 501st location of block 7, whereas the location 4500 containing the BRS 44 is the 500th location of block 4. In most cases this will be a nonsense thing, to do.

Q It is in a different page?

A It's in a different page. The process is immediately executing a wildly different program. Normally you will not do this. Normally when you change your pseudo-relabeling, some of it will be the same as it was before. In particular the instruction which actually does the changing will be in a page which is left alone by the change. It doesn't have to be that way. It's important to understand that this is a possible thing to do, and it's not a priori unreasonable to do this.

In this way the user may acquire memory, read his relabeling to see what blocks he's acquired, keep a record somewhere of what blocks they were, and then set the relabeling to take those blocks away. Then later on he can reset it to put some of them back or to put them back in different combinations or anything he wants. Of course he can also lose track completely of which blocks he has acquired. In that case the blocks will have a slight tendency to be lost. The system of course will still know which ones they were, but the user will not have any handle on them, so he won't be able to get at them anymore. In this case what he can do is go back to the Exec and ask for a listing of the memory assigned to him, so that he can find out again what blocks are assigned, and maybe release them if he doesn't want them anymore. If he doesn't do that, blocks that have been assigned and later lost by him will simply pile up until all his memory is used up and then he won't be able to get any more.

Q He can only use BRS 44 with numbers in the A and B registers pointing to blocks which already exist in PMT?

A That's right, otherwise it doesn't make any sense.

Q What if he put a number in there that wasn't for one of his blocks? What would happen then?

A An illegal instruction trap.

Q In his BRS 44?

A Yes. Right.

Q How can the user cause entries to be created in his PMT table?

A That's the only way. That's just about the only way anyone could cause such an entry to be made. The only way to make an entry in PMT is to grab some memory.

Q He can get memory that way though.

A How?

Q Couldn't he get memory that way?

A By doing `BRS 44`? No.

Q If he uses a new block...?

A No. That's an error. It does not cause any new memory to be assigned. The only way you can get new memory is to address a block which at the moment is not assigned to you, to address a word which corresponds to a block which has a zero in the relabeling. That's the only way. You can attach system memory, namely that the memory corresponding to byte numbers < `NCMEM`. Even though you haven't explicitly got them in your `PMT`, they are always around and you can think of them as always being an appendage on your `PMT`.

Note that all this has nothing to do with forks. Remember, a fork is an independent process. It is something which is executing instructions independent of some other process which is also executing instructions. What we have here is one process which is taking instructions out of one block of memory and then switching for the next few instructions to another block. But there is nothing here any different from calling a subroutine, which doesn't create a new process.

Q Well, do you call this tasking or something?

A No. Tasking is the same as forking, and it is having multiple processes.

The way you can distinguish independent processes from anything else is this: independent processes are processes which would run simultaneously if you had two central processors.

Q Are the imbedded calls assigned...?

A What's an imbedded call?

Q A call to a routine of some sort in the Fortran system.

A OK.

Q Is all of this memory assigned to those subroutines during loading, or only when the call comes to be executed?

A During the loading, in all the Fortran systems I know of that either exist or will exist for the 940. It's not logically necessary to do it that way.

Q Well, they can do it in the GE 645 [the Multics machine].

A I know it's possible. It's not clear whether they do or not. In some systems the memory is assigned, even though stuff isn't there in core, when you do the loading. You can't call the subroutine unless it's been loaded even though the actual linkage may not have been established. You could have that here too, yes.

Q Yes. But what I'm trying to get at here is: is all the memory that the program is going to use assigned to the `PMT` during loading, except with regard to data areas? Is that correct?

A That is the way the Fortran systems that we now have work.

Q The assembler doesn't?

A No. The assembler has nothing to do with assigning memory during loading. The assembler just creates binary. When you load your program with DDT the way you normally do it is to load the entire program. But you don't have to do it that way. You can imagine a situation in which for some reason or other you constructed a system which didn't load all of its pieces all at one time, and when you called a subroutine you would check to see if it had been loaded. If not, you would call on the loader

to load it. That's a possibility, I don't think anyone's done that yet but it's certainly possible. IBJOB overlay works a little like that.

Q You could put an imbedded call in the program to get at the loader.

A Yes. You certainly could.

Q But the normal case is to load everything

A Right. Of course there is really no distinction between program and data memory. What is program memory to you is data to the loader, for instance.

Q Can I take a 32 K program, assemble it, and get it all on the drum and in my PMT?

A Yes, of course. How would you do that? You come into DDT and say ;T 'PROGRAM 1' (this is the load command), so that DDT, which started out with no program memory, grabs the memory the program wants. Now you look at this memory by doing BRS 43 and you find out that, say the blocks it grabbed were 4 and 5. Now you use the DDT command (or you execute the BRS 44— there's two different ways to do it) to clear those blocks from the map. Now you say ;T all over again with program 2, so that DDT grabs some more memory to hold program 2, say blocks 6, 7, and 8.

Q You have to do this yourself in DDT rather than have the monitor take care of it.

A You have to specify it in some way. The situation at the moment is that nobody has thought about the problem enough to figure out an automated way of specifying it. I can't imagine any simple arrangement would be satisfactory. The kind of memory operation available in this system is too limited. After all, when a program has once been loaded all the addresses are fixed, and although the 2 K block which exists is largely independent of other 2 K blocks, if the program was loaded so that it was supposed to run from locations 1000 to 1200, then that block had better be the second block of any map it occurs in. The program is going to have instructions in it like BRU 1014, and if you put this in block 4, it is not going to go into itself.

It is not like the 645 where everything is done with base registers and you really have complete freedom to put any combination of blocks you want in at one time. This means that if you want to be ingenious you have to understand your problem pretty well. This isn't as bad as it sounds because normally you don't go through all the trouble of doing overlays unless you have a complex problem which is worth some trouble, But the 645 can be a lot nicer, there is no doubt about that.

We will now proceed to discuss how to actually release a block of memory. Everything we've explained so far indicates that one can acquire memory and once one has acquired it, it is there for all time. I may not use it, it may not be in my pseudo-relabeling, but it's still there and if I hold onto the number, then having that number I can always put it back in my pseudo-relabeling anytime and get the memory back.

Q There is a permanent PMT entry?

A That's right, and there's physical memory on the drum or in core or wherever it happens to be, the bits that I put there a long time ago are there, unless the system crashes, indefinitely.

There is, however, a mechanism for throwing memory away completely, for pointing to a PMT entry and saying "I don't want that entry anymore, take it out." The way you do that is the following: of course you don't point to a PMT entry explicitly, what you do is you put into A an address, say 4000, and then you do BRS 4. What this does is to release the PMT entry corresponding to the page of your virtual memory in which this address appears. Since 4000 is an address in block 1, if the pseudo-relabeling is

01020410,16000000, this will cause PMT entry 2 to be cleared away, and the 2 here to be replaced by 0. Furthermore any 2's which appear in any other pseudo-relabeling corresponding to any other processes will also be replaced by 0. That block is gone for good. It's empty, free, finished, its contents are lost. BRS 4 is arranged in such a way that if you don't want to think about the possibility of changing pseudo-relabeling, you don't have to. It refers to addresses within the 16 K you now have. The way that is to be interpreted if you are thinking in terms of being able to change the pseudo-relabeling is that it refers to the current pseudo-relabeling. I am never required to point explicitly to a PMT block.

I want to fill in a few things I might have left out in connection with the swapper. One has to do with how you assign blocks on the drum and the other one has to do with the device called write-ahead.

The drum has bands, each with its own head. Each band has 8 sectors of 2 K words. The same sector on each band can be accessed at the same time, but only one band at a time. There's obviously some problem about allocating space on this drum. We are only looking at the part which is used for swapping. This problem has two parts: one is to allocate storage and the other one is to arrange that for each particular user the space is allocated in such a way you can read it in one revolution, or better in a minimum amount of time. In particular, suppose the guy has four blocks. You obviously want to put one of them in sector 1, one in sector 2, one in sector 3, one in sector 4. This means it only takes half a revolution to get them in. If you have all four of them in sector 1 (on different bands of course) it takes four revolutions to get them in, because you can only read one band at a time and you have to wait for a whole revolution before you can do the next one.

It doesn't make any difference which bands the blocks are in, it's only the sectors you're concerned with since there's plenty of time in the intersector gap to switch bands. So we want to be able to keep track of space on the drum, and secondly we want to be able to optimize it. The way we do this is with a bit table. Let's suppose for simplicity that we have 24 bands that we're using for swapping. So we have an 8 word table, one word for each of the eight sectors. A bit in this table is turned on if the corresponding sector is in use. There is one bit for each 2 K block: 24 bands and eight sectors in each band. Now when we're ready to write a guy out, we look to see where the drum is now, and we start writing in the next sector from the one it's in now, to give us time to do the computation; if the drum is sitting in sector 2, then we say we are going to put the first block in sector 3. We pick up the third word of the table and we look to see if there are any 0 bits. When we find a 0 we say "OK, we'll use that sector to put the first block in." So we turn that 0 into 1, and we record the drum address in PMT and set up a drum command; then we look at the next block that we want to write out. We put the first one in the third sector so we'd better put the next one in the fourth sector, because that way when we read it back in, it'll be right there and we won't have to worry. So we do that, we put the next one in the fourth sector, and we do that by picking up the fourth word and scanning for a zero bit. As soon as we find it, we set it to 1 and write out the second block. We keep on doing this until we've taken care of all the blocks. If we find no zero bits in one of these words then we just move on to the next word. That means that the process will be a little less optimal, because one sector will be skipped without any useful work being done. All this means that the writing out of this guy will be optimal and also the reading back in—if you read back in exactly the same blocks the reading will also be optimal.

Second point: the swapper has an algorithm that says, "If you have nothing better to do, find some block that's already in core and more or less inactive and write it out. Then make the block read-only so that if the guy stores into it again you know that he changed it. If he didn't change it by the time you actually get ready to get rid of it it's already written out and you don't have to write it out." That's called write ahead. The swapper does this whenever it doesn't have anything else to do.

Q Nothing else to do?

A That's right. It prefers to do things that it knows are useful. This write ahead is problematic; it might not do any good. The block might get changed before we actually get ready to write it out for real. If it doesn't get changed we gain something. Consequently we only do it when there's nothing else to do. It's not clear entirely that this is a good idea, although in the 940 it will certainly be a good idea because the interference is so small. In our system [at Berkeley] the interference between the drum and the CPU is not negligible even with the priority scheme because the drum is quite a bit faster, and so it's not clear that we really will want to do this.

Q What is the definition of when the swapper has nothing else to do?

A When there are no blocks to be read in or out, as determined by the algorithm that we discussed above. The swapper tries to do as much as it can. Another possibility is this: suppose the swapper knows it has to write a block at sector three and another block at sector 6, but has nothing to do with sector 4 and 5. It might as well find something to do in sectors 4 and 5 because the drum is going to be wasted during that time anyway.

So much for the swapper. We will now discuss how to write re-entrant programs for this machine. Now if everyone understands relabeling so well, there will be absolutely no difficulty in understanding something this simple. What's the idea of a re-entrant program? A re-entrant program is a program that does not modify itself at all. The program exists in certain pages of memory, and no word in any of these pages is changed when the program is running. So what is required for reentrant programs?

1. Do not modify instructions.

That's obvious. If you compute an address and store it in an instruction, you change the program, and that's not allowed because some other guy running the same program is going to compute a different address and store it. The result will be a mess, because the two users are using the *same* word.

2. Keep data storage in a separate page from the program. This means that it is necessary to collect all the temporary storage and see to it that when the loader does its loading, it is loaded in a different page. There are several ways to do this—one of them is to put the `TS` in an entirely separate package and make the load origin for that package the beginning of a new page. The other one is to keep the `TS` in the same package but have some algorithm with it that computes where the end of the program is and how far to go ahead before allocating space for it.

The first scheme is the one normally used, because the external linking facilities in the assembler and the loader are very good, so that there is normally no objection to keeping the temporary storage in a different package.

Q By different package you mean different process?

A No. A package is something that gets assembled separately. It has nothing to do with processes. Re-entrancy is always a question of memory. It has nothing to do with processes.

3. Do not use `BRM`, which is the ordinary subroutine call instruction. When you do a `BRM X`, it puts the link in location `X` and goes to `X+1` where `X` and `X+1` are going to be in a part of the program. That means that you change part of the program, which isn't allowed. What it means is that if you do a `BRM` and then somebody else comes along and does another `BRM`, he'll clobber your link. Instead of doing that use the `SYSPOP` called `SBRM`, which puts the link indirect through its address and goes to the address plus 1. So this means that the subroutine will start with the link storage word address. The code starts with the next word. The link will be stored indirect through the first word. `SBRM` is a cross between `BRM` and `BRM*`. `BRM*` would put the link in the same place that `SBRM` would put it in, but it would transfer to link word plus 1.

The return from the subroutine is not by BRR but by BRR*. There's a SYSPOP called SBRR which is exactly equivalent to BRR* and in fact converts itself into BRR*.

Q And it's a SYSPOP.

A Yes. The reason we put it in is that we thought we might want to change the linking scheme, so we didn't want to have a lot of BRR* instructions built into programs.

If you have done these three things you then potentially have a re-entrant program. We've arranged that everything that changes is in a different page from everything that is not changed. Say the program is in virtual pages 0, 1 and 2, which are PMT entries 7, 8, and 9. We have two processes that want to use this program. Well, it's not hard. Let one process have the map 07101100,20210000 (assuming that the program uses 20000-27777 for storage) and the other the map 07101100,22230000. Then the data is entirely disjoint between the two processes, and no unfavorable interaction can occur.

Fork structure

Today we are going to talk about fork structure, or multiple processes. The relevant sections of the working document are 3 and 4, where 3 really discusses forks and 4 is for miscellaneous goodies. I hope from the preceding discussion of the scheduler everyone's going to have a fuzzy idea of what a fork is, so that we are not going to discuss that in great detail. The idea is that a fork is an independent process. The reason it's called a fork is that one user might have several processes and if he does they are arranged in a hierarchical structure in which one process generates others by forking the path of control so that when there used to be just one path of control executing instructions, represented by the one original process, there are now two paths of control with two processes, one of which may be regarded as the original continuing on, and the other which may be regarded as a sub-process. There's a picture of hierarchical processes on page 3A of the working document. This picture does not exactly show how the path of control runs, so figure 5 is a picture that does show that. It is an exact image of page 3A. There are ten different places where instructions can be independently executed.

Q Why isn't there a line from 1 that says you're going down to 2, 4, and 5, since there are lines 1 to 2, 2 to 1.

A It would be very undesirable to have that line. I'll explain in a minute why, when I discuss in detail the structure of these pointers. It was because we couldn't figure out how we could avoid having that line that we delayed implementing this for a long time.

In figure 5, the lines represent the execution of instructions, and each place where there's a branch represents the execution of a BRS 9. Is it clear what the relationship between this picture and this picture is?

Q No.

A Well, we'll discuss the pointer structure, and then we'll go back and ask again whether it's clear.

Associated with each active process there are three pointers having to do with fork structure, and these pointers are called PFORK, PDOWN and PPAR, PAR standing for parallel. PFORK points up in fork structure to the fork which owns this one, which created this one. It can be zero if there is no creating fork, and the only time when that can happen is when you're in the top level fork which is created by the monitor to start this one up, and that fork is always running the executive. PDOWN is a pointer from the fork to essentially a list of the subsidiary forks—all of them. PPAR is a pointer which chains all the subsidiary forks together. The significance of the picture is that 2, 4, and 5 are forks which are subsidiary to 1, and in order to find 2, 4, and 5 starting at 1, you take the down pointer of 1 which takes you to 2. You

then take the parallel or cross pointer of 2, which takes you to 4, the cross pointer of which takes you to 5, and the cross pointer of 5 which is zero, meaning that the list is finished. Now you only have to have these three pointers for each fork, and the number of pointers you have for a fork is not a function of how many subsidiary forks you have. This is very nice, because it means that there is no storage allocation problem. It has the slight disadvantage that I can't get to all of the subsidiary forks quite as directly from the controlling one, but because I don't want to do that very often it doesn't really matter.

Q There's one thing that bothers me. There's an ambiguity between a down fork created at level 1 and a parallel fork created at level 2.

A Parallel forks cannot be created. A fork which is parallel to me is one that I have no control over. The meaning of it being parallel to me is that we are both created by the same higher level fork, and the only way that a parallel fork can come into existence is for the higher level fork to create it. I cannot create a fork that is parallel to me—I can only create things I have control over.

The difference between `PDOWN` and `PPAR` is that `PDOWN` points to one of the subsidiary forks and `PPAR` points from one subsidiary fork to the next subsidiary fork, so that the subsidiary forks are chained on a list, a particular example of this list being forks 2, 4 and 5 in the picture. `PDOWN` from the creating fork (1) points to one of these subsidiary forks (2) and then `PPAR`, which is the third little box in each big box, points along the list (across the page). So 2 points to 4, 4 points to 5 and 5 says that's it, I don't want any more.

Q We could say that `PDOWN` at a second level is equivalent to `PPAR`?

A More or less. It would be possible to send the `PDOWN` from fork 1 to any of 2, 4 and 5 and adjust the `PPAR` appropriately, so that eventually you can get to all of them.

So, that is the basic idea of fork structure, which is not at all complex. We now see that it is possible for a program to generate subsidiary forks, so the next thing to do is to see specifically how it generates them and just what relationship the fork can have to its subsidiary forks. This is a little bit more complex because there has to be a certain amount of machinery provided to do all the things you want to do.

A controlling fork generates a subsidiary fork by executing `BRS 9`. We must be able to identify the fork in some way. The way in which the fork is identified is that you put into the `A` register the address of a table, called the panic table for the subsidiary fork. A controlling fork identifies the subsidiary fork by its panic table address. All subsidiary forks must have different panic table addresses. The panic table consists of the objects listed at the very beginning of section 3, namely the program counter, the `A`, `B` and `X` registers, the map and a word for status. When the fork is activated, it is defined by the first six words of the panic table; the program counter, the central registers and the map are defined at the time when the fork is activated. In other words, the activating fork specifies the contents of the central registers, specifies the program counter and specifies the map. There are option bits in the word that we put into `A`; we can set certain of the top bits to indicate certain things. In particular, if we don't want to bother figuring out what the map should be, we can leave bit 1 zero, and that will mean that we use the same map as the creating fork. If we want to specify the map explicitly, we turn bit 1 on and then it takes the map from the panic table. In many cases you don't change the map; you are creating a fork for other reasons and you don't want to worry about the map.

When the fork is terminated for any reason, and also under certain other circumstances, its current state will be read into the panic table: the current values of program counter, `A`, `B` and `X` registers and relabeling will be read into the panic table. Furthermore, the status word, the last word of the panic table, will be set up to one of the values which come immediately after the description of the panic table in section 3, which describe what the fork was doing when you looked at its status. I think those are self-

explanatory. You will observe that the fork can do illegal things, and if it does things that are sufficiently illegal, like executing an illegal instruction, it will be terminated. That means, it will cease to exist, it will disappear from the fork structure. The panic table will, however, contain its final state and the status word, which will be set in the case of an illegal instruction to 1. This means that the fork which created it can monitor its progress, and when it is finally terminated, it can find out that it was terminated (there are various mechanisms for doing this which we will look at in a minute) and recognize from the status word what it did wrong.

Q How does the initial state of a fork get defined? By the system?

A We must define the initial state of the fork. It doesn't make any sense to define that automatically. In fact, the user must put into the panic table the value of the program counter and `A`, `B` and `X` registers with which he wants the fork to start its existence. Now, the program counter is always significant, since it tells what instruction is going to be executed first. He may not care about the `A`, `B` and `X` registers; maybe the fork immediately reloads them. I explained how he can avoid setting up the map by using the option bit.

Before we go on, we should look at the other option bits. This will involve an extension of the discussion of memory allocation. Option bit 0 says, "Make the fork executive if the controlling program is executive." The significance of being executive is that lots of things that are illegal in an ordinary program are legal to an executive program. Executivity is being sort of a compromise between being a user and being in system mode. An executive program runs in user mode and is exactly like a user program except that it can make a lot of system calls, it can do a lot of `SYSPOP`s and `BRS`s which are normally illegal for the user or restricted in some way. In other words, it can ask the system to do things for it that the system will not in general do for users. An example would be that an executive fork may change its relabeling with `BRS 44` in an arbitrary way, whereas the user fork can only change its relabeling by moving around blocks which are not executive. The executive program can change the relabeling any way that it pleases. For example, among the permanent entries in `SMT` are entries which point directly to the system itself in lower core. The reason for having these is so that one can look at the system while it is running. A user fork is definitely not ever allowed to get those entries.

We discussed option bit 1, which has to do with whether to set up the relabeling from the panic table or not. Option bit 2 propagates the rubout button. See below. Option bits 3 and 4 have to do with memory and they work in the following way. You remember that we mentioned in our discussion of memory allocation that there were three ways of getting memory assigned. You could be allowed no more memory, you could get new memory when you grabbed it, and there was a third way we did not discuss. One of these three ways is selected by the setting of option bits 3 and 4. If you turn 3 on, you have fixed memory. If you turn 4 on, you have what's called local memory, which means that you get your memory independently. That is the second possibility. But if neither one of those is turned on then you have the third way of acquiring memory, called propagating memory, which is the following. Suppose that we have a creating fork which has relabeling `01020710,04020300`, and it creates a subsidiary fork which it gives the map `01000000,0`. Now, suppose that this fork has been started up with propagating memory and that it addresses block 2, which is not assigned to it at the moment. If it had the fixed memory option, that would be an error. If it had the local memory option, we would put in a new block, say 16, which would be cleared to zero. The third possibility, which is the one we are considering at the moment, is that you look up through the fork structure at all the creating forks until you find one which has virtual block 2 already assigned, and if you find such a fork, you propagate the memory down through all the intervening forks to the one which caused the trap. In other words, in this particular example the relabeling for the subsidiary fork which caused the trap becomes `01000600,0`. If in the course of the upward search you do not find any creating fork which has this memory before you come

to a local memory fork, then you assign a new block that is propagated to all forks looked at. If you encounter a fixed memory fork which does not have the block assigned, you cause a trap.

A specific example of how you might use this facility, which is by no means hypothetical: Suppose that you have a program which is running and it has the following characteristics. There is a small piece of it, say 2,000 words worth of it, which is constantly being used. In those 2,000 words we have the main processing loop and all the common subroutines. In addition there are 8,000 words containing routines which are used relatively infrequently, statistical routines, matrix routines, whatever, it doesn't matter. These things we don't use very often, but they have to be around, because sometimes the exceptional condition occurs and we do use them. One very nice way to handle the situation is the following. You make a fork which has the whole program in it, that is five blocks of program (we said 10,000 words) and data blocks. This fork then creates a subsidiary fork, whose map contains only the commonly used code and the data blocks. It doesn't have the rest of the program. You create this fork with propagating memory. Now you fire this fork up and it starts to run. It runs in the 2000 words, and everything is just fine until suddenly the guy types in some unusual request, say for multiple regression. The program transfers to the multiple regression routine, which is in the infrequently used code, say in block 3. This of course causes a trap, since block 3 is not in the subsidiary fork's map. The system receives the trap and it looks around and says, "This thing is running with the propagating option," so it looks at its creating fork and finds an entry for block 3, say `12`, which is what the higher level fork has in that block. So it propagates the `12` down to block 3 for the lower fork, which starts going again. Everything is fine. It runs. So there is no difficulty.

The lower level fork normally runs with only 8K of memory instead of 16 and therefore can be swapped in and out twice as fast. Then it needs extra memory, When it needs parts of the bigger program, it simply goes ahead and addresses them and it gets them automatically, if they aren't there, brought down from a higher level fork. Now naturally, if this sort of program runs for a long time, it will probably eventually address all of these blocks, and the map will then look exactly like the map of the creating fork and we are not getting any benefit. This means that probably what you want to do with this program is that at suitable intervals you want to clear out the pseudo-relabeling, taking away the four blocks you don't normally use.

In this way, however, you have a mechanism by which the lower level fork can run with less memory than it really needs without any thinking. It doesn't have to keep a little table of subroutines which are not there, and do some special things whenever it wants to call one of them. Everything is being taken care of by the system.

Q The multiple regression routine runs in the original fork?

A The original fork has nothing to do with it. The original fork is dismissed the whole time.

This thing is sort of a substitute for a partial paging out. What we have done is that instead of saying "The system will try to figure out which pages should be swapped in and which pages should not, we say, "I will give the user a mechanism for defining for me a set of pages which he needs and then I will make it automatic that if he forgets some things or occasionally needs other pages, I will get them, so he doesn't have to worry about it." Basically, this transfers the load of partial paging from the system to the user, which for us seems to be quite a reasonable thing to do, since our pages are so large and the user is better informed. In most cases it will not be a great burden on the user to do this because it will be done in some higher level language, where it is appropriate for the programmer of the language to think about this question, and the actual user of the language will not be thinking about it.

So much for the option bits. Now we have explained about the pointer structure, we should now consider how forks get terminated. That is the next thing. A creating fork is connected to its subsidiary fork

in several ways, which are outlined starting on page 3-3. First of all, there are some operations which the creating fork can execute, which will have some influence or provide some information about the subsidiary fork. BRS 30 will read the status of the subsidiary fork into the panic table. BRS 31 will dismiss the creating fork. That means the creating fork will go away, it will no longer be on any scheduling queue in fact, it will be in limbo. And it will stay gone away until one of its subsidiary forks terminates. At this point, the creating fork will be reactivated. So BRS 31 is the kind of operation you use, for instance, when you have DDT and you are starting up a user program. When the user program has been started up, DDT has no reason to run anymore. It wants to dismiss itself until the user program is finished. By finished is meant that it did something illegal or that it said in some specific way that it is finished. Whether it came to a breakpoint or whatever doesn't really matter. You want to dismiss DDT until the user program stops running, at which point you want to come back and start running DDT again, so that the user at his teletype can debug.

BRS 32 is what you use when one of your forks has gone wild. It says, "STOP". Each one of these operations must identify the subsidiary fork which it is working on, and the way it does that is by putting the panic table address in A. If that address doesn't correspond to any panic table, then it is an error. The panic table is one of the entries in the PAC table.

Q Isn't BRS 31 a swapping operation.

A BRS 31 is really a scheduling function and not a swapper function for the following reasons. Suppose that I am running and I don't execute the BRS 31, what am I going to do? I have to do something, since I am executing instructions all the time. I could write a little loop that says, "Did the subsidiary fork terminate? If so, go off, otherwise loop." This is obviously bad; it means I am computing and wasting machine time in this little loop, which actually does me no good at all. So this is really a scheduling function and not a swapping function. The relationship of the swapper is just incidental to the fact, which is primarily important in scheduling, that this program is no longer running. If it's no longer running, it doesn't have to be in core.

When the program goes into limbo it isn't even on any scheduler queue. There is no activation condition for it. The reason there doesn't have to be an activation condition is that the only way it can get reactivated is by the subsidiary fork doing something, and there is a direct pointer from the subsidiary fork up to this fork, so that when the subsidiary fork does something, the system will arrange to put the fork that was dismissed at the beginning, back on a queue. This has an advantage in the sense that it tends to keep the scheduler queues somewhat less cluttered than otherwise.

In addition, there are some more operations. BRS 106 is sort of a global BRS 31. The latter says, "Dismiss me until this particular subsidiary fork terminates." BRS 106 says, "Dismiss me until any of my subsidiary forks terminate." BRS 107 and BRS 108 are more global operations, namely they read the state of or terminate all of the subsidiary forks, not just a particular one. These are very valuable operations. For instance, if you lose the panic table address of some fork, which can happen, you can never identify the subsidiary fork. There has to be some way to get you out of this situation. You can execute the global operation, which works on all forks.

The next thing we will discuss is how forks get terminated. And after that we'll talk about interrupts. How can a fork get terminated? There are the following possibilities. It can do something illegal, it can ask to be terminated or it can get involved with the rubout button. We will discuss these three possibilities carefully.

First of all, it can do something illegal. It can execute an illegal instruction, or it can cause a memory violation, (i.e. address something which is not assigned to it and which for some reason cannot be assigned to it). Those correspond to status conditions 1 and 2. A fork may also terminate itself in the

'normal' way by executing `BRS 10`, which essentially says to the system, "Terminate me and I'm a good guy, so transmit a 0 status word to the controlling fork, which can then recognize this is being a normal termination." The third possibility is the rubout button.

We now have to explain about the rubout button; this is the appropriate place to do it. There is a button on the teletype labeled rubout, which transmits the character `377` and which has the following characteristics in our system. There is for each user one fork which is designated as the fork which will receive rubouts. When a rubout occurs, it is directed to this fork, and one of two things will happen. Either an interrupt will occur, and we'll discuss that in a minute, or the fork will be terminated. So (ignore the interrupt for the moment), there is one fork which is designated as being the fork which receives rubouts, and when a rubout comes in that fork is terminated with a status word of 0, so it looks like a `BRS 10` termination. The difference is of course, that if you look at the location counter in the panic table it will point to a `BRS 10` if it is a `BRS 10` termination and will not if it was a rubout termination.

Now some more things about termination. When a fork is terminated, all of its subsidiary forks are also terminated, regardless of what they are doing. Whenever a fork is terminated, its status is always written into its panic table. There is an explicit instruction for setting the target of the rubout button, i.e. for specifying which fork is terminated by the rubout. If nobody has said anything about it, the highest level fork which is not executive is the one which will be terminated. However, a fork may declare that it is the one to be terminated by executing `BRS 90`.

The rubout button has the following characteristics. If it causes the fork to be terminated, it always clears the teletype input buffer, and furthermore, if the fork being terminated is executive it clears the output buffer as well. Clearing the input buffer means that anything the guy happens to type before the rubout which has not yet been processed will be lost. Clearing the output buffer means that anything the program has output which has not actually been typed out yet will be lost. The reason for this is that the idea of rubout is to stop what you are doing. In particular, if I gave you a command it was probably a mistake, so I don't want you to see it; if you are typing out, I don't want to see any more. One of the things rubout is nice for is that it allows the user to signal the program without the program having to look for the signal.

There are several ways to get the same effect without using the rubout button. One of them is to set up a subsidiary fork which will just do a `TCI`, which means that it will wait for teletype input. When the input arrives, the subsidiary fork will start up again and then it can do whatever it wants, e.g. send a signal to the main fork. But this involves setting up the subsidiary fork, which is tiresome.

The second thing you can do is to sit in the main fork and test at intervals to see whether the input buffer is empty. If it isn't empty, then the guy has typed something and you can look to see what it is. But what you can not do, of course, is execute a `TCI` in the main fork without checking that characters are present in the buffer, because that will cause the main fork to be dismissed. The rubout button is nice because it is automatic. It is a method which is already built into the system.

By the way, there is another special bonus feature attached to the rubout button, which is that when you push it twice in very rapid succession, like within a tenth of a second, which may not be too easy to do by hand, but always easy to do by holding down the repeat key, you can get directly back to the executive, regardless of what you were doing. The reason for this feature is that it is very easy to write a program which, as soon as it recognizes the rubout button, takes immediate action to neutralize it, like setting up again the fork which was terminated. This would be a bug in the program. It is a very bad type of bug because it is a kind of bug I can never recover from. I have no way to get to the system, because what I happen to be talking to is my program and it's not listening. There has to be some sure way of getting back to the system, and the sure way is to push down the rubout button very rapidly twice. You go right back to the executive, no matter what's going on.

You don't have to create a fork to process a rubout. There are other ways to do it, with interrupts. This is discussed below. In many cases, by the way, when you receive a rubout the appropriate thing to do is to keep executing the program, for a while anyway. For example, suppose you have a list processing program and when the rubout occurs you happen to be right in the middle of inserting a word on the list. It is clear that if you just stop at this point, you've left the list structure in a very bad state. If you want to keep things in good shape you must keep going to a point where all tables are in an acceptable form.

Suppose, for instance, that CAL is running, and in the middle of a computation it gets a rubout. CAL has been interrupted in the middle of building a list. Now what you want rubout to do is to cause CAL to tell where it was interrupted and to wait for instructions. If it does that without ever completing the operation on the list, the list will be unusable. So you have to make sure that the fork goes back and finishes whatever it was doing. The way this is handled in CAL is that each time CAL starts to execute a statement it checks to see whether the rubout was pushed during the execution of the preceding statement. In other words, pushing rubout starts a higher fork, which does two things. First of all it checks to see if rubout was pushed twice in a row. If it was, this is an indication that the running program got into some kind of a loop. If it hasn't been pushed twice, all the higher fork does is to turn on the panic flag. It then restarts. Each time the program starts executing a new statement, it checks to see if the panic flag is turned on.

There are two more things about termination. See pages 3-6 to 3-8. BRS 73, which is sort of a super BRS 10, turns off lots of forks at once. Then there is another mechanism which we will not discuss in detail which is specifically designed for DDT. If you don't understand it, it is not important.

As you all know, there exist hardware interrupts which are handled by hardware; the machine is forced to execute an instruction at location 200 + interrupt number instead of doing whatever it was going to do next. These hardware interrupts are of course not available to the user. Instead, he has software interrupts. A process has in its PAC table a 20 bit field called the interrupt mask. It may read this mask with BRS 49 and set it with BRS 78. Each bit is regarded as an arming bit for one interrupt. The first four interrupts are reserved by the system and the remaining 16 are available to the user. He may do whatever he wants with those 16. Let's consider the user interrupts first. If the user arms any one of the 16 user interrupts, nothing happens at that time. At a later time, some other fork may generate this interrupt. The way you generate an interrupt is to put the interrupt number in the A register and execute BRS 79. This causes the interrupt to be generated. The program continues to execute after the BRS 79 as though nothing has happened. It's exactly like a CPU sending signals to other CPUs. Of course, the interrupting fork may dismiss itself after generating the interrupt if it wishes, but it need not.

What happens to the signal is the following. It runs around in the fork structure, passing first of all through each of the forks parallel to the one which generated the signal and after that going up to the hierarchy along the PFORK pointers, i. e. looking at each of the creating forks in turn. To make this precise, look at the diagram on page 3A. If fork 8 generates an interrupt, it will go first to 7 and then to 2 and then to 1, not to 4 and 5. It goes around the forks parallel to the one generated by the signal and then goes directly up to the hierarchy. For each fork it goes through, it looks at the arming bit corresponding to the interrupt. If the bit is off, it just keeps going. If the bit is on, the specified interrupt occurs in that fork and the signal then dies. If none of the arming bits are on, the signal has no effect. When the interrupt occurs, the following thing happens. The fork in which the interrupt occurs is forced to execute an SBRM* through location 200 + the interrupt number. In other words, you must put in locations 200 to 224 (if you intend to use the interrupts) the addresses of 20 interrupt routines. Because of the SBRM* the location at which the interrupt occurred is available to the interrupt routine, and in fact the whole opera-

tion is entirely analogous to that of hardware interrupts. Note that the `SBRM*` does not actually exist; it is simulated by the system.

When the interrupt is generated and arrives at a fork and finds the bit armed and decides that it can interrupt this fork, then it changes the activation condition from whatever it was to the special activation condition that says interrupt number *n* occurred. When the scheduler finally gets around to activating this fork, it will, recognizing the special activation condition, simulate the execution of the `SBRM*`, which means that the location counter will be stored and execution of the interrupt routine will be started. The interrupt routine may do whatever it likes. When it finishes, it has the location counter available so that it can go back and continue executing a main program in exactly the same manner as a hardware interrupt.

All this action takes place in the fork which is being interrupted. The fork which generates the interrupt does nothing except to execute the `BRS 79`. Then it just goes cruising on. It has generated the signal. It then has no further responsibility. The signal goes off into the blue and eventually it may hit something.

Q Is it possible for an interrupt routine to transfer control to a different fork?

A The interrupt routine may do whatever it likes. If, for instance, the fork that got interrupted was the controlling fork of the one that generated the interrupt, it may, of course, terminate the fork that generated the interrupt. Or the fork that generated the interrupt may terminate itself. It doesn't have to do that but it can, since it is a free agent. Again, the fork which is being interrupted may say, "To hell with you, I see this interrupt but I am not interested," and just return to its main program. Anything can happen.

Now, what might you use this facility for? Well, the most obvious use is if you have two forks proceeding on a more or less independent basis, but you want to have some communication between them. Now, there are lots of ways to do this. For instance, one fork can write out a file and the other fork can read it in. Or, they can share memory. One fork can put a flag somewhere and the other fork could test it. But if you want to have the kind of interaction which is commonly associated with interrupts, which essentially means you are not wasting any time of the process that is being interrupted unless the interrupt occurs, then you really want an interrupt. This is what interrupts are for. It is a substitute for a lot of superfluous flag testing.

Suppose, for instance, I have a program that is doing a lot of computation, and it has a fork which is doing some subsidiary computation for it and which also does some I/O. I have a big matrix inversion program and over here on the side, I have a subsidiary fork of it which is doing some little calculation which the matrix inversion is going to need some day—or perhaps the fork on the side is receiving instructions from the user at his teletype which might suggest to the matrix inversion that it should change its policy. Perhaps the user is watching the progress of the iteration and it seems that some iteration constant should be changed. Okay. The fork that is receiving the information from the user will eventually acquire enough information to do something, and at that point it may wish to interrupt the main fork that is doing the computation in order to cause the main fork to actually change its policy somewhat. The device is very simple. What you might use it for is something you might have to turn over in your mind.

Q Is there a limit to the number of forks?

A That is subject to table size limitations. You can change those by changing a parameter when you assemble the system.

Q What happens to subsidiary forks when the main executive fork is dismissed?

A Dismissal has nothing to do with subsidiary forks. Forks interact with each other only in the ways which have been described. A fork is terminated. Subsidiary forks get terminated.

Q What is the relationship between the number of forks and the time-slicing algorithm?

A Every *process* is time-sliced, not every user. The processes are completely independent and act as independent users in all ways except the ways we have discussed.

Q So that a subsidiary fork is like a user all by itself?

A That's correct except for the things that have been described, like the rubout button. Also files go with the user.

There are limits to the total size of the `PAC` table. Furthermore, there is a little thing in the system which checks how many forks you are creating, and if you create too many it stops you on the theory that you probably made a mistake. But you can tell it, "Look, I intend to create 15 forks. That feature is not implemented yet, but it will be. Otherwise, you see, you can create a whole lot of forks and fill up the `PAC` table and that tends to paralyze any other activity that is going on.

There are four special purpose interrupts which are treated in special ways, and they are described, along with the whole interrupt facility, in Section 4, on page 4-1 and, in particular, 4-2.

If you (a fork) arm interrupt 1, and if the rubout button is directed to you, and rubout is pushed, then instead of your being terminated, interrupt 1 will be caused. Termination is pretty drastic and I may not want it. If the rubout button is going to cause termination and I want to maintain control of it, then I essentially have to make a special fork whose sole function is to be terminated by rubout. I would have to have a higher fork over the special one to catch it when it gets terminated. If I don't want to do that, I just want rubout to give me information, it's much nicer to have an interrupt occur.

Q This is a real interrupt?

A What do you mean by real?

Q Well, a hardware interrupt.

A No, this is a software interrupt. The rubout button, of course, has hardware involved in it in the sense that when I push the key, the interface causes the hardware interrupt. This is a software interrupt generated by the system. So if interrupt 1 is on, the rubout button will cause an interrupt rather than termination if it is directed to that fork. In particular, it will cause an execution of `SBRM* 201`.

If interrupt 2 is armed, a memory violation will cause an interrupt, instead of a termination.

If interrupt 3 is armed, it will cause an interrupt whenever any subsidiary fork terminates. There are lots of other mechanisms we have already described for waiting for subsidiary forks to terminate and for finding out whether they terminated or not, by reading their status and looking to see whether they are still running or not. In many cases, however, you start all your forks going and you want to compute yourself, and you just want to know when something happens to one of these subsidiaries. Then you can turn this interrupt on and you will be interrupted.

Finally, interrupt 4 will occur, if it is armed, whenever you get any unusual condition on input-output. It is exactly analogous to the hardware unusual condition interrupts. If, for instance, you're doing an input-output operation and get an end of file, various flag bits are turned on which allow you to find out where the end of file is. You may not want to test these bits every time on the grounds that the end of file only occurs once every 20,000 words. If interrupt 4 is armed, you can just get an interrupt when the end of file occurs and then, at that point, take whatever action is appropriate.

Q What is the relation between the hardware EOF on tape, say, and the software signal provided to the user by the system?

A If you see EOF on the tape, it will be a long time between the time the hardware sees the EOF and the time the program gets a signal that EOF has occurred. I will digress briefly into the discussion of the tape format to make this point clear.

Tape is handled in fixed-length records of 200 words. Each one of these records contains an initial word which tells how many data words there are in the record. This word will always be 199, except for the last record. So when you're reading the tape, suppose you read with WIO, you read one word at a time from a tape file. Every 199 words the tape will be activated to read the new record, and then 199 words will be fed to you out of the tape buffer. When you get to the end of the buffer a new record will be read, but you won't see any of this, of course. These are physical records and they have nothing to do with logical record structure. There isn't a logical record structure on the tape. Well, I take that back, there is a logical record structure on the tape, but we won't discuss it. So when you get to the EOF, what happens? The hardware reads off the last record and sees the EOF mark. The hardware is now sitting just after the file mark and the system is aware that an EOF has occurred.

So what the software does is to turn on this little bit somewhere in the file structure which says, "Look, there really aren't 199 words in this record, there are only 120 and then an EOF." The user program keeps on doing WIOs long, long after the hardware activity has occurred. It eventually uses up the 120 words of the last record and then, at that point only, when it tries to get to the 121st word, the system recognizes that there is no 121st word and that what there really is is an EOF. At that point, it will generate the software EOF interrupt.

Q To make it clearer: When you read the tape, you are not reading the information right from the unit?

A Certainly not. No. It is being buffered. So when you go to read a record from the tape, you are really reading the data from memory. There is some marker at the end of that buffer in memory that says, an EOF occurred a long time ago.

There seems to be a fundamental point that hasn't quite come over, and it is this: that people keep thinking there's some direct connection between all these software things I am talking about and the hardware conditions. Well, there is no direct connection. Hardware always goes through the monitor, and the chain of instructions which is executed between the time the hardware generates a signal and the reaction on the user program is a long and complex one. There is no direct connection at all. A user program can never connect itself directly to a hardware interrupt (except by a very special mechanism which is not entirely thought out and is altogether special and different).

Q There is no hardware interrupt with a direct effect on the user program?

A True, a hardware interrupt has no direct effect on the user program. It has direct effect on pieces of the system, of course, but not on the user program. Usually, it won't even have a direct effect on the scheduler. For instance, the clock interrupt occurs and what happens is that it goes up to the clock interrupt routine, which almost never goes directly to the scheduler. Usually it just increments a bunch of counters and flags and then turns things on and off. At some later date, someone else will look at these things. Other interrupts never go directly to the scheduler. When a drum interrupt occurs, it may cause some more drum I/O, or it may cause an activation condition to be changed, a word to be changed, which will eventually cause some program to be activated. There is no direct connection there with the scheduler.

By the way, there are some things which sort of come under the general heading of the areas we have already discussed that I have not mentioned, for instance, there are these timer BRSs, and some

other items like that. These things are pretty trivial and they are in the working document so that when we finish these lectures, everyone should read the working document again and try to pick up some of the features that are not being discussed.

You can get some sort of idea of all the features in the system in the following way. When the lectures are finished, you should be able to read the BRS table, which is in Appendix A, and the SYSPOP table, which is in Appendix B of the working document. You should be able to recognize what each one of those BRSs and SYSPOPs are. For any one which hasn't been mentioned, there is a reference to some page in the manual which describes what it does. If there is any confusion, then you should come and ask questions. Don't try to do that now because there is a lot that will still be incomprehensible.

Q These locations where the interrupts go (201 etc.) are obviously relocatable locations?

A They're not. Virtual locations, not relocatable.

Q Can they be absolute locations?

A Certainly not. This is entirely a user operation. The user, remember, doesn't know anything about absolute locations.

Q Each user program has its own set of interrupt routines?

A Each *process* can have its own set of interrupt routines if it has a different block zero from other processes. If it has the same block zero, of course, it will have the same interrupt routines.

We want to look at the clock interrupt routine now, which is short. I just want to give you some idea of what actually happens. The clock routine is on page 19 of SPAC, where it says CLINT.

What it does is the following: It comes in. It increments REAL, which is a word which is always incremented at every clock interrupt. It essentially keeps track of real time for the system. Then it increments indirect through TJOB, which contains the address of the word being used to count elapsed time for somebody. That is used for billing purposes and also for keeping track of the actual computer time used by various people. Maybe they are interested in knowing whether they used computer for a minute or six minutes. We saw TJOB already in the scheduler. Then it tests the sign bit of CLINT to see whether the clock interrupted out of system mode or user mode. If the SKN doesn't skip, it came out of system mode and so we increment STIME which is a counter which tells how much time is spent in system mode. It is only for information. Then we count down TTIME, which is the long quantum counter. If the SKR skips, then TTIME has gone negative and in that case we go to CLOUT. This means that we want to dismiss the guy. The way we do that is we look to see if the user mode trap is on—the flag is on—and if it's not, we execute UMTN, which is a macro defined on page 1 of MDBG. Don't bother to look at that. The reason that it is a macro is it is going to be changed when we get the real user mode trap in. Then it will turn on the user mode trap. This will cause the user mode trap, which will send us to the scheduler to occur as soon as we return to user mode. This means that the user will be dismissed as soon as possible. When we get to CLOUT, we want to dismiss the guy for quantum overflow, but we don't want to dismiss him until he comes back into user mode because when he is in system mode, the system may be doing something which it can't be interrupted out of.

If TIME has not run out, then we do the SKR TIME, TIME being the short quantum time. If the short quantum hasn't run out either, then we just return. (If neither the long quantum nor the short quantum has run out, then we definitely do not want to dismiss the guy.) If the short quantum has run out, then we want to test ACTR. If ACTR is still negative, we want to exit, because that means that even though the short quantum has run out, there is nobody who can be activated on the I/O queues. If ACTR is positive, it is possible to activate somebody on an I/O queue and we want to do that, so again we go to CLOUT.

Teletypes

We will now turn to the section on teletypes, and we will begin by essentially reading out of the working document what the user can do. Then we will go and look at the code, and see how he is able to do it. We are looking at Section 7 of the working document.

The most fundamental straightforward thing you can do with the teletype is to use two instructions caused `TCO` and `TCI`. `TCO` stands for teletype character output and `TCI` stands for teletype character input. One of them prints a character from memory on the teletype, and the other one brings the character in from the teletype and puts it in memory.

Now, before we go on and explain the other I/O operations, it is worthwhile to discuss the many ways in which the operations `TCI` and `TCO` can be modified by various things you can do to the teletype. Teletypes have certain properties. They have echo tables associated with them, and they have links associated with them. They also have owners and various status bits. When you type the character in on the teletype, as you know, the system operates in full duplex. The character is not printed by the teletype hardware on to the piece of paper sitting in the teletype. The way the character gets on that piece of paper is that the system looks at the character which is input, figures out what to echo back to the teletype, and echoes it. There are several different modes that are possible for echoing. Which mode you use for echoing depends on which echo table we are using. An echo table is a collection of 32 words, each divided up into three bytes, for a total of 96 bytes, corresponding to the 96 ASCII characters which in internal code are `0-77` and `140-177`. These are all the printing characters and all the characters that you can get by holding down the control button and pushing various keys. These characters, by the way, are not the same as external ASCII. External ASCII is obtained by subtracting `140` from internal mod 2^8.

Q The system does not use the teletype's characters as they are?

A That's right. The system makes a transformation on the characters to convert them from external form to a convenient internal form in which the printing characters range from `0` to `77`.

There is an echo table which contains 96 entries for these 96 characters. When a character comes in, the appropriate byte corresponding to that character is collected from an echo table and something is echoed, depending on what the bottom 7 bits of that byte are. The 8th bit is for something else. If the bottom 7 bits are zero, the character is discarded completely as though it never existed in the world (except that it might be a break). If the bottom 7 bits are one, then the character is put into the input buffer but there is no echo generated. Otherwise, the character which shows up those bottom 7 bits is echoed in place of the character which was typed in.

Now, the 8th bit is used to perform a different function. When a `TCI` is executed, if there is a character in the input buffer of the teletype, it will be delivered immediately to the program. If there is no character in the input buffer, the program will be dismissed waiting for teletype input. When it will be reactivated depends on exactly what is typed on the teletype. Whenever a break character, which we will define in a minute, arrives on the teletype, the program will be reactivated. If the input buffer comes within some fixed number of characters of being filled, the program will be re-activated anyway, regardless of whether any break characters have been typed in or not. This fixed number is called the early warning margin. It is currently 6. This is to make sure that the program has time to take some characters out of the input buffer before enough other characters come in to overflow it. Whether a character is a break character as determined by the top bit of its byte in the echo table. If that bit is on, it is regarded as a break character, otherwise not. In theory, the user can provide his own echo table and he can set up any echoes he wants. In actual practice, we don't allow this because the echo table must be resident.

We provide four standard echo tables in this system, identified by the numbers 0, 1, 2 and 4, and they provide for the following: The first three tables will echo everything, exactly as it was typed in, except, of course, the control characters which take no action (and the only control characters which do take any action in our standard echo tables are bell, carriage return and line feed). This means, for instance, that if a teletype is equipped with tabs, you will never be able to generate a tab on that teletype by pushing `I^c` because the system will never echo it. So those three tables have the same echo characteristics, they differ only in the break characters. In table 0, everything is a break character. In table 1, everything is a break character except letters, digits and space. In table 2, the only break characters are control characters. These three tables are for applications in which you require steadily decreasing amounts of interaction with the program. Table 2, in particular, is suitable when you are interacting with the program on a line by line basis.

The fourth possible echo table is one in which everything is a break character and nothing is echoed. Because nothing is echoed, the program may figure out for itself what it wants to echo, and echo it.

The user may set the echo table by executing a system call. In particular what he does is, he puts an echo table number in the `A` register, and a teletype number in the `X` register and he executes `BRS 12`. That sets the echo table to the number specified in the `A` register. For convenience, he may specify his controlling teletype, the one which he is entered on, with teletype number `-1`, so he does not actually have to know its number. If, however, he is controlling other teletypes explicitly through mechanisms we will describe in a minute, he must then specify the number of the teletype.

One more thing, and that is the matter of deferred echos. It is possible, because the system is full duplex, for the user to type in while the system is typing out. If he does this, a question is raised about what should be done with the echo. The simplest and stupidest thing you could do is just tack the echo on the end of the output buffer, which means that somewhere randomly interspersed with the characters being typed out will be the echos of the characters being typed in. This, however, is probably not what you have in mind, because probably the system is typing out some connected string of characters.

Q What is 8 level mode?

A 8 level mode is a mode in which a teletype operates by transferring 8 bits from somewhere to somewhere else with no conversion, no checking for the possibility that an input character might be rubout, no echos, no nothing. It turns the teletype input routine into a completely transparent thing which makes no changes whatsoever in the data which is coming in from the teletype. In other words, 8 bits come in from the teletype and the same 8 bits are shipped out to the program.

Q What happens if it is converted?

A A transformation is made which changes standard ASCII into the internal form, in which the printing characters are in the range from 0 to 77. When conversion is done a check is also made for the possibility that a rubout was typed in, in which case drastic actions are taken. Finally, an echo is usually generated in the normal mode. All of these things are suppressed in 8 level mode.

Q 8 level mode takes no cognizance of break characters?

A That is correct. The early warning margin, however, is increased in 8 level mode, since it is likely that characters are coming from the paper tape reader. This means that the stuff is coming in pretty fast and it is advisable to leave a little extra warning margin to make sure that the program comes in time.

It is worth pointing out that if by some chance the input buffers should overflow, you normally get an unmistakable indication of this, because if the input buffer overflows, no echo is generated. This means that if you are typing you get a clear indication of what went into the input buffer. Anything that

doesn't get typed out got lost; either it got lost because of input buffer overflow or the program changed the echo table and omitted to echo anything. In 8 level mode you do not have this protection since there is normally no echo in 8 level mode.

Next we wish to discuss teletype links, which work in the following way. Associated with each teletype there are four bit tables called the absolute and the relative link control words, for input and output. In our system, each one of these is one word in length, since there are less than 24 teletypes. In a link control word there is one bit corresponding to each teletype. The basic idea of a link control word, say the output link control word, is that if a character is being output to a teletype (which we may call the source teletype) it will also be output to all teletypes for which the corresponding bit is turned on in the output link word for the source teletype. A character being output to a teletype means any character which the computer causes to be printed; that includes echos. This means that if I am typing and I am linked to somebody, both the echos which the computer generates for me and the things which my program outputs will appear on his teletype. Furthermore, if he is linked back to me anything typed out on his teletype will also appear on mine. There are some slight complications to this idyllic picture; having described the basic idea, I will now proceed to consider the complications.

First of all, it is not necessary for you to be linked to yourself. If you are not linked to yourself then you do not get your own output. Normally, of course, you are linked to yourself. Secondly, the reason that these words are called relative link control words is that there are also words called absolute link control words. The reason for these is the following. Suppose that teletype 1 has a relative link control word saying, "I want to be linked to teletype 2" and that teletype 2 has a relative link control word saying, "I want to be linked to teletype 3." Since the rule is that anything output on teletype 1 will appear on teletype 2 and that anything output on teletype 2 will appear on teletype 3, it follows that anything output on teletype 1 will also appear on teletype 3. There are two ways of handling this problem. One way is to follow down the chain of links explicitly each time a character must be output. This has obvious disadvantages in terms of efficiency. Furthermore, if 2 is linked back to 1, there will be a slight tendency for the character to be typed out an infinite number of times on both teletypes, and the routine following down the chain must check for this explicitly. Instead of that we do the following: whenever anybody changes any relative link control word, we recompute all absolute link control words, and the absolute link control words for a teletype have a bit turned on for each teletype linked to the source teletype or for each teletype which is linked to a teletype which is linked to the source teletype, etc., etc. In other words, the bits in the absolute link control word are turned on for all the teletypes to which the characters should go. This computation is made once and for all, every time a relative link control word is changed. It is then the absolute and not the relative link control words which are used in generating characters through a link.

For those of you who are mathematically inclined, you may recognize that the process of computing the absolute link control words is essentially generating the infinite Boolean product of the Boolean matrix of the relative link control words. That is, you take the Boolean product of the relative link control word matrix with itself and continue doing this until it ceases to change. There is a cute little algorithm in the system for doing that, taken from a *J. ACM* paper by Irons.

So much for output links. There are also input links. If a teletype is input linked to another teletype, every character typed in on the first teletype will appear in the input buffer of the teletype which it is linked to. To summarize, output linking refers to characters which show up on the paper of a teletype; they will also show up on the paper of every teletype to which the original one is output linked. Input linking refers to characters which show up in the input buffer of a teletype; they will also show up in the input buffer of every teletype to which this one is input linked.

It is clear that running through this linking machinery all the time is rather time-consuming, and consequently it is only done when necessary. There is therefore a single bit in the teletype table (by the way, there is a picture of the teletype tables on page 7A of the working document) which indicates that there is something funny about the teletype. This bit is the top bit of the word labeled TTYTBL. Funny things include 8 level input or 8 level output and any kind of linking. If there is nothing funny about the teletype, the link word is never examined. This means that you don't have to pay for this machinery when you are not using it.

It is clear that it is intolerable for teletypes to be randomly linked to other teletypes, since this offers entirely too much scope for inadvertent error or deliberate mischief. The protection against this works in the following way: there are two bits associated with each teletype called the accept messages bit and the accept input bit. When the accept messages bit is off, it is illegal for anybody to output link to that teletype. If the accept input bit is off, it is illegal for anybody to input link to that teletype.

These are separate, because input linking is regarded as more serious than output linking. With output linking all you can do is look at and interfere with the output. With input linking you can actually affect the operation of the program.

There is an operation called read echo table, BRS 40, which accepts a teletype number in the X register and reads back into the A register the following information: the echo table currently being used, the 8 level input and output bits and the accept messages and the accept input bits. This allows you to find out about all these things. Then there is another BRS which essentially accepts all these bits. The normal setting of the accept messages bit is on, of the accept input bit is off. Executive commands as well as BRSs are available for changing the setting of these bits. The normal organization of this system with regard to all these funny features is that they are implemented through BRSs which the user program may execute if it has the proper authorization. The executive then has commands whose programs essentially do nothing but execute these BRSs.

There is one other feature which sort of goes along with linking, and this is described at the very end of section 7 on page 7-8. It is called Simulate Teletype Input. When you do STI, you put a character in the A register and address a teletype. The effect of this is to simulate the typing in of that character on the specified teletype. It is legal only if the program is executive or the accept input bit for the teletype is on. In other words, if you put the character M in the A register, and do STI =14, you simulate the effect of pushing the key labeled M on teletype 14.

There are still some more things associated with teletypes. In particular, there is a lot of machinery for dealing with teletypes other than your own controlling teletype. First of all, if you want to own a teletype, you can attach it. You do this by putting its number in some register and doing the appropriate system call. If you are authorized to attach teletypes and that teletype is not owned by anyone else, the teletype is attached to you. This means that you own it; you may set its echo table and all of its funny bits and do any operation to it regardless of whether the permissive bits are set or not. Generally, you have complete control of that teletype. You keep this control until you explicitly give this teletype up or are logged out.

Q Does this mean that we here internally will be able to attach any teletype from this system, regardless of where it is?

A Yes. If somebody is already logged in on the teletype, of course, it is not attachable.

Q Only dormant teletypes are attachable then?

A Yes.

Then, there are operations for reading input from and directing output to specified teletypes. These are called OST and IST, Output to Specified Teletype and Input from Specified Teletype. They take the character from the A register or leave it in the A register and address the teletype which you wish to read from or write on to. You are not allowed to do IST or OST unless the teletype has the accept messages bit turned on. This provides a facility which in some sense is complementary to that provided by linking. Linking arranges that automatically any characters going out on one teletype go out on another one. This facility allows a program to specifically direct a character to a specific teletype.

You might use these operations to communicate with your own attached teletype or to talk specifically to some other teletype belonging to some other user.

There are actually three ways in which OST can be legal. a) if the teletype is attached to the user; b) if its accept messages bit is on; c) if you are executive. All of these conditions must fail for the OST to be illegal.

Finally, there is one more thing in the teletype user interface. One can also address the teletype using the standard system input-output operations, of which the most important is CIO. CIO stands for Character Input-Output and you address a file number with it. The I/O takes place to and from the A register. Whether the operation is to be input or output is determined by the nature of the file. Normally a file has to be opened before it can be used. The teletypes are however, permanently open files. In particular, there are file 0, which refers to input on the controlling teletype; file 1, output to the controlling teletype; file 1000 + teletype number, input from a specified teletype; and file 2000 + teletype number, output to the specified teletype. These provide an alternate set of instructions which you can use instead of TCI, TCO, OST and IST to do teletype I/O.

They have the following advantage: a program using these instructions can easily be changed to use another I/O device. It is normally possible to use CIO for input-output rather than a specific teletype instruction. This leads to the obvious question: “why do we provide the specific teletype instructions at all?” Which is a good question and has the following answers:

1. That the specific teletype operations are somewhat faster since they have to do less error checking.

2. That the teletype instructions were designed long before CIO was invented.

I believe that we have now said everything that there is to say about the user interface for teletypes. All of these matters are discussed precisely although concisely in the working document. We will now turn to the teletype code. This will enable us to see, among other things, how interrupt routines really work.

The first thing we observe about the teletype code is that in the very beginning there is a whole bunch of tables defined. At the top of that bunch of table definitions, there is a comment that says: “Tables indexed by teletype number.” There are 16 of these tables. This means that every time you add a teletype to this system, you add 16 words, one in each of these tables. A word corresponding to a particular teletype in one of these tables is obtained by using the base address of the table in an indexed instruction and putting the teletype number in the index register.

These tables are roughly explained by the comments and we will see in more detail what they do as we look through the code. The tables labeled TIS-something or other, have to do with teletype input. Those labeled TOS-something or other are involved with teletype output. The link control words are at the bottom of page 1, and in between there are a bunch of words with obscure names like TTYTBL, whose structure and function are described in the working document. The last entry in this collection, TTYTIM, is used to record the value of the clock at the last rubout. It is this word which allows us to im-

plement the feature which I described last time, of having two rubouts which come close enough together send the program directly back to the exec.

Next are the echo tables, which are defined by a macro called ECHR, whose definition is found in MDBG. Then there are the teletype buffers. These are organized in the following way: the input and output buffers share the same words according to the picture at the beginning of section 7: Each word has one character of input buffer, one character of output buffer and one character of deferred echo. This means that the length of an input buffer is the same as the length of an output buffer. These lengths are defined by a system parameter called NTTYC. Notice that the teletype buffers are defined toward the end of page 3 by TTYBUF BSS LTTY, where LTTY has just been defined to be the number of teletypes multiplied by a length of a teletype buffer plus 1.

The code starts off with the teletype input interrupt routine. Before we plunge into that, it will be worth pointing out that in order to start up the teletype logic, it is necessary to execute a routine called TTYSET, which occurs on page 16 of STTY. The function of this routine is to set up properly all of the input and output pointers and all of the little bits and to make sure that everything has its proper initial state. It does one thing which is rather puzzling and may be worth pointing out in detail.

The teletype buffers are, as we have seen, one huge table. Each individual teletype has a section in this table assigned to it. This section is terminated with a word whose contents is minus the length of a teletype buffer. It is this word which allows the teletype routines to detect when they have come to the end of the buffer.

We now return to the teletype interrupt routine. It has space for the A, B and X registers and a few other temporary storage locations. TII at the top of page 4 is the address of the teletype interrupt routine. Location 201 is set up to contain BRM TII. The routine saves the A, B and X registers, does TTYS, which an EOM which tells the teletype interface that the computer wishes to read the character, followed by a PIN which reads in the character.

As we have already discussed, the 24 bits read by the PIN have a character in the top 8 bits and a teletype number in the bottom 6. After the manipulation in the second line of TII we wind up in the third line with the teletype number in the A register. We check to see that it isn't too big and if it is, we go off to TII4, which exits immediately. This can happen only if the teletype interface has fouled up, or if there are more teletypes attached than the system is prepared to deal with. This guarantees that if some random teletype number, say 150, comes in, we do not go out and attempt to find the pointers to teletype 150 in the teletype table. These pointers would, of course, simply be random numbers.

Next, we check the top bit of TTYTBL. If it isn't off, the teletype is funny and we go off to TII4 to check for 8 level input. Otherwise, we get the character to the A register, mask out all but the bottom 7 bits, and check for rubout with the SKG =177. If it is not a rubout we go off to TII1, which processes a normal character. If it is a rubout we do something quite different. First, we look to see if the bits of TTYASG corresponding to the job number are zero or not. If the bits are non-zero, then the teletype is either unassigned or is a controlling teletype. If they are zero, it is an attached teletype, in which case we wish to treat rubout as an ordinary character.

Q How did the job number get in there?

A It was put there when the guy entered.

Q The guy is required to supply a job number?

A No, the system provides it for him. We will see this being done in a few minutes.

If we get through all this we have a valid rubout. We save the value of the clock and then we check to see whether the clock differs from its old value by more than 8 clock cycles. If it does not, then we set some indicator which will cause the rubout to go directly back to the exec instead of simply terminating a single fork.

Q What exactly is going on with TTYTIM?

A Every time a rubout occurs, the current value of the clock is saved. It is also compared with the previous saved value, and if they differ by 7 cycles or less the rubout causes all forks lower than the executive itself to be terminated. This matter is discussed in more detail in the sections on fork structure. The clock value is saved in the TTYTIM table.

What happens next is the following: The teletype interrupt routine is a high priority interrupt, so we cannot afford to spend a long time fiddling around in it. What we are going to do in order to process rubout is to fire up the 207 interrupt. This is done in the following way: the 207 interrupt is permanently tied on; it is normally, however, disarmed. We intend to use this interrupt to process things which we do not have time to process in a high priority interrupt. 207 is the lowest priority interrupt we have, which means that it is OK to spend as much time in 207 as you want, since no other interrupt can be hung by your sitting in 207, only the main program. What we do when we wish to activate 207 is to arm it. As soon as it is armed, the interrupt will occur. Since we are in the 201 interrupt routine, no interrupt will occur until we leave the 201 interrupt. At that time, if no interrupts between 201 and 207 are waiting, the 207 interrupt will occur and the rubout will be processed at leisure. Rubouts are processed in the 207 interrupt routine because of the considerable complexity of the processing involved. The other thing processed in the 207 interrupt is links.

All of the little tables labeled ATIS are designed to tell the 207 interrupt routine what is going on. The way the 207 interrupt routine is worked is that it has a ring buffer called ATTBUF which is used to hold the things for the 207 interrupt to do. ATIS4 is a pointer which tells the 207 interrupt where to read out the next thing to do, and ATIS5 is a pointer which says where to put the next thing to be added to the buffer. ATIS2 tells us how many things there are in the ring buffer. When this number is zero, the 207 interrupt knows that all of the currently scheduled work is complete. The rubout logic then is incrementing the write-in pointer and then checking to see whether the word pointed to by the write-in pointer is negative. The reason for this is that the word following the thirty words of ATTBUF is -30. All entries in that buffer will be positive, which means that if the SKN skips, we have run off the end of ATTBUF. To wrap around the ring buffer, we need only add the -30 to the pointer. This will suffice to reset the pointer to the beginning of the ring buffer.

By the way, it may be as well to comment briefly on what a ring buffer is. A ring buffer is a block of core which is used to maintain a linear list of things which is being added to on one end and taken off from on the other. In this case the teletype interrupt routine is doing the adding and the 207 interrupt routine the taking off. The two pointers, to the last thing added and the next thing to be taken off, precess around the buffer. This is why it is called a ring buffer. When the pointers run off the end of the buffer, we wish to reset them to the beginning, since the end of the buffer is not a significant point.

This is what is going on at this point. Exactly the same device is used in the teletype input and output buffers. Ring buffers are very useful devices, although because of this wrap around business they are a little bit more difficult to manage than ordinary buffers. There are many applications in which they are invaluable.

Q What do you do when the pointers collide?

A That varies. In the case of teletype input you simply throw the character away. In the case of teletype output, we dismiss the guy. In this particular case of the 207 interrupt routine, the collision condition is not checked for. This is unfortunate.

To return to the code under consideration on page 4 at TII6A+6, the second instruction is an SKN* ATIS5. If that skips, then ATIS5 is looking at the -30 and it must wrap around the buffer, as explained above. We now proceed to put the teletype number, which we get from TIIS2, into the 207 buffer. Then we arm the 207 interrupt with this AIR and POT sequence. The AIR is an EOM which says that the next POT will be an interrupt arming word. After arming the 207 interrupt, we exit immediately. We have now transferred responsibility for the rubout to the 207 interrupt. As soon as it gets a chance, it will come in and process the rubout.

We have now dealt with rubout, and we won't look at the 207 interrupt. We will go on to cover all the rest of the teletype input interrupt routine; there are a couple more pages of it.

TII1 is where we go when we have decided that there is nothing special about a character and we are going to treat in a perfectly normal way. We say SKN TTYFLG,2; this word will be negative if the teletype is alive and otherwise it will be non-negative. In other words, if the SKN does not skip, this teletype is dead and we want to ignore the character completely. So we go to TII4. We will see later how TTYFLG gets set. Otherwise, we do this multiplication by this obscure number which happens to be one third in octal. The reason we multiply by one third is that we want to get into the echo table, which is organized three characters per word. The multiply leaves the word address for the echo in A and the character address in the top two bits of B. We add the word address to the echo table address in TTYTBL and get the echo table word. Then we do some shifting and some masking whose net result is to get the echo table byte we want into the top of the B register. Then we pick up the character and tty number into X from TIIS2. We send it to A, and we add X3. This constant is defined in SIO. It is three in the top octal digit of the word, which is 140 in the top character. We are therefore adding 140 to the character mod 2^8; this is the transformation from external to internal.

And then we mask so all we have is the seven bits of the character at the left end of the A register. Now we do two SKBs, which are designed to isolate the possibility that the echo character is 0 or 1, which are the special cases that we mentioned. If the first one skips then the character must be either 0 or 1. If the second one does not skip, then the character must have been 1; we put it into the buffer, but we don't echo it, so we go to TII0, skipping over the echo logic.

In the third case, it is 0 and we ignore it completely except that we have to check for the possibility that it is a break character, which we do by going to TII1A.

If we get to TII1B we have to generate an echo, and the echo character is sitting in the B register where we put it as a result of our machinations with the echo table. The register change instruction has the E bit turned on, which says, "Only manipulate the bottom 9 bits in doing the operation." The bottom 9 bits of X replace the bottom 9 bits of B, which now contains a word which can be used to output the echo, since the tty number is in the bottom 9 bits. Then we save away the echo in TIIS4 for the moment.

Now, we look to see if TTYTBL is negative. The bit that is being tested is the funny bit; if there is anything funny about the teletype, we will defer the echo. This is obsolete code, but its effect is to defer the echo if there is a link. The reason is that the link computations are too long to be done in the 201 interrupt, and the defer guarantees that they will be done in the TCI routine. We now handle links in the 207 interrupt, however.

If there is nothing funny about the teletype, we look at TOS2 and see if it is negative. TOS2 tells us how many characters are left to be typed, and if it's negative the teletype is not typing out anything. Consequently, if the SKN doesn't skip then we know it is typing and we have to defer the echo. If it does skip, we don't have to defer it. We still have to check for the possibility that we deferred some previous echo, in which case we have to defer the current one. The way we do that is to pick up TIS2, the number of characters sitting in the input buffer, and compare it with 0. If there are no characters in the buffer, we definitely don't have to defer. We immediately output the echo by doing the teletype EOM and PCT. Then we MIN TOS2.

If on the other hand TIS2 is not 0, we go to TII3A which checks whether the preceding character was deferred. TIS4 points to this character, so we load indirect through it and look to see whether any of the deferred echo bits are non-zero. If all are zero, we go back and output the thing.

When we've made it through all of this, we have arrived at TII0, which is going to put the character away in the input buffer. We increment TIS4, which is the write-in pointer of the input buffer. If the 200 bit of the word it is pointing to is on, then that word is the one after the end of the input buffer, and it is necessary to wrap around the buffer.

Having taken care of this possibility, we store A into the first and third characters of the buffer word (the input and echo fields). Then we increment the character count and check to see whether it is bigger than the length of the input buffer. This is a test for collision. If the buffer is full, then we are going to make TTYFLG positive. TTYFLG was tested way back, and if it was positive we just ignored the character completely. It can be positive under two circumstances: if the teletype is dormant, or if it gets set positive because of collision. Every time a TCI is executed it is set negative.

Then we pick up the character count and test for early warning. TTY early warning is a number which is currently equal to 6. If you get within six characters of filling up the buffer, you want to pretend that you saw a break character, even if you really didn't. If we don't have early warning, we check the top bit of the echo character, which is still in the B register, to see if we are processing a break character. In either case we go to TII7, which is responsible for doing the right thing about break characters. Otherwise, we go to the standard exit at TII4 which restores the A, B and X registers and returns.

So we've now made it through the main sequence of this routine. There are still a number of things to clean up. The first is the break character handler at TII7. Here we MIN location TTYBRK, which is negative if no break characters have come in. Every time we see a break character, we increment it. The activation condition will be TTYBRK ≥ 0. Then we increment ACTR, since now we have someone to activate, and we go to TII4.

Next is TII6, where we will check for 8-level input. We pick up TTYTBL and look at the 8-level bit. 8RB is just a number which has one bit on, namely the bit corresponding to the 8-level input bit; it is defined in MDBG. If the bit is off, we go back to TII6A. If we do have an 8-level input, then we proceed quite differently, checking the possibility that the character might be the 8-level EOF character, which is kept in TTYTBL. If it is, we are about to leave 8-level, so we go through a lot of rigmarole whose function is to turn off the 8-level mode and also to turn off the funny bit if there is no reason to leave it on. Having disposed of that possibility, we go on to TII2, which is where we have an 8-level character which is not an end of file. In that case we pick up the character from TIIS2, we mask everything else out, clear the B register (no break) and go off to TII0, where we file it away.

I think we will pass over the rubout handling logic, which is in the 207 interrupt, and go on to consider the TCI routine, which actually processes a user request for a teletype character. This routine starts

on page 12 where it says TCI. It begins by storing away the B and X registers; it doesn't need to save the A register.

It picks up the user teletype number into X and calls TI, which is to be found in the middle of page 11. What this routine does is to pick up TIS2, which is the character count, and look to see if it's bigger than zero. If not, it goes to TI3 and dismisses the program, because there aren't any characters. Otherwise, we increment the read pointer TIS5 and pick up the character out of the buffer (testing to see whether the 200 bit of that word is on, in which case we have to wrap around). Then we get the character into the bottom of A and check for a deferred echo. If there is one, we go off to TI1. Otherwise, we count down the character count of the buffer, put -1 into TTY FLG and exit.

Next, we look at the place where we do the dismissing, which is TI3. Send -1 to TTYBRK, which is the word that is going to be tested by the dismissal condition, which we now proceed to construct. We copy the teletype number into A, mask it, and add in 200000 + TTYBRK. This gives dismissal condition 2 and the address of the TTYBRK corresponding to this teletype in the address field. TI returns with a skip, leaving that dismissal condition in the B register.

If we have to print a delayed echo (TI1 on the top of page 12) we get the echo character and call TO, which is the routine which is responsible for outputting characters. If we are successful in outputting it, we go to TI6, which restores the character and goes back to what we were doing before. If TO is not able to output the character, (because the output buffer is full), then it returns with a skip and the dismissal condition in B, and we have to dismiss the guy. We have to move the read pointer back down because we already incremented it. Remember, what is going to happen when the program is reactivated is that the TCI will be re-executed.

Going back to the place where we call TI, if it does not skip, we go to TCIP1, otherwise to TIODMS, which is where we are going to dismiss the guy, putting QTI in X and going to POPDMS in the scheduler. So then the guy is gone and we won't see him again until we start him up again at the TCI.

If we get to TCIP1, we pick up the X register, we pick up the B register, we store the A register indirect to zero (because TCI sends the character not only to the A register but also to memory) and we return.

I would like to go on now by going through the teletype code and pointing out what all of the routines do and any peculiarities they may have, without actually looking at any of the code. We have already gotten ourselves as far as page 6, which is where the 207 interrupt routine is. On page 6, we see the logic to process rubout which is tedious. Then on page 7 there is a routine called TIP which is called from the phantom user and which is responsible for doing the final job on rubout. The way rubouts are being handled now is that in the 201 routine we immediately activate the 207, in the 207 routine we set up a phantom user request and the phantom user then comes along and executes TIP, which is responsible for actually processing the rubout. There are the following cases. It may be an ordinary rubout, which goes to RPAN; it may be a catastrophic rubout (one in which rubout was pushed twice within seven clock cycles); or it may be an initial rubout.

Furthermore, if it is ordinary, it may be one which is going to cause an interrupt, if the interrupt is on (we discussed interrupts and rubout when we talked about the fork structure). It will also be necessary to clear the input buffer, and it may be necessary to clear the output buffer if the program is executive.

If the teletype is not attached to anyone, the significance of the rubout is that a new user has arrived and we must initialize him. You remember when we talked about forks, we said there was a mystical

process which got the user started up. Well, here it is, TSON. We find the user a job number and we go through all kinds of rigmarole to get him started up. So much for rubouts.

Now, at the bottom of page 9 there is a routine called TSOF which is responsible for turning off a teletype. I don't think it has any particular peculiarities. The teletype is a controlling teletype, which means that there is a job number that has to be released and a PAC slot which has to be released in the course of turning off this teletype. This is the routine that essentially performs the function of LOGOUT and it is called by a BRS. It is possible to cause the logging out of a different teletype than your own. Notice in the middle of page 10 it says, BRU PACGO (suicide). If this routine discovers that the teletype it is logging out is its own controlling teletype, it goes straight to PACGO and the scheduler will activate somebody else.

Now there is the teletype output interrupt routine, TOI, which is pretty trivial. It is started by the interrupt which says: "I have just finished output of the characters", so it looks to see if there are any more characters to output. If there are, it picks one up and outputs it. Otherwise, it POTs out to the teletype interface a standard word which say: "I am through with you, go away." In the case where it actually outputs something, it will consider the possibility that maybe it should indicate that the guy can be activated, if the output buffer is nearly empty.

On the bottom of page 12 and page 13 there is a routine called STI, which is responsible for processing the STI SYSPOP, simulate teletype input. This routine duplicates a lot of the things in the interrupt routine; it is not possible to call on the interrupt directly, because if an interrupt came along in the middle there would be a terrible mess.

Then there is TO, which is responsible for teletype output. This is a fairly complex routine, which has to consider the possibility that the buffer might be full, in which case it will return with the dismissal condition. If it didn't have to dismiss the guy, it goes right into the buffer, just the way the teletype input interrupt routine does, putting the character to be output into the buffer. It also has to check links, which it does down here at TO6. If it finds links, it goes into the link code, which starts at the bottom of page 14 and goes on for some time. The rest of the link logic is to be found starting at the middle of page 19 and going on page 21. That part computes link words. Here is the part that actually does the linking, in this routine called TLOM, and the way this works is that at TLM3 (toward the beginning of page 15) it is picking up the link control word and doing this NOD, which essentially shifts the word to the first one bit. It takes action on that teletype and it goes on to the next one. The link logic has the following peculiarity: it goes through all the linked teletypes twice, once to make sure that there is room in all the output buffers and a second time actually to do the outputting. If it finds, in looking through the output buffers, any that are full, then it abandons the whole process and dismisses the guy. When you are outputting a character to a single teletype, you check whether its buffer is full and if it is, you dismiss. If you are outputting to linked teletypes you must make sure that all the buffers are free. TLOM is not a routine which actually does any work. What it does, is to go scanning through all the teletypes, finding all the ones that are linked to the one we are looking at. For each one it calls a routine that is given to it as a parameter so that you can call it with several different routines, depending on what you want done.

The next thing is teletype output, which is in TOF, in the middle of page 15 (the actual teletype output POP is TCO, at the bottom of page 15). Then OST is handled at the top of page 16. In the middle of page 16 there is a routine called TTYSET which is responsible for initialization of the teletype tables and clearing out the teletype interface. It has no peculiarities except the funny rigmarole that it goes through to set up the word consisting of minus the length of the teletype buffer at the end of each teletype buffer, which is the thing that makes the wrap-around work.

On page 17, there are routines to implement several BRSs dealing with things like clearing input buffers, and then on page 18 there is a routine to handle accept message bits and attaching and releasing teletypes, and also there is a routine to turn a teletype off, that is to empty out all of its buffers and set all its counters to standard values. On page 19 are the routines that turn on and turn off 8-level mode,

The most obscure part of it, I think is the part we looked at, namely the teletype input and output routines.

Input-output

Today we're talking about I/O.

First we're going to describe the sequential file machinery, and then we will go to see how it works inside. A sequential file is the most general kind of file in the system. Any I/O device may be thought of as a sequential file. A sequential file is identified for the purposes of doing input-output on it by a file number. Some sequential files are always available for input-output: that is, they are always open. In this case they have permanently assigned file numbers. Other sequential files must be opened explicitly by the user when he wishes to use them. At the time when the user opens the file, the system will give him a file number which he can use to identify it while he does his input-output, and which he also uses to identify it when he closes it.

The files which are permanently open are the teletype and a file called nothing which simply absorbs any output directed to it and returns no indication whatsoever of anything. It is a convenient file to have around when one has a program that generates output and one doesn't want the output.

The teletype files which are permanently opened are described in Section 9. They are the user's own controlling teletype, which is 0 for input and 1 for output, or any particular teletype which he may choose to specify which is 1000 + tty number for input and 2000 + tty number for output. All other files do not have permanent file numbers and must be explicitly opened.

You can open a file by using an operation called BRS 1, and what this does is to accept some identification of the file from you, cause the file to be opened and return a file number to you. In general, a user is not allowed to execute BRS 1; only the executive is allowed to execute BRS 1, and if the user wishes to open a file, he must supply to the executive the name of the file. The executive will then figure out where the file is and obtain the necessary information to supply BRS 1.What you give to BRS 1 is something called a device number (and in the case of some devices a unit number) in A. For a drum file, this would be the location of the first index block, which then identifies the file to the world. The device numbers are listed on page 9-6 with the exception of those for sequential drum files, which are listed on page 9-1. BRS 1 skips if it is successful, and if it returns without skipping this means that for some reason it was unable to open the file. When it returns without skipping, it returns some kind of status word which explains why it was unable to open a file. And after that it's up to you to hold onto the file number, since you can't do anything with the file unless you have the file number.

In the case of a device such as paper tape or mag tape, when the file is opened by one user it is denied, of course, to all other users. This can be one reason why BRS 1 could fail. The other reason which would cause BRS 1 to fail would be an illegal device number. In that case, however, it will not return at all, but will generate an illegal instruction trap.

Now, once you open the file, you can do I/O, which we'll describe in a minute, and you can also close it. You do that by putting the file number A and executing BRS 2. In case you forgot about your file numbers you can close all your files by executing BRS 8. Whenever you go back to the exec, it closes all the files for you.

Now, input-output for sequential files is done through three instructions called CIO, WIO and BIO. CIO takes a character from the A register and outputs it to the file; or if it is working on input, it reads the character from the file and puts in the A register. WIO does the same thing except for a word. BIO takes a core starting address in A and a count in X and inputs or outputs a specified number of words to memory. These are the three sequential file operations. You notice they work indifferently for input and output; the reason for this is that whether the file in input or output is determined by the way it is opened. You use different numbers for files depending on whether they are being used for input or output.

Q With BIO, is output in words or characters?

A It takes words out of core.

The success of a CIO or WIO does not depend on whether the device involved is word or character oriented, because the system will do all necessary packing and unpacking. It is, for instance, quite legal to do a WIO on the teletype. The result is that three characters will be typed on the teletype. If you say BIO with a block length of 5 (output 5 words), this may involve transmitting 5 words to the device or 15 characters, depending on whether it is a character or a word oriented device. In any case, all 120 bits will be written out.

Now, it is often convenient to be able to switch between input and output in a sequential drum file. It doesn't make sense on paper tape to switch between input and output, but in a drum file it often does. Consequently there is this operation called BRS 82 which allows you to do this switch. There is machinery provided, which is discussed in Section 12, for making a file read only. If the file is made read only then, of course, it's not legal to switch it to output; you get an illegal instruction trap if you try to do that.

There are a couple of other things to know. (All these things are written down in Section 9.3). Certain flag bits are set on every I/O operation. The flag bits are 0, 6, 7 and 8 in the file number, which is the thing addressed by CIO. Bit 0 is turned on whenever any flag is set, and the other bits indicate end of record, end of file and error conditions on the file. So you can find out whether some funny thing happened by looking at the flag bits in the file number. In addition there is a convention for what the system sends back when it sees end of record or end of file on input. End of record sends back 134 characters and end of file sends back 137 characters. If you're operating in word mode you'll get a whole word of 134s or 137s.

There's two more ways of finding out if you've got unusual conditions on the file. One is that BIO will skip if it didn't see any unusual conditions and fail to skip if it did see an unusual condition. The second is that you can get interrupt 4, if it is armed, whenever any flag bit is turned on during an I/O operation.

Some files have additional structure; they are more than just being strings of bits. Among these are mag tape files and drum files. Drum files have a lot of structure, mag tape files have less. So I think I'll dispose of mag tape and other funny device files before taking up the drum.

The mag tape is organized in the following way. The system writes 200 word physical records on the mag tape. You (the user) write logical records on the mag tape and read them, and you can do this disregarding the physical records completely; the system takes care of all necessary conversion. A two word logical record is quite inefficient, because it takes at least one physical record to hold each logical record. There are a certain number of operations provided for doing things to the mag tape other than just reading or writing. They are called control operations and they are defined on page 9-6.

To do a control operation you address the file number and put the control you want to do in the A register. The following ones are available: 1 = end of record, 2 = back space record, 3 = forward space file, 4 = back space file, 5 = write three inches of blank tape, 6 = rewind and 7 = write end of file. So this allows you to move the tape around and do other funny things on it that you might want to do.

Q These are logical records?

A Everything refers to logical records, right.

Q Is there any way to read the physical records?

A No.

Q Say you have an 80 character record. Can you read it?

A No. The only kind of mag tape records the system can handle are the 200 word ones that it writes itself.

Q So how can we read a tape with card images on it?

A There are two possibilities. I'm not sure what would be the best thing to do. One way to handle the problem would be to modify the system to handle more general records. The other way would be to write a special program to run out of time-sharing and convert the tape to TSS format. It might be worthwhile to indicate specifically what the format of a record is on the tape. It consists of 200 words. The first word contains a word count—how many data words are significant of the 199 data words actually written. Following that there are the 199 data words. The records on the tape are 200 words, one word count word and 199 data words. The sign bit is turned on for the last physical record of a logical record. There is no numbering of the records,

Now, there are some other devices. There is the paper tape reader, which also has logical records. These are exactly the same as the physical records on paper tape and are simply defined by gaps (sections of blank tape). There is also the paper tape punch. These devices have no control operations, except for a single end of record one for the punch: write end of record. This simply generates a gap on the tape by punching an inch of leader.

Then there is the card reader. Substantial transformations are made on the card reader input. First of all, code conversion is done (which it is not on any other device) to convert the SDS code that the card reader sent to the CPU into ASCII. Our card reader driver, the one that's built in, reads BCD cards but does not read binary cards. To put in a binary driver wouldn't be very hard.

There are some other transformations made on the card reader input (see page 9-7). A carriage return and a line feed are generated after each card. If you read characters from a card with 80 zeros on it you'll get 80 zeros followed by carriage return followed by line feed, 82 characters in all. Secondly, any trailing blanks on the card are suppressed. Any embedded blank strings of more than 2 blanks are converted to a 135 character followed by the number of blanks. This is the system convention for storing multiple blanks, and it also used by the teletype output routine. Finally, the card is not regarded as a logical record; there is no possible way to get an end of record from the card reader. You can get an end of file from the card reader if you push the end of file button.

That takes care of all the devices other than the drum files, which I will now proceed to explain in a little bit more detail than the manual.

A sequential drum file consists of logical records. Operations within one logical record are just the CIO, WIO and BIO that I have already described. You can also, however, perform controls on drum files, and they are considerably more flexible than those for any of the other devices. They are described on page 9-4. You can write end of record, you can backspace a specified number of records, you can for-

ward space a specified number of records, you can delete a specified number of records, you can space to the end of the file and backspace, or you can space to the beginning of the file and forward space. These are controls 1 through 6. Control 7 is insert logical record, which inserts a new logical record at the point that you are now sitting at. These operations allow you to move around freely on the drum file and to treat the logical records as independent or interdependent, whichever you choose.

Then there are a lot of other piddling operations. You can read the amount of information in your file. If you are executive you can set up an index block, give up an index block, or you can read an index block explicitly into core.

The drum is physically organized into 256 word blocks. The user, however, never sees this organization. He only sees what I have already talked about. The way this miracle is accomplished is that associated with each drum file, in addition to the data blocks which hold the actual information in the file, are things called index blocks. Any index block contains two pointers to the preceding and following index blocks. A file can have any number of index blocks, which are chained together by these pointers. The first index block, of course, has zero in the backwards pointer and the last index block has zero in the forward one. In addition to these two pointers, the index block contains the addresses of 78 drum blocks. Each word of the index block corresponds to one physical drum block and points directly to it. The bottom 15 bits are used to hold the address of the drum block which is being pointed to by the word in the index block.

The top bit of the index word is turned on if it points to the last physical block of a logical record. The remaining bits are used to specify the number of data words in this block. This, of course, will always be 256 except for the last block of a logical record. The convention, in other words, is very similar to the mag tape convention. For a logical record consisting of 700 words, there will be three index block words. The first one will point to a full 256 word data block, the second will point to a second full 256 word date block and the third one will have the sign bit turned on and will have a word count of 700 minus 512.

When you insert new logical records, the system takes care of moving the pointers in the index block. In a sequential file, the pointers in the index block are always compact. Each word points to a physical block. By scanning down the pointers, you can find all the words of the file. Now, this should make it reasonably clear how these control operations are actually implemented.

The reason for this mechanism, by the way, the reason that we just didn't put in something more simpleminded, is that this provides a facility which very many user programs need. Of course, a user could simulate the whole thing by having a big file and explicitly keeping pointers to the logical records. It's nice to have the system do it, first of all because the system can do it more efficiently than you can do it; and secondly, because it means a lot less for the user to worry about. He has a very nice and flexible piece of machinery built. It will be satisfactory for many of his applications without any modifications at all.

Q If the record is written out with CIO, does he have to read it back in with CIO?

A Definitely not. Remember, a record is not written out by a single command. A single command writes out one character or word. The only way an end of record can be specified is by explicitly giving the control operation which says end of record, or by closing the file, which implies end of record. If you mix CIO and WIO, somewhat peculiar things will happen; that should be avoided. But certainly you can write the file out with one of the operations and read it in with another.

I think that more or less covers the user interface for sequential files. I would like now to talk about the implementation. In the Tymshare system, and I presume in yours also, there will be a mechanism for actually keeping files on the disc and only bringing them onto the RAD when they seem to be needed,

since the RAD isn't big enough to hold them all. Half the disc space is reserved for your fixed data and the other half is available for files. The block operation, BIO, is done directly out of the user's memory in units of 256 words. In other words, if there is any 256 word block which falls within the scope of one BIO, it will be done directly out of the user's core and will go much faster. In all other cases, I/O is done through a drum buffer. Drum buffers are kept in the users TS storage block, where there is room for 4 of them. A drum buffer consists of a 256 word area for a data block, a 78 word section for an index block, (only 78 words are used in the index block, although it occupies a full 256 word drum sector), and about 15 words of pointers and things which the system needs in order to keep track of what's going on in the file. All of these things constitute a drum buffer.

Now, all files that are not permanently opened have associated with them things called file control blocks. The file control blocks are taken from a standard table in the system which has room for some reasonable number of file control blocks—50 or 60. It's a system parameter whose value depends on how many users you expect to run the system with. The file control block format is depicted on page 9-8. It is essentially indexed directly by the file number, so when you deliver the file number to the system, it goes immediately to the file control block which is suppose to tell it everything it needs to know about the file: first, who the user is who owns the file. This is very important, because the first thing it does is to check that the user who is trying to use the file is the same guy who owns it. If he isn't, then that's an error.

Q How are users identified?

A By the I/O number, which is not quite the same as the teletype number. Normally it is. Normally one controlling teletype equals one job. However, background jobs are possible, and they don't have teletypes.

Then there is a number which essentially identifies the file if it's one which is not sufficiently identified by the device number. This number is the index block address, the tape unit number, or the address of the subroutine for a subroutine file (which we have not yet discussed). Next is a word which is used for packing and unpacking operations, followed by a word called FD which has a whole slew of little bits telling one thing and another, and also the device number. Finally there is a word which contains the character count for the packing and unpacking operations and the drum buffer address in the case of drum files. I think it will be fairly clear to you superficially what the significance of the various fields in the file control block are. When we come to places in the system where they are used, you won't be so baffled.

All right, with this introduction, I would like to start looking at some of the code for input/output. Now, this code comes in three major sections. There's a package called IO which is essentially responsible for the user interface for sequential files. There's a package called W which is responsible for the W buffer, and there's a package called DRM which is responsible for the drum. I think that most of our attention will be concentrated on the first two packages.

I direct your attention to the beginning of the package labeled IO. It starts out with a large number of tables. First of all there are four words labeled ERRBIT, EOFBIT, EORBIT and EOTBIT, which have the appropriate bits turned on corresponding to the flag bits which can be set by these conditions. Then there's a whole bunch of symbols called X0 through X7 and XN7 through XN1, which are just the numbers 0 through 7 in the top octal digit of the word. Then some more words with useful bits, then the file control block area, which I just finished discussing, and finally the device dispatcher, which is called DEV and which has one entry in it for each of the possible device numbers. The exact significance of the entry is obscure as hell—the table called DEV is not written in a way which makes it very clear what's going on. There is a number in the address field which is either just the device number or is the address

of a special routine for the file. In the latter case, bit 1 will be turned on (,2 tag). The bottom bit of the opcode field is turned on if the device is an output device. The top bit of the opcode field is turned on for drum files and the second opcode bit for random drum files. The 2 bit is used to indicate a character-oriented device.

Q Why the funny opcodes?

A These words are definitely not instructions. It's unfortunate that Peter chose to set the opfield bits with specific opcode mnemonics; it's very confusing because you can't tell what's going on.

This DEV table also contains entries for the permanent teletype files. It is indexed by the device number, which you normally get out of the file control block. In the case of the permanently opened files, you get the device number from the file number by a little computation which we'll see.

The next table, BUF, contains the addresses of the buffers for various devices. The buffers themselves are mostly defined in the package labeled W. This table is also indexed by the device number. Then there's a dispatcher, mostly for W buffer devices, called the driver dispatcher. You go indirect though this table when you're actually ready to activate the routine which does the I/O. Finally, there's a BIO driver dispatcher which you go indirect through in activating the routines to do BIO.

Then there is ACTR, a word we all know well, and AIRWD, which is the word the system uses to arm interrupts.

Before we look at any code in detail, I want to go briefly over this whole package and explain what everything is. The first thing is the CIO driver, followed by the BIO driver. These are somewhat interconnected. Then there is the kludge BIO driver which is used to handle block I/O for devices such as the paper tape reader which do not lend themselves to efficient handling of block I/O. Next is a routine called IOI which is used to check legality of the file number. You get back from it various information which you need about the file, as indicated in the comment in front of it (this is at the bottom of page 3).

Then there is a glorious routine called GPW (on page four) which is essentially responsible for activating device drivers. You feed it the device number and the word or character on output. It takes care of dismissal, if that is necessary, and of sorting out the various kinds of devices. That is all the basic I/O stuff there is. The drivers themselves are in other packages, the drum drivers in DRM and the W buffer drivers in W.

Now there is some more stuff in this package. There is the routine to handle controls, which is on page 5. Then on pages 6, 7 and 8 there is the PDP5 link I/O, which we will definitely not look at in detail. It is significant because it is an example of how to program a DMC (data multiplex channel) within the framework of the system.

At the top of page 6 there's a routine called IOPSET which is the initialization routine for the tables in this package. Then there are some things that haven't anything to do with I/O: SBRM and SBRR SYSPOPs and the BRS table, and the code for BRS 3, 43 and 44.

The BRS dispatch table contains one entry for each BRS in the system. It is called BST and is at the bottom of page 8. Each entry contains the address of the routine to be called in the address field. In the opcode field it contains one of three things depending on what kind of BRS it is. If it is a monitor BRS which does not execute any SYSPOPs, the opcode is BRU. What you do, by the way, is to execute this instruction with EXU. The BRU will therefore send control immediately to the BRS. If it doesn't have BRU in the address field, there are two other possibilities, namely it could be a swapper BRS, or it could be a BRS which is resident and runs in system mode but calls other SYSPOPs (usually to do I/O).

BRSs of this last kind are, for instance, the one which reads numbers, BRS 38. It does a TCI and at that point will get dismissed if the guy hasn't typed a character. This means that special action has to be taken to see to it that when it gets reactivated, everything is still all right. For instance, the return location must be explicitly saved from BRSs of that type have EAX in the opfield.

Those which have NOP in the opfield are swappable and run in user mode. They require special treatment. Appropriate machinery surrounds the BST tables to take care of these three possible cases.

Finally, there are two routines called MONOPN and MONCLS which are responsible for handling BRS 1 and BRS 2, i.e. for opening and closing files.

I will now try to explain more or less the sequence of events involved in getting a CIO executed. It puts the character it is supposed to be handling in T and checks explicitly for the possibility that the file number is 0 or 1. Those are the controlling teletype files. If it's one of those two, then CIO is essentially going to simulate TCI or TCO. It does that explicitly by transferring off to FTCI or FTCO, depending on which one it was. Otherwise it comes down to CI1, where it calls IOI, which guarantees that the file number is good and sets up the device number in DEV, the file number in FILE and the buffer address (which it gets out of the buffer table that we looked at earlier) in BUFF.

Let's turn to IOI briefly, IOI has to check the file number for the possibility that it refers to a teletype file i.e. if it's equal to 1000+ teletype number or 2000+ teletype number. If it is a teletype file, it checks that the teletype number is legal. Otherwise it checks that the file number is not too big for the file control block table.

If not then we pick up the first word of the file control block and check that the job number is the same as the number of the guy who executed the instruction. If it's not, then it's illegal for the guy to use that file and he gets an illegal instruction trap. That's taken care of at IOI4 at the top of page 4.

The next thing we do is to set up BUFF and to check for a drum file, in which case we consider the possibility that it is a random drum file. If it's a random drum file we go out to DIODMS, otherwise set up BUFF from the file control block.

We now return to CIO. The main thing CIO is responsible for is doing the necessary packing and unpacking of characters to accommodate itself to whether it is working on a word or a character oriented device. This fact is determined by the setting of CHRBIT in the DEV table, as we have already seen. If it's a word oriented device, then CIO either packs the character away in the FW word, or if the word is full it calls GPW to get the I/O done. If it's a character oriented device, then the CIO transmits the character directly to GPW. Note that input files are treated differently than output files in the packing and unpacking for obvious reasons. If CIO is called for output and only one character has been provided, then that's not enough to fire the device up; you have to get two more characters before you fire the device up. If on the other hand, the device is a character oriented device, then you do the I/O directly by calling GPW.

We haven't nearly gotten to the point of doing physical I/Os yet, you realize. We've only gotten to the point of calling GPW, which is still pretty far away from doing physical I/O. The function of CIO is simply to accomplish the character to word conversion that may be necessary.

Now, WIO is a very similar operation which has similar problems. If it is working on a word oriented device, it transmits the word directly to GPW or gets it from GPW. If, on the other hand, WIO is working on a character oriented device, then it calls WIT, which is just after WIO and is essentially responsible for doing the character to word conversion. In other words, if you deliver an output word to WIO it has to break that word apart into three characters and call GPW three times to get the three characters output. On

input, it has to collect three characters to make into a word before it can give the word to you, since you asked for a whole word.

Q Does GPW stand for something?

A It stands for Get and Put Word.

Then there's BIO, which is essentially like WIO except that it has a loop. It does the job the necessary number of times. Notice that the BIO either calls GPW directly, for word oriented devices, or it calls WIT.

Q What's the BRU* BSEL2?

A There's a BIO dispatch table which we saw a little earlier, BSEL. The first thing the BIO does is to branch through the dispatch table. That will send it in most cases to BIO which handles the normal BIO. In the case of the drum it will send it off to an entirely different place which handles drum BIO.

Now we come to GPW. It puts the device number in X, picks up the word out of the DEV table and looks at the X2 bit. If the X2 bit is turned on, the word has a special driver. This is the case for most devices. The special drivers perform a wide variety of functions, as we shall see. If we don't have any special driver, we just go to these three very interesting instructions which test whether the contents of the word BUFF is bigger than the contents of the word addressed by BUFF. To explain why we're making this test, I will have to explain the structure of the system I/O buffers. Such a buffer consists of n+2 words. The first 2 words are the header and the remainder of the buffer is data. The convention is that the first word of the buffer points to the currently active word of data and the second points to the end of data. If the first word of the header is less than its own address, i.e. if it does not point into the data, then the buffer is busy; that means you are not allowed to perform operations on it. If the buffer is busy, normally I/O is actually being done on it. The driver has been called and the channel is reading out of or into the buffer. During that time, it is obviously not acceptable to allow programs to write into the buffer or read from it because the state of the buffer is unknown.

Q You've only got one buffer?

A One buffer for each device.

Q Why not more?

A More complicated than it's worth. We considered having double buffers, but we decided the amount of overhead and miscellaneous junk added were just far too big to be worthwhile. You can do your own double-buffering with forks, so we save a lot of energy in the system as a result of not having to do that.

So the function of CXA; SKG 0,2 is to test whether the buffer is busy. BUFF, of course, has been set up by IOI to have in it the address of the first word of the buffer, so whenever the buffer is not busy, the first word of the buffer is pointing into the buffer, to the currently active word in the buffer. So if the buffer is busy, we have to dismiss the program, and the way we do that is to set up activation condition 6 with the contents of BUFF in the address field. We go to POPDMS with QIO in X, which means the guy is going to be dismissed on to the I/O queue. Activation condition 6 is a screwball activation which says "Test whether the word is equal to the address of the word plus 2." That is the normal state in which the buffer will be left when the channel is finished whatever it's doing. When the interrupt routine comes along and sees that the buffer should not be busy anymore, it will set up the first word of the buffer to point to the third word, which is the first data word.

If the buffer is not busy, we go to GPW3 (near end of the page), which is actually going to do the I/O. It picks up the second word of the buffer, subtracts the first pointer, and looks to see whether the address field is 0. If so, then the buffer is empty. The top of the second word is used to hold flags, the flags that I

mentioned in connection with CIO. This is why we test for 0 with SKA. If the buffer is empty, we test to see whether any flag bits are on. If there are no flag bits, we start the device. If there are flags, we go to GPW8, which is responsible for transmitting the flags back to the file number and generating an interrupt if that's necessary. Here we test for the error bit and the end of record bit and go off to GPW8A or GPW8B depending on which one is on. If it's not error or end of record, it must be end of file, in which case we go through all the gyrations at GPW12. When you come to GPW12 the situation is like this: you have in the B register either a word full of 134 characters (for end of record) or a word full of 137 characters (for end of file) and in the A register the appropriate flag bits. GPW12 puts the flag bits into the top of the file number and stores the result in the word addressed by the I/O pop.

Then it sends B to A and extracts with GPWC, which will contain either 377 in the case of character I/O or -1 in the case of word I/O. The resulting word or character will be transmitted back to the user if input is taking place. That's what all the machinations with FC and FD are. Next we call IIR, which is the routine which generates an interrupt. The 2000B tells what interrupt to generate (4 for unusual I/O condition). If the IIR succeeds in generating an interrupt, we return immediately, because the program will be sent off to the interrupt location. If the interrupt is not caused, we just return to GPW in the normal way.

If we get to GPW8B, then we have an end of record rather than end of file. We make sure that the only flag which is turned on is the end of record flag, because we're not interested in the others. Then we proceed as we do with end of file. If we get to GPW8A, then we have an I/O error and go through similar but slightly different machinery, which can also generate an interrupt and do various other things, but doesn't generate the 134 and 137 words. We won't look at that in detail. Note that all the physical machinations involved in the error have already been taken care of long ago. For example, if it was a mag tape error, the system has already made all reasonable attempts to correct the error. The user sees the error only when the system has decided that there is no longer any hope for correcting it. This can happen after many attempts to correct in the case of mag tape, or it could happen after no attempts to correct if for instance it's a card reader which the system has no chance to correct.

Now let's return to the normal case at GPW3 at the bottom of page 4. We have just succeeded in convincing ourselves that there aren't any flags, so we're going to start the device up in the normal way. This is actually what is going to get us down to the driver. What we do is to pick up the file number into A, and the old original contents of the X register, which was saved away in SSO3 this whole time, into X. Then we look to see whether 0 has the indirect bit on or not. If it has the indirect bit on, then GPW was called out of CIO or WIO, and we want to clear the flag bits in the file number. So we store the file number with no flag bits, indirect through 0. We restored X to allow for the possibility that the POP was indexed. If on the other hand, GPW was not called directly out of a POP we avoid this step.

We are now ready to call the device. We pick up the device word into B and check to see whether it is a drum file. If it is a drum file we go to GPW7. Otherwise we move the device into X and pick up the proper word of SEL, which is the driver dispatcher table in the middle of page 2. We pick up BUFF into X and call EDW. EDW is the W buffer device driver and it is found in the W buffer package, which we will look at later. It has three possible exits, depending upon the degree of disaster which occurred in the attempt to activate the W buffer. The idea is that in all cases when you activate the W buffer, you get the guy dismissed, because always you have to wait for the I/O to finish. The way you get him dismissed can vary. If EDW doesn't skip, it means that everything went through, and you go to IOPDMS which generates the dismissal condition which says wait for buffer to become non-busy. Otherwise you would go to IOQDMS which means that EDW has generated its own dismissal condition which it's put into B already. This will be some different dismissal condition which will have been generated because the W

buffer was tied up in some obscure way. We'll see what these things are in more detail when we look at EDW, but is the idea clear?

If it was a drum file, on the other hand, GPW7 goes either to DRMSO or DRMSI, depending on whether it's an output or an input file. Those routines are in the drum package.

Now there are some other things in GPW which are worth pointing out. GPW is equipped for handling subroutine files, and when we come to discuss subroutine files as part of the user interface, we'll look and see how it's done. In addition, GPW has special drivers in it to handle mag tape and to do a couple of special things about the drum. These special drivers are accessed through the address field of the words in the DEV table.

You've now seen I/O carried down to the point where we are actually ready to activate the physical device and I think now that rather than go and look at controls, which are the other important thing in SIO, we will go and actually look at the W-buffer code.

The W buffer has two interrupts, 31 and 33, one of which happens to be for end of word and one for end of transmission. We just used 31 exclusively except in the mag tape routines. Now, the difficulty with the W buffer arises basically because when an interrupt comes in, you don't exactly know what to do with it because it could have been caused by a lot of things, depending on what it was that last set up the W buffer. The way this problem is handled is that there is a cell called BLK31 which contains the address of the subroutine which is responsible for the next 31 or 33 interrupt, or 0 if no interrupt is expected. An I31 interrupt always comes to INT31 at the top of page 2, and you notice that INT 33 is equated to INT 31 so the 33 interrupt comes there too. What this routine does is to store the central registers, pick up the contents of BLK31 and do BRM* through it after setting BLK31 to 0.

On to returns from the I31 interrupt. Before returning, however, it looks at BLK31, which was set to zero before the subroutine was called, but which may have been changed by it. If BLK31 was changed, namely if it was made bigger than zero, then that essentially means that nothing interesting happened. The W buffer is still being used by the same device that was using it before. If it was left equal to zero, the interrupt routine decided it didn't want the W buffer anymore. In that case, ACTR is incremented, since the I/O operation is complete and the program waiting for it can be activated.

Q What is the significance of the skip?

A We'll see that when we look at these routines. It has to do whether you need to call the phantom user or not.

Now, the W buffer is always activated through the routine called EDW. It looks to see whether the W buffer is busy and if it's not, it starts the device. If it is, it puts the request on the phantom user queue. EDW is called with the buffer address in X and the contents of the appropriate word of the SEL table, which is the device driver address, in B. It stores the driver address in EDWD. Then it tests to see whether the buffer is busy. If it is, we go to EDWB and dismiss with activation condition 6; EDW returns with one skip. If we get past the buffer busy test, we compute the distance between the contents of the first word in the buffer and the beginning of the data part of the buffer, which is the third word. I.e., we compute the number of data words, and put it back in the first word of the buffer. Now, we check to see whether B31 is zero or not. If not, there is some interrupt pending on I31; in other words, someone is using the channel.

In that case, we go up to EDWS. Otherwise we call the driver with BRM* EDWD. If this returns without a skip, the device has been started successfully. Otherwise we also wind up at EDWS; this happens if the driver decides that the device cannot be used, e.g. because the tape is rewinding or the card reader is not

ready. EDWS puts something on the phantom user queue. It sets up the file number in A. Then it disables the interrupts and calls EPU, and then it MINs EDW so that in this case EDW will skip twice on returning. EPU is simply a routine which you will find around the middle of PAC which puts something on the phantom user queue.

To summarize, there are three possibilities. The first is that we call the driver and return normally. The second is the buffer was busy, in which case we come down to EDWB, set up a dismissal condition to wait for the buffer not to be busy and return with one skip. The third is that we cannot initiate the operation for some other reason in which case we come to EDWS, put an entry on the phantom user queue and skip twice.

We will now proceed to look at some devices and interrupt routines in detail. I think a good one to start with is the paper tape reader, because it's so simple. It is at the beginning of page 3. First is the reader buffer. It contains RTCNT+2 words. Next is the reader driver, RTX, at the top of page 3. It says SETINT RTI—You will find the macro defined in MDBG; what it does is to put RTI into BLK31—and does the necessary EOMs to fire up the W buffer for paper tape. Then it returns.

RTI is the interrupt routine. It does this ACW which tells how many words got read in off the paper tape. It initializes the buffer by putting the address of the first data word of the buffer into the first pointer word. It tests for error and end of record and sets the bits in the second pointer word of the buffer, which as you will remember from GPW, is where the flag bits are kept. Then it returns. You notice that it does not tamper with BLK31. This means it is always assumed in the case of paper tape that one interrupt equals one operation, and the paper tape reader routine releases the W buffer after performing one operation. The W buffer will not be grabbed again until the guy who had the paper tape does some more CIOs, since he must exhaust the characters which have just been read in first.

Next we look at the paper tape punch driver; it is a good example of an output driver. The whole thing is enclosed by this IF PEXF, which means you can suppress it if you don't have a paper tape punch. Then comes the paper tape punch buffer, which is called PNBUF. It is preceded by a magic word which contains the address of the word immediately following the punch buffer. An output buffer always has this word preceding it. The paper tape punch driver is called PNX, and you notice it's a little more complex than the paper tape reader driver. The tail of it is the same: It fires up the punch and exits. But before that, it computes the word to be filled out and stores it in PNI. This is necessary because the number of words to be written depends on the number supplied by the user, whereas a full block is read on input. PENOR is a routine which is called by control 1 for the paper tape to punch an inch of leader.

The other routines are more complicated because of idiosyncrasies of particular I/O devices. The card reader has a conversion table which converts SDS internal code into ASCII. It also does a number of other interesting things which we have already discussed. This is all done in the card reader interrupt routine, and that is why the card reader interrupt routine is so complicated. The driver is still pretty simple. It's pretty much the same as the paper tape. The difference is that it starts by executing CRTW, the test for card reader ready. If the card reader is ready, it skips and goes on to set the interrupt. If the card reader is not ready, it goes to CRX1 where it tests for end of file. If it doesn't find end of file, CRX returns with a skip without doing anything. This causes the program to be dismissed and a phantom user entry to be made. The function of the PU entry will be to keep testing for card reader ready at suitable intervals. If there is an end of file, then we normalize the buffer, turn on the end of file bit and return without doing any I/O. That just transmits back to GPW the information that end of file was received. The interrupt routine we won't look at because it's sort of complicated, and the complexity arises entirely from these transformations it's performing.

Next thing we come to is the glorious mag tape routines, which start on page 6. We're not going to look at those too carefully because they are not simple, but we'll go over all the routines and point out what they all do. First of all, there's this big table of things, each consisting of one call of TDT, which is another one of the system macros. Its function is to make two of the same EOM for two different tape units, 0 and 1. The SDS method 6or driving tapes has a serious drawback; the tape unit number is embodied in the I/O instruction itself. This means that you actually have to physically change the instructions to handle different units. Whenever the tape routines want to do anything with the tape, instead of directly executing an EOM, they put the tape unit number in the X register and do an EXU on one of these little two word tables.

Then there are some odds and ends like the proper word for writing file marks; TNO which tells which tape number is currently being used; and TXS1, which is where the system reads in how many words were actually transmitted by the W buffer. TXS2 and TXS3 are obvious. Then there is the tape buffer, TBUF. There a special routine for opening the mag tape called MTPOPN, which is called from the BRS 1 routine which is in SIO. Its function is to test a number of things untested by the normal opening routine. Among them are the fact that a program must be executive in order to open the mag tape, that there are various conditions like the tape not being ready under which the attempt to open it may fail, and that the monitor has in it a cell called MTFL which keeps track of how many words have been written on or read from the tape, and it forcibly prevents you from writing or reading any more words than that. This call is initially set up to be the largest possible number. It is usually reset by the exec. Next is a little routine called MTDI which I don't pretend to understand, whose function is to turn off a runaway tape unit. Then there's the tape reader driver, TRX. It checks for tape not ready and end of tape. If neither condition is present we do the read. We don't do it directly but call T21, which is one of the routines that's also used by an interrupt routine.

You see, life for the mag tape is a lot more complex than it is with other devices, because of the fact that in the interrupt routine you may wish to re-initiate something. For instance, if the read takes place and the interrupt routine sees than an error has occurred, what you want to do is to initiate a backspace followed by an attempt to read again, all in the interrupt. This means the tape unit is run through lots of tittle tiny subroutines, all of which call each other. These are explained by the comments. A noise record is a record in which you read less than some fixed number of words. Such a record is ignored.

Then there's a lot of machinery to enable you to backspace the tape correctly to try again on read error or write errors.

The write driver, TWX, is in the middle of page 8. It has special features which allow you to write a three inch gap at the load point, because it turns out that if you try to write on the tape at the load point, it gives unpredictable errors. Then there is another routine to write a three inch gap anywhere. It is used to make the spaces between files. There is a routine to write file marks, and all kinds of machinery to clean up after errors. That takes us down to the middle of page 9.

We write a lot of extra space after each file. This is to make our overwriting scheme work. The way we have been handling our mag tape is that when you write a file on the tape you write a fixed number of words, followed by a good deal of blank tape, followed by a file mark. When re-writing the file, you write the same number of words, followed by a smaller amount of blank tape, and space forward to the file mark. The logic to do this is in the exec, but it makes use of the control operations implemented here. The function of MTFL is to prevent too many words from being written. When a tape file is closed, the exec writes out enough zero words to bring MTFL to 0. You never overwrite the file mark; it stays there forever. Using this device we have successfully been able to overwrite an IBM format mag tape.

At the bottom of page 9 is the routine to change the mag tape unit number. It checks to see whether the tape is busy and whether the user is executive. In either case, the request is illegal. Otherwise the tape unit number is set as requested. Then there are the mag tape control routines on page 18. These are responsible for taking care of forward spacing and back spacing. You see a lot of commands. about `I31`s and `I33`s. I don't think we'll look at these too carefully; they're complicated.

There's also rewind and write end of record and erase and all kinds of things—all pretty straightforward. I should probably describe the operation of forward space and backspace file. The forward space file routine scans forward until it sees the file mark and leaves the tape positioned immediately after the file mark, before the first record of the file. Backspace file backspaces over one record, which it ignores—it doesn't matter whether it's a file mark or not. After that, it backspaces until it sees a file mark, and then forward spaces over it, positioning the tape immediately after the file mark. That exhausts the W buffer routines.

Q What about the disc?

A I don't know anything about this since I never programmed it and we don't have one, but it seems to me that the following considerations are relevant. The disc, first of all, is a device which will be running all by itself on its own channel. Secondly, it is always accessed through several levels of system routines. And thirdly, it's always accessed in the same way. In the case of our file system, you always read 256 word blocks; and for your fixed data you always read 64 word sectors. There's nothing funny going on at all. You don't have requests raining in on you from all kinds of strange places.

Furthermore, the disc doesn't provide you with a whole lot of unorthodox conditions like backspace and rewinding and all these strange things that the mag tape requires you to do. Finally, it has seeks, and it has these eight independent arms. It is highly likely that over a fairly short period of time, like a second, you will get a lot of disc requests. It seems to be extremely desirable to optimize the disc, because if you just sort of go seeking around at random, accepting and processing each request as it is arrived, you will waste a lot of time. When you have to move the arm a long distance, you'd like to be able to service more than one request, since you're going to tie everything up for a pretty long time.

Subroutine files

I think then that I will discuss some things that I haven't yet mentioned about the user file interface. After that we will look at the drum I/O if we have time.

There are two kinds of files that I haven't described. There are things called subroutine files, which are a special kind of sequential file, and there are also random drum files, which are not like sequential files at all. Subroutine files and described in section 12 of the working document. The idea is pretty simple. It is possible to open a file explicitly with `BRS 1` and specify that the file, instead of being associated with a physical device or with some index block on the drum, it is to be associated with a subroutine which you yourself provide. You must specify whether this subroutine is an input file or an output file, and you must also specify whether it is a character oriented or word oriented subroutine. There after, whenever input/output is done to this file, it will result in a call on the subroutine. If, for instance, it's a character oriented subroutine, and I do a `CIO` to it, the subroutine will be called with the contents of `A`, `B` and `X` exactly as they were when t did the `CIO`. It will be expected to return whatever it likes, and in particular, it's suppose to return the character I asked for in the `A` register. If it's a character oriented operation to a character oriented subroutine, there will be no transformation done on the `A`, `B` and `X` registers. That character will be just shipped right back to the tty. If, on the other hand, the subroutine was word oriented and the call was `CIO`, the first `CIO` would cause the subroutine to be called, and the word the

subroutine returned in the A register would be unpacked into three characters, which would be delivered to three successive CIOs.

There are two points to be aware of. The first one is that the map is not changed between the execution of the CIO and the calling, of the subroutine. This makes the whole feature not quite as nice as one would like. What we would really like to be able to say is, "Look, I'm making this fancy program that's doing I/O and I want it to be completely separate of whatever calls on it." Unfortunately the overhead of changing the map is too great. But aside from that restriction, the subroutine is really independent of the things that call it, and it can do any amount of computation it wants including calling other subroutine files or doing any other. I/O that it wants, before returning. The significance of this is that you can write a program that accepts as one of its parameters the file that it works with. You may originally have in mind that those files were to be the teletype, for input end output, but later on you may decide that you'd like to drive it with another program. Well, rather than go through the program and find all the I/O instructions, you can simply change the file to a subroutine file, so that in effect, whenever this program wants I/O it will call a driving program.

The effect of a subroutine file is to make a CIO equivalent to an SBRM except for the packing and unpacking operations involved.

Random drum files

There is another kind of drum file called a random drum file. It is to be regarded as an extension of core. Its structure is quite different from that of a sequential file. For referencing random drum files, there are four SYSPOPs called drum word input (DWI), drum word output (DWO), drum block input (DBI) and drum block output (DBO). All of this is discussed in detail in section 10 of the working document. A random drum file I/O operation requires a file number, which you provide by addressing it with the SYSPOP as usual. It also needs the address in the file of the word that you want to read or write, and possibly a core address. Drum word input and drum word output do not require a core address, they go through the A register. In other words, if you put into the B register an address in the drum file and do a DWI, this causes that word of the drum file to be brought into the A register. If you do drum word output, the contents of the A register is stored in the file at the address specified by the B register. It is also possible to do block transfers on a random file, and you do that with the drum block input and drum block output operations, which take the file address in B, the initial core address in X and the number of words in A.

Q That is the address?

A A drum file is to be thought of as a very large core memory, the addresses starting at zero and extending as far as you want, so you can, for instance, address word 10, you can address word 6420, or you can address word 4296504. The storage allocation scheme for drum files is the same as it is for core memory, namely a particular 256 word block in the file is not assigned to you unless you address that part of the file, so that for instance, if you have addressed word 10, word 1000 and word 10000, exactly three blocks will be assigned, namely the block containing word 10, the block containing word 1000 and the block containing the word 10000. All the intervening space will have words allocated for it in the index block, but the index block words will be zero, indicating that no physical drum block is assigned.

So the index block is used to keep track of things. In order to get at word 10000, the system divides 10000 by 256. This tells it what block to use, and it goes and picks up that word of the index block. If that word is non-zero, then it knows where on the drum to find that block and it brings it in and

uses the remainder of the division by 256 to select the word within that block you addressed. If, on the other hand, the index block entry is zero, then it must create a new block, which is zeroed out, of course, before it can do the operation. The significance of this is clearly that you can have a random file in which you have built tables in several different places, which relieves you of storage allocation problems, without your having to worry about wasting a lot of space in between, because these spaces are not actually assigned. The only space that's wasted is the space in the index block, which is 1/256 of the total amount of space and therefore pretty negligible.

Q How efficient is this?

A How long does it take to write or read one word? I don't know—about 100 microseconds, something like that. The block operations are very efficient, especially the large ones because they go directly through core the way BIO does.

There's an additional frill: namely, it is possible for you to specify that one random file is to be your *secondary memory*. What this means is that you have available two more operations: load A from secondary memory (LAS) and store A in secondary memory (SAS) which are described on page 10-3. These operations are essentially equivalent to LDA* and STA* except, of course, that they refer to the drum file. You can use indexing, if you want, and indirect addressing. Needless to say, these operations are slower than LDA and STA. I think that they take something on the order of 100 microseconds whereas the DWI and DWO take about 150. Of course if you bounce around in the secondary memory, it will be a lot slower than that because you will have to keep swapping blocks in and out of the one buffer that's available.

Q What about interference between users.

A Everything is going on within one particular drum file. When you declare a file to be secondary memory, then you are addressing words within that file, which is disjoint not only from everybody else's files but from all of your other drum files.

It's also worth pointing out (see the bottom of page 10-3) that it's possible to release space in a random drum file. If you release whole blocks they are explicitly thrown away. This operation is essentially equivalent to BRS 4 logically. Naturally they are handled quite differently.

Drum I/O code

We will now pass on, I think, to look at DRM, rather casually. All the drum EOMs are to be found in MDBG, by the way. The repeats after DBUF are being used to set up words which contain the addresses of the various drum buffers and index blocks in the user's TS block. The repeats are controlled by how many drum buffers and index blocks there are supposed to be. There is a thing called DRQ, which is a queue for drum I/O requests.

On page 2 DRMSET initializes the drum I/O by clearing everything out. DRMOPN and DRMCLS are routines called by BRS 1 and BRS 2 when drum files are being opened or closed. DRMOPN is essentially responsible for finding a free buffer somewhere and initializing it and reading in the initial index block.

On page 3 we have LAS and SAS, which are the code to implement those two SYSPOPs. Then comes IDM, which is the drum interrupt routine. The drum is set up with a list of drum commands, and there are two words at the top of page 4 which essentially specify the list. Then there is a whole lot of rigmarole to deal with drum errors, setting the next command and one thing and another. DPU on page 5 is where the drum I/O goes when it has to dismiss itself onto the phantom user queue. This happens mostly in cases having to do with multiple index blocks. The guy asks for an I/O operation and you need several

drum references before you can carry out what he wants. For instance, suppose that he asks to read in a logical record. If the logical record you have in there now has been changed (this would be for random files) you have to write it out, and you may have to get a new index block to find the next thing, and then you have to read in the data for the next thing. In the course of doing all these gyrations you may have to dismiss the guy several times. The way you do this is by going on the phantom user when you have to wait.

DRMSI and DRMSO on page 5 are the sequential I/O drivers. They are called from GPW when GPW is ready to transmit a word to or from the drum. Starting on page 6 there is a whole slew of drum routines which do what it says in the comments. These routines are called on by the other parts of the drum I/O. On page 9 we see DTA, which is responsible for acquiring a new drum block. The logic for acquiring an optimal block is the same as that we described for getting an optimal block for swapping. We have this bit table which keeps track of which blocks on the drum are assigned and which are not assigned. The table is larger than the swapper table, of course.

On page 12 we see the random drum file routines which are responsible for taking care of DWI and DWO, and on page 13 is the code for fast block I/O to the drum, both BIO and DBI-DBO. It is quite messy because it has to allow for the possibility that the block transfer is not even. If the block transfer is an even number of 256 word blocks, then there's no difficulty; you just start the drum up in the user's core, reading or writing 256 words blocks. The only tricky thing is you have to compute the absolute core address corresponding to the relabeled address. The drum does not go through the relabeling. Unfortunately, there are several complications. It is possible that the I/O can start and end not on an even 256 word block, in which case you have to go through one of the drum buffers to take care of the starting and ending sections, since the drum always works in 256 word blocks.

Another possible complication is one we discussed in connection with hardware, namely that a 256 word block may overlap page boundaries and possibly may not be contiguous in absolute memory. If that happens, you have to invoke the drum channel map, and the code at the top of page 14 is doing that. It's got an IF in it which decides whether it uses the map or forces the I/O to go through the buffer.

On page 15 we have an important routine which is essentially responsible for getting a data block into core. DTG is called whenever any part of the drum I/O wants to get a new data block. It's responsible for cleaning up the old state of the buffer, whatever it is—this may involve writing out an old data block if it has been changed—for getting a new index block if necessary, and for getting the new data block in. It may call on the routine to create new data blocks if it needs to get a new one. Then on page 16 there's some miscellaneous junk—routines for declaring random files to be secondary memory, changing the mode of sequential files.

On page 17 are the routines to for handling index blocks. Then you have on page 18 a routine to delete the contents of a drum file and one which counts the length of a drum file.

On page 19 CBRF deletes information from a random file; that's the operation equivalent to BRS 4. On page 20 there are operations for inserting and deleting logical records from sequential files. These routines call on the bunch of routines that we passed over earlier to do the actual work of moving the pointers around, as do the routines to implement the control operations for sequential files. All the things on page 20 are called on by the various drum controls, and to see exactly how that is done you should go back and look in SIO.

SPS

SPS comes in the following pieces. There are some basic things that can be used by themselves, the stuff for hash tables, and the stuff for input/output. We'll start with the basic things and go systematical-

ly. We already discussed what a string pointer is—it's a two word object, the first word containing the character address of the character before the first character of the string, and the last word containing the character address of the last character of the string. A character address is the word address multiplied by three plus zero, one or two. Characters are 8 bits long and are stored three per word.

The following wonderful operations are available. LDP and STP, which are double word load and store, are of course used whenever you have double words, not just for string pointers. They take much longer than LDA; LDB, about six times as long. Then there are operations for reading and writing strings. There's GCI—get character and increment—this is sort of basic. If there are any characters in the string, i.e. if the second character address is greater than the first character address, it reads out the first character of the string and puts it in the A register, increments the first character address by one, and skips. In other words, it reads out the first character and takes it out of the string. If the string pointer is null, that means if the second character address is equal to or less than the first character address, GCI returns without skipping. It preserves the X register and clobbers the B register, and takes 84 microseconds.

Then there is WCI, which is sort of complimentary to GCI. It writes the character on the end of the string which is pointed to by the pointer addressed by the WCI. It doesn't look at the first word of the string pointer at all, but simply writes at the character address specified by the second word and increments the second word by one. Now there's a little goody called GCD, which is exactly like GCI except that it reads from the end of the string rather than beginning. This means that a WCI followed by a GCD leaves you right back where you were.

Then there are the string comparison operations—skip on string equal and skip on string greater. They shouldn't cause any great problem. And that's all of the basic things.

Next, there are the string input/output instructions. For output you have BRS 34, which has an elaborate convention which is described in the manual and which I won't describe in detail; and you have BRS 35, which outputs strings. For input there is BRS 37; you give it a string pointer and a terminating character and it reads characters until it comes to the terminating character and appends them to the string that you gave it.

Finally there is the stuff that has to do with hash tables. A hash table is a table of three word entries; the first two words are the string pointer and the third word is a so-called value word and is at the disposal of the programmer. A hash table is used for putting strings away in such a manner that we can look them up afterwards. The exact mechanism which is involved is this: you can look a string up in a hash table by using BRS 5 in a manner which I will describe in a minute. You give BRS 5 a string, and from this string it computes a more or less random number. The algorithm actually used is that it takes the first four characters, adds them to the last four characters, folds the result into a 12 bit number by adding the two halves of it together and multiplies by the length. The precise algorithm is not critical—it is only necessary that it produce more or less uniformly distributed numbers. Then you take this number, modulo the length of the hash table, you make sure that it's an even multiple of three and start at that point in the hash table looking for the string. If you find the string before you find an empty entry in the hash table, then it is in the table. If you find the empty entry first it's not in the hash table, because when a string is inserted in the hash table, it is always inserted in the empty entry which you find when you discovered it wasn't there, using BRS 5. When you insert a string in the table, you insert it with BRS 6, which puts it exactly in the empty entry found by BRS 5. To make it convenient, BRS 5 skips if it finds the string. It doesn't skip if it doesn't find it and you can put the BRS 6 right after the BRS 5 if you want to insert it. This means it will automatically execute the BRS 6 immediately if it doesn't find the string. If you don't want to automatically insert it, of course you don't have to.

There is one thing to bear in mind: you can't delete entries arbitrarily from the hash table. If, say A, B and C have been inserted and have wound up in sequential locations, and you delete B, then when it searches for C it will find the empty entry you left by deleting B first, and will assume that C is not in. An empty entry is indicated by zero in the first word. What you do instead in order to delete an entry is to put -1 in the first word. Then it will skip over it during the scan, but if you insert something, it will put it in a -1 entry if possible.

After a successful BRS 5 or a BRS 6, you get back two useful pieces of information: the third word of the three word entry, which is the value; and the address of the first word, so you know where the entry is. To call BRS 5 or 6 you put into X the address of a three word table and into AB the pointer to the string you want to look up. The three word table contains the address of the beginning of the hash table in the first word, the address of the word after the end of the hash table in the second word, and anything you want in the third word, which is used for temporary storage.

Now, in addition to BRS 5 and BRS 6, there is BRS 37, which is a little bit different and is used for command recognition. I'm not going to describe it in detail because it explains in the manual exactly how it works. BRS 37 takes the address of a control table identical to the one used in BRS 5 and BRS 6. It should be pointed out, however, that if you are going to use BRS 5, you can only look up what was entered by BRS 5 and 6. Otherwise the hash table algorithm obviously isn't going to work. BRS 37, however, will work on any table. Because it has to scan a table linearly, it doesn't use the hash coding algorithms. Therefore, you may construct the table to be used by BRS 37 explicitly in the assembler by simply setting up the string pointers properly. You cannot construct the table to be looked up with BRS 5 that way.

OK, that covers SPS with one exception, which is WCH, which addresses a little table which essentially defines the boundaries of the string storage. It works like WCI except for what it does when there isn't any more room in the string storage. See the manual for details.

Summary

The monitor features we have discussed fall into the following categories. There are things having to do with the scheduler, things having to do with fork structure, things having to do with memory, things having to do with teletypes, and things having to do with input-output generally.

Scheduler

The significant characteristics of the scheduler are its three queues; one for teletypes, one for I/O, and one for compute limited processes. The entries on these queues are slots in the program active or PAC table. Each entry has associated with it all the information necessary to define a process, i.e. the contents of the central registers and the map, together with a lot of other information, such as the user number and the fork structure and the interrupt mask and other things that you will find in the picture on page 3A. Also a word called the activation word, which tells under what circumstances the process should be activated, and a word which contains information as to how long it should be allowed to run. An activation word consists of an opcode field which specifies which activation condition is involved, and an address field which tells where you go to actually find the thing to be tested. Activation conditions are usually pretty simple-minded, They allow you to test for things like negative words, positive words, words bigger than the value of the clock and simple things like that.

When the scheduler is entered, it starts at the top of the teletype queue and runs through all three queues in order looking for a process which can be activated. You will notice that a process on the bot-

tom queue can always be activated, since the only reason it was dismissed was because somebody else wanted to run. The processes on the teletype and I/O queues cannot necessarily be reactivated; they can only run when whatever input-output operation it was dismissed for is complete.

When the scheduler reactivates the process, what it does is to simply start it executing at the location specified by the program counter word in the PAC table entry. This means in the case of input-output operations, that the instruction which caused the program to be dismissed would be re-executed when the program is reactivated. This means that this instruction must be organized in such a way that it doesn't mind being re-executed. To give a specific example from the things we have been discussing the last couple of days, with reference to the new disc interface, the only instruction which could cause dismissal is the one which says, "Wait until all my disc references are finished." So what this instruction will do is very simple. It will look at the disc reference counter and if it is bigger than zero, it will merely dismiss the guy waiting for the counter to become less than zero. Where it is reactivated, it will test the counter again. This time it is bound to be less than zero.

These are the characteristics of the scheduler. We have these three queues which allow you to determine the priority of a process by the queue you put it on. Apart from that it works essentially on a round robin basis. It is worth noting that there can be processes which are not on any scheduler queue, in particular the running process. Processes which are dismissed waiting for subsidiary forks to terminate are not on any scheduler queue either. They do not need to be. They are pointed directly by the PFORK pointers of the subsidiary fork.

The scheduler has timing mechanisms built into it. A program has a short quantum and a long quantum, each of which is variable. It is always allowed to run for the full length of the short quantum unless it dismisses itself, and whenever it is reactivated it is given a new short quantum. The process is allowed to continue to run for its long quantum if no process waiting for input-output is ready to run. If any such process is ready to run, then the process running in its long quantum is dismissed. In this case it will be given a new long quantum when it is restored. If the process running in long quantum is dismissed because it does I/O, it will continue with the same long quantum that it had before when it is restarted.

Fork structure

Then there is fork structure. The user is able to create essentially an unlimited number of processes, each one being more or less independent of the others, with its own central registers and its own map. A process can create subsidiary processes by executing the appropriate system call BRS 9 and specifying the location of a table which contains initial settings of the central registers and the map. The subsidiary process commences to execute and proceeds more or less independently of the main (creating, controlling) process. The controlling process can specify the way in which memory is allocated to the subsidiary process. A controlling process may read the status of the subsidiary process, which includes the contents of the central registers, the map and a status word indicating that the subsidiary process is doing. It may dismiss itself waiting for the subsidiary process to terminate and it may kill the subsidiary process.

The subsidiary process runs, except for these operations, independently of the controlling process. It can terminate if it executes an illegal instructions, causes a memory violation or executes a BRS 10, the third being considered a normal means of termination. It can also be terminated if its controlling process is terminated. Finally, it can be terminated by rubout.

The rubout button on each controlling teletype (that is, each teletype on which a user is entered as opposed to teletypes which are attached), is connected to one of the user's forks. When a rubout occurs, it causes this fork to be either interrupted or terminated. The fork will be interrupted if interrupt 1 is armed; otherwise it will be terminated. The termination looks exactly like the termination produced by

BRS 10, except of course that the BRS 10 instruction is not actually executed. In the absence of specific action by the user, the highest user fork will be the rubout target, because when the executive fires up the highest level user fork it will propagate the rubout to it. There is an option bit in the word which specifies the initial state of a fork which allows you to propagate the rubout button, if you had it before.

The other important thing about the fork structure is the interrupt mechanism. Every process has associated with it 20 interrupts, 4 reserved by the system and 16 available to the user. It may arm or disarm any or all of these interrupts. A fork may also generate an interrupt by putting the interrupt number in the A register and executing an appropriate system call. The interrupt signal is propagated to all the parallel forks and then up to all of the creating forks or ancestors of the fork causing the interrupt. If in the course of its wandering, the interrupt signal encounters a fork which has the appropriate arming bit on, it will interrupt that fork and die. If it doesn't encounter any such fork, it will just die without doing anything. The fork causing the interrupt will proceed undisturbed. In the interrupted fork an SBRM indirect through location 200 + interrupt number is simulated by the system. This is essentially equivalent to what happens in a hardware interrupt.

Interrupts 1 through 4 are reserved by the system. Interrupt 1 is generated by the rubout button, 2 by memory violation, 3 by termination of any subsidiary fork and 4 by any unusual I/O condition. The idea behind the interrupts is that they allow programs to execute without having to test for any unusual conditions. If any unusual conditions occur the program will be sent automatically to the appropriate routines.

The idea behind fork structure is that you can make yourself independent processes. You can use them in any way to expand the amount of memory that is available to you at one time; to buffer I/O; to perform parallel computations; to receive interrupts from unusual conditions or whatever. There are many possibilities. You can do whatever you want with them. Because of the flexibility of this facility, many other features in the system are not quite as general as we would otherwise wish them to be. One can usually get the effect desired by using one of the general system facilities in a fork, which is independent of the rest of the computation.

For example, normally when you do an I/O operation in the system, you are dismissed until the operation is complete. You many not want to be dismissed, you may want to continue computing, if you do not need the result of the operation for some time. You can get this effect by doing the operation in a fork. The fork will get dismissed, but the main program can keep running. Then when the fork is reactivated it can set a flag or interrupt the main program or whatever it wants to signal the fact that the operation has been completed. It is because this facility is available that we do not think it necessary to have explicit mechanisms by which a program can declare that it wants I/O buffered. By using the fork structure you may set up your own buffering scheme and you can make it just as simple or just as complicated as you choose.

Memory

Then there is memory. I think I will pass over the hardware relabeling. Each fork has associated with it mapping or pseudo-relabeling registers which define the relationship between the 8 virtual pages of the process and the actual pages of memory which the user may possess. A page of memory is defined by an entry in a table called PMT, which gives essentially the location of the memory, either in core or on the drum. The relabeling registers consist entirely of pointers to PMT, or possibly to a table called SMT, which consists of memory which is shared between the users. The user may acquire memory by addressing a block of virtual memory which corresponds to a zero in his pseudo-relabeling. When it does this, new memory will be assigned to it according to one of three algorithms; which algorithm is applied will be determined by the creating fork: a) the subsidiary fork is not allowed to acquire new memory. It must make do with the memory it was given when it was started; b) the subsidiary fork will acquire a brand

new memory block cleared to zero whenever it addresses a block which was formerly unassigned; c) the subsidiary fork will acquire memory which is propagated from corresponding blocks in the creating forks. In addition to this automatic mechanism for changing (adding to) the relabeling, a fork can also change the relabeling explicitly, reading it with `BRS 43`, and setting it with `BRS 44`.

Basically, between the memory allocation algorithm and `BRS 43` and `44` the user is given the ability to select at any given time any combination of 8 or fewer blocks of memory out of the larger number of blocks of memory assigned to him. This is obviously an extremely important facility, for any large scale subsystem consists of many pieces which do not all reside in core at the same time. It should be pointed out that this mechanism is not a panacea for all ills, because the time required to change the pseudo-relabeling is not negligible.

I/O

Then there is the I/O system, which essentially comes in two major sections, one dealing specifically with teletypes and one dealing with I/O in general. The most important I/O interface is the sequential file interface. Such a file can properly be described as a string of bits. The user may read from this string of bits or add to it in units of either 8 bits (1 character) 24 bits (1 word) or some multiple of 24 hits (a block of words). The operations involved are `CIO`, character input-output; `WIO`, word input-output; and `BIO`, block input-output. The first two of these go through the `A` register, the third goes directly to or from the user's memory. The system is responsible for taking care of all packing and unpacking which may be required to make the input-output operation compatible with the nature of the device on which the I/O is taking place. Naturally, for some devices it is more likely you would want to read character by character: for instance, the teletype. Other devices you are more likely to read by blocks. However, you can use any operation with any device. Files are identified by things called file numbers.

A considerable number of teletype files are built in, particular files. 0 and 1 always refer to the user's controlling teletype for input and output respectively. File 2 is an output file which simply forgets about the data. All other files must be explicitly opened by the user. This can be done by execution of `BRS 1`, which is given information specifying the file. That is, a device number telling what device is involved, and possibly an address within the device, for instance the address of an index block on the drum. `BRS 1` is normally restricted to executive type programs, and users are required to go through the executive naming machinery, which we have not discussed. Once the file is open, the file number is used to identify it during input and output operations, and also when it is closed by `BRS 2`.

Unusual conditions which arise during I/O are signaled in several ways: they cause `BIO` not to skip and they may cause interrupt 4 to be generated if it is armed. In some instances, such as the teletype, it is impossible for any unusual condition to occur. The system of course makes whatever efforts it considers reasonable to recover from an error before notifying the user that an error has occurred. Sequential files are of the following kinds: There is the teletype; there are various minor devices: tape, cards and mag tape; there is the drum; and there are subroutine files. A subroutine file is a device which allows the user to turn an I/O instruction into a subroutine call, The user specifies the nature of the subroutine file, whether it is input or output, word or character, and the system thereafter calls on the subroutine essentially as through it were a device. The significance of this mechanism is that it allows you to take a routine which was written expecting to use a file of a certain kind, and give it a subroutine file which convinces it that it really looking at that special kind of file, while actually it is looking at a quite different file.

Sequential files on certain devices have control operations associated with them which allow you for instance to write end of record, rewind, and back space or forward space records and files. The paper tape punch has one: write end of record. Sequential drum files have a considerable number, since they

have a logical record structure. Within these logical records I/O is sequential. However, one may space past logical records and insert or delete logical records more or less freely.

The random file interface for drum files is quite a bit different. It essentially regards a file as a large block of addressable memory. When you do I/O on a random file, you must specify not only the file number but also the address of the word within the random file which you are reading or writing. You can read or write a single word or a block. In the case of a block, you must specify an initial address on the random drum file, initial address in core, and the number of words to be transmitted. There are four operations here, drum word input, drum word output, drum block input and drum block output. It is worth noticing that data blocks are assigned only for the memory which is actually referenced. If a 256 word block which is not referenced occurs among those which are, it will not have any space on the drum. This is exactly analogous to the memory allocation mechanism for core. All blocks have space in the index block, of course.

It is possible to declare a random file to be the secondary memory file, and the operations load `A` from secondary memory and store `A` into secondary memory are available. These operations work in the following way. The effective address of the `POP` is computed in the normal way. That word is then interpreted as follows: The bottom 22 bits are taken as an address and the index bit is checked. If it is on, the contents of the index register is added to this address. The resulting address is then taken as an address in the drum file. Naturally, if you insist on using an address which has a one bit in the 22nd bit, you will generate a lot of index blocks, and this should be discouraged.

Finally, we come to teletypes. There are two operations called `TCI` and `TCO` to read characters from the teletype and print them on the teletype. They are essentially equivalent to `CIO 0` and `CIO 1`, although the conventions as to what goes into memory and what goes into the `A` register are a little bit different. The latter operations are normally preferable, since you can make them more general by changing the file number, which cannot be done with `TCI` and `TCO`. In addition there are the operations for typing out characters on specified teletypes and reading characters from specified teletypes, and an elaborate linking mechanism for linking teletypes either for output or for input. There are protection bits to prevent you from getting output on your teletype and to prevent other people from reading your teletype, if you wish to prevent these things. There is an operation called simulate teletype input which allows a program to generate a character as though it had been typed in an other program.

It is important to understand a little bit about how teletype I/O is actually handled through input and output buffers. When you do output, characters are piled up in the output buffer and are fed out to the teletype at the maximum rate in which it can accept them. When you do input, characters arrive from the teletype and go into the input buffer, even if the program is not there. Echos are generated, the choice of each echo depending on what echo table the program has specified. The alternatives are: echo everything as it is, or don’t echo anything at all.

Whether a program which is waiting for teletype input will be reactivated when a particular character comes in depends on whether or not the character is a break character. The break characters are also determined by the choice of echo table, and there are three possibilities: everything; everything except letters, digits and space; and nothing except control characters.