11.0 Hints that may be useful to Programmers

This section is a write-up of observations made by users. In no way is this section an official document of how "things" ought to be done, and users follow these hints at their own risk.

11.1 Documentation of Programs

All programs should be comprehensively documented i.e. the comment facility should be used. Every effort should be made to keep the documentation of a program up to date.

11.2 Program Storage media

Programs must initially be prepared on paper tape or cards and usually the "character version" which may contain comments is punched. It may be advisable in some cases (e.g. a long program) to transcribe this "character version" to magnetic tape using the utility routine, COPYDOC or EDITOR during this run a line printer copy can also be obtained. Note that both Basic and Symbolic will read "character versions" from magnetic tape. Advantages of storing the "character version" on magnetic tape are that magnetic tape affords the best safeguard against errors since it is less likely to be maltreated, thus reducing peripheral failures on reading, less likely to be mislaid,, faster input etc. Thus the paper tape or cards are read once only and magnetic tape read several times either to run the program (e.g. a Basic program or a Symbolic load-and-go program) or to compile (a Symbolic program to Basic program) or to re-edit the program (a Basic program or a Symbolic program) using EDITOR, or to binary-and-map (i.e. map) the program (a Basic program) using MAPPER etc.

11.3 Information about Basic Input Routine which may be helpful in speeding up input of an object program.

Basic uses some of the core allocated to it and drum for working space. Basic needs at least 1008 words of core and reserves more drum as it needs it. If Basic is given more core than the minimum then less drum may be used which will result in fewer drum transfers. There will be, however, for an object program an optimum amount of core which will result in a significant increase in input speed; this is about 2000 words for "average" lengthed programs. Reducing drum transfers usually results in faster input.

Basic uses core in the following way:-

1) Basic's program area for its chapters is 800 words - this cannot be increased.

2) The L identifiers (the L's) are stored in blocks of 64 words (one word per L). When access to an L is required, the whole block has to be in the core. By giving Basic more core more blocks of L's can be kept in the core rather than on the drum.

3) There is always in the core an index of the blocks of L's being used; this is one word per block used.

4) There is one word par forward reference outstanding at any one point in time. By giving Basic more core more of these can be held in the core rather then on the drum.

5) The directives which are implemented later (e.g. NEW at ENTER time) have to be stored. By giving Basic more core these can be kept in the core instead of on the drum. Fewer than 64 MONITOR directives cause 64 words to be used.

6) The main purpose of Basic is to store object program on the drum. The object Program's s storable words are assembled in the core and written to the drum in convenient sized chunks. The more core Basic is given the larger the chunks (up to a certain point), and so there will be fewer drum transfers involved.

If a program wants to be entered with fewer than 2000 words of core then instead of the job-tape

```
JOB RUN 100
READ MY/PROG/-
```

have the job-tape

```
JOB RUN 2000
RES *CORE 100
READ MY/PROG/-
```

Since the RES CORE directive is reducing reservations, it is remembered until enter time and there will be no change of datum point.

Note that if a Reserve core directive which increases core reservations is read then Basic makes use of the extra core but note that the datum point may be changed and so increasing core reservations directive should appear before any storable words.

11.4. Example of how some users develop a Basic Program

This is to use the technique of “patching”. This obviates the need to re-edit the program each time a correction is made since the correction is inserted at “run time” from the job tape.

N.B. This technique should be used with care!

The program when originally written is organised as follows:-

(i) The programmer has to allocate extra drum and core space (the patch area).

(ii) It probably is advisable to divide the program into two sections. One is the job-tape which will contain amongst other things the directives for reserving core, peripherals, drum etc. and will terminate with the ENTER directive to enter the program (using this patching technique the ENTER directive on the program section will have to be replace by END directive). The second section is the “storable word” part of the program and will be a document with the name for example

MY/PROG/CHAR/VERSION/MK6/XYZ/1.2.66

and will be terminated by END directive (the job tape will ask for this document to be read by a USE directive).

An example to explain "patching” follows:-

The character version will be organised thus

MY/PROG/CHAR/VERSION/MK6/XYZ/1.2.66

```
              V1 = 0
              V2 = A100
              150     L5      L6      50
              00S     L7      L8      0
  L7))L5)     ..........................
              ..........................
      L8)     14      A1      0              | Start
              ..........................
  L13))L14)   04      L20     L21
              ..........................
  L15))L16)   14      L22     L23
              ..........................
  L17))L18)   12      L24     L25
              ..........................
  L100))L101) V0 = V2+40                     |Patch area 40 words long
              L6 = V2-L1                     |Length of program
                                             |including patch area.
              END                            |Not ENTER 0 which will
                                             |appear in job-tape
```

Examples of corrections to be made

(a) An Alteration e.g. that the word in drum address L13 (at L14 when in the core) is to be altered to

```
                    44    L20    L21
```

(b) A Deletion e.g. that the word in drum address L15 (at L16 when in the core) is to be effectively deleted by inserting a dummy instruction.

(c) An effective Insertion of several lines e.g. after the word in drum address L17 (at L18 in the core) - these lines being

```
                    34    L26    L27
                    60    V2+2    A3
                    30    L28    L29
                    15    L30    L31
```

The job-tape to insert those corrections and run the program would be:-

```
JOB       CANDR
RES       *DRUM
RES       *CORE
RES       *Peripherals
USE       MY/PROG/CHAR/VERSION/-
| Separator
V1 = L13                          | Correction (a)
          44      L20     L21     |

V1 = L15                          | Correction (b)
          74      0       0       | Dummy instruction which
                                  | never jumps
V1 = L17
          75      L101    0       | Put instruction when obeyed
                                  | jumps to patch.
V1 = L100                         | Store instructions in patch
                                  | area.
V2 = L101
          12      L24     L25     | Repeat the overwritten
          34      L26     L27     | )
          60      V2+2    A3      | ) Insertions
          30      L28     L29     | )
          15      L30     L31     | )
          75      L18+1   0       | When obeyed jumps back

          ENTER 0                 | Causes entry to program
```

Notes:
Correction (a). When the storable word

```
          44      L20     L21
```

comes to be stored in drum address L13 there should not be any outstanding forward references to be filled in for this word if they are filled in before ENTER is read.

Correction (b). This type of correction will work in certain cases. If a string of instructions (compound string) is being corrected, then it would be advisable to insert the dummy instruction

```
          117     0     0
```

instead.

The program and job-tape will not be left in this form once the corrections work. It is then advisable to re-edit the character version using EDITOR obtaining a new line printer copy, and to punch a new job-tape. This will leave a clean patch area for further corrections .

Some users map the program even for development. The purpose of mapping a program is to speed up its input. Assuming that during development the program is read in several times, the time saved in reading will make up for the time taken to map the program in the first place. The mapping run of the program section will ask for the L identifiers etc. to be appended so that when the mapped version is read in, it is as though the character version had been read in, and so the above patching technique can be used. Note that the use of some Basic facilities (e.g. optional settings) are lost when mapped. All Basic programs produced by Symbolic can be mapped correctly.

For details of MAPPER see the Orion Library Specification. An example job tape for mapping the program section is:-

```
JOB MAP

RES *MT1 MY/PROG/CHAR/VERSION/-

SCR *MT10 1.6.67 MY/PROG/BIMAP.VN/WITH.LS/MK6/XYZ/1.2.66

L4 = 1

READ ORION/SYSTEM/MAPPER/-
```

Note that MAPPER inserts a RESERVE *DRUM directive (and a setting of V1) in the mapped part of the mapped version, and so if the object program requires more drum (than just for the storage of the object program itself), say for space to hold intermediate results then the RESERVE DRUM directive must appear after the USE directive in the “patching” and run job tape example already given. The document request name with the USE directive will be that of the mapped version.

Once a program is fully developed(!) and re-edited, a new mapped version without the L identifiers .perhaps would be used since this version would be shorter and use lass drum during input.

11.5 Information about Symbolic Compiler which may be helpful in Speeding up Compilation

There are two phases of compilation
Reading in
Wind up

(i) Reading in

During this phase each line is checked for errors and is compiled if possible into Basic, storing this on either tape or drum. Lists are made of the Symbolic identifiers read and their settings. A Basic identifier for each Symbolic identifier is generated and a list of these kept. Other lists of directives, lists for library subroutines to assemble ate. are also made.

(ii) Wind up

During the first half of this phase the job will have no active peripherals because Symbolic works on the lists. Checks are made that all identifiers have been set; unset or reset ones being put into appropriate lists.

The second half of this phase includes, if report level 2 has bean specified, outputting the symbolic identifiers . As reporting and compilation are carried out together reporting does not slow up the job too much. This second half also consists of outputting the compiled program, normally this is magnetic tape (or handing over to Basic if load and go). Note that if a punch is used for Report Level 2 or for the compiled program then the job may become punch limited.

Core-store and Drum Requirements

Symbolic needs at least 2544 words of core (see 7.2.C.2). Some of this is for working space, symbolic uses drum for working space and reserves more as it needs it. Symbolic has to store the Basic character version of the compiled program somewhere and the answer to this question (see 7.2.R.1.1) tells Symbolic whether this is to be tape or drum.

Lists have to made of (among other things) the Symbolic identifiers mentioned, their setting, their corresponding Basic identifiers, lists of directives, assembly lists (e.g. which library subroutines to get) etc. Larger core area will eliminate some of the drum references for searching these lists and so decrease the elapsed time but of course mill time used will be the same. The Symbolic identifiers are stored five characters to a word (routine name prefixes are not stored with each identifier and do not count towards the five) and so some saving in space and searching time can be achieved by using identifiers not more than five characters long.

11.6 Examples of How Some Users Develop a Basic Program produced by Symbolic Compiler

Since Symbolic produces a program in Basic Input language, the patching technique already described (in 11.4) can be used. This means that corrections to the program can be inserted at run time , thus saving time during development since recompilation will not be necessary each tine an error is found.

The Symbolic program will be thought of as being in two parts; firstly the job tape information containing the directives for reserving core, drum and peripherals and secondly the document containing only storable word information. The latter will be organised in the following way:

(i) will be terminated by END directive (see 7.2.C.2.9) and not by ENTER directive which will appear on the run job tape.

(ii) will have a patch area, the first word of the patch area will be identified with both a core and a drum label. Each chapter may have its own patch area.

(iii) will identify the first storable word of each chapter with both a core and a drum 1abel. It is assumed that each chapter will have only one TRANSFER directive of the TRA A100 type.

For example:

```
DOCU  MY/PROG/SYMBOLIC/CHAR/VERSION
      TRA A100
      DRUM START)) C RESTART) +3
                             ..........
                             ..........
                             ..........
                              START 0
                             ..........
                             ..........
                             ..........
                       BUZZ)   04 JOE A6
                             ..........
                             ..........
                             ..........
                        JOE = A13
                        ..................
                        ..................
                        ..................
      PATCHDRUM))PATCHCORE) TRA +40+            | Patch area 40 words long
      END                                       | Not ENTER
```

This document originally on paper tape, will probably be copied onto magnetic tape from which a Basic character version will be obtained also on magnetic tape. The Compiling run to produce this Basic character version will have asked for "Report level 2" (see 7.2.C.2.6 and 7.2.R.2.2) information giving a list of the Symbolic identifiers and their equivalent Basic identifiers (there is no need to have a printout of the Basic character version). The Compiling run thus gives a document, e.g.

```
      DOCUMENT MY/PROG/BASIC/CHAR/VERSION
      The "Report level 2" will give, for example
            L1026  DRUMSTART
            L1315  CORESTART
            L335   BUZZ
            L2167  PATCHDRUM
            L2025  PATCHCORE
            L5172  JOE
```

CORRECTIONS

Assuming that the line "labelled" BUZZ (core label) is to be altered, deleted or lines inserted, then it is necessary to give the drum address of this BUZZ line (i.e. where BUZZ is stored in the drum). This is (if only one TRA A100 type in the chapter in which BUZZ occurs)

Drum starting address + (current core address - core starting address)

```
            DRUMSTART+BUZZ-CORESTART
```

but of course, on the "correct and run" job tape the equivalent Basic

identifiers have to be given - these are found from the Report level 2 list. The required value of V1 (drum transfer address) is

V1 = L1026+L335-L1315

and specifies BUZZ line on the drum.

Alteration

The BUZZ line is to be altered to effectively be

```
04      JOE+1    A6
```

then the Basic equivalent of this is

```
04      L5172+1  A6              |JOE ≡ L5172
```

Thus on the "correct-and-run" job tape would appear the lines

```
V1 = L1026+L335-L1315
04      L5172+1  A6
```

which results in the required alteration on the drum.

Deletion

See the Basic write up i.e. the instruction in some cases would be replaced by

```
74      0        0
```

Insertion

For example if the following lines are to be effectively inserted after the BUZZ line

```
04    JOE+5    A20
60    +2+      A3
10    A1       0
11    A2       5
```

then one would need on the job tape

```
V1 = L1026+L335-L1315
75    L2025    0        |75 PATCHCORE 0
                        |in place of BUZZ line
```

followed by the instructions to be in the patch area

```
V1 = L2157             |V1 = PATCHDRUM
V2 = L2025             |V2 = PATCHCORE
04    L5172    A6      |repeat BUZZ line
04    L5172+5  A20     |)
60    V2+2     A3      |)
10    A1       0       |) insertions
11    A2       5       |)
75    L335+1   0       |Jump back.

      ENTER    0       |To transfer the chapter including its
                       |patch area, which contains START 0
                       |from drum to core and enter the Program
```

Note that there is the facility of specifying the Basic identifiers L0 to L255 in a Symbolic program (see 7.2.G.2.3) and so one could have for example

```
 TRA A100
*L1))*L2)   +3
          .........
          .........
          START 0
          .........
    BUZZ) 04 JOE A6
          .........
          .........
          .........
      JOE = A13
*L3))*L4) TRA +40+
 END
```

so that the Report level 2 reporting is given for BUZZ and JOE only and thus

V1 = L1+L535-L2

would give the drum position of the BUZZ line etc.

i.e. L1 replaces L1026 (DRUMSTART)
L2 replaces L1315 (CORESTART)
L3 replaces L2167 (PATCHDRUM)
L4 replaces L2025 (PATCHCORE)

A job tape to insert corrections in the patch area and run the program would be

```
JOB  RUNCOR
RES  *DRUM
RES  *CORE
RES  PERIPHERALS
USE  MY/PROG/BASIC/CHAR/VERSION/-
| Separator
V1 = L1+L355-L2
75   L4  0

V1 = L3
V2 = L4
04   L5172    A6
04   L5172+5  A20
60   V2+2     A3
10   A1       0
11   A2       5
75   L335+1   0

ENTER 0

END
```

Mapping a Program

A Basic program produced by Symbolic Compiler can be mapped and in some cases it may be worthwhile mapping the program even for development.

11.7 Efficiency of Programs written by a User in Basic Input Language and run on Orion 1

The remarks that follow are observations made by a user and may apply in his case only. Once a program has been written it is worth while reviewing it to ensure reasonable running times (by perhaps giving the program more core store?). The programmer worked out whether the job would be peripheral limited or mill limited (i.e. allowing for other jobs in the machine it was possible to calculate roughly from Flexowriter output percentage mill time/elapsed time). He mentions that he learnt much from watching the peripherals when the program was running. In some cases he expected the peripherals to move at full speed, and in those cases where not expected to move at full speed he expected them to move smoothly or not unnecessarily jerkily. For example he expected an updating program on a low activity file (10% say) to keep the tapes moving at $^{2}/_{3}$ full speed with 2 controls (at ½ full speed with 1 control). He expected an off line job (a peripherals limited job) to keep the slow devices moving quickly. When much mill was used and the peripherals were moving slowly he ran tests to find out to where this mill time was "disappearing". He decided that there was something wrong with a top priority job when the mill time used was significantly less than elapsed time and the peripherals were moving evenly but more slowly than expected. In this case he concluded that the job was held up because of lockouts (he used double or multiple buffering to prevent this), or perhaps because monitoring other than Style 0 was switched on, or perhaps 150 instructions, mainly chapter changing instructions, were unnecessarily obeyed (perhaps in an inner loop). If a program ran intermittently (and it was the only program: in the machine) he wondered if branching the program would be worth while. In the case of mill limited jobs he attempted to detect which part of the program was slow by timing sections of the program and in particular, he examined general purpose routines (especially much used input and output routines) which may have caused all his programs to run more slowly than expected.