*Technical Report*

UCAM-CL-TR-15
ISSN 1476-2986

Number 15

**UNIVERSITY OF CAMBRIDGE**

**Computer Laboratory**

# The implementation of BCPL on a Z80 based microcomputer

I.D. Wilson

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

*http://www.cl.cam.ac.uk/*

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

*http://www.cl.cam.ac.uk/TechReports/*

ISSN 1476-2986

Introduction. The main aim of this project was to achieve as full an implementation as possible of BCPL on a floppy disc based Z80 microcomputer, running CP/M or CDOS (the two being essentially compatible). On the face of it, there seemed so many limiting factors, that when the project was started, it was not at all clear which one (if any) would become a final stumbling block. As it happened, the major problems that cropped up could all be programmed round, or altered in such a way as to make them soluble.

The main body of the work splits very conveniently into three sections, and I hope in covering each section separately, to be able to show how the whole project fits together into the finished implementation.

The Sections are:

[A] - The Z80 Code Generator

[B] - The Run-time system (BCPLMAIN and BLIB)

[C] - The Bootstrap and final implementation

Acknowledgements.

I would particularly like to thank the following people:

Dr. Martin Richards

for constant advice and suggestions, but mainly for his experience in past implementations on other systems.

Dr. Arthur Norman

for help on anything which happened to be problematic, especially for advice on multi-precision arithmetic.

Dr. Anthony Fretwell-Downing and
Mr. Christopher Webster of F. Fretwell-Downing Ltd.
for the loan of a 48K North Star Horizon system on which to do the development work, and for seemingly endless encouragement and cynical comments (respectively).

## Table of Contents

### Section A

Table of Contents (Contd.)

Section B



Summary

Section C



Summary

Bibliography



(The Appendices can be obtained on request from the Author)

# Table of Figures

# Section A

# A Z80 Code Generator for BCPL

## A.1 Background

This was by far the most important, and in fact the most difficult of the programs to write, the reason being the stringent limitations that a processor such as the Z80 places on the compiler writer. By far the worst of these limitations was the amount of real memory available. The machine on which the final compiler had to run had 48K bytes of memory, of which about 10K bytes was taken by the operating system, leaving about 38K bytes of workspace. Another space limitation is that of the size of the floppy discs. Nominally, these were single sided, double density five inch discs, having a capacity of about 180K bytes each. When the system tracks have been removed (directory track, and two tracks of operating system), about 150K bytes is available to the user.

## A.2 Initial Design Decisions.

This section contains references to OCODE, Z80 assembler code and INTEL standard OBJ modules. Specifications of these items can be found in references [2], [10], and Appendix I respectively.

The task of any BCPL code generator is to take as input, OCODE produced by the frontend of the compiler, and produce as its result, some form of target machine code. OCODE is designed in such a way as to deal with the idealised BCPL machine, and the code generator must be capable of translating, and optimising the code for this 'ideal' machine, tailoring it for the hardware available.

## A.3 An OCODE interpreter.

For sheer simplicity, there is a tendency to macro-generate machine code from OCODE, producing a semi-interpretive OCODE machine with a run time system designed to provide the OCODE primitives. Advantages of such a system are that the code generator is small, easy to write, wordsize independent, and hopefully, easy to debug.

An example of such a macro expansion might be:

| OCODE | | Machine code | |
|---|---|---|---|
| STACK | 12 | // no code | |
| LN | 3 | CALL $LN | ; DW 3,12 |
| LP | 4 | CALL $LP | ; DW 4,13 |
| PLUS | | CALL $PLUS | ; DW 12 |
| RV | | CALL $RV | ; DW 12 |
| LP | 5 | CALL $LP | ; DW 5,13 |
| STIND | | CALL $STIND | ; DW 13 |

In each case, there is a routine in the run time system to deal with the OCODE operator, and this is followed by one or more words of information needed by the interpreter. For instructions that deal with the stack, the current stack size is needed, so the code for RV has a word containing 12 (the current stack size) after it. For OCODE statements that require arguments, these are also stored after the routine call. Thus statements like LP or LN which require both an argument and a stack size have two parameters stored.

The above machine code is no more than a glorified interpretive code. There would be very little difference to the running speed of the code if the 'CALL $xxxx' instructions were replaced by a byte representing '$xxxx' this would then be a true interpretive code, and much more compact than the example given above.

There were two possible problems with designing a code generator on these principles. Firstly, since the translator has no internal optimisations of its own, the OCODE itself is not of particularly good quality. Secondly, there is the problem of inefficiency, and an interpreter based on OCODE did not have the makings of one of the world's fastest programs.

Unperturbed by these considerations, a code generator which generated interpretive code was written, just to see how big and poor the resulting compiled code would be. The

code generator was small - about 800 lines of BCPL to be precise, and its structure was extremely simple. After all, it was no more than an OCODE translator. The major drawback with this system as it stood, was that these 800 lines macro generated into about 50K bytes of very poor machine code. Considering that the implementation had to be accomplished on a 48K byte machine, this method was somewhat unfortunate.

At this point it was clear that an optimising code generator would have to be written. The ideal situation would be to reach the trade-off point between the quality and compactness of the compiled code, and the size and complexity of the code generator itself.

## A.4 Initial structure of the code generator.

After reading carefully three other BCPL code generators (those for the IBM/370, NOVA and PDP11) it became clear that the task was nowhere near as daunting as it had first appeared. All three of the code generators were based on the same basic structure, and it was only the machine dependent detail that varied.

There seemed little point in redesigning a system that had obviously worked many times before, and so the essential logic of the code generator was taken from these three. The overall structure is adapted heavily from the IBM/370's code generator, whereas most of the fine detail was taken from the PDP11's, a machine whose internal architecture is much more akin to that of the Z80. Another major reason for keeping to a well proven algorithm is that it would be possible to find out where the bugs were likely to show themselves, and, when they did, how they had been fixed on previous systems.

The next step was the design the abstract machine for which to generate the code. Primary considerations were simplicity of structure, leading hopefully to a code generator small enough to fit on the target machine, and compactness of compiled code. Speed was a somewhat secondary issue, though as it turned out, the code produced by the BCPL compiler is as fast, if not faster in some cases, than that produced by its FORTRAN counterpart.

Since the Z80 possesses two index registers, it seemed natural to use these to point to the stack and global vector, (IX and IY respectively). This left the other registers and the Z80's stack. At first, it seemed feasible to operate a two stack system - the BCPL stack running upwards, holding parameters to procedures, local variables

etc., and the Z80's stack running downwards, holding stacked return addresses. This scheme would work fine on most BCPL programs, except those using the library procedures APTOVEC and LONGJUMP, the latter now requiring three parameters, rather than the usual two. The idea was scrapped very early on.

The Z80's stack is used to pass arguments to routines in the run time system, while the registers are treated as three general purpose BCPL registers, HL - DE - BC, with an eight bit work register, A.

## Code Generator Data Structures

### A.5 Fundamental Structures

The code generator depends very heavily on two items of data structure, both of which are defined to be interfaces between the abstract BCPL machine represented by OCODE, and the actual target machine. These are the SIMULATED STACK and the REGISTER SLAVE .

Since most of the information about the flow of control of a BCPL program has been lost once the OCODE stage is reached, there is very little that the code generator can do to optimise this aspect of the program. On the other hand, OCODE is ideal for stack access optimisation i.e. the simulation of the run time stack and the slaving of registers within the code generator. It is not possible to know the exact values of all items on the run time stack, nor is it possible to know exactly the values in the machine registers at a particular time. However, through cunning use of simple data structures, it is possible to hold all information needed to run an efficient register slave, and to optimise the code compiled to access the run time stack.

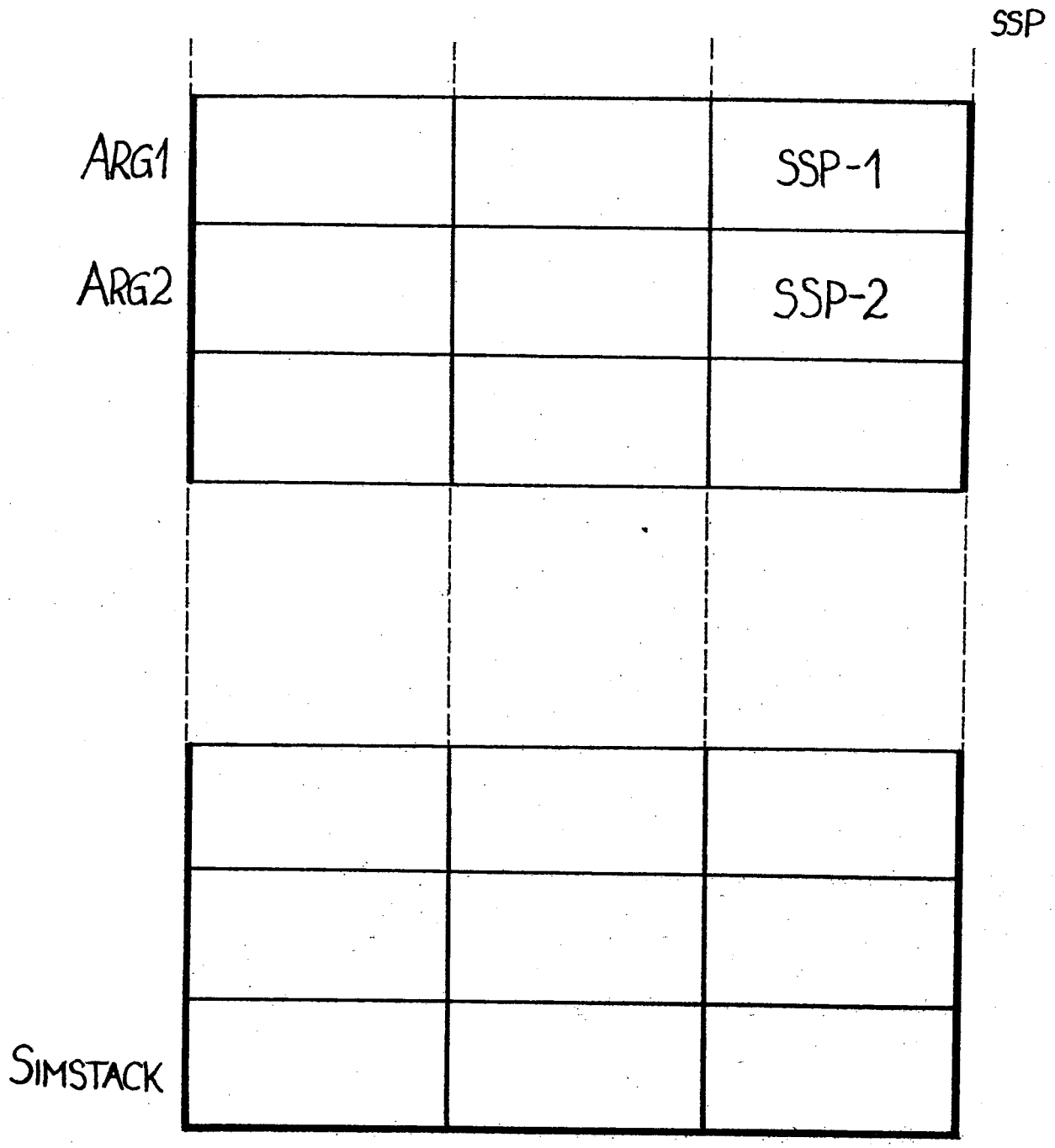



Fig A.5.1 The Simulated Stack

Each simulated stack item can be considered as a triplet containing information about the item at a corresponding position on the real run time stack.

A general simulated stack item (triplet) is given below:

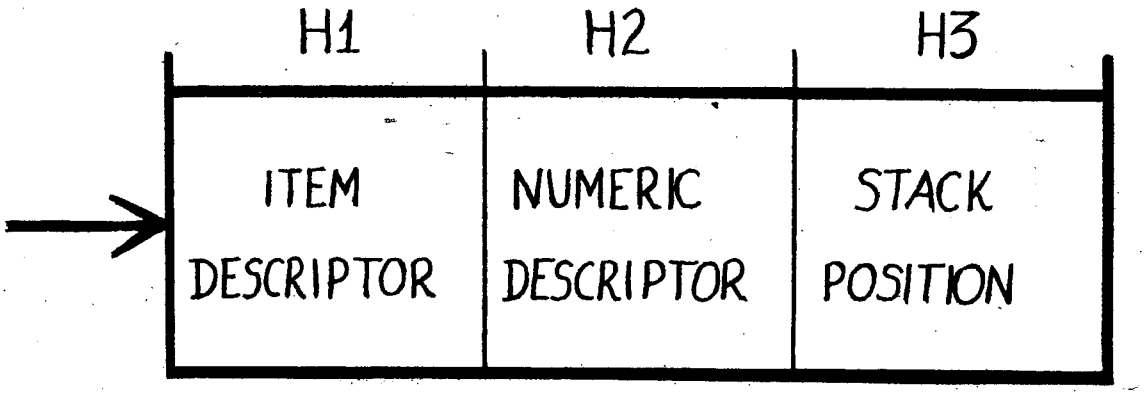



Fig A.5.2 A Generalised Simulated Stack Item

All items which deal with the simulated stack data structure are BCPL pointers, i.e. ARG1 points to the triplet representing the top item on the simulated stack. The one exception to this rule is the variable 'SSP' which is set to the numerical position of the next free item on the simulated stack. Position zero is considered as the base of the current stack frame.

## A.6 Simulated Stack Item descriptors.

The item descriptor can have one of nine values, representing the possible item on the run time stack. The numerical descriptor in item H3 of the triplet is here referred to as 'x'.

| Type | Meaning... |
|---|---|
| LOC | The value of the local variable found at offset 'x' from the current stack pointer. |
| GLOB | The value of the global variable found at offset 'x' from the current global pointer. |
| NUMB | The value of the numerical constant 'x'. |
| LAB | The value held in the STATIC location at CG label 'x'. |
| LVLOC | The address of local variable 'x'. |
| LVGLOB | The address of global variable 'x'. |
| LVLAB | The address of static variable 'x'. |
| REG | The value currently stored in machine register 'x'. |
| STCK | The item currently on the top of the Z80's stack. |

## A.7 The Register and ZSTACK slaves.

Related to the REG and STCK items are two more pieces of data structure, the REGISTER SLAVE and the ZSTACK slave.

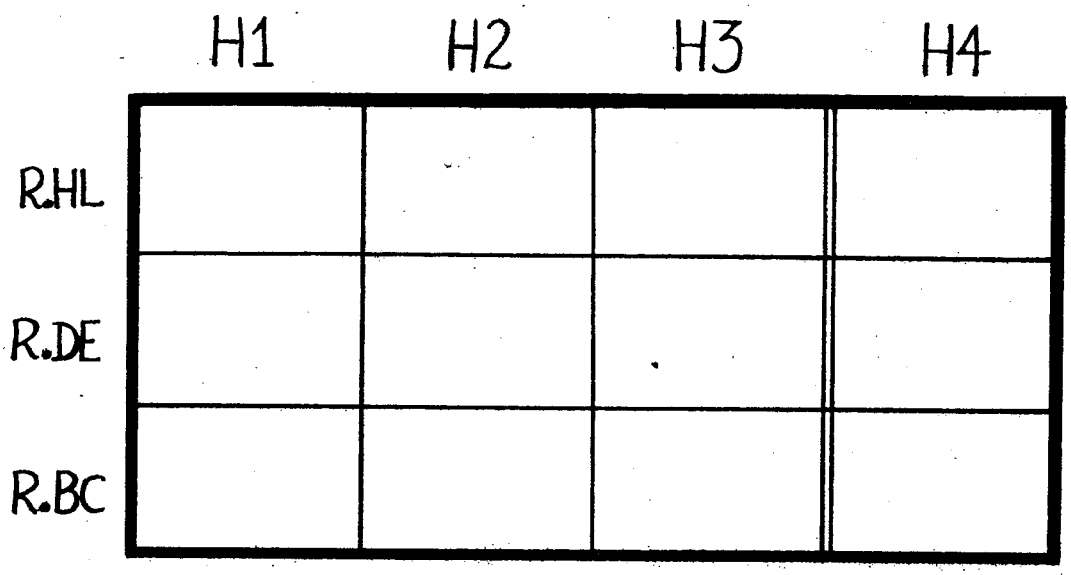



Fig A.7.1 The Register Slave

Here the triplet has become expanded by the addition of one more word. This word holds the internal manifest value of the register concerned, for use in the routines which generate code. This does not affect the actions of the slave at all, and the register slave should be thought of as being composed of triplets, just like the simulated stack.

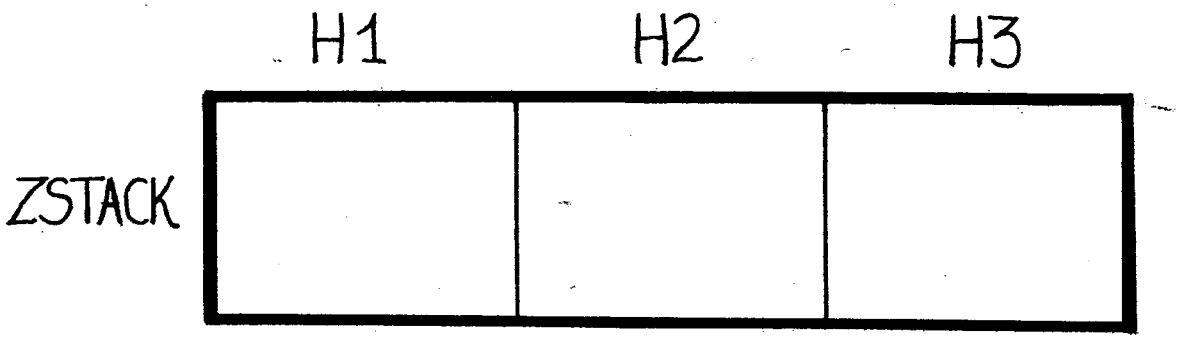



Fig A.7.2 The ZSTACK Slave

The ZSTACK slave is a one-triplet slave holding information about the item currently on the top of the Z80's stack. As far as slaving is concerned, the ZSTACK is treated almost as another register.

There is one extra type of item for these two slaves. Since it is possible that the states of registers, or of the Z80's stack are undefined, a type 'NONE' has to be introduced to represent this.

It is worth pointing out at this juncture that the register and ZSTACK slaves can never contain information about themselves. The reason for this is to avoid recursive definitions of slave information. For instance, it is theoretically possible for the ZSTACK slave to contain information leading to the stunning conclusion that the top item on the Z80's stack is in fact the ZSTACK item, i.e. the top item on the Z80's stack...

## A.8 Summary of Information held in Slaves.

At any point in the compilation of a program, the code generator knows the state and size of the run time stack, the state of the machine registers, and the state of the top item on the Z80's stack. The information is stored simply, but by the way that the various parts of the data structures inter-relate, it provides a full description of the state of the run time machine, and facilitates the production of respectably good machine code.

## A.9 Inter-relation of the slaves

It is essential at this point for the reader to be familiar with the simulated stack item descriptors in relation to the run time objects which they represent. By example, I hope to show how the REGISTER and SIMSTACK slaves interact as items are brought into registers, and then stored again. By analogy, the ZSTACK slave acts just as a single item register slave, and is treated as such by the Code Generator.

## A.10 Examples of Slave Manipulation

| | | | |
|---|---|---|---|
| R.HL | NONE | ? | ? |
| R.DE | GLOB | 76 | 7 |
| R.BC | NUMB | 12 | 4 |

SSP=5

| | | | |
|---|---|---|---|
| ARG1 | LVLAB | 127 | 4 |
| ARG2 | LOC | 2 | 3 |
| | NUMB | 20 | 2 |
| | LOC | 1 | 1 |
| | LOC | 0 | 0 |

Fig. A.10.1 Initial state of the Slaves

| | | | |
|---|---|---|---|
| R.HL | NONE | ? | ? |
| R.DE | GLOB | 76 | 7 |
| R.BC | NUMB | 12 | 4 |

SSP=6

| | | | |
|---|---|---|---|
| ARG1 | GLOB | 100 | 5 |
| ARG2 | LVLAB | 127 | 4 |
| | LOC | 2 | 3 |
| | NUMB | 20 | 2 |
| | LOC | 1 | 1 |
| | LOC | 0 | 0 |

Fig. A.10.2 State after the OCODE statement 'LG 100'

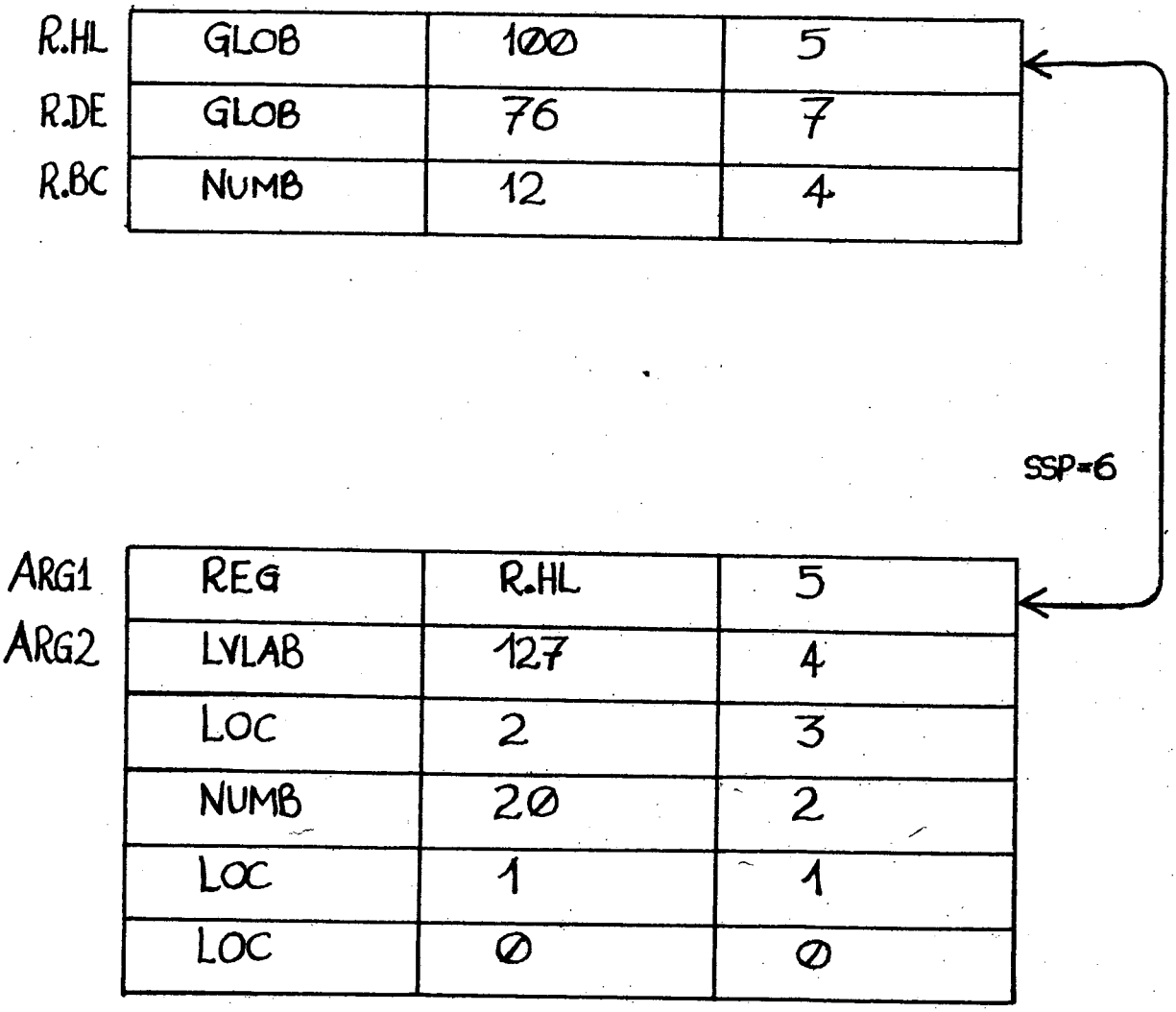



Fig. A.10.3 State after ARG1 loaded into HL

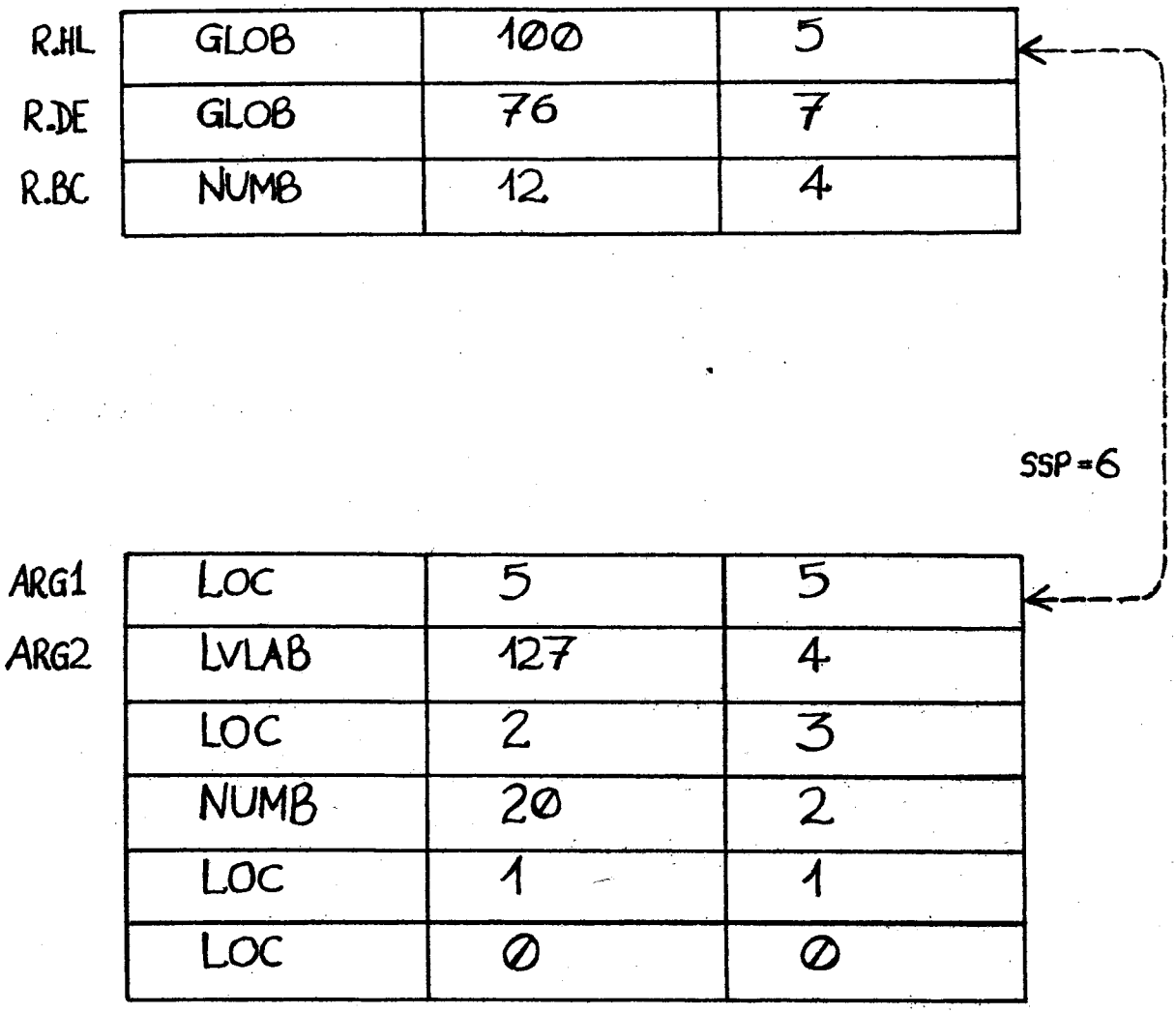



Fig. A.10.4 State after HL stored onto the stack

## Code Generator Primitives

Having described the data structures, it is now necessary to introduce the code generator primitives required for manipulating these structures in a useful and efficient manner.

At any point in the compilation, the code generator must be able to:

(1) Move any simulated stack item into a machine register of its choice.

(2) Dump any machine register in the store location of its choice.

(3) Set the Simulated Stack to be any arbitrary size.

(4) Set the Simulated Stack to be in some pre-defined standard state.

(5) Load any item onto the Simulated Stack.

(6) Search the Register Slave for a particular item.

(7) Discard the values held in any or all of the slaves.

## Routines providing the Code Generator Primitives

### A.11 Move a Stack item into a Register.

There are three routines provided for this purpose, with varying degrees of generality.
These are:

```
MOVETOR ( <register> , <SIMSTACK item> )
R := MOVETOANYR ( <SIMSTACK item> )
R := MOVETOANYBUT ( <register> , <SIMSTACK item> )
```

Here a <register> is a pointer to a triplet in the register slave, such as one of the variables R.HL, R.DE or R.BC . A <SIMSTACK item> is a simulated stack item, i.e. a pointer to a triplet on the simulated stack, such as ARG1 or ARG2.

The purpose of 'MOVETOR' is to move a specific stack item into a register, used for example, when code for an operator is about to be compiled. All updates to the SIMSTACK and register slave are done in this routine.

A typical call might be:

```
MOVETOR ( R.HL , ARG1 )
```

which would result in the top item on the SIMSTACK being moved into the HL register pair, and the triplets for R.HL and ARG1 being updated.

MOVETOANYR and MOVETOANYBUT are used in more general cases. MOVETOANYR will move a simulated stack item to the most suitable register available, whereas MOVETOANYBUT will move the item to any register BUT the one specified. Both these routines necessitate a search of the register slave table to find, if possible, a register with the required value already available. If this is not possible, then the function NEXTR returns the next available register for use by the code generator.

NEXTR uses the following algorithm for the allocation of registers:

(1) Allocate an UNDEFINED and UNUSED register

(2) Allocate an UNUSED register

(3) Allocate the register which was least recently referenced, dumping its current value at the required position.

UNDEFINED in this context means that the register slave contains 'NONE' in the data type field for this register; UNUSED means that there are no references to it on the current simulated stack. It can be seen that an UNUSED register is not necessarily undefined, although the converse is usually true.

In all the above cases, NEXTR would allocate the registers in the order HL - DE - BC, i.e. the most general and usable register first.

MOVETOANYBUT is analagous, but will <u>never</u> allocate the register specified in its arguments. Here the register allocation is done by the routine NEXTBUTR, corresponding exactly to NEXTR, but never allocating the specified register.

A typical use of each of these routines might be:

```
R := MOVETOANYR ( ARG1 )
R := MOVETOANYBUT ( R.HL , ARG2 )
```

Both MOVETOANYR and MOVETOANYBUT actually load the item into the allocated register using MOVETOR. The result is that all register slave and simulated stack updating is done within the central routine, rather than in its more general dependents.

## A.12 Storing of Registers

There are two completely separate occasions when this is necessary:

(1) When an OCODE 'Sx' instruction is being dealt with, and the top item on the SIMSTACK has to be explicitly stored.

(2) When a register is required for another purpose, and the value needs to be dumped for use later. This is needed specifically when the stack needs to be in a standard state for a transfer of control.

These two occasions are treated independently. For the first, there is a routine 'CGSTORE' which takes two arguments: the type and position of where to store the top item on the SIMSTACK (ARG1). For example the OCODE 'SG 100' would result in a call 'CGSTORE( GLOB, 100 )'.

The philosophy adopted by the code generator is that an item cannot be stored until it has first been loaded into a machine register. If the item is not already in a register (H1 ! ARG1 \= REG) then ARG1 is moved into an available register. Once loaded into a register, the item can be stored by a mechanism which is effectively the reverse of that for loading.

The other possible reason for dumping a register is that, either the register is needed for another purpose, and the value in it is too valuable to ignore, or to get the stack into a standard state ready for a transfer of control, e.g. a JP (jump) instruction, or a procedure call. The routines provided to do this are 'STORE' and its subsidiaries 'STORET' and 'FREEREG'.

FREEREG takes two arguments: a register, represented by a pointer into the register slave like R.HL, and a simulated stack item. Its purpose is to free the register given as the first argument, from all references on the simulated stack OTHER than that given as the second argument. This is useful, especially in the routine MOVETOR described earlier, where the register given needs to be freed of all references, other than the one which will eventually be

loaded into it.

A typical call might be:

```
FREEREG ( R.HL , ARG1 )
```

which has the effect of removing all references to the HL register pair in the simulated stack, apart from the one in ARG1 (if any).

STORE also takes two arguments: low and high water mark references to the simulated stack. A call of 'STORE(x,y)' has the result of putting locations between positions x and y on the current stack frame into a standard state, i.e. actually stored on the real run time stack. STORE is most useful in conjunction with the variable 'SSP', which is the position of the next available simulated stack item. Thus a call of 'STORE( 0, SSP )' has the effect of stacking all items between the base and the top of the current simulated stack, i.e. putting the whole of the current stack frame into a standard state. Similarly, 'STORE( 0, SSP-2 )' has the effect of stacking all but ARG1. (In this context 'SSP' and 'SSP-1' have the same meaning, as SSP holds the position of the first free cell, and not the top item on the stack).

Subsidiary to STORE is the routine STORET which stores a simulated stack item, if necessary, onto the run time stack. It does this by inspecting the triplet passed to it, and generating code to do the store if the item is <u>not</u> of the form 'LOC, n, n', (i.e. already stacked).

## A.13 Manipulation of Simulated Stack Size

The two routines provided to do this are STACK and INITSTACK, both of which take as arguments, the size which the simulated stack will be after return from the routine.

A call of 'INITSTACK( n )' has the effect of initialising a new simulated stack frame to the size 'n'.

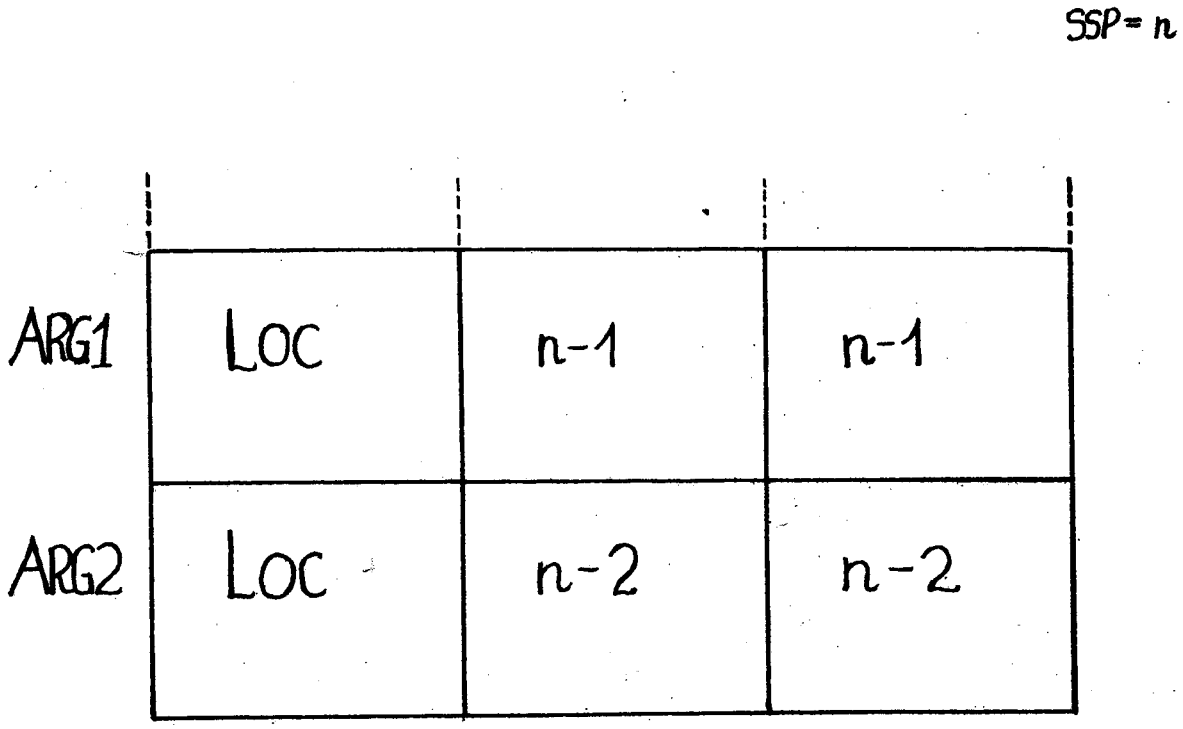



Fig A.13.1 An initialised Simulated Stack Frame

A triplet of the form 'LOC, n, n' has the meaning of a LOCAL variable, already stored at its pre-destined position on the run time stack.

A typical call might be:

INITSTACK ( 2 )

which initialises a new simulated stack frame of size 2. This is the state of the run time stack frame just on entry to a procedure, with only two items on the stack: the return address and old stack pointer.

INITSTACK is essential in all situations when a new stack frame is being set up. Examples of this are on entry to a procedure, on setting a code generator label, or if a relative stack size change would cause overflow.

STACK is a more general routine, which sets the simulated stack size to its argument without re-initialising the stack. This is related directly to the OCODE 'STACK n' directive, but is applicable to many other situations as well. The majority of stack operations are relative: e.g. code for a diadic operator has just been compiled, and the stack needs bringing down by 2. Again the variable SSP comes in useful here, as all relative stack size changes are just increments or decrements of this variable. Thus 'STACK ( SSP - 2 )' would bring the simulated stack down by the required amount after generating code for the diadic operator.

Depending on the magnitude of the size change, STACK changes the stack size in three different ways.

Firstly, if the change, either up or down, is greater than 4, then the entire stack frame is 'STORE'd, and the new stack frame is initialised using INITSTACK.

Secondly, if a stack increment of <= 4 is required, then this is accomplished by 'LOAD'ing items of the form 'LOC, SSP' until the stack is of the required size. (The LOAD routine is discussed more fully in the next section).

Thirdly, if a stack decrement of <= 4 is required, then this is done by repeatedly decrementing ARG1, ARG2 and SSP until either the bottom of the stack is reached, in which case a new stack frame is 'INITSTACK'ed, or the required stack size is reached.

## A.14 Loading a new SIMSTACK item

The routine provided for this facility is LOAD, which takes as its arguments the first two parts of a simulated stack triplet. The third part is redundant, and can be filled in by the system from the variable SSP, which holds the position of the next available simulated stack cell.

Given the current state of the simulated stack:

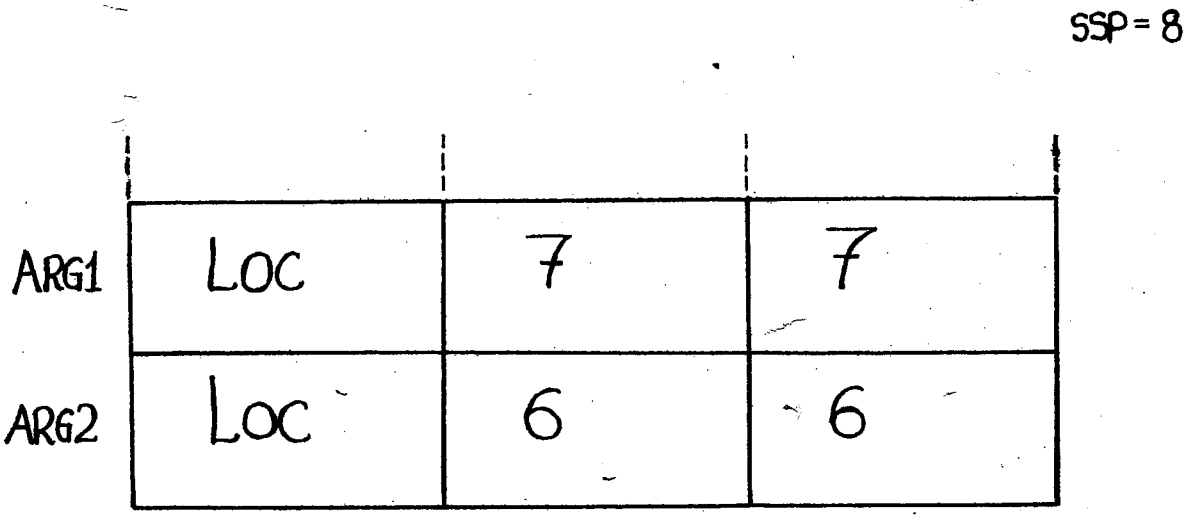



Fig. A.14.1 Simulated stack before a LOAD

After the execution of 'LOAD ( GLOB , 76 )' the state of the simulated stack would be:

SSP = 9

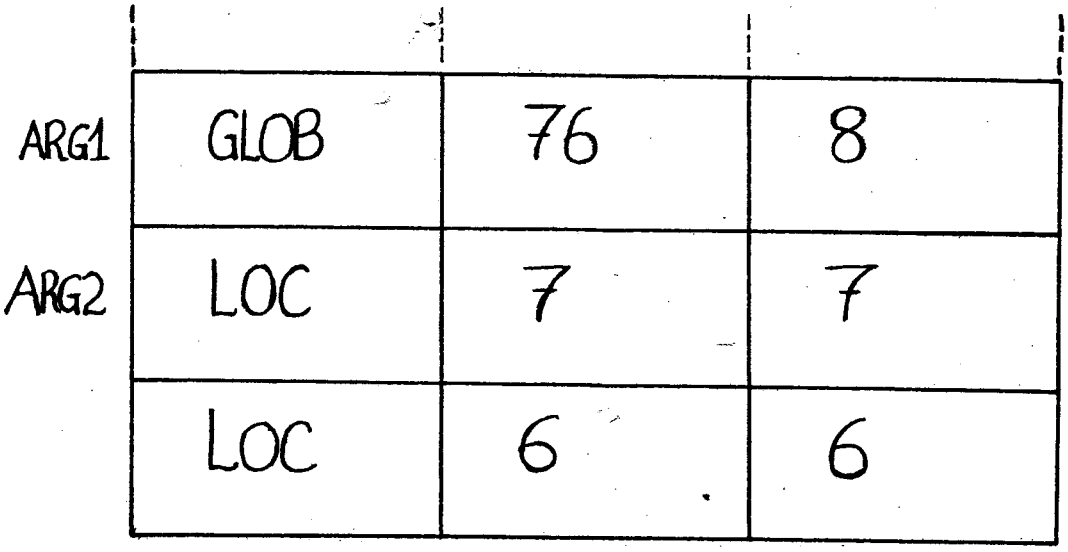



Fig. A.14.2 Simulated stack after a LOAD

This corresponds directly to the OCODE instruction 'LG 76'. LOAD is also used to stack the result, after code has been compiled for an operator.

Typically the code to deal with a diadic operator runs like:

```
MOVETOR ( R..., ARG1 )
MOVETOR ( R..., ARG2 )
----------------------
- code for operator -
----------------------
STACK ( SSP - 2 )
LOAD ( result... )
```

## A.15 Searching the Register Slave

The function LOOKINREGS is provided to seach for an occurrence of a particular stack item within the register slave. It returns as its result, a pointer to a register in the register slave, if the item has been found, or -1 if not.

The mechanism is a scan of the pairs of descriptive cells in the slave (items H1 and H2), comparing them with the item given as argument to the function.

A typical call might be:

```
R := LOOKINREGS ( ARG1 )
```

which returns a pointer to a register if the item ARG1 were already in a register, or another item with the same description was found. Given the following states of the simulated stack and register slave, the above call would have resulted in R.DE being handed back as the result.

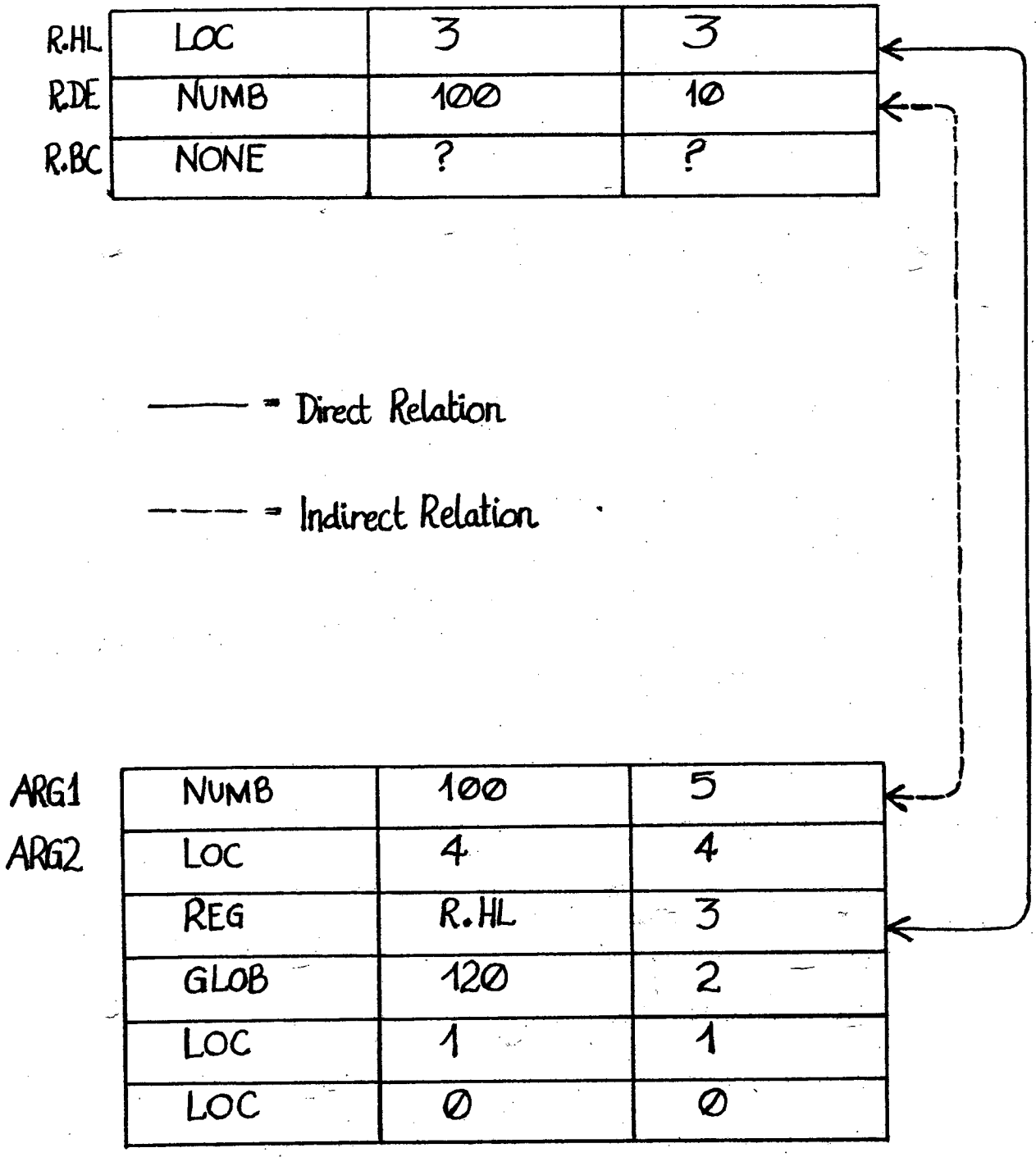



Fig A.15.1 LOOKINREGS - relationship between the Simulated Stack and the Register Slave

## A.16 Discarding Slave Information

There are several occasions when the information held in particular slaves has to be discarded. The ZSTACK slave needs to discarded after an item has been POPped from it into a register. The entire register slave must be discarded after encountering a label, an indirect assignment, or a procedure call. This is because, in the first case, the label can be accessed from any part of the compiled program, and so the registers will be in an undefined state at the label.

In the second case, this is the rather unfortunate side effect of the generality of the BCPL indirect assignment, which could have updated any word in store, and in particular, one referred to by an item in the register slave. Although the chances of this happening are somewhat unlikely, there is no alternative but to lose all information at this point.

The third case is similar to the first, i.e. the registers will be in an undefined state on return from a procedure call, and so the register slave must be discarded to take account of this.

The routines provided are:

DISCARDSTACK, which discards the information in the ZSTACK slave.

DISCARDREG(R), which discards the information in the register slave, relating to register 'R'.

DISCARDREGS, which discards the information in the entire register slave.

DISCARDADDRESS(R), which discards the information about register 'R' if it refers to a LOCAL variable which happens to be above the current simulated stack pointer. This is of necessity whenever the size of the simulated stack is reduced.

## A.17 Routines to generate Code

All routines which generate code begin with the prefix 'CODE.', and take arguments related to the instruction they represent. For example, consider the 'INC' instruction. The routine provided to generate code for this instruction is 'CODE.INC', and takes the manifest value of a 16 bit register pair.

So:

```
CODE.INC ( K.HL )
```

would cause an 'INC HL' instruction to be compiled. Certain routines take a variable number of arguments, e.g. the routine to generate code for the 'LD' instruction, 'CODE.LD'. On generating code for a register to register load, the call might be:

```
CODE.LD ( K.A, K.L )
```

whereas, if an indirected load were being compiled, e.g. 'LD L,(IX+10)', the call would be:

```
CODE.LD ( K.L, K.IX, 10 )
```

Other types of arguments to be passed to these routines are:

| | |
|---|---|
| K.I.HL | HL indirected, i.e. (HL) |
| K.LAB | The address of a label, followed by the label number |
| K.I.LAB | The contents of a labelled static cell, again followed by a label number |
| K.NN | A 16 bit numeric quantity |
| K.N | An 8 bit numeric quantity |
| A.xxx | The addresses of routines in the run time system. |

## The Top-Down Structure of the Code Generator

### A.18 Introduction

Given the primitives described in the previous section, it is possible to design an arbitrarily optimising code generator on top of them. It is in the macro-structure of the code generator that the major optimisations occur, not as would first appear, in the primitives themselves. Once the primitives have been tried and tested, then it is the changes to the way they are used, and in particular, to when they are used, that make by far the most striking differences to the compactness and quality of the compiled code.

Always at the back of my mind when designing the optimisation tactics for the code generator, was the somewhat sobering thought that the program eventually had to run in 38K bytes of memory. This is a fact which I hope will help to defend some of the decisions taken, and especially, explain why certain blatant optimisation opportunities have been ignored completely.

### A.19 The Main Program

The code generator is entered at START - a routine which does no more than print a logon message, and set up the workspace areas for the rest of the program. The rest of the driving program is taken up by the routine MAIN, which initialises essential variables, enters a main loop, generating code for successive BCPL sections.

The sizes of the individual work vectors are given by manifests in the header file "Z80HDR". Typical values are given in the following table for the three machines on which the code generator is now implemented.

| | Machine | | |
|---|---|---|---|
| | North Star 48K | Cromemco Z2/H 64K | IBM 370/165 200K |
| PROGSIZE | 2500 | 5000 | 10000 |
| RELOCSIZE | 650 | 1000 | 2000 |
| DATASIZE | 650 | 1250 | 2000 |
| LABSIZE | 600 | 600 | 1000 |

Fig. A.19.1 Workspace allocations

## A.20 Other Data Structures

Most of the variables initialised by MAIN are related to the remaining data structures used by the code generator. These consist mainly of buffers, holding essential information about forward references, data items or relocation addresses.

These buffers are:

PROGBUFF length=PROGSIZE words. Holds a packed buffer of all code compiled so far. PROGBUFFP is a byte pointer, marking the high water mark of the compiled code.

RELOCBUFF length=RELOCSIZE words. Holds a buffer of all addresses in the current compiled section which require to be relocated by the linker. RELOCBUFFP is a word pointer to the high water mark.

DATAV length=DATASIZE words. Holds a two-word buffer of data items. The first word is the 'type' of them item, the second, a numerical descriptor of the first. Types can be DATALAB, ITEML or ITEMN, and correspond directly to the OCODE statements. All data is buffered until the end of the program. DATAP holds a word pointer to the high water mark.

LABV length=LABSIZE words. Holds a word buffer of all translator and code generator 'generated labels'. Entries have the following meaning: Zero means an undefined label; Negative means Label already defined - negate to get the address; Positive means a pointer to a chain of label references in the PROGBUFF. The chain is terminated by a zero.

Fig. A.20.1 gives a diagrammatic representation of the work vectors, along with their associated pointers, showing inter-relations between them.

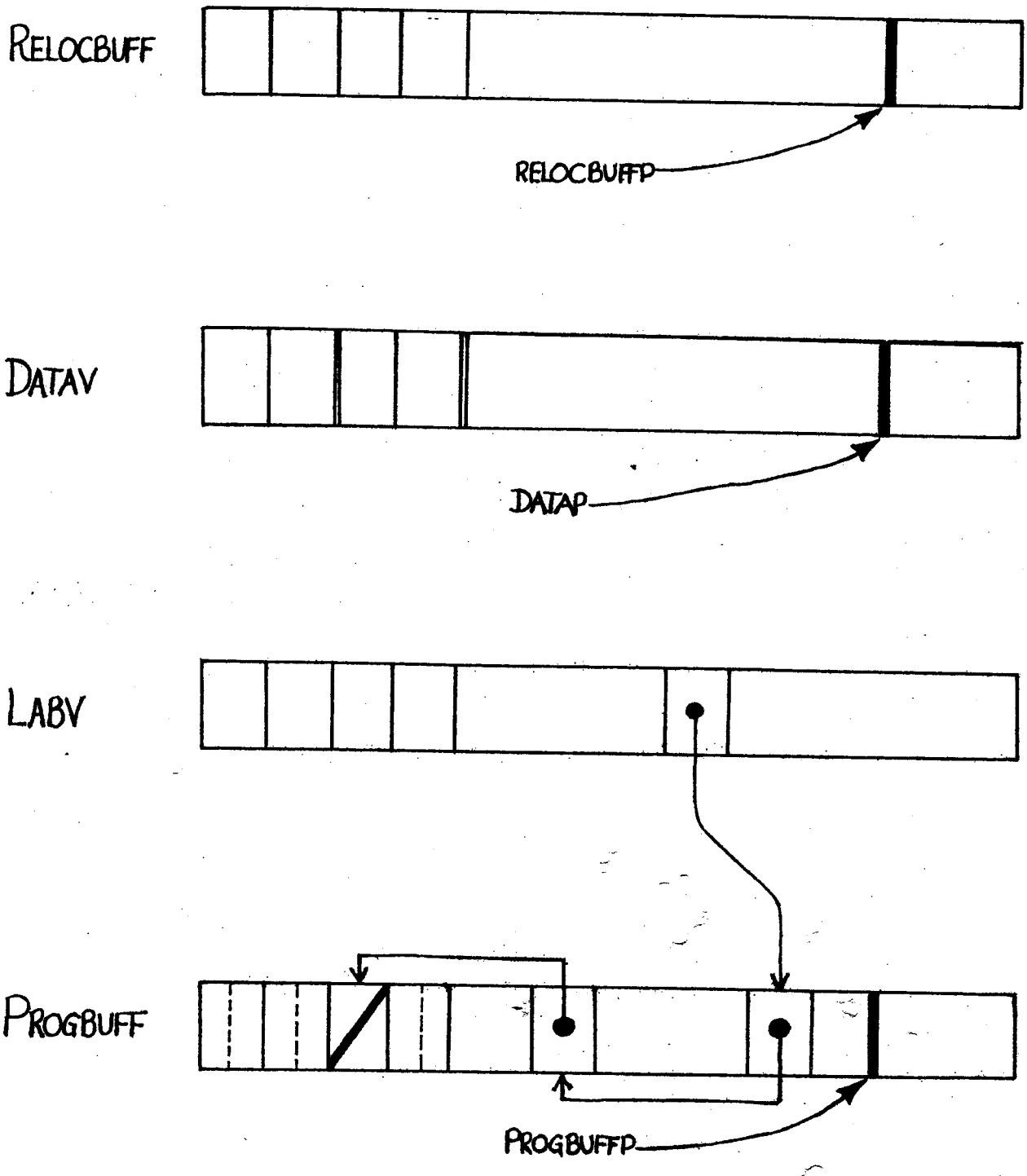



Fig. A.20.1 Buffer Structures

## Selected Code Generation Examples

### A.21 Introduction

Given the shortage of available space in this dissertation, it would neither be practical, nor particularly informative, just to treat each OCODE statement in turn, and discuss the code which it compiles into. I therefore propose to treat in more depth the overall strategy of the generated code, and to illustrate this with examples of specific constructs which proved especially difficult or interesting.

### A.22 The Inadequacy of the Instruction Set

The Z80's instruction set is particularly badly suited for the code generation of high level languages for the following main reasons.

Firstly, it is enormous - over 150 instructions in all. It would be impossible in a code generator which must fit on to the target machine, to exploit to its full advantage this amazingly diverse instruction set.

Secondly, the Z80 is essentially an 8 bit processor, which means that, on average, two instructions must to be compiled to deal with the the wordsize of 16 bits. An example of this is the code required to load a register from a directly addressable item on the stack, or in the Global Vector. This requires two 8 bit LD (load) instructions, each three bytes long.

Thirdly, the Z80 possesses only two index registers, the IX and IY registers, which are only addressable to $\pm$ 128 bytes from their current base. This means that, for maximum addressability, these registers must point 128 bytes (64 words) above the area they are needed, i.e. the current stack frame, or base of the global vector. Even this provision will mean that Globals over 127 and stack frames with large vectors, will contain variables not

directly addressable from the index registers, and special provision will have to be made for these items. The way round this addressability problem is to use a call to the run time system. The routines which deal with local and global variables which are out of range are $LIX, $LIY, $SIX and $SIY, each loading or storing, indexed off the IX and IY registers respectively. The item loaded or stored is always passed on the ZSTACK, as with other calls to the run time system.

Fourthly, the instruction set suffers from being extremely unorthogonal, with each register being different, and specific for particular instructions. This means that much time is spent in shuffling registers, to ensure that the right things are in the right registers at the right time.

## A.23 Initial Optimisation Decisions

There are two main areas where optimisation is possible in any programming language. These are flow of control, and data access. OCODE is especially badly designed for flow of control optimisation, for the simple reason that by the time the code generator is reached, all information as to the flow of control has been removed completely by the Translator. On the second count, OCODE scores much better, and there is much scope for for simulated stack manipulation and register slaving.

It was because of this, and the essentially incompatible nature of Z80 machine code, that most of the compiler's optimisation effort goes into minimising the number of memory accesses and optimising the code compiled for various operators. Very little time is spent in optimising jumps, and other transfers of control, for example, the chance of replacing some of the long JP (jump absolute) instructions by the shorter JR (jump relative). The amount of effort needed to drive such an optimisation is much larger than the benefits gained in space saving on including it. The NET size of the compiled code generator increases dramatically when this optimisation is included, and hence it is an advantage to omit it.

## A.24 Housekeeping of the Slaves

To aid adaptations, and further additions to the code generator, most of the work needed to maintain the simulated stack and various slaves is done by the various underlying code generator primitive routines discussed earlier. Optimisation of stack and global access stem directly from using these facilities sensible, and most importantly, only at the last possible opportunity given by the OCODE. If the handling of operators is done efficiently, then the rest of the optimisation comes as a by-product. It has already been shown that OCODE statements such as 'LG 100' do not cause any code to be compiled, but merely alterations in the state of the simulated stack. The item (GLOB,100) will not be accessed until absolutely necessary, either because it is needed as one of the operands for an operator, or because the state of the stack needs to be standardised for a change in the flow of control.

Very little can be done to optimise the stack standardisations, as all information about the flow of control has been lost by this stage. Operators on the other hand, can be dealt with much less rigidly, and hence, there is much greater scope for optimisation in this area.

## A.25 Generation of Code for Operators

The main prospect for the optimisation of code for operators comes from the possibility of delay tactics, putting off the compilation of any code at all until the last possible moment. This introduces the concept of a PENDINGOP - an operator which has been read from the OCODE stream, but which has not yet had code compiled to deal with it.

Thus the OCODE sequence:

```
LN 100  LG 150  EQ
```

will cause the items (NUMB,100) and (GLOB,150) to be LOADed onto the simulated stack, and the PENDINGOP to be set to 'EQ'.

If the next OCODE sequence were:

```
LP 5  LOGAND
```

then, since the operator LOGAND requires to work on the item (LOC,5) and the result of the previous operator, then there is no choice but to compile code for the previous operator before the item (LOC,5) can be loaded on to the simulated stack. What in fact would happen, is that the code for 'EQ' is compiled, (LOC,5) is loaded on to the simulated stack, and the PENDINGOP is set to 'LOGAND'.

If, on the other hand, the next OCODE sequence were:

```
JF L25
```

then this presents an ideal opportunity to produce optimal code for a conditional jump, by compiling instructions which set the Z80's condition codes, and then utilising the 'JP cc,address' instructions.

In this case, the optimal code is to subtract 100 from G150, and to jump on a non-zero (NZ) result. Assuming that HL contains G150 and DE contains 100, the code compiled is:

```
AND  A       ; clear the 'C'arry flag
SBC  HL,DE   ; subtract G150 and 100
JP   NZ,L25  ; jump on NZ result to L25
```

a piece of code which not only runs efficiently, but is only 6 bytes in length. Compare this to the code produced if the operator were not held in a pending state:

```
PUSH HL      ; ARG1 onto the stack
PUSH DE      ; ARG2  "    "    "
CALL $EQ     ; leaves T or F on the stack
POP  BC      ; answer into BC
LD   A,B     ; test to see if BC=false
OR   C       ; sets 'Z' flag if BC=0
JP   Z,L25   ; jump on Z result to L25
```

which not only runs slower, but is approximately twice as big - 11 bytes in fact.

## A.26 The routine CGPENDINGOP

CGPENDINGOP, as its name suggests, generates code to deal with the current PENDINGOP. It is called whenever

a) The stack requires to be in a standard state
b) The result of the operation is required
c) Another operator is read in

It is inside CGPENDINGOP that all shuffling of

operands, or calculation of purely numerical quantities is done.

All possible numeric calculations are done within CGPENDINGOP, so BCPL statements like:

```
LET A = 100 * 200
```

will cause the number 20000 to be assigned to 'A', and not compile code to multiply 100 and 20. This may seem an unlikely thing to optimise, because any normal user would actually assign 20000 to 'A' in the first place. A more likely occurrence where numeric calculations are needed is where MANIFEST variables are used. These are turned into 'numbers' by the Translator, so the code generator sees no difference between the two.

There are, however, operators which must always have code compiled for them, even if the operands are both numeric. These are operators which deal with essentially run time quantities, such as RV (!) and GETBYTE (%).

Taking the more complicated example of operators with variable operands, CGPENDINGOP sorts the operators into three categories:

(1) In-Line operators. Code is compiled for these operators directly. They correspond to the operators for which the Z80 has 16 bit instructions, i.e. Add and Subtract. All code is compiled by CGINLINEOP.

```
e.g.   ADD  HL,DE
```

(2) Logical Operators. All operations go via the 'A' register. These operators correspond to those for which the Z80 has 8 bit instructions, i.e. AND, OR and XOR. All code is compiled by CGLOGOP.

```
e.g.   LD  A,H
       OR  D
       LD  H,A
       LD  A,L
       OR  E
       LD  L,A
```

(3) "ARITH" operators. These are the operators for which no comparable instructions exist, and so must be implemented by means of calls to the Run Time System. All

operands are passed on the Z80's stack, which is where the result is placed. i.e. MULT, DIV, REM and the relational operators. All code is compiled by CGARITHOP.

```
e.g.  PUSH  HL
      PUSH  DE
      CALL  $REM
      POP   HL
```

Certain obvious chances of optimisation for the In-Line operators are taken. Examples of this are:

<item> ± 0

is optimised out. This may seem a trivial optimisation, but the structure of OCODE is such that the BCPL expression 'V!0' is translated as:

```
Lx [V]  LN 0  PLUS  RV
```

and access to the zero'th element of a vector is far from rare! Despite its simplicity, it is remarkable how few code generators take the opportunity to include this optimisation.

Another example is:

<item> ± <small number>

which is converted into loading the item into a register, and then INCrementing or DECrementing to the required value. For <small number> lying between 1 and 3, there is quite a considerable space saving over the register to register adds or subtracts, and also, a register which would have been needed, is free for allocation elsewhere in the program.

## A.27 Relation of "ARITH" operators to the ZSTACK

Not only is the Z80's stack used to pass operands to the routines which deal with the ARITH operators, but also to return the result of the operation. This can be optimised quite heavily, providing that the Z80's stack is slaved in the same manner as the registers.

After compiling code for an ARITH operator, the item (STCK,SSP) is LOADed onto the simulated stack, and the ZSTACK slave is updated to hold the item (LOC,SSP,SSP). This means that if the ZSTACK item is needed in a register, then this is accomplished by POPping it from the Z80's stack, and discarding the item in the ZSTACK slave. If, on the other hand, the item is required as on operand for another ARITH operator, then it turns out to have been already stacked, and so will not require stacking again.

Thus, a BCPL statement like:

```
A := B * C * D
```

will compile into:

```
LD  HL,[ B ]    ; Load item 'B' into HL
LD  DE,[ C ]    ; Load item 'C' into DE
PUSH HL         ; ARG1 onto the stack
PUSH DE         ; ARG2 onto the stack
CALL $MULT      ; leave B*C on the stack
LD  BC,[ D ]    ; Load iten 'D' into BC
PUSH BC         ; other operand onto stack
CALL $MULT      ; leave B*C*D on the stack.
```

Particular care has to be taken if the operator is non-symmetric, and in such a case, the result may have to be POPped, only to be PUSHed back a couple of instructions later, just because the operands were in the wrong order.

Since most of the more complicated operators are dealt with by the ARITH section of BCPLMAIN, the slaving of the ZSTACK is an extremely important part of the code optimisation strategy. Unfortunately, since the slave is only one item in size, it is impossible to optimise fully a BCPL statement like:

```
A := (B / C) * (D / E)
```

where, theoretically, it is possible to leave the result of (B/C) on the stack while calculating (D/E), and then both operands would be in the stacked state ready for the call to $MULT. After much consideration, it was decided not to bother with an arbitrarily sized ZSTACK, like the one required to cope properly with the above example, but to keep to the simple, single-item slave. In the example given, the intermediate result of (B/C) would be POPped into a register until (D/E) is calculated, and then PUSHed onto the stack again, ready for the multiplication.

## A.28 Conditional Jumps

It is in compiling code for conditional jumps that the full advantage of keeping a PENDINGOP can be seen. I have already given an example of the optimisation possible when an OCODE sequence like 'EQ JF L25' is compiled. EQ and NE are in fact extremely easy to generate conditional jumps for, due to the fact that a comparison for equality can never suffer from numerical overflow. Other relations, such as GR or LE suffer from this problem, and it is due to this that all these types of comparison are done via the routines in ARITH, where overflow is trapped and catered for.

Exceptions to this rule are comparisons with zero. These can never overflow, in fact, there are certain tricks which can be used to optimise such conditional jumps. All comparisons with zero (especially equality) are so frequent, that it turns out to be advantageous to optimise heavily these types of conditional jumps. Examples of the type of code compiled are:

To jump on HL = 0:

```
LD   A,H    ; high byte of item
OR   L      ; sets 'Z'ero flag if HL=0
JP   Z,Lxx  ; jump on zero to...
```

For HL \= 0 jump on Not Zero.

To jump on HL < 0:

```
XOR  A      ; clear A
XOR  H      ; sets 'M'inus flag if -ve
JP   M,Lxx  ; jump on minus to...
```

For HL >= 0 jump on 'P'ositive.

To jump on HL > 0:

```
XOR  A      ; clear A and reset 'C'arry
SBC  A,L    ; low byte, sets 'C' if carry
LD   A,0    ; clear A again, don't reset carry
SBC  A,H    ; sets M if HL > 0
JP   M,Lxx  ; jump on minus...
```

For HL <= 0 jump on 'P'ositive

All comparisons other than the ones illustrated here are done by calls to the run time system, and then comparing the result with zero to check for a false compare. Advantage is taken here of the fact that the relational operators can be reversed, and there is only a need for two routines to deal with them.

```
A <= B      is equivalent to      B >= A
A > B       is equivalent to       B < A
```

so, by just swapping the operands for the LE and GR cases, the two routines, $LS and $GE can deal with all four operators.

It must be stressed at this point, that relational assignments such as:

```
A  := B = C
```

are always compiled using calls to the run time system (in this case $EQ), because in cases such as this, it is the TRUTH values which a required, not condition codes.

## A.29 Procedure Applications

This introduces for the first time, a problem the solution to which must be rigidly adhered to, once decided upon, and as compact and efficient as possible: the procedure calling mechanism. There is a discussion of the mechanism, and machine code implementation of this problem in section B.4, so the description here is purely of the background to the problem, and the reasons for certain decisions.

Perhaps the most important aspect of the problem was that the solution must be efficient. BCPL is built very much on the philosophy of the modularity of programs, and undue overheads in this area would distinctly limit the programming capabilities of the implementation.

One piece of advice which turned out to be wise to follow, was to have as many of the parameters to the procedure as possible passed in machine registers, and if the procedure returns a result, then this should be passed in the register corresponding to the first parameter of the procedure call. It was decided to pass the first three

parameters to procedures in the HL, DE and BC registers respectively, with the returned result from functions being passed in the HL register.

Another reason for choosing the registers in this manner was that the implementations of FORTRAN running under CP/M and CDOS, both use this mechanism for the passing of parameters. Since I hope soon to produce an interface package linking the two languages, it seemed sensible to keep as much as possible to the established mechanisms.

Having decided on using all three 16 bit registers for parameter passing, it was imperative to use the Z80's alternative register set to pass parameters required by the BCPL run time system.

The final standardisation of registers is:

Argument register set

```
HL   -   Argument one to the procedure
DE   -      "     two  "  "       "
BC   -      "    three "  "       "
```

System register set

```
HL   -   Address of routine to be applied
DE   -   Old BCPL stack pointer (IX)
BC   -   Increase in size of old stack frame
```

This means that the lead up to a typical procedure application will produce code like:

```
LD   HL,[ Arg1 ]    ; first argument
LD   DE,[ Arg2 ]    ; second...
LD   BC,[ Arg3 ]    ; third...
;
; Note - arguments above 3 are already stacked
;
EXX                 ; change register sets
LD   BC,[ ssi  ]    ; ssi = stack size increase
LD   HL,[ @rt  ]    ; address of the routine
CALL $APPLY         ; apply the routine.
```

The old stack pointer is loaded into DE inside $APPLY, so that by the time the code for the routine is entered, the registers are as previously defined.

Return from procedures is equally simple. If a result is to be passed back to the calling program, then this is loaded into the HL register. The actual return is via a routine in the run time system - $RETN, which resets the

previous stack frame, and jumps to the return address.

## A.30 The code generation of SWITCHON commands

By far the most complex OCODE statement is the SWITCHON. The frontend causes very little change to the structure of this command from the original source code, leaving the code generator great freedom to choose the best way of compiling this construct.

There are two main objectives when compiling code for this statement. Firstly, the code must be efficient, as most large programs rely heavily on SWITCHONs for their flow of control. Secondly, the code must be compact, to make the use of SWITCHONs advantageous over the multiple 'TEST... THEN... ELSE...' construction.

It does not take much time to realise that it is impossible to have a single code generation strategy which is optimal for all possible set of CASEs. A method which can cope efficiently for a SWITCHON with a very small range of CASE constants will be disastrously inefficient on space for occasions with a wider spread. Conversely, code which is compact for a wide spread of CASE constants, will be unnecessarily inefficient in execution time when presented with a much smaller range.

It is for this reason that there are two methods used, the one chosen depending on the spread of the CASE constants. As an essential part of both systems, the cases are sorted as they are read in, so as to aid the setting up of address look-up tables if needed, or to help with searching of the data.

The two methods are widely used in most BCPL systems, and the main algorithms have been taken from those used many times before. These are the LABVECSWITCH (a label vector look-up) and the BINTREESWITCH (a binary chop search, followed by linear look-up).

Given a SWITCHON with range of cases (difference between the maximum case and the minumum case) R, and number of cases N, the following comparison is made:

R * 2 + 20 <----> N * 4 + N / 2

If the left hand side is the lesser, then the LABVECSWITCH is taken, otherwise, the BINTREESWITCH. The two

calculations are estimates of the number of bytes each of the two methods would compile into, given the current SWITCHON to compile. It turns out that if the label vector would be more than half full, then it is advantageous to take the LABVECSWITCH.

The algorithms for the two systems are as follows.

For the LABVECSWITCH, the SWITCHON item is compared, first with the maximum case, and then with the minimum case, jumping to the DEFAULT label if it is greater than the first or less than the second. Then the address to jump to is picked up from LABVEC!(CASE-MINCASE), and the jump is executed.

For the BINTREESWITCH, the cases are repeatedly split in half recursively until a section of <= 8 cases remain. These are then put into a table, along with the corresponding addresses to jump to, and the label is searched linearly by a routine in the run time system - $LINSCH.

## Summary

Given more time and space, it would have been possible to go into greater detail on the aspects of the code generator which I have covered, and also deal with others which I have had to omit altogether.

I have tried to include the areas of code generation which were particularly interesting or difficult, but also concentrate mainly on those which anybody modifying or enhancing the code generator would need to know about.

# Section B

# The Z80 BCPL Run Time System

## B.1 Introduction

In this section, the run time system is discussed as a separate suite of programs, each in isolation, with very little reference to the rest of the BCPL implementation.

The main part of the run time system comprises two modules - BCPLMAIN, the machine code library, and BLIB, the machine independent I/O library, built on top of BCPLMAIN. Since BLIB is written entirely in BCPL, and is essentially machine independent, the version used in this implementation is heavily adapted from the standard BLIB provided with the BCPLKIT, and so will not be covered here.

## B.2 BCPLMAIN - The machine code library

BCPLMAIN splits very conveniently into three parts:

a) $MAIN$ initialisation section
b) ARITH arithmetic section
c) Standard BCPL procedures

These are essentially separate sections, which tend to inter-communicate very little. Communication with the outside world is different for each of the sections.

$MAIN$ is entered right at the beginning of a BCPL run, i.e. at location 0100H, the base of CP/M's transient program area. The initialisation process is irreversible, and this part of the program is never returned to after START has been entered.

ARITH is entered via a jump table, starting at 0100H+3. This corresponds to the manifest JUMPTABLE in "CGHDR". Within BCPLMAIN, calls are made directly to the routines concerned, rather then via this jump table. An exception to this is $APPLY, which may go via location 0103H, if it is

wished that the call should go through DEBUG. There is more about the interactive debugging facility in section B.7 and Appendix IV.

The remaining part of BCPLMAIN is made up of routines with BCPL routine/function linkage conventions. This is the machine code library, consisting mainly of routines which implement the BCPL I/O facilities.

The rough memory layout of an initialised BCPL program at run time is given in Fig. B.2.1, along with relevant globals or symbolic equates.

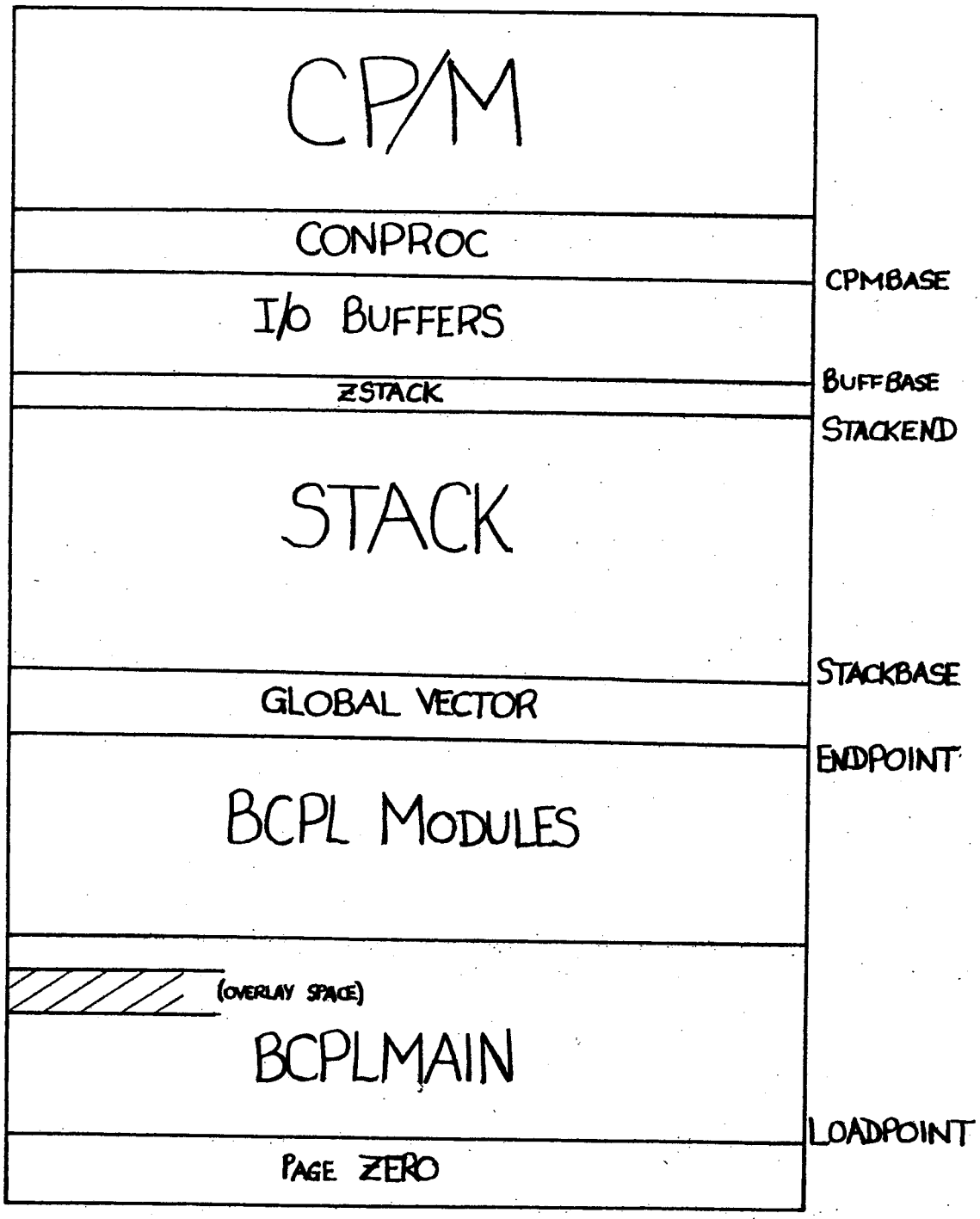



Fig. B.2.1 Run time memory layout

On smaller systems, the CONPROC (CP/M console processor) can be overlaid, giving another 2K bytes of workspace to the run time system. In such a case, the CONPROC is rebooted by returning to CP/M via a 'JP 0' rather than a direct return. The symbolic equate REBOOT governs whether the CONPROC is rebooted.

## B.3 $MAIN$ - The initialisation stage

The main task of $MAIN$ is to set up the various structures used by the rest of the BCPL run time system. These include the Stack, Global Vector and I/O buffers. To do this, the BCPL modules must be scanned to pick up such information as the first address beyond the compiled code which can be used as workspace, and the addresses of all global routines which need to be initialised in the global vector. In fact, it requires two scans to accomplish this. The first finds the end address of the compiled code, and the maximum global number allocated in the routines scanned. This is needed to determine the absolute size of the global vector. On their second scan, since the global vector is now set up, all updates to it can be done.

Each scan is accomplished by picking up the end address of BCPLMAIN (ENDMAIN), and working backwards from this address in two-word steps until the '<maxgn>,0' pair is found. This end address is then treated as a possible BEGIN address for the next module loaded, if it is present. If it is, then the first four bytes will be the characters "BCPL". This being so, the length of this new module is picked up from offsets 4 and 5, and this is added to the begin address, to give the new end address of this module. The whole process is then repeated until the end of the BCPL modules is reached.

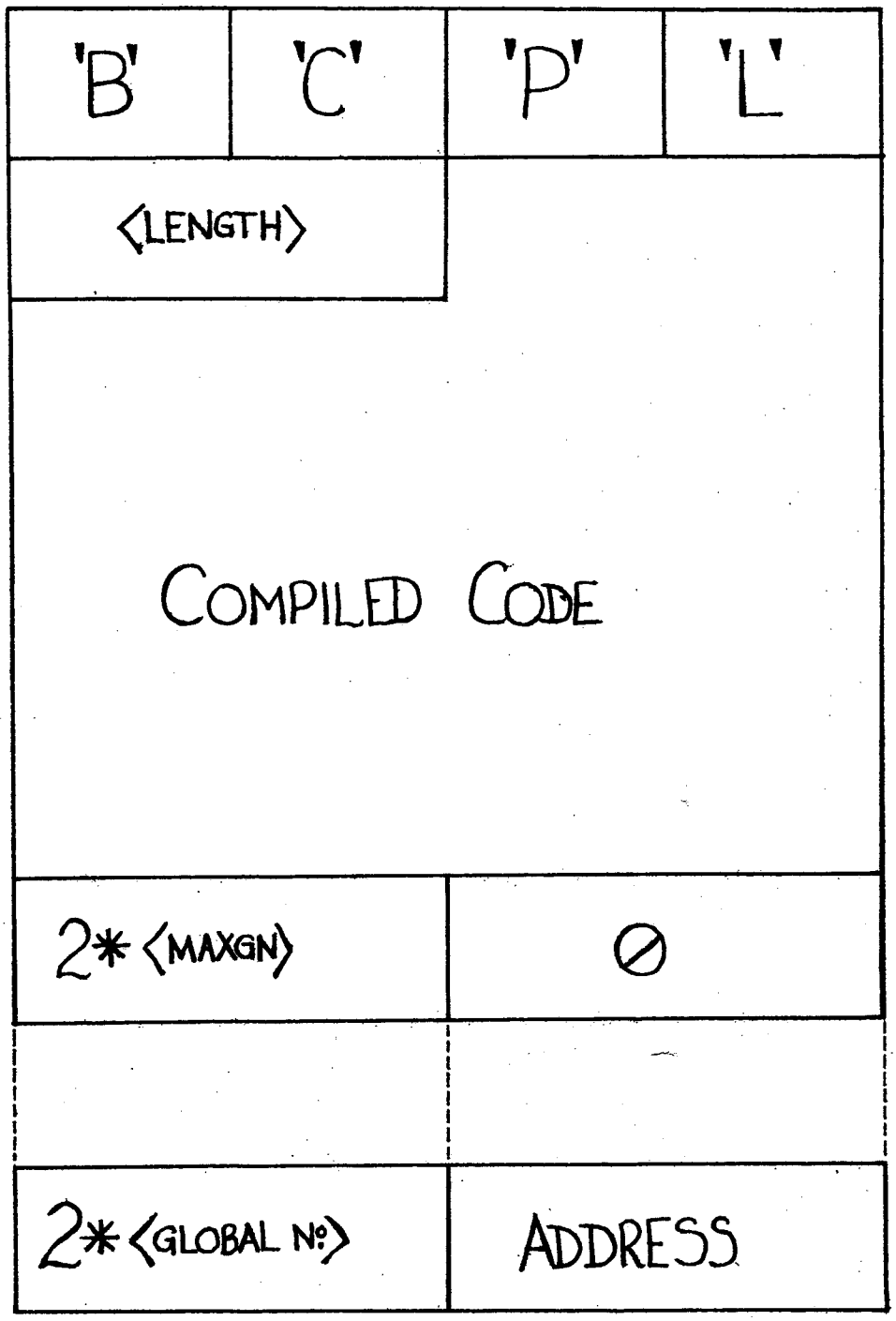



Fig. B.3.1 The layout of a BCPL module

## B.4 ARITH - The Arithmetic Library

Due to the lack of certain essential instructions on the Z80, e.g. 16 bit compares and Multiply/Divide, these operations have to be implemented in software by calls to this section of the run time system. Here I am indebted to the writers of the Z80 LISP system for providing the algorithms for implementing software multiply and divide.

ARITH does not only deal with the complex operators, but also with facilities to aid mechanisms such as procedure calling, table searching, or out of range indirect data access. The procedure calling mechanism is worth examining in detail, because it was specifically designed to be fast, compact, and also easy to bipass for anybody writing machine code libraries.

The arguments are passed in either machine registers (Args 1,2 and 3), or already stacked (Args 4 upwards). Assuming that the arguments have been set up in advance, the following code would be produced to apply the procedure (in this case WRITEF, G76).

```
EXX                 ; swap to alt reg set
LD   L,(IY+24)      ; low byte of G76
LD   H,(IY+25)      ; ...and high byte
LD   BC,20          ; increase in stack size
CALL $APPLY         ; apply WRITEF
.... ......         ; return here.
```

So, on entry to $APPLY, the alternative register set holds the arguments to the procedure, and in the current register set, HL holds the address of WRITEF, BC holds the increase in stacksize. DE is used later on to hold the value of the old stack pointer.

$APPLY shuffles the registers, ready for entry to the procedure.

```
PUSH  IX     ; save old stack pointer
POP   DE     ; ...into DE
EX  (SP),HL  ; swap calling & return addresses
RET          ; and jump to the procedure.
```

On entry to the procedure, HL contains the return address, DE contains the old stack pointer, and BC, as before, the increase in stacksize. One of two things now happens. If the procedure being applied is written in BCPL, then there is no choice but to set up a new stack frame, with all the overheads this entails. If, on the other hand

the procedure is a library routine written in machine code, which does not call any other routines, then the entire overhead of setting up a new stack frame can be avoided. The simplified mechanism for the entry point to a machine code routine is:

```
PUSH   HL      ; save the return address
EXX            ; get argument register set
  ...          ;
  ...          ; code of the routine
  ...          ;
RET            ; return
```

Given that a new stack frame has to be set up, the very first instruction to be executed on entry to a procedure is:

```
CALL $SETL
```

which sets up all linkage information, ready for stack accesses relative to the new stack pointer, or another procedure call. $SETL stacks the old stack pointer and return address as the first two words of the new stack frame, and returns with the argument register set restored.

```
ADD    IX,BC          ; IX +:= increase in stacksize
LD     (IX-128),E     ; low byte of old IX
LD     (IX-127),D     ; ...and high byte
LD     (IX-126),L     ; low byte of return address
LD     (IX-125),H     ; ...and high byte
EXX                   ; get argument register set
RET                   ; and return
```

Return is accomplished via the run time system call

```
JP $RETN
```

which reverses the process of $SETL, restoring the old stack pointer and jumping to the link address.

```
LD     E,(IX-128)     ; low byte of old IX
LD     D,(IX-127)     ; ...and high byte
LD     C,(IX-126)     ; low byte of return address
LD     B,(IX-125)     ; ...and high byte
PUSH   BC             ; stack return address
PUSH   DE             ; stack old IX
POP    IX             ; restore IX
RET                   ; return to link address
```

Note here that the HL register pair cannot be used as a work register, because it may contain the returned value from a function.

## B.5 BCPL I/O Library

The most basic form of BCPL I/O library must provide the following routines:

```
RDCH  FINDINPUT  SELECTINPUT  ENDREAD  INPUT  and
WRCH  FINDOUTPUT SELECTOUTPUT ENDWRITE OUTPUT
```

The specification if each of these routines is identical to the 370 implementation, with one exception. That is the fact that FINDINPUT and FINDOUTPUT must take File Names rather that DDnames. Decoding of the filenames is such as to allow any name of the form:

```
[ <drive name>: ] <file name> [ .<extension> ]
```

so a call like:

```
X := FINDOUTPUT( "B:GARBAGE.IDW" )
```

would open the file GARBAGE.IDW on drive B, creating it if necessary. As well as the standard I/O routines, there are five others provided. These are:

```
UNRDCH  FINDFILE  BINRDCH
BINWRCH WRITETOLOG
```

As in the 370 implementation, UNRDCH will backspace only one character. The reason for this is that, to avoid anomalies over characters like '*N', the last character returned by RDCH is stored, and UNRDCH sets a bit in the status byte of the SCB, causing the next call of RDCH to return the saved character rather than a new one. UNRDCH from the terminal is a no-op, because all terminal access is direct, and not buffered.

FINDFILE is a routine which takes advantage of the CP/M file formatting facility. On typing a command to the CLI like:

```
<transient command> <filename>
```

the transient command is taken to be the first eight characters of a file with the extension '.COM', and this is loaded at 0100H, the base of the transient program area. The filename is formatted into a file control block at 005CH, ready to be used by the transient program. FINDFILE exploits this, and opens a stream to this file. Whether the stream is opened for input or output depends on the argument to FINDFILE. Similar to the 370's 'FINDTERMINAL', a zero argument will open the stream for OUTPUT, and a non zero

(usually 1) argument for INPUT.

BINRDCH and BINWRCH work in just the same manner as RDCH and WRCH, but no character translation is performed.

WRITETOLOG writes a string to the console without requiring BLIB to be present, i.e. without using WRITES. This is useful for printing out error messages if BLIB is not present, or just for sending some message to the console without having to de-select the current output stream.

All I/O transfers to disc are done via the BCPL Stream Control Block, (SCB), and extension to the CP/M File Control Block, (FCB). Each SCB is 164 bytes in length, and the number of such available blocks in any configuration of BCPLMAIN is given by the symbolic equate NBUFFS. The normal value is 4. At any one time, CP/M has a currently selected 'DMA' address, which is a pointer to a 128 byte area of store, used to buffer disc sector transfers. On each disc read or write, the DMA has to be re-selected, the address being set to SCB+35, i.e. the beginning of the 128 byte buffer area of the stream control block.

The CP/M file control block is 33 bytes long, and comprises the drive number, file name, current extent, current sector number, and a block allocation bit map. The BCPL stream control block has three bytes added onto the front of the FCB - a status byte, a pointer into the character buffer and an 'unread' or 'unwrite' character - and a 128 byte character buffer added to the end. This is shown diagrammatically in Fig. B.5.1.

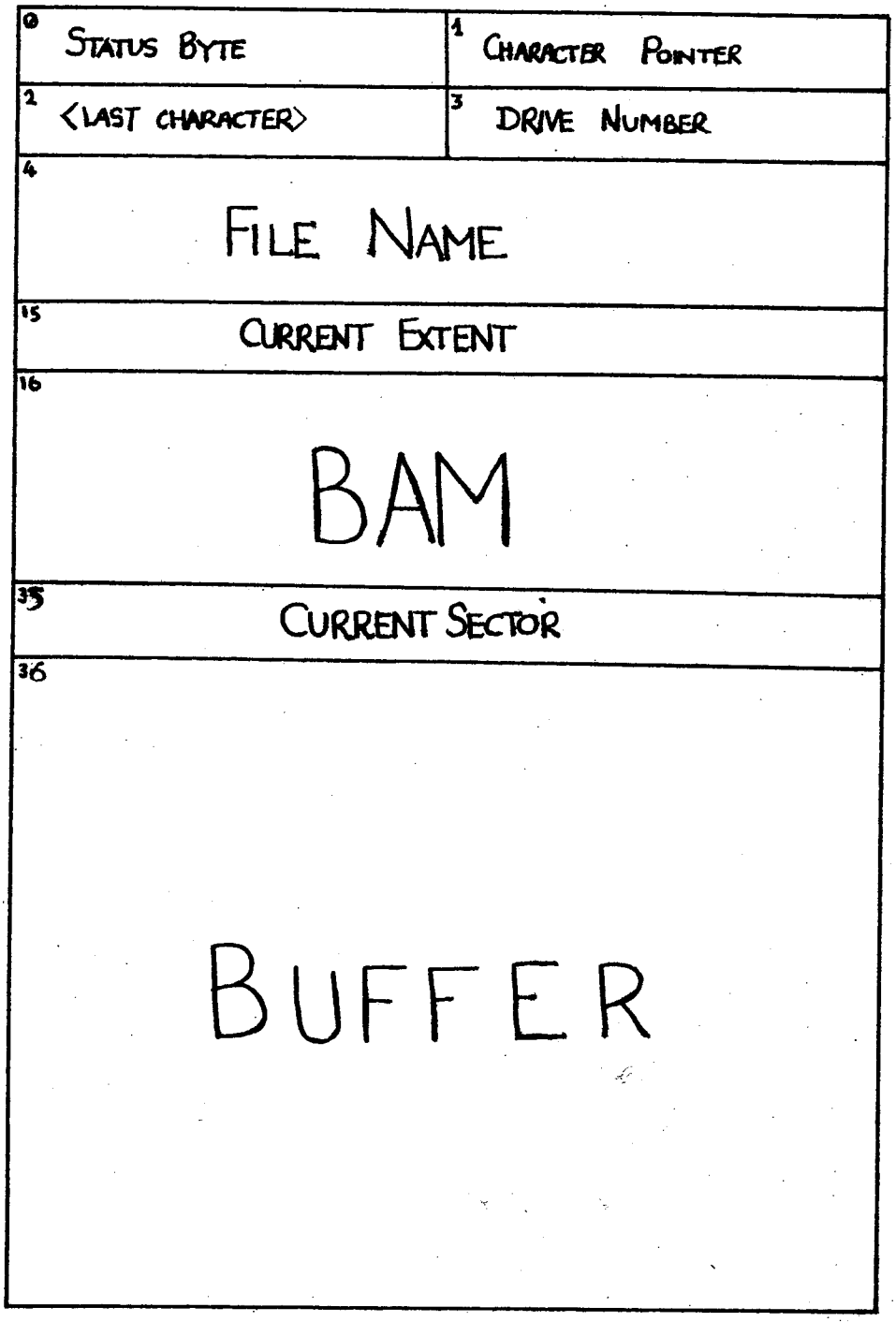



Fig. B.5.1 The BCPL Stream Control Block

## B.6 Overlaying of BCPL modules

For the frontend of the compiler to be implemented, the Translator must be overlaid on top of the Syntax Analyser. Since CP/M has no primitives for overlaying, then the mechanism designed had to be integral to BCPLMAIN.

Overlaying if implemented by leaving a 'gap' of OVLBLKS*128 bytes in BCPLMAIN, defined as a storage area at assembly time. Specifying the overlay space to be a specific number of disc blocks makes the check for overflow much easier, and since the minimum quantity of information which can be brought from disc at any one time is 128 bytes, unless each transfer is going to be buffered somewhere before being overlaid, an integral number of sectors is essential.

Once loaded into this region, (ORG'ed at 0D00H), the second pass of the initialisation stage of $MAIN$ is called again to initialise the new global variables brought in with this overlay. There is no attempt to de-initialise the globals which where set up by the previous overlay.

Only the function LOAD is provided, since, because there is no internal space management, one module is effectively UNLOADed as soon as another one is overlaid on top.

Overlay modules are produced by LINKing the object modules for the overlay section, setting the starting link address to be 0D00H. Because the CP/M Loader requires its HEX file to be ORG'ed at 0100H, another utility had to be provided. OVERLAY.COM takes a file in HEX format, and produces an equivalent OVL module, which is absolute binary, as with the COM files, but ORG'ed at the overlay position. Examples of the overlaying procedure can be found in the notes on bootstrapping the compiler, in Appendix II.

Overlaying tends to be slow on a floppy disc system, due to the hopeless inefficiency of the disc block allocation, and the very slow rotation speeds concerned, (about 6 revs per second). Typically, the time taken to overlay about the 12K bytes of the Syntax analyser section of the compiler frontend is approximately 10 seconds on the Horizon (5" floppy discs), 6 seconds on a Cromemco System C/3 (8" floppy discs), and about half a second on the Cromemco System Z2/H (Winchester Hard Disc).

## B.7 DEBUG - An Interactive Debugger

Given the problems of running an unchecked language on a non-protected machine, there is a need for a run time debugging system. Machine code debuggers (such as DDT or Cromemco's TRACE) are unsatisfactory because of the very small correspondence between the compiled machine code, and the source BCPL.

The basic design of DEBUG was taken from the debuggers running under TRIPOS and RSX. As yet, the Z80 debugger is not as sophisticated as these two, but but still provides a powerful for debugging runaway BCPL programs.

DEBUG is entered automatically if it is linked with the other BCPL modules. A break point is set up, and DEBUG re-enters itself for servicing the first break point. After this, DEBUG is entered whenever breakpoints are set, or an area of program is traced.

As well as being a debugger, DEBUG is also a calculator, working in decimal or hexadecimal arithmetic. The specification of DEBUG at the time of completion of this dissertation is given in Appendix IV.

## Summary

This section is not intended as a manual to aid the maintenance of the run time system, but more as an introduction to some of the ideas and design decisions which went into this area of the BCPL implementation. In most cases, I have tried to make the source code as self explanatory as possible, considering the generally incomprehensible nature of large machine code programs!

Given more time and space, much more of the detail could have been discussed, and full specifications of the library routines given. All that I can say here is that, as far as possible, I have tried to keep to the specification set out in the proposed BCPL standard, and in particular, the implementation was strongly influenced by Ken Moody's machine code library for the 370.

# Section C

# The Z80 BCPL Implementation

## C.1 Introduction

In this section, the BCPL implementation as a whole is discussed, along with the many problems associated with the bootstrap process, and testing of compiled code and run time system. This is in contrast to the previous two sections, which have looked at their subject matter in a somewhat blinkered manner, concentrating on the particular program concerned, and taking for granted the rest of the implementation.

## C.2 Choice of Interface Standards

The interfaces which needed to be standardised were:

```
BCPL    ----> "FRONTEND" ----> [ OCODE ]

OCODE   ----> "CODE GEN" ----> [ TARGET ]

TARGET  ---->  "LINKER"  ----> Absolute Binary
```

Thus the two standards which had to be decided upon were those which affected the input and output interfaces of the code generator, i.e. the OCODE type, and OBJECT module type.

Already available on the target system were:

An INTEL standard Linker (OBJ to HEX)
An INTEL standard Loader (HEX to COM)
An INTEL standard Z80 Assembler

Given these software tools, the decision to use the INTEL standard was somewhat predictable.

A definition of this standard is given in Appendix I.

This left the decision of which OCODE format to use. There were three choices, each with their individual merits:

(1) Mnemonic OCODE. This is OCODE where all keywords and directives are in their mnemonic form.
e.g. STACK 2 JUMP L5 LAB L6 RTRN
This form is readable, and so could be easily debugged, but problems with it are, firstly that parsing the OCODE is somewhat more difficult than it needs to be, and secondly, the sheer size of an OCODE file is enough to make its use prohibitive on a floppy disc system.

(2) Numeric OCODE. This is similar to mnemonic OCODE, but the keywords and directives are given in their numeric form, i.e the value of the MANIFEST constant which represents them in the Translator. This has the advantage of being both compact, and relatively readable.
The example given in the previous section becomes
91 2 85 L5 90 L6 97

(3) Binary OCODE. This is one level of compaction further on from numeric OCODE. In this type, all numbers are held in binary, and the superfluous 'L's before label numbers are omitted. The result is the most compact form of OCODE possible, but one which is difficult to debug, and more important, difficult to transfer.

After due consideration, the one chosen was the Numeric OCODE. This had the benefits of being reasonably compact, and also of being in character form, making the job of transferring it between machines that much easier.

## C.3 Data Transfer

When the code generator was being debugged on the 370, the code produced had somehow to be transferred to the North Star Horizon for linking, and subsequent running under a debugger. The interface to PHOENIX is especially bad for the setting up of any but the very simplest of handshake protocols, due to the PDP11 which buffers all characters sent in either direction.

There are distinct problems in either sending data to, or receiving data from the PHOENIX interface, problems which have totally different solutions depending on the direction

of data travel! Transferring data from the 370 is relatively easy. All that needs to be done is to convince the PDP11 that the machine on the end of its line is just a dumb terminal. This was an easy thing to simulate using the Horizon, so it was possible to maintain the link to PHOENIX via a second serial port, while still keeping primary input from the master console. Working on the assumption that the files being transferred would be small ( <40K bytes ), it was feasible to buffer all the characters sent by PHOENIX into memory, and only file them onto the slower floppy discs when the entire transfer was completed.

The method for transferring data from the 370 to the Horizon was standardised to the following procedure.

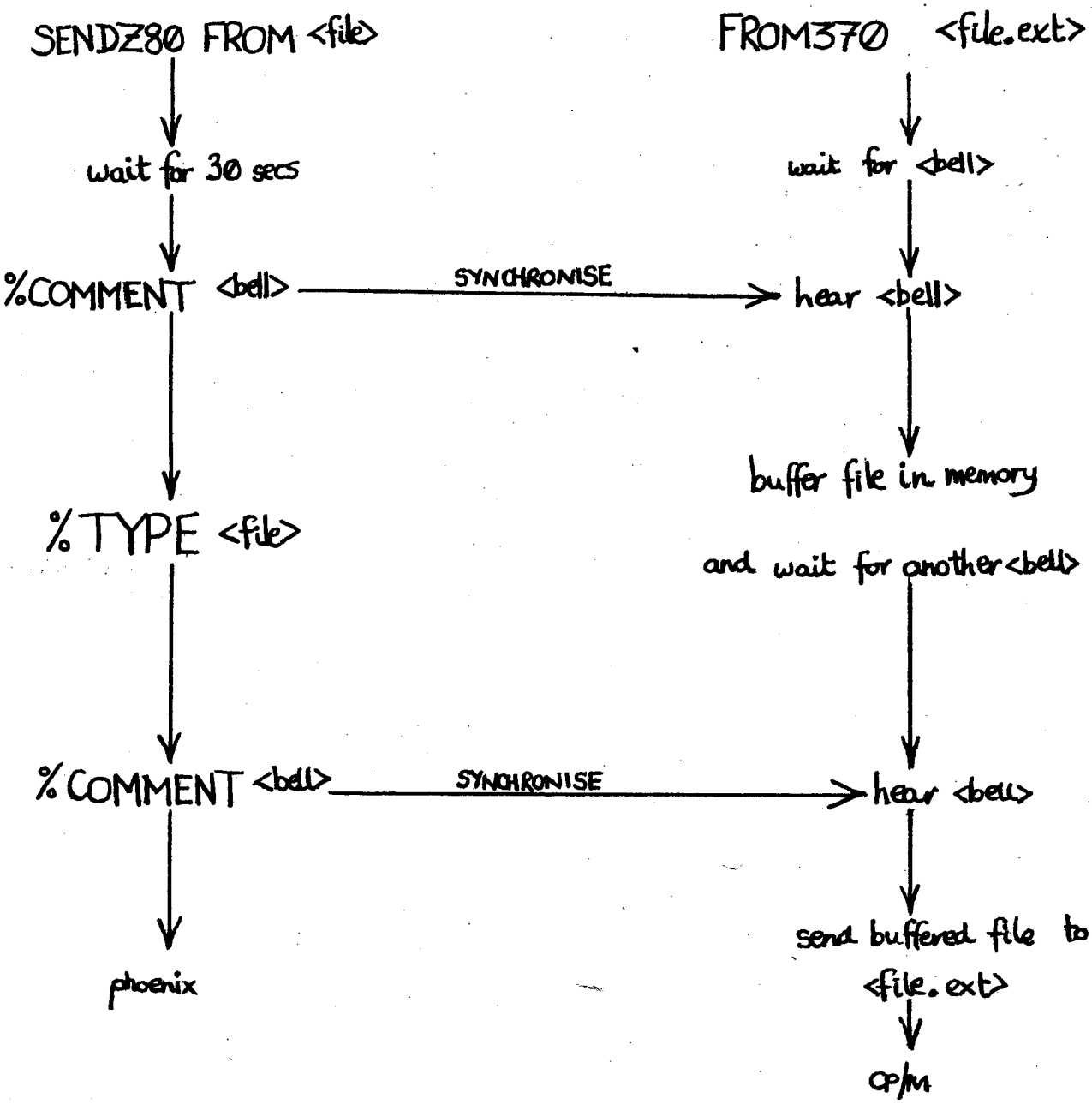



Fig. C.3.1 Transfer Protocols - 1
Phoenix to Horizon

Transferring files from the Horizon to the 370 is somewhat more complicated. This facility is needed much less than its simpler counterpart, but since the only way to obtain listings of files held primarily on the Horizon's discs is via the 370, there was no option but to try and make the best attempt possible.

Here, the onus is on the PDP11 to accept characters sent to it - a job which it does non too well if the character stream speed is close to that of the maximum speed of the line being used. Two cunning modifications can be made to the simple character stream method, both of which improve the reliablility of data transfer by a significant amount.

(1) On sending each character, wait for it to be reflected by the PDP11. On sending <cr>, wait for both the <cr> and the <lf> to be reflected.

(2) After sending each <cr>, wait for a short time - typically up to 1 second - before sending any more characters. This gives PHOENIX a chance to catch up with the file being sent.

Using these two simple devices, there is only one situation in which an error in data transfer can occur. This is after sending a <cr>, and not allowing quite enough time for PHOENIX to catch up. The result is that the first character of the next line is sent, but this is never reflected, because it has been missed altogether. This means that the Horizon goes into a tight loop, waiting for the character which it has just sent to be reflected, while PHOENIX hangs, waiting for another input character, not realising that one has been missed. This is fixed when the message '*** 5 MINUTE WARNING' is sent by the 370, breaking the deadlock, and causing the transfer to continue.

The protocol for transferring data TO the 370 is as follows.

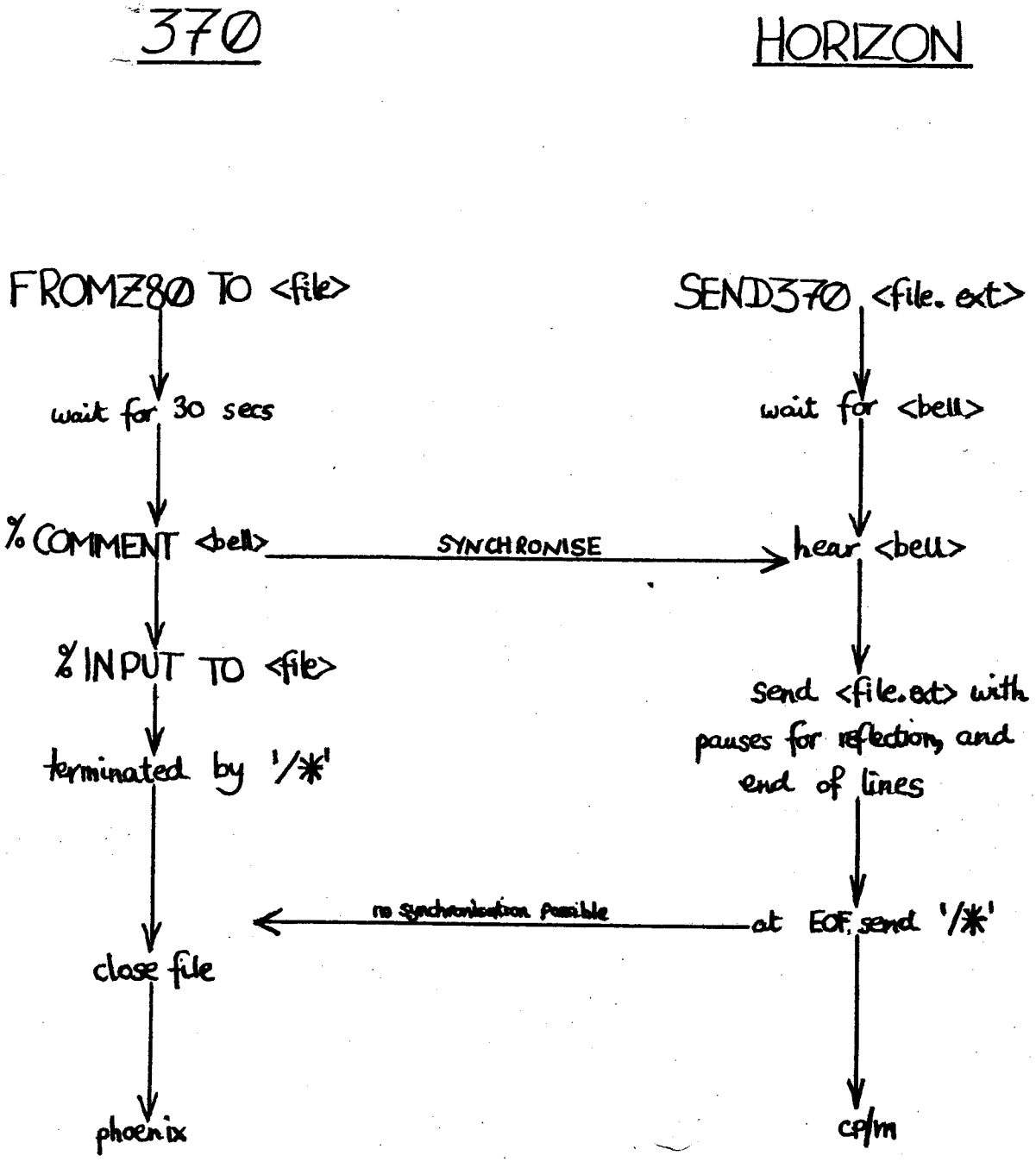



Fig. C.3.2 Transfer Protocols - 2
Horizon to Phoenix

So far, I have said nothing of how the commands come to be typed to each machine. The method used is a program which continuously loops, taking input only from the console (serial port #1), and selectively sending the typed commands to Phoenix (serial port #2), or the CLI internal to CP/M. This very simple program provides the Horizon with the facility of being an intelligent terminal for any computer using the RS232 serial interface standard. As is happens, the Horizon was occasionally connected as a 'TITAN VDU' on the ring, enabling the transfer of data from its discs to one of the LSI/4's or the RSX system.

I am indebted to the University Computing Service for the provision of a 1200 baud RS232 Phoenix line to aid the transfer of data between the two machines. Without it, the initial bootstrap process would have been painfully slow, and in no way so reliable or accurate.

The entire Horizon/Phoenix set-up is shown diagrammatically in Fig. C.3.3, along with the algorithm used in the 'intelligent terminal' utility.

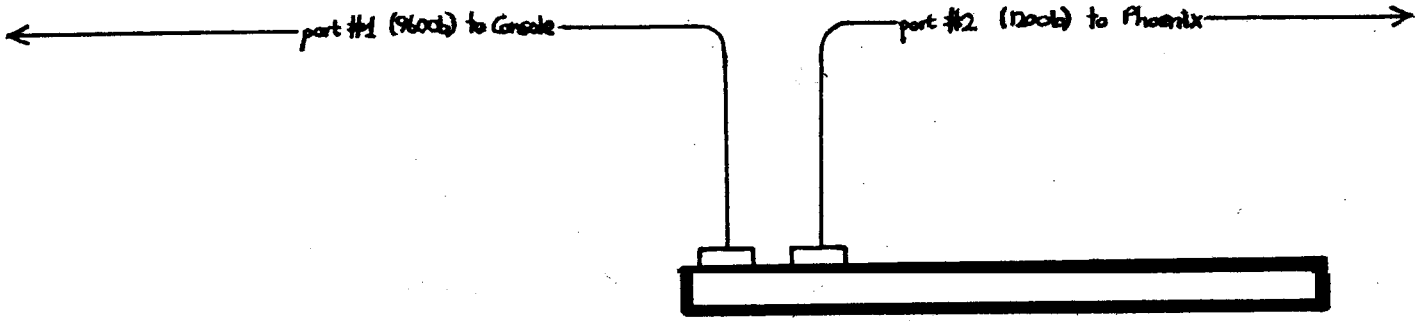
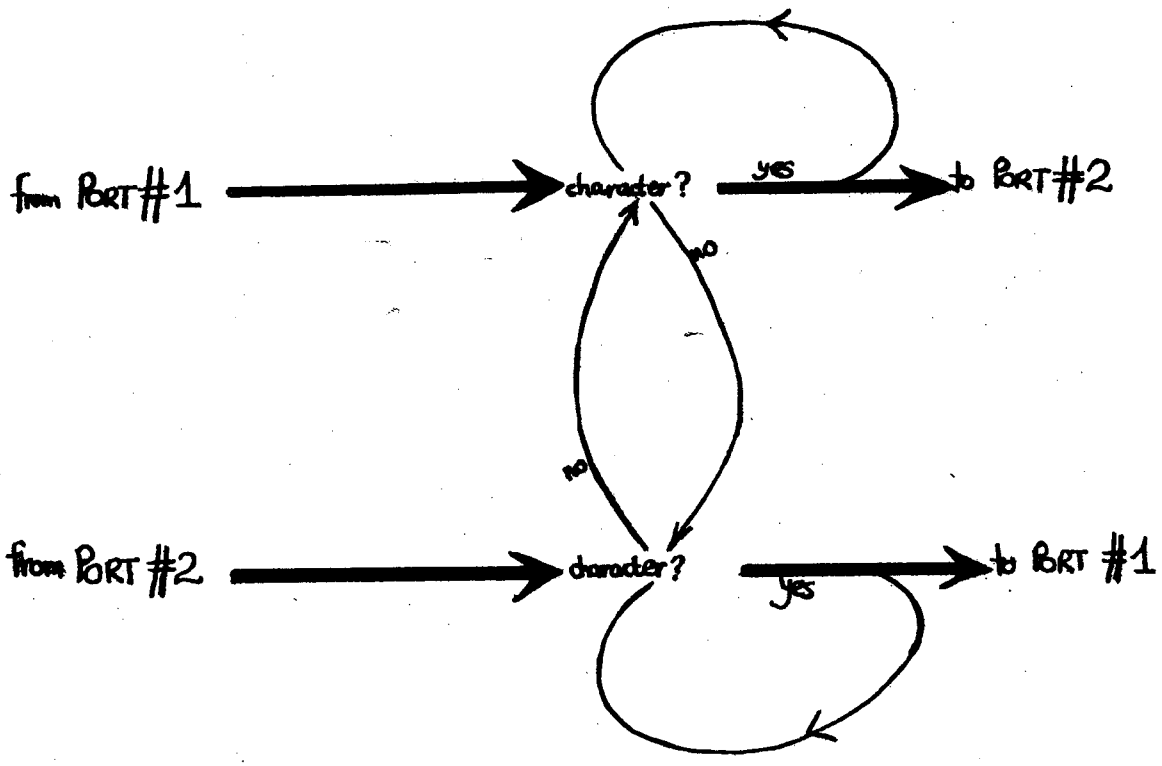



Fig. C.3.3a The data transfer set-up



## C.4 Initial Testing

With the utilities to transfer data written, it was possible to start testing in earnest, some of the code compiled by the code generator. At the very early stages, it was not at all clear where bugs were likely to show themselves. There were two possibilities. Firstly, they could be in the logic of the compiled code, in which case the cause would have to be located, corrected, and the relevant parts of the code generator re-compiled. Secondly, there were bound to be bugs in the newly written run time system, and once corrected, the whole of BCPLMAIN would need to be re-assembled.

Added to this problem, it turned out that the only interactive debugger available was DDT, CP/M's Dynamic Debugging Tool. Although a reasonably powerful program, DDT is written in, and caters for 8080 machine code, causing havoc when the extra instructions of the Z80 were used. CP/M itself is an 8080 system, and so all system utilities assume an 8080 instruction set, even though the majority of CP/M systems in the world are in fact running on Z80 processors.

The result of this was that DDT was of little or no use as a disassembler for the compiled code, because Z80 instructions (such as those using the IX or IY registers) are invalid 8080 opcodes, causing the disassembler to become a byte out of step with the actual instructions. Also, since the IX and IY registers, and the alternative register set, are purely Z80 additions to the 8080, it was impossible to ever inspect the contents of these registers, which made simple jobs, such as locating the global vector, much more complicated than they needed to be.

The initial form of the compiled code was that of an Assembly code listing - a form which is easily debugged by eye, but requires to be assembled before being ready to be run on the machine. The option to produce a listing of the compiled code was kept in the code generator, even after the routines to provide relocatable binary had been added. It was eventually removed as part of the 'code trimming' process, required before the code generator could be bootstrapped onto the Horizon.

The very first test programs were of the form:

```
GLOBAL   $( START : 1  ;  WRCH : 13   $)

LET WRITES ( S ) BE
    FOR  I  =  1 TO S % 0  DO  WRCH ( S % I )

AND START()  BE  WRITES( "Wake Up!" )
```

The run time system at this time was barely minimal. The only I/O facilities available were RDCH and WRCH working to the console, and a simple minded version of stop. This was in fact all that was needed to debug the simple OCODE constructs, and eventually bootstrap BLIB, to provide more comprehensive I/O facilities.

Bugs in the initialisation portion of BCPLMAIN were soon ironed out, leaving all debugging activities concentrated on the logic of the compiled code. A very useful tool for testing all possible OCODE constructs of BCPL is the utility code generator test program "CMPLTEST", written by Martin Richards to test an earlier BCPL implementation, but which has been used successfully to debug code generators ever since.

CMPLTEST always proved a useful tool to have around, even after the compiled code had been debugged enough to cause it to detect no errors. Every time an addition or modification was made to either the run time system, or the code generator, CMPLTEST was re-run, and many potentially elusive bugs were picked up at an early stage as a result of this. The source of CMPLTEST, and of other programs used to test specific aspects of the implementation are given in Appendix VII.

When the basic language constructs had been fully debugged, the rest of the run time system could be added.

These routines fell into two main categories:

a) Short simple routines, written in machine code by necessity or for efficiency. These include GETBYTE, PUTBYTE, INPUT, OUTPUT, LEVEL, LONGJUMP and APTOVEC.

b) The I/O section of the library - a proper RDCH and WRCH, along with FINDINPUT, FINDOUTPUT, SELECTINPUT, SELECTOUTPUT, ENDREAD and ENDWRITE. Also to be added later were UNRDCH, FINDFILE, BINRDCH and BINWRCH.

I have already described the format of the BCPL Stream Control Block, but the standardisation of this format was only a small part of the problems of designing and debugging a disc I/O system to run under CP/M. The code to deal with this proved to be the hardest to write, and by far the most difficult to debug of any of the code in the run time system. It was during the time of debugging the I/O system that an undocumented 'feature' of CP/M raised its ugly head. This was that, when running in a multi-I/O stream environment, CP/M uses the currently selected DMA disc buffer as workspace during a system call for OPEN, CLOSE or CREATE. This had the somewhat mystifying effect of occasionally corrupting either input buffers, or the last but one record written to a disc file. Once this had been

found and fixed, the run time system was in a state where it could be trusted to test much larger programs than had previously been possible.

## C.5 The Bootstrap of the FRONTEND

The large program selected to test the newly written run time system was the frontend of the compiler. One more routine needed to be added to BCPLMAIN before this could be done. This was the routine LOAD, which deals with the overlaying of the SYN and TRN sections of the frontend. Section B.5 contains information about the overlay procedures used, so there is little point on covering it here.

By the time the overlaying routines were working, the code generator no longer produced an assembler listing as its final output, but an OBJ standard INTEL object module. This added yet another variable to the list of possible causes of bugs in the running of a BCPL program. Also, since many alterations were made to frontend to ensure that it fitted into 48K bytes, this meant that a slip in an editing session could cause yet another source of run time error! Before the frontend was working satisfactorily, bugs of each kind were detected, each one more obscure than the last. The most difficult bugs to track down were those associated with the object module. There was one instance where a typing error caused a two-byte subtract instruction, to actually be compiled into a one-byte register to register load instruction, and a byte of, what was now, a random instruction.

## C.6 Bootstrap of the Code Generator

Compared with the frontend, the job of bootstrapping the code generator was exceedingly simple. No overlaying, and only a fixed number of I/O streams are needed. The first bootstrap failed, only because certain MANIFEST declarations in "CGHDR" controlling the workspace size had be set too high. All subsequent bootstraps have proved successful.

This now meant that there was yet another way in which bugs could creep into the system. Now, not only could bugs come from direct errors in the code generator or run time system, but also the bugs which only show on the second

level of compilation, i.e. code produced by the code generator, compiled by itself.

A bug in the register slaving algorithm caused bad arguments to a routine of the code generator, which in turn caused bad code to be compiled for a call the the run time system. The bug - an error in handling GLOBAL or LOCAL variables out of range of the direct IX or IY instructions - took almost two days of constant searching through core dumps, disassembled programs and OCODE listings before the error was traced back to a logical bug in the register slaving.

Since then, no more bugs have shown themselves, and a compiler, compiled through itself, is successfully running on a Cromemco system Z2/H. Self compilation on the Horizon is impractical due to the shortage of available workspace, but this system has been stretched to its limits, and still appears to work.

## Summary

No doubt, as time goes on, the more obscure bugs will show themselves, but hopefully these should be of such a nature as to still allow the recompilation of the compiler through itself. As an assurance against a situation where this is impossible, a version of the code generator is running on the 370 acting as a cross-compiler, providing, hopefully, an error free source from which to re-bootstrap the system.

What I have tried to do in this project, is to write the BCPL system in such a way as to make the user unaware that he is working on an 8 bit microprocessor. The code compiled is good considering the size of the code generator, and the execution speed is comparable to the FORTRAN implementation on the same system. I hope that the immense power that BCPL brings to a system, can be harnessed in such a way as to make the work worth-while, and will be of help to the many people in the country running Z80 based microcomputers.

## Bibliography


[1] Richards, M. 1969 "BCPL - A Tool for Compiler writing and systems programming" (Proceedings of the Spring Joint Computer Conference, Vol. 34)

[2] Richards, M. 1971 "The portability of the BCPL compiler" (Software, Practice and experience Vol. 1)

[3] Richards, M. 1973 "The BCPL programming manual" (Computer Laboratory, Cambridge)

[4] Richards, M. 1973 "Bootstrapping the BCPL compiler using INTCODE" (Computer Laboratory, Cambridge)

[5] Richards, M. 1972 rev 1975 "INTCODE - An Interpretive machine code for BCPL" (Computer Laboratory, Cambridge)

[6] Richards, M. and Whitby-Strevans, C. 1979 "BCPL - The Language and its Compiler" (Cambridge University Press)

[7] Richards, M., Firth, R., Willers, I. and Middleton, M.D. 1979 "A proposed definition of the language BCPL" (BCPL Standards Committee)