1987 Honours Project

A BASIC Translator
Supervisor: P. J. Ashton

Department of Computer Science
University of Canterbury

Simon Dear
## Contents

1. Introduction
   1.1. The Aim of The Project 1
   1.2. A Brief Introduction to the SDR2 1

2. Design of SDR-BASIC
   2.1. SDR-BASIC Should Provide a Useful Subset of Common BASIC Features 3
   2.2. SDR-BASIC Should Reflect Its Compiled Nature 3
   2.3. Control of Memory Usage Is Required 4
   2.4. SDR-BASIC Should Take Into Account The Limitations Of The SDR2 Hardware 5
   2.5. SDR-BASIC Should Meet The SDR2 Requirements 5
      2.5.1 Access to the SDR2 Heap 5
      2.5.2 NULL fields in the heap 6
      2.5.3 Input Output Facilities 6

3. Design of the Intermediate Language
   3.1. The Style of Code Produced - ADL or Intermediate language? 8
   3.2. The Choice of the Intermediate Language 10
   3.3. The Intermediate Language Adopted 11

4. The SDR-BASIC Translator
   4.1. Issues in Translator Design 12
   4.2. Issues in Translator Implementation 13
      4.2.1. The Symbol Table 14
      4.2.2. The Scanner 15
      4.2.3. The Parser 16
         4.2.3.1 Recursive Descent Parsers 17
         4.2.3.2 Error Handling 18
         4.2.3.3 Type Checking 18
         4.2.3.4 Automatic Type Checking 19
      4.2.4. Code Generation 20
         4.2.4.1 The Code Produced 21
   4.3. Translator Output 22
   4.4. Issues in Porting the Macintosh Version to an IBM PC 22

5. The Interpreter 24

6. Results
   6.1. What has been done 27
   6.2. What more could be done 27
      6.2.1. Enhancing SDR-BASIC 27
         6.2.1.1 Adding More BASIC features 28
         6.2.1.2 Providing better interfacing with the SDR2 28
      6.2.2. Improving the Translator 29
      6.2.3. Improving the Interpreter 30
      6.2.4. Improving the Development Environment 31

7. Conclusion 32

8. References 33
## -- Appendices --

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. The SDR2</td>
<td>34</td>
</tr>
<tr>
<td>A.1. The Hardware</td>
<td>34</td>
</tr>
<tr>
<td>A.2. The Software</td>
<td>34</td>
</tr>
<tr>
<td>A.3. Data Storage</td>
<td>36</td>
</tr>
<tr>
<td>A.4. External Programs</td>
<td>37</td>
</tr>
<tr>
<td>A.5. System Parameters</td>
<td>37</td>
</tr>
<tr>
<td>B. SDR-BASIC</td>
<td>39</td>
</tr>
<tr>
<td>B.1. A Brief Overview of SDR-BASIC</td>
<td>39</td>
</tr>
<tr>
<td>B.2. SDR-BASIC in BNF</td>
<td>43</td>
</tr>
<tr>
<td>C. The Intermediate Language</td>
<td>45</td>
</tr>
<tr>
<td>C.1. The Stack</td>
<td>45</td>
</tr>
<tr>
<td>C.2. An Overview of the Intermediate Language</td>
<td>45</td>
</tr>
<tr>
<td>C.3. Predefined Variables</td>
<td>49</td>
</tr>
<tr>
<td>C.4. Heap Record Types</td>
<td>50</td>
</tr>
<tr>
<td>D. The SDR-BASIC translator</td>
<td>51</td>
</tr>
<tr>
<td>D.1. Using the Translator</td>
<td>51</td>
</tr>
<tr>
<td>D.2. An example of an error listing</td>
<td>52</td>
</tr>
<tr>
<td>D.3. An example of the code listing</td>
<td>52</td>
</tr>
<tr>
<td>D.4. An example of the dump listing</td>
<td>53</td>
</tr>
<tr>
<td>D.5. The Translator Code</td>
<td>54</td>
</tr>
</tbody>
</table>
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 to 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 account 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 language 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 downloaded 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 worthwhile 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 than 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 Useful 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 that 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,

\[ \text{fred}\{128\} = "" \]

sets the upper limit of the length of string \text{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 SDR2's Hardware.

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

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

\[ \text{INPUT } \{\text{optional prompt;}\} \text{ variable (..variable)} \]

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 SDR2's 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 retrieving. Also BASIC doesn't provide for any way of handling records. To get around this problem the following methods were considered:

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

(b) For retrieving, string and numeric values can be read successively. 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 retrieved 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 possibility 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 appearance 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:
- Retrve 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 retrieve 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 language. 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 language 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 straightforward, 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 undoubtedly 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.
Project Report

(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 existence. 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 would have to be changed. As there could be several versions of the interpreter in existence (one for each version of the SDR2 EPROM), each version would require modification.

With ADL code, the translator would require modification 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 existence 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 nasty 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 language 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 occurring.
However, because the intermediate language approach is easier to implement and possibly easier to maintain and port, it has been the one adopted.


There are many types of intermediate languages 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 interpreters 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 language 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:

\[
\text{FOR loop variable = initial value TO upper bound \{STEP increment\}}
\]

\[
\text{loop body}
\]

\[
\text{NEXT \{loop variable\}}
\]

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 language 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 than 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 account 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 always 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 external 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  
using Turbo 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](image)

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:

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

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 variables, 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:

\[
\text{index} = (\text{ordinal value of first character} \times 17 + \text{ordinal value of last character} \times 103 + \text{length of identifier} \times 55) \mod 503
\]

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.

![Diagram of the scanner process](image)

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 beginning 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

\[
\text{<program>} ::= \text{<lines>}
\]

\[
\text{<lines>} ::= \text{<line>} | \text{<line>} \text{eoln} \text{<lines>}
\]

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.

<table>
<thead>
<tr>
<th>operator</th>
<th>valid types</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary -</td>
<td>integer, real</td>
</tr>
<tr>
<td>+</td>
<td>integer, real, string</td>
</tr>
<tr>
<td>-</td>
<td>integer, real</td>
</tr>
<tr>
<td>*</td>
<td>integer, real</td>
</tr>
<tr>
<td>/</td>
<td>real</td>
</tr>
<tr>
<td>div</td>
<td>integer</td>
</tr>
<tr>
<td>^</td>
<td>real</td>
</tr>
<tr>
<td>&lt;, ≤, &gt;, ≥, ≠</td>
<td>integer, real, string</td>
</tr>
<tr>
<td>and, or, not</td>
<td>integer</td>
</tr>
</tbody>
</table>
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 occurred. If they are, then the two operands are coerced 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.

![Diagram](image)

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 variables (procedure CreateStringSpace)
- Create space for real variables (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 method, 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:

- **Initiation**
- **Code**
- **Data Area**
  - Integer Variables
  - Real Variables
  - String Variables

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 stored 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 because only one program's variables are active at one 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.

```
<table>
<thead>
<tr>
<th>Address MSB</th>
<th>Address LSB</th>
<th>00</th>
<th>Byte 0</th>
<th>Byte 31</th>
<th>Check-sum</th>
</tr>
</thead>
</table>
```

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 straightforward 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 highlighted.

(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,
\[ a! = b_1! \wedge b_2! \wedge b_3! \wedge b_4! \wedge \ldots b_n! \]

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 proceed 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 straightforward. 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.
4a) if the sign of the increment value is positive then:
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 if 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:

\[
\begin{array}{c|c}
\text{stack pointer} & \text{return address} \\
\hline
\text{frame pointer} & \text{Previous FP} \\
& \text{parameter } n \\
& \text{parameter 1} \\
\end{array}
\]

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.

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. Retrieving 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 spent 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 background 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.


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 \texttt{STRING$(string, n)$} which returns a string consisting of the given string duplicated \texttt{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 \texttt{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 which keys are being depressed or to wait until one key is pressed. Many BASICS provide an \texttt{INKEY$} or \texttt{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 programs 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 configuration 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 retrieved 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 degrees Celsius and Fahrenheit,

\[ \text{DEF } \text{FNCtoF (degC)} = \text{degC} \times \frac{9}{5} + 32 \]

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 amount 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

\[ \text{REM } *\text{norange} \]

would turn range checking off, and

\[ \text{REM } *\text{range} \]

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 occurred. For this reason, including debugging 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 strange errors occurring. 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 adequate. 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 ≤ operand ≤ 32767. If the operand was in the range 0 ≤ 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 be 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 pinpointed 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, downloaded 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*


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 truly 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 permanently, 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 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 electronic 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 instructions 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 Storage.

Data that has been collected is stored as records in a heap. The heap is partitioned into sections, each of which contains records pertaining to a particular surveying job or site. Each job is logically independent from all others. If more data is collected, or previous data is reviewed, it is 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 type 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**
  - **Degrees 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 restrictions 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 corrections 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 highlight 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 indicate 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 BASICSs.
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 BASICSs, adjoining NEXT statements can be elided. Consider,

\[
\begin{align*}
\text{FOR } i & \text{% } = 1 \text{ } \text{TO } 10 \\
\text{FOR } j & \text{% } = 1 \text{ } \text{TO } 10 \\
\text{NEXT } j & \% \\
\text{NEXT } i & \%
\end{align*}
\]

Instead of the two nexts we could have

\[
\text{NEXT } j, i
\]

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,

\[
\text{banana$ [128] = "}\]

defines the string \text{banana$} to be up to 128 characters 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.

Retrieving:
- Retrieve the last element added to the heap. (GETLAST)
- Retrieve the record that was added after the last one referenced. (GETNEXT)
- Retrieve 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 retrieve 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#)
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.

```plaintext
<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>:
<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>
<then> ::= then | ,
```
The grammar for a BASIC translator:

\[
\begin{align*}
\langle \text{for} \rangle &::= \text{for } \langle \text{identifier} \rangle = \langle \text{expression} \rangle \text{ to } \langle \text{expression} \rangle \\
&\quad \{ \text{ step } \langle \text{expression} \rangle \} \\
&\quad \langle \text{separator} \rangle \langle \text{body} \rangle \langle \text{next} \rangle \\
\langle \text{body} \rangle &::= \langle \text{statement} \rangle \langle \text{separator} \rangle | \\
&\quad \langle \text{statement} \rangle \langle \text{separator} \rangle \langle \text{body} \rangle \\
\langle \text{separator} \rangle &::= \text{eoln} | : \\
\langle \text{next} \rangle &::= \text{next } \langle \text{identifier} \rangle \\
\langle \text{on} \rangle &::= \text{on } \langle \text{expression} \rangle \text{ goto } \langle \text{label-list} \rangle | \\
&\quad \text{on } \langle \text{expression} \rangle \text{ gosub } \langle \text{label-list} \rangle \\
\langle \text{label-list} \rangle &::= \langle \text{label} \rangle | \langle \text{label} \rangle , \langle \text{label-list} \rangle \\
\langle \text{goto} \rangle &::= \text{goto } \langle \text{label} \rangle \\
\langle \text{gosub} \rangle &::= \text{gosub } \langle \text{label} \rangle \\
\langle \text{return} \rangle &::= \text{return} \\
\langle \text{end} \rangle &::= \text{end} \\
\langle \text{while} \rangle &::= \text{while } \langle \text{expression} \rangle \text{ seperator } \langle \text{body} \rangle \text{ wend} \\
\langle \text{repeat} \rangle &::= \text{repeat } \langle \text{separator} \rangle \langle \text{body} \rangle \text{ until } \langle \text{expression} \rangle \\
\langle \text{io} \rangle &::= \langle \text{read} \rangle | \langle \text{print} \rangle | \langle \text{input} \rangle | \langle \text{data} \rangle | \langle \text{restore} \rangle \\
\langle \text{read} \rangle &::= \text{read } \langle \text{var-list} \rangle \\
\langle \text{var-list} \rangle &::= \langle \text{variable} \rangle | \langle \text{var-list} \rangle , \langle \text{variable} \rangle \\
\langle \text{print} \rangle &::= \text{print } \langle \text{print-list} \rangle \\
\langle \text{print-list} \rangle &::= \langle \text{print-item} \rangle \langle \text{print-list} \rangle | \langle \text{null} \rangle \\
\langle \text{print-item} \rangle &::= \langle \text{expression} \rangle | \langle \text{io-sep} \rangle \\
\langle \text{io-sep} \rangle &::= ; | , \\
\langle \text{input} \rangle &::= \text{input } \{ \langle \text{string-literal} \rangle \langle \text{io-sep} \rangle \} \langle \text{variable} \rangle \\
\langle \text{data} \rangle &::= \text{data } \langle \text{data-list} \rangle \\
\langle \text{data-list} \rangle &::= \langle \text{data-item} \rangle , \langle \text{data-list} \rangle | \langle \text{data-item} \rangle \\
\langle \text{data-item} \rangle &::= \langle \text{literal} \rangle | \langle \text{null} \rangle \\
\langle \text{restore} \rangle &::= \text{restore } \{ \langle \text{label} \rangle \} \\
\langle \text{declaration} \rangle &::= \langle \text{typedef} \rangle | \langle \text{funcdef} \rangle | \langle \text{dim} \rangle \\
\langle \text{dim} \rangle &::= \text{dim } \langle \text{dim-list} \rangle \\
\langle \text{dim-list} \rangle &::= \langle \text{dim-item} \rangle | \langle \text{dim-item} \rangle , \langle \text{dim-list} \rangle \\
\langle \text{dim-item} \rangle &::= \langle \text{identifier} \rangle \{ \langle \text{integer-literal} \rangle \} \\
\langle \text{typedef} \rangle &::= \langle \text{deftype} \rangle \langle \text{range-list} \rangle \\
\langle \text{deftype} \rangle &::= \text{defint } | \text{defstr } | \text{defsng} \\
\langle \text{range-list} \rangle &::= \langle \text{range} \rangle | \langle \text{range} \rangle , \langle \text{range-list} \rangle \\
\langle \text{range} \rangle &::= \langle \text{char} \rangle | \langle \text{char} \rangle - \langle \text{char} \rangle \\
\langle \text{funcdef} \rangle &::= \text{def } \langle \text{identifier} \rangle \{ \{ \langle \text{ident-list} \rangle \} \} = \\
&\quad \langle \text{expression} \rangle
\end{align*}
\]
A BASIC translator

<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>
The intermediate language is stack based. By this it is meant that operations are done on stack elements rather than 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.
(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 pointer is needed to reference parameters that have been put on the stack.


The intermediate language consists of the following instructions.

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>Operands</th>
<th>Stack State Before</th>
<th>Stack State After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dereferencing</td>
<td></td>
<td>addr</td>
<td>integer stored at addr</td>
</tr>
<tr>
<td>0. LoadInteger</td>
<td></td>
<td>addr</td>
<td>real stored at addr</td>
</tr>
<tr>
<td>1. LoadReal</td>
<td></td>
<td>addr</td>
<td>string stored at addr</td>
</tr>
<tr>
<td>2. LoadString</td>
<td></td>
<td>addr</td>
<td></td>
</tr>
<tr>
<td>Literals</td>
<td></td>
<td>literal integer</td>
<td>integer</td>
</tr>
<tr>
<td>3. LoadConstInteger</td>
<td>literal</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>4. LoadConstReal</td>
<td>literal</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>5. LoadConstString</td>
<td>literal</td>
<td>string</td>
<td></td>
</tr>
<tr>
<td>NOTE: reals are stored in a 6 byte format.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: strings are stored as a length byte followed a sequence of characters.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address of variable</td>
<td></td>
<td>addr</td>
<td>addr</td>
</tr>
<tr>
<td>6. VariableAddr</td>
<td></td>
<td>addr</td>
<td></td>
</tr>
<tr>
<td>NOTE: this is the same as the instruction LoadConstInteger</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexing into arrays</td>
<td></td>
<td>factor offset, addr</td>
<td>(offset * factor) + addr</td>
</tr>
<tr>
<td>7. ArrayIndex</td>
<td>factor</td>
<td>offset, addr</td>
<td></td>
</tr>
<tr>
<td>Range checking</td>
<td>int1</td>
<td>int2</td>
<td>integer, boolean (int2≤int1)</td>
</tr>
<tr>
<td>8. CheckRange</td>
<td>int1</td>
<td>int2</td>
<td></td>
</tr>
<tr>
<td>9. CheckForFatal</td>
<td>int1</td>
<td>int2</td>
<td>integer</td>
</tr>
<tr>
<td>NOTE: if int2 ≤ int1 in CheckForFatal, a range error has occurred and the interpretation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page 45
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. LTRelal real1, real2 real1 < real2
28. LSTString str1, str2 str1 < str2
29. LTEInteger int1, int2 int1 ≤ int2
30. LEREal real1, real2 real1 ≤ real2
31. LESString str1, str2 str1 ≤ str2
32. LEInteger int1, int2 int1 = int2
33. BQReal real1, real2 real1 = real2
34. EQString str1, str2 str1 = str2
35. GTInteger int1, int2 int1 > int2
36. GTRelal real1, real2 real1 > real2
37. GTSString str1, str2 str1 > str2
38. GEEInteger 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. NESString 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

Page 46
52. Cosine real \( \cos \) real
53. ArcTangent real \( \arctan \) real
54. Logarithm real \( \log \) real
55. Exponent real \( e \) real
56. SquareRoot real square 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
73. CallFunc addr fp, return address
74. ReturnFunc n pl, .. 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, along with the n parameters to
the function.
NOTE: The offset in 75-77 is relative from the frame pointer. I.e., 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: PrintControl 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.

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

(c) Reals

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

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

**C.4. Heap Record Types**

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

<table>
<thead>
<tr>
<th>Heap Record Type</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Instrument details</td>
</tr>
<tr>
<td>INSTR</td>
<td>Station details</td>
</tr>
<tr>
<td>STN</td>
<td>Target detail</td>
</tr>
<tr>
<td>TARGET</td>
<td>Instruments</td>
</tr>
<tr>
<td>COL</td>
<td>Environment details</td>
</tr>
<tr>
<td>ATMOS</td>
<td>Scaling factor</td>
</tr>
<tr>
<td>SCALE</td>
<td>Back bearing details</td>
</tr>
<tr>
<td>BKB</td>
<td>Coordinates</td>
</tr>
<tr>
<td>OBS</td>
<td>Observation</td>
</tr>
<tr>
<td>JOB</td>
<td>Job identifier</td>
</tr>
<tr>
<td>RED</td>
<td>Reduced Measurements</td>
</tr>
<tr>
<td>SET</td>
<td>Setting out details</td>
</tr>
<tr>
<td>NOTE</td>
<td>Alphanumeric note</td>
</tr>
<tr>
<td>EXT</td>
<td>Reserved for Future Extensions</td>
</tr>
</tbody>
</table>
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 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 occurred. 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.

\texttt{test.err}
An error listing which contains all program lines with all errors.

\texttt{test.code}
The intermediate code generated is written to an ascii file in the SDR2 external program format.

\texttt{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 sieve

1 rem Prime Number Generator
2 rem Generates Primes from 1 to 100 using Sieve 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      for j% = 2*i% to 50 step i%
11          a%(i%) = 1
12          next
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.

```
:200000006100B20300010600B40700020300010A06017E0300010A0601803000206017E67
:20004000001A0601860300320A0601880E000600B40700020300010E42
:2000600006018406018653005206017E060180530250300256590617E030001A0601F1
:20008000800300320A0601820300010A06017E000600B4070002004E00A70300206017E43
:2000A00001403001085606017E0601805308C5555PPFF0000000000000000000000000031
:2000C000000000000000000000000000000000000000000000000000020
:2000E500000000000000000000000000000000000000000000000000000008F
:200100000000000000000000000000000000000000000000000000000000000007F
:200140000000000000000000000000000000000000000000000000000000000005F
```
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 Mullnteger
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
```
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.

Déclarations: 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