1987 Honours Project

A BASIC Translator

Supervisor: P. J. Ashton

Department of Computer Science

University of Canterbury

Simon Dear

-- Contents --

-- Appendices --

Section One.
Introduction.

Datacom Software Research has developed software for a hand-held data recorder used by surveyors, called the SDR2. They would like to enable users to write programs in a BASIC-like language, that can be run on the SDR2.

This project involves writing a translator that will convert a program written in a BASIC-like language into an intermediate code form. The translated programs are to be run on an SDR2. Programs will be developed on a micro-computer (probably an IBM PC or a compatible), and the intermediate code produced will be loaded into the SDR2 where it will be interpreted.

The BASIC-like language (which will be called SDR-BASIC from here on) should provide a useful subset of commonly implemented BASIC features. SDR-BASIC must also provide access to the data stored in the SDR2 memory as well as access to input and output facilities.

1.1. The Aim of The Project.

There are five sub-goals that must be reached in completing this project:

1. Getting to know the SDR2.
Before any other part is undertaken, a survey of the features of the SDR2 needs to be made. An overview of the SDR2 environment is given in Appendix A.

2. Specifying the BASIC-like language.
SDR-BASIC needs be designed so that commonly found BASIC features are present. More importantly, there should be features to allow access to SDR2 internal data structures and input/output facilities. An overview is given in Appendix B.

3. Specifying the intermediate language.
The intermediate language is what will be loaded into the SDR2 for a program to be executed. It will have to be designed to reflect the hardware and software of the SDR2, especially taking into acount the memory limitations. In-depth details are given in Appendix C.

4. Writing the translator.
The translator will need to check the syntax and semantics of an SDR-BASIC program, and produce an intermediate langauge form of the program along with appropriate listings. Documentation is given in Appendix D.

5. Writing the intermediate language interpreter.
The interpreter will be developed on a PDP-11 in ADL, the language in which the software for the SDR2 has been developed. If there is not enough time, a specification of the intermediate language and an outline of the interpreter will be given.

1.2. A Brief Introduction to the SDR2.

The SDR2 collects, verifies and stores observations taken from surveying instruments. It comes with a collection of data collection and calculation programs for surveying applications.

Observations are taken automatically from electronic survey instruments, via a special

interface and cable. Once stored in memory, the data can be used for further calculations, browsed through, or down loaded to a micro-computer using the communications interface.

The SDR2 has 32k bytes of RAM. Observations are stored here as heap records, which vary in length depending on their type. Also the RAM can be used to store external programs, which can be loaded in through the communications interface. Programs that run on the SDR2 are either written in assembler or ADL, a stack-based language rather like Forth.

Unfortunately, the SDR2 is not programmable by the user. As there are competing systems on the market that provide user programming it was considered a worth while project to implement a system which allows the user to run BASIC programs.

Section Two.
Design of SDR-BASIC.

In the original specification of the program there was some flexibility given to the nature of the language. Datacom require that it is BASIC-like. This provides the opportunity to design a language that is significantly more powerful that your everyday BASIC.

The reason why Datacom decided that a BASIC-like language should be implemented is because the SDR2 is a tool for surveyors. It is more likely that surveyors are acquainted with BASIC than any other language.

In specifying SDR-BASIC there were several design goals that had to be met.

2.1. SDR-BASIC Should Provide a Uselful Subset of Common BASIC features..

Some research had to be done to find out what features of BASIC were desirable and should be retained. BASIC is the most common language implemented on small personal computers. BASICs normally vary in the features added to take advantage of the sound and graphics hardware. Control structures, variable types, and built-in mathematical functions are almost standard across BASICs.

It should be noted thet there are many dialects of BASIC. No one has true claim to be the BASIC. There is an ANSI standard for BASIC, but it is not widely implemented. Microsoft BASIC[4], or close relatives, is probably the most widely used dialect. It is almost identical to the BASIC distributed with IBM PCs. Because it is so widely available SDR-BASIC has been based on Microsoft BASIC.

The features that are to be implemented should be ones that appear in most BASICs. They should be common enough for the programmer not to have any difficulty when programming. Also it is considered important that SDR-BASIC has the look and 'feel' of other BASICs. It would have been possible to enhance BASIC to produce a version that had procedures, local variables, user defined data types, and sophisticated file handling. But this is probably a bit excessive, considering that the language is going to be used by surveyors who have only been acquainted with BASIC.

An outline of SDR-BASIC is given in Appendix B.

2.2. SDR-BASIC Should Reflect Its Compiled Nature.

BASIC is traditionally used in an interactive environment. The program is entered line by line, usually by typing the line in directly from the command line. Most systems provide line editors or screen editors so programs can be modified. At any stage, the program can be run. At run time interpreter checks for syntax and semantic errors. When an error occurs, the offending line can be corrected and the program can be run again. Once the program has finished running, the variables can still be interrogated from the command line.

As an SDR-BASIC program is compiled into an intermediate code, many of the features that come with interactive BASICs simply will not apply. There will be no need for commands that are normally issued from the command line, like DELETE (a line), SAVE, LOAD (a program), RUN, CONT or LIST. Also commands like STOP which are normally issued from inside the program will not be needed.

It is desirable to do as much work at translation time as possible. All syntax checking is done at translation time, rather than at run-time in many interactive systems. Also, type checking and memory allocation should be done at translation time. Therefore, if any enhancements to commonly used BASIC features further this aim they should be adopted.

One of the most noticeable features of BASICs is that every line has a line number or label attached to it. In an interactive environment this is one way of being able to specify the logic ordering of lines without resorting to a screen editor. In a compiled version of BASIC beginning each line with a label is rather pointless. The logical ordering of the lines is simply the physical ordering of the lines as they appear in the source. The other use for line numbers is to specify the line where execution will continue after a branch. For this reason, line numbers have been retained in SDR-BASIC. They only need to be at the start of the line that is branched to. Even though line numbers are integers, in SDR-BASIC they can appear in any order, as with Fortran.

In some BASICs the binding between FOR and NEXT in loops is done at run time rather than at compile time. For example, the following will loop around 10 times.

```
10 GOTO 40
20 NEXT i%
30 END
40 FOR i% = 1 TO 10
50 GOTO 20
```

However, this dynamic binding only leads to spaghetti-like looping structures and makes the program hard to follow. Therefore SDR-BASIC forces the programmer to have the NEXT following the FOR. This means that the FOR is always bound to the same NEXT. This has the advantage that less run-time management of for loops is needed.The very same restriction applies to REPEAT...UNTIL and WHILE...WEND loops as well.

2.3. Control of Memory Usage is Needed.

Memory allocation poses a major problem for most BASICs. The space requirements for strings and arrays is normally determined at run time. This is not at all desirable in a compiled BASIC, especially SDR-BASIC as memory on the SDR2 is a limited resource.

The dimensions of an array in BASIC are specified by using a DIM statement, which has the following syntax:

```
DIM arrayname (dimensions)
```

The problem that arises here is the fact that the dimension is specified as an integer *expression*. It is almost impossible for the translator do determine the value of the expression at translation time as the expression might have operands that have yet to be initialised. Therefore, in SDR-BASIC the dimensions must be specified as integer *constants*, thus avoiding any problems. A possible enhancement to this scheme is to allow constant *expressions* (that is, expressions involving only constants), but this was not attempted.

BASIC strings have dynamic string lengths. Every string has a maximum possible length (normally 256 characters), but the space needed to store the string depends only on its current length. On the SDR2 this scheme could cause problems, as there is little or no control on the string space that is required. Consider for example:

```
DIM A$(100)
```

There is a 3 byte (16 bit address of string and one byte string length) overhead for each of the 101 array elements. This creates a minimum of 303 (=101 * 3) bytes storage requirement. However, if all strings were their maximum length, then a maximum of 26159 (=(256 + 3) * 101) bytes is needed. As the SDR2 has only 32k of RAM, some steps are needed to avoid memory-hungry programs.

For this reason, in SDR-BASIC strings have a default maximum length of just 32 characters. If larger strings are needed (or shorter ones are desired!), the programmer has to specify the space needed. To do this, the length of the string should be enclosed between brackets after the string identifier, at the first reference to the string in the program. For example,

```
fred$[128] = ""
```

sets the upper limit of the length of string `fred$` to be 128.

Using this method, the storage requirements for strings is known exactly at translation-time, and the amount can be kept to a manageable level.

2.4. SDR-BASIC Should Take Into Account The Limitations Of The SDR2s Hardware.

As SDR-BASIC is to be run on a hand-held computer which has its own set of idiosyncrasies, constriants have had to be imposed on some BASIC features.

The main one concerns the INPUT statement. In most BASICs the syntax is:

INPUT {*optional prompt*;} *variable* (..*variable*)

However, remember that the SDR2 has only a 16 character display. If a large number of variables needed to be inputed, the prompt could very well scroll off the screen to make room for the typed data. This would be very off putting. Therefore the restriction that only one input variable can be included in the INPUT statement has been made. This is probably desirable from the point of view that there could be difficulty in getting the SDR2s input routines to handle multiple input values anyway.

2.5. SDR-BASIC Should Meet The SDR2 Requirements.

An important requirement of SDR-BASIC was to provide access to the internal data structures and input/output facilities of the SDR2.

2.5.1. Access to the SDR2 Heap.

The user needs to be able to add as well as retrieve records from the heap. There does not need to be any provision for deletion and editing of records. This is because, under New Zealand law, once a surveyor has made an observation he or she is required not to delete or modify it.

The heap consists of records of varying types. This in itself poses problems for adding and retreiving. Also BASIC doesn't provide for any way of handling records. To get around this problem the following methods were considered:

(a) For retreiving from the heap, successive characters (or possibly one string) are read. The programmer has to convert numeric strings to numbers, and interpret the data him/herself. Writing is done in a similar way.

(b) For retreiving, string and numeric values can be read successivly. The programmer has to make sure that he or she is reading the right type at the right time. Writing is done in a similar way.

(c) Implement records structures in BASIC. Whole heap records could be retreived and added at one time.

(d) Several variables could be predefined, each one representing a different heap field. Retreiving would be done by one command that would retrieve the record type off the heap, and then update appropriate variables. Adding records would be done by looking at the type of record to be added and the using the appropriate variables.

All Options require some heap manipulation functions to be implemented. Options (c) and (d) would be the easiest for the programmer to use. (a) and (b) involve a lot of fiddling around, and the prossibility of errors arising due to unread or unwritten data is quite high. Option (c) requires major enhancements to be made to BASIC, as record structures need to be implemented. The last option does not require any further enhancements to BASIC (other than the few extra heap functions), and the resulting apperance does not deviate from BASIC at all.

Weighing all this up, option (d) was chosen, where predefined variables would be used. This method also involves a memory saving as the predefined variables could reside in the translator's memory space rather than in the code space.

SDR-BASIC allows for the following operations to be made on the heap. Functions that support searching are already built into the SDR2.

Retreiving:

Retreive the last element added to the heap. (GETLAST)
Retreive the record that was added after the last one referenced. (GETNEXT)
Retreive the record that was added before last one referenced. (GETPREVIOUS)

Searching:

Search through the heap, and retrieve the last record of the given type.(GETTYPE (*type*))
Search through the heap and retreive the last record which contains the given point number (either as a source point or a target point). (GETPT (*point*))

Adding:

Add a record to the end of the heap. The records type is in stored in the variable RECORDTYPE. (ADD).

2.5.2. NULL fields in the heap.

In heap records some REAL fields have a special value to indicate they are empty, as opposed to containing the value zero. This special value is NULL. NULL is a special form of real. It has the property of being able to be propagated through expressions. That is, if one of the operands in a real expression is NULL then the result is also NULL.

SDR-BASIC has a 'pre-defined' constant called NULL. It can be used anywhere a real value could normally be present. That is, in expressions or in comparisons.

2.5.3. Input Output Facilities.

SDR-BASIC should be able to let programmers to input and output data. Output to the screen and input from the keyboard have been considered earlier.

Input can come from two places: readings taken automatically from electronic surveying equipment, and data sent via the RS232 port. As the one interface has a dual role of being the connector for to surveying equipment, these two are basically the same.

Output can be via the RS232 port, or via the acoustic coupler. The SDR2 has an internal parameter called ACOUSTIC which determines where the output is to be directed to. To avoid the programmer having to deal with this, two output commands are supported by SDR-BASIC.

Output: Output via the RS232 port. (LPRINT)
Output via the acoustic coupler. (PRINT#)

Input: Input via the RS232 port. (LINPUT)
Input from surveying equipment. (INPUT#)

Section Three.
Design of the Intermediate Language.

SDR-BASIC programs will be translated into some other langauge. It is this other language that will be loaded into the SDR2 when a program is to be run. This section addresses the choice and design aspects of this language.

3.1. The Style of Code Produced. ADL or Intermediate Language?

There are many possible choices for the langauge that the translator produces.
(a) The machine code native to the SDR2.
(b) An intermediate language, such as P-codes.
(c) A tokenised form of BASIC.
(d) ADL, the language in which many of the SDR2 program have been written.

It is desirable for the translator to do as much work (breaking high level commands into more primitive ones, converting expressions into post-fix notation etcetera) as possible in the translation process. Therefore the choice of simply tokenising is not a wise one. Also, many high level features of the SDR2 need to be accessible. Producing machine code would make this very difficult. Therefore options (b) and (d) are the only sensible options.

To determine the better of translating into ADL or into an intermediate language, the following issues should be considered:

(a) Ease of Implementation.
If the intermediate language approach was taken then both a translator and an interpreter would have to be written. However intermediate language generation is relatively straight forward, especially as the author has had much experience in this approach.

Producing ADL code has the advantage that only the translator need be written. However, it will undoubtably be much more complicated than with the intermediate language approach. One reason for this is that libraries would need to be used to avoid including large pieces of unnecessary code. Linking would be done as a part of the translation process. There would also be problems with the constraints placed upon external ADL programs. In particular, an external program can have only 128 variables. Also there is a limit on how large the program can be. Therefore it is doubtful that large SDR-BASIC programs could be written.

(b) Memory Requirements
Memory usage is an important issue with the SDR2. Therefore the end approach should require as little of it as possible.

The intermediate language would be interpreted. This requires both the code of translated program and the interpreter to be in memory at one time. However, the intermediate code should be quite compact, as the code to execute high level instructions would appear in the interpreter only once. Also, it is possibility for the variable storage space to be controlled by the interpreter. This would mean that this space would only need to be allocated for the one running program.

The alternative results in there only being the translated program in memory at one time. However, the code for high level commands would be duplicated in all programs that were currently in memory.

(c) Maintainability.
(i) Changes to the way ADL is implemented might occur in the future (see Appendix A). It is possible for several versions of the SDR2 EPROM to be in existence at one time.

The intermediate language approach gives some independence between the code generated and the version of SDR2 being run. If a new version of the SDR2 is released, only the interpreter would need to be modified (or recompiled). The translator itself would not require any modification.

The ADL approach gives no independence from the contents of the EPROM. Therefore, it is possible that several versions of the translator might be in existance. When a new EPROM is released, the code generation of the translator would need to be modified, as well as the ADL libraries required for linking. Also, all translated programs would have to be retranslated.

(ii) Enhancements to SDR-BASIC might be made in the future (see Section 6.2.1.). The translator will need to be modified to reflect these changes.

With an intermediate language, the translator would have to be modified to make allowances for the changes. If the intermediate code also had to be updated then the interpreter twould have to be changed. As there could be several versions of the interpreter in existance (one for each version of the SDR2 EPROM), each version would require modification.

With ADL code, the translator would require modifiaction and possibly some new entries to the libraries.

(d) Portability.
Translated programs could be given to SDR2 users who do not have access to the translator. Also, high level functions could be developed by Datacom and then distributed to SDR2 users.

As the intermediate language is independent from the SDR2 EPROM version, intermediate code would be portable. Users without access to a translator would require a copy of the interpreter. Several versions of the intermediate language could be in existance at any one time (see part (c)). Therefore file transfer could only be done between systems running the same interpreter.

If ADL code was produced by the translator, code could only be ported between SDR2s with the same version EPROM.

(e) Speed.
As ADL is itself interpreted (by a machine code program), using an interpreter would add another level of complexity. It is likely that interpreted programs will run several times slower than ADL programs doing the same thing.

(f) Run Time Errors.
As inexperienced users will be writing the programs it is possible that run time errors will occur. Having an interpreter gives one more layer of protection from nastey side effects than does the ADL code approach. Meaningful error message could be generated by the interpreter when something untoward happens. This possibility does not arise with the alternative approach.

When Datacom suggested this project they had envisaged the intermediate langauge approach would be adopted. From the previous discussion, it is very hard to conclude if this is indeed the best. To arrive at any conclusion one must assume probabilities of certain factors occuring.

However, because the intermediate language approach is easier to implement and possibly easier to maintain and port, it has been the one adopted.

3.2. The Choice of the Intermediate Language.

There are many types of intermediate langauges that could be used for this project. Much time could be saved if a standard intermediate language were adopted.

The advantages of using a standard intermediate language are many. There are code optimisers and improvers available that could be improve the compactness of the code. Also there are many interpretters available, which would come in useful in the testing stage at least. Using a standard intermediate language increases the portability of SDR-BASIC to other machines.

However, SDR-BASIC calls for many hardware and SDR2 specific instructions. It is possible that interfacing to the SDR2 could be achieved using an standard intermediate language. This would be at the sake of compactness, or would involve introducing new instructions resulting in losing the standardness aspect. For example, the instruction for adding a new heap element could simply be 'Add'. In a standard intermediate langauge the code to do the same thing would be broken down into loading all relevant variables on the stack, and then call an external procedure.

Designing a custom intermediate language for SDR-BASIC has the advantage that the choice of instructions can be tailored to BASIC. It is not always desirable to break down BASIC statements down into primitive instructions. Using a higher-level instructions instead of many low level ones has the advantage of using less memory as well as improving the execution speed. An example is the code generated for the FOR...NEXT loop. The syntax of the loop is:

FOR *loop variable* =*initial value* TO *upper bound* {STEP *increment*}
loop body
NEXT {*loop variable*}

The step increment can be an expression and because of this the value of the expression might only be known at run time. Therefore the code generated under a standard coding scheme must take into account the fact that the sign of the increment affects the test that has to be done at the end of each loop iteration. If the increment is positive then the loop cycles around until the loop variable is greater than the upper bound. Otherwise the loop cycles around until the loop variable is less than the 'upper' bound. A much more elegant way is to have an instruction called NEXT. The interpretation of NEXT takes into account the sign of the increment.

3.3. The Intermediate Language Adopted.

The intermediate langauge that has been used is outlined in Appendix C. The main features of it are:

(a) The name of the instructions are meaningful!
The names of the instructions are given meaningful names, rather that cryptic four letter mnemonics. For example, there are

```
AddInteger
BranchTrue
PrintString
```

(b) It is based on a stack machine rather than one with registers.
This method was chosen because it is far simpler to generate intermediate code. With a register-based machine the translator must keep acount of which registers hold values to be operated upon. Also, the instructions themselves are more complicated as information about source and destination addresses must be stored. In addition, the interpreter will be easier to write as ADL is itself stack based.

(c) The stack elements are 16 bit signed integers.
This size was chosen because ADL integers (and addresses) are also 16bits long.

REALs and STRINGs are represented on the stack by their address. REALs are in the same format as in the SDR2; 6 bytes. Strings are stored as a length byte followed by the string. Note that this effectively limits strings to be less the 256 characters long.

(d) Instructions vary in length.
Instructions can consists from between one byte and 257 bytes. Most instructions are either one or three bytes long. One byte is alway used for the operation part of the instruction. Two bytes are used for the operand part. Exceptions to these rules are:

```
LoadConstReal real
LoadConstString length string
BranchIndexed numLabels labelList
CallIndexed numLabels labelList
```

(e) There are special purpose instructions for SDR2 access.
To interface with SDR2 features there are several simple instructions. It is left up to the interpreter to carry out the more detailed aspects of these instructions, such as assigning values to the appropriate variables.

Section Four.
The SDR-BASIC Translator.

The SDR-BASIC translator should do the following.

(a) Check the syntax of an SDR-BASIC program and give appropriate error messages.
The translator should generate error messages that are meaningful to the programmer, and give some indication where they have occurred. To avoid errors cascading, some form of resynchronisation is needed.

(b) Check that typing is consistent and give appropriate error messages.
The parser should trap errors in expressions that are in conflict with the type rules of SDR-BASIC.

(c) Produce an intermediate code version of the SDR-BASIC program.
A valid intermediate code version (or none at all) should be produced. Space for variables should be allocated. The code should be dumped to a file in the SDR2 extrenal program format.

(d) Produce appropriate listings.
An listing showing where errors occur in the program.

There are many ways of writing a translator. There are systematic methods such as recursive descent parsers. Given an LL grammar for the language, there are well defined methods for writing a compiler in a high level language. Such an approach can be fast and is very flexible.

Translators can also be developed with automatic parser generators, or so-called compiler compilers. An example is YACC that is supplied with the Unix operating system. The advantage of using such software tools is that it is very easy to produce a prototype. However from personal experience they have been found to perform badly when error detection and recovery is added to the parser.

If an automatic parser generator was to be used for developing this project, it would probably be done with using YACC on the departmental version of Unix. However, Datacom do not have Unix. Any future maintenance would be quite inconvenient for them. The parser would require porting to an IBM PC every time modifications to the translator were made.

For these reasons it was considered that a recursive descent translator should be developed in a micro-computer environment. The prefered language to use was Pascal, and as Datacom have many IBM PCs using Turbo Pascal, this product was used. Due to the unavailability of IBM PCs or clones, the development was done on an Apple Macintosh usingTurbo Pascal, and the final product will be ported to an IBM PC.

4.1. Issues in Translator Design.

Conceptually a translator consists of four main program units
(a) scanner (lexical analyser),
(b) parser (syntax analyser),
(c) semantic analyser,
(d) code generator.

The scanner processes the input characters and recognises the symbols of the language. The parser takes these symbols and recognises the constituent parts of the program. With knowledge of these constituent parts the semantic analyser can gather information about what the program means. With this information the code generator can then generate equivalent code.

In recursive descent parsers, the actual structure of a translator differs slightly this conceptual view. Central to it are *compiling procedures* that do the syntax analysis, which call the scanner, semantic analyser, and code generator. Often the semantic analysis is done along with the syntax analysis in the compiling procedures.

Symbolic information about symbols and variables are stored in a symbol table. All four program units of the translator need to access the table. The scanner interrogates it to check if the string of characters read in correspond to a reserved word, or a user variable. The parser has to update information about variables as the are declared. The semantic analyser updates and uses information about variable types. The code generator stores and retrieves information about memory addresses of variables.

The interaction between programs units and the symbol table is shown in figure 1.

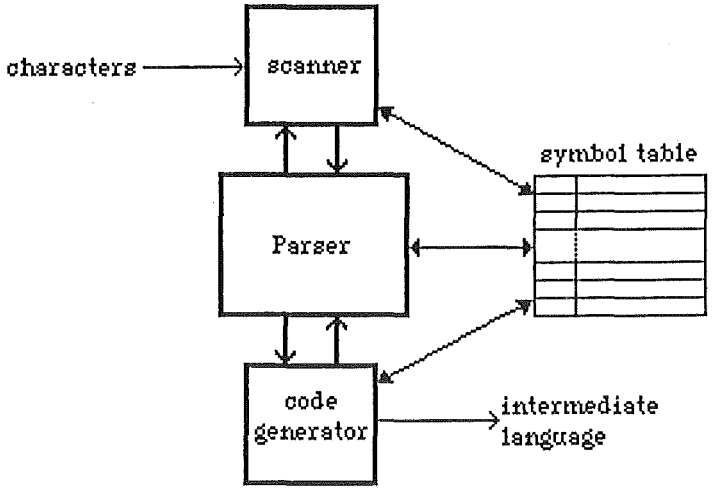


Figure 1.

4.2. Issues in Translator Implementation.

The translator that has been implemented is based very loosely on the P4 public domain Pascal compiler [2,3]. The translator is a one pass compiler. It stores the code that it builds up in memory until all parsing has finished. Because of this, the size of an SDR-BASIC program is limited. However, as memory is critical on the SDR2 anyway this does not pose much of a problem. At present, up to 8k of code can be generated successfully.

4.2.1. The Symbol Table.

In the SDR-BASIC translator, the symbol table consists of an array of table elements. Each element consists of an identifier, a class identifier, and various other fields whose relevance depends on the class.

The classes of symbol table records and their relevant fields are:

Class	Fields
Reserved words.	The corresponding symbol.
Built-in functions.	The corresponding symbol.
	The type the function returns.
Labels.	A flag to indicate that the label has been referenced.
	A flag to indicate that the label exists.
	The address of where the code for the line starts. †
	The address of where the users data starts after this line.
	The address in the code space of last used RESTORE statement to this line. †
Variables.	The type.
	For string variables, the maximum size.
	The address of the variable. †
Array variables.	The type.
	For string variables, the maximum size.
	Number of dimensions.
	Linked list of array dimensions.
	The address of the variable. †

Note: The exact location of these might not be known at the time they are first referenced. In such cases, all references to them are chained together until the actual address is known. When it is, the chain is used to determine the addresses in the code space that need to be patched up. The last element in the chain has a value of -1.

In the SDR-BASIC translator, there are 503 elements in the hash table. Of these about 130 are taken up with reserved words and built-in functions. When the number of elements used is 450, no more elements can be elements. This is to avoid degrading of performance when the table gets full. It is unlikely that this will happen anyway, as there is room for 320 vraiables, arrays, user defined functions, and labels. If more space is needed, a constant in the program is all that need be changed.

A hash function is used to find particular elements in the symbol table. The key for the function is the identifier. The hash function used is:

```
index = (ordinal value of first character * 17 +
         ordinal value of last character * 103 +
         length of identifier * 55) modulo 503
```

If the element at `index` is not the desired one, 37 is added to the index (modulo 503). The resolving continues until either the correct or an unused element is found. This hash function was chosen because it gives good results. Arriving at it was a bit of a hit and miss affair. The SDR-BASIC translator keeps a record of the number of retries that had to be made. By monitoring this, and changing the values of the factors, the hash function was able to be fine tuned.

4.2.2.The Scanner

The scanner converts input characters to symbols. The following diagrams shows how the input symbol is determined from the first non-blank character.

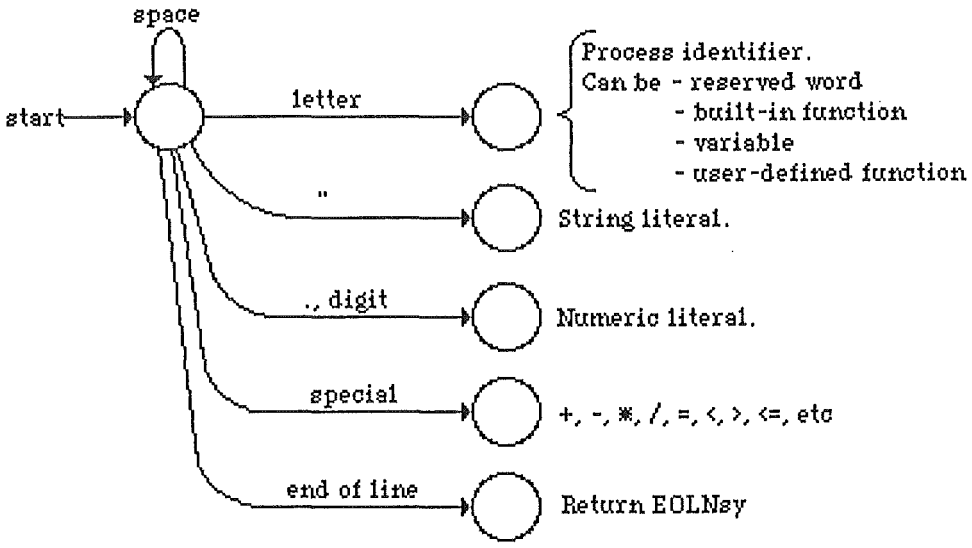


Figure 2.

Several unusual features of the scanner need explaining.

In SDR-BASIC a variable is identified by its identifier and type. This means that all the following variables are all different.

`banana%`	(integer)
`banana$`	(string)
`banana!`	(real)
`banana%()`	(integer array)
`banana$()`	(string array)
`banana!()`	(real array)

To complicate matters, the following can appear:

`banana`	(default type for identifiers starting with 'b')
`banana()`	(array of default type for identifiers starting with 'b')

At some stage, the type of the identifier read in must be determined. To simplify the parser, this is done in the scanner. As variables have unique entries in the symbol table, a unique identifier is created from the one read in. For example, if '`banana`' was read in, and the default type for variable begining with 'b' was string, then the unique identifier used is '`banana$`'. Similarly, if the array '`banana`' was read, '`banana$(`' would be used.

Labels are handled in the scanner rather than in the parser. A label always appears at the start of the line. When the scanner finds a integer constant at the start of the line, it adds the label to the symbol table. Also, if there is are any previous references to this line earlier in the program, the scanner patches up the code space so that the branches are directed to the current value of the code pointer.

4.2.3. The Parser.

The parser is modelled on a recursive descent approach. Error detection and resolution, and type checking have been added. These aspects are very well documented in the literature on the subject [5]. Therefore the coverage here is rather superficial.

4.2.3.1. Recursive Descent Parsers.

Recursive descent compilers are designed for LL grammars. An LL grammar for SDR-BASIC is given in Appendix B.

The parser (for an LL1 grammar) is constructed as follows. For each terminal symbol in the grammar, a procedure is written that checks to see if the last scanned symbol is correct, and then reads in the next symbol by calling the scanner. For each production, a procedure is written which calls the procedures corresponding to the symbols in the right hand side of the production in order. The parser is started by calling the scanner, and then calling the top-level parsing procedure.

This is a very basic recipe. It will normally be modified to avoid recursive calls. Also, terminal symbols that appear only in one production might be handled in the procedure for that production. For example, in the SDR-BASIC grammar, we have

```
<program> ::= <lines>
<lines>   ::= <line> | <line> eoln <lines>
```

For these productions the following Pascal procedures could be constructed.

```
procedure pPROGRAM;
begin
   pLINES;
   end;

procedure pLINES;
begin
   pLINE;
   while (insy = eolnsy) do begin
      insymbol; {skip past end of line}
      pLINE;
      end;
   end;
```

The parsing procedures are normally padded out with error detection code. Also, in some places type checking code needs to be added.

4.2.3.2. Error Handling.

To detect errors a check is made to see if the symbol last read in from the scanner is the same as the one that is expected. If they are different, an error message should be displayed, and the symbol stream from the scanner should be resynchronised.

Resynchronisation requires knowing all the symbols that can follow the one that is being checked in the given context. To do this, a follow set is passed as a parameter to each of the parsers procedures. The follow set passed on consists of the follow set passed to the calling procedure plus the symbols that can follow in the given context. To resynchronise, input symbols are skipped until one if found to be in the follow set.

4.2.3.3. Type Checking.

Most of the type checking is concerned with expression parsing. The parser has to make sure that the types of operands being operated upon are the same. If they are not, either type coercion must be done or a type mismatch error must be given.

The way type checking for expressions is done is the same as if a parse tree were built up for the expression. At the leaves of this tree there are operands. At the nodes there are operators. When the scanner reads in an operand (variable, user defined function, literal, or built-in function) it can determine its type. Starting at the nodes of the tree, the types can be propagated up until the type of the root node has known. The expected type of the expression can then be checked with the expected type of the top node for compatibility.

The propagation of types up the tree is done as follows. Each operator can only operate on a subset of types.

operator	valid types
unary -	integer, real
+	integer, real, string
-	integer, real
*	integer, real
/	real
div	integer
^	real
<,≤,>,≥,≠	integer, real, string
and, or, not	integer

Therefore, a check has to be made to see if an operator and its two operands are compatible. If they are not, an error has occured. If they are, then the two operands are corerced to the same type if need be, and the type is passed up the tree.

4.2.3.4, Automatic Type Conversion.

Coercion (automatic type conversion) occurs when the given and expected types in an expression are different, and it is possible to convert the given type to the expected one. In SDR-BASIC, this can happen between integers and reals. There are occasions when reals are coerced into integer as well as when integers are coerced into reals. If an operand is valid for both integers and reals, and coercion is needed, then the integer will be widened to form a real. An exception to the widening rule is with assignment, where coercion is always to the type of the left hand side variable.

Consider the following example.

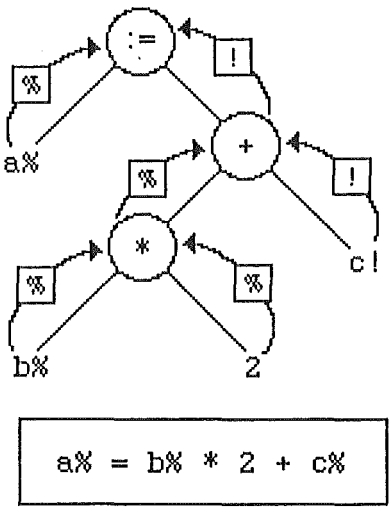


Figure 3.

Here, the * operator is operating on two integer operands. Therefore it produces a result of type integer. The + operator is operating on an integer and a real. To make the operands the same type, the integer is widened to a real. The assignment has a left hand of type integer and a right hand side of type real. On assignment, the right hand side is always coerced to the type of the left hand side.

4.2.4. Code Generation.

Code is generated as the program is being parsed. Calls to the code generating procedures are made by the parser. There are procedures to do the following.

Add a instruction to the code space (procedure Gen1)
Add an integer (2 bytes) to the code space (procedure Gen2)
Add a byte to the code space (procedure Gen3)
Add a real (6 bytes) to the code space (procedure Gen4)
Add a string (1-256 bytes) to the code space (procedure Gen5)
Create space for integer variables (procedure CreateIntegerSpace)
Create space for string variabels (procedure CreateStringSpace)
Create space for real variabels (procedure CreateRealSpace)

These routines maintain the code space and various pointers that reference it.

Each part of the parser handles the code generation for the part of the language that it parses. As an example, consider the SDR-BASIC GOTO statement. The syntax of the statement is:

```
GOTO label
```

The simplified code for this looks like:

```
(**)Gen1 (Branch);    {Generate Branch instruction}
pLABEL (...);         {Let pLABEL handle the rest!}
```

Using this methed, code generation can be approached in a modular way.

4.2.4.1. The Code Generated.

The code space generated by the translator looks like:

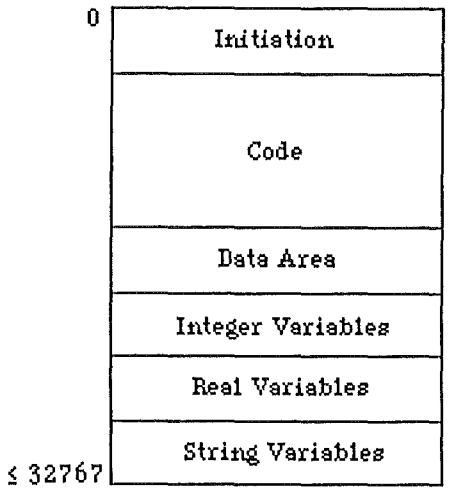


Figure 4.

The initiation part of the code space consists of instructions to initialise the data pointer and variables to their default values.

The length of the code varies in length. It has all the code corresponding to the SDR-BASIC program. The code area always ends in an `Exit` instruction.

The data area is variable length and always ends in two bytes of 255.

All variables of the same type are strored contiguously. This has been done for two reasons.
(a) To simplify the initialisation of variables.
(b) For future enhancements to the interpreter. At some later stage, variable space storage could be allocated in the interpreter. This would make the object code file much smaller. If variable storage space was deallocated when a program terminates, memory would be saved in the SDR2. This is becaues only one program's variables are active at on time.

4.3. Translator Output.

The translator must produce an object code file in the form of an SDR2 external program. The format consists of lines of the following ASCII characters.

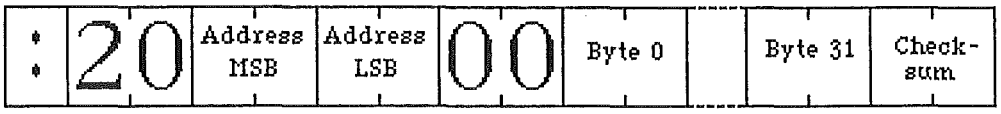


Figure 5.

Bytes appear as the ASCII representation of their hexadecimal notation. For example, 32 is represented as the two characters '20'. The check sum is the negative of the sum of all the bytes on the line.

4.4. Issues in Porting the Macintosh Version to an IBM PC.

As the Macintosh version of the translator has been written in Turbo Pascal, it should be relatively stright forward to port it to an IBM PC environment.

Macintosh Turbo Pascal and IBM PC Turbo Pascal have the following differences:

(a). In both versions large programs must be broken up into segments. There are separate segments for code, data and heap space. The size of the segment depends on the computer system being used.

On the Macintosh, the segments are 32k long. There is a very simple compiler directive for doing this. One simply includes a Pascal comment before a function or procedure in the form of

```
{S+ segment name}
```

All code generated after this point will be placed in the named segment.

On the IBM PC, the segments are 64k long. If more than one segment is needed, a complicated method of overlays is required.

The amount of 68000 code that is generated on the Macintosh is about 52k. This means that on the Macintosh segmentation has had to be done. However, assuming that about the same amount of 8086 code is generated on an IBM PC, there will be no need for segmentation.

(b) The Macintosh Version of Turbo Pascal has many enhancements over the PC version. For example, there are more powerful string to number conversions. Such functions have been avoided if they make for harder porting.

Section Five.
The Interpreter.

Translated programs will be loaded into the SDR2 where they will be run by an interpreter. The interpreter will be written in ADL and loaded as an external program.

Because of lack of time, this part of the project has not been completed. This section gives an outline to what the interpreter should do. Possible difficulties with implementation are high lighted.

(a) Determining which program to interpret.
The interpreter needs to know which program to interpret. As there can be many in the SDR2's memory at any one time, some sort of menu selection is needed. Once the program has been selected interpretation can start.

(b) Machine variables.
Effectively, the interpreter emulates a virtual machine, whose machine language is the same as the intermediate language used for this project. The interpreter will therefore need to have some variables and data structures in order to manage the code and data.

i. Program Counter.
The program counter holds the address of the next instruction to be interpreted. Before a program is interpreted, the program counter is initialised to zero.

ii. Program Stack.
The intermediate language that is used is a stack oriented one. Therefore a stack must be implemented. It is possible that the ADL stack can be used for this purpose. The stack will be empty when the program starts and when it finishes.

The stack elements are 16 bit signed integers.

iii. Stack Pointer.
If a stack is implemented, a stack pointer is needed to point to the first empty element in the stack.

iv. Frame Pointer.
A frame pointer is needed to reference parameters to user defined functions and built-in functions in the stack.

v. Data Pointer.
The data pointer is needed to reference the data area, where the values in the SDR-BASIC DATA statement are stored. The data pointer is reset by the `Restore` *`address`* instruction.

vi. Real and String Operand Space.
As stack elements are 16 bit values, real and string operands cannot be stored there. Some area is needed internally in the interpreter for this purpose. A problem arises here because there is no limit to the number of real or string operands that need to be stored. For example,

$$a! = b1! \ \hat{} \ b2! \ \hat{} \ b3! \ \hat{} \ b4! \ \hat{} \ \ldots \ b_n!$$

requires space for n reals.

The values placed in the stack need only be what is convenient for the interpreter, as the type of operands is known from the context of the program. Addresses would suffice, as would indexes into the array where the real and string values are stored.

(c) Interpreting Instructions.
The interpreter proceeds by starting at the first instruction in the code. It must then interpret it. This requires reading relevant operands from the code, and stack, and updating the stack, program variables and 'machine' variables. After the instruction has been interpreted, the interpreter should procede to the next.

As the intermediate code will reside in the SDR2 memory at an address not known at translation time, all addresses in the code are in effect an offset from the start address of the code. Therefore, address conversion will need to be made at run-time.

Interpretation stops when an error or an Exit instruction is reached.

(d) Problem Instructions.
Most of the intermediate language instructions are straight forward. However, some are quite complicated.

i. NextInteger, NextReal.
These are high level instructions for controlling the FOR looping construct in BASIC. The format of these is:

```
NextInteger start of loop address
NextReal start of loop address
```

On the stack both expect the following:

```
Address of loop variable (Next to top)
Address of control block (Top of Stack)
```

The control block consists of the following:

```
Upper bound of loop
Loop increment value
```

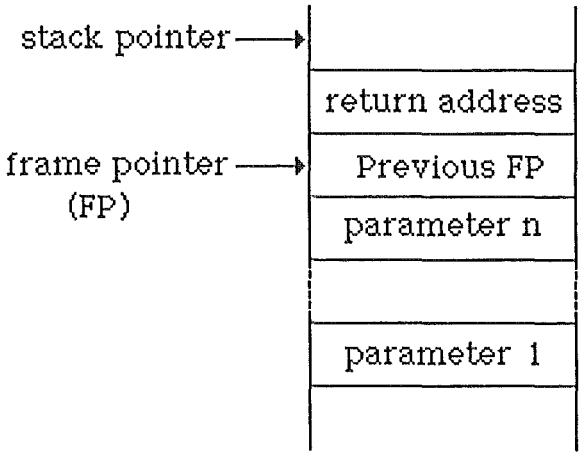
The type of these and the loop variable are the same.

The processing that has to be done is:
1) Load the value of the loop variable onto the stack.
2) Load the loop increment value.
3) Add these values together to give the next value of the loop variable.
4) Store the new value of the loop variable.
4) If the sign of the increment value is positive then:
4a) if the upper bound is greater than the new loop variable value, branch to start of loop, else terminate loop, else:
4b) if the upper bound is less than the new loop variable value, branch to start of loop, else terminate loop.

ii.ReadInteger, ReadReal, ReadString
Only strings are stored in the data area. Therefore, to read an integer or a real, the string will need to be converted.

Also, a check must be made to see it all the data has been used. Two bytes of 255 indicate the end of the data area. When these are reached, an error should be issued and the interpretation should cease.

iii. CallFunc, ReturnFunc
When a user-defined function is called a stack frame must be built up. In this frame, the previous value of the frame pointer and the return address must be stored. After a function with n parameters has been called, the stack looks like:



The user-defined function will cause the resulting value to be left on the stack. On return from a function, this stack frame must be broken down. The program counter will need to be set to the return address, the frame pointer reset. The parameters on the stack should be removed. The resulting stack will have the result of the function at the top.

iv. Heap Access.
The approach for heap access was designed to be easy for the programmer to use and for the writer of the translator to write. Unfortunately, the interpretation will be very difficult.

1. Fields
SDR-BASIC has a set of predefined variables that correspond to the names of the fields in heap records. The variables are stored in the interpreters address space rather than in the translated program's variable storage area. The reason for this is that there are many of them, and having to include them in all programs would require a lot of memory.

2. Adding records to the heap.
When a record is added to the heap, the interpreter has to determine the type of the record. This is done by looking in the appropriate record. It must then transfer the variables corresponding to fields in the records of this type to the heap. To do this efficiently, a look-up table is required.

3. Retreiving records from the heap.
When the record in the heap is located, its type is checked. Variables corresponding to fields in records of this type must then be updated from the fields in this record.

Section Six. Results.

6.1. What has been done.

Up until mid term 1987, much time was spend researching relevant topics. A survey of the SDR2 was undertaken. What was found is given in Appendix A.

Many versions of BASIC were examined for suitable features. Of these Microsoft BASIC was chosen as the one to base SDR-BASIC (see Appendix B) upon.

From personal experience and from relevant literature, the intermediate language was designed. An overview is given in Appendix C.

Programming was started in mid term 1987. It was originally intended for the development work be done on an IBM PC, lent by Datacom. Because of high demand for IBM PCs at Datacom, this was not possible.

The translator was written using Turbo Pascal on an Apple Macintosh. Over 3500 lines of code have been produced. The translator has been completed to the stage where it correctly parses, type checks and generates code. If there are no errors, the code generated is output in the SDR2 external file format. An error listing is produced. Also, a dump of the code in intermediate language format is produced. Some documentation for the translator is given in Appendix D.

There are still one or two loose ends to tidy up with the translator. The author did not have time to determine the format of real numbers as they are stored internally in the SDR2. Where reals must be put in the code (in the LoadConstInteger instruction) an appropriate sized space is allocated.

The translator has not yet been ported to an IBM PC. However, there should be little difficulty in doing this.

Because of lack of time, the implementation of the interpreter was not attempted. Much back ground work would need to be done before any programming could start. ADL has to be mastered, along with understanding the many details of the internal structures of the SDR2.

6.2. What More Could Be Done.

There is still a lot of work that could be done in this project. It there was time the following would have been attempted.

6.2.1. Enhancing SDR-BASIC.

As it stands, SDR-BASIC provides some powerful features, including many enhancements to allow the programmer to use SDR2 features. However, there are still many areas which could be improved.

6.2.1.1. Adding more BASIC features.

SDR-BASIC is based on Microsoft BASIC. There are many features that have not been implemented so far because they were considered to be not important. Many of these are string functions, like STRING$(string, n) which returns a string constisting of the given string duplicated n times. There are also numerical functions like a random number generator. These would not be useful for most surveying applications. However, they would be if the programmer would like to develop recreational programs.

An important omission is that there is no way of reading one input character at a time from the keyboard. SDR-BASIC INPUT reads in a whole line of characters that end in a new-line. Often there are times when it would be convenient to scan the keyboard to see whick keys are being depressed or to wait until one key is pressed. Many BASICs provide an INKEY$ or GET function to do precisely this. These could be adopted by SDR-BASIC.

Experienced SDR-BASIC programmers might like there to be structured programming constructs. BASIC programs tend to turn quickly into a rats nest of branches. Structured programming constructs like procedures, functions, and local variables could make SDR-BASIC programms much more readable.

6.2.1.2. Providing better interfacing with the SDR2.

With SDR-BASIC a programmer can access heap records and write their own input/output drivers. There are many other features of the SDR2 that would be useful.

(a) Access to the configuaration variables.
Configuration variables specify the working environment of the SDR2. They can be viewed using the Parameters menu option. The variables include:
Unit used for input and output. (Values stored in the heap are in standard units).
- Angle (degrees)
- Distance (meters)
- Pressure (mm Hg)
- Temperature (degrees Celsius)

Optional correction flags
- sea level correction
- Atmosphere correction

Tolerances
- Vertical observation tolerance
- Horizontal observation tolerance

Input/Output parameters
- Tranfer speed
- Parity option flag
- Word length
- Checksum option flag
- Acoustic
- Timeout period

These parameters are stored in the heap, and so they can be retreived with the heap manipulation commands. However, it would be much nicer if there were a set of pre-defined variables for these. These variables would need to have values assigned to them just before a program is run.

Configuration parameters should only be able to be referenced. If they could be modified, it is possibility that they might not be reset after the program. This would leave the SDR2 in a undesirable state, which is not desirable.

(b) Unit conversion functions.
Values are stored in the heap in standard units. It is these that the programmer must deal

with. With the present version of SDR-BASIC, the programmer must write user defined functions to convert between units. For example, to convert between degress Celcius and Fahrenheit,

```
DEF FNCtoF (degC) = degC*9/5 + 32
```

Functions to convert between these units and other commonly used ones will make customising input and output much easier.

(c) Angle input.
On the SDR2 angles can be entered in as decimal values or in a special degrees/minutes and seconds format (ddd.mmss). However, SDR-BASIC only has routines to input integers, reals and string. It is not a trivial task to provide angle input using the present features of SDR-BASIC. Therefore, introducing an angle input command would be extremely useful.

(d) Input/Output modifications.
In SDR-BASIC there is no way of telling if output or input was successfully completed. Knowing the status of i/o operations makes for more robust programs. A pre-defined variable could be introduced called STATUS. It could be updated after every input and output to reflect how successful the operation was.

In SDR-BASIC the input routines read in information until a new-line is reached. Often, the delimiter might be something different, such as a space or comma. Some way of specifying the delimiter would make input more flexible.

6.2.2. Improving the Translator.

The translator reads in a SDR-BASIC program and produces equivalent intermediate code. There are many ways in which this code can be improved.

(a) Adding range checking information.
The intermediate language has instructions designed for range checking. (see Appendix C). To keep the amout of code to a minimum, the translator does not make use of them. However, as inexperienced programmers will be using the translator, it might be desirable to add range checking.

(b) Compiler directives
Once a program has been written, tested and found to be without errors, the code size could be reduced by turning range checking off. To do this compiler directives could be introduced. These could form a part of an SDR-BASIC comment. For example

```
REM *norange
```

would turn range checking off, and

```
REM *range
```

would turn range checking on again.

Many other compiler directives could be included. Ones to control listings, code generation and case sensitivity are obvious candidates.

(c) Adding debugging information.
Inexperience users are likely to be faced with run-time errors from time to time. It is up to the implementor of the interpreter to provide meaningful error messages. When an error occurs, it will be very difficult to determine where it occured. For this reason, including dubugging information with the intermediate code is a good idea.

This can be achieved by adding information to the end of the data area (see section 4.2.4.). This information could consist of a table of pairs of address and line numbers. The addresses are where the code for the corresponding line number starts in the object code. When an error occurs, the interpreter can determine which line the error occurred.

(d) Warnings against bad programming practices.
BASICs tend to be very liberal when it comes to branching. It is possible to branch into a loop. Such practices often lead to stange errors occuring. To deter the programmer from doing this, warnings could be given at translation time.

(e) Optimisation.
As memory and speed requirements of the intermediate code are very important for this project, an attempt at optimisation would be worth while.

(f) The intermediate language could be improved.
Some attention to compactness was made when the intermediate language was designed. The end result was quite adaquate. However, more time could be spent fine-tuning. The result would be far more compact code.

All operands are 16 bit signed integers. Often only 8 bit integers are needed. Therefore, variable length operands could be used. For example, the instruction LoadConstInteger has one operand which is always in the range $-32768 \leq$ operand ≤ 32767. If the operand was in the range $0 \leq$ operand ≤ 255 only 8 bits are needed.

6.2.3. Improving the Interpreter.

The specifications for the interpreter in section 5 are for the translated code as it now stands. If changes were made elsewhere they would have to reflected in the interpreter also. There are also a few ways to improve the interactivity of the interpreter.

(a) Allocation of Variable Storage Space.
At present variable storage space is included as a part of every translated program. However, the variables only remain active during program execution. Before and after a program is run they are not used. To save on space, the code and interpreter could be modified so that the variables were allocated when a program starts running, and are deallocated when the program ends. In this way, at most one set of program variables occupy memory at one time.

(b) Debuging Facilities.
The suggestion of adding debugging information to the code was mentioned in 6.2.2. part c. The interpreter would be able to make use of this information, especially when run-time errors occur. Also, a trace facility could be introduced. The interpreter could process one line, display the line number and then wait for a key press. This would allow non-fatal errors to be pin-pointed more accurately.

(c) Loading in SDR-BASIC Programs.
At present, programs are loaded into memory as external programs. It would be nice if the interpreter could handle this input. It would help the interpreter keep track of which programs have been loaded into the SDR2. Also, a more flexible code file could be employed, which has a header to give useful information about the code.

6.2.4. Improving the Development Environment.

At the moment, programs will be written and translated on an IBM PC, and the intermediate code will then be loaded into the SDR2. Testing of the program will be done on the SDR2. If any modification is needed, the source on the IBM PC must be updated, retranslated, down loaded to the SDR2, and retested. Ideally, the turn around time between testing and modification should be kept to a minimum. The following will make for a better development environment.

(a) IBM PC testing phase.
An IBM PC-based interpreter could be implemented so that the whole testing phase could be done in the same environment. To do this, ADL and the SDR2 software would need to be ported to an IBM PC. This is a major task in itself.

An alternative is to write an interpreter in Pascal which would test the standard features of SDR-BASIC. Though not totally useful, it would give some assurance to how error-free a program is.

(b) Include a debugger in the SDR2-based interpreter.
This will help in the diagnosis of run-time errors.

Section Seven. Conclusion.

Implementing a BASIC translator for the SDR2 is a worthwhile endeavor both from a practical and theoretical point of view. There is a need for this software product in the surveying world. At present, users of the SDR2 electronic field book have to use a powerful yet inflexible computer. There are many opportunities that will open up when the surveyor can develop and use his or her own software.

Much has been learnt about the practical aspects of translator design. From designing the BASIC language, through to designing the intermediate code and the translator, many compromises have had to be made to take into account the environment that the final product will be used.

The end result of this project is that a translator for SDR-BASIC has been written. As the intermediate code interpreter has not yet been implemented, the code produced could not be tested. However, the author is confident that the code produced is satisfactory, because the code that is produced look sensible on close analysis.

There is still much work to be done before the translator will be ready to be distributed among the surveying community. Much of the ground work has now been done. The main thing left is to implement the interpreter that runs in the SDR2. This is surely a major project in itself.

Section Eight.
References.

---

[1] *SDR2 Electronic Field Book, Application Guide*

[2] S. Pemberton and M.C. Daniels, *Pascal Implementation, Compiler and Assembler/Interpreter*, Ellis Horwood 1982

[3] S. Pemberton and M.C. Daniels, *Pascal Implementation, The P4 Compiler*, Ellis Horwood 1982

[4] *MBASIC-86 Reference Manual*, Microsoft Corporation, 1982

[5] Aho, A.V. and Ullman, J.D., *The Principles of Compiler Design*, Addison Wesley 1977

Appendix A.
The SDR2

Datacom Software Research has developed software for a hand-held data recorder used by surveyors, called the SDR2 Electronic Field Book. The SDR2 collects and stores observations from survey instruments. A complete range of field data collection programs and calculation programs are provided so that observations can be checked and verified in the field. Observations are taken automatically from electronic survey instruments and can be entered via the keyboard for optical instruments. The stored data in the SDR2 can be transmitted to a variety of different types of data processors in a variety of ways.

A.1. The Hardware

The SDR2 hardware is based on the MSI/85 portable terminal. The MSI/85 is 9cm x 15cm x 4cm and weighs about 0.5kg. It can be held in one hand, as therefore is truely portable. Enhancements to the basic MSI model have been made. These include a keypad especially designed for the SDR2, and onboard EPROM containing the SDR2 software.

The SDR2 is based on an 8-bit microprocessor. The address bus is 16 bits wide, giving 64k of addressable memory. This is divided into 32k of ROM, and 32k of RAM. More recent versions of the SDR2 have up to 128k of RAM, which is achieved through page switching. The contents of memory is retained permenantly, even when the SDR2 is not in use.

The SDR2 has a 16 character liquid crystal display.

The keypad consists of 33 rubber keys. Included in these are numbers 0 through 9, decimal point, minus, arrow keys (for moving around through menu options and data records), on and off keys, shift, clear, enter, and 10 keys for accessing the various menus and programs. Alphabetic characters can be entered from the same keypad by using a shift key to toggle between numeric and alphabetic modes.

To be able to connect directly to electronic survey equipment the SDR2 has an in-built (15 pin, non-standard) interface, and comes with the appropriate cable. The cable has a special 6 pin connector to interface with survey equipment. The same connector can can be connected to a printer adaptor. This allows hard copies of the collected data to be made. With an optional communications adaptor (RS232) the SDR2 can be connected to a computer port using the same cable.

On top of these there is a built-in modulator or acoustic coupler. This can be used do download collected data via a telephone line to a remote computer. The acoustic coupler is also used to provide audio feedback on key depression.

A.2. The Software.

The software that comes with the SDR2 consists of data collection programs, calculation programs, searching programs that retrieve particular records from the data collected so far, and input/output programs.

Data collection programs automatically take an observation from electornic surveying equipment.

The calculation programs perform typical surveying calculations. They are:

Traverse:	Provides the field data collection procedures necessary for a transverse including side shots, coordinates and closure calculations.
Inverse:	Computes distances and angles between two known points.
Topography:	Automatic recording of observations, reduced data or coordinates for topographical surveys.
Resection:	Determines the coordinates of the instrument station from sets of observations to known points.
Remote elevation:	Calculation of the relative height of an object directly above or below a sighted target.
Collimation:	Determination of the instrument's collimation error for automatic correction of observations.
Slope reduction:	Calculation of the vertical and horizontal components of an observation.
Coordinates:	Calculation of the coordinates of a point.
Setting out:	Location of points in the field using known coordinates.
Keyboard input:	Input of known coordinates or directions.

The software in the SDR2 resides in the 32k of EPROM. It has been written in a combination of ADL and assembler.

ADL is very similar to Forth. ADL is stack based and the language syntax is in postfix notation. It is a 'threaded' language. There are a few primitives that are written in assembler. Higher level commands are defined in terms of these and other ADL commands. Input and output routines have been written in assembler to achieve a high level of performance.

ADL is implemented using tables. Every ADL instruction is represented by a one byte or two byte number. When an ADL program is interpreted, the bytes are used as indexes into these tables. The table contains the address of where the code for each instruction starts. This table driven approach causes ADL to be very compact, even more so than assembler.

As memory is a very important resource in the SDR2, any steps that can be taken to minimise its usage will be taken. Often, by cunning reshuffling of the numbers representing instrunctions many bytes can be saved. For example, if an instruction represented by two bytes was found to be used more frequently than an instruction represented by one byte, their representations might be interchanged. Because of this, the SDR2 has undergone many revisions over the years, and will undoubtably continue to do so in the future.

A.3. Data Strorage.

Data that has been collected is stored as records in a heap. The heap is partitioned sections. Each section consists of a collection of records pertaining to on particular surveying job or site. Each job is logically independent from all others. If more data is collect, or previous data is reviewed it id done so within the current job.

The records in the heap are variable length. Every record has many fields. The first two are always the same. They are:

```
Type Code
Derivation Code
```

The type code defines the type of the record. The derivation code specifies where the particular record was created.

The elements in the fields have varying types. For example, an observation record has:

Type code 09	(integer)
Derivation Code	(2 characters)
Source point number	(integer)
Target point number	(integer)
Slope distance	(real)
Vertical angle	(real)
Horizontal angle	(real)
Description	(16 characters)

A complete listing of the fields in each record types is given in the SDR2 Application Guide [1] on page 121.

All measurement values in the heap are stored in standard units. They are:

Measurement	Unit
Length	Metres
Angle	Degrees
Pressure	mm of mercury
Temperature	Degress Celcius.

It should be noted that other units can be used for input and output. These units just specify how the data is stored internally.

A.4. External Programs.

The SDR2's RAM is also used to store programs that have been loaded in. These programs are normally written in ADL. There are size restriction on external programs. Also, due to the table-driven nature of ADL there is a limitation on the total number of variables and functions used in the external program.

A.5. System Parameters.

The SDR2 has a set of parameters and configuration variables that give much flexibility to the system. They are explained in detail in the SDR2 Application Guide. [1].

(a) Instrument Parameters.
These parameters are needed to define the electronic surveying equipment being used. Included here are:

INSTRUMENT type
SERIAL NO of instrument
VERTICAL ANGLE, where the vertical angle is measured from.

(b) Measurement Units.
The unit types for measurements can be chosen from a set of options. For example

DISTANCE UNIT: metres or feet
PRESSURE UNIT: MmHg, Inch Hg or mbar
TEMPERATURE UNIT: Celcius or fahrenheit

(c) Corrections.
Allowances can be made for several corrections if desired. These correction include

SEA LEVEL CORRECTION
PRESSURE & TEMPERATURE CORRECTION

(d) Tolerances.
The tolerances for several readings can be specified.

VERTICAL ANGLE TOLERANCE
HORIZONTAL ANGLE TOLERANCE

(e) Input and Output Parameters.

The linespeed, format and mode of data transmission can be selected.

TRANSMISSION SPEED
PARITY SETTING
WORD LENGTH
CHECKSUM generation option
ACOUSTIC, use RS232 cable or integrate modulator

Appendix B.
SDR-BASIC.

SDR-BASIC is based on Microsoft BASIC [4]. Because of this, this section will only high light the areas where SDR-BASIC deviates radically from the Microsoft version.

B.1. A Brief Overview of SDR-BASIC.

(a) Line Format.
Program lines in SDR-BASIC have the following format (curly brackets idicate optionals):

```
{label} statement {: statement ...} <carriage return>
```

The programmer has the option of placing more than one statement on a line, but each statement must be separated from the last by a colon.

Labels are optional in SDR-BASIC. When they are used, they can appear only once, in any order. Labels must be an integer value in the range of 0 to 32767.

(b) Standard Features That Have Been Implemented.
Briefly, the following are supported:

Variable identifiers:	a letter followed by up to 15 alphanumeric characters.
Types:	integers, reals, and strings, plus multidimensional arrays in each of these types.
Default types:	All variables are assumed to be reals, unless: a) Followed by a "!" (real), "%" (integer) or "$" (string), b) Default type changed with DEFINT (integer), DEFSTR (string), or DEFSNG (single precision real). There are no allowances for double precision reals.
Assignment:	LET is optional. CLEAR sets all numeric values to 0, and strings to null ("").
Built-in Functions:	truncate (INT), absolute value (ABS), sine (SIN), cosine (COS), arctangent (ATN), square root (SQR), sign (SGN), $\log_e$ (LOG), e^x (EXP), substring (MID$, LEFT$, RIGHT$), character to integer conversion (ASC),

integer to character conversion (CHR$),
string length (LEN),
string to numeric conversion (VAL),
numeric to string conversion (STR$).

User-Defined Functions: The user can specify his or her own function, being any type and having any number of parameters of any type.
e.g. DEF FNarcsin (x) = ATN (x / SQR(-x *x+1))

Control Structures: branch to a label (GOTO),
conditional statements (IF..THEN..ELSE),
for-loops (FOR..NEXT),
subroutine calls (GOSUB..RETURN),
while loops (WHILE..WEND),
repeat loops (REPEAT..UNTIL),
computed goto and gosub (ON..GOTO../ON..GOSUB..),
program termination (END).

Input/Output: write to display (PRINT),
input from keyboard (INPUT),
user data (DATA),
read user data (READ),
reset data pointer (RESTORE).

Other: Comments (REM). The rest of the line is ignored.

(c) Deviations from Standard BASICs.

1. The DIM statement.

The size of a dimension must be expressed as an integer constant.

2. The INPUT Statement.

Only one variable can be input at one time,

3. The FOR..NEXT loop.

In most BASICs, ajoining NEXT statements can be elided. Consider,

```
FOR i% = 1 TO 10
FOR j% = 1 TO 10
NEXT j%
NEXT i%
```

Instead of the two nexts we could have

```
NEXT j%,i%
```

This is not supported by SDR-BASIC.

4. Strings.

In SDR-BASIC strings are any sequence of up to 31 characters, by default. For longer strings, the maximum length must be included in square brackets after the variable name. For example,

```
banana$ [128] = ""
```

defines the string `banana$` to be up to 128 charactes in length, and assigns it to have the value null.

(d) Enhancements To Standard BASICs

Several new features have been added to provide access to SDR-2 features.

Retreiving:	Retreive the last element added to the heap. (GETLAST) Retreive the record that was added after the last one referenced. (GETNEXT) Retreive the record that was added before last one referenced. (GETPREVIOUS)
Searching:	Search through the heap, and retrieve the last record of the given type.(GETTYPE (*type*)) Search through the heap and retreive the last record which contains the given point number (either as a source point or a target point). (GETPT (*point*))
Adding:	Add a record to the end of the heap. The records type is in stored in the variable RECORDTYPE. (ADD).
Output:	Output via the RS232 port. (LPRINT) Output via the acoustic coupler. (PRINT#)
Input:	Input via the RS232 port. (LINPUT) Input from surveying equipment. (INPUT#)

B.2. SDR-BASIC in BNF

A vertical bar (|) separates production options.
Braces ({}) are used to show optional symbols.
Non-terminal symbol appear inbetween < and >.
Quotes ("") are used to show terminal symbols which could be confused with other symbols.

```
<program>          ::= <lines>
<lines>            ::= <line> | <line> eoln <lines>
<line>             ::= { <label> } <statements>
<statements>       ::= <statement> | <statement> : <statements>
<statement>        ::= <assignment> | <control> | <io> | <declaration> |
                       <rem> | <enhancement> | <null>

<assignment>       ::= clear | {let} <identifier> = <expression>
<expression>       ::= ( <expression> ) |
                       <operand> <operator> <expression> |
                       <operand> |
                       <unary-op> <operand>
<operator>         ::= + | - | * | / | ^ | = | > | >= | "<" | "<=" |
                      "<>" | and | or
<operand>          ::= <literal> | <variable> | <function>
<literal>          ::= <string-literal> | <real-literal> |
                       <integer-literal>
<variable>         ::= <identifier> |
                       <identifier> ( <expr-list> )
<function>         ::= <funct-name> { ( <expr-list> ) }
<funct-name>       ::= <identifier> | <built-in>
<built-in>         ::= sgn | sqr | abs | int | sin | cos | tan |  atn |
                       mid$ | left$ | right$ | chr$ | asc | len
<expr-list>        ::= <expression> | <expression> , <expr-list>
<unary-op>         ::= + | - | not

<control>          ::= <if> | <for> | <goto> | <gosub> | <return> |
                       <on> | <end> | <while> | <repeat>
<if>               ::= if <expression> <then> <statements>
                       { else <statements> }
<then>             ::= then | ,
```

```
<for>           ::= for <identifier> = <expression> to <expression>
                    { step <expression> }
                    <separator> <body> <next>
<body>          ::= <statement> <separator> |
                     <statement> <separator> <body>
<separator>     ::= eoln | :
<next>          ::= next { <identifier> }
<on>            ::= on <expression> goto <label-list> |
                     on <expression> gosub <label-list>
<label-list>    ::= <label> | <label> , <label-list>
<goto>          ::= goto <label>
<gosub>         ::= gosub <label>
<return>        ::= return
<end>           ::= end
<while>         ::= while <expression> <separator> <body> wend
<repeat>        ::= repeat <separator> <body> until <expression>

<io>            ::= <read> | <print> | <input> | <data> | <restore>
<read>          ::= read <var-list>
<var-list>      ::= <variable> | <var-list> , <variable>
<print>         ::= print <print-list>
<print-list>    ::= <print-item> <print-list> | <null>
<print-item>    ::= <expression> | <io-sep>
<io-sep>        ::= ; | ,
<input>         ::= input { <string-literal> <io-sep> } <variable>
<data>          ::= data <data-list>
<data-list>     ::= <data-item> , <data-list> | <data-item>
<data-item>     ::= <literal> | <null>
<restore>       ::= restore { <label> }

<declaration>   ::= <typedef> | <funcdef> | <dim>
<dim>           ::= dim <dim-list>
<dim-list>      ::= <dim-item> | <dim-item> , <dim-list>
<dim-item>      ::= <identifier> ( <integer-literal> )
<typedef>       ::= <deftype> <range-list>
<deftype>       ::= defint | defstr | defsng
<range-list>    ::= <range> | <range> , <range-list>
<range>         ::= <char> | <char> - <char>
<funcdef>       ::= def <identifier> { ( <ident-list> ) } =
                     <expression>
```

```
<rem>           ::= rem <stuff>
<stuff>         ::= eoln | <char> <stuff>

<enhancement>   ::= add | getlast | getnext | getprevious |
                    gettype | getpt ( <expression> ) |
                    input# <variable> | print# <print-list> |
                    linput <variable> | lprint <print-list>
```

Appendix C.
The Intermediate Language.

The intermediate language is stack based. By this it is meant that operations are done on stack elements rather then between registers.

C.1. The Stack.

The stack consists of 16 bit elements, which can be considered to consist of any of five types.

(a) 16 bit signed integers integers.
(b) Addresses.
(c) Booleans. A false value is represented by a zero value. Anything else is true.
(d) Reals. A pointer on the stack points to where the real is stored.
(e) Strings. A pointer on the stack points to where the length of string plus string itself are stored.

A stack pointer points to the first empty element on the stack.

A frame pointer points to the last frame on the stack. A frame consists of a copy of a previous pointer, plus a return address. A frame pointers is needed to reference parameters that have been put on the stack.

C.2. An Overview of the Intermediate Language.

The intermediate language consists of the following instructions.

	Instruction Name	Operands Following	Stack State Before	Stack State After
(a) Dereferencing				
0.	LoadInteger		addr	integer stored at addr
1.	LoadReal		addr	real stored at addr
2.	LoadString		addr	string strored at addr
(b) Literals				
3.	LoadConstInteger	literal integer		integer
4.	LoadConstReal	literal real		real
5.	LoadConstString	literal string		string

NOTE: reals are stored in a 6 byte format.
NOTE: strings a re stored as a length byte followed a sequence of characters.

(c) Address of variable				
6.	VariableAddr	addr		addr

NOTE: this is the same as the instruction LoadConstInteger

(d) Indexing into arrays				
7.	ArrayIndex	factor	offset , addr	(offset * factor) + addr
(e) Range checking				
8.	CheckRange	int1	int2	integer, boolean (int2≤int1)
9.	CheckForFatal	int1	int2	integer

NOTE: if int2 ≤ int1 in CheckForFatal, a range error has occured and the interpretation

should terminate.
NOTE: These are not used by the translator at present.

(f) Storing values in variable storage

10.	StoreInteger		addr, integer
11.	StoreReal		addr, real
12.	StoreString	length	addr, string

NOTE: the length is the maximum length string can be. If string is longer than this, an error results.

(g) Clear variable storage area

13. Clear

(h) Arithmetic operators

14.	AddInteger	int1, int2	int1 + int2
15.	AddReal	real1, real2	real1 + real2
16.	NegInteger	integer	-integer
17.	NegReal	real	-real
18.	SubInteger	int1, int2	int1 - int2
19.	SubReal	real1, real2	real1 - real2
20.	MulInteger	int1, int2	int1 * int2
21.	MulReal	real1, real2	real1 * real2
22.	DivInteger	int1, int2	int1 div int2
23.	DivReal	real1, real2	real1 / real2
24.	Power	real1, real2	real1 ^ real2
25.	Catenate	str1, str2	str1 + str2

(i) Relational operators

26.	LTInteger	int1, int2	int1 < int2
27.	LTReal	real1, real2	real1 < real2
28.	LTString	str1, str2	str1 < str2
29.	LEInteger	int1, int2	int1 ≤ int2
30.	LEReal	real1, real2	real1 ≤ real2
31.	LEString	str1, str2	str1 ≤ str2
32.	EQInteger	int1, int2	int1 = int2
33.	EQReal	real1, real2	real1 = real2
34.	EQString	str1, str2	str1 = str2
35.	GTInteger	int1, int2	int1 > int2
36.	GTReal	real1, real2	real1 > real2
37.	GTString	str1, str2	str1 > str2
38.	GEInteger	int1, int2	int1 ≥ int2
39.	GEReal	real1, real2	real1 ≥ real2
40.	GEString	str1, str2	str1 ≥ str2
41.	NEInteger	int1, int2	int1 ≠ int2
42.	NEReal	real1, real2	real1 ≠ real2
43.	NEString	str1, str2	str1 ≠ str2

(j) Logical operators.

44.	LogicalAnd	int1, int2	int1 and int2
45.	LogicalOr	int1, int2	int1 or int2
46.	LogicalNot	integer	not integer

NOTE: in SDR-BASIC true values are represented by non-zero integers. False is tos represented by zero

(k) Mathematical functions

47.	SgnReal	real	sign of real
48.	SgnInteger	integer	sign of integer
49.	AbsReal	real	absolute value of real
50.	AbsInteger	integer	absolute value of integer
51.	Sine	real	sin real

52.	Cosine	real	cos real
53.	ArcTangent	real	arctan real
54.	Logarithm	real	$\log_e$ real
55.	Exponent	real	e^{real}
56.	SquareRoot	real	sqaure root of real

(l) String functions

57.	MidString	string, a, b	b chars starting at index a
58.	LeftString	string, a	leftmost a chars of string
59.	RightString	string, a	rightmost a chars of string
60.	ChrString	integer	char with ascii of integer
61.	AscString	string	ascii value of integer
62.	LenString	string	length of string

NOTE: These perform the same functions as their BASIC equivalents.

(m) Conversion functions

63.	CvtIR	integer	real
64.	CvtRI	real	integer
65.	CvtSR	string	real
66.	CvtSI	string	integer
67.	CvtIS	integer	string
68.	CvtRS	real	string
69.	CvtNtosIR	integer, ?	real, ?
70.	CvtNtosRI	real, ?	integer, ?

NOTE: 69 and 70 convert the value at the next to top of stack.
NOTE: 65-68 are not used by the translator.

(n) Function calls

71.	Call	addr		return address
72.	Return		addr	
73.	CallFunc	addr		fp, return address
74.	ReturnFunc	n	p1,..pn, fp,ra, result	result
75.	LoadParamInteger	offset		integer
76.	LoadParamReal	offset		real
77.	LoadParamString	offset		string

NOTE: On CallFunc the value frame pointer is loaded onto the stack, and the frame pointer is set to point to this cell. The return address is then placed onto the stack.
NOTE: On ReturnFunc, the stack frame must be removed, allong with the n parameters to the function.
NOTE: The offset in 75-77 is relative from the frame pointer. Ie, actual address of operand = fp - offset.

(o) Branching instructions

78.	BranchFalse	addr	integer	
79.	BranchTrue	addr	integer	
80.	Branch	addr		
81.	BranchIndexed	n, a1..an	index	
82.	CallIndexed	n, a1..an	index	return address

NOTE: The operands for 81 and 82 consists of a number followed by a block of n addresses. The value on the stack is used as an index into this table to determine the jump address. If index < 0 or index > n then the instruction does nothing.

(p) Looping constructs

83.	NextInteger	addr	addr1, addr2
84.	NextReal	addr	addr1, addr2
85.	Exit		

NOTE: addr1 is the address of the loop variable. Addr2 is the address of a block of data consisting of upper bound for loop and step value. Addr is the address of the start of the loop. The type of all values is the same as that of the instruction.

(q) Input and Output

86.	PrintInteger		integer
87.	PrintReal		real
88.	PrintString		string
89.	PrintControl		
90.	ClearScreen		
91.	InputInteger		addr
92.	InputReal		addr
93.	InputString	length	addr
94.	ReadInteger		addr
95.	ReadReal		addr
96.	ReadString	length	addr
97.	Restore	addr	

NOTE: PrintContol informs the interpreter that the next item to be printed will be on the same line as the last one.
NOTE: ClearScreen isn't used by the translator.
NOTE: length is the maximum string length that can be read in.
NOTE: addr is the address of where the result read in should be stored.

(r) Access to SDR2 features

98.	AddToHeap		
99.	GetLast		
100.	GetPrevious		
101.	GetType		
102.	GetPt		integer
103.	InputHashInteger		addr
104.	InputHashReal		addr
105.	InputHashString	length	addr
106.	PrintHashInteger		integer
107.	PrintHashReal		real
108.	PrintHashString		string
109.	LPrintInteger		integer
110.	LPrintReal		real
111.	LPrintString		string
112.	LInputInteger		addr
113.	LInputReal		addr
114.	LInputString	length	addr

NOTE: 98-102 perform the same function that their SDR-BASIC counterparts do.

(s) Predefined values and variables

115.	Null			null
116.	LoadPredefInteger	number		integer
117.	LoadPredefReal	number		real
118.	LoadPredefString	number		string
119.	StorePredefInteger	number	integer	
120.	StorePredefReal	number	real	
121.	StorePredefString	number	string	

NOTE: number is explained in section C.2.

C.3. Predefined Variables.

Instructions LoadPredefInteger, LoadPredefReal, LoadPredefString, StorePredefInteger, StorePredefReal and StorePredefString all have a single operand. This operand along with the type of the instruction identify the fields in heap records.

(a) Integers.

Identifying Number	Field Description	Participates in Record Types (See C.4.)
1	type code	All
2	serial number	Header
3	angle unit	Header
4	distance unit	Header
5	pressure unit	Header
6	temperature unit	Header
7	coord prompt option	Header
8	angles left right	Header
9	edm type	INSTR
10	edm serial number	INSTR
11	theodolite serial no	INSTR
12	mounting type	INSTR
13	vertical angle option	INSTR
14	point number	STN, POS
15	source point number	BKB, OBS, RED, SET
16	target point number	BKB, OBS, RED
17	count of observations	SET

(c) Reals

Identifying Number	Field Description	Participates in Record Types (See C.4.)
1	edm offset	INSTR
2	reflector offset	INSTR
3	prism constant	INSTR
4	northing	STN, POS
5	easting	STN, POS
6	elevation	STN, POS
7	theodolite height	STN
8	target height	TRGET
9	vertical collimation	COL
10	horizontal collimation	COL
11	pressure	ATMOS
12	temperature	ATMOS
13	scale factor	SCALE
14	azimuth	BKB, RED
15	horizontal observation	BKB, OBS
16	slope distance	OBS
17	vertical angle	OBS
18	horizontal distance	RED
19	vertical distance	RED

(c) Strings

Identifying Number	Field Description	Participates in Record Types (See C.4.)
1	derivation code	All
2	version number	Header
3	time and date	Header
4	edm description	INSTR
5	theodolite description	INSTR
6	station description	STN
7	description	POS, OBS, RED
8	job identifier	JOB
9	alphanumeric note	NOTE

C.4. Heap Record Types

The following heap record types exist at present on the SDR2.

Heap Record Type	Brief Description
Header	
INSTR	Instrument details
STN	Station details
TRGET	Target detail
COL	Instruments
ATMOS	Environment details
SCALE	Scaling factor
BKB	Back bearing details
POS	Coordinates
OBS	Observation
JOB	Job identifier
RED	Reduced Measurements
SET	Setting out details
NOTE	Alphanumeric note
EXT	Reserved for Future Extensions

Appendix D.
The SDR-BASIC translator.

D.1. Using the Translator.

(a) The program prompts the user for the name of the file that needs to be translated. If the the file does not exist, the program gives an error message and the program halts.

(b) As the translator is converting the SDR-BASIC program into intermediate code, the line being processed is displayed. Most errors are given near where the error occured. An exception is when a label that does not exist in the program is used by a GOTO, GOSUB or RESTORE.

After translation, the total number of errors is given.

(c) Several files are created by the translator during translation. If the program being translated is from a file called `test`, then the following result.

`test.err`
An error listing which contains all program lines with all errors.

`test.code`
The intermediate code generated is written to an ascii file in the SDR2 external program format.

`test.dump`
This is a textual form of the intermediate code version of the program. It is useful for testing the translator is working correctly.

D.2. An example of an error listing.

```
Error Listing For Program seive

   1  rem Prime Number Generator
   2  rem Generates Primes from 1 to 100 using Seive method
   3
   4     dim a%(100)
   5
   6     a%(1) = 1
   7
   8     for i% = 1 to 50
   9        if not a%(i%) goto 10
  10
  11        for j% = 2*i% to 50 step i%
  12           a%(i%) = 1
  13           next
  14
  15  10    next
  16
  17     print 2,
  18     for i% = 1 to 50
  19        if a%(i%) then print 2*i% + 1
  20        next
  21     end

*****      0  errors encountered
```

D.3. An example of the code listing.

The sample program shown in section D.2. generates the following code listing.

```
:200000006100B20300010600B4070002030010A06017E0300010A0601800300320A060198
:2000200082030010A06017E000600B4070002002E4E00375000690601840300020601 7E67
:2000400000140A0601860300320A06018806017E000A06017E000600B4070002030010A42
:200060000601840601865300520601 7E0601805300250300025659060 17E0300010A0601F1
:2000800080030032 0A0601820300010A06017E000600B4070002004E00A703000206017E43
:2000A00000140300010E5606017E06018053008C5555FFFF000000000000000000000031
:2000C00000000000000000000000000000000000000000000000000000000000000020
:2000E00000000000000000000000000000000000000000000000000000000000000000
:20010000000000000000000000000000000000000000000000000000000000000000DF
:20012000000000000000000000000000000000000000000000000000000000000000BF
:20014000000000000000000000000000000000000000000000000000000000000000 9F
:20016000000000000000000000000000000000000000000000000000000000000000 7F
:20018000000000000000000000000000000000000000000000000000000000000000 5F
```

D.4. An example of the dump listing.

```
The Intermediate Code Generated For seive

    0  Restore              178
    3  LoadConstInteger     1
    6  VariableAddr         180
    9  ArrayIndex           2
   12  LoadConstInteger     1
   15  StoreInteger
   16  VariableAddr         382
   19  LoadConstInteger     1
   22  StoreInteger
   23  VariableAddr         384
   26  LoadConstInteger     50
   29  StoreInteger
   30  VariableAddr         386
   33  LoadConstInteger     1
   36  StoreInteger
   37  VariableAddr         382
   40  LoadInteger
   41  VariableAddr         180
   44  ArrayIndex           2
   47  LoadInteger
   48  LogicalNot
   49  BranchFalse          55
   52  Branch               105
   55  VariableAddr         388
   58  LoadConstInteger     2
   61  VariableAddr         382
   64  LoadInteger
   65  MulInteger
   66  StoreInteger
   67  VariableAddr         390
   70  LoadConstInteger     50
   73  StoreInteger
   74  VariableAddr         392
   77  VariableAddr         382
   80  LoadInteger
   81  StoreInteger
   82  VariableAddr         382
   85  LoadInteger
   86  VariableAddr         180
   89  ArrayIndex           2
   92  LoadConstInteger     1
   95  StoreInteger
   96  VariableAddr         388
   99  VariableAddr         390
  102  NextInteger          82
  105  VariableAddr         382
  108  VariableAddr         384
  111  NextInteger          37
  114  LoadConstInteger     2
  117  PrintInteger
  118  PrintControl
  119  VariableAddr         382
  122  LoadConstInteger     1
  125  StoreInteger
  126  VariableAddr         384
```

```
129  LoadConstInteger    50
132  StoreInteger
133  VariableAddr        386
136  LoadConstInteger    1
139  StoreInteger
140  VariableAddr        382
143  LoadInteger
144  VariableAddr        180
147  ArrayIndex          2
150  LoadInteger
151  BranchFalse         167
154  LoadConstInteger    2
157  VariableAddr        382
160  LoadInteger
161  MulInteger
162  LoadConstInteger    1
165  AddInteger
166  PrintInteger
167  VariableAddr        382
170  VariableAddr        384
173  NextInteger         140
176  Exit
177  Exit
  :
178  * Start of Data Storage
  :
180  * Start of Integer Storage
  :
394  * Start of Real Storage
  :
394  * Start of String Storage
  :
394  * END OF CODE
```

D.5. The Translator Code.

The translator is written in Turbo pascal. It is based around a recursive descent parser. Error, scanning and code generation routines have been added. The program is structured as follows.

Declarations	350 lines
Error handling routines	70
Symbol table manipulation	70
Code generation routines	400
The Scanner	400
The Parser	1900
Initialisation of symbol table	350
Tidying up routines	280
TOTAL	3820 lines