12.2 Optimization of Program Loops

The following table gives the approximate times in microseconds achieved on Atlas for various instructions. The figures are averages for obeying long sequences of each instruction, with the instructions and operands in different stacks of the core store.

Type of Instruction	Number of Address Modifications	Time
am′ = am + s	0	1.6
	1	1.9
	2	2.4
am′ = am × s	0, 1 or 2	5.9
am′ = am/s	0, 1 or 2	22.5
ba′ = ba + s	0	1.7
	1	2.0

It is not possible to time a single instruction because, in general, this is dependent on

(a) the exact location of the instruction and operand in the store;
(b) the instructions preceding and following; for whenever possible one instruction is overlapped in time with some part of three other instructions,
(c) whether the operand address has to be modified,
(d) for floating-point instructions, the numbers themselves.

This is illustrated when evaluating a polynomial, using a central loop involving a singly modified accumulator addition, an accumulator multiplication, and a test, count and jump instruction. The average time for this loop is 8.3 μsec, of which the accumulator operations would take 7.8 μsec if their individual times are simply added. This leaves only 0.5 μsec for the test, count and jump, although the average figure for a series of jump instructions on their own might be ten times as large.

We shall consider those factors which control the time taken to obey instructions, to show what advantage can be taken of them in optimizing a program loop which has to be executed many times.

12.2.1 Store Access

The main core store consists of pairs of 4096-word stacks. Each stack can be regarded as a physically independent store, and sequential address positions occur in the two stacks of a pair alternately, the even addresses in one stack, the odd in the other. The cycle time of the core store is 2 μs., that is, the time after reading or writing a number before another number can be read from or written to the same stack is 2 μs.

To reduce the effective access time, instructions are always read in pairs and held in two buffer registers called Present Instruction Even (PIE) and Present Instruction Odd (PIO) whilst waiting to be obeyed. Instructions are executed from PIE, the odd instruction being copied from PIO into PIE as soon as the even instruction has been initiated.

Because of the 2μs cycle time for each stack the programmer should separate instructions which refer to operands in the same stack.

For example, the instructions

```
121   1   0   0
324   0   0   A4
362   0   0   A4
```

would be executed more quickly if written as

```
324   0   0   A4
121   1   0   0
362   0   0   A4
```

The maximum overlap is obtained when alternate operands come from alternate stacks.

The Supervisor attempts to organise the store so that instructions and operands are placed in different pairs of stacks. On the Manchester University Atlas, wherever possible, instructions are kept in pages 0-15 and operands in pages 16-31. The programmer can assist the Supervisor to do this by using the extracodes described in 12.1. These are of most use for jobs with a large amount of data. It is then useful to request drum transfers in anticipation, and to release from the core store blocks which will not be wanted again for some time.

12.2.2 The overlapping of Instructions

Instructions on Atlas are overlapped as far as possible. For example, in a sequence of singly-modified accumulator instructions, the computer is obeying four instructions for one quarter of the time, two instructions for one fifth of the time, and three instructions for the remainder of the time.

This overlapping is possible because the accumulator arithmetic, the B-register arithmetic, the function decoding, the B-store, and the main core store are independent of each other to a large extent. A number of rules which enable the programmer to gain as much advantage as possible from the overlapping are given below. It should be noted that these rules cannot always be guaranteed to establish the best way of arranging any particular loop, as in some cases this can only be done by actually running the program; nevertheless the application of these rules, as far as possible, will normally lead to a time reasonably close to the optimum being obtained.

(a) Instructions writing to the main store (usually referred to as store-write instructions) should normally be in odd-numbered locations.

(b) In general, B-type instructions can be obeyed whilst accumulator operations (other than store-write instructions) are going on. Only one accumulator operation can be queued up whilst a previous one (e.g. a division) is proceeding. If a third accumulator operation is encountered, nothing further can be done until the first one is finished. This third accumulator operation should therefore be delayed until all B-instructions and B-tests which can be obeyed before the first accumulator instruction is completed, have been initiated.

(c) Following a store-write instruction, no further instructions or operands can be extracted until the writing operation is completed. Many typical program loops, however, include such an instruction. It is usually possible to have this instruction at the beginning of the loop, and this enables the B-type instruction and return jump to be obeyed and overlap any accumulator arithmetic still going on. As mentioned in (a) above, the store-write instruction should preferably be in an odd-numbered address. From these two rules, two possible ways of arranging a loop, depending on whether it has an odd or an even number of instructions, emerge.

Examples:

1. Odd number of instructions

```
        Even     ┌
        Odd   ↗  └ Store-write
        Even  │  ┌ Accumulator instruction
        Odd   │  └ Accumulator instruction
        Even  │  ┌ Step B-line
        Odd   └  └ Return jump
```

2. Even number of instructions

```
        Even  ↗  ┌ Step B-line
        Odd   │  └ Store-write
        Even  │  ┌ Accumulator instruction
        Odd   └  └ Return jump
```

(d) Instructions are extracted from the store in pairs, and, subject to the above rules, a loop with an even number of instructions should begin at an even address, so as to minimise the number of store references.

(e) Test instructions cause more delay when successful than when unsuccessful, and it is usually best to arrange the uncommon case (if it can be determined) to be the one which changes ba.

(f) Jump instructions where the jump will frequently not take place, should preferably be placed in an even-numbered address. Note that this does not apply to return jumps in loops, as these fail to jump only when control leaves the loop.

(g) Singly-modified A-type instructions should always be modified by bm, not ba.

(h) A delay occurs if a B-register is operated on in the Ba position and then used as a modifier in the next instruction. This should therefore be avoided if possible e.g. by inserting some other instruction in between. Note, however, that

```
124   1   0   1
300   1   2   0
```

is preferable to

```
124   1   0   1
300   2   1   0
```

(i) Given a pair of accumulator instructions, one modified and one not, the unmodified one should occur in the even-address, and the modified one in the odd-address, if possible.

(j) Given an accumulator operation and a B-register operation as an even/odd pair, they should be in this order if possible.

Where the above rules conflict, the order in which they are given should be taken as the order of importance.

12.3 Branching

Branching is a facility which enables different parts of the same program to operate in parallel, using the time-sharing process. Such parallel operation is of value if some parts of the program are liable to be held up waiting for peripheral transfers whilst other parts are still able to proceed. It is important to note that simple operation of peripheral devices in parallel with computing is available without recourse to branching; normally, the program itself is only held up if it attempts to refer to the locations involved in a transfer before the transfer has been completed. Branching is an additional facility which is intended to permit parallel operation of two or more different processes which are liable to be held up by peripheral transfers, where each process involves some computation or organization and does not consist merely of peripheral transfers.

12.3.1 Existing Parallel Operations

When a block transfer to or from a drum or magnetic tape has been initiated, by means of a drum or tape block transfer extracode, the program is allowed to proceed as long as it does not refer to the main store block involved in the transfer. If it does refer to that block, it is held up until the transfer has been completed.

Variable length tape transfers operate by using part of the main store as a buffer. It is usually possible to keep sufficient information in the buffer to permit the actual transfer, between the buffer and the specified store address, to take place as soon as the transfer instruction is encountered. Otherwise the program will be held up until the transfer is complete.

Other peripheral devices, apart from the drums and magnetic tapes, are not normally controlled directly by the program. Instead, the input documents are read and stored on a system magnetic tape before the program is initiated, and output documents are stored on a system tape and printed after the program has been completed.

12.3.2 The Branch Instructions

1103 Permit Ba Branches ($2 \leq Ba \leq 32$)

Before any branching can take place, the program must obey an 1103 instruction, which enables the Supervisor to prepare for branching.

This instruction normally takes the form

```
        1103     Ba     0     pD1
```

After obeying it, the program is permitted to have up to Ba live branches, including the main program, in progress at any one time; the main program is defined as branch 0. When the Supervisor switches from one branch to another it will preserve certain standard information and also the contents of index registers Bp, B(p + 1), B(p + 2),..............., B99. Note that if the N-address is zero all index registers are preserved, and if p = 91 only the extracode index-registers are preserved (these are usually essential).

1104 Start Branch Ba at n ($0 \leq Ba \leq 63$)
The current branch of the program continues at the next instruction, but a new branch, with number and priority Ba, is started at address n. The highest priority is given to the highest-numbered branch: if other branches with the same number Ba have been defined previously, they will take higher priority than the new branch. The main program is initially defined as branch number 0.

1105 Kill Branch Ba or Current Branch
Kill all branches with the number Ba. If Ba = 64, kill the current branch. This prevents any further instructions being obeyed in the specified branches, but peripheral transfers already requested will be completed.

1106 Wait until Branch Ba is Dead.
Halt the current branch of the program if any branch numbered Ba is still live. Proceed to the next instruction when all branches numbered Ba are dead.

1107 Jump if Branch Ba Live.
Transfer control to address n if any branch numbered Ba is still live. Otherwise proceed to the next instruction.

12.3.3 The Use of Branching

A branch is usually started at some point in a program where it is required to carry out two different processes, at least one of which is liable to be held up by peripheral transfers. Usually, the more severely peripheral limited process is put in the new branch, and this is given higher priority. When the program is obeyed, the higher priority branch is allowed to proceed until it is held up waiting for a peripheral transfer; control is then transferred to the other branch, which proceeds either until it is held up, or until the higher priority branch is ready to resume. Similarly, if there are several branches, the Supervisor ensures that control always passes to the highest-priority branch able to proceed. Each time control is switched from one branch to another, the Supervisor stores and restores the contents of the following registers and indicators:

The Accumulator

B119, B121, B124, B126 and B127.

The Index Registers specified in the 1103 instruction.

The Selected Magnetic Tape Number.

B-Test, B-Carry, Accumulator Overflow (V-store Line 6).

Extracode Working-Space.

Thus, each branch can use these registers as though it were one single program uninterrupted by other branches. It is, however, necessary to ensure that two branches which may operate simultaneously do not use the same main store locations, or index registers which are not preserved. It should be noted that the selected Input and Output are not preserved, and therefore input and output can each take place in only one branch at a time.

Once a branch has been started it can be regarded as a 'live' branch, and it remains live, even when it is held up, until its task has been completed. When a branch has completed its task, it must die, and this it does by obeying an 1105 instruction, usually with Ba = 64.

When one branch of a program is ready to make use of the work done by another, it must first ensure that the work has been completed. This may be done by obeying an 1106 instruction, which causes the current branch of the program to be held up until the specified branch is dead, having completed its task.

A simple example of the need for branching arises when it is required to scan a magnetic tape in order to process a selected sample of the information on it. The processing routine and the tape-scanning routine can then be written as two separate branches, with the tape-scanning routine as the higher-priority branch.

Example:

It is required to scan sections 1 to 3000 of tape 4 and to apply a lengthy processing routine R3 to the information in about 25% of these sections. The sections to be processed are to be identified by having a number greater than 0.32 in the first word of the section. The program to do this could be written as follows:

```
                                   Branch 0
    1103    2     0    89D1        Prepare to use 2 branches,
                                   preserving b89 - b99.
    1001    4     0    1           Search for section 1, tape 4
    121     89    0    2999        Set count for 3000 sections
    121     16    0    0           Clear marker in B16
    1104    1     0    A6          Start branch 1 at A6
5)  1106    1     0    0           Wait until branch 1 dead
    215     127   16   A12         Exit if last section processed
    121     10    0    4: )
                                   Rename block 3 as block 4
    1164    10    0    3: )
    1104    1     0    A7          Start branch 1 at A7
    121     90    0    A5          Set Link for return
    121     127   0    A1/3        Enter R3 to process block 4
                                   Branch 1
6)  1002    4     0    3:          Next section to block 3
    324     0     0    3:          First number in section
    321     0     0    1A8         Subtract 0.32
    236     127   0    A8          Exit if number ≥ 0.32
7)  203     127   89   A6          Count tape sections
    121     16    0    7           Mark b16 non-zero
```

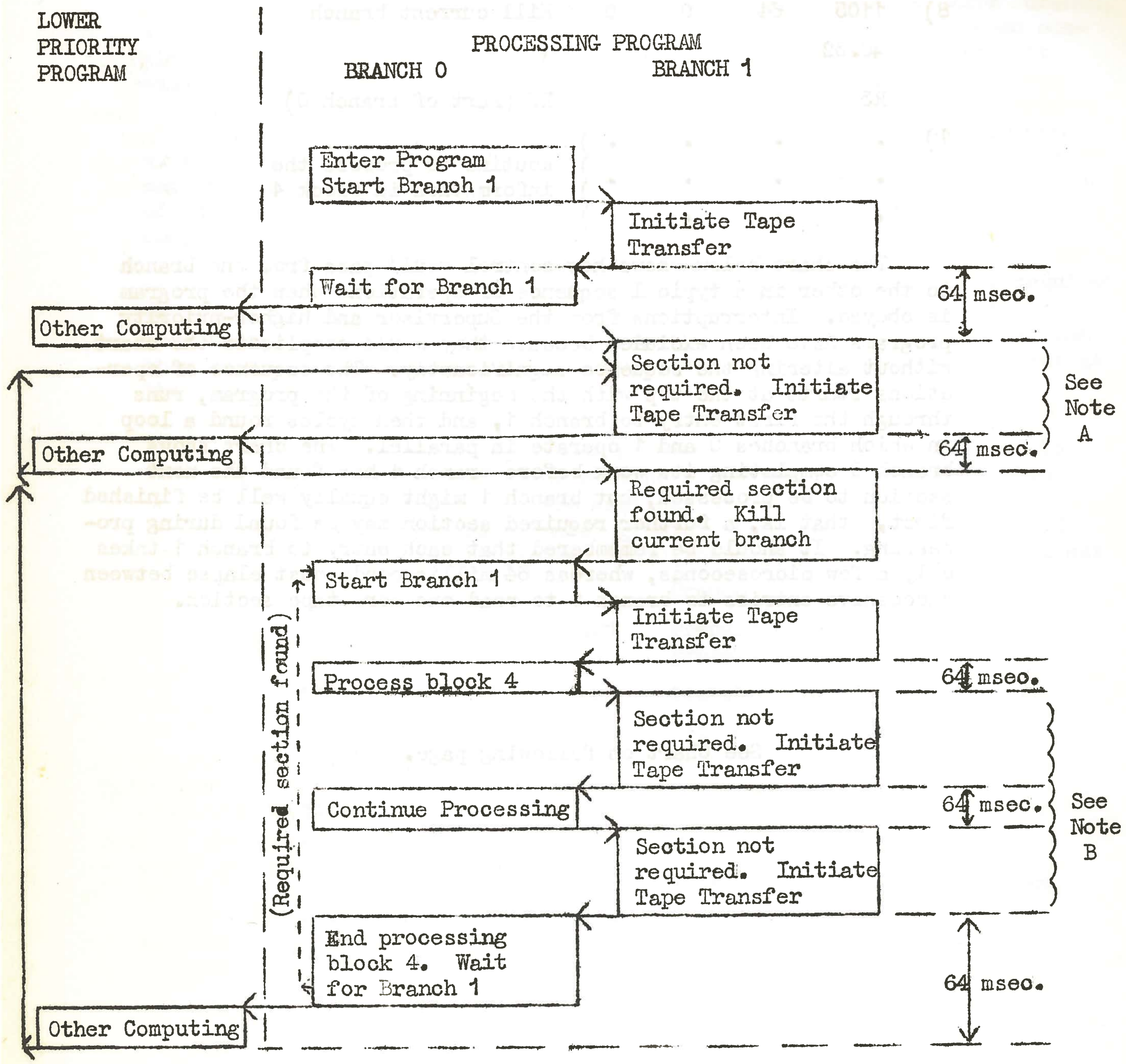
(Program continues on following page)

```
8)  1105    64    0    0           Kill current branch
    +0.32

    R3                             R3 (Part of Branch 0)
1)  .       .     .    . )
    .       .     .    . )         Routine to process the
                         )         information in block 4
    .       .     .    . )
```

The chart below shows how control would pass from one branch to the other in a typical sequence of operations when the program is obeyed. Interruptions from the Supervisor and higher-priority programs have been excluded because they would complicate the chart without altering the sequence significantly. The sequence of operations starts at the top with the beginning of the program, runs through the first entry to branch 1, and then cycles round a loop in which branches 0 and 1 operate in parallel. The chart shows branch 0 completing its work before branch 1 has found the next section to be processed, but branch 1 might equally well be finished first, that is, a further required section may be found during processing. It should be remembered that each entry to branch 1 takes only a few microseconds, whereas 64 milliseconds must elapse between successive entries to branch 1 to read one more tape section.

See Chart on following page.



Notes:

A. This loop will be repeated until a required section is found.

B. If a required section is found, then Branch 1 will be killed. When the current block has been processed, Branch 0 will start Branch 1 again, and then process the required section.

12.3.4 Store Requirements

When an 1103 instruction is obeyed, the Supervisor assigns sufficient storage space for the specified number of branches. This storage space is taken out of the main store allocated to the program, either by the job description or by a subsequent use of the extracode 1171, and will be counted in the estimates made by extracodes 1172 and 1173; it is therefore necessary for the programmer to know how much store is required by the Supervisor for branching purposes. Many cases should be covered by the following table, showing the maximum number of branches that can be accommodated in 1, 2 or 3 blocks, depending on the number of index registers preserved.

Index Registers Preserved	Storage Space Allocated		
	1 Block	2 Blocks	3 Blocks
	Maximum Number of Branches Permitted		
B0 to 99	3	10	18
B30 to 99	4	14	24
B50 to 99	5	17	29
B70 to 99	6	22	32
B80 to 99	8	27	32
B90 to 99	10	32	-

If it is necessary to estimate the store required in some case not covered by the above table, it is probably easiest to do this by considering the way in which the Supervisor allocates this store. It takes 300 words at the beginning of the first block to store branching routines, and follows these by 5 words for each branch requested in the 1103 instruction. Each branch is then allocated a further $(11 + \frac{1}{2}m)$ words, where m is the number of index registers in the range 0 to 99 that are to be preserved. The $(11 + \frac{1}{2}m)$ words for one branch must all be in the same block, and if less than $(11 + \frac{1}{2}m)$ words are left at the end of a block the $(11 + \frac{1}{2}m)$ words for the next branch will start at the beginning of a new block.

12.4 Instruction Counters

As each basic instruction is obeyed, an instruction counter is stepped on, normally by one, but by two for multiplication orders, and by four for division. Each time the counter reaches 2048, an interrupt occurs, and an instruction interrupt counter is stepped on by one. This latter counter is used by the Supervisor in monitoring the program, but may also be read by the program using extracode 1136.

1136 Read instruction count.
Set am′ to the number of instructions obeyed from the start of the program; this will be a fixed-point integer with exponent 16, and will be a multiple of 2048.

Besides this count, the program may also use a local instruction counter. A trappable fault will be recognized when this count expires, which may provide a convenient way to end an iterative loop, since the counter may be set as well as read by program.

1123 Set local timer.
Set the local instruction counter to n x 2048 instructions. The Supervisor will override any attempt to set the counter to a figure in excess of the amount of allotted time remaining. — local timer′ = 2048n

1122 Read local timer.
Read local instruction counter into Ba in units of 2048 instructions. — ba′ = local timer

12.5 Re-entering the Compiler

Most programs are compiled completely before they are entered, and therefore it is not normally necessary to retain the compiler in store during the program′s execution. The E-type of directive is the only enter directive which deletes the compiler from the store before transferring control to the object program, and so is the most commonly used.

In some circumstances, however, it is necessary to enter the program, and then compile more program later. The first entry may be to actually execute part of the object program, or it may be only to set certain parameters. The compiler must be retained in store for these purposes, and so either an ER or EX type of enter directive must be used. The compiler uses store locations J3 ($3/4 \times 2^{20}$) and above which should not normally be altered by the program, although no check is made except when actually compiling.

The EX-directive is intended for obeying ′interludes′ during compiling; an interlude would normally consist of a few instructions only, or of none at all. For example, if it were required to have any ABL fault printing on some output stream other than Output 0, then a one instruction interlude to ′select output′ would suffice. If it is only required to set parameters, then the address specified in the EX-directive should cause immediate re-entry to the compiler; such a directive in fact occurs near the beginning of L100, the general input routine, to determine the various optional parameter settings. The EX-directive does not call down any library routines; if these are required in the interlude, they must be called by one of the L-directives before obeying the EX entry. No distinction is made by the Supervisor between compiling proper and obeying an interlude, i.e. the ′Compile/Execute′ switch is not changed.

The ER-directive is designed to allow part of a program to be compiled and executed before reading more program, and provides most of the facilities of an E-directive, including the compilation of any library routines mentioned but not called earlier in the program. The routine current when the ER-directive is obeyed will be terminated before more program is read. (The EX-directive does not do this.) The Supervisor recognises that an object program is being executed, and as with the E-type of directive, the ′Compile/Execute′ switch is set to ′Execute′.

Two types of list within the compiler are used in connection with parameters in a program. The parameter lists contain all those parameters which are determinate, and if the program refers to a set parameter, these lists are used to replace the parameter by its value. If a program refers to a routine or global parameter before it has been set, then this is noted on a forward reference list, from which it is deleted when the parameter is determinate, and hence, so long as this list is not empty, there are some parameters still to be set. When the compiler is retained in store, these lists also remain, in the same state as when the enter directive was obeyed. If more program is to be read which uses parameters to refer back to the program compiled previously, then it is essential that the lists remain unaltered. If, however, the subsequent sections of program are to be compiled independently of the earlier part, or if the same parameters are to be used again with different values, then the lists must be cleared on re-entry to the compiler. Different re-entry points provide for both requirements, and are listed below. In every case, re-entry to the compiler does not alter the 'Compile/Execute' switch. After compiling program, B1-B88 will be cleared, and B89 will contain the final transfer address. The other B-lines may be destroyed.

P120 When the compiler is re-entered at address P120, all parameters, forward references after EX, and * (the transfer address) are left unchanged, and more program is read from the current input stream. If there is no *-directive before the first items are read, these will be placed in store sequentially in the usual way, after the last item of program before the previous enter directive. After an ER-directive, library routines may have been compiled into locations beyond the end of the written program.

P120B Re-entry at this address causes the compiler to behave as it does when first called by the Supervisor, but the contents of the store below J3 are left undisturbed. Hence * = 1:0, and the forward reference list and parameter lists are cleared.

The compiler may also be used as a subroutine by a program, control returning to the main program when no more items are to be read. Again there are two modes of entry, depending on what is required of the compiler lists.

If the transfer address is written to location Y4P121, and the link to 129P121, then re-entry to the compiler at P120 will read more program, retaining the compiler lists. Return to the link address in the main program is effected by

EP129
ERP129
or EXP129

If the transfer address is written to B89, and the link to B90, when the compiler is re-entered at P120BY6, more program will be read as if the re-entry were to P120B, except that an attempt to compile into store at an address less than b89 will be faulted, and that the transfer address will be taken as * = b89 unless b89 = 0, when * = 1:0. Return to the main program is again by EP129 etc.

P120, P121, and P129 are examples of special preset parameters, which are described in the next section.

12.6 Special Preset Parameters

Although for normal purposes only Preset Parameters 0 to 99 may be used, some above 100 do exist and these are used in special ways for special purposes. In some cases use of them causes special action by the compiler; in other cases they are used to convey information between the compiler and the program.

P100 to P109 are in many ways like ordinary Preset Parameters; they can be reset by the programmer and no special action is taken by the compiler on encountering them. However, they are initially set by the compiler at the start of compilation and they are referred to by the compiler during the course of compilation.

P110 to P119, if defined, may be reset by the programmer but will have initial values set for them by the compiler. However, whenever an Equation Directive for resetting them is encountered, special action is required by the compiler. An attempt to set one of these parameters not listed below is faulted.

P120 to P129 are preset by the compiler, but may not be reset by program. An attempt to do so is faulted. They are used to convey information from the compiler to the program.

P100 - Optional Printing

At the start of compiling ABL sets P100 to zero. Non-zero settings of P100 cause ABL to print various kinds of information during compiling.

P100 is treated by ABL as made up of 8 octal digits abcdefgh. Each octal digit controls the printing of one kind of information, as indicated.

If the least significant bit of an octal digit is 1 the information controlled by this digit will be printed - on a new line if the middle bit of the digit is 1 and on the same line if the middle bit is 0. If the least significant bit is 0, then the other two bits are ignored. If the most significant bit of the second digit (b) is 1, then printing on a new line will occur when a library routine is compiled. Otherwise the most significant bits of the digits are ignored.

P100 is preserved and set to zero before compilation of each library routine and restored afterwards, so that there will be no other optional printing, unless the library routine contains a 'P100 =' directive.

The kind of printing controlled by each octal digit is as follows. All printing is preceded by a space, except R. L is printed on a new line.

Octal digit	ABL prints this	when it meets this
a	$*=p$	$* =$ expression
b	Ra $*=q$ La.b N $*=q$	Ra Library routine named N compiled
c	Z $*=q$	Z
d	this digit is unassigned and ignored	
e	p	P 111 = expression (see P111)
f	E p	E expression
g	E R p	ER expression
h	E X p	EX expression

where p is the value of the expression met,

q is the current value of asterisk

a and b are integers.

Some examples of useful settings of P100 are

P100 = -Y1 ABL prints on R, L, *, Z and all types of E, each item on a new line

P100 = J03 ABL prints on R only

P100 = J031 ABL prints on R and Z

P101 - Permitted Number of Errors

At the start of compiling ABL sets P101 to 0.2. This is equivalent to infinity since for each error met ABL reduces P101 by 1, and when it reaches zero stops compiling and ends the run after printing

TOO MANY ERRORS

The program may set P101 = n where n is any expression. Compiling will stop when n + 1 errors have been met. P101 = 0 causes ABL to stop on the first error met, which may be useful for a developed program.

No matter how many labels remain unset when the E directive is met ABL lumps them all together as one error for the purpose of counting errors.

P102 - Entry Despite Faults

At the start of compiling ABL sets P102 = 0.2. If any errors have been found, an EX directive will be obeyed, but an E or ER will not, and ABL will print

ERRORS DO NOT ENTER

and end the run.

P102 = 0 allows all 3 E directives to be obeyed despite errors

P102 = 0.3 forbids all 3 E directives after errors

P102 = 0.1 allows E and ER but forbids EX after errors

P104 - Setting Private Monitor

P104 is the address of a private monitor routine, which is set up each time any Enter Directive is encountered. Thus if any monitors occur after the Enter Directive (including an immediate entry to an address holding an Illegal Function due to a wrong Enter Directive address), these will give rise to an entry to Private Monitor according to the rules of extracode 1112 (see section 11.3). If P104 is negative any current setting is terminated.

P110 - Change of Program Location

At the start of compiling, ABL sets P110 = 0. A non-zero setting of P110 specifies the difference between * as evaluated in expressions (say $*_1$) and * indicating where items are to be stored (say $*_2$)

i.e. $P110 = *_1 - *_2$

Setting P110≠0 permits the compiling of program into one set of store addresses - $*_2$ - for later execution in another set of store addresses - $*_1$ (e.g. after 'Renaming' or after storing on magnetic tape). Thus, for example, a program which is to be executed starting at address J3 but compiled initially into store starting at J1 would have the directives

P110 = J2

$* =$ J3 at its start

P111 - Expression printing

When ABL meets the equation

P111 = expression

it evaluates the expression and sets P111 in the usual way. If the appropriate bit of P100 is set, the value of the expression is immediately printed out.

P112 - Unused

This parameter is used by ABL on Atlas 2. No fault will occur if the program sets P112, the value being assigned in the usual way, but the remainder of the line on which the setting occurs will be ignored. The compiler initially sets P112 to J3. P112 should not normally be used with ABL on Atlas 1.

P115 - Change of Input Stream

The equation P115 = n(n is any expression) causes ABL to start reading program from the programmer's input stream n. The rest of the line on which P115 occurs will be ignored.

P120 - Re-entry to the Compiler

This is described in the previous section.

P121 - Preset Parameter List

P121 is the address of the start of the compiler's Preset Parameter list. Halfword P121+n contains the value of parameter n if it has been set. This list can of course only be referred to by a program entered by an ER or an EX directive. This is a list of alternate halfwords, the other halfwords of which are used for other purposes by the compiler and should not be disturbed.

P122 - State of Preset Parameters

P122 is the address of the start of the compiler's list which indicates whether the Preset Parameters are set or not. Bit 1 of halfword P122+n is a 1 if parameter n is unset, 0 if set. This is a list of alternate halfwords, and neither the other bits of the halfwords, nor the other halfwords should be disturbed if subsequent use of the compiler is intended.

Example: An 'interlude' to set P17 = 0 if P35 = 0 and to leave P17 unaltered otherwise

```
5)  121     1    0    P35
    215   127    1    P120
    113     0    0    P121+17
    121     1    0    J2'
    117     1    0    P122+17
    121   127    0    P120
    EXA5
```

P123 - Characters Count with C directives

P123 indicates the number of characters read by means of the previous C directive, described in section 5.10.

P129 - Return from Compiler

P129 is the return address when the ABL compiler is used as a subroutine (see previous section).

12.7 Private Library Routines

12.7.1 Library Routine Titles

Library routines are given numbers and names; the program refers to them by number, but the name may be useful to indicate their purpose. The standard input and output routines, described in Chapter 8, have been given names as follows:

L1	GENERAL OUTPUT
L100	GENERAL INPUT
L199	LINE RECONSTRUCTION

L199 is used by L100, and hence, if the library routine is being compiled implicitly, (whether by an 'L' directive or by an Enter directive), L199 will automatically be compiled with L100 because the latter refers to it, but, if it is required to compile the 'input library' explicitly into a part of the programmer's store area, then it is necessary to use both of the directives

```
L100
L199
```

12.7.2 Undefined Library Routines

All undefined library routines have the name NONEXISTENT and there is a special device to make the optional printing as described for P100 (section 12.6) compulsory for non-existent routines. For example, output such as

```
L10    NONEXISTENT * = 2:36.3
```

will result from either an attempt to call for L10 explicitly or when an attempt is made to compile it by an L or Enter Directive when it has been referred to implicitly. The latter would also result in monitor printing about unset labels.

Any defining of a Private library routine (see below) will cause suspension of the NONEXISTENT monitoring for that routine.

12.7.3 Preparing a Private Library Routine

Private Library Routines may be incorporated in the normal program input stream and may be referred to in the progr m in the same way as public library routines.

The routine is headed by two lines:

```
RLc
< Name >
```

where c is the number assigned to the library routine and < Name > is the name of the routine. If no name is required, this line must be left blank, and then a blank title will appear in any optional printing. The Name must not consist of the two-character record ZL.

The routine is terminated by the two-character record ZL. This is not line-reconstructed and may not contain any spaces, erases, backspaces, tabs etc.

The library routine may consist of a single routine (routine 0) or of one or more routines headed by routine directives and optionally terminated by Z directives in the usual way. It may contain any of the normal ABL forms except the directives

L, La, La.b, ER expression, E expression, RLc

All will be monitored. No T or C directive within the library routine may be followed by the two character record ZL.

When the RL directive is encountered, the library routine following is simply copied character by character into the compiler's store area. The routine is not, at that time, compiled or placed in the programmer's store area. This is achieved in the normal way, by an E, ER or 'L' - type of directive.

If a private library routine is given the same number as a public library routine, it replaces the public one for the remainder of that program. This is convenient for the development of new versions of existing public routines.

Private Library Routines must precede any calls for them in the body of the program. The best place for them is at the beginning of the program stream.

A private library of routines required by people working in some limited field (e.g. properties of steam) may be formed by putting the routines on to a titled paper tape (as pseudo-data) and terminating them by P115=0. The master programmer may then write, for example,

```
INPUT
15 STEAM LIBRARY
```

in his Job Description, and

P115 = 15

at the head of his program to incorporate these routines effectively as above.

Each private library routine incorporated in the way described in this section counts as one line from the point of view of line counting for error monitoring of the subsequent program.

The directive

P115 = expression

within a library routine, will not cause monitoring, although it should never normally be needed or used. Its effect is in fact to cause switching of the input stream after completing the compilation of the current library routine or routines at the point where these are compiled (i.e. at an L or Enter directive).

12.7.4 Incorporating a new Library Routine into the Public Library

This is done by means of a special job, using a standard program of the system. Essentially, the routine to be incorporated is put at the head of the program stream in the normal way, as described above, and is followed by the standard program. This program uses no labels as the non-empty parameter list would otherwise become part of the compiler.

If the routine being incorporated is a new version of an already existing routine, then this latter is not destroyed or overwritten on the compiler tape. The reference to it in the compiler's library routine list is simply changed and so it becomes 'dead'. A separate special program (or prelude to the above program) may be used from time to time to clear out all 'dead' library routines, but this clears out all live ones as well, so that after this clearing out operation all public library routines must be re-incorporated.

12.7.5 Conventions

The following conventions are recommended for library routine writers and users:-

(i) Communication of parameters and addresses for use at compile time should be by routine parameters of routine 0 of the library routine, since

(a) using routine, global or preset parameters of the master program could easily lead to clashes with other library routines if allowed, and

(b) the master programmer will not be interested in the breakdown of the library routine into sub-routines.

(ii) If preset parameters are used within the library routine, they should be high numbered ones, say P90-99, and should be unset at the end of the library routine. Their use should be mentioned in the specifieation for the library routine.

(iii) Any special preset parameters used (except P100 and P123) should be preserved and restored.

12.7.6 Referring to the master program from within a library routine

A routine parameter of the master program can be referred to within a library routine by treating the master program as if it were library routine 0, e.g. A6/3L0 is A6/3 of the master program.

12.8 Correction of Programs, and System Peculiarities

12.8.1 Program Alterations

To correct a small program, it is usually simplest to re-punch the tape or cards, making alterations as necessary. This is impracticable for larger programs,but the facilities of ABL may be used to help make corrections.

Very often it is possible to make corrections by overwriting certain store locations, using a *-directive. The corrections, however, must be compiled after the faulty items, or the faults will overwrite the alterations. Hence, the corrections are normally placed just before the enter directive. An enter directive may cause library routines to be compiled from the current transfer address, and so to prevent the program being overwritten from the faulty item onwards it is necessary to insert the correction in one of the following ways.

```
a)  1)  <Last item of program proper>
          * = 6A10/4
        <Correction>
        * = 1A1
        EA40

b)      <Last item of program proper>
        L
        * = 6A10/4
        <Correction>
    1)  EA40.
```

In case b), however, if data is read to A1 onwards, this will also overwrite program when the correction is inserted.

It may be convenient to end a program with

```
        R10
    1)  <Last item of program proper>
        P115 = 15
        * = 1A1/10
        EA40/3
```

so that corrections, if any, will be read from input stream 15, which ends with P115 = 0.

When routine parameters are mentioned in a correction, without specifying any routine, i.e. /n is omitted, the routine to which they refer will be that current before the correction, rather than that at the location to be corrected.

Difficulties may be encountered when using '*=' to overwrite an item containing forward references. The result can be predicted from the

following notes on ABL's handling of forward references.

Two lists are concerned, the forward reference list in which are partly evaluated expressions containing forward references, and the parameter list in which are all parameters found in the program with their values if set.

(i) The forward reference list is initially empty.

(ii) When an expression is read it is added to the end of the forward reference list, which is then condensed starting from that expression. In the case of indeterminate parameters which need to be evaluated before compiling continues, i.e. after EX, ?, *=, Pa=, and Pa?= when Pa has not been set earlier (a is an integer), then the expression will be faulted as EXPRESSION INDETERMINATE (see section 11.6).

(iii) Whenever a routine or global parameter is set the forward reference list is condensed from the beginning.

(iv) On reading an E or ER directive all necessary library routines are read, any outstanding 'A?=' or 'G?=' directives are implemented in the order in which they occur in the program (this may lead to further settings of parameters and so further condensations of the forward reference list), and if the forward reference list is not empty its contents are output as Indeterminate Expression errors.

(v) On reading EX, A?= or G?= are implemented as in (iv).

Condensation of the forward reference list.

1. For each expression in turn, each set routine or global parameter in it is replaced by its value and the expression is partly evaluated. If no parameters remain unset the expression is completely evaluated; in that case, if the expression is not the right-hand side of an 'A?=' or 'G?=' directive, it is planted in the program area or the parameter list and deleted from the forward reference list.

2. If, on reaching the end of the forward reference list, any parameters have been set during 1, the process is repeated from the beginning.

   For example, suppose the program begins

```
            *=0, HA3
            A3=A4-1
            *=0, HA3-1
            A4=4
```

On reaching the 4th line the forward reference list will contain the expressions A3, A4-1, A3-1. When the 4th line is read the list becomes

A3, A4-1, A3-1, 4;

the '4' is evaluated immediately and A4 becomes set; the list is then completely re-condensed.

A3 is not yet set and so remains.
A4-1 is evaluated and A3 gets set.

A3-1 is evaluated and planted.

The list now consists of only A3 and since a parameter was set in the last condensation the list is re-condensed.

A3 gets evaluated and planted.

It is seen that in this case the half-word ends with the value of A3, i.e. 3.

If a routine or global parameter is optionally set more than once, but is not set otherwise, then the first optional setting will be implemented. The subsequent optional settings will not be checked for faults.

A different mode of correction uses the library facilities of the ABL compiler. A copy of the compiler would be dumped initially onto a private magnetic tape, and subsequently used to compile routines as a library on to the tape. As these routines are corrected, the new versions are introduced, replacing the faulty library routines as described in section 12.7.4.

12.8.2 Further Peculiarities

Floating-point numbers may be represented in the form a(b:c):d or Ka(b:c):d as described in section 5.11. Although the number may be within the range of the accumulator, compiling it may cause exponent overflow unless the following three limitations are observed.

(i) $|b+c| < 100$

(ii) $|b| < 1000$

(iii) a consists of not more than 20 digits

When a parameter optionally set by a ?= directive is actually set elsewhere, the right hand side of the equation for the optional settings may not be checked.

After the final ABL fault printing, no new line is output.

ABL will read program incorrectly after $8191 = 2^{13} - 1$ printed lines without the implicit setting of a routine parameter by labelling.

In fault pointing, the line count will be taken modulo 2^{12}.

No check is made that function codes exist. One and two digit functions are right justified into the function bits.

When obeying program, the compiled value of * will be different from the current value of control, as b127 is stepped on by 1 before starting to obey an instruction. Thus

```
121     69     127     -*
```

would set b69' = 1. For the same reason

```
121     127    127     -1
```

causes a loop stop.

When more than twelve digits in a number are printed by the general output routine L1, digits after the twelfth may be wrong. L1 also may give exponent overflow attempting to print the following numbers:-

Non-zero Mantissa	Exponent
-1	+127
$\pm x (x \neq -1)$	-127
$\pm x$	-128

12.9 Compiler and Supervisor Extracodes

The extracodes given below are used mainly by system programmers. They complete the list of Atlas 1 extracodes.

1126 v7′ = n
and hoot if the least significant integer bit of n is 1. (bit 20)

1127 ba′ = v7 & n
Mask the digits of the engineer's handswitches with n, and read them to bits 16-23 of Ba.

Line 7 of the central computer V-store consists of 8-bits. Only bits 16-23 may be read, being set from the engineer's handswitches. Other bits are read as zero. Bit 20 controls the hooter and may be set by program; writing to the other bits is ignored.

1140 Read 'parameter' Ba of program to store starting at location S.

Ba	Parameter
0	Job title (10 words)
1	Computing time estimate, in seconds, in digits 0-23 (One half-word)
2	Execution time estimate, (One half-word)
3	Number of store blocks required, in digits 1-11 (One half-word)
4	'Parameter' in Job Description (One-half-word)
5	Logical tape numbers defined (8 half-words). The j^{th} digit ($0 \leq j \leq 15$) of the i^{th} half-word is a 1 if tape number 16 i + j is defined.
6	Inputs defined (One half-word). The i^{th} digit ($0 \leq i \leq 15$) is 1 if input stream i is defined.
7	Outputs defined (One half-word) As 6.

1141 Define Compiler

Ba = Tape Number to which the compiler is to be written. If Ba = 127, and if the compiler name specified (see below) appears in the Supervisor Directory, the compiler will be written to the current Supervisor tape. In this case there will be two loop stops with J70707070 in B120 since this extracode will use 1143, 0, 0, 0.1 and 1143, 0, 0, 0.2 (see below).

The five half-words S to S+2 contain the following parameters:

a) First four characters of name.

b) Second four characters of name.

c) Main store starting Address (of where the compiler is now).

d) Main store finishing Address (of where the c ompiler is now).

e) Actual main store starting Address (of where the compiler is to be placed when in use).

Notes:

i) The compiler name should be right justified within each half-word, but the first four characters should be put in the first half-word,

e.g. ABL = J00414254, 0

HARTRAN = J50416264, J00624156

ii) If the first four characters are zero, the second four will be used as the starting block address (digits 0 to 21) of the compiler on tape. This facility cannot be used when Ba = 127.

iii) The starting address and the actual starting address should have bits 12 to 23 zero, and the starting address must be greater than zero, since the block before this is used to set up the compiler title block.

iv) The following fault indications may be printed:

COMPILER NAME NOT LISTED

COMPILER NOW U/S TAPE FAIL

COMPILER TOO BIG

WRITING TO SPECIFIED BLOCKS ON SYSTEM
TAPE IS PROHIBITED

The first and third can only occur if Ba = 127. The fourth implies the facility described in Note (ii) has been used with Ba = 127.

1142 End compiling

(i) If Ba ≠ 0, set Compile/Execute,switch to Execute, and reduce store allocation to that specified in Job Description. Lose all store bl.cks with block labels greater than or equal to digits 1-11 of ba, unless ba less than 0, in which case lose no blocks. Transfer control to address n, unless n less than 0, in which case End Program.

(ii) If Ba = 0, do none of the above; the only effect of this extracode is then to inform the Supervisor that the copy of the compiler being used has been, or may have been, spoilt, so that a new copy must be brought from magnetic tape for any subsequent job. The Compile/Execute switch is not changed. This extracode is useful where a compiler may have been spoilt by an interlude.

1143 Reserve Supervisor Tape

Ba should be zero

n should be 0.1, 0.2 or 0, with the following meanings:

n = 0.1 The Supervisor will come to a loop-stop with J70707070 in B120 waiting for the Write Permit switch to be switched on on the Supervisor tape. When this has been done, the program will be allowed to write to, or read from, the Supervisor Tape (logical number 127), and normal use of the tape (e.g. reading compilers) will be halted.

n = 0.2 The Supervisor will come to a loop-stop with J70707070 in B120 waiting for the Write Permit switch to be switched off on the Supervisor tape, after which normal use of the Supervisor Tape will be resumed.

n = 0 The Supervisor Tape will be reserved as with N = 0.1. but for reading purposes only. There will be no loop stop, but 1143, 0, 0, 0.2 must be obeyed after reading, in order to release the tape for normal purposes. This facility is used if it is necessary to print out part of the Supervisor tape.

An operator request will be necessary before using this extracode.

1147 Call Compiler

(i) If n is even (digit 23 = 0), then n will be interpreted as a compiler number, and the compiler in question will be called from the Supervisor Tape and entered at the address specified by ba. The numbering of the standard compilers is given in Part 1 of the Operator's Manual (CS 411). This facility is used by the Supervisor, and by programs using compilers as subroutines.

(ii) If n is odd (digit 23 = 1), the compiler will be called from block b of tape a, where a = digits 15 to 21 of n
b = digits 2 to 14 of n

It will be entered at ba.

In both cases, if ba = 0, the standard entry point will be used.

With 1150 and 1151, ba, P and K are as defined in section 12.1.

1150 Assign ba blocks, labels P to (P + ba -1) to overflow K.

This extracode enables a program or compiler to temporarily hand blocks to the Supervisor, which may write them to the system dump tape. Subsequent use of these labels in the program causes new blocks to be assigned. The block labels are retained in the 'overflow' region and additions to this region must bear distinct labels. If ba = 0, one block is transferred.

1151 Set up ba blocks, labels P onwards, from overflow K.

This extracode recalls blocks previously written to the overflow region by use of 1150. Any existing blocks having these labels are overwritten. If ba = 0, one block is recalled. If these blocks do not exist in the overflow region, the program is monitored.

1156 Enter extracode control at n if the 'In Supervisor' switch is set.

This extracode is used by various Supervisor Extracode Routines which are obeyed on main control. If the 'In Supervisor' switch is not set, the program will be monitored.

1157 Enter extracode control at n if the 'Process' switch is set.

This may only be used by Supervisor routines such as the monitor called in during the running of a main program.