2920 SIGNAL PROCESSING APPLICATIONS COMPILER USER'S GUIDE

Manual Order No. 121529-001 Rev. A

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

i	iSBC	Multimodule
ICE	Library Manager	PROMPT
iCS	MCS	Promware
Insite	Megachassis	RMX
Intel	Micromap	UPI
Intelevision	Multibus	μScope
Intellec		

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.

PREFACE AND READER'S GUIDE

The audience for this manual is engineers who design products for signal processing.

The purpose of this manual is to familiarize you with the features of this Compiler such that with a few hours of reading and practice you can feel comfortable using it to design electrical filters and other applications, and generate 2920 programs for them.

The Compiler can be used to generate 2920 assembly language code for a wide variety of signal processing applications. In the present manual, electrical filters are a central focus of discussion. A brief review of digital and analog filter design appears in Appendix H. The formulas used by the Compiler for computing the frequency responses of the filter appear in Appendix I.

The primary purpose of this Compiler is to make it easy to specify the frequency response of a desired digital filter and implement the filter in AS2920 assembly language code. It enables you to

- specify and modify design parameters, e.g., set sampling rate (Chapter 3), place poles and zeros at chosen S-plane or Z-plane coordinates (Chapter 4), and set error bounds on gain (Chapter 5);
- see immediately on a graph or list the frequency or time response of existing poles and zeros at specified frequencies, or display any design parameter (Chapters 5, 6);
- generate, store, and review AS2920 assembly language code for each filter or stage, subject to constraints you specify on error functions and program size, after response appears appropriate to your design (Chapter 7);
- use diskette files as scratchpads to store, review, modify, and retrieve files of parameters, code, or commentary (Chapter 8);
- create your own name for command sequences (macros) to facilitate your interactive design sessions and unattended test sessions (Chapters 9 through 11).

  This feature greatly multiplies the power of the Compiler because it enables you to use additional transforms, constraints, and procedures in any of your subsequent design sessions, after defining them once.

The minimum hardware configuration for the SPAC20 Compiler is as follows:

- INTELLEC or INTELLEC-II with 64K bytes of random access memory (RAM),
- Teletypewriter, CRT, or equivalent for console input and output,
- One diskette drive unit, single or double density.

If a line printer is available, it can be used for large-volume or hard-copy output, including graphics. If the system includes an iSBC-310 math board the SPAC20 module will use it to speed computations significantly.

The SPAC20 Compiler is designed to be used in conjunction with other Intel products, most clearly the AS2920 Assembler and SM2920 Simulator. Further, the Compiler uses the ISIS-II keyboard and disk input/output functions. You may wish

to refer to other documents containing valuable information about the supervisor, the Intellec system, and the other software (e.g., the Editor) used on it. These include:

- *INTELLEC Operator's Manual* 9800129
- *INTELLEC/DOS Diskette Operating System Operator's Manual* 9800206
- *ISIS-II System User's Guide* 9800306
- *2920 Assembly Language Manual* 9800987
- *2920 Simulator User's Guide* 9800988
- *ISIS-II CREDIT User's Guide (CRT-Based Text Editor)* 9800902
- *Intellec Microcomputer Development System Hardware Reference Manual* 9800132
- *iSBC-310 High Speed Mathematics Unit Hardware Reference Manual* 9800410

Digital signal processing is an extensive subject. This manual assumes the reader has at least a rudimentary knowledge of it, and does not attempt more than a review at that level. A more detailed understanding can be obtained through the study of texts written for that purpose, e.g.,

Digital Signal Processing
by A. V. Oppenheim and R. W. Schafer, Prentice Hall, Inc. 1975

Digital Signal Processing
by Abraham Peled and Bede Liu, John Wiley, 1976

Theory and Application of Digital Signal Processing
by L. R. Rabiner and B. Gold, Prentice Hall, Inc. 1975

After first scanning Chapter 1 of this manual, you might next scan areas of interest among Chapters 3 through 8. Although Chapter 2 is necessary for a full definition and understanding of the SPAC20 Compiler, its technical detail can perhaps be better absorbed later, after you build some general familiarity with how you might use the Compiler commands to achieve your purposes. It is placed second because its contents are later taken for granted. This may not trouble you much if you've used compilers before or if you use the index to fill in the gaps as you go. Appendices A, B, and E also help tie things together by supplying explanations, tables, and charts about the language.

Chapters 9 through 11 assume you have mastered the other chapters. Compound commands are very powerful and useful, but they might be confusing in your first pass through the book.

This Compiler differs from many others in that it is to be used in the process of interactive design, rather than after the design is complete. In addition, it produces partial rather than complete code. Further editing of its output code is required by considerations of analog-to-digital conversion, signal scaling, and code compaction, e.g., the merging of instructions for a pole with those for a zero (see Appendices G and J).

There is a certain symmetry about the commands which may help you learn and remember them. You can display nearly any object (e.g., one or all poles, gain, bounds) by typing its name followed by a carriage return. You can change nearly any object by typing its name followed by an equal sign and the new desired value.

The commands fall into functional categories, and their syntax is relatively uniform. For example, nearly all the file commands use the same set of objects or modifiers. Similarly, all the pole and zero commands use one set of objects or modifiers, and all the graph commands use one such set. This is most clearly shown by the syntax charts in Appendix E.

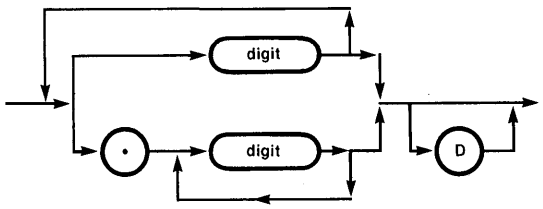
The syntax discussion in Chapter 2 is deliberately simplified and incomplete, because the full range of expressions is rarely relevant until you have mastered the simple commands, and moved on to learn the power of compound commands and macros in Chapters 9 through 11.

The Compiler's matched-Z transform is not always the function of choice for mapping S-plane poles and zeros to the Z-plane, although one of its advantages is easy movement from S to Z or Z to S-planes. Once macros are understood, it becomes a simple matter to create alternate transform(s), such as bilinear, and thereafter use them as desired. The discussions prior to Chapter 9 assume the use of the built-in matched-Z transform without further comment. The review in Appendix H contains a discussion of various other transforms.

Notation of this Manual

Most of the constructs and commands in this book are set forth using syntax charts, a pictorial representation of how the building blocks of the SPAC20 commands are legally combined.

These charts always begin at the left with an arrow that leads to an elliptical box or a set of such boxes, indicating necessary or optional parts of the construct. The charts always end with a single arrow off to the right. You create legal constructs by following the arrows within these graphs, looping back to repeat some element only if there is an arrowed line showing that you may do so. When a choice must be made among several options, a single arrow will lead to a vertical bar, from which multiple arrows permit you to go along the path containing the option you want. Here are some examples:

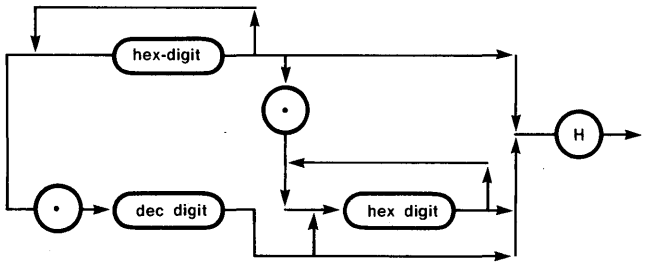


unsigned decimal constant 121533-06

where digit means any one of the set 0,1,2,3,4,5,6,7,8,9.

This says that to construct an unsigned decimal constant for use in some command, you must start with either a digit or a decimal point. If you start with a digit, you may continue to place digits after it (up to the limit of 31 digits), or you may choose the other branch at any time, placing a decimal point next, or you may quit, exiting the chart at the right. As you exit the chart you have the choice of appending a D after your digits, explicitly indicating this is a decimal number rather than a binary or hexadecimal number. This is optional because decimal is assumed whenever no suffix is present.

If you chose to insert a decimal point, you must now enter at least one digit before exiting. You may add as many as 31 (less the number of digits already entered), but you may not enter another decimal point—the arrowed lines offer no path to it.



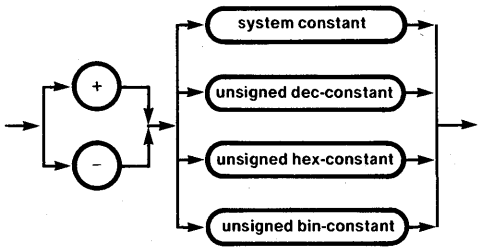
unsigned hexadecimal constant 121533-06

where hex-digit is any one of the set 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Dec-digit, like digit above, includes only the first ten of these.

This chart shows that to create a hexadecimal constant you may begin with a decimal point or a hex-digit. If you start with a hex-digit, you may continue appending hex-digits or place a decimal point next, after which you may again append hex-digits until you choose to exit the chart on the right, where you are required to append the H suffix labeling this a hexadecimal constant.

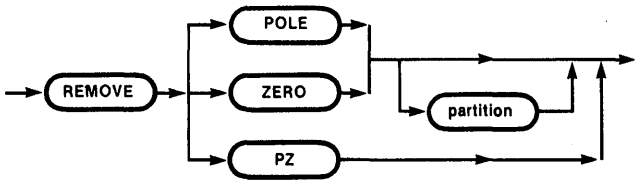
If your constant is a fraction only, i.e., has no digit to the left of the decimal point, then the digit immediately after the decimal point must be a decimal digit, i.e., not a letter. Once you have begun with "point dec-digit", succeeding digits may be any legal hex-digit. Exiting the chart is as above. The reasons for the restrictions embodied in the chart are explained in Chapter 2.

Syntax charts often contain the names of other syntax charts, indicated by lowercase words as opposed to the capitals used for keywords. The charts are combined simply by substituting the full chart of the named item in place of the name. As an example, the chart for "numeric constant" includes by reference the two charts above:



numeric constant 121533-06

To give one command example, the remove command for poles and zeros has the following chart:



The REMOVE Command for Poles and Zeros 121533-18

indicating you must enter the word REMOVE and one of the three object words POLE, ZERO, or PZ. With that, you're done unless you wish to use a partition, which is indicated to be optional by being off the required exit line. (The chart for partition appears in Chapter 2. This command is explained in Chapter 4.)

CONTENTS

ILLUSTRATIONS

TABLES

CHAPTER 1
INTRODUCTION

Concepts of Filter Design

Designing a filter involves choosing operations to perform on input signals in order to produce modified signals as output. These operations are usually linear. The theory relating continuous analog filters to sampled digital filters is reviewed in Appendix H. Digital filters are covered in great detail by the books on digital signal theory in the bibliography.

Filters are usually designed to achieve certain gain and phase characteristics. These can be viewed as implementing a mathematical transfer function, whose factors relate directly to the desired attributes. Factors which represent complex frequencies at which the gain is zero are called the zeros of this filter. Factors representing frequencies at which the gain grows indeterminately large are called the poles of the filter.

These poles and zeros are complex numbers of the form a + bj, where $j = \sqrt{-1}$. Thus a filter's poles and zeros can be specified on a complex-valued graph, such as the S-plane or the Z-plane.

The desired output amplitude and phase can now be approached in an interactive design session by placing poles and zeros in the S-plane (for continuous filters) or the Z-plane (for sampled filters), and viewing the resultant output. Moving these poles and zeros can then change that output to more closely approximate what is needed. The compiler capabilities facilitate this interactive process of specification, modification, and review by providing simple commands and graphs for these functions.

Introduction to the Compiler

The SPAC20 Signal Processing Applications Compiler accepts high-level (English-like) language input and produces 2920 Assembly language code. The Compiler is also a filter design aid which permits substantial interactive manipulation of a wide variety of parameters and constraints, both in design of filter stages and in optimization of code size and/or error limits.

The two principal functions of the SPAC20 Compiler are:

1. to make it easy for you to specify, alter, and review design parameters for your signal processing application, and

2. to save your writing all the detailed steps required to implement the necessary functions in assembly language code.

One example is specifying the frequency response of a desired digital filter, checking gain, phase, or other aspects and changing parameters as needed. You then implement the filter by issuing one CODE command for each pole or zero, thus generating the AS2920 code.

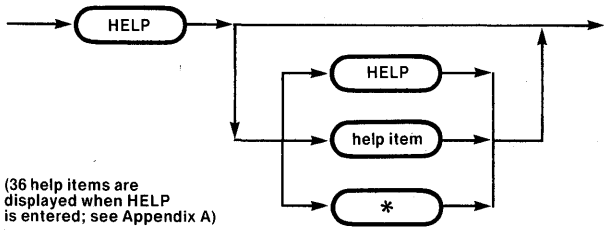
NOTE

The specific functions and features of the Intel 2920 processor are described in the data sheet for that device. The AS2920 Assembler converts the user-edited output of this Compiler into object code that will directly run on an Intel 2920 processor. The assembler is described in the manual entitled *2920 Assembly Language Manual*, order number 9800987.

Once a program has been converted to object code, it can be tested using another Intel product, called the SM2920 Simulator. This product is described in the *2920 Simulator User's Guide*, order number 9800988. Many features of this Compiler show a family resemblance to the Simulator, and to Intel's In-Circuit-Emulator-86 (ICE-86).

This resemblance includes the use of compound commands, which are featured in the Intel product which handles multiple In-Circuit-Emulators, called multi-ICE. It is described in manual number 9800762 entitled *Mutli-ICE Operating Instructions for ISIS-II Users.*

The Help Messages



HELP Command 121533-27

The SPAC20 Compiler has a help message facility, which can inform or remind you of the form, function, or interrelationships of compiler keywords, concepts, and commands. These messages can be requested any time no other command is being executed, i.e., whenever the compiler has issued its prompt character (asterisk):

```
*HELP
```

Typing in this word will cause an index of all help messages available to be displayed on the console. If you enter the following sequence

```
*LIST :LP:
*HELP *
```

then all the messages will be printed out on the line printer. (They appear in Appendix A in this form.)

You can get the help message for only specific items by typing the item name after HELP, e.g.

```
*HELP DEFINE
```

will cause the help message on the define command to be displayed.

Requesting help on scales or normalization or bounds will get you a description of certain system variables and how to access them, e.g., FSCALE, GREF, BOUNDS, etc. Similarly, requesting help on change or display can remind you of how the different forms or references can be used.

Probable Sequence of Use

1. Define or include symbolic constants for easy use; define sample interval TS;
2. a. define poles and zeros, move or remove them as desired in TS plane, that is, the sampled S-plane (see below); implement desired filter as if continuous;
   b. define poles and zeros, move or remove as desired in Z-plane to implement desired filter;
   c. use a macro, e.g., Chebyshev or Bilinear, to locate poles and zeros for a particular filter;
3. set bounds to be able to determine closeness to desired spec and to guide code generation; look at MSQE , GERROR, etc. to see closeness;
4. graph filter response and other filter behavior;
5. if the filter is not yet "good enough", return to step 2;
6. save current poles and zeros in diskette file via put or append command;
7. code the (next) pole or zero with either gain bound constraint or other constraint; examine resulting filter responses;
8. if not satisfactory, restore poles and zeros and return to step 7 to use a different constraint, or return to step 1 to begin anew;
9. if satisfactory, put or append code;
10. if there is another pole or zero, return to step 7 for the next one;
11. if all poles and zeros are coded, you now have final filter;
12. determine scaling; for example, save coded poles and zeros; delete all but the first n; MAGAIN reveals scale for $(n+1)^{th}$ stage; append comment for code file; restore earlier poles and zeros; repeat. Alternatively, can delete poles or zeros in reverse order one by one;
13. as needed, generate code for other functions, using code, macro, and compound commands, e.g., sawtooth wave generator, and append to code file;
14. exit; edit code file to insert additional scaling if needed (per code comments), add I/O, end statement; assemble with AS2920; simulate with SM2920; program the EPROM in the 2920 chip.

Defining a Filter

You can define a filter by specifying the location of poles and zeros in the S-plane or the Z-plane corresponding to a sample rate which you supply. Complex poles always occur as conjugate pole pairs so that the filter is realizable. Poles and zeros can be created, moved, or removed. Continuous poles and zeros in the S-plane describe portions of the total filter which will be implemented outside of the 2920 chip.

Poles and zeros can be specified in the S-plane and then designated as sampled. During calculation, these poles and zeros are mapped to the Z-plane using the matched-Z transform, i.e., a pole or zero at x+jy on the S-plane is transferred to a pole or zero at $e^{2\pi TS(x+jy)}$ on the Z-plane, where TS represents the sample interval in the sampled S-plane. In polar coordinates, this Z-plane location is ($e^{2\pi TSx}$, $2\pi TSy$)

This transform allows an analog design engineer who is not already familiar with digital filter theory to work in more familiar territory. For low frequencies, relative to the sample rate, it does offer a relatively faithful translation of analog filters to digital. However, the matched-Z transform is not ideal in many cases, and therefore some users prefer to work directly in the Z-plane.

Displaying the Response

You can examine the frequency response of your filter as you manipulate the position of poles and zeros. The gain, phase, group delay, and time responses can be graphed or listed. You can specify the frequency range of interest for these outputs. While emphasis is placed on the gain versus frequency response of the filter, you can take advantage of the compound command capability described in Chapter 9 to use the phase, impulse, or step responses.

The graphs do not require the console device to have any cursor controls; e.g., the ability to move the beam directly by pressing a button for up, down, left, or right. You specify the X-axis and the Y-axis ranges. The last curve plotted is always available for redisplay upon entering the command GRAPH, regardless of the effect of intervening commands. It is also possible to superimpose the last curve plotted and a new curve, regardless of intervening scale changes. The graphs can be sent to a diskette file, or hard copies can be produced on a line printer, since no special cursor control characters are assumed.

Generating Code

Once the filter responses (e.g., gain, error, phase) look adequate to meet your design specifications, you can generate the code for each pole or zero with a single command. The Compiler enables you to implement the filter as a cascaded series of first and second order stages. You can also generate code to compute independent variables of the form Y=C*X or Y=C*X+Y or Y=C*Y, where C is a constant, and X and Y are variable names. These are useful for propagation and scaling of the digital signal between stages.

These modules of assembly language code can be accumulated into a file to be used as input to the 2920 Assembler. Prior to submitting the code to the Assembler, you will need to do some editing to implement the analog-to-digital and digital-to-analog functions, and also the above propagation and scaling. (See the ISIS-II or Editor manuals for editing instructions.) During this editing it is also possible to compact the program by merging the code for a pole with the code for a zero, as described in Appendix J. The analog-to-digital and digital-to-analog code can be merged with the arithmetic code which implements the filter by appending analog instructions to arithmetic/logic instructions: see the AS2920 manual.

During code generation for one stage of a filter, you may wish to sacrifice numerical accuracy in order to get a shorter program. Towards this end, code generation is performed subject to constraints which you specify. One such constraint consists of the "error" relative to piecewise linear bounds on the gain, in decibels, as a function of (log of) frequency. The Compiler then strives to minimize the mean-square deviation from these bounds.

Before saving the resulting code into a file, you may wish to interactively adjust the number of instructions you have allowed the Compiler to generate, or to adjust the error you are willing to tolerate, in order to achieve the shortest program which meets your error requirements. You can then save this new code in a file.

Filing and Retrieving Code or Parameters

This process of filter specification, display, and adjustment is extremely interactive. The file commands have been structured to facilitate the restart of an interrupted design or test session. They also make it easy to accumulate, into one or many files,

the partial results of specifying parameters or creating code. Parameter files saved from an interrupted design session can then be INCLUDEd at a later date, to resume that design session with all relevant variables restored to their condition at the time the session was interrupted.

You also have the ability to display any ISIS-II diskette file on the console, or to add arbitrary test or other data to the end of existing diskette files. This permits you to use disk files as scratch pads, and also to perform simple, low-level editing of files. The latter facility can be useful for building final 2920 assembly language files for submission to the AS2920 Assembler. The procedure for such submission is described briefly in Appendix G (see the Assembler manual for full details.)

Compound Commands

This Compiler contains macro and compound command facilities. They enable you to extend the language itself by defining your own commands using sequences of the simple commands described in Chapters 2 through 8.

Such macros make it possible, for example, to define an iterative process of moving poles and zeros and graphing the resultant response, without having to type all the commands during each iteration of that process, or to perform other design or test experiments, including code generation and display. Such a macro is interruptible at any time by hitting the console ESCape key. Macros can also be used as code generators for functions other than filters. Examples of macros which may be useful (and also used as models) appear in Chapters 10 and 11.

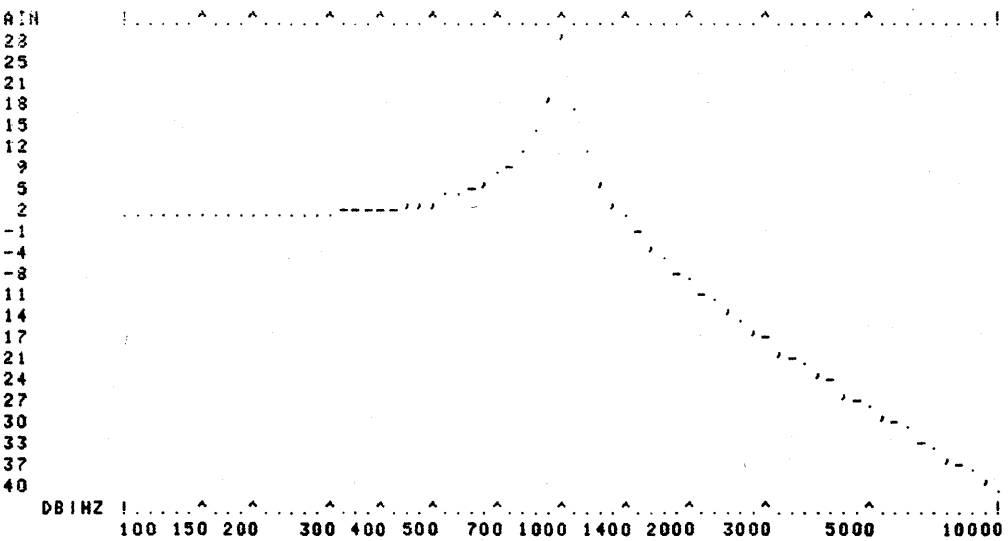
Simple Sample Session

This printout is a complete copy of an interactive session, produced by a LIST command (see Chapter 8).

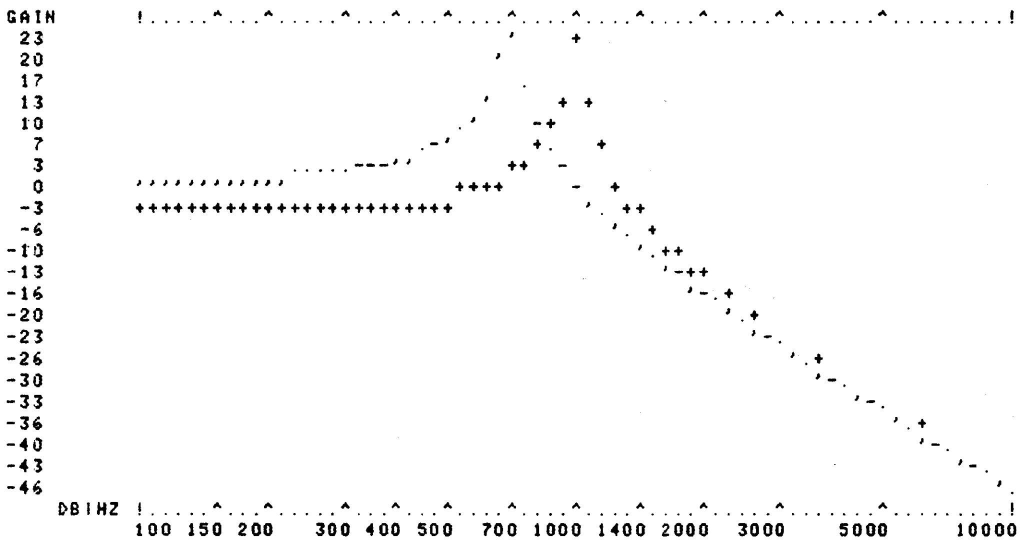
The macro used in this session is not intended for study at this point. It appears here only to illustrate the facility for retrieving macros and the form in which they are defined. Shown below is a macro to produce a Chebyshev filter. Macros and compound commands are explained in Chapter 9, with numerous examples in Chapters 10 and 11.

```
*
*; 13 DEC 79
*
*; ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS COMPILER, V1.0
*
*HELP    ; WHEN IN DOUBT, ASK FOR HELP
***  Help is available for the following items.  Type HELP followed  ***
***  by the item name.  Do not type the angle brackets.  (For more   ***
***  information on the help command, type HELP HELP.)               ***
Filters and Filter Responses:
   DEFINE,<FILTER$RESPONSE>,GREF,HOLD,MOVE,<POLE$OR$ZERO$LOCATION>,
   <POLES$OR$ZEROES>,REMOVE
Graphics:
   FSCALE,GRAPH,YSCALE
Code Generation:
   BOUNDS,CODE,<MA$CONSTRAINT>,<PZ$CONSTRAINT>
File/Display/Compound Commands:
   <COMPOUND$COMMANDS>,EVALUATE,EXIT,<FILE$COMMANDS>,HELP,MACRO,
   <PATH$NAME>,WRITE
Miscellaneous:
   <BOOLEAN$EXPR>,<CONSTANT>,<EXPR>,<FUNCTION>,<IDENTIFIER>,
   <INTEGER$EXPR>,<NUMERIC$CONSTANT>,<PRIMARY>,<PZ$REF>,<SPAC$REF>,
   <STRING>,<SYMBOL>,<SYMBOLIC$REF>
```

```
<FILTER$RESPONSE> - One of the following filter responses:
   GAIN       Gain in decibels as a function of frequency in Hertz.
   AGAIN      Absolute gain as a function of frequency in Hertz.
   PHASE      Phase delay in radians as a function of frequency in Hertz.
   GROUP      Group delay in seconds as a function of frequency in Hertz.
   GERROR     Signed deviation of GAIN from the LBOUND and UBOUND
              in decibels as a function of frequency in Hertz.
              See <BOUNDS>.
   IMPULSE    Impulse response as a function of time in seconds.
   STEP       Step response as a function of time in seconds.
Each <FILTER$RESPONSE> can be used as a command to tabulate the values.
Each can be graphed.  Each except IMPULSE and STEP can be used as a
function in an expression, e.g. AGAIN(60).
See <FUNCTION>, GRAPH, <PRIMARY>.

*
*DEFINE POLE 1 = -20,1000     ; CREATE A POLE IN CONTINUOUS S-PLANE
*
*PZ     ; LIST ALL POLES AND ZEROES
POLE 1 = -20.000000,1000.00000,CONTINUOUS
*
*GREF     ; GREF DETERMINES THE GAIN NORMALIZATION
GREF = 0.00000000 AT 0.00000000
*
*; GREF DEFAULTS TO 0 DECIBELS AT 0 HERTZ.
*
*YSCALE     ; YSCALE DETERMINES THE VERTICAL GRAPHICS SCALE
YSCALE = AUTO ; 0.00000000,0.00000000
*
*; AUTO MEANS THAT THE VERTICAL RANGE IS AUTOMATICALLY DETERMINED
*
*; BY THE MINIMUM AND MAXIMUM VALUES OF THE CURVE BEING PLOTTED.
*
*; CURRENTLY THE MINIMUM AND MAXIMUM ARE BOTH 0.
*
*FSCALE = 100, 10000     ; ESTABLISHES FREQUENCY RANGE OF INTEREST
*
*; WELL, LET'S SEE WHAT THE GAIN LOOKS LIKE FOR OUR ONE POLE FILTER.
*
*GRAPH GAIN
```



```
*
*; THE UNITS USED IN GRAPHING GAIN ARE SHOWN IN THE LOWER LEFT.
*
*; GAIN IN DECIBELS IS GRAPHED VERSUS FREQUENCY IN HERTZ.
*
*; NOTE THAT THE GAIN PEAKS AT ABOUT 1000 HZ.
*
*GAIN     ; CAN ALSO TABULATE GAIN VALUES
GAIN(100.000000) = 0.087190039
GAIN(107.006896) = 0.099910137
GAIN(114.504760) = 0.114497473
GAIN(122.527992) = 0.131230859
GAIN(131.113403) = 0.150431115
GAIN(140.300384) = 0.17246827
GAIN(150.131088) = 0.19777049
GAIN(160.650619) = 0.22683386
GAIN(171.90724) = 0.26023186
PROCESSING ABORTED
*
*; TABULATION WAS INTERRUPTED WITH THE ESCAPE KEY.
*
*; ESCAPE KEY CAN ALWAYS BE USED TO ABORT THE COMPILER'S PROCESSING.
*
*MOVE POLE 1 BY 0,-300     ; SHIFT GAIN PEAK DOWN TO 700 HZ
1 POLES/ZEROES MOVED
*
*OGRAPH GAIN     ; OVERGRAPH NEW GAIN CURVE
```




```
*
*
*
*; PLUS SIGNS INDICATE OLD CURVE.
*
*; WE CAN ALSO EXAMINE THE PHASE RESPONSE.
*
```

```
*GRAPH PHASE

PHASE     !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
-0.01     ,,,,,,,,,,,,,,,,,,,,,,,,,,___..
-0.16                               -
-0.31
-0.46                               -
-0.60
-0.75
-0.90
-1.05
-1.20
-1.35
-1.50
-1.65
-1.80
-1.95
-2.09                               -
-2.24
-2.39
-2.54
-2.69
-2.84                                -
-2.99                                 -.
-3.14                                   ,,,,_______________________________.
   RADIHZ !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
          100 150 200   300 400 500  700 1000 1400 2000  3000    5000     10000
*
*
*; PHASE IS EXPRESSED IN RADIANS.
*
*; NOTE THAT THE VERTICAL SCALE RANGES FROM ABOUT -PI TO 0.
*
*YSCALE    ; DISPLAYS ACTUAL MIN AND MAX VERTICAL VALUES
YSCALE = AUTO ; -3.1375729,-0.0083262023
*
*EVALUATE -PI    ; DISPLAYS DECIMAL VALUE OF -PI FOR COMPARISON
-3.1415927*10**0
*
*YSCALE = -PI,PI    ; A MORE APPROPRIATE VERTICAL SCALE FOR PHASE
*
*GRA PHA    ; GRAPH PHASE (ONLY FIRST THREE LETTERS ARE EVER NEEDED)

PHASE     !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
 3 14
 2 84
 2.54
 2 24
 1 94
 1.65
 1.35
 1 05
 0.75
 0 45
 0 15     ......................
-0 15                      ,,,,,__
-0.45                           -
-0.75
-1 05
-1 35
-1.65
-1 94
-2 24                           ,
-2.54
-2 84                            -
-3.14                             ,,,_____________________________________
   RADIHZ !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
          100 150 200   300 400 500  700 1000 1400 2000  3000    5000     10000
*
*
*
*; RATHER THAN CREATING A FILTER BY MANUALLY POSITIONING POLES AND ZEROES,
*
*; LET'S USE A PREDEFINED MACRO.
*
```

```
*INCLUDE :F1:CHEB.MAC     ; DESIRED MACRO IS DEFINED IN THIS FILE
*DEF MAC CHEB  ;*****************************************************
.*;      A CHEBYSHEV FILTER GENERATOR FOR SPAC20                     **
.*:
.*: CALLING SEQUENCE  :CHEB ORDER, Fco, LABEL, R.F.                  **
.*: WHERE ORDER is the order of the filter                           **
.*:       Fco is the cutoff frequency in Hz                          **
.*:       LABEL is the starting point for pole numbering             **
.*;       R.F. is the desired (or allowable) ripple factor in dB.    **
.*
.*:        EXAMPLE    :CHEB 6,500,23,0.12
.*:                     this will generate a CHEBYSHEV filter of order  **
.*:                     6, cutoff=500,and a peak-to-peak ripple of 0.12 **
.*;                     dB, producing 3 complex poles labeled 23,24,25  **
.*:
.*DEF .?CHEBYRIP=10**(ABS(%3)/10)-1                  ;* BEGIN THE       **
.*DEF .?SINHP=1                                      ;* CHEBYSHEV       **
.*DEF .?COSHP=1                                      ;* BY SETTING      **
.* IF .?CHEBYRIP <> 0 THEN                           ;* DEFAULT VALUES, **
.*    :TEMCHB .?SINHP,.?COSHP,.?CHEBYRIP,%0          ;** OR USE THE     **
.* END                                               ;** SUB-MACRO      **
.*REM .?CHEBYRIP                                     ;** TO GENERATE    **
.* DEFINE .?BUTSTART = ( HPI ) + ( HPI/%0 )          ;** THE VARIABLES. **
.* DEFINE .?BUTDELTA = ( PI/%0 )                     ;**                **
.* DEFINE .?BUTINDEX = 0                             ;* A MODIFIED      **
.* DEFINE .?BUTANGLE=0                               ;* BUTTERWORTH     **
.* REPEAT                                           ;* MODULE IS       **
.*    .?BUTINDEX = .?BUTINDEX + 1                    ;* INCORPORATED     **
.*    .?BUTANGLE=.?BUTSTART - .?BUTINDEX*.?BUTDELTA  ;* TO GENERATE     **
.*      IF .?BUTANGLE < .?BUTDELTA/4 THEN            ;* THE APPROPRIATE **
.*         .?BUTANGLE=0                              ;* PATTERN OF      **
.*      END                                          ;* POLES FOR THE   **
.*   DEF POL(.?BUTINDEX+%2-1)=                &      ;* FILTER. (the &s **
.**          -%1*.?SINHP*COS(.?BUTANGLE),     &       ;* allow greater   **
.**                %1*.?COSHP*SIN(.?BUTANGLE)         ;* readability of  **
.*   WHILE .?BUTINDEX + 1 <= ( %0 + 1 ) / 2          ;* the formula.)   **
.* END
.* REM .?BUTINDEX                                    ;** REMOVE THE     **
.* REM .?BUTDELTA                                    ;** VARIABLES      **
.* REM .?BUTSTART                                    ;** INTRODUCED     **
.* REM .?BUTANGLE                                    ;** IN THIS MACRO, **
.* REM .?SINHP                                       ;** TO SAVE SPACE. **
.* REM .?COSHP                                       ;**               **
.*PZ
.*EM                              ;******* END OF CHEBYSHEV MACRO ********
*
*;  TEMCHB     GET VARIABLES FOR CHEBYSHEV FILTER ######################
*DEF MAC TEMCHB
.*; CALLING SEQUENCE                                ## THIS IS THE    ###
.*;      :TEMCHB   SINHP,COSHP,E**2,N               ## SUB-MACRO.     ###
.*DEF .?INVSXTEMP=(1/SQR(%2))+(SQR((1/%2)+1))
.*DEF .?INVSXTEMPP=.?INVSXTEMP **(1/%3)                              ;###
.*DEF .?INVSXTEMPN=.?INVSXTEMP **(-1/%3)
.*%0=(.?INVSXTEMPP - .?INVSXTEMPN)/2                                 ;###
.*%1=(.?INVSXTEMPP + .?INVSXTEMPN)/2
.*REM .?INVSXTEMP                                                    ;###
.*REM .?INVSXTEMPP
```

```
*; ALL THE ABOVE MACRO DEFINITION CAME FROM THE INCLUDED FILE :F1:CHEB.MAC.
*
*REMOVE PZ     ; DELETES ALL OLD POLES AND ZEROES
1 POLES/ZEROES REMOVED
*
*; NOW INVOKE MACRO CHEB TO BUILD A CHEBYSHEV LOW PASS FILTER.
*
*:CHEB 7,1000,0,2     ; 7TH ORDER, 1000 HZ CUTOFF, 1ST LABEL 0, 2 DB RIPPLE
.*;      A CHEBYSHEV FILTER GENERATOR FOR SPAC20                          **
.*;
.*; CALLING SEQUENCE  :CHEB ORDER, Fco, LABEL, R.F.                       **
.*; WHERE ORDER is the order of the filter                                **
.*;       Fco is the cutoff frequency in Hz                              **
.*;       LABEL is the starting point for pole numbering                 **
.*;       R.F. is the desired (or allowable) ripple factor in dB.        **
.*
.*;        EXAMPLE    :CHEB 6,500,23,0.12
.*;                     this will generate a CHEBYSHEV filter of order   **
.*;                     6, cutoff=500,and a peak-to-peak ripple of 0.12  **
.*;                     dB, producing 3 complex poles labeled 23,24,25   **
.*;
.*DEF .?CHEBYRIP=10**(ABS(2)/10)-1                    ;* BEGIN THE        **
.*DEF .?SINHP=1                                       ;* CHEBYSHEV        **
.*DEF .?COSHP=1                                       ;* BY SETTING       **
.* IF .?CHEBYRIP <> 0 THEN                            ;* DEFAULT VALUES,  **
. *     :TEMCHB .?SINHP,.?COSHP,.?CHEBYRIP,7          ;** OR USE THE      **
. *; CALLING SEQUENCE                                  ## THIS IS THE     ###
. *;       :TEMCHB   SINHP,COSHP,E**2,N                ## SUB-MACRO.      ###
. *DEF .?INVSXTEMP=(1/SQR(.?CHEBYRIP))+(SQR((1/.?CHEBYRIP)+1))
. *DEF .?INVSXTEMPP=.?INVSXTEMP **(1/7)                                 ;###
. *DEF .?INVSXTEMPN=.?INVSXTEMP **(-1/7)
. *.?SINHP=(.?INVSXTEMPP - .?INVSXTEMPN)/2                                   ;###
. *.?COSHP=(.?INVSXTEMPP + .?INVSXTEMPN)/2
. *REM .?INVSXTEMP                                                      ;###
. *REM .?INVSXTEMPP
. *REM .?INVSXTEMPN                                   ;## END OF TEMCHB ###
. *EM
.* END                                                 ;** SUB-MACRO       **
.*REM .?CHEBYRIP                                      ;** TO GENERATE     **
.* DEFINE .?BUTSTART = ( HPI ) + ( HPI/7 )            ;** THE VARIABLES. **
.* DEFINE .?BUTDELTA = ( PI/7 )                       ;**                 **
.* DEFINE  ?BUTINDEX = 0                              ;* A MODIFIED       **
.* DEFINE .?BUTANGLE=0                                ;* BUTTERWORTH      **
.* REPEAT                                            ;* MODULE IS        **
. *    .?BUTINDEX = .?BUTINDEX + 1                     ;* INCORPORATED     **
. *    .?BUTANGLE=.?BUTSTART - .?BUTINDEX*.?BUTDELTA   ;* TO GENERATE      **
. *     IF .?BUTANGLE < .?BUTDELTA/4 THEN              ;* THE APPROPRIATE **
. *          .?BUTANGLE=0                               ;* PATTERN OF       **
. *     END                                             ;* POLES FOR THE    **
. *   DEF POL(.?BUTINDEX+0-1)=              &          ;* FILTER. (the &s **
. **          -1000*.?SINHP*COS(.?BUTANGLE),     &       ;* allow greater   **
. **                1000*.?COSHP*SIN(.?BUTANGLE)         ;* readability of  **
. *   WHILE .?BUTINDEX + 1 <= ( 7 + 1 ) / 2             ;* the formula.)    **
. * END
.* REM .?BUTINDEX                                      ;** REMOVE THE      **
.* REM .?BUTDELTA                                      ;** VARIABLES       **
.* REM .?BUTSTART                                      ;** INTRODUCED      **
.* REM .?BUTANGLE                                      ;** IN THIS MACRO,  **
.* REM .?SINHP                                         ;** TO SAVE SPACE.  **
.* REM .?COSHP                                         ;**                 **
.*PZ
.*EM                              ;******* END OF CHEBYSHEV MACRO ********
POLE 0 = -34.566349,986.62048,CONTINUOUS
POLE 1 = -96.852775,791.20825,CONTINUOUS
POLE 2 = -139.956344,439.08737,CONTINUOUS
POLE 3 = -155.339813,0.00000000,CONTINUOUS; REAL
*
*; ONE REAL POLE AND THREE COMPLEX POLE PAIRS WERE CREATED.
*
*YSC = -5,1     ; SELECT A NEW VERTICAL SCALE FOR GAIN
*
```

```
*GR G     ; GRAPH GAIN (MANY KEYWORDS HAVE ONE OR TWO LETTER ABBREVIATIONS)

GAIN      !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
 1.00
 0.71
 0.43
 0.14                          .-
-0.14                         .  -      '.
-0.43                        .
-0.71                               '
-1.00     '-.               '          '
-1.29        '-.           .       '       .
-1.57           -.       .'            .   .
-1.86             '--...-           '.      .
-2.14
-2.43
-2.71
-3.00
-3.29
-3.57
-3.86
-4.14
-4.43
-4.71
-5.00                                        **********************************
     DBIHZ !.....^...^.....^...^...^....^....^....^....^.....^.......^.........!
          100 150 200   300 400 500  700 1000 1400 2000  3000     5000     10000
*
*; ASTERISKS (ABOVE 1000 HZ HERE) INDICATE VERTICAL SATURATION.
*
*; LET'S ZOOM IN ON THE REGION NEAR 1000 HZ.
*
*FSCALE = 400,500,600,700,800,900,1000,1100     ; NEW FREQUENCY RANGE
*
*GR G     ; GRAPH GAIN

GAIN      !........!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
 1.00
 0.71
 0.43
 0.14      ..--.                             .--.                ..
-0.14     '     '-                          -    -.
-0.43             '                       .'      '.            '  .
-0.71               '.                   -          .          .
-1.00                 -.                -            .
-1.29                   -             .'              .       -
-1.57                    '-.        .-                 -     -
-1.86                       '-.....-'                   '-.'        .
-2.14
-2.43
-2.71                                                               .
-3.00
-3.29
-3.57                                                                .
-3.86
-4.14
-4.43
-4.71
-5.00                                                                 *********
     DBIHZ !........!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
          400     500 540   600 640   700 740  800 840   900 940  1000 1050 1100
*
*
*
*; WELL, LET'S NOW MOVE INTO THE WORLD OF SAMPLED FILTERS.
*
*MOVE POLES TO TS     ; CONVERT FILTER TO SAMPLED VIA MATCHED-Z TRANSFORM
ERR 73:SAMPLE RATE UNDEFINED
*
*; SAMPLE RATE MUST BE SPECIFIED FIRST.
*
*TS = 1/13020     ; REASONABLE RATE FOR FULL 192 INSTRUCTION 2920 PROGRAM
TS = 7.6805004/10**5
*
```

```
*MOVE POLES TO TS    ; TRY AGAIN
4 POLES/ZEROES MOVED
*
*PZ    ; LET'S SEE WHAT WE HAVE
POLE 0 = -34.566349,986.62048,TS
POLE 1 = -96.852775,791.20825,TS
POLE 2 = -139.956344,439.08737,TS
POLE 3 = -155.339813,0.00000000,TS; REAL
*
*HOLD ON    ; THIS INTRODUCES COMPENSATION FOR SAMPLE AND HOLD DISTORTION
*
*OGR G    ; GRAPH GAIN OF SAMPLED VERSION OF FILTER OVER ORIGINAL VERSION

GAIN      !.....!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
 1 00
 0 71
 0 43                                                .--.                ''
 0 14      .--''-.                                  -++++'.              -++
-0 14      +     +'-                              .'+    +.                 '
-0 43             +'.                           -++      ++-           '+  +
-0 71               +-.                       -+          +-         .+
-1 00                 +-                    .'+            +-       .
-1 29                   +-.               -++               +'.   . +      .
-1 57                   ++-..  ..-'+                        + '-' +
-1 86                      ++'''+++                          +++++       +
-2 14
-2 43
-2 71
-3 00                                                                    .
-3 29
-3 57                                                                    +
-3 86
-4 14
-4 43
-4 71
-5 00                                                                -********
    DB|HZ !.....!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
          400    500 540   600 640   700 740  800 840   900 940  1000 1050 1100
*
*; OBSERVE THAT THE SAMPLED FILTER CLOSELY RECREATES THE CONTINUOUS FILTER.
*
*; IF WE HAD CONVERTED TO SAMPLED USING BILINEAR TRANSFORM INSTEAD OF
*
*; MATCHED-Z, THE GAIN CURVES WOULD MATCH EVEN MORE CLOSELY.
*
*; NOW THAT WE'RE SATISFIED WITH THIS FILTER, LET'S GENERATE 2920 CODE FOR IT.
*
*PUT :F1:BEFORE.TMP PZ    ; SAVE THE CURRENT POLES IN A DISK FILE BACKUP
*
*CODE POLE 0 INST<11    ; WILLING TO DEDICATE 10 INSTRUCTIONS TO POLE 0
B1=1.7481516  B2=-0.96718828

INST=4
 POLE 0 = 0.00000000,3255.0000,TS
  BEST YET
 POLE 0 = -718.16894,1627.50000,TS
  BEST YET

INST=5
 POLE 0 = 0.00000000,1497.64807,TS
  BEST YET
 POLE 0 = -32.894828,3255.0000,TS
 POLE 0 = -32.894828,3255.0000,TS
 POLE 0 = -785.03710,1558.37561,TS

INST=6
 POLE 0 = 0.00000000,1047.20678,TS
  BEST YET
 POLE 0 = 0.00000000,1497.64807,TS
 POLE 0 = 0.00000000,1047.20678,TS
 POLE 0 = 231.19877,960.76947,TS
 POLE 0 = -298.06701,1084.99975,TS
 POLE 0 = -34.920330,3255.0000,TS
 POLE 0 = -34.985778,3255.0000,TS
 POLE 0 = -820.16253,1520.12536,TS
```

```
INST=7
 POLE 0 = 0.00000000,978.23999,TS
  BEST YET
 POLE 0 = 0.00000000,1047.20678,TS
 POLE 0 = 0.00000000,1051.37915,TS
 POLE 0 = -32.894828,985.61517,TS
  BEST YET
 POLE 0 = -32.894828,985.61517,TS
 POLE 0 = -298.06701,1084.99975,TS
 POLE 0 = -32.894828,985.61517,TS
 POLE 0 = 204.96688,906.54724,TS
 POLE 0 = -276.70343,1121.26489,TS
 POLE 0 = -34.479980,3255.0000,TS
 POLE 0 = -34.462657,3255.0000,TS
 POLE 0 = -820.16253,1520.12536,TS

INST=8
 POLE 0 = 0.00000000,987.10296,TS
 POLE 0 = 0.00000000,1051.37915,TS
 POLE 0 = 0.00000000,1051.18432,TS
 POLE 0 = -32.894828,986.73706,TS
  BEST YET
 POLE 0 = -32.894828,986.73706,TS
 POLE 0 = -32.894828,985.61517,TS
 POLE 0 = -32.894828,990.09570,TS
 POLE 0 = -34.920330,981.67358,TS
 POLE 0 = -34.985778,981.54577,TS
 POLE 0 = -276.70343,1121.26489,TS
 POLE 0 = 191.59779,877.43200,TS
 POLE 0 = -34.549556,3255.0000,TS
 POLE 0 = -34.593387,3255.0000,TS

INST=9
 POLE 0 = 0.00000000,986.55126,TS
 POLE 0 = 0.00000000,1051.18432,TS
 POLE 0 = 0.00000000,1051.18432,TS
 POLE 0 = -32.894828,990.09570,TS
 POLE 0 = -32.894828,989.88684,TS
 POLE 0 = -34.920330,986.17498,TS
  BEST YET
 POLE 0 = -34.985778,986.04791,TS
 POLE 0 = -34.985778,986.04791,TS
 POLE 0 = -34.479980,982.53204,TS
 POLE 0 = -34.462657,982.56579,TS
 POLE 0 = -34.549934,3255.0000,TS

INST=10
 POLE 0 = 0.00000000,1051.18432,TS
 POLE 0 = 0.00000000,1051.20874,TS
 POLE 0 = -32.894828,989.88684,TS
 POLE 0 = -32.894828,989.88684,TS
 POLE 0 = -34.920330,986.73651,TS
  BEST YET
 POLE 0 = -34.985778,986.60949,TS
 POLE 0 = -34.985778,986.04791,TS
 POLE 0 = -34.985778,985.83789,TS
 POLE 0 = -34.479980,987.02893,TS
 POLE 0 = -34.462657,987.06262,TS
 POLE 0 = -34.724136,986.55566,TS
  BEST YET
 POLE 0 = -34.549556,982.39642,TS
 POLE 0 = -34.593387,982.31103,TS
 POLE 0 = -34.550563,3255.0000,TS

INST=10
 POLE 0 = -34.724136,986.55566,TS
  BEST
PERROR = 0.157787329, 0.064819335
```

```
; NOTE: MAKE SURE SIGNAL IS <0.57206704
LDA OUT2_P0,OUT1_P0,R00
     ; OUT2_P0=1.00000000*OUT1_P0
LDA OUT1_P0,OUT0_P0,R00
     ; OUT1_P0=1.00000000*OUT0_P0
SUB OUT0_P0,OUT1_P0,R02
     ; OUT0_P0=1.00000000*OUT0_P0-0.25000000*OUT1_P0
SUB OUT0_P0,OUT1_P0,R09
     ; OUT0_P0=1.00000000*OUT0_P0-0.25195312*OUT1_P0
ADD OUT0_P0,OUT1_P0,R00
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0
ADD OUT0_P0,OUT2_P0,R05
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0+0.031250000*OUT2_P0
ADD OUT0_P0,OUT2_P0,R09
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0+0.033203125*OUT2_P0
SUB OUT0_P0,OUT2_P0,R12
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0+0.032958984*OUT2_P0
SUB OUT0_P0,OUT2_P0,R00
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0-0.96704101*OUT2_P0
ADD OUT0_P0,IN0_P0,R00
     ; OUT0_P0=1.00000000*OUT0_P0+0.74804687*OUT1_P0-0.96704101*OUT2_P0+1.00000000*IN0_P0
*
*; THE DISPLAY SHOWS EVERY ATTEMPT TO GENERATE CODE FOR POLE 0, GIVING IN
*
*; EACH CASE THE POLE POSITION CORRESPONDING TO THE CODE.  THE MESSAGES
*
*; "BEST YET" AND "BEST" INDICATE THE PROGRESS OF THE CODING ATTEMPTS.
*
*; SINCE WE DID NOT SPECIFY AN ERROR CONSTRAINT IN THE CODE COMMAND,
*
*; DISTANCE IN THE S-PLANE FROM THE ORIGINAL POLE 0 WAS MINIMIZED.
*
*; POLE 0 IS NOW MOVED TO THE POSITION CORRESPONDING TO THE BEST CODE.
*
*; THE CODING ALGORITHM SELECTED REQUIRES THAT THE OUTPUT SIGNAL NOT
*
*; EXCEED 0.572 IN ORDER TO PREVENT INTERMEDIATE CALCULATIONS FROM
*
*; OVERFLOWING.
*
*PUT :F1:CODE.SRC '; 7TH ORDER CHEBYSHEV', CODE     ; SAVE CODE IN NEW FILE
*
*; INSERT IN GROWING CODE FILE CODE TO PASS SIGNAL FROM FIRST STAGE TO SECOND.
*
*APPEND :F1:CODE.SRC 'IN0_P1 EQU OUT0_P0'     ; INPUT TO SECOND IS OUTPUT FROM FIRST
*
*CODE POLE 1 INST<11     ; CODE POLE 1 NEXT IN 10 INSTRUCTIONS OR FEWER
B1=1.7712245  B2=-0.91075761

INST=4
 POLE 1 = 0.00000000,3255.0000,TS
  BEST YET
 POLE 1 = -718.16894,1627.50000,TS
  BEST YET

INST=5
 POLE 1 = -66.868217,3255.0000,TS
 POLE 1 = -66.868217,3255.0000,TS
 POLE 1 = -1016.23577,1275.39575,TS
  BEST YET

INST=6
 POLE 1 = 0.00000000,736.50292,TS
  BEST YET
 POLE 1 = 0.00000000,1047.20678,TS
 POLE 1 = 0.00000000,1047.20678,TS
 POLE 1 = 122.034751,704.20874,TS
 POLE 1 = 122.034751,704.20874,TS
 POLE 1 = -99.763069,3255.0000,TS
 POLE 1 = -101.993606,3255.0000,TS
 POLE 1 = -1013.54125,1279.19873,TS
```

•
•
•

```
INST=10
 POLE 1 = 0.00000000,1000.82678,TS
 POLE 1 = 0.00000000,1000.81738,TS
 POLE 1 = -66.868217,791.33288,TS
 POLE 1 = -66.868217,791.33288,TS
 POLE 1 = -66.868217,862.28106,TS
 POLE 1 = -99.763069,788.89288,TS
  BEST YET
 POLE 1 = -101.993606,791.89245,TS
 POLE 1 = -101.993606,774.43041,TS
 POLE 1 = -101.993606,778.24249,TS
 POLE 1 = -97.537277,805.69842,TS
 POLE 1 = -97.537277,805.69842,TS
 POLE 1 = -102.551963,793.35278,TS
 POLE 1 = -97.537277,794.43249,TS
  BEST YET
 POLE 1 = -96.838409,850.94934,TS
 POLE 1 = -96.981559,850.62072,TS
 POLE 1 = -118.310440,799.93048,TS
 POLE 1 = -118.310440,799.93048,TS
 POLE 1 = -96.837249,3255.0000,TS
INST=10
 POLE 1 = -97.537277,794.43249,TS
  BEST
PERROR = 0.68450166, -3.2242431

; NOTE: MAKE SURE SIGNAL IS <0.56512143
LDA OUT2_P1,OUT1_P1,R00
     ; OUT2_P1=1.00000000*OUT1_P1
LDA OUT1_P1,OUT0_P1,R00
     ; OUT1_P1=1.00000000*OUT0_P1
ADD OUT0_P1,OUT1_P1,R02
     ; OUT0_P1=1.00000000*OUT0_P1+0.25000000*OUT1_P1
ADD OUT0_P1,OUT0_P1,R06
     ; OUT0_P1=1.01562500*OUT0_P1+0.25390625*OUT1_P1
ADD OUT0_P1,OUT1_P1,R01
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1
ADD OUT0_P1,OUT2_P1,R04
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1+0.062500000*OUT2_P1
ADD OUT0_P1,OUT2_P1,R05
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1+0.093750000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R08
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1+0.089843750*OUT2_P1
SUB OUT0_P1,OUT2_P1,R00
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1-0.91015625*OUT2_P1
ADD OUT0_P1,IN0_P1,R00
     ; OUT0_P1=1.01562500*OUT0_P1+0.75390625*OUT1_P1-0.91015625*OUT2_P1+1.00000000*IN0_P1
*
*APPEND :F1:CODE.SRC CODE     ; SAVE CODE AT END OF GROWING FILE
*
*APPEND :F1:CODE.SRC 'IN0_P2 EQU OUT0_P1'     ; INPUT TO THIRD IS OUTPUT FROM SECOND
*
*CODE P 2 INST<11     ; CODE POLE 2 NEXT
B1=1.8275704  B2=-0.87364599

INST=4
 POLE 2 = 0.00000000,3255.0000,TS
  BEST YET

INST=5
 POLE 2 = -138.351623,3255.0000,TS
  BEST YET
 POLE 2 = -138.351623,3255.0000,TS
 POLE 2 = -1314.30297,704.20874,TS
  BEST YET

INST=6
 POLE 2 = 0.00000000,366.79486,TS
  BEST YET
 POLE 2 = 0.00000000,736.50292,TS
 POLE 2 = 0.00000000,736.50292,TS
 POLE 2 = 31.882545,362.57061,TS
 POLE 2 = 31.882545,362.57061,TS
 POLE 2 = -140.377334,3255.0000,TS
 POLE 2 = -139.508758,3255.0000,TS
 POLE 2 = -1285.91467,780.04730,TS
```

•
•
•

```
INST=10
 POLE 2 = 0.00000000,866.71618,TS
 POLE 2 = 0.00000000,866.76428,TS
 POLE 2 = -138.351623,438.47937,TS
 POLE 2 = -138.351623,438.47937,TS
 POLE 2 = -138.351623,446.13854,TS
 POLE 2 = -140.377334,434.17916,TS
 POLE 2 = -139.508758,438.24417,TS
 POLE 2 = -139.508758,438.24417,TS
 POLE 2 = -139.508758,441.13485,TS
 POLE 2 = -140.015365,512.25738,TS
 POLE 2 = -140.087631,511.97113,TS
 POLE 2 = -156.432159,442.24893,TS
 POLE 2 = -139.363952,438.91845,TS
  BEST YET
 POLE 2 = -89.232948,427.92999,TS
 POLE 2 = -89.232948,427.92999,TS
 POLE 2 = 39.337009,402.49118,TS
 POLE 2 = 39.337009,402.49118,TS

INST=10
 POLE 2 = -139.363952,438.91845,TS
  BEST
PERROR = -0.59239199, 0.16891479

; NOTE: MAKE SURE SIGNAL IS <0.54700859
LDA OUT2_P2,OUT1_P2,R00
     ; OUT2_P2=1.00000000*OUT1_P2
LDA OUT1_P2,OUT0_P2,R00
     ; OUT1_P2=1.00000000*OUT0_P2
ADD OUT0_P2,OUT1_P2,R01
     ; OUT0_P2=1.00000000*OUT0_P2+0.50000000*OUT1_P2
ADD OUT0_P2,OUT1_P2,R03
     ; OUT0_P2=1.00000000*OUT0_P2+0.62500000*OUT1_P2
ADD OUT0_P2,OUT0_P2,R03
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2
ADD OUT0_P2,OUT2_P2,R03
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2+0.125000000*OUT2_P2
ADD OUT0_P2,OUT2_P2,R10
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2+0.125976562*OUT2_P2
SUB OUT0_P2,OUT2_P2,R13
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2+0.125854492*OUT2_P2
SUB OUT0_P2,OUT2_P2,R00
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2-0.87414550*OUT2_P2
ADD OUT0_P2,IN0_P2,R00
     ; OUT0_P2=1.12500000*OUT0_P2+0.70312500*OUT1_P2-0.87414550*OUT2_P2+1.00000000*IN0_P2
*
*APPEND :F1:CODE.SRC CODE     ; SAVE CODE
*
*APPEND :F1:CODE.SRC 'IN0_P3 EQU OUT0_P2'     ; INPUT TO FOURTH IS OUTPUT FROM THIRD
*
*CODE P 3 INST<11     ; CODE POLE 3 LAST
B1=0.92777714  B2=0.00000000

INST=2
 POLE 3 = 0.00000000,0.00000000,TS; REAL
  BEST YET

INST=3
 POLE 3 = 0.00000000,0.00000000,TS; REAL
 POLE 3 = 0.00000000,0.00000000,TS; REAL
 POLE 3 = -133.736541,0.00000000,TS; REAL
  BEST YET

INST=4
 POLE 3 = -133.736541,0.00000000,TS; REAL
 POLE 3 = -133.736541,0.00000000,TS; REAL
 POLE 3 = -151.077224,0.00000000,TS; REAL
  BEST YET

INST=5
 POLE 3 = -151.077224,0.00000000,TS; REAL
 POLE 3 = -151.077224,0.00000000,TS; REAL
 POLE 3 = -155.401062,0.00000000,TS; REAL
  BEST YET

INST=6
 POLE 3 = -155.435150,0.00000000,TS; REAL
 POLE 3 = -155.401062,0.00000000,TS; REAL
 POLE 3 = -155.356887,0.00000000,TS; REAL
  BEST YET
```

```
INST=7
 POLE 3 = -155.435150,0.00000000,TS; REAL
 POLE 3 = -155.356887,0.00000000,TS; REAL
 POLE 3 = -155.344894,0.00000000,TS; REAL
  BEST YET

INST=8
 POLE 3 = -155.435150,0.00000000,TS; REAL
 POLE 3 = -155.344894,0.00000000,TS; REAL
 POLE 3 = -155.337036,0.00000000,TS; REAL
  BEST YET

INST=9
 POLE 3 = -155.435150,0.00000000,TS; REAL
 POLE 3 = -155.337036,0.00000000,TS; REAL
 POLE 3 = -155.340621,0.00000000,TS; REAL
  BEST YET

INST=10
 POLE 3 = -155.435150,0.00000000,TS; REAL
 POLE 3 = -155.340621,0.00000000,TS; REAL
 POLE 3 = -155.340621,0.00000000,TS; REAL

INST=9
 POLE 3 = -155.340621,0.00000000,TS; REAL
  BEST
PERROR = 8.0871562/10**4, 0.00000000

LDA OUT1_P3,OUT0_P3,R00
    ; OUT1_P3=1.00000000*OUT0_P3
SUB OUT0_P3,OUT1_P3,R07
    ; OUT0_P3=1.00000000*OUT0_P3-0.0078125000*OUT1_P3
SUB OUT0_P3,OUT0_P3,R04
    ; OUT0_P3=0.93750000*OUT0_P3-0.0073242187*OUT1_P3
SUB OUT0_P3,OUT1_P3,R07
    ; OUT0_P3=0.93750000*OUT0_P3-0.0151367187*OUT1_P3
ADD OUT0_P3,OUT0_P3,R09
    ; OUT0_P3=0.93933105*OUT0_P3-0.0151662826*OUT1_P3
ADD OUT0_P3,OUT1_P3,R11
    ; OUT0_P3=0.93933105*OUT0_P3-0.0146780014*OUT1_P3
ADD OUT0_P3,OUT0_P3,R08
    ; OUT0_P3=0.94300029*OUT0_P3-0.0147353378*OUT1_P3
SUB OUT0_P3,OUT1_P3,R11
    ; OUT0_P3=0.94300029*OUT0_P3-0.0152236190*OUT1_P3
ADD OUT0_P3,IN0_P3,R00
    ; OUT0_P3=0.94300029*OUT0_P3-0.0152236190*OUT1_P3+1.00000000*IN0_P3
*
*APPEND :F1:CODE.SRC CODE    ; SAVE CODE
*
*; NOW ALL OUR POLES ARE CODED.
*
*; LET'S SEE WHAT THE GAIN RESPONSE LOOKS LIKE FOR OUR FILTER AS CODED.
*
*; RECALL THAT THE LAST ITEM GRAPHED WAS THE GAIN FOR THE SAMPLED UNCODED FILTER.
*
*OGR G     ; GRAPH GAIN AS CODED OVER ORIGINAL SAMPLED GAIN

GAIN      !.....!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
 1.00
 0.71                                                                  .-
 0.43                                              +.-.                ++
 0.14      .--''-.                                .'   '.              '   .
-0.14            '.                          .+-       -             .    +
-0.43              -.                       +'          '           +
-0.71                -+                    .'             '.       '
-1.00                 '-+                +.                +-     -
-1.29                   '.+             .-                  +'- -'       -
-1.57                     '-.+   ++.-                        +++
-1.86                        '----'
-2.14
-2.43
-2.71
-3.00                                                                  -
-3.29
-3.57
-3.86
-4.14
-4.43
-4.71
-5.00                                                              '********
    DBINZ !.....!...^.....!...^.....!...^....!...^.....!...^.....!....^....!
          400    500 540   600 640   700 740  800 840   900 940  1000 1050 1100
```

```
*
*; CLOSE ENOUGH FOR OUR PURPOSES.
*
*; WE MUST NOW SELECT APPROPRIATE SCALING FACTORS FOR OUR GENERATED CODE.
*
*PUT :F1:AFTER.TMP PZ     ; SAVE IN A FILE THE CURRENT POLE POSITIONS
*
*REMOVE POLES 1 THRU 3     ; LEAVE ONLY FIRST POLE
3 POLES/ZEROES REMOVED
*
*HOLD OFF     ; SAMPLE AND HOLD COMPENSATION NOT NEEDED DURING SCALING
*
*YSCALE = AUTO     ; SO WE CAN FREELY GRAPH ABSOLUTE GAINS
*
*GR AGAIN     ; GRAPH THE ABSOLUTE (MULTILPICATIVE) GAIN FOR POLE 0

AGAIN      !.....!...^.....!...^.....!...^....!...^.....!...^.....!...^....!
65.9                                                                  ,,
63.0                                                                 -
60.1                                                                ,
57.2
54.4                                                                  .  ,
51.5
48.6                                                                     -
45.7                                                                ,
42.9                                                                   .
40.0                                                               -
37.1                                                                  .
34.2                                                               ,
31.4                                                              -      -
28.5                                                             -         .
25.6                                                            -           .
22.7                                                          .'              -
19.9                                                        -                 -
17.0                                                     .-'                   ,
14.1                                                  .--'
11.2                                            ...-''
 8.4                                    .....---'''
 5.5       ----------,,,,,,,,
    AMPIHZ !.....!...^.....!...^.....!...^....!...^.....!...^.....!...^....!
           400    500 540   600 640   700 740  800 840   900 940  1000 1050 1100
*
*
*
*; THE MAXIMUM AGAIN IS A GOOD GUIDLINE FOR THE NECESSARY SCALING.
*
*; WE MUST ALSO TAKE INTO ACCOUNT THE CODE BUFFER COMMENTS WHICH
*
*; INSTRUCTED US TO LIMIT SIGNALS TO PREVENT INTERMEDIATE
*
*; CALCULATIONS OVERFLOW.
*
*; TO BE VERY SAFE, WE SHOULD ACTUALLY MAKE SURE THE SIGNAL COMING
*
*; OUT OF EACH STAGE IS LESS THAN ONE FOURTH.
*
*EVALUATE MAGAIN     ; THE MAXIMUM AGAIN CAN ALSO BE ACCESSED THIS WAY
6.5861554*10**1
*
*EVALUATE LOG(MAGAIN)/LOG(2)     ; I.E. LOG BASE 2 OF MAXIMUM AGAIN
6.0413646*10**0
*
*; THIS SIGNAL MUST BE SCALED BY AT LEAST THIS MUCH BEFORE ENTERING FIRST POLE.
*
*; THIS SCALING MUST BE EDITED INTO THE CODE FILE LATER BY HAND,
*
*; BUT WE'LL APPEND NOTES TO THE CODE FILE NOW TO REMIND US.
*
*APPEND :F1:CODE.SRC '; SCALE INPUT TO POLE 0 WITH RIGHT 8 SHIFT'     ; MAKE NOTE
*
*REM POLES     ; REMOVE POLES
1 POLES/ZEROES REMOVED
*
*INCLUDE :F1:AFTER.TMP     ; GET OUR FILTER AS CODED BACK AGAIN
*DEFINE POLE 0 = -34.724136,986.55566,TS
*DEFINE POLE 1 = -97.537277,794.43249,TS
*DEFINE POLE 2 = -139.363952,438.91845,TS
*DEFINE POLE 3 = -155.340621,0.00000000,TS; REAL
*
```

```
*
*REM P 2 THRU 3     ; MUST LOOK AT FIRST TWO STAGES NEXT
2 POLES/ZEROES REMOVED
*
*EVALUATE LOG(MAGAIN)/LOG(2)     ; SCALING FACTOR NEEDED BEFORE SECOND STAGE
9.6666240*10**0
*
*; WE HAVE ALREADY SCALED BY 2**8 SO ANOTHER 2**4 WILL SUFFICE.
*APPEND :F1:CODE.SRC '; SCALE INPUT TO POLE 1 WITH RIGHT 4 SHIFT'     ; MAKE NOTE
*
*REM P     ; REMOVE POLES
2 POLES/ZEROES REMOVED
*
*INCLUDE :F1:AFTER.TMP     ; GET OUR FILTER AS CODED BACK AGAIN
*DEFINE POLE 0 = -34.724136,986.55566,TS
*DEFINE POLE 1 = -97.537277,794.43249,TS
*DEFINE POLE 2 = -139.363952,438.91845,TS
*DEFINE POLE 3 = -155.340621,0.00000000,TS; REAL
*
*
*REM P 3     ; MUST LOOK AT FIRST THREE STAGES NEXT
1 POLES/ZEROES REMOVED
*
*EVALUATE LOG(MAGAIN)/LOG(2)     ; SCALING FACTOR NEEDED BEFORE THIRD STAGE
1.22287941*10**1
*
*; WE HAVE ALREADY SCALED BY 2**12 SO ANOTHER 2**3 WILL SUFFICE.
*
*APPEND :F1:CODE.SRC '; SCALE INPUT TO POLE 2 WITH RIGHT 3 SHIFT'     ; MAKE NOTE
*
*REM P     ; REMOVE POLES
3 POLES/ZEROES REMOVED
*
*INCLUDE :F1:AFTER.TMP     ; GET OUR FILTER AS CODED BACK AGAIN
*DEFINE POLE 0 = -34.724136,986.55566,TS
*DEFINE POLE 1 = -97.537277,794.43249,TS
*DEFINE POLE 2 = -139.363952,438.91845,TS
*DEFINE POLE 3 = -155.340621,0.00000000,TS; REAL
*
*
*EVALUATE LOG(MAGAIN)/LOG(2)     ; SCALING FACTOR NEEDED BEFORE FOURTH STAGE
1.33596049*10**1
*
*; WE HAVE ALREADY SCALED BY 2**15 SO NO FURTHER SCALING IS NEEDED.
*
*APPEND :F1:CODE.SRC '; NEEDN''T SCALE INPUT TO POLE 3'     ; MAKE NOTE
*
*; THE FILE :F1:CODE.SRC NOW CONTAINS THE FINAL CODE FOR OUR FILTER EXCEPT
*
*; FOR THE SCALING (WHICH IS INDICATED ONLY BY COMMENTS) AND THE INPUT
*
*; AND OUTPUT ANALOG SECTIONS.  THESE CAN BE ADDED BY EDITING THE FILE
*
*; MANUALLY.  WITH CARE IT IS POSSIBLE TO AVOID THIS EDITING STEP AND CREATE
*
*; THE FINAL SOURCE CODE FILE HERE.
*
```

```
*GRAPH STEP   ; JUST OUT OF CURIOSITY LET'S LOOK AT THE STEP RESPONSE

STEP      !............^......^.....^......^.....^......^.....^......^........!
10395                     -''                         .--.
 9900                  . -   '                      .'    '--....--''--.....--'
 9405                         '        .-''-.      -
 8910                   '      -      .      -.  .'
 8415                           -    -         ''
 7920                  '         '--'
 7425
 6930                 .
 6435
 5940
 5445                -
 4950
 4455               .
 3961
 3466
 2971              '
 2476
 1981             '
 1486            .
  991           .
  496          .
    1     .---'
  AMPISEC !............^......^.....^......^.....^......^.....^......^........!
          0.0000   0.001 0.0015 0.002 0.0025 0.003 0.0035 0.004 0.0045   0.0052
*
*
*
*; RISE TIME IS ABOUT A MILLISECOND AND FINAL AMPLITUDE IS ABOUT ONE THIRD.
*
*; OF OUR TOTAL SCALE FACTOR 2**15 (ABOUT 32K).
*
*EVALUATE 2**15    ; ABOUT 32K
3.2767984*10**4
*
*EXIT    ; THAT'S ALL FOR TODAY
```

CHAPTER 2
LANGUAGE ELEMENTS

Introduction

The SPAC20 software provides you with an easy-to-use English language command set for controlling SPAC20 execution in a variety of interactive functions. Commands are keyed in one line at a time, each ending with a carriage-return, and are more fully discussed in Chapter 3. The current chapter deals with the more elemental components of the Compiler language, the building blocks out of which expressions and commands are later created.

An example of one complete SPAC20 command is shown in Figure 2-1. This command is made up of separate tokens or mnemonic codes (character strings): DEFINE, POLE, etc. Each of these tokens provides a particular element of information necessary to inform the SPAC20 Compiler of the specific action to be taken. Table 2-1 defines the function of each of these tokens. Every SPAC20 command is composed of one or more such tokens.

```
DEFINE POLE 12 = -10, 250, TS
```

Figure 2-1. Example of a DEFINE Command 121533-44

Table 2-1. Token Functions in the Above Command

Token Number	Name	Function
1	DEFINE	Command keyword; causes definition of some object, in this case a pole
2	POLE	Object keyword; names a type of object to be dealt with in this command
3	12	Constant token; used here as the label of the pole being defined
4	=	Operator token; indicates creation or replacement of the object to its left using the values given to its right
5,6	−10	Unary operator and constant token; denotes decimal negative ten
7	,	Punctuation token; separates other tokens, here −10 and 250 and TS
8	250	Constant token; denotes decimal two hundred fifty
9	,	Punctuation token as above
10	TS	Modifier token; indicates which plane (of three) is to contain the pole or zero (i.e., sampled S-plane TS as opposed to either CONTINUOUS or sampled Z-plane Z.)

This command defines a sampled pole, labeled 12, at X-Y coordinates (−10), (250) in TS (the sampled S-plane whose sample rate you would have set earlier in a separate command).

Thus, the SPAC20 command language is composed of a character set and vocabulary of mnemonic tokens. The character set is used to construct mnemonics and, in turn, the mnemonic tokens are used to construct SPAC20 commands.

Character Set

The valid characters in the SPAC20 command language include upper- and lower-case alphabetic characters A through Z and the set of digits 0 through 9. The space serves to indicate the end of a token, and carriage-returns or line-feeds are used for delimiting (ending) command input lines. The question mark [?], at-sign [@], underline [_], and dollar sign [$] are also valid in user-defined names.

Other valid characters are the ASCII (American Standard Characters for Information Interchange) algebraic operators [+] and [−] (binary and unary), asterisk [*], slash [/], relational operators [=,<,>,], ampersand [&], semicolon [;], period [.], parentheses [(,)] and comma [,]. Special characters listed below are valid in certain contexts. All other characters are ignored unless occurring within comments or strings, as discussed below. ESCape interrupts processing and is not legal within a command.

Alphabetic characters are:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Numeric characters are:

```
0123456789ABCDEF (last six only allowed as hexadecimal digits)
```

Special characters are:

```
+ - > = < $ ' & ) . ( , / ; ? * _ @ : %
```

This character set is used to construct the vocabulary that constitutes the command language. The special characters, explained briefly below, are discussed more extensively where they are directly relevant, e.g., in user names or expressions.

Special Character Usage

A semicolon that is not included in a string (defined below) causes the semicolon and the rest of the current input line to be treated as a comment. Blank lines are allowed and ignored, i.e., lines containing only a carriage-return (CR), a line-feed (LF), or both (CRLF). Ampersands outside of comments or strings permit input command lines (Chapter 3) to be continued on the next input line. The other special characters above have meaning in numeric expressions, discussed later in this chapter and in Chapter 9. An example of a comment is

```
DEFINE POLE 8 = 0, 100; at 100 Hz
```

The special characters [@, _, and ?] are allowed in user names (Chapter 3), as in @POLE_12 or ?WHICH_INCREMENT. Dollar signs embedded within user-names or constants are ignored but echoed as input, providing visual separation as in the name FILTERONEGAIN.

Tokens

A token in SPAC20 command language is roughly equivalent to a "word" in the English language. It consists of a string of alphanumeric or special characters, sometimes augmented by the one-character prefix (period) that serves to identify user-symbols.

All SPAC20 mnemonics are referred to as tokens or special tokens. Tokens encompass identifiers, strings, keywords, and numeric constants (integer and floating point). Special tokens include relational operators, arithmetic operators, logic operators, and punctuation.

The following special-character sequences are tokens in the SPAC20 command language:

```
+ - = > . < , ) * ( / <= >= <> **
```

Example:

```
.A+.B PI>=(.X+.Y/HPI) (SIN(2*PI*.FREQ))**2
```

Their uses as operators and punctuation are discussed further in later sections on arithmetic and logical expressions.

Identifiers

You create an identifier as a sequence of alphanumeric characters, at-signs, question marks, and underlines. The first character of an identifier must not be a digit (or dollar sign). Only the first 31 characters of an identifier or constant are significant, and additional characters are ignored.

Strings

A string is a sequence of characters preceded by an apostrophe (') and extending to the next apostrophe. Any character, printing or non-printing, is allowed in a string. A doubled apostrophe embedded in a string denotes the use of one apostrophe as part of that string, rather than the end of the string itself.

Examples:

```
'This is a string of 33 characters'
WRITE 'This string of 54 characters gets sent to the console.'
'This 56-character string isn''t complex, don''t you agree?'
```

(When a macro is invoked (called) with an actual-parameter that includes a comma or a quote, the entire parameter must be sent as a string, i.e. preceded and followed by a quote, and the quotes within must be doubled [see Chapter 9])

Keywords

The SPAC20 Compiler recognizes a fixed list of predefined tokens called keywords, divided loosely into four classes: commands, objects, modifiers, and constants/operators/functions. Most of these have short abbreviations, and none are checked for spelling beyond the first 3 characters. Here is a partial list of keywords:

Commands	Objects (Note 1)		Modifiers	Constants,	Operators,	Functions
APPend	AGAin	*MSQe	AT	HPI	MASK	COS ACOS
CODe	*BOUnds	PERror	AUTo	PI	**	SIN ASIN
COUnt	*ERRor	PHAse	BY	TPI	*	TAN ATAN
DEFine	FSCale	POLe	THRough		/	EXP LOG
DISplay	GAIn	PZ	TO		+	SQR
EVAluate	GERror	STEp	ON		-	ABS
EXIt	GREf	SYMbol	OFF		MOD	
GRAph	GROup	TS	Z			
HELp	IMPulse	UBOund				
HOLd	*INSt	XSIze				
INClude	LBOund	YSCale				
LISt	MACro	YSIze				
MOVe	*MAGain	ZERo				
OGRaph	*MERror					
PUT						
REMove						
WRIte						

(The complete list, including definitions, appears in Appendix B.)

Note 1: If you enter any object name as a command, the current value(s) will be displayed. The underlined words may be read or written (changed); the other objects either require additional keywords (discussed in later chapters) to manipulate them, or are functions computed by SPAC20, or are read-only (indicated by a single asterisk to their left).

* read-only.

The table below shows a different view of these keywords.

	Scalars	Non-Scalars
Changeable	XSIZE YSIZE TS	LBOUND UBOUND FSCALE GREF
Non-Changeable	ERROR INST MAGAIN MSQE MERROR	AGAIN GAIN GERROR PHASE GROUP STEP IMPULSE

Scalars have single numeric values. Non-scalars have either multiple numeric values, like GAIN and PHASE, or non-numeric values, like GREF and LBOUND. The scalars comprise a category of keywords usable in expressions as well as in display commands: keyword references. This category is discussed further later in this chapter, and used in the examples of Chapter 10 and 11.

Examples:

```
TS = 1/13020
GREF = 1 AT 450; reference gain is 1 dB at 450 Hz
LBOUND = 10 AT 500, 20 AT 1500; lower bound on gain is 10 dB at 500 Hz
; rising linearly in dB against a log f scale to 20 dB at 1500 Hz (see
; Chapter 5)
```

In the command

```
MOVE POLE 12 BY .DELTA_REAL, .DELTA_IMAG
```

MOVE is the command keyword, POLE is an object keyword, and BY is a modifier keyword. The token "12", a constant, is seen by context as a label identifying which pole is to be moved. The tokens .DELTA_REAL and .DELTA_IMAG are recognized as user-defined symbols by the presence of the leading period.

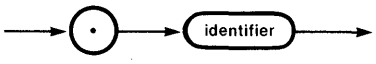
The scalar keywords are called keyword references. They are used to display, or access in expressions, all of the scalar numeric-valued system variables. They are used in three ways in SPAC20 Compiler commands:

- When one appears in an expression, the value used or displayed is the contents of the referenced object at the time the expression is evaluated.
- When one appears alone for display, its current contents are displayed.
- When one appears on the left side of "=", indicating a change, the contents of the referenced object are set to the current value of the expression on the right side of the "=". If the value on the right represents an illegal value for the referenced object, an error is reported instead. For example, TS=−3 would result in an error because a negative sampling-interval is meaningless.

Keyword Reference	Brief Description	Read/Write Status	Integer/Floating Point
TS	Sample time (in seconds) for poles and zeros in TS plane	Both	Positive floating point
XSIZE	Width of console display screen	Both	12 <= integer <= 79
YSIZE	Height of console display screen	Both	4 <= integer <= 25
MAGAIN	Maximum AGAIN (absolute gain) considered over the frequencies on FSCALE	Read-only	Positive floating point
MSQE	Mean square error in gain as compared to gain bounds, considered over the frequencies on FSCALE	Read-only	Positive floating point
MERROR	Maximum absolute error in gain as compared to gain bounds, considered over the frequencies on FSCALE	Read-only	Positive floating point
INST	Number of AS2920 instructions generated by most recently entered CODE command	Read-only	Positive integer
ERROR	Signed error in multiplier from last CODE command	Read-only	Floating point

TS has initial value 0. This value cannot be legally entered from the keyboard, and thus indicates that you have not yet specified an actual value. TS must be assigned a value before it is used, that is before any sampled (in TS or Z- planes) poles or zeros are created. An error will also be issued if it has not been assigned a value when it is needed in any other context, e.g., in calculating a filter response like IMPULSE or STEP, or when turning HOLD ON (see Chapter 5).

User Names



Symbolic Reference Chart 121533-08

The command language permits you to use symbolic (as opposed to numeric) references to variables and constants through the use of these names.

Symbols you create are stored in a symbol table. One way you can create a symbol is by using the DEFINE command, e.g.,

```
DEFINE .THETA_4 = 2.718281*4*PI
```

(The full syntax for DEFINE appears later in this chapter.) Other methods will be covered in later chapters. Symbols can be DEFINEd or REMOVEd from the symbol table.

A user symbol preceded by a period is called a symbolic reference. When a symbolic reference appears as part of a command, its value is taken from the symbol table. It may be used anywhere such a value is valid, e.g., a floating point value may not be used where an integer is required.

There may be intervening spaces between a period and the identifier following, but they are not part of the symbol.

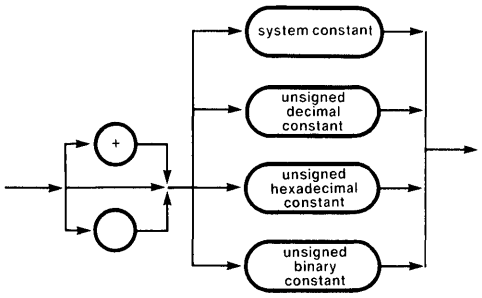
When you initialize the Intellec system with the SPAC20 module, you can restart a prior design session. You do this by INCLUDing the file of commands and parameters you earlier created using a PUT or APPEND command (see Chapter 8). By typing such an INCLUDE, you can get back the symbol table used in that earlier session (as well as certain other parameters you also sent to that file, by name or by default as described in Chapter 8).

You can add symbols to the symbol table, or remove any or all symbols currently stored there. The SPAC20 symbol table therefore contains symbols INCLUDEd or DEFINEd but not yet REMOVEd.

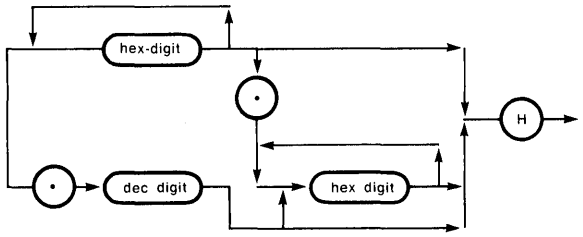
Examples of user-defined identifiers (user-names):

```
.VAR123        .GAN1  .FAZ23  .VAR66     .ERR@INST3
.F_OF_T_@_P1  .F$T$@ZER$3    .@MYFLAG
```

Numeric Constants



numeric constant 121533-06



unsigned hexadecimal constant 121533-06

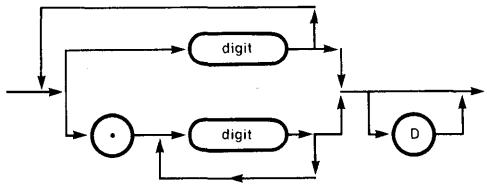
A constant is a token that represents a fixed numeric value. The SPAC20 compiler recognizes numeric constants, including floating point constants, as well as the system constants named above. A numeric constant is assumed decimal unless it carries an explicit suffix of H (for hexadecimal) or B (for binary). An explicit suffix of D means decimal. If a constant contains characters invalid in the designated number base, it will be flagged as an error.

Examples of valid numeric constants:

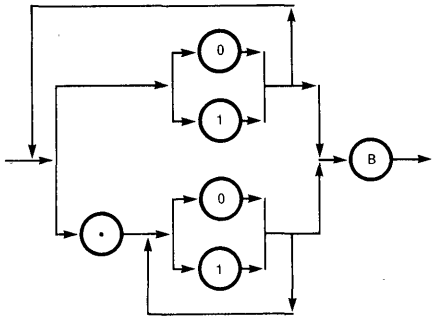
12AH	'12A' is valid in hexadecimal	2.71828
12D	'12' is valid in decimal	31.4159
1011011B	'1011011' is valid in binary	A.2CFH
0.1111$1111$1111B	dollar signs are ignored	.001B

Examples of invalid numeric constants:

12AF	Hexadecimal digits used without an H suffix, hence invalid in the default (decimal) interpretation.
12AD	Here the final D could be a suffix but the A is not a decimal digit. If hexadecimal is intended, a final H is necessary.
101A2B	'A' and '2' are not valid binary digits. If hexadecimal is intended, a final H is necessary.
2ADGH	'G' is not a valid hexadecimal digit.
1.B	Needs to be 1.0B or 1.BH
13.	May not end with point
E.4C	Needs suffix H



unsigned decimal constant 121533-06



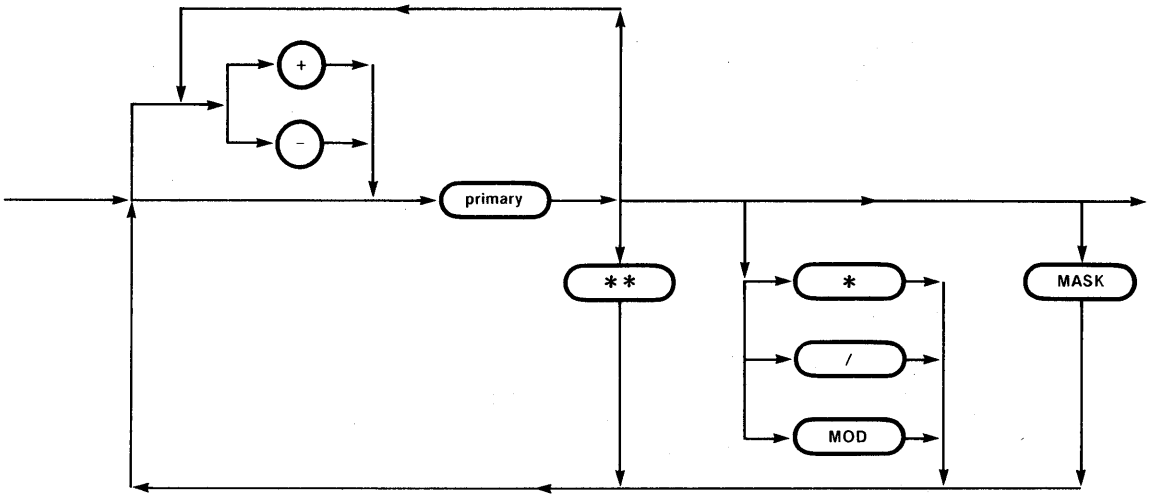
unsigned binary constant 121533-06

All numeric values are stored internally as Intel-standard-format single precision floating point numbers, positive or negative, ranging in magnitude from 1.2*10**−38 to 3.4*10**+38. They have mantissas with 24-bit precision. Manual number 9800452, entitled *8080/8085 Floating Point Library*, discusses this standard format.

A numeric constant may appear as (or contain) a point followed by a fractional part. (It may not end with a point only.) However, a leading zero is needed in the case of a hexadecimal constant like .FFH, which would otherwise be interpreted as a symbolic reference whose value is to be taken from the symbol table. Constants like 0.FFH or .2FFH will not be interpreted as symbols.

Dollar signs ($) are allowed within numbers and are ignored.

Arithmetic Expressions



An integer-expression is an arithmetic-expression which evaluates to an integer.

Arithmetic Expression 121533-07

An arithmetic expression is a construct of numeric-valued operands and operators that evaluates to a numeric-value. (The fully general definition of expressions is not needed until Chapter 9, where it appears.)

The SPAC20 Compiler evaluates expressions in a left-to-right scan modified by operator precedence, following an algebraic sequence in the form:

```
operand [operator operand]. . .
```

Operators and operands are explained below. Examples of arithmetic expressions include

```
5+7  .AVAR1*.AVAR2  (.BVAR1+.CVAR5)/(.AVAR1+.AVAR2*7)
```

Primaries are a restricted set of expressions, whose charts appear later in this chapter.

Operators

Operators are used in expressions and in commands. A summary of SPAC20 operators is shown below. The binary (arithmetic) operators are listed in their group order of precedence from highest precedence to lowest, i.e.,** has highest precedence, MOD, *, and / have equal precedence higher than + or −, which have equal precedence. MASK, which gives bitwise conjunction of two quantities, has lowest precedence. When several operators of the same precedence appear in an expression, they are evaluated left to right. Expressions in parentheses are evaluated first, before any external operators are applied.

Type	Operator	Interpretation
Precedence	( )	Controls order of evaluation
Binary (arithmetic)		
	**	Exponentiation
same precedence	MOD	Remainder EXAMPLES: 5 MOD 3 = 2; 10 MOD 3 = 1; 15 MOD 3 = 0
same precedence	*	Multiplication
same precedence	/	Division
same precedence	+	Addition
same precedence	–	Subtraction
same precedence	MASK	Bitwise AND EXAMPLES: 1.011B MASK .1B = 0.0B 1.011B MASK .111B = 0.011B
Unary-op	+	single positive quantity
	–	single negative quantity

Operands

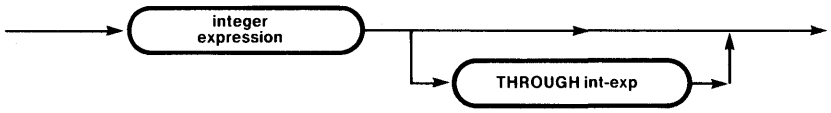
Operands are numeric values, and have the general forms shown below. These are the "primaries," which are allowed as restricted expressions in later discussions of SPAC20 commands.

		Examples
[unary-op]	numeric constant	+4, –PI, HPI, 2.71
	symbolic reference	.ALPHA, .BETA_1
	keyword reference	TS, XSIZE
	function(expression)	SIN(45/PI), SQR(.XVAR)
	(expressions)	(EXP(–.A*.TIME)*SIN(2*PI*.FREQ1)
	frequency response function	GAIN(60), PHASE(.25/TS)
	coordinate (p/z-expression)	REAL (POLE 12), IMAG (ZERO 9)

The functions referred to above are a familiar group: sine, cosine, tangent, arcsine, arccosine, arctangent, square root, absolute value, and the natural powers and logs (to the base e = 2.718281...). They may appear anywhere a floating point value is appropriate. They are evaluated, in a left to right scan of the complete expression, subject to the precedence hierarchy explained above. Their arguments, within parentheses, are of course evaluated before the function is computed.

The last operand in the list above, "coordinate (p/z expression)," represents a set of four functions: REAL, IMAG, RADIUS, ANGLE. These return the real or imaginary coordinate of a pole or zero defined in the S-plane, or the radius or angle of a pole or zero defined in the Z-plane. The angle is always assumed in radians.

Partitions



Partition 121533-05

Partition refers to a range of poles or zeros. It is specified in the form

arithmetic-expression

or

arithmetic-expression [THROUGH arithmetic-expression]

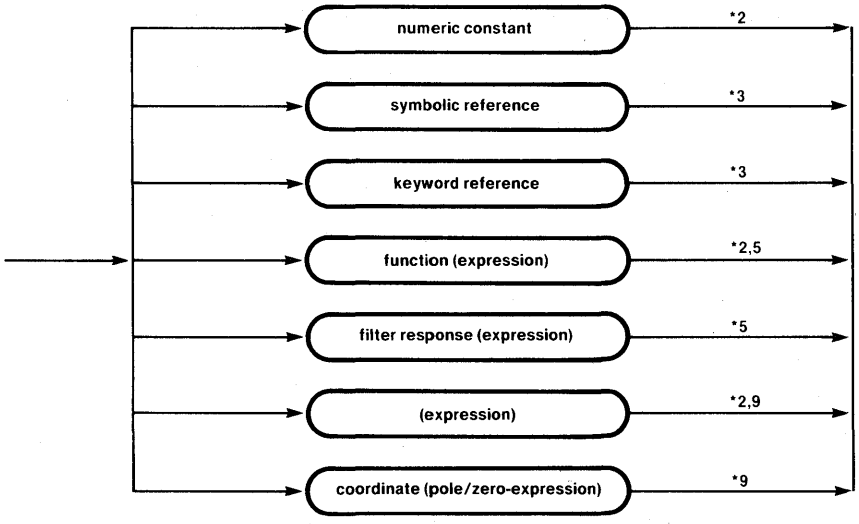
which will cause a command to affect all poles or zeros (whichever is specified) that fall in the range. Each expression is evaluated to a number, and the two numbers designate the range.

Example:

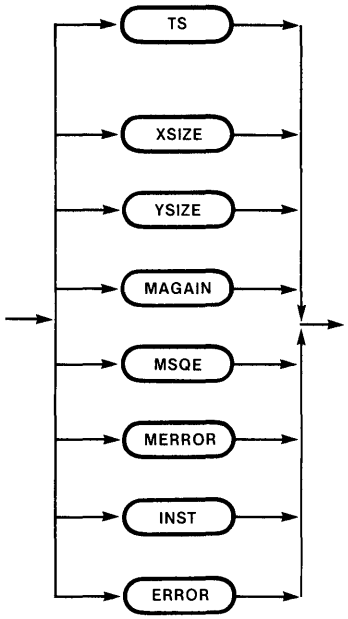
REMOVE POLE 1 THROUGH 13
This command will remove all poles numbered from 1 to 13 (inclusive) from the table of currently-defined poles (see Chapter 4).

Charts for Primaries

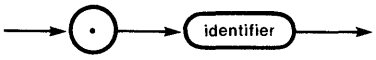
The charts below show the names and forms of all items usable as primaries, i.e., the set of restricted expressions permitted where the word "primaries" appears in a syntax chart.



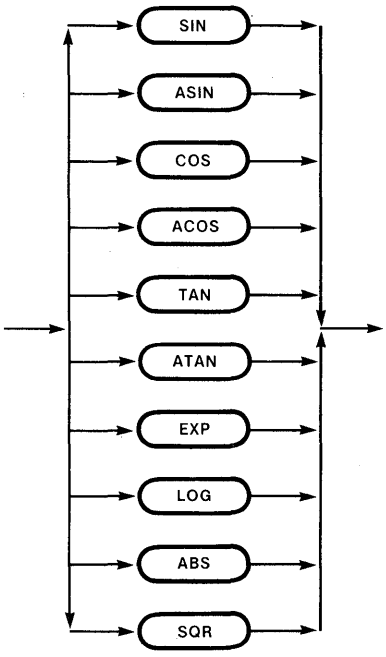
Primaries 121533-04



Keyword Reference 121533-08

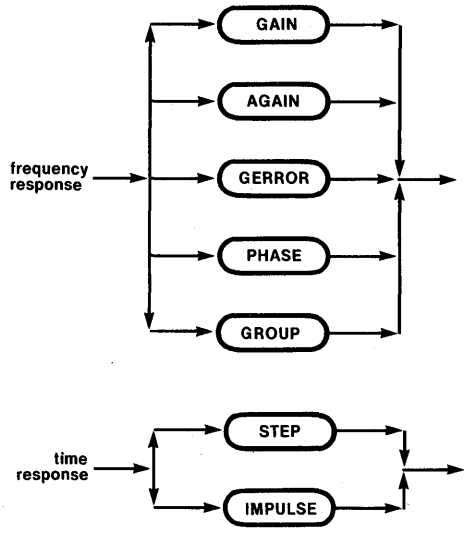


Symbolic Reference 121533-08

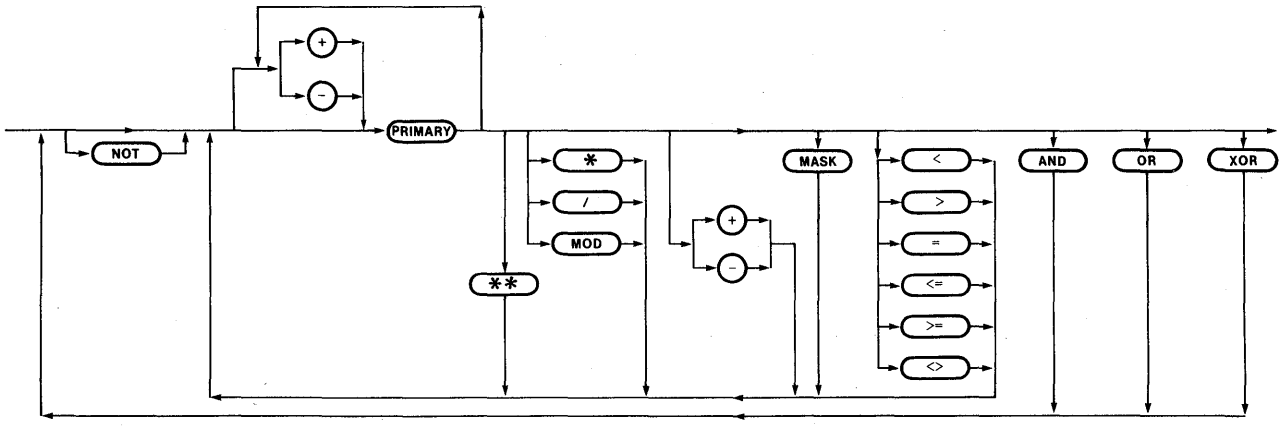


EXPonentiation and LOGarithms
to the base e = 2.718281

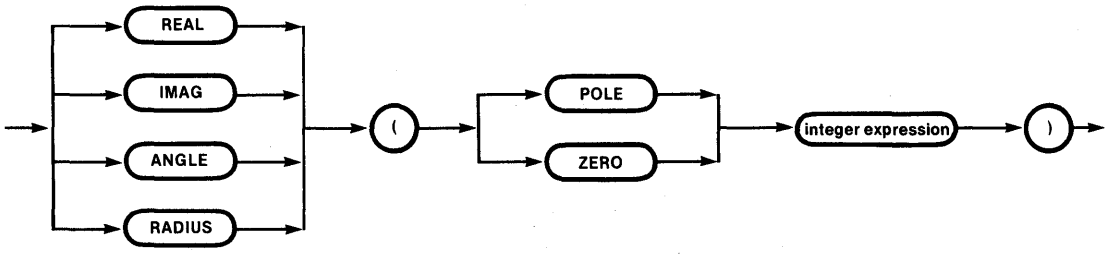
Functions 121533-28



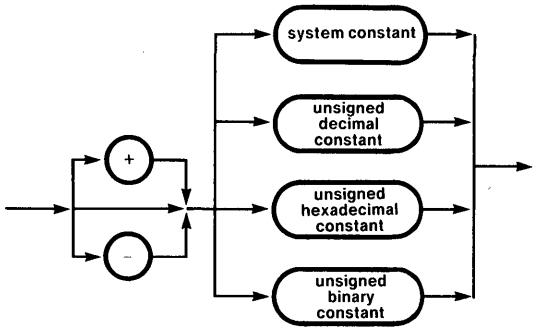
Filter Responses 121533-09



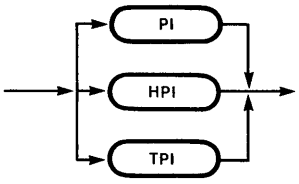
Expression 121533-26



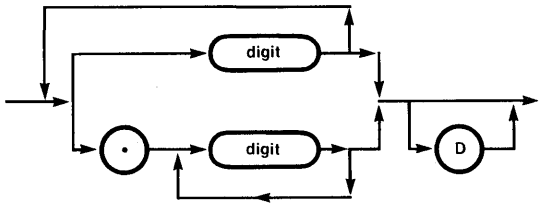
Coordinate (Pole/Zero Expression) 121533-43



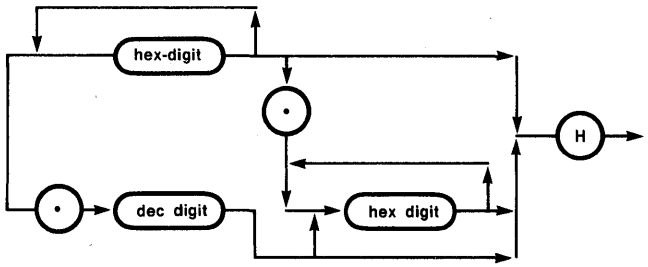
numeric constant



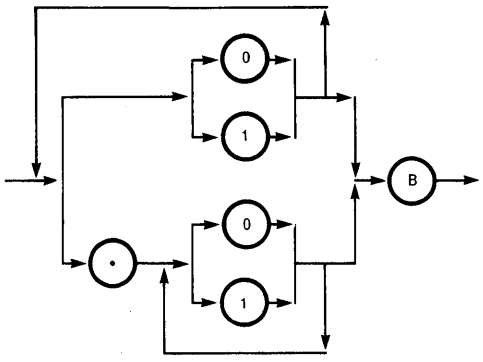
system constant



unsigned decimal constant



unsigned hexadecimal constant



unsigned binary constant

121533-06

CHAPTER 3
INTRODUCTION TO SIMPLE COMMANDS

The SPAC20 capabilities described in the Preface are reflected in the Compiler commands, constructed from the elemental units discussed in Chapter 2. The SPAC20 Compiler accepts as input both simple and compound commands. Simple commands are discussed in Chapters 3 through 8. Compound commands use sequences of simple commands, combined with control commands which determine the flow of control, i.e., branching and looping. Compound commands are discussed in Chapter 9.

This chapter describes the structure of commands and the procedure for typing them in for execution. It then introduces the simplest commands: those which define, change, or display the values of symbolic objects.

Entering and Editing Command Lines at the Console

The SPAC20 Compiler displays an asterisk prompt (*) at the left margin when it is ready to accept a command from the console.

You enter commands (one or more tokens) through the keyboard, terminating each with a carriage return [CR] (or a line feed [LF]). The system then executes the command.

Tokens in the command are separated by blanks unless the construct requires another form of separator. For example, tokens in a list are separated by commas; in this case, blanks may be inserted for clarity but are not required.

An input line may include comments. A semicolon (;) must precede the comments. If the input line contains any portion of a command, this must precede the semicolon, for characters in a comment are not interpreted by the Compiler and are not stored internally. The main use of comments is to document a design or code session while it is in progress. This is particularly useful when executing compound commands or running the Compiler under the ISIS-II SUBMIT facility.

ISIS-II limits input lines to 120 characters maximum. Commands longer than this can be broken up into more than one input line by entering an ampersand (&) just prior to the line terminator. (The & must not be contained in a comment or string.) The system acknowledges continued lines by prompting with two asterisks (**). Characters between the ampersand and the line terminator are ignored, and the ampersand is treated as a space.

You can use ISIS-II editing capabilities to correct errors in the current input line. Once a line terminator (carriage return or line feed) has been entered, that line can no longer be edited.

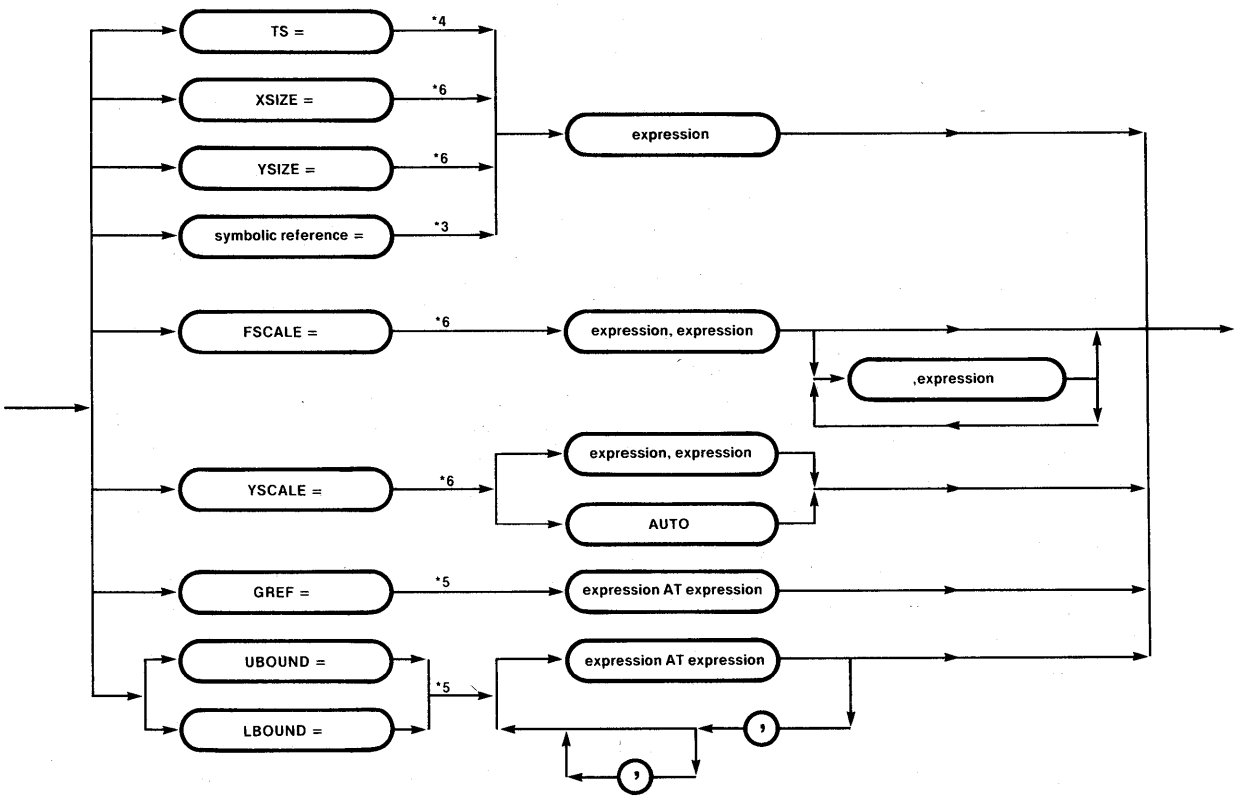
The line-editing characters are as follows:

Characters	Results
RUBOUT	Deletes last character entered in current input line. The deleted character is echoed immediately. (However, in the latest versions of ISIS-II the cursor backs up over the offending character, deleting it from view.) The RUBOUT function can be repeated, deleting one character each time it is pressed.

CTRL X	Deletes entire current input line.
CTRL R	Displays entire input line as entered so far. This is useful after a RUBOUT, to review which characters have been deleted.
ESC	Cancels entire command being entered or executed.
CTRL P	Inputs next character literally.
CTRL S	Stops display temporarily.
CTRL Q	Continues display interrupted by CTRL S.
Carriage Return	Terminates input line or command line; if command, begins processing.
Line Feed	Terminates input line or command line; if command, begins processing.

Setting or Changing Symbol Values: Equal Sign, DEFINE, REMOVE

The Change Commands



The Change Commands 121533-02

When a user-name preceded by a period appears in a command, it is called a symbolic reference. The value of any symbolic reference or any writeable keyword reference may be changed by entering the reference to be changed on the left side of an equal sign (=), followed by the new value on the right side. If the keyword reference is read-only, as mentioned in Chapter 2, you get an error.

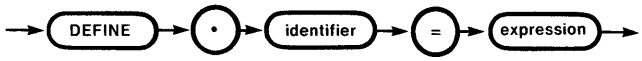
Examples:

```
TS = 1/13020
.ALPHA = 4.4
.BETA_1 = EXP (.ALPHA / TS)  ;= 2.718281 ** (4.4/13020)
.BETA_2 = -.BETA_1 * EXP (2*.ALPHA)
.ERR$SAVE = 0.5
.ORIG_POLE_1_REAL = 0
.ORIG_POLE_1_IMAG = 100
```

(The examples above assume that the symbolic references used (e.g., .ALPHA) are already defined, as discussed in the next section. Otherwise these change commands would be rejected by the Compiler as errors.)

```
XSIZE = 55  ;graph columns; see Chapter 6
YSIZE = 22  ;rows
UBOUND = 500 AT 1500  ;dB at Hz (Chapter 5)
FSCALE = 100,500,1500,4500  ;in Hertz (Chapter 6)
```

The DEFINE Command for Symbols



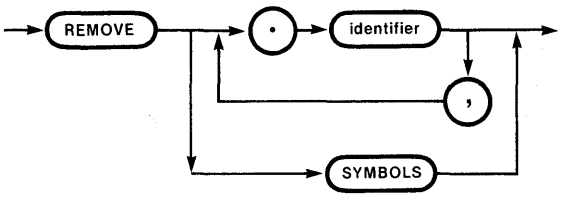
The DEFINE Command for Symbols 121533-11

The DEFINE command places the symbol you supply into the SPAC20 Compiler's symbol table, and associates with it the floating point value of the expression you give with it. This symbol table value remains fixed until you issue a change command. Symbolic integers are recognized when used where needed (e.g., as pole labels) despite being stored as floating point values. If the symbol already exists, you get an error. (However, you may change the value of an existing symbol by the change command, as above.)

Examples:

```
DEFINE .GAIN_POLE_ONE = GAIN(60)
DEFINE .POLE_TWO_GAIN = 2.34
.GAIN_POLE_ONE = 0.78
DEFINE .RADIUS_POLE_8 = RADIUS (P 8)
DEFINE .ANGLE_POLE_8 = TPI * .FREQ * TS
 ; for later use as temporary storage for
   pole 8 coordinates, z plane.
```

The REMOVE Command for Symbols



The REMOVE Command for Symbols 121533-12

The REMOVE command keyword followed by a symbol in the table causes that symbol to be deleted from the table. REMOVE may also be followed by a list of symbols, separated by commas, or the object keyword SYMBOLS, which causes *all* symbols to be deleted from the table. This does not, of course, affect system-defined keywords.

```
REMOVE .GAIN_POLE_ONE
REMOVE .POLE_TWO_GAIN
REMOVE .RADIUS_POLE_8, .ANGLE_POLE_8
REMOVE SYMBOLS
```

Displaying Object Values

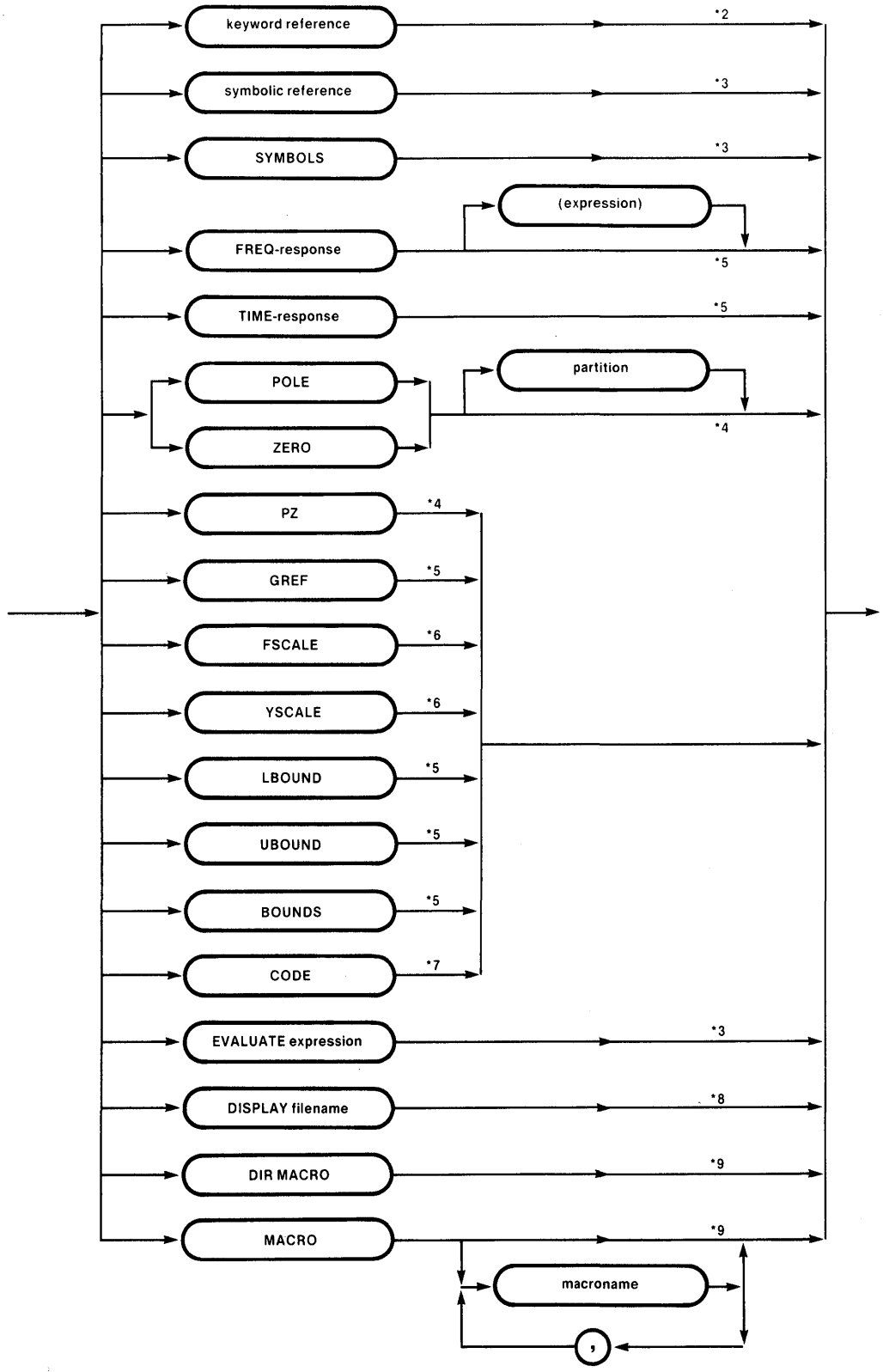
The current value defined for keyword and symbolic references may be displayed by entering the reference as a command; i.e., followed immediately by a line terminator. Its value will be shown on the next line.

Entering the keyword SYMBOLS as a command causes every symbol in the symbol table to be displayed with its associated numeric value.

The EVALUATE command displays the decimal value of the expression you enter. This command can be used as a keyboard calculator to compute the value of any arbitrary expression. For example,

```
EVALUATE TPI
```

will display the value 6.2831852.



Display Commands 121533-03

CHAPTER 4
POLE AND ZERO HANDLING

Planes and Coordinates For Poles and Zeros in DEFINE, MOVE, REMOVE Commands

When you define the location of a pole or zero, you may do so in one of three ways:

- in the Z-plane, giving its radius and angle (in radians),
- in the S-plane,
- or as a Z-plane pole or zero which is defined by an equivalent S-plane location using the matched Z transform. The sampled S-plane specification is TS, as described in the next section.

The Compiler maintains a table of all poles or zeros currently defined. Continuous filter sections are presumed to be implemented outside the 2920 processor, and if no plane is specified, CONTINUOUS is assumed. Being able to combine sampled and continuous sections allows you to evaluate the effect of external anti-aliasing filters.

Sampled poles and zeros are created or moved in the S- or Z-plane, using the predefined units and coordinates specific to that plane. This means if the plane specified is Z, the coordinates you give are used as the radius and angle (in radians) which locate that pole or zero in polar coordinates.

If the plane specified is not Z, then the Cartesian coordinates you give are used as the real and imaginary part of that pole or zero. S-plane units correspond to Hertz.

Sampled S-plane poles or zeros are actually mapped to the Z-plane during calculation, using the matched Z-transform, i.e., a pole or zero at $x+jy$ on the S-plane is transferred to a pole or zero at $e^{2\pi TS(x+jy)}$ on the Z-plane, where TS represents the sample interval in seconds. In polar coordinates, this Z-plane location is ($e^{2\pi TSx}$, $2\pi TSy$).

NOTE

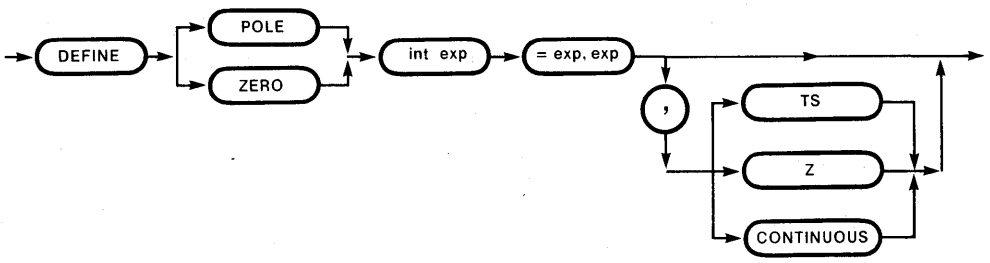
One consequence of choosing the TS plane is that the frequency at which a pole or zero is defined is fixed and independent of the sampling interval. Doubling the sample rate has no effect on the frequency. Different sampling rates will, however, cause different geometric coordinates when such a pole or zero is mapped to the Z-plane, and different 2920 code to implement the filter.

Conversely, defining a pole or zero in the Z-plane fixes the geometry, i.e., the radius and angle. The frequency, however, depends on the sample rate. Doubling the rate (halving the interval) doubles the frequency represented by the fixed polar coordinates.

For example, suppose POLE 1 in Z has polar coordinates 1.00, PI/4, with the sample interval set at 1/5000 seconds. This implies a frequency for POLE 1 of 5000/8, or 625 Hz. If the sample interval is halved to 1/10000, the geometry of the pole in the Z-plane is unchanged, but its frequency doubles to 1250 Hz. An equivalent pole specified in the TS plane keeps the same frequency regardless of sample interval.

Thus the three possible terms used for specifying the plane in a DEFINE (or MOVE) command are CONTINUOUS, TS, and Z. If no plane is given, then the default (or, for MOVE, a prior specification of plane) is used. Further, the pair of expressions used in DEFINE (or MOVE) indicate the coordinates of (or increment for) the pole or zero. These expressions will be taken to mean either real part and imaginary part (for S-plane), or radius part and angle part (Z-plane). The charts and discussions below will show all of these forms.

The DEFINE Command For Poles and Zeros



The DEFINE Command for Poles and Zeros 121533-13

As shown in the syntax chart above, defining a pole or zero begins with the command keyword DEFINE and the appropriate object, POLE or ZERO. The next token is usually the number to be used as the label for this pole or zero. However, the token may in fact be any valid arithmetic expression as discussed in Chapter 2. (For this use as a label, the expression must evaluate to an integer.)

Following this "label", an equal sign is required, leading to the two expressions which define the location of the pole or zero, separated by a comma. Usually these will simply be numbers, i.e., the real/imaginary or radius/angle coordinates specifying the desired location. However, any legal expression may be entered (see Chapters 2 and 9). As mentioned above, angles are always taken as being in radians. They must be greater than −PI and not greater than +PI. Radii must be non-negative.

The syntax chart next indicates that no further tokens are required, but you have the option of specifying the plane. To do so, you must enter a comma after the second coordinate-expression, and then pick one of the three choices shown.

Up to 20 poles or zeros can exist at one time, e.g., 12 poles and 8 zeros or 5 poles and 15 zeros, etc. Their numeric labels are arbitrary, that is, you may define them using whatever numbers you choose, in any order, e.g., 1, 5, 88, 13, 46, 22. The numbering scheme has no effect on later calculations, but you may wish to assign meaningful labels, particularly if you wish to manipulate them later with compound commands.

NOTE

The only effects of a chosen numbering scheme will appear when a partition is used, as discussed in Chapter 2, or if you need to see the individual effect of each pole or zero, which will be discussed in Chapter 5. (As a brief partition example, if you defined poles in the above order and later displayed the first few by typing POLES 1 THROUGH 20, only poles 1, 5, and 13 would be printed out.)

Each complex pole or zero represents a conjugate pair so that the filter can be realized. That is, during calculation a conjugate pole or zero is assumed to exist for each pole or zero with a nonzero imaginary part.

Conversely, a continuous pole or zero is considered real if its imaginary coordinate is zero. A sampled pole or zero is considered real if, after mapping to the Z-plane, its imaginary coordinate is zero. Thus a pole at −5,0.5/TS,TS is considered real because, at half the sample rate, it maps onto the real axis in the Z-plane.

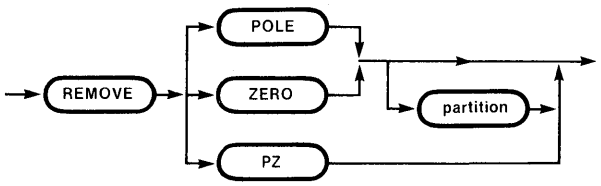
Poles and zeros are numbered independently and uniquely. You may not DEFINE a new pole or zero numbered the same as an existing object of the same type. That is, if POLE 1 exists and ZERO 1 does not, you may say DEFINE ZERO 1 but not DEFINE POLE 1. (However, you can MOVE or REMOVE it as shown below.)

Examples:

```
DEFINE POLE 1 = 0, 100, TS
DEFINE ZERO 1 = -3, 5, TS
DEFINE POLE 2 = -10, 450
DEFINE ZERO 2 = -16, 0
DEFINE POLE .GAMMA = 0.67, PI/6, Z
; these 2 examples assume that .GAMMA has been previously defined
; as an integer value.
DEFINE ZERO .GAMMA = 0.55, PI, Z
```

(Poles and zeros defined in the Z-plane must have radius >= 0 and an angle which conforms to −PI < angle <= PI)

The REMOVE Command For Poles and Zeros



Remove Command for Poles and Zeros 121533-14

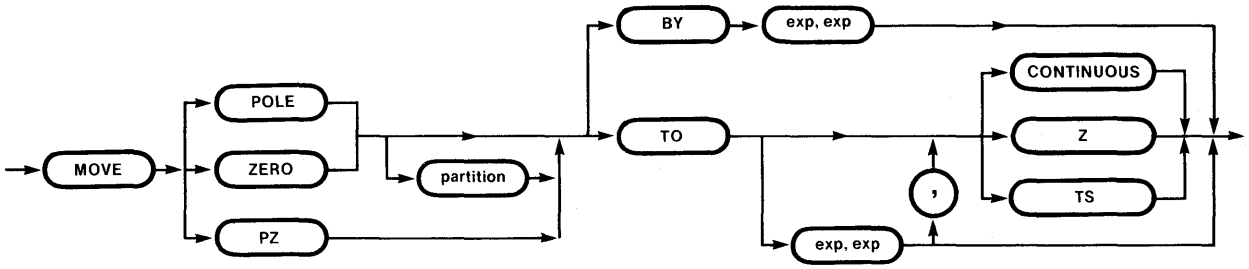
To remove one or more poles or zeros from the table, you must enter REMOVE followed by one of three object keywords. PZ means *all* poles *and all* zeros are to be removed, and no further tokens are needed for this command. If POLE or ZERO is entered with no further tokens, then *all* poles (or zeros) are removed. If there is a next token, it represents the first pole or zero to be removed. If the command is terminated here, only that pole or zero is removed.

It is also possible to remove a range of poles and zeros by using a partition, i.e., by following the first pole or zero with the token THROUGH and the label of the last pole or zero to be expunged. Usually the token identifying the intended pole or zero is simply an integer, but it is valid to use any legal arithmetic expression to specify which one is meant. (In this context, however, the expressions must evaluate to an integer. The terms partition and arithmetic expression are discussed in Chapter 2.)

Each time REMOVE is used, a message is displayed giving the number of poles or zeros actually deleted. Once a pole or zero has been removed, its numeric label can be reused in defining a new pole or zero.

```
REMOVE POLE 1
REMOVE ZERO 1
REMOVE POLE 2 THROUGH 10
REMOVE ZERO 2 THROUGH 5
REMOVE PZ
REMOVE POLES ; spelling is not checked beyond 3 characters
REM ZEROes
```

The MOVE Command



The MOVE Command 121533-15

The MOVE command contains similarities to both the DEFINE and REMOVE commands. Its objects are the same as those for REMOVE, and its modifying phrases are similar in form to those for DEFINE.

After entering the command keyword MOVE, you must choose one of the three object keywords shown. PZ means *all* poles *and* zeros are to be moved using the modifying phrases which follow. The object POLE (or ZERO) allows you to specify one or a range of poles or zeros to be moved, using a partition. If POLE (or ZERO) is entered with no further tokens, then *all* poles (or zeros) are moved as directed.

The modifying phrase

```
BY expression, expression
```

specifies an increment for each of the coordinates originally defined for this pole or zero, in the units appropriate to that plane. That is, if this pole or zero is continuous or sampled in TS, these increments are to the real and imaginary parts respectively. If the original pole or zero specification was in Z, then these increments are to the radius and angle, respectively. The new Z-plane position must satisfy RADIUS >= 0, and −PI< angle <= PI.

A cautionary note is needed here, because some moves could create unexpected conditions. If the coordinates of a complex pole or zero are moved in such a way as to make the imaginary component zero, then what were two poles (or zeros) have become one. (If two real poles are desired at the new coordinates, you must then define one new real pole there.) Similarly, if a move causes the imaginary part of a real pole or zero to become non-zero, what was one pole or zero has become two.

The use of the modifier TO permits two broad possibilities. It can cause a change of plane, directly, for one or more poles or zeros, e.g.,

```
TO TS
TO Z
```

It also can specify a new position, with or without a change of plane, for one or more poles or zeros. In the latter case, the expressions used designate final location (not increments as they do in the BY modifier), e.g.,

```
TO 20,450,TS
TO 0.75,PI/4,Z
TO -15,0
```

(In this last case, please note that when you don't specify a plane for the move command, the original plane of definition is used. This differs from the define command for poles and zeros in that omitting a plane there makes the default CONTINUOUS.) Each time MOVE is used, a message is displayed giving the number of poles or zeros actually moved.

Examples:

```
MOVE POLES 1 THROUGH 3 TO TS
MOVE ZERO 2 BY 0.875,0.125
MOVE PZ TO Z
MOVE POLE 5 to 0.625,PI/2,Z
```

Changing from a Z-plane to any other involves an application of the matched-Z transform or its inverse. This transform is not one-to-one. When its inverse is used, the value with imaginary part closest to zero is selected from the set of possible inverses. This in effect selects the lowest frequency that could be aliased by the filter.

CHAPTER 5
FUNCTIONS OF FILTER RESPONSE

The behavior of the filter defined by the existing poles and zeros can be investigated with respect to its gain, phase, and deviation from gain bounds. Group delay and time response can also be calculated and listed or graphed. The keywords used to specify reference and boundary levels and those for displaying the response are listed in Table 5-1.

Table 5-1. Keywords for Gain Reference, Gain Boundaries, and Response Display

AGAIN	G	F	
BOUNDS			
GAIN	G	F	
GERROR	G	F	
GREF			(initial value 0 AT 0, i.e., 0 dB at DC)
GROUP	G	F	
IMPULSE	G		
LBOUND	G		(initially −1000000 AT 1)
MAGAIN			
MERROR			
MSQE			
PHASE	G	F	
STEP	G		
UBOUND	G		(initially +1000000 AT 1)

When one of the above keywords is entered as a command, its current value is displayed. If it is a multivalued object, e.g., GAIN, a list is displayed. Those marked with a G are graphable (all but BOUNDS, GREF, MAGAIN, MSQE, and MERROR. Those marked with an F can act as functions with frequency arguments, e.g., GAIN(145) will display the gain at 145 Hz due to all currently defined poles and zeros.

The filter responses GAIN, AGAIN, GERROR, PHASE, and GROUP are functions of frequency. STEP and IMPULSE are functions of time. The response is calculated only for a specific range of frequencies or time. This range is determined by the setting of FSCALE, for frequency responses. XSIZE and TS determine the range for time responses (see Chapter 6).

GAIN and GREF

GAIN refers to a *normalized-gain* in decibels due to *all* existing poles and zeros. The normalization factor is the current GREF setting, namely a specified gain at a specified frequency. You set GREF by typing

```
GREF = expression AT expression
```

in which the first expression is the reference gain at the frequency given by the second expression. The frequency specified by the second expression need not be contained in the range of frequencies you set up as FSCALE. The initial GREF setting is 0 AT 0, which is to say, the gain at DC is 0 dB.

The GREF frequency must have nonzero absolute gain in order to compute the decibel GAIN. If AGAIN is zero at the reference frequency, an error message will be issued.

When multiple poles exist, to see the gain for each individual pole you must remove them one at a time and compare the resulting filter responses with the earlier values. You can achieve this comparison by graphs or lists, or by defining symbols for storing each intermediate value.

AGAIN and MAGAIN

AGAIN refers to the *absolute-gain*, expressed as a multiplier, again due to *all* existing poles and zeros. This absolute gain can only be meaningfully determined for sampled poles and zeros. If nonsampled poles and zeros exist, the AGAIN will include a contribution for each such pole or zero, which is arbitrarily scaled. AGAIN is useful only when all currently-defined poles and zeros are sampled. For the case of continuous poles and zeros, GAIN is much more meaningful than AGAIN.

The maximum absolute gain, MAGAIN, taken over the 64 or so frequencies contained in FSCALE, is accessible as a read-only keyword reference. This quantity can be displayed by name or used in expressions, and is useful in determining the scaling factors necessary between successive stages of the filter.

NOTE

MAGAIN is only the maximum AGAIN over the frequencies in FSCALE, and not the true maximum AGAIN. The true maximum AGAIN may occur *between* points in FSCALE or *outside* the FSCALE range entirely. It is therefore necessary to choose FSCALE appropriately to capture the frequency range of interest, or manipulate FSCALE to focus in on that range, such that seeing a smooth curve will correctly imply there are no hidden spikes.

Upper and Lower Bounds

The bounds are piecewise linear functions of frequency, with possible regions of "don't care," meaning any gain is acceptable therein. The region boundaries are specified as

```
expression AT expression
```

meaning a gain of the first expression at the frequency determined by the second expression. Up to 10 lower bounds and 10 upper bounds may be specified.

The initial lower bound, LBOUND, is −1000000 AT 1; the initial upper bound, UBOUND, is 1000000 AT 1. For all practical purposes these bounds amount to a "don't care" condition. You can use similar settings to obtain this condition at any frequency.

You set the bounds by typing one or more frequencies, separated by commas, after the word LBOUND or UBOUND and an equal sign, e.g.,

```
LBOUND = 1 AT 5000,1.50 AT 6500,3.30 AT 13000
```

This means the minimum acceptable gain at 5000 Hz is 1 decibel, rising to 1.50 decibels at 6500 Hz, rising to 3.30 decibels at 13000 Hz. The frequency scale is logarithmic. The gain between these frequencies is a straight line, viewed on this log scale, connecting the specified gains at each point. The frequencies must be greater than 0 and given in increasing order, e.g., specifying

```
UBO = 1 AT 0
```

or

```
LBO = 1 AT 5000, 5 AT 4000
```

is illegal.

If the bounds are separated by two commas instead of one, this specifies a don't care region, in which any gain is acceptable, between the two given frequencies. (That is, the deviation of the gain is zero regardless of how high or low the gain is.) Similarly, the regions below the first frequency specified and above the last frequency are don't cares. If LBOUND and UBOUND are both specified, LBOUND must be less than UBOUND (or you get an error message). Bounds frequencies need not lie within the range of frequencies determined by FSCALE.

NOTE

"Don't care" conditions permit the CODE command (discussed in Chapter 7) complete latitude in frequency responses and pole/zero repositioning.

Other Filter Responses and Keywords: GERROR, MSQE, MERROR, PHASE, STEP, IMPULSE

GERROR refers to the deviation of the gain response from the bounds you set. A positive GERROR indicates a gain exceeding the upper bound set for that frequency. A negative value of GERROR means a gain less than the lower bound for that frequency.

The mean square error, MSQE, and the maximum absolute error, MERROR, are accessible as keyword references. The former is the mean square deviation from the bounds taken over the frequencies in FSCALE. The latter is the maximum absolute deviation from the bounds, taken over the same frequencies. (In order to see the exact frequency at which this maximum occurred, you must graph the error as described in Chapter 6.)

As with MAGAIN, note that MERROR is only the maximum error over the frequencies in FSCALE, and not the true maximum error. The true maximum error may occur *between* points in FSCALE or *outside* the FSCALE range entirely. It is therefore necessary to choose FSCALE appropriately to capture the frequency range of interest, or manipulate FSCALE to focus in on that range, such that seeing a smooth curve will correctly imply there are no hidden spikes.

Thus MSQE and MERROR are functions of the existing poles and zeros, the bounds, and FSCALE. MAGAIN is a function of the existing poles and zeros and FSCALE.

PHASE refers to the phase delay response of the filter, in units of radians. GROUP refers to the group delay of the filter in seconds, i.e., the negative of the derivative of the phase with respect to the frequency. (See Appendices H and I for formulas, graphs, and a brief review of these functions.)

STEP refers to the filter output in response to a unit up-step at time zero.

IMPULSE refers to the filter output in response to a unit up-impulse at time zero.

Since continuous convolutions are implemented by approximating them with discrete convolutions, the accuracy of these time responses is dependent on the setting of TS and the location of continuous poles and zeros.

In particular, those defined at high frequencies (relative to TS) will contribute noticeably to this inaccuracy. If there are any continuous poles or zeros, then the magnitude of the impulse functions is defined as 1/TS. Otherwise, the impulse is 1. All time responses for continuous poles and zeros are normalized so that the final output level to a step input matches the gain at DC. However, for sampled poles and zeros, these responses are not normalized.

Except for STEP and IMPULSE, these filter responses may also be used as functions, computing the indicated response at a frequency specified as an expression inside parentheses following the filter response keyword. The value of this expression need not lie within the range of the FSCALE (or the time scale determined by TS and XSIZE). These functions can be displayed or used in expressions, interactively or in compound commands (Chapter 9.)

Response Keyword	Computation Uses				
	GREF	FSCALE	State of HOLD	BOUNDS	Time Scale
GAIN	X	X	X		
AGAIN		X	X		
GERROR	X	X	X	X	
PHASE		X	X		
GROUP		X	X		
STEP					X
IMPULSE					X

HOLD

In typical 2920 applications, after a signal is sampled and filtered, it is usually kept in a sample-and-hold buffer until digital-to-analog conversion and output takes place. There is an implicit distortion due to this sampling, holding, and converting which amounts to a high-frequency droop cut off at 1/TS. GAIN attenuates by about 4 dB at half the sample rate. Below half the sample rate this distortion approaches zero. The attenuation increases above half the sample rate.

The correction for this distortion is to multiply AGAIN by |sin(x)/x|, where x is half the digital frequency, i.e.,

```
|SIN(X)/X| where X=TS*FREQ*PI
```

The effect of this on GAIN is to add 20*log|sin(x)/x| (to the base 10). PHASE is corrected by adding X. GROUP is corrected by subtracting TS/2. No other filter responses are affected.

The command HOLD OFF removes these corrective contributions.

HOLD is initially OFF, and in this state the SPAC20 Compiler accurately describes analog filters. Most 2920 filter designers should have HOLD ON when examining the responses of the filter as a whole, and HOLD OFF when examining AGAIN to determine inter-stage signal scaling (see Appendix J).

CHAPTER 6
GRAPHICS CAPABILITY

SCALES

All of the graphs and most of the calculations performed by the SPAC20 Compiler depend on a few key independent variables that you set early in any interactive session. The frequency and time scales are two of these. The former is FSCALE. The time scale depends on the values of TS and XSIZE, as explained below. YSCALE controls the values on the vertical scale of each graph. XSIZE and YSIZE specify the size of the CRT screen.

Frequency and Time Scales

The frequency scale (FSCALE) serves to format the graphic display and to restrict the domain of interest, i.e., the frequency range for which filter responses are computed and graphed.

The frequency response of the filter is only calculated for the approximately 64 frequencies in FSCALE, as are GERROR, MERROR, MAGAIN, and MSQE. The automatic initial FSCALE setting is

```
FSCALE = 10,10000
```

meaning 10 to 10,000 Hz. New settings are in the form

```
FSCALE = expression, expression, ...
```

permitting N expressions, (up to 10) of increasing positive values, separated by commas.

The effect of these is to divide the graphics area of the screen (determined by XSIZE and YSIZE) as evenly as possible into N-1 partitions, with the N equally spaced points given the indicated expression values. Then, within each partition, the frequency scale (in Hertz) is filled in logarithmically. This enables you to achieve a nearly linear scale or to emphasize certain frequency regions of interest. Note, however, that zero is not allowed on FSCALE, and the frequencies must be in ascending order.

Examples:

```
FSCALE = 500,5000
FSCALE = 500,1500,2500,5000
FSCALE = 500,1000,1500,2000,2500,3000,3500,4000,4500,5000
```

In the case of the time scale, the domain you create via TS and XSIZE impacts the amount of computation time needed to calculate the STEP or IMPULSE response. Time responses are computed for a number of sample intervals. The initial default value is 69. The number depends on XSIZE, e.g., if XSIZE>= 79, then 69 sample intervals are used, from 0 to 68*TS.

The time scale is in units of seconds, and its automatic setting is from 0 to the product of TS and the plot size, i.e., XSIZE-10 up to a maximum of 69. Thus if the screen width XSIZE has been set to 60, and the TS sample interval is set at 1/10000, then the time scale will run from 0 to (60-10)/10000, or from 0 to .005 seconds. TS must be nonzero, i.e., you must set it to a sampling rate meaningful to your problem.

XSIZE, YSIZE, and YSCALE

You set the system variables XSIZE and YSIZE to determine the size of the graphics area. Since three horizontal rows are dedicated to labeling the graph's X (horizontal) axis, YSIZE-3 rows are left for graphics (a minimum of 1 up to a maximum of 22). Ten columns are needed for labeling the Y (vertical) axis, leaving XSIZE-10 columns for graphics (from 2 up to 69 maximum).

XSIZE also determines the number of frequencies for which the gain of the filter is calculated, and thus also affects the calculation of GERROR, MERROR, MAGAIN, and MSQE.

The last remaining scale you set is named YSCALE, referring to the vertical scale (dependent variables). It serves to format the graphic displays obtained with the GRAPH or OGRAPH commands explained later in this chapter. If you specify YSCALE = AUTO, this means each curve plotted by the SPAC20 Compiler should entirely fill the screen, using a vertical scale selected by the Compiler to achieve this purpose.

If the numbers necessary to represent the range in YSCALE require more than 10 columns, the scale will appear as percentages instead of the actual YSCALE numbers. If this appears on your graphs, you can display the actual range used by typing YSCALE as a command, i.e., followed by a carriage-return. If the original specification was AUTO, then this will display the word AUTO followed by the actual numbers used. This situation can occur also if you specify more than 10 digits in a range for YSCALE, or if the range is so narrow as to require more than 10 columns, e.g., '10 TO 10.000001'.

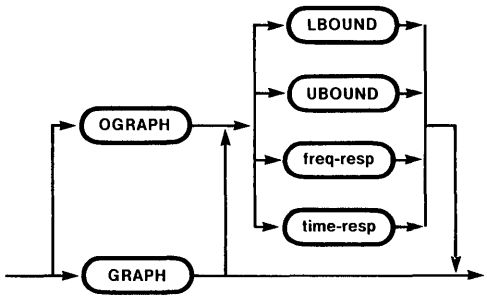
If you specify two expressions, as in setting FSCALE, this means the bottom of the display should correspond to the value of the first expression, and the top of the display should correspond to the value of the second expression. These numbers need not be in increasing order. Thus if you specify

```
YSCALE = 0,-40
```

the GAIN graph will look upside down, like an attenuation graph.

Values exceeding the range specified by YSCALE are explicitly indicated on the graph by an asterisk, denoting saturation. The vertical resolution is actually the vertical range divided by 3*(YSIZE-3), since three different characters are used to represent three different levels on each console line.

GRAPH and OGRAPH



GRAPH and OGRAPH Commands 121533-25

When used with a valid object, e.g., GRAPH GAIN, the commands GRAPH and OGRAPH fill a buffer area with characters whose positions represent the values of the object. The proper positions are selected within the graphics area you have defined using XSIZE and YSIZE (or the default area, which is the full screen).

The characters used for the latest curve are the period, dash, and apostrophe (. - '), so that effective vertical resolution is three times that of a single letter. When OGRAPH is used, the plot also contains the previous curve graphed, but with the characters all replaced with the character "+", to distinguish the older curve. The scale appropriate to the newer curve is displayed, but the physical positions of the old graph on the screen remain unchanged.

If GRAPH is used alone, with no object, the latest graph displayed is redisplayed regardless of intervening changes in any state variables. This can be used to recover a display which has scrolled off the console screen.

Hard-copies of the graphics output and all other console activity can be obtained by defining a list file or device using the LIST command (see Chapter 8). No special graphics capabilities are required of the device.

CHAPTER 7
CODE GENERATION FOR THE INTEL 2920 PROCESSOR

The Code Command and Constraints

After manipulating the pole and zero positions to get the frequency and time responses of the filter to match the characteristics you want, you may create AS2920 assembly language code for each pole or zero (or conjugate pair) with one CODE command per object.

These commands perform compilation, generating as many AS2920 instructions as are needed to correspond either to a pole or zero or to an equation. (Such equations are useful in propagating and scaling the signal passed between filter stages.) The code generated by the most recent CODE command is maintained in a code buffer which can be displayed (by simply entering CODE), or sent to a file (Chapter 8).

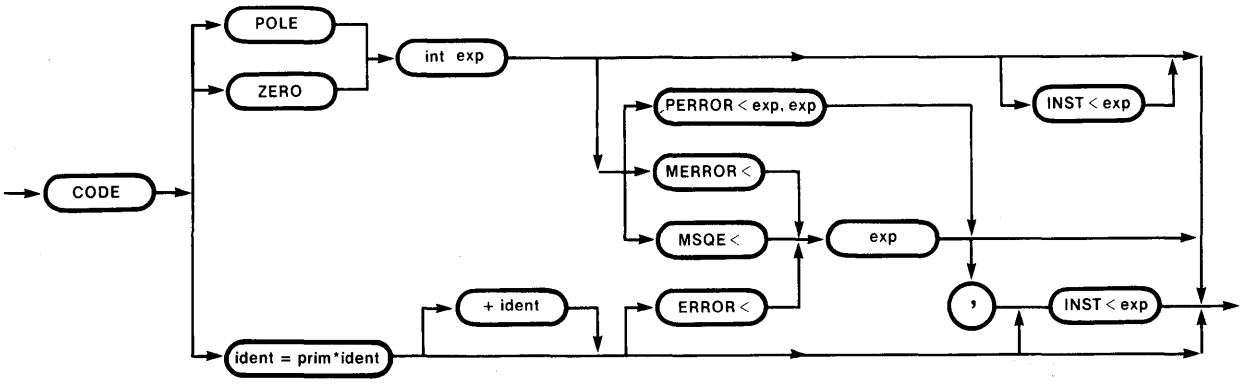
The code will automatically contain comments to identify the location and label of the pole or zero, or to specify the value of the multiplier in the case of an equation. Comments usually show the current contents of the destination operand in each instruction, in terms of the constant and variable names supplied in the CODE command. The AS2920 Assembler ignores these comments.

In general, the code generated will not implement the requested object exactly. Instead, the code is generated with respect to the constraints explained in this chapter: INST, ERROR, MSQE, MERROR, and PERROR. The keyword INST shows the number of instructions compiled. For an equation, the accuracy of the resultant compilation is reflected in the value of the keyword ERROR.

For a pole or zero, the accuracy of the code is reflected in the values of MSQE and MERROR, if referenced immediately thereafter. If there are intervening commands, MSQE and MERROR may be rewritten to reflect them. The imprecision of compilation for a pole or zero, i.e., the object's "movement," is explained below under PERROR. The "error", therefore, is also reflected in how far the actual object (as compiled) differs from its original position.

If the constraint is too severe and cannot be achieved in the number of 2920 instructions requested, then the Compiler selects that set of generated code which minimizes the constraint you gave, within that number of instructions. If the code object is a multiplication when this happens, ERROR is given the minimum error (signed) achievable in the number of instructions given by the value of INST. Interrupting the Compiler with an ESCape key while it is generating code causes the best yet code to be entered into the CODE buffer, and then halting generation.

The elements of the chart below are discussed in subsequent paragraphs.



Code Command 121533-16

Coding a Single Pole or Zero

As the chart above indicates, there are four constraints for use on coding poles or zeros. Each is explained below. The command begins with the word CODE, followed by the desired object and its label, leading to one of the four constraints, as follows:

```
MSQE < expression
```

or

```
MERROR < expression
```

or

```
PERROR < expression_1 , expression_2
```

Of these three, at most one may be chosen for any given CODE command. If none appears, the default PERROR<0,0 is used, resulting in minimum movement from the original coordinates of the pole or zero (see below). If one of these three constraints does appear, it may optionally be followed by a comma and the program size constraint, as follows,

```
, INST < expression
```

which restricts the number of instructions generated to fewer than the value of the expression given. The INST phrase may appear alone, i.e., without a comma and without any other constraint. If it is not supplied, a default limit of 20 is automatically used.

Using the constraint

```
MSQE < expression
```

means that the gain of the coded filter is to deviate from the bounds by less than the value of the expression given. Further, this must be achieved in fewer than 20 instructions (or the number you supply in the INST phrase).

Examples:

```
CODE POLE 1 MSQE < 0.2
CODE ZERO 9 MSQE < 0.02, INST < 6
```

Using the constraint

```
MERROR < expression
```

means that the maximum absolute error of the coded filter's gain is to be kept below the value of the expression given, also within the instruction limit supplied. ("Error" means deviation from the gain bounds you supplied in earlier commands.)

Examples:

```
CODE POLE 1 MERROR < 0.2
CODE ZERO 9 MERROR < 0.02, INST < 17
```

Using the constraint

```
PERROR < expression_1 , expression_2
```

means that there is a limit on the movement (explained below) of the coded pole or zero from the original position of the defined pole or zero. That is, the difference of their first coordinates must be less than expression_1, and the difference of their second coordinates must be less than expression_2. Once again, this must be achieved within the INST constraint, i.e., in fewer instructions than the limit supplied.

This constraint requires some further explanation. PERROR is needed because the assembly language program generated by the CODE command implements a filter stage corresponding to a pole or zero at a slightly different location than the specified original pole or zero. After the compilation has been performed, the pole or zero is moved to the location matching the code generated. (However, this move will never change a complex conjugate pole [or zero] pair to a single real pole [or zero], nor a real into a complex conjugate pair.)

PERROR constrains the amount of that movement, in each coordinate, in the home plane (TS or Z) of the pole or zero. For poles or zeros defined in TS, these coordinate increments will be <real,imaginary> in Hertz. For those in Z, the increments will be <radius, angle (in radians)>.

MSQE and MERROR will always contain values which reflect the actual position of currently existing poles and zeros, i.e., including this implicit move. PERROR cannot be displayed. The original position of the coded pole or zero is lost. Thus you may wish to save it using a PUT or APPEND command (described in Chapter 8) or by saving its coordinates in the symbol table, e.g.,

```
.ORIG_POLE_3_REAL = REAL (POLE 3)
.ORIG_POLE_3_IMAG = IMAG (POLE 3)
```

Examples:

```
CODE POLE 1 PERROR < 4.2, 2
CODE ZERO 9 PERROR < 4.02, 0.06, INST < 9
```

In any case if the MSQE, MERROR, or PERROR constraint cannot be met within the INST constraint, the Compiler selects that set of code which minimizes the specified constraint in the given number of instructions. In the case of PERROR, what is minimized is the variance from your specified constraints on coordinate changes, as follows:

Suppose, for POLE 1 at XORIG, YORIG, you give constraints XCON and YCON, as in

```
CODE POLE 1 PERROR < XCON, YCON
```

then call the actual pole position XTRY, YTRY (for each set of code attempted by the Compiler).

This represents a movement of XMOV, YMOV, i.e.,

```
|XTRY - XORIG| = XMOV
|YTRY - YORIG| = YMOV
```

Using these definitions, then, the Compiler selects that set of code which minimizes

```
(XMOV-XCON)**2 + (YMOV-YCON)**2
```

Minima and Error Constraints

None of the minimizations above are necessarily true minima. True minima would require trying every possible code sequence, because the constraints MSQE and MERROR, which depend on the bounds you supply, can in fact behave quite non-linearly. The SPAC20 Compiler's algorithms for selecting the approximate minimum work best when the error bounds are "reasonable". Therefore, it is required that before coding is begun, the MSQE or MERROR for the gain curve must already meet your intended constraint. Thus if the code generated corresponds exactly to the specified pole or zero, the MSQE or MERROR constraint will be satisfied.

Coding Equations

The second form of the CODE command generates AS2920 code for calculations of the form YY=C*XX or YY=C*YY or YY=C*XX+YY, where C is a constant and XX, YY are variable names. The INST constraint can be used as above. Code is produced which minimizes the error in the multiplier (C) as much as possible in the number of instructions specified (or in 20, the default).

The other constraint allowed (besides INST) is

```
ERROR < expression
```

(which may be followed by a comma and an INST constraint). This specifies that the error in approximating C must be less than the value of the given expression, within the number of instructions desired. After the coding is completed, the value of ERROR shows the absolute value of error in the multiplier. The variable names used to request this coding will appear in the generated code.

In the absence of an explicit ERROR constraint, the default is ERROR<(multiplier/2**16), i.e., create the least ERROR possible (out to 16 binary places) within the INST constraint. If the given error constraint cannot be met, ERROR is minimized.

Examples:

```
CODE YVAR = 1.58 *XVAR
CODE ZVAR = 0.692475 * AVAR INST < 6
CODE XVAR = 2.3975 * YVAR ERROR < 0.0025
CODE YVAR = 0.11825 * XVAR + YVAR ERROR<0.00125, INST < 5
```

Note

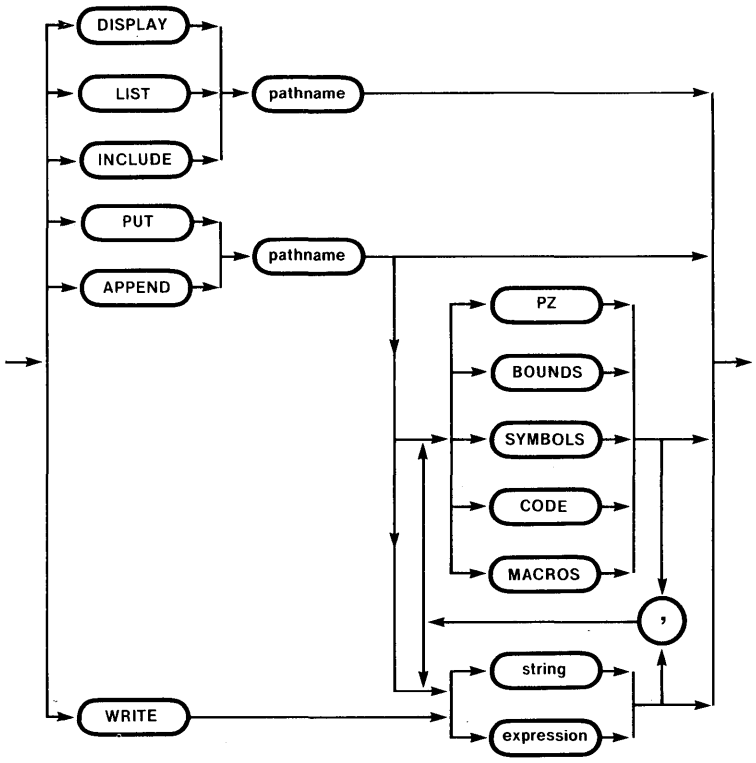
YY = YY + C*XX is not acceptable. The equation must be requested in the form YY = C*XX + YY

In general, it takes no more than one 2920 instruction per 2 significant bits in the specified constant. Thus 24-bit accuracy could be theoretically obtained in at most 12 instructions. However, the SPAC20 algorithms are not substantially effective beyond 16 bits, so that in most cases an INST constraint of less than 8 is sufficient. Greater accuracies can be obtained through techniques explained in Appendices H and J. Code generated for equations of the form YY=C*XX is the most efficient, achieving approximately 3 significant bits in the constant per instruction generated.

The code generated for a multiplication by a constant C will not overflow as long as the multiplicand is less than 1/C in absolute value. If the multiplicand is greater than 1/C in absolute value, an overflow will occur on the last instruction (and possibly earlier), yielding a result of ±1.

CHAPTER 8
FILE HANDLING

This chapter covers the commands EXIT, LIST, WRITE, DISPLAY, APPEND, PUT, and INCLUDE. Certain features of the operating system on INTELLEC computers are used by these commands.



File Commands 121533-24

Interface with ISIS-II

The Intel Systems Implementation Supervisor (ISIS-II) is the diskette operating system for the Intellec Microcomputer Development System. The Signal Processing Applications Compiler runs under ISIS-II control, and can call upon ISIS-II for file management functions. To execute the SPAC20 Compiler, you enter the characters SPAC20 (possibly preceded by a drivename, e.g., :F1:SPAC20) after an ISIS-II prompt character (hyphen "-" or angle bracket ">").

The Compiler signs on with a message

```
ISIS-II 2920 SIGNAL PROCESSING APPLICATIONS COMPILER, V1.0 - MATH BOARD VERSION
*
```

indicating by the asterisk prompt that it is ready to receive commands.

With the exception of EXIT and WRITE, the commands in this chapter are used to reference files or devices via ISIS-II pathnames. For diskette files, the format of pathname is as follows (a pathname may not contain blanks):

```
:drive:filename
              e.g., :F1:MYFILE,
                    :F0:YOFILE,
                        FILE79
```

The entry :drive: stands for one of the references to INTELLEC system diskette drives. F0 is assumed when drive is omitted. See the ISIS-II User's Guide for further detailed data.

The entry filename must follow the colon after drive without any intervening spaces. A filename has the following components:

```
identifier[.extension]
```

The above identifier is a name you assign, and is one to six alphanumeric characters. The extension is an optional part of the filename, consisting of one to three alphanumeric characters preceded by a single period. The extension must be used if it is present in the directory listing of the file on the diskette.

If used, the extension follows the identifier without any spaces. Some extensions (e.g., .BAK, .LST) are assigned by system processors; others can be assigned as you like. An extension provides a second level of file identification; it can be used to distinguish different versions of the same program, or to give supplemental information about the file (e.g., author, date, version).

For devices other than diskette files, the format of pathname is as follows:

```
:device:
```

The following devices are commonly accessed in SPAC20 Compiler commands:

:DEVICE:	OUTPUT DEVICE
:LP:	Line Printer
:HP:	High-speed tape punch
:TO:	Teletypewriter printer
:CO:	Console display
:CI:	Console Input
:HR:	Paper tape reader

For more information on ISIS-II filenames and device codes, refer to the ISIS-II User's Guide.

EXIT

The EXIT command keyword returns control from the SPAC20 Compiler to ISIS-II. It is as simple as it looks. Any files opened during the session are closed. After the line terminator has been entered following the command EXIT, access is no longer possible to any prior commands, parameters, macros, calculations, and graphs or lists which have not been saved into diskette files or onto hard-copy listings.

LIST

All output is normally sent to the console device (:CO:). The LIST command saves a duplicate record of the console input and output during a SPAC20 Compiler session, including high-volume data such as graphs or listings, on a hard-copy device or on a diskette file.

Only one LIST device or file other than the console can be specified (active) at a given time. Devices that can be specified, if present, are a line printer (:LP:), high-speed paper tape punch (:HP:) or a teletypewriter printer (:TO:).

Instead of a hard-copy device, a diskette file can be specified. If so specified, the file is opened when the LIST command is invoked. If a file of that name already exists, *its directory entry is deleted* and the name will thereafter refer to the list file being generated.

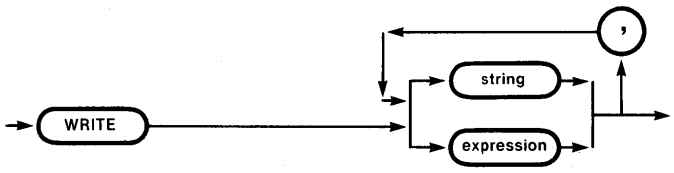
When LIST is in effect (with a device or file other than :CO:), all commands input (including comments) and all output from the SPAC20 Compiler (including system prompts, commands, graphs, and error messages) are sent both to the named device or file and to the console display.

To restore output to the console only (no other device), use the command LIST :CO:.

Examples:

```
LIST :LP:
LIST :CO:

LIST :F1:DESIGN.930
LIST :TO:
```

WRITE



Write Command 121533-45

The command WRITE puts out a single line of output to the LIST file and the console. It evaluates any expression you supply prior to output. In interactive sessions you would rarely use it, since the EVALUATE command provides the same function without the file output effect. WRITE is normally used within compound commands to provide periodic reporting on an automatic iterative process you have designed to test or generate special capabilities (see Chapters 9 through 11).

Examples:

```
WRITE 'Process now beginning step', .STEPNO, 'of section', .SECTNO
WRITE 'Stage number 3 of filter number 1, device MDM'
```

Display Command

The DISPLAY command copies the contents of the named file to the console. It enables you to examine the results of an earlier PUT or APPEND command without invoking execution of the results. The INCLUDE command, explained later in this chapter, does invoke execution.

Examples:

```
DISPLAY :F3:PARAM.FIL
DISPLAY CODFIL.POL
DIS MYMACR.OS1
DIS PUT3OK
```

Here, as in general, ESCape can be used to abort the command, terminating the display and returning you to the command level (asterisk prompt) of the Compiler.

One scenario for the use of this command is this: after saving all poles and zeros (using a put or append command as discussed below) and altering some of them, possibly via CODE commands, the display command permits a review of the earlier positions without disturbing the current conditions. The display can be interrupted with control-S (holding down the control key while pressing S), and restarted using control-Q. However, it is not possible to display a file that is currently open, e.g., a LIST file in use recording this session, or the current MAC.TMP (see Chapter 9).

APPEND Command

APPEND adds the specified (or default) objects described below to the end of the named file, if it exists. If it does not, the command creates it. Most file objects (other than CODE, strings, or expressions) cause SPAC20 commands to be output to the file. These commands will recreate the stated objects when invoked (executed) by a subsequent INCLUDE command. PZ, BOUNDS, SYMBOLS, and MACROS will cause the restoration, respectively, of all poles and zeros, bounds, user symbols, and macros.

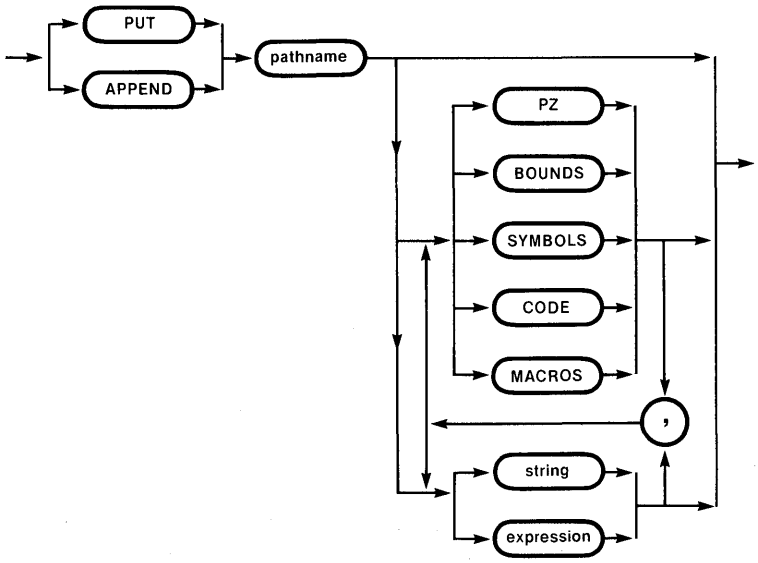
If no file object is supplied, commands are generated to permit restoring as much as possible of the state of this session. In particular, if you re-invoke the Compiler and include the file that you just PUT, your state will be restored exactly. This means, in addition to PZ, BOUNDS, SYMBOLS, and MACROS, commands to restore TS, XSIZE, YSIZE, GREF, HOLD, FSCALE, and YSCALE.

The file object CODE refers to the current contents of the code buffer, i.e., the results of the last CODE command. By APPENDing the results of successive such commands, the user can build up a file containing the assembly language code implementing the successive stages of a filter.

A list of expressions and strings can be used as a file object, resulting in the ASCII text of the strings and the decimal values of the evaluated expressions to be output on one line to the file. This can be used to insert comments in the growing assembly language file, or to insert assembly language code to perform scaling and propagation between filter stages.

Examples:

```
APPEND :F2:PZALL PZ
APPEND POLE9.COD CODE
APPEND NEW.MAC MACROS
APPEND :F1:PARAMS.ALL
APPEND FILTER.CUR PZ,SYM
APP LATEST.COD CODE,';above is for stage 3, filter 2, 11/15/79'
```



Put/Append Command 121533-46

PUT Command

PUT operates identically to APPEND with a single difference: if the file named in the command already exists, PUT overwrites it with the supplied (or default) file objects. (A message is sent to the console if such an overwrite occurs.) In general, APPEND should be used in all cases save where you are absolutely sure you will not regret destroying any possible earlier file of the same name. When in doubt, you can use the DISPLAY command to check if the file exists, and if so, what's in it.

Examples:

```
PUT :F2:PZALL PZ
PUT POLE9.COD CODE
PUT NEW.MAC MACROS
PUT :F1:PARAMS.ALL
PUT :F1:OLDFIL '';this overwrites and empties out OLDFIL
PUT FILT07.COD CODE ';above ',INST,' instructions implement
   stage ', .STAGE, ', filter 7'
;(assuming .STAGE was earlier defined to label the stage
   currently coded)
```

INCLUDE Command

This command enables you to restore some or all of the key parameters/states/tools from a prior interactive session for use during this one. When you issue the command

```
INCLUDE pathname
```

(where pathname is usually a diskette filename, e.g., CODFIL.922 or :F1:MACROS.921) the commands stored in that file are executed as if you had typed them directly from the console. Thus, if in your earlier session you had issued the command

```
APPEND NEWFIL.921 PZ
```

then when you INCLUDE NEWFIL.921 in this session, all poles and zeros defined at the time of the earlier command will be reestablished by DEFINE POLE commands for this session. This is true for any of the parameter-related file-objects for APPEND or PUT (i.e., INCLUDE should not be used for files of CODE, strings, or expressions). If the earlier PUT or APPEND had no specified file object, then all relevant parameters would have been saved. Your current INCLUDE command would then cause the restoration of all poles, zeros, bounds, scales, symbols, macros, sample rate, reference gain, hold state, and screen-size parameters to their earlier values, effectively restarting that session.

The INCLUDE command is particularly useful when building a library of macros (Chapter 9). Macros can be created but not edited interactively. If your macros contain more than two or three commands, you may wish to create the macros using an editor (e.g., CREDIT) in the form of an INCLUDE-able file.

The SPAC20 Signal Processing Applications Compiler can also be run under SUBMIT (see ISIS manual for detailed instructions). However, macros to be used under SUBMIT should be INCLUDEd rather than defined in-line in the submit-file. This preserves the distinction between the formal parameters of the submit command and the formal parameters for macro definition.

CHAPTER 9
ADVANCED (COMPOUND) COMMAND SYNTAX

This chapter discusses macros and the other compound commands: IF, REPEAT, and COUNT. Compound commands consist of sequences of simple commands to be executed in order as described below.

When commands are being input as part of compound commands, the normal asterisk prompt character is preceded by a period, to indicate the compound construction. If the compound command is itself embedded within another compound command, then the commands in its subordinate command block will have two periods before the prompt, and so on for deeper levels of nested compound commands.

Some typing or syntax errors cause only the current line to be rejected. This is indicated by the Compiler repeating the same sequence of prompt-characters that began the last line, e.g., "..*". However, more serious errors cause rejection of the entire compound command, forcing you to retype the command from the very beginning. This is indicated by a single asterisk (*).

Macros

A macro is a named block of commands, executed in sequence (or containing branches, if you so specify) when the macro name is typed as a command (invoked). The block of commands is also called the macro body.

The sequence is stored as you define it. This saves you repetitive entry of every command in the sequence, and also permits you to capture conditional logic (instruction branches or loops) only once, for potentially frequent use in future sessions. The macros you define are saved on a temporary file on diskette, but this file is not saved when you exit. If you use the commands

```
PUT filename MACROS
```

or

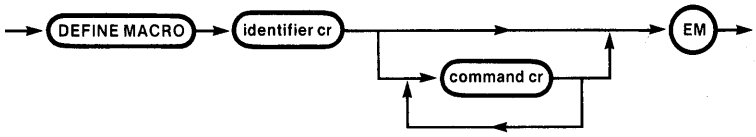
```
APPEND filename MACROS
```

prior to exiting, then the macros may be INCLUDEd for use during any future session.

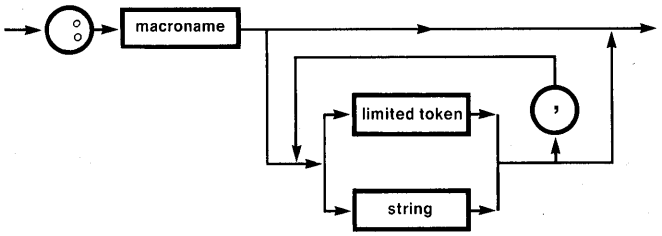
The macro commands described in this chapter allow you to perform the following functions:

- Define a macro, specifying the macro name, the command block, and any formal parameters (points in the macro definition where text can be replaced by actual parameters when the macro is invoked)
- Invoke (call) a macro by name, giving actual parameters to substitute for chosen formals (if any), beginning the execution of the defined command block
- Display the text of any macro as it was defined
- Display the names of all macros currently defined
- Remove one or more macros from those currently defined

Defining and Invoking Macros



Define Command for Macros 121533-19



Invoking Macros 121533-20

Each macro used in any design/test session must be defined during that session. Once defined, it may be invoked as often as desired, even within other macros.

The definition can occur either by typing it or by bringing it in from a file via the INCLUDE command (see Chapter 8). The macro name must be an identifier as described in Chapter 2, and must not duplicate any other macro name used in this session. (Thus you may not redefine a macro name, nor include it from a file, unless it is first removed from the macro table (directory) as described later in this chapter.) It may, however, duplicate symbol or keyword identifiers.

The DEFINE MACRO command causes the macro name (and the block of commands you supply) to be stored in a table of macro definitions in a temporary ISIS-II file named MAC.TMP (on the same disk drive containing SPAC20). Upon exit, they will not be retained. If you create new macros during this session, in addition to any you may have INCLUDEd, then to save them for future use you must save them as described above. Since macros cannot be edited within SPAC20, you may wish to use the Editor to create INCLUDE files for long or complex macros, making it easier to correct errors in typing or command constructs.

A macro definition (or removal) may not appear within any other command. This means you may not define a macro within another macro definition sequence, nor within any other compound command. Any other command may appear in *any* compound command.

When you attempt to invoke a macro, the macro name you supply must be already defined.

Here is a simple macro definition:

```
DEFINE MACRO GRAPHER
    REPEAT
        MOVE POLE 1 BY 0,5
        OGRAPH GAIN
    END
EM
```

To invoke this macro and cause its command block to begin executing, you enter the macro name preceded by a colon(:), as follows:

```
:GRAPHER
```

This will continue to move the pole by the indicated increment, and overgraph the new gain, until you hit ESCape.

A macro definition can include commands that define user symbols and other identifiers, such as poles or zeros, sample rates, etc. Macros that include such definitions can be used to set certain initial conditions for many of your interactive sessions. INCLUDE files can also be used for this.

A macro definition can include calls to other macros, but not to itself. If you inadvertently create one that tries to call itself, it will expand indefinitely when it is first invoked, without ever executing any later commands. (Press ESCape to terminate such an infinite expansion.)

Macro calls can be nested, i.e., one macro calls another, which in turn calls another, and so on. The level of nesting is limited only by the memory space required to contain the macro expansions and to stack the macro calls.

When a macro is invoked, the following operations occur:

- The text of each actual parameter in the call is substituted for the corresponding formal parameter in the definition
- The expanded command block is executed if all commands are valid as expanded
- The macro exits. Control returns to the console (asterisk prompt), or to the next command in sequence if the macro was invoked inside a compound command.

It is usually more efficacious to define several small macros rather than one large one incorporating all their features. They are eaiser to type in and more likely to fit in memory. The Chebyshev macro shown in Chapter 10 is a good example of a maximum-sized macro.

Formal and Actual Parameters

A formal parameter marks a place in a macro definition, where text will be replaced when the macro is invoked. A formal parameter can represent part of a token or a field of one or more tokens. When you invoke the macro, you supply the actual text which is to replace the formal parameter as the macro is expanded. A macro definition can contain up to ten formal parameters, each having the form:

```
%N
```

where N is a decimal digit, 0 through 9. For example, if you modify the macro GRAPHER above to read

```
DEFINE MACRO GRAPHER2222
    REPEAT
        MOVE POLE 1 by %0, %1
        OGRAPH GAIN
    END
EM
```

then each call to GRAPHER2222 can specify different increments for the MOVE, e.g.,

```
:GRAPHER2222 0,5        ;(then ESCape to stop execution)
:GRAPHER2222 0.5,12     ;(ESC to stop)
```

and so forth.

Formal parameters can appear in the body of the macro definition in any order, and each one can appear any number of times. This means that %3 can be used in a command before %1 is used, and either can appear often or not at all. The number implies the order in which the actual parameters will appear in the call, i.e., %0 means use the first actual parameter supplied, %1 means use the second, %4 means use the fifth, etc.

A string can be supplied as an actual parameter to a macro. In fact, if the parameter contains a quote mark, a carriage-return, or a comma, the parameter *must* be sent as a string, or errors will occur. (Of course, this means any embedded quote marks must be doubled to avoid looking like the end of the string.)

The quote marks surrounding the string in the macro call (invocation) are stripped off before the macro uses the string. If the command that uses this string, within the macro, requires the string to have quote marks around it, then either the macro definition must supply them or the string used in the call must have an extra set of quotes surrounding it. This will be shown in examples below.

If an actual parameter is omitted in some call, the comma which normally would follow that parameter must be typed anyway to retain the necessary positional order of supplied parameters. This naturally does not apply to the actual parameter corresponding to the last formal, which would have no comma after it. In fact, if the omitted parameters are all at the end of the list, no extra commas are required.

Omitted parameters result in the corresponding %N being replaced by the null string. If you supply, in the call, more actual parameters than there are defined formals in the definition, the extra actuals are ignored.

As an example, suppose you had defined this macro:

```
DEF MAC BATCH
    %0
    %1
    %2
    %3
EM
```

This would permit you to string out, on one line, up to four commands. You could type, for example,

```
:BATCH GRAPH GAIN, OGRAPH PHASE, GRAPH IMPULSE, OGRAPH STEP
```

If you supplied only 3 or 2 commands, the last formals would expand to the null character and this macro will exit normally.

However, if there are any actual parameters being supplied after an omitted actual, the extra comma mentioned above must be supplied. The examples below will illustrate this.

If a formal parameter does not appear in the macro's command block, then anything supplied in that position in the call will be ignored. For example, if your command block never referred to %2, then the third parameter in the call would always be ignored. Conversely, if the block does refer to %2 but the call does not supply a third parameter, the null (empty) string will be supplied. The command containing that reference to %2 must be a valid command even in the absence of an actual parameter, or the macro will abort when that command is encountered during expansion of the macro.

One example of such a possibility is the APPEND command:

```
DEF MAC SAVER
    APPEND %0 %1 %2
    APPEND %3 %4 %5
EM
```

Note that if you supply only %0 and %3, the filenames, the APPEND commands are still valid because no object is required—the default will be used.

Using the macro SAVER, you can now type a single line to establish (or add to) one or two files, old or new, using the APPEND command with any combination of its possible file-objects: PZ, BOUNDS, SYMBOLS, MACROS, CODE, strings, and/or expressions. One advantage to this hypothetical macro is being able to specify two separate files.

For example, one call to SAVER could add new macros to the accumulated set of macros, and in the same invocation put out the latest CODE to the growing file of coded filter-stages:

```
:SAVER MAC.NOW,MACROS,,CODE.NOW,CODE
```

Or, with parameter %4 being CODE, parameter %5 could create a comment line identifying the filter stage or other data pertinent to this code block:

```
:SAVER MAC,MACROS,,COD,'CODE,',''''STAGE 4 of FILTER 2'''
:SAVER MAC,MACROS,,COD,CODE,',''STAGE 4 of FILTER 2'''
```

The outer apostrophes around 'CODE,' are required due to the embedded comma, which in turn is needed to separate the objects of the second APPEND. (Note also that in order to supply the APPEND with a string in quotes, it is necessary in the macro-call to surround the quoted string with another pair of quotes. An actual macro parameter given with quotes has the outermost pair, i.e., the first quote and the last, stripped off during the process of being substituted for the formal parameter in the macro body. Also, quotes appearing within a string must be doubled.)

Possibly you might wish to have one file for poles and zeros and bounds (PZ, BOUNDS) and one for SYMBOLS, MACROS:

```
:SAVER PZBOUN, 'PZ,', BOUND, MACSYM, MACROS, ', SYMBOLS'
         0       1      2       3       4        5
```

Here each actual parameter is flagged with the formal it replaces. Again, the embedded commas are needed for valid APPEND commands, and so are put inside quotes, to be sent as part of a string. The other commas simply separate the actual-parameters in the macro-call.

More Examples:

```
:SAVER PZ.9,PZ,,COD.9,''';this was for stage 4 of filter 5.'''
```

The above macro call appends to PZ.9 all commands necessary to duplicate the current pole/zero configuration. These commands will be executed when an INCLUDE PZ.9 command is input. The comment '; this was for stage 4 of filter 5' is appended to the file COD.9 by this same SAVER invocation. The two commas after PZ reflect the absence of %2.

If we define a new macro

```
DEFINE MACRO SAVCOD
    APPEND %0 COD, '; this was for stage', %1, 'of filter', %2
EM
```

then the following call would add the current contents of the code buffer, followed by the same comment used in the last example:

```
:SAVCOD COD.9, 4,5

:SAVCOD FILE1, 7 ;This performs just as the example
; above but omits the filter number from the comment, which
; becomes '' this was for stage 7 of filter''

:SAVCOD FILE1,,7; These both operate similarly to the examples
; above, but the first omits the stage number
:SAVCOD FILE1 ; and the second omits both stage and filter
; numbers.
```

Thus the comment arising from the first of the pair above will be

" ; this was for stage of filter 7"

and the comment from the last command above will be

" ; this was for stage of filter"

The comment embedded in the APPEND command (in the macro body) is used with no identifying numbers.

The last two calls to this macro differ in omitting parameter %1 or %2. The commas delimiting the parameters must be typed even when a parameter will be omitted, if there are additional desired or required parameters coming after the omitted ones. When no such parameters are required or wanted, as in the last case, the extra commas are not needed.

```
:SAVER FILE1,PZ,,FILE2,CODE ;This saves poles and zeros in one
; file and code in another with no comment.
```

When you invoke the SAVER macro, you must supply the first and fourth parameter, %0 and %3, or the APPEND commands will have no file to append to, and this will cause the invocation to abort. If you suppy neither %1 nor %2, the default file-objects for the APPEND command will be assumed, which means all the

objects denoted by the five keywords will be saved as commands added to the end of the file whose name you supplied as %0. The command for this could have been simply APPEND filename. Using SAVER, you have the option of also filing the code, e.g.,

```
:SAVER PARAMS.ALL,,,CODFIL,CODE
```

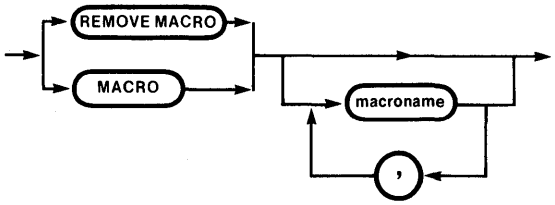
Macro Expansion and Syntax Checking

The syntax and semantics of the commands in a macro block are ignored at the point of definition; they are not determined until invocation, and may be different on each invocation through the use of formal parameters.

When a macro is called, its definition is expanded by replacing the formal parameters in the definition, using the text of the actual parameters in the call. If the expanded macro contains any calls to other macros, the text of any such macro is also expanded, forming in effect one overall block of commands. The results of expansion are displayed at the console. Expansion continues until the last EM is reached. If the expansion results in a set of complete, valid commands, the commands are executed. An error results if any command is incomplete or invalid after expansion. Examples of macro expansion and syntax checking appear in Chapters 10 and 11.

Displaying or Removing Macros

Macro Directory



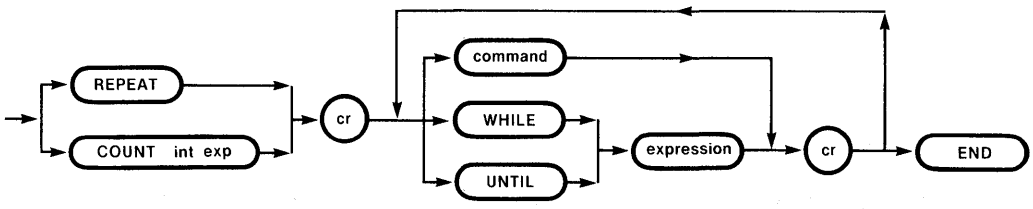
macroname::= an identifier appearing as above in a legal define-macro command

Remove or Dispaly Macros 121533-21

The DIR MACRO command lists the names (but not the bodies) of all macros currently defined. Macro bodies may be displayed by typing a list of one or more macro-names after the keyword MACRO, followed by the usual carriage-return. Similarly, macros may be removed by typing a list of one or more macro-names after the keywords REMOVE MACROS. If no list follows MACRO or REMOVE MACRO, then *all* current macros are displayed or removed, respectively.

Examples of these commands appear in Chapters 10 and 11.

Controlling a Loop: REPEAT, COUNT, WHILE, and UNTIL



REPEAT, COUNT, WHILE, and UNTIL 121533-23

These compound commands permit the blocks of contained commands to be executed indefinitely, a specific number of times, or conditionally. REPEAT or COUNT commands may contain any number of conditional exits using the keyword phrases WHILE condition-expression or UNTIL condition-expression. The paragraphs that follow give explanations and examples of the use of these commands and modifying phrases.

As the charts above indicate, these loop control commands begin with the word REPEAT or the phrase COUNT integer-expression, meaning any expression which evaluates to a positive integer quantity. This quantity is evaluated immediately and used as the number of iterations (maximum) for the commands contained in the command block that follows. REPEAT has no such limit. (You can use the ESCape key to abort command processing, returning you to the Compiler.)

The command block may have in it any number of any commands except DEFINE MACRO or REMOVE MACRO. The Compiler's awareness that you are entering commands within a compound command is shown by the period it types at the beginning of each such line. If a new compound command is begun as part of defining a prior compound command, a second period (and third, etc., as needed) is typed by the Compiler to indicate the nesting. Each REPEAT or COUNT ends with an END statement.

A macro invoked in a REPEAT or COUNT command is expanded immediately after the macro call command is entered. Thus, a macro called in a REPEAT or COUNT command is expanded only once, though perhaps used repeatedly thereafter.

If the block of commands within a compound command is to continue executing only under certain conditions, you can use the WHILE or UNTIL clause to specify them. This can involve a wider class of expressions than the arithmetic ones explained in Chapter 2. The full range of legal expressions is discussed in the next section, after which the discussion of compound commands is continued.

Relational and Boolean Expressions

Relational expressions involve a comparison of the values of two objects, using these relational operators:

- < Less than
- = Equal to
- > Greater than
- <= Less or equal
- <> Not equal
- >= Greater or equal

Relational expressions are evaluated to a FALSE or TRUE value, meaning the least significant bit of the values is 0 or 1, respectively.

Examples:

```
GAIN(60) < .GAIN_LAST_POLE
ANGLE(POLE 1) > ANGLE(POLE 4)
.FIRST_VALUE <= .NEXT_VALUE
```

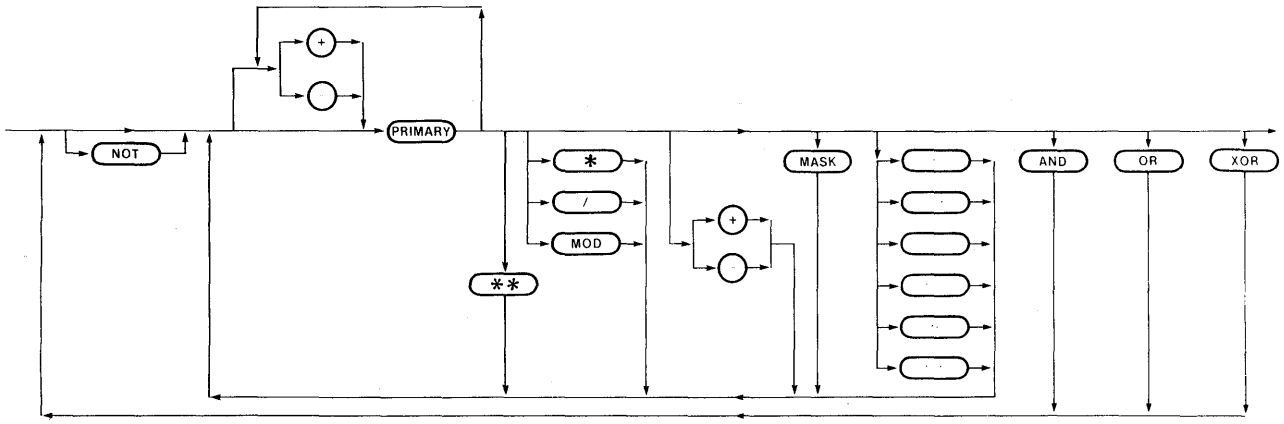
Boolean expressions represent combinations of TRUE and FALSE values using conjunction (AND), disjunction (OR), negation (NOT), and exclusive-disjunction (XOR). As an example,

```
X<PI AND NOT X<0
```

is either true or false for any given value of X. The operands for the boolean operators can be any integer or relational expression.

Examples:

```
AGAIN (250) < .AGAIN_LAST AND MAGAIN < 300
INST < 9 OR MSQE < 147
```



Expression 121533-26

WHILE and UNTIL (Continued)

After the WHILE or UNTIL you place a boolean or relational-expression, which is evaluated to a FALSE or TRUE value. (Actually, any integer-valued expression is legal. FALSE means the lowest-order bit was a 0, TRUE, a 1.)

The WHILE clause terminates execution of the loop upon a value of FALSE; the UNTIL clause does so upon a value of TRUE. The commands in a block continue to be executed until one of these clauses causes a halt or until the count limit is reached. Execution then continues after the END for that block.

In both the WHILE and UNTIL clauses, the relational-expression is evaluated each time the clause is encountered, i.e., once per iteration. Evaluation at each iteration involves looking up the values of any references in the expression. Thus, the result can change with each evaluation.

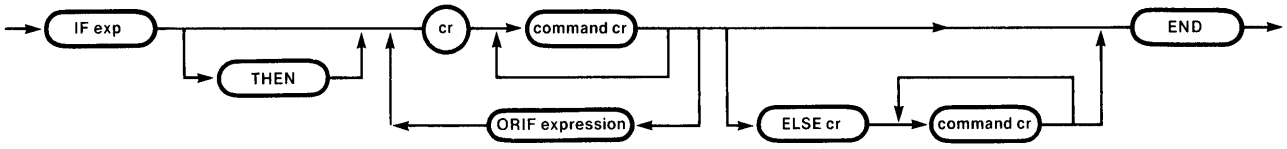
This is different from the expression that follows an exterior COUNT, which is evaluated once, the first time it is encountered. (A COUNT embedded within a WHILE or UNTIL clause could use an expression dependent on varying variable references. Its value would nevertheless be fixed for the block of commands under its control.)

The use of WHILE or UNTIL is usually a matter of style or preference, since there is always a way to convert one into the other: WHILE expression_1 is equivalent to UNTIL NOT expression_1. If the expression_1 used in this "escape" clause is false, the loop is exited as soon as this is evaluted. If the clause comes at the end of the command block it affects, the prior commands in the block will be done once even when the expression is false. If the clause comes first, no commands in the block are done.

Examples:

```
REPEAT
   GRAPH GAIN
   MOVE PZ BY 0.005,0.005
END


COUNT 5
   CODE POLE .NEXT PERROR < 1,1
   APPEND FILCOD.819 CODE
   APPEND FILCOD.819 '; THE ABOVE CODE WAS FOR POLE ', .NEXT
   .NEXT = .NEXT + 1
END


(.NEXT is assumed to be initialized prior to the above commands.)


REPEAT
   GRAPH GAIN
UNTIL GAIN(655) <= 80
   MOVE PZ BY 5,5
END


COUNT 5
   WHILE .NEXT < 16
   CODE POLE .NEXT PERROR < 1,1
   APPEND FILCOD.819 CODE
   APPEND FILCOD.819 '; THE ABOVE CODE WAS FOR POLE ', .NEXT
   .NEXT = .NEXT + 1
END
```

The IF Command



The IF Command 121533-22

This compound command permits you to specify blocks of commands whose execution is contingent upon tests of certain values or relationships. It is a powerful capability, making it possible to specify in advance the consequences you wish to invoke under varying circumstances, e.g., error variances.

This command must have the IF clause and the final END. The word THEN is optional. There may be any number of ORIF clauses, or none, and one ELSE clause, or none. The expression following IF is evaluated to FALSE or TRUE. If it is TRUE, the command block immediately following is executed, and subsequent ORIF or ELSE expressions are ignored along with their associated command blocks.

If the IF expression evaluates to FALSE, the next sequential ORIF or ELSE expression is evaluated, with similar consequences: if this expression evaluates TRUE, only its command block is executed and all other parts of the IF command are skipped. A macro called in an IF command is expanded whether the condition in the IF or ORIF clause that contains that call is TRUE or FALSE. Here is a simple example of an IF command:

```
.VAR8 = 1.4142

.LIMT = 1

IF .VAR8 < .LIMT THEN

     EVALUATE .VAR8/PI ; PI = 3.1415926

ORIF .VAR8 < 2

     EVALUATE .VAR8/HPI ; HPI = 1.707963

ORIF .VAR8 < 3

     EVALUATE .VAR8/TPI ; TPI = 6.2831852

ELSE

     EVALUATE .VAR8/(PI * PI)

END
```

This example would display the result of EVALUATE .VAR8/HPI and then terminate. The first condition is FALSE so the first potential consequence is skipped. The second condition is true, so the second command block is executed and the IF command terminates. The third condition is not tested, so its associated command block is not executed despite the condition being true.

Another Example:

```
*DEFINE MACRO COMPARE
.*   IF %0 < %1 THEN
.*      WRITE 'LESS'
.*   ORIF %0 = %1 THEN
.*      WRITE 'EQUAL'
.*   ELSE
.*      WRITE 'GREATER'
.*   END
.*EM
```

This macro can later be invoked with any two actual parameters, e.g.,

```
* :COMPARE .VAR8, .LIMT
.*  IF .VAR8 < .LIMT THEN
.*     WRITE 'LESS'
.*  ORIF .VAR8 = .LIMT THEN
.*     WRITE 'EQUAL'
.*  ELSE
.*     WRITE 'GREATER'
GREATER
* :COMPARE .LIMT, .VAR8
.*  IF .LIMT < .VAR8 THEN
.*     WRITE 'LESS'
.*  ORIF .LIMT = .VAR8 THEN
.*     WRITE 'EQUAL'
.*  ELSE
.*     WRITE 'GREATER'
LESS
* :COMPARE PI, TPI/2
.*  IF PI < TPI/2 THEN
.*     WRITE 'LESS'
.*  ORIF PI = TPI/2 THEN
.*     WRITE 'EQUAL'
.*  ELSE
.*      WRITE 'GREATER'
EQUAL
```

This example displays the expansion of the macro each time, and the result of comparing the two actual parameters. The IF/ORIF/ELSE blocks cause execution of only one block, so that the output of COMPARE can only be 'LESS', 'EQUAL', or 'GREATER'.

Nesting Compound Commands

The REPEAT, COUNT, and IF commands can be nested to provide a variety of control structures.

Each nested compound command must have its own END. When entering a nested command sequence, you may wish to use the keywords ENDR, ENDC, and ENDIF to help you keep straight which command you intend to close off at that point. Nesting levels are not checked when the command is being entered, and if an END is omitted, the resulting error makes it necessary to enter the entire command again. Further, even if the correct number of ENDs is supplied, their position in the command sequence is critical to achieving your intended flow of control.

When entering a compound command, some syntax errors allow recovery to the state at the last prompt. You can recognize such recovery by the ...* prompt, indicating you are still within the definition phase of entry. Other syntax errors are fatal, requiring you to retype the entire command. This you can recognize by the * prompt.

Each nested REPEAT or COUNT command can contain its own exit clauses (WHILE or UNTIL). Each such exit clause can terminate the loop that contains it, but has no effect on any outer loops or commands.

Examples of nesting appear in Chapters 10 and 11.

CHAPTER 10
ADVANCED TECHNIQUES: FILTER DESIGN EXAMPLES

Introduction

As Chapter 9 has illustrated, the macro capability is a powerful tool, enabling you to define command sequences under a single name and then use that name as a new command. There are few restrictions on these sequences, allowing very general routines to be created. Parameters you may wish to vary from one use to the next can be built into the sequence as formals, to be supplied at the time of use, i.e., macro invocation. This permits tailored sequences to be produced from the general pattern you developed.

As with any general-purpose computer feature, the ingenuity of user-developed applications for this macro capability cannot be fully defined or predicted. It greatly extends the range of the signal processing Compiler language.

Examples of useful macros directly relevant to filter design comprise the main body of this chapter. More general examples, applicable to a wider range of signal processing requirements, appear in Chapter 11. All of these examples can be replicated for other filters or other operations once you see the process for defining and generating them. Their use as models is one of the main motivations for supplying them, in addition to their intrinsic utility. The ones you want for immediate use shoulde be edited from the supplied file SPAC20.MAC into a separate file to conserve space when they are later INCLUDEd.

The first two macros given in this chapter produce Butterworth and Chebyshev filters based on user-supplied parameters for cut-off frequency, ripple, etc. The Bilinear transform macro performs S to Z transformation. The last macro produces code for the current state of an entire filter: each pole and zero, plus A to D and D to A conversion.

These are Intel-supplied macros, and appear in the file SPAC20.MAC. They have been checked for correct functionality, i.e., that given the right input in the correct order, they will produce the described output.

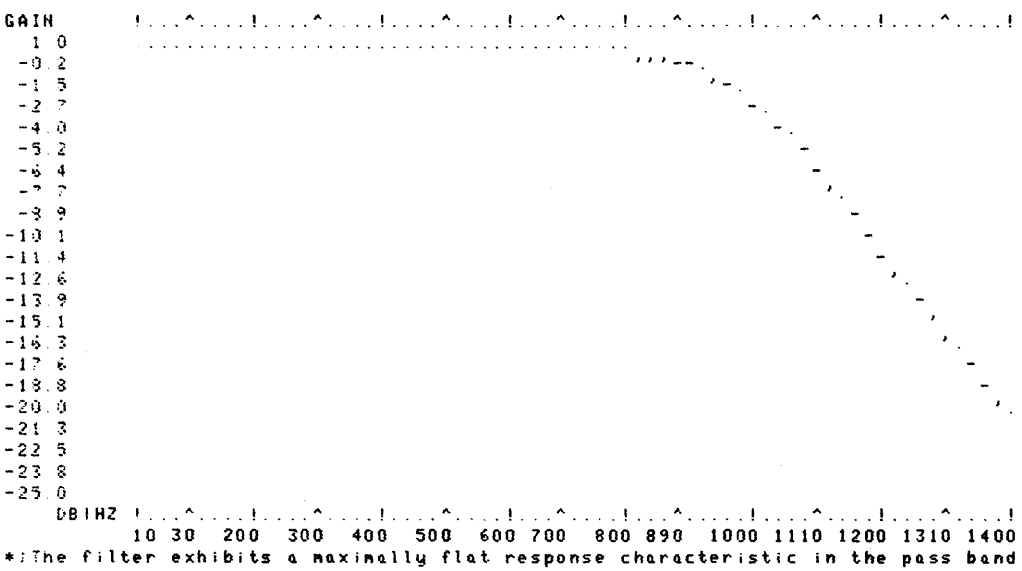
They are, however, macros as opposed to commands. As such, the degree of error checking is not (and cannot be) as extensive as for the basic set of built-in commands. If the parameters supplied are not of the correct type or not in the proper order, the results cannot be fully predicted or guaranteed; it is conceivable that prior work could be altered or erased.

It is therefore extremely important, before using any macro, to understand its expectations, as reflected in the nature and order of the parameters to be supplied.

```
*:***** BUTTERWORTH FILTER MACRO *****
*INC :F1:BUTTER.MAC
*
*DEFINE MACRO BUTTER ;************************************************
.*; This is a BUTTERWORTH FILTER GENERATOR for SPAC20;                ***
.*;
.*; Calling sequence :BUTTER ORDER, Fco, LABEL where                  ***
.*;    :BUTTER calls the MACRO,                                       ***
.*;    ORDER is the order of the filter                               ***
.*;    Fco is the cut-off frequency in Hz                             ***
.*;    LABEL is starting point for POLE numbering.                    ***
.*;
.*;       EXAMPLE     :BUTTER 6,500,234                               ***
.*;                    this will generate a BUTTERWORTH filter        ***
.*;                    of order 6, cutoff=500 Hz , producing          ***
.*;                    3 complex poles labeled 234,235,236           ***
.*;
.* DEFINE .?BUTSTART = ( HPI ) + ( HPI/%0 )     ;** BEGIN THE          ***
.* DEFINE .?BUTDELTA = ( PI/%0 )                ;** BUTTERWORTH;       ***
.* DEFINE  ?BUTINDEX = 0                        ;**   INITIALIZE       ***
.* DEFINE .?BUTANGLE = 0                        ;**   VARIABLES;       ***
.* REPEAT                                       ;** BEGIN LOOP;        ***
.*    .?BUTINDEX = .?BUTINDEX + 1               ;**  CORRECT FOR       ***
.*    .?BUTANGLE = .?BUTSTART - .?BUTINDEX * .?BUTDELTA ;* SMALL       ***
.*    IF .?BUTANGLE < .?BUTDELTA/4 THEN         ;** ANGLE ERROR;      ***
.*          .?BUTANGLE=0                        ;**                    ***
.*    END                                       ;** NOW CREATE         ***
.*    DEF POL(.?BUTINDEX+%2-1) = -%1*COS(.?BUTANGLE),&    THE          ***
.**                    %1*SIN(.?BUTANGLE)        ;**  NEXT POLE;       ***
.*    WHILE .?BUTINDEX + 1 <= ( %0 + 1 ) / 2     ;**  CONTINUE ?        ***
.* END                                          ;** END OF LOOP;       ***
.* REM .?BUTANGLE                               ;**  REMOVE ALL        ***
.* REM .?BUTINDEX                               ;**  VARIABLES         ***
.* REM .?BUTDELTA                               ;** OF THIS MACRO;     ***
.* REM  ?BUTSTART                               ;**DISPLAY POLES;      ***
.*P2                                           ;**END BUTTERWORTH     ***
.*EM  ;************************************************************
*
*;!    This macro generates a Butterworth filter with the cutoff frequency
*;and the order specified. Poles and zeroes are placed in the S plane in a
*;circle centered around the origin.  The following example illustrates how
*;it is invoked
*;.
*:BUTTER 7,1000,0;creates a 7th order filter ,cutoff=1000 hz
```

(The expansion has been deleted.)

```
POLE 0 = -222.52087,974.92791,CONTINUOUS
POLE 1 = -623.48980,781.83148,CONTINUOUS
POLE 2 = -900.96893,433.88366,CONTINUOUS
POLE 3 = -1000.00000,0.00000000,CONTINUOUS; REAL
*;;the macro expansion has been suppressed here
*;the macro outputs the PZ( poles and zeroes) it has generated
*;Let's look at the response
*FSC=10,200,400,600,800,1000,1200,1400     ;set new frequency range
*YSC=-25,1                                 ;set new vertical scale
*GRA GAIN
```

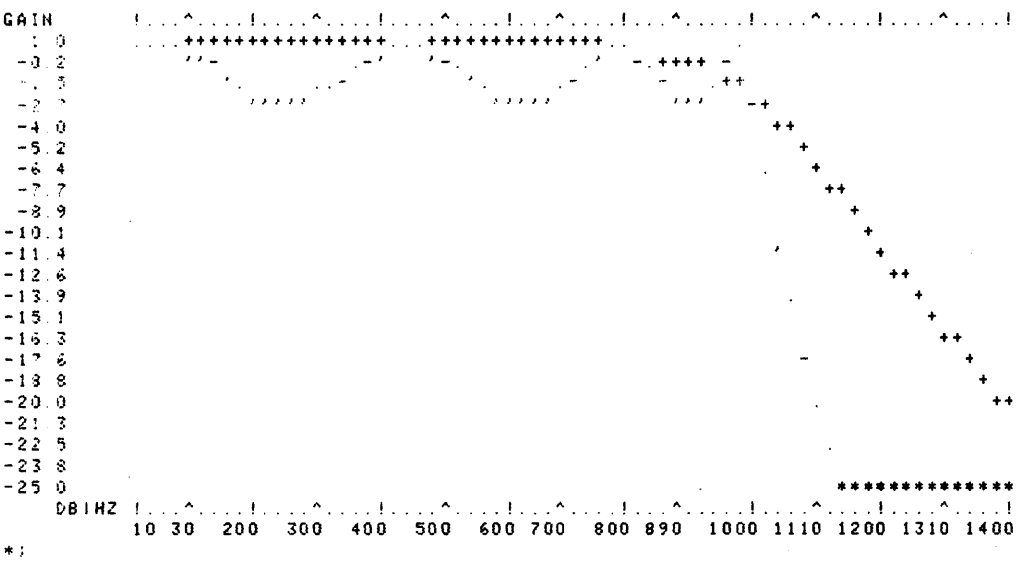



```
*;The filter exhibits a maximally flat response characteristic in the pass band
*;as expected
*
*
*
*
*
*
```

```
*
*
*
*
*
*
*rem macro
*REM PZ
4 POLES/ZEROES REMOVED
*;***** CHEBYSHEV FILTER MACRO *****
.*;      A CHEBYSHEV FILTER GENERATOR FOR SPAC20                          **
.*;
.*; CALLING SEQUENCE  :CHEB ORDER, Fco, LABEL, R.F.                       **
.*; WHERE ORDER is the order of the filter                                **
.*;       Fco is the cutoff frequency in Hz                              **
.*;       LABEL is the starting point for pole numbering                 **
.*;       R.F. is the desired (or allowable) ripple factor in dB.        **
.*
.*;        EXAMPLE   :CHEB 6,500,23,0.12
.*;                    this will generate a CHEBYSHEV filter of order   **
.*;                    6, cutoff=500,and a peak-to-peak ripple of 0.12  **
.*;                    dB, producing 3 complex poles labeled 23,24,25   **
.*;
.*DEF .?CHEBYRIP=10**(ABS(%3)/10)-1                    ;* BEGIN THE      **
.*DEF .?SINHP=1                                        ;* CHEBYSHEV      **
.*DEF .?COSHP=1                                        ;* BY SETTING     **
.* IF .?CHEBYRIP <> 0 THEN                             ;* DEFAULT VALUES, **
.*    :TEMCHB .?SINHP,.?COSHP,.?CHEBYRIP,%0            ;** OR USE THE     **
.* END                                                 ;** SUB-MACRO      **
.*REM .?CHEBYRIP                                       ;** TO GENERATE    **
.* DEFINE .?BUTSTART = ( HPI ) + ( HPI/%0 )           ;** THE VARIABLES. **
.* DEFINE .?BUTDELTA = ( PI/%0 )                      ;**                **
.* DEFINE .?BUTINDEX = 0                              ;* A MODIFIED      **
.* DEFINE .?BUTANGLE=0                                ;* BUTTERWORTH     **
.* REPEAT                                             ;* MODULE IS       **
.*    .?BUTINDEX = .?BUTINDEX + 1                     ;* INCORPORATED    **
.*    .?BUTANGLE=.?BUTSTART - .?BUTINDEX*.?BUTDELTA   ;* TO GENERATE     **
.*      IF .?BUTANGLE < .?BUTDELTA/4 THEN             ;* THE APPROPRIATE **
.*           .?BUTANGLE=0                             ;* PATTERN OF      **
.*      END                                           ;* POLES FOR THE   **
.*   DEF POL(.?BUTINDEX+%2-1)=              &         ;* FILTER. (the &s **
.**          -%1*.?SINHP*COS(.?BUTANGLE),     &       ;* allow greater   **
.**                %1*.?COSHP*SIN(.?BUTANGLE)         ;* readability of  **
.*   WHILE .?BUTINDEX + 1 <= ( %0 + 1 ) / 2           ;* the formula.)   **
.* END
.* REM .?BUTINDEX                                     ;** REMOVE THE     **
.* REM .?BUTDELTA                                     ;** VARIABLES      **
.* REM .?BUTSTART                                     ;** INTRODUCED     **
.* REM .?BUTANGLE                                     ;** IN THIS MACRO, **
.* REM .?SINHP                                        ;** TO SAVE SPACE. **
.* REM .?COSHP                                        ;**               **
.*PZ
.*EM                              ;******* END OF CHEBYSHEV MACRO ********
*
*;  TEMCHB     GET VARIABLES FOR CHEBYSHEV FILTER ######################
*DEF MAC TEMCHB
.*; CALLING SEQUENCE                                 ## THIS IS THE    ###
.*;      :TEMCHB   SINHP,COSHP,E**2,N                ## SUB-MACRO.     ###
.*DEF .?INVSXTEMP=(1/SQR(%2))+(SQR((1/%2)+1))
.*DEF .?INVSXTEMPP=.?INVSXTEMP **(1/%3)                              ;###
.*DEF .?INVSXTEMPN=.?INVSXTEMP **(-1/%3)
.*%0=(.?INVSXTEMPP - .?INVSXTEMPN)/2                                 ;###
.*%1=(.?INVSXTEMPP + .?INVSXTEMPN)/2
.*REM .?INVSXTEMP                                                    ;###
.*REM .?INVSXTEMPP
.*REM .?INVSXTEMPN                                  ;## END OF TEMCHB ###
.*EM
*
*
*;This macro generates a Chebyshev filter with the order,cutoff frequency
*;and peak-to-peak ripple as specified.  The following is an example of
*;how the macro may be invoked:
*;CHEB 7,1000,0,3 ;7th order 1000 Hz filter, p-p ripple=3
```

(The expansion has been deleted.)

```
POLE 0 = -28.145643,982.69561,CONTINUOUS
POLE 1 = -78.862358,788.06072,CONTINUOUS
POLE 2 = -113.959419,437.34060,CONTINUOUS
POLE 3 = -126.485404,0.00000000,CONTINUOUS; REAL
*;The macro expansion has been suppressed
*;
*;Let's look at the response
*OGR GAIN
```

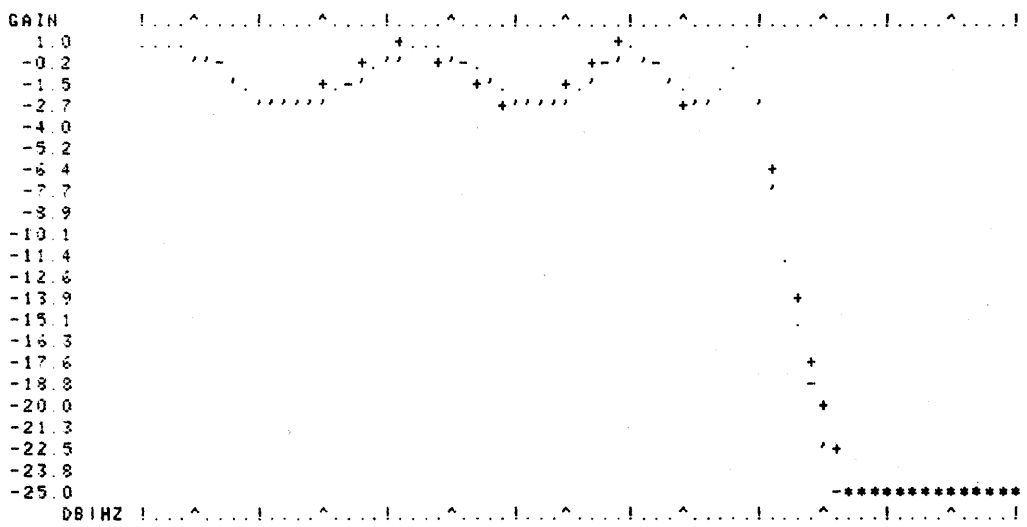



```
*;
*;The plus signs represent the old gain curve of the Butterworth filter.
*;The peak-to-peak ripple is 3dB as specified and the dropoff is faster than
*;the Butterworth  filter
*
*
*
*
*
*
```

```
*
*
*
*;***** BILINEAR TRANSFORM MACRO
*INC :F1:BITRAN.MAC
*
*
*DEF MACRO BTP
.*;This macro generates a Bilinear transform of a given pole
.*         ;Calling sequence    :BTP  POLE # IN S, POLE # IN Z
.*         ;EXAMPLE             :BTP  3,90
.*         ;                    this will take a pole in the S plane,(P 3)
.*         ;                    and produce a pole in the Z plane (P 90)
.*         ;                    and 1 or 2 extra zeroes (Z 90 / Z 91)
.*         ;                    depending on whether P 3 is a real pole or complex
.*;BLTCOM %0, %1, POLE, ZERO
.*EM
*
*
*
*DEF MACRO BTZ
.*;This macro generates a bilinear transform of a given zero
.*         ;Calling sequence       :BTZ  ZERO # IN S, ZERO # IN Z
.*         ;EXAMPLE                :BTZ  3,90
.*         ;                       this will take a zero in the S plane, (Z 3)
.*         ;                       and produce a zero in the Z plane (Z 90)
.*         ;                       and 1 or 2 extra poles (P 90 / P 91)
.*         ;                       depending on whether Z 3 is real or complex
.*;BLTCOM %0, %1, ZERO, POLE
.*EM
*
*
*
*DEF MACRO CKREAL
.*; SUB MACRO used in another sub-macro BLTCOM to check for real pole/zero and adjust
.*;            angle etc
.*IF ABS(COS(TPI*IMAG(%2 %0)))=1 THEN
.*       REM %3 %1+1          ;REMOVE EXTRA PZ
.*       %5=SQR(%5)           ;ADJUST SCALE FACTOR
.*       IF REAL(%2 %0)(%4 THEN    ;ADJUST ANGLE
.*MOV   %2 %1 TO RAD(%2 %1),0
.*       ELSE
.*MOV   %2 %1 TO RAD(%2 %1),PI
.*       END
.*END
.*EM
*;
*;
*
;
*DEF MACRO BLTCOM
.*; This macro is called from BTP/BTZ, which supply pole/zero
.*; labels ,and the proper 'pole'/'zero'
.*
.*; THE TRANSFORM EQUATION HERE IS
.*;
.*             S   = (C) * (1-Z**-1) / (1 + Z**-1)
.*;
.*DEF .?QC = 2/TS;This is the constant C used in this macro
.*DEF .?QA = - REAL (%2 %0)
.*DEF .?QB = IMAG (%2 %0)
.*DEF .?QA0 = (.?QC + .?QA)**2 + .?QB **2
.*DEF .?QA1 = 2*(.?QA**2 + .?QB**2 - .?QC**2)
.*DEF .?QA2 = (.?QC-.?QA)**2 + .?QB**2
.*DEF .?QRADIUS = SQR (.?QA2/.?QA0)
.*DEF .?QANGLE = ACOS (-.?QA1/ (.?QA0*2*.?QRADIUS))
.*DEF %2 %1 = .?QRADIUS, .?QANGLE, Z
.*DEF %3 %1 = 1, PI, Z
.*DEF %3 (%1+1) = 1, PI, Z
.*;
.*ADJUST FOR REAL IF NECESSARY
.*;CKREAL %0,%1,%2,%3,.?QC,.?QA0
.*
.*WRITE 'SCALE FACTOR = ', 1/.?QA0
```

```
.*
.*REM .?QC
.*REM .?QA
.*REM .?QB
.*REM .?QA0
.*REM .?QA1
.*REM .?QA2
.*REM .?QRADIUS
.*REM .?QANGLE
.*EM
*
*;This macro performs a Bilinear transform and takes a pole/zero in
*;the S plane and generates a pole/zero in the Z plane. Additional poles
*;or zeroes may be added to the point 1,PI in the Z plane.  The user
*;will have to cancel poles and zeroes that lie on top of each other (at the
*;1,PI point on the Z plane )manually.
*;      Let us try to generate a 1000 Hz Chebyshev filter in the Z plane
*;First, we have to warp the frequency , and obtain an analog filter
*;then we can proceed to design a digital filter by a Bilinear transform.
*;Let's design a digital Chebyshev 7th order filter with a cutoff frequency
*;of 1000 hz and 3dB ripple
*;First we obtain a prewarped analog filter
*REM PZ
4 POLES/ZEROES REMOVED
*TS=1/8000
TS = 1.24999975/10**4
*:CHEB 7,2/TS*TAN(PI*TS*1000),0,3

(The expansion has been deleted.)

POLE 0 = -186.53292,6512.7338,CONTINUOUS
POLE 1 = -522.65380,5222.8076,CONTINUOUS
POLE 2 = -755.25665,2898.4389,CONTINUOUS
POLE 3 = -838.27160,0.00000000,CONTINUOUS; REAL
*;INVOKE THE BILINEAR TRANSFORM MACRO
*:BTP 0,1000

(The expansion has been deleted.)

SCALE FACTOR = 3.3302805/10**9
*:BTP 2,1200

(The expansion has been deleted.)

SCALE FACTOR = 3.2849423/10**9
*:BTP 1,1100

(The expansion has been deleted.)

SCALE FACTOR = 3.3302805/10**9
*:BTP 2,1200

(The expansion has been deleted.)

SCALE FACTOR = 3.4585420/10**9
*:BTP 3,1300

(The expansion has been deleted.)
```

```
1 POLES/ZEROES REMOVED
1 POLES/ZEROES MOVED
SCALE FACTOR = 5.9388589/10**5
*;
*PZ
POLE 0 = -186.53292,6512.7338,CONTINUOUS
POLE 1 = -522.65380,5222.8076,CONTINUOUS
POLE 2 = -755.25665,2898.4389,CONTINUOUS
POLE 3 = -838.27160,0.00000000,CONTINUOUS; REAL
POLE 1000 = 0.98019560,0.77321420,Z
POLE 1100 = 0.94265810,0.63161171,Z
POLE 1200 = 0.91259267,0.35917749,Z
POLE 1300 = 0.90043251,0.00000000,Z; REAL
ZERO 1000 = 1.00000000,3.1415927,Z; REAL
ZERO 1001 = 1.00000000,3.1415927,Z; REAL
ZERO 1100 = 1.00000000,3.1415927,Z; REAL
ZERO 1101 = 1.00000000,3.1415927,Z; REAL
ZERO 1200 = 1.00000000,3.1415927,Z; REAL
ZERO 1201 = 1.00000000,3.1415927,Z; REAL
ZERO 1300 = 1.00000000,3.1415927,Z; REAL
*;the poles labeled 0 to 3 have been transformed to those PZ labeled 1000-1302
*;Let's compare with the analog Chebyshev filter
*;
*;Let's compare the gain with the analog Chebyshev filter
*REM P 0 THR 3
4 POLES/ZEROES REMOVED
*OGR GAIN
```




```
*;
*;The plus sign represents the points where the old curve differs
*
*
*
```

```
*;
*;
*;***** MACRO TO CODE AN ALL-POLE FILTER
*
*INC :F1:CODFIL.MAC
*;
*DEFINE MACRO CODFIL
.*;
.*;MACRO to code an all-pole filter, with all poles
.*;       labeled consecutively starting at 0.
.*;       Scale factors will be approximated from MAGAIN.
.*;       The comment generated assumed that the PZ are in the TS plane,
.*;       and therefore output the PZ position in cartesian coordinates
.*;
.*;WARNING :  THERE IS A  ( REM PZ ) COMMAND WHICH WILL REMOVE ALL YOUR
.*;            PZ FROM THIS SESSION. THE PZ AFTER CODING WILL HAVE BEEN SAVED
.*;            IN A NEWLY CREATED FILE TEMPXX.TMP
.*;
.*;
.*;CALLING SEQUENCE
;;CODFIL    FILENAME, NO. OF POLES, CONSTRAINT
.*;
.*;EX. :CODFIL  LOPAS.SRC, 10, INST<10
.*;                  this will code all poles labeled 0 through 9, and all code
.*;                  generated will be appended to the file LOPAS.SRC. Each pole
.*;                  is coded with the constraint INST<10. In addition, scaling
.*;                  factors required between stages are also approximated from
.*;                  MAGAIN , and documented in the output file.
.*;
.*APP %0 ';
.**;START OF CODE GENERATED BY MACRO CODFIL
.**;
.**;'
.*;FHEAD %0
.*DEF .?QI=0
.*;
.*;
.*COUNT %1;START LOOP FOR CODE GENERATION
.*APPEND %0 ';'
.*APPEND %0 ';CODE FOR POLE ',.?QI,' AT ',REAL(P .?QI),', ',IMAG(P .?QI)
.*CODE POLE .?QI %2
.*APPEND %0 CODE
.*APPEND %0 '; ABOVE CODE MOVED POLE ',.?QI,' TO ',REAL(P .?QI),' , ',IMAG(P .?QI)
.*APPEND %0 ';'
.*.?QI=.?QI + 1
.*END;END OF LOOP FOR CODE GENERATION
.*;
.*;
.*APPEND %0 ';
.**;BE SURE TO DO THE FOLLOWING SCALING FOR EACH STAGE
.**;PLUS ANY OTHER SCALING COMMENTED IN THE GENERATED CODE
.**;'
.*PUT TEMPXX.TMP PZ;SAVE ALL PZ
.*WRITE 'PZ SAVED IN TEMPORARY FILE'
.*;
.*;QSFAC %0,%1                    ;GET SCALE FACTORS
.*;
.*APP %0 ' ;END OF CODE GENERATED BY MACRO CODFIL'
.*REM .?QI
.*;
.*WRITE' THIS MACRO HAS REMOVED ALL PZ.
.**       THE PZ ARE SAVED IN THE FILE TEMPXX.QQQ.
.**       TO RESTORE ALL PZ FROM THAT STAGE, TYPE
.**
.**               INC TEMPXX.TMP
.**
.**'
.*EM
```

```
*;
*;
*DEFINE MACRO QSFAC
.*;       SUB-MACRO TO GET SCALE FACTOR
.*;
.*;       WARNING : IF USED INDEPENDENTLY, THIS WILL REMOVE ALL YOUR PZ
.*;
.*;       CALLING SEQUENCE     :QSFAC  FILE-NAME, # OF POLES
.*DEF .?QSI=1
.*DEF .?QS1=LOG(MAGAIN)/LOG(2);START APPROXIMATING SCALE FACTOR
.*DEF .?QFS1=0
.*DEF .?QFS2=0
.*DEF .?QS2=0
.*DEF .?QSO=0
.*;
.*;
.*COUNT %1;GET SCALE FACTOR
.*REM POLE %1-.?QSI
.* .?QFS1=.?QS1 MASK 0.FFFFFFH
.*IF .?QFS1 <> 0 THEN;ROUND UP
.*      .?QS1= .?QS1 - .?QFS1 + 1
.*END
.* .?QS2=LOG(MAGAIN)/LOG(2)
.* .?QFS2=.?QS2 MASK 0.FFFFFFH
.*IF .?QFS2 <> 0 THEN;ROUND UP
.*      .?QS2= .?QS2 - .?QFS2 + 1
.*END
.* ?QSO=.?QS1-.?QS2
.*APPEND %O ' ;SHIFT RIGHT INPUT OF POLE # ',%1-.?QSI,' BY ',.?QSO,'  BITS
.*.?QS1=.?QS2
.* .?QSI=.?QSI + 1
.*END
.*;
.*;
.*REM .?QSI
.*REM .?QS1
.*REM .?QS2
.*REM .?QSO
.*REM .?QFS1
.*REM .?QFS2
.*EM
*
*INC :F1:FHEAD.MAC
*DEFINE MACRO FHEAD
.*;MACRO TO GIVE EQU HEADINGS TO FILE
.*;This macro only generates propagation for 11 stages,
.*;for more stages,add in the necessary APPEND'S using the Editor
.*;
.*
.*;CALLING SEQUENCE
.*;         :FHEAD FILENAME
.*APP %O ' IN0_P1   EQU OUT0_P0'
.*APP %O ' IN0_P2   EQU OUT0_P1'
.*APP %O ' IN0_P3   EQU OUT0_P2'
.*APP %O ' IN0_P4   EQU OUT0_P3'
.*APP %O ' IN0_P5   EQU OUT0_P4'
.*APP %O ' IN0_P6   EQU OUT0_P5'
.*APP %O ' IN0_P7   EQU OUT0_P6'
.*APP %O ' IN0_P8   EQU OUT0_P7'
.*APP %O ' IN0_P9   EQU OUT0_P8'
.*APP %O ' IN0_P10 EQU OUT0_P9'
.*EM
*
*;
*;Code all the poles of a filter
*;First define the poles of this filter
*TS=1/8000
TS = 1.24999975/10**4
*DEFINE POLE 0=-100,1000,TS
*DEFINE POLE 1=-700,1000,TS
*;ADCONV :F1:FIL.SRC,IN2,IN0_P0
```

(The expansion has been deleted.)

(This macro is explained in Chapter 11.)

```
FILE CREATED
*:CODFIL :F1:FIL.SRC,2,INST<10

(The expansion has been deleted.)

B1=1.30739128  B2=-0.85463593

INST=4
 POLE 0 = 0.00000000,1999.9998,TS
  BEST YET
 POLE 0 = -441.27120,999.99993,TS
  BEST YET

INST=5
 POLE 0 = 0.00000000,1333.33325,TS
 POLE 0 = -441.27120,999.99993,TS
 POLE 0 = 0.00000000,1140.39587,TS
  BEST YET
 POLE 0 = -85.008674,1999.9998,TS
 POLE 0 = -85.008674,1999.9998,TS
 POLE 0 = -299.21356,1128.18835,TS

INST=6
 POLE 0 = 0.00000000,920.21374,TS
  BEST YET
 POLE 0 = 0.00000000,1140.39587,TS
 POLE 0 = 0.00000000,1088.56665,TS
 POLE 0 = -85.008674,1281.96582,TS
 POLE 0 = -85.008674,1281.96582,TS
 POLE 0 = -299.21356,1128.18835,TS
 POLE 0 = -85.008674,1068.34033,TS
  BEST YET
 POLE 0 = -96.479698,1999.9998,TS
 POLE 0 = -96.479698,1999.9998,TS
 POLE 0 = -238.53721,1175.85119,TS

INST=7
 POLE 0 = 0.00000000,978.85827,TS
 POLE 0 = 0.00000000,1088.56665,TS
 POLE 0 = 0.00000000,1092.88586,TS
 POLE 0 = -85.008674,1068.34033,TS
 POLE 0 = -85.008674,1068.34033,TS
 POLE 0 = -85.008674,1068.34033,TS
 POLE 0 = -85.008674,1009.94763,TS
  BEST YET
 POLE 0 = -96.479698,1274.66479,TS
 POLE 0 = -96.479698,1274.66479,TS
 POLE 0 = -238.53721,1175.85119,TS
 POLE 0 = -132.187332,1024.49133,TS
 POLE 0 = -100.027595,1999.9998,TS
 POLE 0 = -99.379959,1999.9998,TS
 POLE 0 = -245.81303,1170.32641,TS

INST=8
 POLE 0 = 0.00000000,993.08331,TS
 POLE 0 = 0.00000000,1092.88586,TS
 POLE 0 = 0.00000000,1092.88586,TS
 POLE 0 = -85.008674,1009.94763,TS
 POLE 0 = -85.008674,1009.94763,TS
 POLE 0 = -85.008674,1009.94763,TS
 POLE 0 = -85.008674,1014.83471,TS
 POLE 0 = -96.479698,1057.95434,TS
 POLE 0 = -96.479698,1057.95434,TS
 POLE 0 = -132.187332,1024.49133,TS
 POLE 0 = -96.479698,998.55297,TS
  BEST YET
 POLE 0 = -100.027595,1272.38793,TS
 POLE 0 = -99.379959,1272.80419,TS
 POLE 0 = -245.81303,1170.32641,TS
 POLE 0 = -129.134002,1027.42260,TS
 POLE 0 = -100.004852,1999.9998,TS
 POLE 0 = -100.107147,1999.9998,TS
 POLE 0 = -247.64495,1168.92749,TS
```

```
INST=9
 POLE 0 = 0.00000000,1000.13574,TS
 POLE 0 = 0.00000000,1092.88586,TS
 POLE 0 = -85.008674,1002.46459,TS
 POLE 0 = -85.008674,1002.46459,TS
 POLE 0 = -85.008674,1014.83471,TS
 POLE 0 = -85.008674,1014.83471,TS
 POLE 0 = -96.479698,998.55297,TS
 POLE 0 = -96.479698,998.55297,TS
 POLE 0 = -96.479698,998.55297,TS
 POLE 0 = -96.479698,1003.52844,TS
 POLE 0 = -100.027595,1054.70727,TS
 POLE 0 = -99.379959,1055.30126,TS
 POLE 0 = -129.134002,1027.42260,TS
 POLE 0 = -97.203498,997.82714,TS
  BEST YET
 POLE 0 = -100.004852,1272.40246,TS
 POLE 0 = -100.107147,1272.33679,TS
 POLE 0 = -247.64495,1168.92749,TS
 POLE 0 = -127.612823,1028.87805,TS
 POLE 0 = -100.008880,1999.9998,TS
 POLE 0 = -100.016181,1999.9998,TS
 POLE 0 = -248.10377,1168.57666,TS

INST=9
 POLE 0 = -97.203498,997.82714,TS
  BEST
PERROR = -2.7965011, 2.1728515

; NOTE: MAKE SURE SIGNAL IS <0.76190478
LDA OUT2_P0,OUT1_P0,R00
     ; OUT2_P0=1.00000000*OUT1_P0
LDA OUT1_P0,OUT0_P0,R00
     ; OUT1_P0=1.00000000*OUT0_P0
ADD OUT0_P0,OUT1_P0,R02
     ; OUT0_P0=1.00000000*OUT0_P0+0.25000000*OUT1_P0
ADD OUT0_P0,OUT1_P0,R04
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0
ADD OUT0_P0,OUT2_P0,R03
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.125000000*OUT2_P0
ADD OUT0_P0,OUT2_P0,R06
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.140625000*OUT2_P0
ADD OUT0_P0,OUT2_P0,R10
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.141601562*OUT2_P0
SUB OUT0_P0,OUT2_P0,R00
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0-0.85839843*OUT2_P0
ADD OUT0_P0,IN0_P0,R00
     ; OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0-0.85839843*OUT2_P0+1.00000000*IN0_P0
B1=0.81611074  B2=-0.33301838

INST=4
 POLE 1 = -882.54235,1999.9998,TS
  BEST YET
 POLE 1 = -441.27120,999.99993,TS
  BEST YET

INST=5
 POLE 1 = -882.54235,1333.33325,TS
 POLE 1 = -441.27120,999.99993,TS
 POLE 1 = -882.54235,643.44500,TS
 POLE 1 = -624.41516,1999.9998,TS
 POLE 1 = -740.48474,1999.9998,TS
 POLE 1 = -526.27990,908.74194,TS
  BEST YET

INST=6
 POLE 1 = -882.54235,920.21374,TS
 POLE 1 = -882.54235,643.44500,TS
 POLE 1 = -882.54235,792.42419,TS
 POLE 1 = -624.41516,783.65319,TS
 POLE 1 = -740.48474,590.33453,TS
 POLE 1 = -526.27990,908.74194,TS
 POLE 1 = -624.41516,986.73602,TS
  BEST YET
 POLE 1 = -709.42401,1999.9998,TS
 POLE 1 = -709.42401,1999.9998,TS
 POLE 1 = -549.43231,881.42303,TS
```

```
INST=7
 POLE 1 = -882.54235,978.85827,TS
 POLE 1 = -882.54235,792.42419,TS
 POLE 1 = -882.54235,784.39038,TS
 POLE 1 = -624.41516,986.73602,TS
 POLE 1 = -740.48474,1063.76843,TS
  BEST YET
 POLE 1 = -624.41516,986.73602,TS
 POLE 1 = -740.48474,964.17120,TS
  BEST YET
 POLE 1 = -709.42401,649.02075,TS
 POLE 1 = -709.42401,649.02075,TS
 POLE 1 = -549.43231,881.42303,TS
 POLE 1 = -651.50946,958.45196,TS
 POLE 1 = -700.02026,1999.9998,TS
 POLE 1 = -701.88989,1999.9998,TS
 POLE 1 = -555.35443,874.23944,TS

INST=8
 POLE 1 = -882.54235,993.08331,TS
 POLE 1 = -882.54235,783.85223,TS
 POLE 1 = -882.54235,784.53430,TS
 POLE 1 = -624.41516,998.29034,TS
 POLE 1 = -740.48474,1014.92211,TS
  BEST YET
 POLE 1 = -740.48474,964.17120,TS
 POLE 1 = -740.48474,958.08459,TS
 POLE 1 = -709.42401,1091.25781,TS
 POLE 1 = -709.42401,1091.25781,TS
 POLE 1 = -651.50946,958.45196,TS
 POLE 1 = -709.42401,996.21044,TS
  BEST YET
 POLE 1 = -700.02026,665.58825,TS
 POLE 1 = -701.88989,662.33392,TS
 POLE 1 = -555.35443,874.23944,TS
 POLE 1 = -653.24188,956.59979,TS
 POLE 1 = -699.94952,1999.9998,TS
 POLE 1 = -700.02026,1999.9998,TS
 POLE 1 = -556.84350,872.42010,TS

INST=9
 POLE 1 = -882.54235,1000.13574,TS
 POLE 1 = -882.54235,784.53314,TS
 POLE 1 = -882.54235,784.53375,TS
 POLE 1 = -624.41516,999.72717,TS
 POLE 1 = -740.48474,1002.42456,TS
 POLE 1 = -740.48474,957.67767,TS
 POLE 1 = -740.48474,958.19348,TS
 POLE 1 = -709.42401,996.21044,TS
 POLE 1 = -709.42401,996.21044,TS
 POLE 1 = -709.42401,996.21044,TS
 POLE 1 = -709.42401,990.42437,TS
 POLE 1 = -700.02026,904.42840,TS
 POLE 1 = -701.88989,902.25109,TS
 POLE 1 = -653.24188,956.59979,TS
 POLE 1 = -701.88989,1003.74481,TS
  BEST YET
 POLE 1 = -699.94952,665.71081,TS
 POLE 1 = -700.02026,665.58825,TS
 POLE 1 = -556.84350,872.42010,TS
 POLE 1 = -654.10968,955.66998,TS
 POLE 1 = -700.02026,1999.9998,TS
 POLE 1 = -557.58941,871.50683,TS
```

```
INST=9
 POLE 1 = -701.88989,1003.74481,TS
  BEST
PERROR = 1.8898925, -3.7448120

LDA OUT2_P1,OUT1_P1,R00
     : OUT2_P1=1.00000000*OUT1_P1
LDA OUT1_P1,OUT0_P1,R00
     : OUT1_P1=1.00000000*OUT0_P1
SUB OUT0_P1,OUT1_P1,R03
     : OUT0_P1=1.00000000*OUT0_P1-0.125000000*OUT1_P1
SUB OUT0_P1,OUT1_P1,R04
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1
SUB OUT0_P1,OUT2_P1,R02
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.25000000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R04
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.31250000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R06
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.32812500*OUT2_P1
SUB OUT0_P1,OUT2_P1,R08
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.33203125*OUT2_P1
ADD OUT0_P1,IN0_P1,R00
     : OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.33203125*OUT2_P1+1.0000000
FILE DELETED
PZ SAVED IN TEMPORARY FILE
1 POLES/ZEROES REMOVED
1 POLES/ZEROES REMOVED
 THIS MACRO HAS REMOVED ALL PZ.
       THE PZ ARE SAVED IN THE FILE TEMPXX.QQQ.
       TO RESTORE ALL PZ FROM THAT STAGE, TYPE

              INC TEMPXX.TMP

*
*EXIT
```

Following is the contents of the file created by the CODFIL macro.

```
;A/D CONVERSION ROUTINE ADDED BY MACRO ADCONV
IN2
IN2
IN2
NOP
NOP
CVT5
ADD DAR,KM2,R00,CND6
NOP
NOP
CVT?
NOP
NOP
CVT6
NOP
NOP
CVT5
NOP
NOP
CVT4
NOP
NOP
CVT3
NOP
NOP
CVT2
NOP
NOP
CVT1
NOP
NOP
CVT0
NOP
NOP
LDA IN0_P0,DAR ;SCALE INPUT HERE
;END OF MACRO ADCONV
```

```
;
;START OF CODE GENERATED BY MACRO CODFIL
;
;
 IN0_P1  EQU OUT0_P0
 IN0_P2  EQU OUT0_P1
 IN0_P3  EQU OUT0_P2
 IN0_P4  EQU OUT0_P3
 IN0_P5  EQU OUT0_P4
 IN0_P6  EQU OUT0_P5
 IN0_P7  EQU OUT0_P6
 IN0_P8  EQU OUT0_P7
 IN0_P9  EQU OUT0_P8
 IN0_P10 EQU OUT0_P9
;
;CODE FOR POLE 0.00000000 AT -100.000000, 1000.00000
; NOTE: MAKE SURE SIGNAL IS <0.76190478
LDA OUT2_P0,OUT1_P0,R00
      OUT2_P0=1.00000000*OUT1_P0
LDA OUT1_P0,OUT0_P0,R00
      OUT1_P0=1.00000000*OUT0_P0
ADD OUT0_P0,OUT1_P0,R02
      OUT0_P0=1.00000000*OUT0_P0+0.25000000*OUT1_P0
ADD OUT0_P0,OUT1_P0,R04
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0
ADD OUT0_P0,OUT2_P0,R03
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.125000000*OUT2_P0
ADD OUT0_P0,OUT2_P0,R06
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.140625000*OUT2_P0
ADD OUT0_P0,OUT2_P0,R10
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0+0.141601562*OUT2_P0
SUB OUT0_P0,OUT2_P0,R00
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0-0.85839843*OUT2_P0
ADD OUT0_P0,IN0_P0,R00
      OUT0_P0=1.00000000*OUT0_P0+0.31250000*OUT1_P0-0.85839843*OUT2_P0+1.00000000*IN0_P0
; ABOVE CODE MOVED POLE 0.00000000 TO -97.203498 , 997.82714
;
;
;CODE FOR POLE 1.00000000 AT -700.00000, 1000.00000
LDA OUT2_P1,OUT1_P1,R00
      OUT2_P1=1.00000000*OUT1_P1
LDA OUT1_P1,OUT0_P1,R00
      OUT1_P1=1.00000000*OUT0_P1
SUB OUT0_P1,OUT1_P1,R03
      OUT0_P1=1.00000000*OUT0_P1-0.125000000*OUT1_P1
SUB OUT0_P1,OUT1_P1,R04
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1
SUB OUT0_P1,OUT2_P1,R02
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.25000000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R04
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.31250000*OUT2_P1
SUB OUT0_P1,OUT2_P1,R06
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.32812500*OUT2_P1
SUB OUT0_P1,OUT2_P1,R08
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.33203125*OUT2_P1
ADD OUT0_P1,IN0_P1,R00
      OUT0_P1=1.00000000*OUT0_P1-0.18750000*OUT1_P1-0.33203125*OUT2_P1+1.00000000*IN0_P1
; ABOVE CODE MOVED POLE 1.00000000 TO -701.88989 , 1003.74481
;
;
;BE SURE TO DO THE FOLLOWING SCALING FOR EACH STAGE
;PLUS ANY OTHER SCALING COMMENTED IN THE GENERATED CODE
;
 ;SHIFT RIGHT INPUT OF POLE # 1.00000000 BY 1.00000000  BITS
 ;SHIFT RIGHT INPUT OF POLE # 0.00000000 BY 4.0000000  BITS
 ;END OF CODE GENERATED BY MACRO CODFIL
```

CHAPTER 11
ADVANCED TECHNIQUES: OTHER ROUTINES FOR SIGNAL PROCESSING

Introduction

The macros given as examples in this chapter go beyond filter design into more general routines such as multiplication and division of variables, input/output coding, and oscillators. They too are supplied in the file SPAC20.MAC. As mentioned in Chapter 10, the intention in supplying these macros (in addition to their immediate utility) includes their role as models for your own development of macros, routines pertinent to signal processing and other functions important to your 2920-based product development. As with all macros, particular attention must be paid to supplying correct parameters in the appropriate order, to avoid erroneous operation or output.

```
*DEFINE MACRO MULVAR
.*;
.*;  This macro generates code for
.*;  a four quadrant, 9-bit multiplication of two variables.
.*; Calling sequence :MULVAR  OUTFILE,PRODUCT,MULTIPLICAND,MULTIPLIER.
.*'
.*                    EXAMPLE :MULVAR  F.MUL,W,X,Y
.*;                            this will put in the file F.MUL the code to
.*                             implement the equation W=X*Y
.*;
.*  APPEND %0  '                                  ; BEGIN MACRO MULVAR *****'
.*  APPEND %0  ' SUB   %1,   %1,  R00              ; Clear the product     *'
.*  APPEND %0  ' LDA   DAR,  %3,  R00              ; Multiplier to DAR     *'
.*  APPEND %0  ' ADD   %1,   %2,  R01,  CND7       ;     *     This        *'
.*  APPEND %0  ' ADD   %1,   %2,  R02,  CND6       ;     *    is the       *'
.*  APPEND %0  ' ADD   %1,   %2,  R03,  CND5       ;     *    multiply     *'
.*  APPEND %0  ' ADD   %1,   %2,  R04,  CND4       ;     *    process,     *'
.*  APPEND %0  ' ADD   %1,   %2,  R05,  CND3       ;     *    bitwise      *'
.*  APPEND %0  ' ADD   %1,   %2,  R06,  CND2       ;     *   adding and    *'
.*  APPEND %0  ' ADD   %1,   %2,  R07,  CND1       ;     *    shifting     *'
.*  APPEND %0  ' ADD   %1,   %2,  R08,  CND0       ;     *                 *'
.*  APPEND %0  ' SUB   %2,   %2,  L01              ;  These two supply     *'
.*  APPEND %0  ' ADD   %1,   %2,  R00,  CNDS       ;  the correct sign.    *'
.*  APPEND %0  '                                  ; END MACRO MULVAR *******'
.*
.*;  This performs a 9-bit multiply. If more bits of multiplier
.*; precision are required, the high order bits of the multiplier
.*; may be masked off, and the remaining bits shifted left and
.*; loaded to the DAR. The masking operation is necessary to
.*; prevent overflow saturation. ( See AS2920 manual for more info )
.*;  The last two steps above can be eliminated if the multiplier
.*; is known to be positive. The first step must be eliminated
.*; if the operation is to be of the form:   Y = W*X + Y
.*
.*EM
```

```
*DEFINE MACRO DIV
.*;
.*;          This macro generates code for a four quadrant division
.*;                      of 2 variables
.*; Calling sequence :DIV  FILE, DIVIDEND, DIVIDEND$SCALE, DIVISOR, QUOTIENT.
.*;                  EXAMPLE   :DIV  F.DIV,X,R02,Y,W
.*;                             this will put in the file F.DIV code to implement
.*;                             the equation   W = ( X * 2**-2 )/ (Y)
.*;
.*  APPEND %0 '                                ; BEGIN MACRO DIV *************'
.*  APPEND %0  ' ABS   DV1,  %1,  %2            ;  These two extract the     *'
.*  APPEND %0  ' ABS   DV2,  %3,  R00           ;       magnitudes.          *'
.*  APPEND %0  ' SUB   DAR, DAR,  R00           ;  This clears the DAR.      *'
.*
.*  APPEND %0  ' SUB   DV1, DV2,  R00,  CNDS    ;          This is           *'
.*  APPEND %0  ' SUB   DV1, DV2,  R01,  CND7'   ;              ♦
.*  APPEND %0  ' SUB   DV1, DV2,  R02,  CND6    ;        the divide,         *'
.*  APPEND %0  ' SUB   DV1, DV2,  R03,  CND5'   ;
.*  APPEND %0  ' SUB   DV1, DV2,  R04,  CND4    ;        progressing         *'
.*  APPEND %0  ' SUB   DV1, DV2,  R05,  CND3'   ;
.*  APPEND %0  ' SUB   DV1, DV2,  R06,  CND2    ;          one bit           *'
.*  APPEND %0  ' SUB   DV1, DV2,  R07,  CND1'   ;
.*  APPEND %0  ' SUB   DV1, DV2,  R08,  CND0    ;         at a time.         *'
.*
.*  APPEND %0  ' ADD   DAR, KP4,  L01           ;This forces overflow (#) *'
.*  APPEND %0  ' LDA    %4,  %1,  R13           ;    These two establish   *'
.*  APPEND %0  ' XOR    %4,  %3,  R13           ;     the correct sign.    *'
.*  APPEND %0  ' XOR    %4,  DAR               ;   Xfer result to output.  *'
.*  APPEND %0  '                               ; END MACRO DIV **************'
.*;
.*;
.*;  Note that the first two operations extract the magnitudes for the
.*; division. The DAR is cleared and the carry set by the third instruction.
.*; After the division sequence, an overflow (#) will be forced by the 'ADD'
.*; instruction fourth from the end, if the dividend exceeded the divisor
.*; ( note that SUB....CNDS behaves differently from the other CND inst ).
.*; The last three instructions serve to establish the sign of the result,
.*; and transfer the result to the output. As the division is carried only
.*; to nine bits, the sign correction routine is allowed to leave some
.*; error in positions beyond the thirteenth.
.*EM


*DEFINE MACRO SAW
.*;THIS MACRO GENERATES A SAWTOOTH WAVEFORM
.*;WARNING: TS MUST BE ASSIGNED AN APPROPRIATE VALUE BEFORE CALLING THIS MACRO.
.*;CALLING SEQUENCE:  :SAW CODEFILENAME.EXT, FREQUENCY(HZ), ERROR(HZ), OUTPUTVAR
.*;EXAMPLE:  TO CODE A SAWTOOTH FOR 257 HERTZ WITH AN ERROR OF LESS THAN 0.5HZ
.*;AND SAVE THE RESULTING CODE MODULE IN A FILE NAMED SAW257.ALS WITH THE
.*;OUTPUT VARIABLE NAMED  SAWOSC , YOU ENTER:
.*.;SAW SAW257.ALS, 257, 0.5, SAWOSC
.*IF TS=0 THEN
.*GRAPH IMPULSE     ; FORCE "SAMPLE RATE UNDEFINED" ERROR EXIT
.*ELSE
.*CODE %3 = (%1 * TS * 4) * KM2 + %3 ERROR < %2 * TS * 4;DECREMENT THE SAWTOOTH
.*WRITE '
.**CODING COMPLETED - NOW APPENDING TO FILE
.**'
.*APPEND %0 CODE;BY FREQ X SAMPLETIME
.*APPEND %0 'LDA DAR,%3';
.*APPEND %0 'ADD %3,KP2,L02,CNDS';IF RESULT<0, ADD 1
.*END
.*EM
```

```
*DEFINE MACRO ADCONV
.*'
.*:
.*:ADCONV    MACRO to append to a file the A/D conversion
.*:          routines
.*
.* WARNING     The user should be aware that the correct A/D conversion
.*'            routine (the number of IN instructions, NOP's )is dependent
.*             on the external environment of the system (clock rate,
.*             input sample and hold capacitor etc.) and this macro should
.*             be adjusted accordingly.
.*
.* CALLING SEQUENCE :    ADCONV FILENAME,INPUT STATEMENT, INPUT_NAME
.*                   EX. :ADCONV FILTER.FIL ,IN0, STAGE1IN
.*                        this will put in the file FILTER.FIL the code
.*                        to sample input port 0,do a 9 bit A/D conversion
.*                        and put the result in STAGE1IN
.*'
.*APP %0
.**:A/D CONVERSION ROUTINE ADDED BY MACRO ADCONV
.**%1
.**%1
.**%1
.**NOP
.**NOP
.**CYTS
.**ADD DAR,KM2,R00,CND6
.**NOP
.**NOP'
.*APP %0 'CYT7
.**NOP
.**NOP
.**CYT6
.**NOP
.**NOP
.**CYT5
.**NOP
.**NOP
.**CYT4
.**NOP
.**NOP
.**CYT3
.**NOP
.**NOP
.**CYT2
.**NOP
.**NOP
.**CYT1
.**NOP
.**NOP
.**CYT0
.**NOP
.**NOP
.**LDA %2,DAR ;SCALE INPUT HERE
.**:END OF MACRO ADCONV
.**''
.*EM
*;


*DEFINE MACRO TRIANG
.*:THIS MACRO TRANSFORMS A SAWTOOTH INTO A TRIANGULAR WAVEFORM
.*:CALLING SEQUENCE:TRIANG CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
.*APPEND %0 'LDA %1,%2'
.*APPEND %0 'ADD %1,KM4'; SUBTRACT 1/2
.*APPEND %0 'ABS %1,%1,L01'; TAKE ABSOLUTE VALUE & DOUBLE IT
.*APPEND %0 'ADD %1,KM4'; SUBTRACT 1/2
.*EM
```

```
*DEFINE MACRO SINFIT
.*;THIS MACRO TRANSFORMS A TRIANGULAR WAVEFORM OF AMPLITUDE 1/2
.*;INTO A SINUSOIDAL WAVEFORM OF AMPLITUDE 1
.*;CALLING SEQUENCE:SINFIT CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
.*APPEND %0 'LDA %1,%2,LO2'; THIS TRANSFORMATION USES
.*APPEND %0 'SUB %1,%1,RO2'; A PIECEWISE LINEAR APPROXIMATION
.*APPEND %0 'SUB %1,%1,RO1'; TO  SIN(PI*INPUTVAR)
.*APPEND %0 'SUB %1,%2,RO4'; WHICH IS IMPLEMENTED USING
.*APPEND %0 'ADD %1,%2,RO1'; OVERFLOW SATURATION, THEREFORE
.*APPEND %0 'ADD %1,%2'; LIMITING MUST NOT BE DISABLED.
.*EM

*;
*DEFINE MACRO SINOSC
.*;THIS MACRO GENERATES A SINUSOIDAL WAVEFORM AT A USER-SPECIFIED FREQUENCY
.*;CALLING SEQUENCE:
.*;;SINOSC CODEFILENAME.EXT, FREQUENCY(HZ), FREQERROR(HZ), OUTPUTVARNAME
.*;EXAMPLE: TO GENERATE CODE FOR A SINUSOIDAL OSCILLATOR WHICH HAS A FREQUENCY
.*;OF 245 HERTZ, PLUS OR MINUS AT MOST 0.1 HERTZ, WITH AN OUTPUT NAMED  OSC245
.*;AND APPEND THIS CODE TO DISK FILE MYCODE.ALS, YOU ENTER:
.*;SINOSC MYCODE.ALS, 245, 0.1,  OSC245
.*;SAW %0,%1,%2,SAWTOOTH
.*;TRIANG %0,TRIANGULAR,SAWTOOTH
.*;SINFIT %0,%3,TRIANGULAR
.*EM
```

The following are examples of invoking these macros. The invocation line calls the macro by "colon name", e.g., :MULVAR, followed by the actual parameters to replace the formals in the macro definition. The expansion of the macro then follows, with execution delayed until every command has been verified as conforming with the SPAC20 Compiler language. (It is possible for a valid macro to expand into invalid commands due to the substitution of actuals for formals.)

After this test is passed, execution of the macro proceeds from the first executable command to the last, with the results displayed at the console (and on this listing file). Thus the macro commands, with formals replaced, are seen twice. The first two macros below illustrate this, and enable you to see fully the substitutions performed. This double display has been suppressed in the later examples.

```
*
*TS=1/10000
TS = 0.99999980/10**4
*
*:MULVAR :F1:CHAP11.OUT,PRODUCT,MULTIPLICAND,MULTIPLIER
.*
.*;  This macro generates code for
.*;  a four quadrant, 9-bit multiplication of two variables.
.*; Calling sequence :MULVAR  OUTFILE,PRODUCT,MULTIPLICAND,MULTIPLIER.
.*
.*;                    EXAMPLE :MULVAR  F.MUL,W,X,Y
.*;                             this will put in the file F.MUL the code to
.*;                             implement the equation W=X*Y
.*
.*  APPEND :F1:CHAP11.OUT  '                                ; BEGIN MACRO MULVAR *****'
.*  APPEND :F1:CHAP11.OUT  ' SUB   PRODUCT,   PRODUCT,  RO0          ; Clear the product     *'
.*  APPEND :F1:CHAP11.OUT  ' LDA  DAR,   MULTIPLIER,  RO0          ; Multiplier to DAR     *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO1,  CND7   ;    *     This       *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO2,  CND6   ;    *     is the     *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO3,  CND5   ;    *     multiply   *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO4,  CND4   ;    *     process,   *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO5,  CND3   ;    *     bitwise    *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO6,  CND2   ;    *   adding and   *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO7,  CND1   ;    *     shifting   *'
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO8,  CND0   ;    *                *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   MULTIPLICAND,   MULTIPLICAND,  LO1          ;  These two supply
.*  APPEND :F1:CHAP11.OUT  ' ADD   PRODUCT,   MULTIPLICAND,  RO0,  CNDS   ;  the correct sign.   *'
.*  APPEND :F1:CHAP11.OUT  '                                ; END MACRO MULVAR *******'
.*
.*;  This performs a 9-bit multiply. If more bits of multiplier
.*; precision are required, the high order bits of the multiplier
.*; may be masked off, and the remaining bits shifted left and
.*; loaded to the DAR. The masking operation is necessary to
.*; prevent overflow saturation. ( See AS2920 manual for more info )
.*;  The last two steps above can be eliminated if the multiplier
.*; is known to be positive. The first step must be eliminated
.*; if the operation is to be of the form:   Y = W*X + Y
.*
.*EM
```

```
                              ; BEGIN MACRO MULVAR *****
 SUB   PRODUCT,   PRODUCT,  R00          ; Clear the product     *
 LDA  DAR,   MULTIPLIER,  R00          ; Multiplier to DAR     *
 ADD   PRODUCT,   MULTIPLICAND,  R01,  CND7   ;     *     This        *
 ADD   PRODUCT,   MULTIPLICAND,  R02,  CND6   ;     *     is the      *
 ADD   PRODUCT,   MULTIPLICAND,  R03,  CND5   ;     *     multiply    *
 ADD   PRODUCT,   MULTIPLICAND,  R04,  CND4   ;     *     process,    *
 ADD   PRODUCT,   MULTIPLICAND,  R05,  CND3   ;     *     bitwise     *
 ADD   PRODUCT,   MULTIPLICAND,  R06,  CND2   ;     *    adding and   *
 ADD   PRODUCT,   MULTIPLICAND,  R07,  CND1   ;     *     shifting    *
 ADD   PRODUCT,   MULTIPLICAND,  R08,  CND0   ;     *                 *
 SUB   MULTIPLICAND,   MULTIPLICAND,  L01          ;  These two supply    *
 ADD   PRODUCT,   MULTIPLICAND,  R00,  CNDS   ;  the correct sign.   *
                              ; END MACRO MULVAR *******
*;
*
*
*;
*:DIV :F1:CHAP11.OUT,DIVIDEND,R03,DIVIDSOR,QUOTIENT
.*;
.*;          This macro generates code for a four quadrant division
.*;                    of 2 variables
.*  Calling sequence :DIV  FILE, DIVIDEND, DIVIDEND$SCALE, DIVISOR, QUOTIENT.
.*                EXAMPLE   :DIV  F.DIV,X,R02,Y,W
.*                          this will put in the file F.DIV code to implement
.*;                         the equation   W = ( X * 2**-2 )/ (Y)
.*;
.*  APPEND :F1:CHAP11.OUT '    .                          ; BEGIN MACRO DIV *************'
.*  APPEND :F1:CHAP11.OUT  ' ABS   DV1,  DIVIDEND,  R03          ;  These two extract the     *'
.*  APPEND :F1:CHAP11.OUT  ' ABS   DV2,  DIVIDSOR,  R00          ;      magnitudes.           *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   DAR, DAR,  R00          ;  This clears the DAR.     *'
.*
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R00,  CNDS   ;          This is          *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R01,  CND7'  ;
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R02,  CND6   ;        the divide,        *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R03,  CND5'  ;
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R04,  CND4   ;        progressing        *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R05,  CND3'  ;
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R06,  CND2   ;          one bit          *'
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R07,  CND1'  ;
.*  APPEND :F1:CHAP11.OUT  ' SUB   DV1, DV2,  R08,  CND0   ;         at a time.        *'
.*
.*  APPEND :F1:CHAP11.OUT  ' ADD   DAR, KP4,  L01          ;This forces overflow (#) *'
.*  APPEND :F1:CHAP11.OUT  ' LDA    QUOTIENT,  DIVIDEND,  R13          ;    These two establish    *'
.*  APPEND :F1:CHAP11.OUT  ' XOR    QUOTIENT,  DIVIDSOR,  R13          ;    the correct sign.      *'
.*  APPEND :F1:CHAP11.OUT  ' XOR    QUOTIENT,  DAR              ;   Xfer result to output.  *'
.*  APPEND :F1:CHAP11.OUT  '                                  ; END MACRO DIV **************'
.*;
.*'
.*;  Note that the first two operations extract the magnitudes for the
.*; division. The DAR is cleared and the carry set by the third instruction.
.*; After the division sequence, an overflow (#) will be forced by the 'ADD'
.*; instruction fourth from the end, if the dividend exceeded the divisor
.*; ( note that SUB....CNDS behaves differently from the other CND inst ).
.*; The last three instructions serve to establish the sign of the result,
.*; and transfer the result to the output. As the division is carried only
.*; to nine bits, the sign correction routine is allowed to leave some
.*; error in positions beyond the thirteenth.
.*EM
                              ; BEGIN MACRO DIV *************
 ABS   DV1,  DIVIDEND,  R03          ;  These two extract the     *
 ABS   DV2,  DIVIDSOR,  R00          ;      magnitudes.           *
 SUB   DAR, DAR,  R00          ;  This clears the DAR.     *
 SUB   DV1, DV2,  R00,  CNDS   ;          This is          *
 SUB   DV1, DV2,  R01,  CND7
 SUB   DV1, DV2,  R02,  CND6   ;        the divide,        *
 SUB   DV1, DV2,  R03,  CND5
 SUB   DV1, DV2,  R04,  CND4   ;        progressing        *
 SUB   DV1, DV2,  R05,  CND3
 SUB   DV1, DV2,  R06,  CND2   ;          one bit          *
 SUB   DV1, DV2,  R07,  CND1
 SUB   DV1, DV2,  R08,  CND0   ;         at a time.        *
 ADD   DAR, KP4,  L01          ;This forces overflow (#) *
 LDA    QUOTIENT,  DIVIDEND,  R13          ;    These two establish    *
 XOR    QUOTIENT,  DIVIDSOR,  R13          ;    the correct sign.      *
 XOR    QUOTIENT,  DAR              ;   Xfer result to output.  *
                              ; END MACRO DIV **************
*;
*
*
*
```

```
*;
*;ADCONV ;F1:CHAP11.OUT,IN2,INPUT1ST
.*;
.*;
.*;ADCONV    MACRO to append to a file the A/D conversion
.*           routines
.*
.*;WARNING     The user should be aware that the correct A/D conversion
.*             routine (the number of IN instructions, NOP's )is dependent
.*             on the external environment of the system (clock rate,
.*             input sample and hold capacitor etc.) and this macro should
.*             be adjusted accordingly.
.*
.*;CALLING SEQUENCE :    ADCONV FILENAME,INPUT STATEMENT, INPUT_NAME
.*                    EX. :ADCONV FILTER.FIL ,IN0, STAGE1IN
.*                         this will put in the file FILTER.FIL the code
.*                         to sample input port 0,do a 9 bit A/D conversion
.*                         and put the result in STAGE1IN
.*
.*APP ;F1:CHAP11.OUT
.**;A/D CONVERSION ROUTINE ADDED BY MACRO ADCONV
.**;IN2
.**;IN2
.**;IN2
.**NOP
.**NOP
.**CVT5
.**ADD DAR,KM2,R00,CND6
.**NOP
.**NOP'
.*APP ;F1:CHAP11.OUT 'CVT7
.**NOP
.**NOP
.**CVT6
.**NOP
.**NOP
.**CVT5
.**NOP
.**NOP
.**CVT4
.**NOP
.**NOP
.**CVT3
.**NOP
.**NOP
.**CVT2
.**NOP
.**NOP
.**CVT1
.**NOP
.**NOP
.**CVT0
.**NOP
.**NOP
.**LDA INPUT1ST,DAR ;SCALE INPUT HERE
.**;END OF MACRO ADCONV
.**;
.*EM
```

```
;A/D CONVERSION ROUTINE ADDED BY MACRO ADCONV
IN2
IN2
IN2
NOP
NOP
CVT5
ADD DAR,KM2,ROO,CND6
NOP
NOP
CVT7
NOP
NOP
CVT6
NOP
NOP
CVT5
NOP
NOP
CVT4
NOP
NOP
CVT3
NOP
NOP
CVT2
NOP
NOP
CVT1
NOP
NOP
CVT0
NOP
NOP
LDA INPUT1ST,DAR ;SCALE INPUT HERE
;END OF MACRO ADCONV
;
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*;
*:SAW :F1:CHAP11.OUT,257,0.5,SAWOSC
.*:THIS MACRO GENERATES A SAWTOOTH WAVEFORM
.*:WARNING: TS MUST BE ASSIGNED AN APPROPRIATE VALUE BEFORE CALLING THIS MACRO.
.*:CALLING SEQUENCE:  :SAW CODEFILENAME.EXT, FREQUENCY(HZ), ERROR(HZ), OUTPUTVAR
.*:EXAMPLE:  TO CODE A SAWTOOTH FOR 257 HERTZ WITH AN ERROR OF LESS THAN 0.5HZ
.*:AND SAVE THE RESULTING CODE MODULE IN A FILE NAMED SAW257.ALS WITH THE
.*:OUTPUT VARIABLE NAMED  SAWOSC , YOU ENTER:
.*:SAW SAW257.ALS, 257, 0.5, SAWOSC
.*IF TS=0 THEN
..*GRAPH IMPULSE     ; FORCE "SAMPLE RATE UNDEFINED" ERROR EXIT
..*ELSE
..*CODE SAWOSC = (257 * TS * 4) * KM2 + SAWOSC ERROR < 0.5 * TS * 4;DECREMENT THE SAWTOOTH
..*WRITE '
..**CODING COMPLETED - NOW APPENDING TO FILE
..**'
..*APPEND :F1:CHAP11.OUT CODE;BY FREQ X SAMPLETIME
..*APPEND :F1:CHAP11.OUT 'LDA DAR,SAWOSC';
..*APPEND :F1:CHAP11.OUT 'ADD SAWOSC,KP2,L02,CND5';IF RESULT<0, ADD 1
..*END
.*EM
```

```
ADD SAWOSC,KM2,R03
    ; SAWOSC=1.00000000*SAWOSC+0.125000000*KM2
SUB SAWOSC,KM2,R06
    ; SAWOSC=1.00000000*SAWOSC+0.109375000*KM2
ADD SAWOSC,KM2,R10
    ; SAWOSC=1.00000000*SAWOSC+0.110351562*KM2
ADD SAWOSC,KM2,R12
    ; SAWOSC=1.00000000*SAWOSC+0.110595703*KM2
SUB SAWOSC,KM2,R07
    ; SAWOSC=1.00000000*SAWOSC+0.102783203*KM2

CODING COMPLETED - NOW APPENDING TO FILE
ADD SAWOSC,KM2,R03
    ; SAWOSC=1.00000000*SAWOSC+0.125000000*KM2
SUB SAWOSC,KM2,R06
    ; SAWOSC=1.00000000*SAWOSC+0.109375000*KM2
ADD SAWOSC,KM2,R10
    ; SAWOSC=1.00000000*SAWOSC+0.110351562*KM2
ADD SAWOSC,KM2,R12
    ; SAWOSC=1.00000000*SAWOSC+0.110595703*KM2
SUB SAWOSC,KM2,R07
    ; SAWOSC=1.00000000*SAWOSC+0.102783203*KM2
LDA DAR,SAWOSC
ADD SAWOSC,KP2,L02,CNDS
*
*
*
*
*
*
*
*
*
*
*
*
*
*
*:TRIANG :F1:CHAP11.OUT,OUT1,IN1
.*:THIS MACRO TRANSFORMS A SAWTOOTH INTO A TRIANGULAR WAVEFORM
.*:CALLING SEQUENCE:TRIANG CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
.*APPEND :F1:CHAP11.OUT 'LDA OUT1,IN1'
.*APPEND :F1:CHAP11.OUT 'ADD OUT1,KM4'; SUBTRACT 1/2
.*APPEND :F1:CHAP11.OUT 'ABS OUT1,OUT1,L01'; TAKE ABSOLUTE VALUE & DOUBLE IT
.*APPEND :F1:CHAP11.OUT 'ADD OUT1,KM4'; SUBTRACT 1/2
.*EM
LDA OUT1,IN1
ADD OUT1,KM4
ABS OUT1,OUT1,L01
ADD OUT1,KM4
*
*:SINFIT :F1:CHAP11.OUT,OUT,IN
.*:THIS MACRO TRANSFORMS A TRIANGULAR WAVEFORM OF AMPLITUDE 1/2
.*:INTO A SINUSOIDAL WAVEFORM OF AMPLITUDE 1
.*:CALLING SEQUENCE:SINFIT CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
.*APPEND :F1:CHAP11.OUT 'LDA OUT,IN,L02'; THIS TRANSFORMATION USES
.*APPEND :F1:CHAP11.OUT 'SUB OUT,OUT,R02'; A PIECEWISE LINEAR APPROXIMATION
.*APPEND :F1:CHAP11.OUT 'SUB OUT,OUT,R01'; TO  SIN(PI*INPUTVAR)
.*APPEND :F1:CHAP11.OUT 'SUB OUT,IN,R04'; WHICH IS IMPLEMENTED USING
.*APPEND :F1:CHAP11.OUT 'ADD OUT,IN,R01'; OVERFLOW SATURATION, THEREFORE
.*APPEND :F1:CHAP11.OUT 'ADD OUT,IN'; LIMITING MUST NOT BE DISABLED.
.*EM
LDA OUT,IN,L02
SUB OUT,OUT,R02
SUB OUT,OUT,R01
SUB OUT,IN,R04
ADD OUT,IN,R01
ADD OUT,IN
*
*
```

```
*;SINOSC :F1:CHAP11.OUT,245,0.1,OSC
.*;THIS MACRO GENERATES A SINUSOIDAL WAVEFORM AT A USER-SPECIFIED FREQUENCY
.*;CALLING SEQUENCE:
.*;;SINOSC CODEFILENAME.EXT, FREQUENCY(HZ), FREQERROR(HZ), OUTPUTVARNAME
.*;EXAMPLE: TO GENERATE CODE FOR A SINUSOIDAL OSCILLATOR WHICH HAS A FREQUENCY
.*.OF 245 HERTZ, PLUS OR MINUS AT MOST 0.1 HERTZ, WITH AN OUTPUT NAMED  OSC245
.*;AND APPEND THIS CODE TO DISK FILE MYCODE.ALS, YOU ENTER:
.*;SINOSC MYCODE.ALS, 245, 0.1,  OSC245
.*;SAW :F1:CHAP11.OUT,245,0.1,SAWTOOTH
..*;THIS MACRO GENERATES A SAWTOOTH WAVEFORM
..*;WARNING: TS MUST BE ASSIGNED AN APPROPRIATE VALUE BEFORE CALLING THIS MACRO.
..*;CALLING SEQUENCE:  :SAW CODEFILENAME.EXT, FREQUENCY(HZ), ERROR(HZ), OUTPUTVAR
..*;EXAMPLE:  TO CODE A SAWTOOTH FOR 257 HERTZ WITH AN ERROR OF LESS THAN 0.5HZ
..*;AND SAVE THE RESULTING CODE MODULE IN A FILE NAMED SAW257.ALS WITH THE
..*;OUTPUT VARIABLE NAMED  SAWOSC , YOU ENTER:
..*;;SAW SAW257.ALS, 257, 0.5, SAWOSC
..*IF TS=0 THEN
.. *GRAPH IMPULSE     ; FORCE "SAMPLE RATE UNDEFINED" ERROR EXIT
.. *ELSE
..*CODE SAWTOOTH = (245 * TS * 4) * KM2 + SAWTOOTH ERROR < 0.1 * TS * 4;DECREMENT THE SAWTOOTH
.. *WRITE '
.. **CODING COMPLETED - NOW APPENDING TO FILE
.. **'
.. *APPEND :F1:CHAP11.OUT CODE;BY FREQ X SAMPLETIME
.. *APPEND :F1:CHAP11.OUT 'LDA DAR,SAWTOOTH';
.. *APPEND :F1:CHAP11.OUT 'ADD SAWTOOTH,KP2,LO2,CNDS';IF RESULT<0, ADD 1
.. *END
..*EM
.*;TRIANG :F1:CHAP11.OUT,TRIANGULAR,SAWTOOTH
..*;THIS MACRO TRANSFORMS A SAWTOOTH INTO A TRIANGULAR WAVEFORM
..*;CALLING SEQUENCE:TRIANG CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
..*APPEND :F1:CHAP11.OUT 'LDA TRIANGULAR,SAWTOOTH'
..*APPEND :F1:CHAP11.OUT 'ADD TRIANGULAR,KM4'; SUBTRACT 1/2
..*APPEND :F1:CHAP11.OUT 'ABS TRIANGULAR,TRIANGULAR,LO1'; TAKE ABSOLUTE VALUE & DOUBLE IT
..*APPEND :F1:CHAP11.OUT 'ADD TRIANGULAR,KM4'; SUBTRACT 1/2
..*EM
.*;SINFIT :F1:CHAP11.OUT,OSC,TRIANGULAR
..*;THIS MACRO TRANSFORMS A TRIANGULAR WAVEFORM OF AMPLITUDE 1/2
..*;INTO A SINUSOIDAL WAVEFORM OF AMPLITUDE 1
..*;CALLING SEQUENCE:SINFIT CODEFILENAME.EXT, OUTPUTVAR, INPUTVAR
..*APPEND :F1:CHAP11.OUT 'LDA OSC,TRIANGULAR,LO2'; THIS TRANSFORMATION USES
..*APPEND :F1:CHAP11.OUT 'SUB OSC,OSC,RO2'; A PIECEWISE LINEAR APPROXIMATION
..*APPEND :F1:CHAP11.OUT 'SUB OSC,OSC,RO1'; TO  SIN(PI*INPUTVAR)
..*APPEND :F1:CHAP11.OUT 'SUB OSC,TRIANGULAR,RO4'; WHICH IS IMPLEMENTED USING
..*APPEND :F1:CHAP11.OUT 'ADD OSC,TRIANGULAR,RO1'; OVERFLOW SATURATION, THEREFORE
..*APPEND :F1:CHAP11.OUT 'ADD OSC,TRIANGULAR'; LIMITING MUST NOT BE DISABLED.
..*EM
.*EM
```

```
ADD SAWTOOTH,KM2,R03
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.125000000*KM2
ADD SAWTOOTH,KM2,R08
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.128906250*KM2
ADD SAWTOOTH,KM2,R12
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.129150390*KM2
ADD SAWTOOTH,KM2,R13
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.129272460*KM2
SUB SAWTOOTH,KM2,R05
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.098022460*KM2

CODING COMPLETED - NOW APPENDING TO FILE
ADD SAWTOOTH,KM2,R03
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.125000000*KM2
ADD SAWTOOTH,KM2,R08
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.128906250*KM2
ADD SAWTOOTH,KM2,R12
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.129150390*KM2
ADD SAWTOOTH,KM2,R13
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.129272460*KM2
SUB SAWTOOTH,KM2,R05
     ; SAWTOOTH=1.00000000*SAWTOOTH+0.098022460*KM2
LDA DAR,SAWTOOTH
ADD SAWTOOTH,KP2,L02,CNDS
LDA TRIANGULAR,SAWTOOTH
ADD TRIANGULAR,KM4
ABS TRIANGULAR,TRIANGULAR,L01
ADD TRIANGULAR,KM4
LDA OSC,TRIANGULAR,L02
SUB OSC,OSC,R02
SUB OSC,OSC,R01
SUB OSC,TRIANGULAR,R04
ADD OSC,TRIANGULAR,R01
ADD OSC,TRIANGULAR
*
*
*
*
*
*
*
*
*
*EXIT
```

APPENDIX A
HELP MESSAGES

```
*HELP
***  Help is available for the following items.  Type HELP followed  ***
***  by the item name.  Do not type the angle brackets.  (For more   ***
***  information on the help command, type HELP HELP.)               ***
Filters and Filter Responses:
   DEFINE,<FILTER$RESPONSE>,GREF,HOLD,MOVE,<POLE$OR$ZERO$LOCATION>,
   <POLES$OR$ZEROES>,REMOVE
Graphics:
   FSCALE,GRAPH,YSCALE
Code Generation:
   BOUNDS,CODE,<MA$CONSTRAINT>,<PZ$CONSTRAINT>
File/Display/Compound Commands:
   <COMPOUND$COMMANDS>,EVALUATE,EXIT,<FILE$COMMANDS>,HELP,MACRO,
   <PATH$NAME>,WRITE
Miscellaneous:
   <BOOLEAN$EXPR>,<CONSTANT>,<EXPR>,<FUNCTION>,<IDENTIFIER>,
   <INTEGER$EXPR>,<NUMERIC$CONSTANT>,<PRIMARY>,<PZ$REF>,<SPAC$REF>,
   <STRING>,<SYMBOL>,<SYMBOLIC$REF>
*
*HELP *
<BOOLEAN$EXPR> - A Boolean valued (TRUE or FALSE) expression:
   <BOOLEAN$PRIMARY> [<BOOLEAN$OPERATOR> <BOOLEAN$PRIMARY>]*
      Boolean primaries connected with logical operators.
         Ex:   (.I < 10) AND (.I > 5)
<BOOLEAN$OPERATOR>:
   AND   OR   XOR
<BOOLEAN$PRIMARY> - Either a comparison of two (arithmetic) expressions
or a parenthesized Boolean expression:
   [NOT] <EXPR> <COMPARISON> <EXPR>
      Ex:   SIN(.X+1/.X) > .5
   ( <BOOLEAN$EXPR> )
      Ex:   ((.I <> 0) OR (.I<>1))
<COMPARISON>:
   <   <=   =   >=   >   <>

<BOUNDS> - Piecewise linear upper and lower bounds on the gain.
Used during coding to constrain the gain of the coded filter.
   LBOUND/UBOUND
      Display the current lower or upper bound setting.
         Ex:   LBO
   LBOUND/UBOUND = <EXPR> AT <EXPR> [, [,] <EXPR> AT <EXPR>]*
      Each <EXPR> AT <EXPR> establishes a vertex of the piecewise
      linear bound.  Two commas in a row indicate a don't care region.
         Ex:   UBO = 0 AT 10, 0 AT 59, -20 AT 60, 0 AT 61, 0 AT 1000
               LBO = -1 AT 10, -1 AT 59, , -1 AT 61, -1 AT 1000
The deviation of the gain from these bounds is revealed by the
<SPAC$REF>s MSQE, and MERROR, and the <FILTER$RESPONSE> GERROR.
See <FILTER$RESPONSE>, <GRAPH>, <PZ$CONSTRAINT>, <SPAC$REF>.

CODE - Command to generate 2920 assembly code into code buffer or
display current code buffer contents:
   CODE
      Display current code buffer contents.
   CODE POLE/ZERO <INTEGER$EXPR> [<PZ$CONSTRAINT>]
      Generate code for specified pole or zero subject to the specified
      constraints.  The default constraint is PERROR<0.0, INST<20
      (i.e. minimize positional error in fewer than 20 instructions).
         Ex:   CODE POLE 1
               CODE ZERO 3 MSQE<.01, INST<11
   CODE <IDENTIFIER> = <PRIMARY> * <IDENTIFIER> [+ <IDENTIFIER>]
   [<MA$CONSTRAINT>]
      Generate code for multiplication of one variable and a
      constant, leaving result in another (or same) variable.
      If the third <IDENTIFIER> is present, it must be same as first;
      code generated adds the result of the multiplication to the
      first variable.  The default constraint is
      ERROR< <PRIMARY>/2**16, INST<20.
         Ex:   CODE Y = (1/3)*X+Y
               CODE INO_P1 = .03*DAR INST<5
               COD X1 = .965*X1 ERROR<.002
```

```
<COMPOUND$COMMANDS> - Several compound commands exist:
   IF <BOOLEAN$EXPR> [THEN] <CR> ..... [ORIF <BOOLEAN$EXPR> [THEN]
   <CR> .....]* [ELSE <CR> .....] END
      Here ..... indicates any number of commands (possibly compound).
      <CR> is carriage-return.
         Ex:   IF MAGAIN>1 THEN
                 WRITE 'SCALE BY ', LOG(MAGAIN)/LOG(2)
               END
Two other compound commands allow looping:
   REPEAT <CR> [..... [WHILE/UNTIL <BOOLEAN$EXPR> <CR>]]* END
      The ..... commands  are repeated until the WHILE or UNTIL clause
      permits termination or until the Escape key is pressed.
   COUNT <INTEGER$EXPR> [..... [WHILE/UNTIL <BOOLEAN$EXPR> <CR>]]*
   END
      As in REPEAT, but loop terminates after a specified number
      of passes.
         Ex:   COUNT 20
                 EVALUATE SIN(.N*TPI/360)
                 .N = .N+1
               END

<CONSTANT> - A keyword which has a fixed numeric value.
   PI       3.1415927
   TPI      Twice PI
   HPI      Half of PI

DEFINE - Command to define a pole or zero, a symbol, or a macro:
   DEFINE POLE/ZERO <INTEGER$EXPR> = <POLE$OR$ZERO$LOCATION>
      Create a pole or zero at the specified location.
         Ex:   DEFINE POLE 0 = -200,0,TS
               DEFINE Z 1000 = 1.01,PI/10,Z
   DEFINE .<SYMBOL$NAME> = <EXPR>
      Add <SYMBOL$NAME> to end of symbol table and assign it a value.
         Ex:   DEFINE .LABEL = 100
   DEFINE MACRO <MAC$NAME> <CR> ..... <CR> EM
      Define a macro to consist of ....., a sequence of commands.
      Formal parameters %0, %1, ..., %9 will be substituted when macro
      is invoked.  <CR> is carriage-return.
See MACRO, SYMBOL.

EVALUATE - Command to evaluate an expression.  The value is displayed
in decimal scientific notation.
   EVALUATE <EXPR>
      Ex:   EVAL (LOG(MAGAIN)/LOG(2)) MASK 255

EXIT - Command to exit the debugging session and return control to
ISIS-II.
   EXIT

<EXPR> - A numeric value expressed as an algebraic sequence of
operand(s) and operator(s).  It has the following form:
   <OPERAND> [<OPERATOR> <OPERAND>]*
<OPERAND>:
   [<UNARY$OP>] <PRIMARY>
<UNARY$OP>:
   +    -
<PRIMARY>:
   <NUMERIC$CONSTANT>      (Ex:   1979, 0.1011B, 0.FFF5H, 3.14159D)
   <SYMBOLIC$REF>          (Ex:   .MAX$AGAIN, .STAGE$2)
   <SPAC$REF>              (Ex:   TS, INST, MSGE)
   <CONSTANT>              (Ex:   PI, HPI, TPI)
   <FUNCTION> (<EXPR>)     (Ex:   SIN(TPI*TS*60), GAIN(60))
   <PZ$REF>                (Ex:   REAL(POLE 1), ANGLE(Z 99))
   (<EXPR>)                (Ex:   (LOG(AGAIN)/LOG(2)), (.FRED+14D))
   (<BOOLEAN$EXPR>)        (TRUE=1, FALSE=0;   Ex:   (.X<100))
<OPERATOR>:
   **   *   /   +   -   MOD   MASK
```

```
<FILE$COMMANDS> - Several commands manipulate ISIS files or devices:
   LIST <PATH$NAME>
      Send a copy of all console input and output to specified
      log file or device.  To terminate copying, type LIST :CO:.
   DISPLAY <PATH$NAME>
      A command to display an ISIS file.  The display can be
      interrupted with the Escape key.
         Ex:   DISPLAY :F1:PZ.TMP
   PUT/APPEND <PATH$NAME> [<FILE$OBJECT> [, <FILE$OBJECT>]*]
      Write the specified objects to an ISIS file or device.  PUT
      indicates the file should be deleted first if it already
      exists.  APPEND indicates the objects should be appended to
      the end of the file which will be created if it does not
      yet exist.  If no objects are specified, commands are output
      which when INCLUDED will recreate the current state.
         Ex:   PUT :F1:PZ.TMP PZ
               APPEND :F1:FILTER.SRC '; STAGE ', .I, CODE
   INCLUDE <PATH$NAME>
      Take commands from specified file.
<FILE$OBJECT>:
   CODE   PZ   MACRO   SYMBOLS   BOUNDS   <STRING>   <EXPR>

<FILTER$RESPONSE> - One of the following filter responses:
   GAIN       Gain in decibels as a function of frequency in Hertz.
   AGAIN      Absolute gain as a function of frequency in Hertz.
   PHASE      Phase delay in radians as a function of frequency in Hertz.
   GROUP      Group delay in seconds as a function of frequency in Hertz.
   GERROR     Signed deviation of GAIN from the LBOUND and UBOUND
              in decibels as a function of frequency in Hertz.
              See <BOUNDS>.
   IMPULSE    Impulse response as a function of time in seconds.
   STEP       Step response as a function of time in seconds.
Each <FILTER$RESPONSE> can be used as a command to tabulate the values.
Each can be graphed.  Each except IMPULSE and STEP can be used as a
function in an expression, e.g. AGAIN(60).
See <FUNCTION>, GRAPH, <PRIMARY>.

FSCALE - A command to display or specify the frequency range of interest
during calculation and graphing of filter responses:
   FSCALE
      Display current frequency scale.
   FSCALE = <EXPR> , <EXPR> [, <EXPR>]*
      Break the horizontal range of the graph into a number of piecewise
      logarithmic segments.  A nearly linear frequency scale can be
      created this way.  At most 10 values are allowed.
         Ex:   FSC = 10, 10000    (initial condition)
               FSCALE = 1000,1500,2000,2500,3000,3500    (nearly linear)

<FUNCTION> - A keyword which invokes a predefined function of its
argument when appearing in a <PRIMARY> or <EXPR>.
   SIN      Sine function
   COS      Cosine function
   TAN      Tangent function
   ASIN     Inverse sine function
   ACOS     Inverse cosine function
   ATAN     Inverse tangent function
   EXP      Exponentiation of e (2.7182817)
   LOG      Natural logarithm (inverse of EXP)
   SQR      Square root
   ABS      Absolute value
   <FILTER$RESPONSE>   Any frequency dependent filter response can be
            invoked as a function:   Ex:   PHASE(60)
See <EXPR>, <FILTER$RESPONSE>, <PRIMARY>.

GRAPH - A command to graph a filter response or bound:
   GRAPH
      Redisplay the previous graph.
   GRAPH <FILTER$RESPONSE>/LBOUND/UBOUND
      Graph the specified filter response or bound.
         Ex:  GRAPH GAIN
   OGRAPH <FILTER$RESPONSE>/LBOUND/UBOUND
      Graph the specified filter response or bound on top of
      the last curve graphed.  Old curve is indicated with plus signs
      in display.
         Ex:   OGR LBOUND
See <BOUNDS>, <FILTER$RESPONSE>.
```

```
GREF - Set or display the reference frequency and decibel gain used
in calculating gain:
   GREF
      Display current gain reference.
   GREF = <EXPR> AT <EXPR>
      Specify the gain to be a certain value at a certain frequency.
         Ex:   GREF = 0 AT 440    (0 decibels at 440 Hz)
The absolute gain (AGAIN) at the reference frequency must be nonzero.

HELP - Command to display a summary of the syntax of a command
or component of a command.
   HELP
      Display a list of all items for which there is help.
    - Items appearing in this list without angle brackets are
      command keywords.  Items with angle brackets are command
      components.
   HELP <IDENTIFIER>
      Display summary of specified item.  <IDENTIFIER> may contain
      dollar signs, but not angle brackets.
         Ex:   HELP POLES$OR$ZEROES
   HELP *
      Display the summaries of all items.
The following notation is used in the help summaries:
    [A]     means A is optional.
    [A]*    means A is optional and may be repeated any number of times.
    A/B     means either A or B may be used.
    <A>     means there is also a help summary for the command
            component A.

HOLD - Command to turn on or off sample and hold compensation, or to
display the current state:
   HOLD
     Display hold on or off.
   HOLD ON
     Turn compensation on.  This should be used when examining the
     characteristics of entire 2920 filter implementation.
   HOLD OFF
     Turn compensation off.  This should be used when calculating
     scaling factors between filter stages or when using the SPAC20
     compiler to design analog circuits.

<IDENTIFIER> - A sequence of one or more of the following characters:
         ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789@?_
The first character can not be a decimal digit and only the first 31
characters are significant.  Dollar signs are allowed but are ignored.
   Ex:   SCALE$FACTOR   ?TEMP$1   IN0_P12

<INTEGER$EXPR> - An expression <EXPR> with an integer value:
   Ex:   PI - (PI MOD 1)
         15.000
         (LOG(MAGAIN)/LOG(2)) MASK 255
See <EXPR>.

<MA$CONSTRAINT> - In a CODE multiplication command, specifies the
constraint guiding the code generation:
   ERROR < <EXPR> [, INST < <EXPR>]
      Generate fewer than a specified number of instructions which
      effect a multiplication by a constant which differs by less
      than a specified value from the desired constant.
      The default INST constraint is INST<20.
         Ex:   ERROR<.0001
               ERROR<0, INST<10    (minimize ERROR with 9 instructions)

   INST < <EXPR>
      Generate fewer than a specified number of instructions which
      multiply by a constant differing by less than one part in 2**16.
         Ex:   INST < 10
After coding has been completed, the <SPAC$REF> ERROR gives the
signed difference between the requested multiplier and the actual
multiplier.  See CODE, <SPAC$REF>.
```

```
MACRO - Macros are user-created, named, command sequences.  Parameters
are substituted when macros are invoked.  Macros are analagous to
the ISIS SUBMIT facility:
   MACRO [<MAC$NAME> [, <MAC$NAME>]*]
      Display all, or only the specified macro definitions
   DIR MACRO
      Display macro directory, i.e. the list of defined macro names.
   REMOVE MACRO [<MAC$NAME> [, <MAC$NAME>]*]
      Remove all, or only the specified macro definitions.
   DEFINE MACRO <MAC$NAME> <CR> ..... <CR> EM
      Define a macro to consist of ....., a sequence of commands.
      Formal parameters %0, %1, ..., %9 will be substituted when macro
      is invoked.  <CR> is carriage-return.
   : <MAC$NAME> [<PARAMETER> [, [<PARAMETER>]]*]
      Invoke a macro with specified parameters.
         Ex:   :MOG POLE 1, .001
<PARAMETER> - Any sequence of tokens not containing a comma or <CR>,
or a quoted string <STRING> possibly containing a comma or <CR>.
   Ex:   POLE .I+1
         'A,B,C'
<MAC$NAME> - Any <IDENTIFIER>.

MOVE - Command to move the locations of existing poles and zeroes.
   MOVE <POLES$OR$ZEROES> TO CONTINUOUS/TS/Z
      Move some poles or zeroes to a different plane.  Movement from
      CONTINUOUS/TS to Z involves the matched-Z transform, and in
      the other direction, the inverse matched-Z transform.
         Ex:   MOVE PZ TO TS    (convert to sampled filter)
   MOVE <POLES$OR$ZEROES> TO <POLE$OR$ZERO$LOCATION>
      Redefine the position and possibly the plane of the specified
      poles or zeroes.
         Ex:   MOVE POLE 1 TO -20,1010    (same plane)
               MOVE ZEROES TO 0.0,TS      (new plane)
   MOVE <POLES$OR$ZEROES> BY <EXPR> , <EXPR>
      Change the coordinates of the specified poles or zeroes by the
      two expression values.
         Ex:   MOVE P 10 BY .01,0    (change radius if Z-plane pole)
               MOVE POLES BY 0,10    (change frequency if S-plane pole)

<NUMERIC$CONSTANT> - A sequence of numeric characters (digits)
optionally including a period (".") followed by an optional suffix to
specify the constant's base.  If no suffix is specified, then the
constant is evaluated with default decimal suffix.  Dollar
signs may be used to improve readability, but are ignored.
   Ex:   0.1001$1011B      (binary)
         .999D             (decimal)
         1FA9H             (hexadecimal)
         1.5               (default decimal)

<PATHNAME> - The name of an ISIS-II file or device:
      Ex:   :F1:MYPROG
            :F2:TEST.001
            :LP:

<POLE$OR$ZERO$LOCATION> - The location of a pole or zero on one of the
three planes:
   <EXPR> , <EXPR> [, CONTINUOUS/TS/Z]
      Specifies coordinates in a plane.  If CONTINUOUS (S-plane) or
      TS (sampled S-plane), coordinates refer to real and imaginary
      components expressed in Hertz.  If Z (sampled Z-plane),
      coordinates are polar;  the first is the radius and the second
      is the angle in radians.   If no plane is specified when
      defining a pole or zero, the default is CONTINUOUS.  If no
      plane is specified when moving a pole or zero, the plane remains
      unchanged.
         Ex:   -5, 60    (60 Hz)
               -5, 60, TS    (60 Hz sampled)
               .99, TPI/5, Z    (One fifth the sample rate)
```

```
<POLES$OR$ZEROES> - A specification of a range of poles, a range of
zeroes, all poles, all zeroes, or all poles and zeroes:
   POLE <INTEGER$EXPR> [THROUGH <INTEGER$EXPR>]
      One or more poles specified by integer labels.
         Ex:   PO 7
               POLES 10 THRU 19
   POLE
      All poles.
   ZERO <INTEGER$EXPR> [THROUGH <INTEGER$EXPR>]
      One or more zeroes specified by integer labels.
         Ex:   ZERO 32
               ZEROES 10 THRU 19
   ZERO
      All zeroes.
   PZ
      All poles and all zeroes.


<PRIMARY> -  A numeric value.  It has the following forms:
   <NUMERIC$CONSTANT>      (Ex:   1979, 0.1011B, 0.FFF5H, 3.14159D)
   <SYMBOLIC$REF>          (Ex:   .MAX$AGAIN, .STAGE$2)
   <SPAC$REF>              (Ex:   TS, INST, MSQE)
   <CONSTANT>              (Ex:   PI, HPI, TPI)
   <FUNCTION> (<EXPR>)     (Ex:   SIN(TPI*TS*60), GAIN(60))
   <PZ$REF>                (Ex:   REAL(POLE 1), ANGLE(Z 99))
   (<EXPR>)                (Ex:   (LOG(AGAIN)/LOG(2)), (.FRED+14D))
   (<BOOLEAN$EXPR>)        (TRUE=1, FALSE=0;   Ex:   (.X<100))

<PZ$CONSTRAINT> - In a CODE pole/zero command, specifies the
constraint guiding the code generation:
   PERROR < <EXPR> , <EXPR> [, INST < <EXPR>]
      Generate fewer than a specified number of instructions
      which implement a pole or zero whose coordinates differ from
      the original coordinates by less than the two specified values.
      The default INST constraint is INST < 20.
         Ex:   PER < 5,10, INST < 14
               PERROR < 0,0    (minimize positional error)
   MSQE < <EXPR> [, INST < <EXPR>]
      Generate code so that the mean squared deviation of the gain
      from the gain bounds (i.e. MSQE) is less than a specified value.
         Ex:   MSQE < .1
               MSQE<0, INST<10    (minimize MSQE with 9 instructions)
   MERROR < <EXPR> [, INST < <EXPR>]
      Generate code so that the maximum absolute deviation of the gain
      from the gain bounds (i.e. MERROR) is less than a specified value.
         Ex:   MERROR < .1
               MER<0, INST<10    (minimize MERROR)
   INST < <EXPR>
      Minimize positional error.  Same as PERROR<0,0, INST< <EXPR>.
See <BOUNDS>, CODE, <SPAC$REF>.

<PZ$REF> - A reference to a coordinate of a pole or zero location,
used as a <PRIMARY> in an expression <EXPR>:
   REAL ( POLE/ZERO <INTEGER$EXPR> )
      X coordinate of specified pole or zero in Cartesian coordinates.
         Ex:   REAL (POLE 3)
   IMAGINARY ( POLE/ZERO <INTEGER$EXPR> )
      Y coordinate of specified pole or zero in Cartesian coordinates.
   RADIUS ( POLE/ZERO <INTEGER$EXPR> )
      Radius of specified pole or zero in polar coordinates.
   ANGLE ( POLE/ZERO <INTEGER$EXPR> )
      Angle in radians of specified pole or zero in polar coordinates.
For a pole or zero in an S-plane the X and Y coordinates are the same
as the <POLE$OR$ZERO$LOCATION> and the radius and angle are the result
of converting these to polar coordinates.  For a pole or zero in the
Z-plane the radius and angle are the same as the <POLE$OR$ZERO$LOCATION>
and the X and Y coordinates are the result of converting these to
rectangular coordinates.
```

```
REMOVE - Command to delete a pole or zero, a symbol, or a macro.
   REMOVE <POLES$OR$ZEROES>
      Remove poles or zeroes specified by integer labels.
         Ex:   REMOVE POLE 1
               REM Z .BLOCK$1 THRU .BLOCK$1+10
               REMOVE POLES
               REM PZ
   REMOVE <SYMBOLIC$REF> [, <SYMBOLIC$REF>]*
      Remove one or more symbols from the symbol table.
         Ex:   REMOVE .LABELX
   REMOVE SYMBOLS
      Remove all symbols from the symbol table.
   REMOVE MACRO <MAC$NAME> [, <MAC$NAME>]*
      Remove one or more macro definitions.
         Ex:   REMOVE MACRO CHEB
   REMOVE MACROS
      Remove all macros.
See MACRO, SYMBOL.

<SPAC$REF> - A keyword reference to an SPAC20 system variable.  Each may
be displayed by simply typing the keyword.  Those which are not read
only may be changed by typing the keyword followed by "=" and an <EXPR>.
Each may be used as a <PRIMARY> in an expression <EXPR>.
   ERROR      Signed error in multiplier from last code multiplication
              command (read only).
   INST       Number of instructions in code buffer generated by last
              code command (read only).
   MAGAIN     Maximum absolute gain over frequencies in the
              frequency scale (read only).
   MERROR     Maximum absolute deviation of gain from lower and upper
              bounds (see BOUNDS) (read only).
   MSQE       Mean square deviation of gain from lower and upper bounds
              (see BOUNDS) (read only).
   TS         Sample time in seconds (positive nonzero).
   XSIZE      CRT screen width.  Also determines frequency scale and time
              scale widths (12<=XSIZE<=79).
   YSIZE      CRT screen height (5<=YSIZE<=25).

<STRING> - A quoted string of characters used in a WRITE, PUT, or
APPEND command, or used as a macro parameter.
   Ex:   'HELLO'
         'POLE 1     ; ANYTHING''S ALLOWED IN A STRING'

SYMBOL - Refers to a symbol in the symbol table:
   <SYMBOLIC$REF>
      Display the value of a symbol.
         Ex:   .STAGE$1
               .SUM
   SYMBOL
      Display the entire symbol table.
   DEFINE .<SYMBOL$NAME> = <EXPR>
      Add <SYMBOL$NAME> to end of symbol table and assign it a value.
         Ex:   DEFINE .FACTOR = .13FH
   <SYMBOLIC$REF> = <EXPR>
      Change the value of a symbol.
         Ex:   .LABELX = 5
               .FACTOR = .ONE/16
   REMOVE <SYMBOLIC$REF> [,<SYMBOLIC$REF>]*
      Remove one or more symbols from the symbol table.
         Ex:   REMOVE .SAVEIT
   REMOVE SYMBOL
      Remove all symbols from the symbol table.
<SYMBOL$NAME> - An <IDENTIFIER>.

<SYMBOLIC$REF> - A reference to a symbol in the symbol table.
See SYMBOL for related commands.
   .<SYMBOL$NAME>
      Access symbol in table with specified symbol name.
         Ex:   .LOOP$INDEX

WRITE - A command to display to the console (and list device) strings
and/or expression values.  Most useful in compound commands and macros:
   WRITE <WRITE$OBJECT> [, <WRITE$OBJECT>]*
      Ex:   WRITE 'CODING POLE NUMBER', .I, ' NOW'
<WRITE$OBJECT>:
   <STRING>   <EXPR>
```

```
YSCALE - A command to display or set the vertical graphics scale:
   YSCALE
      Display the current YSCALE setting.  If the setting is AUTO,
      the current minimum and maximum are also displayed.
   YSCALE = AUTO
      Specify that the vertical scale is to adjust to fit the
      minimum and maximum of the curve being graphed.
   YSCALE = <EXPR> , <EXPR>
      Specify a fixed vertical scale.
         Ex:   YSCALE = -PI,PI

*
*EXIT
```

APPENDIX B
KEYWORDS: DEFINITIONS AND DEFAULTS

Constants, Operators, Functions

ABS	Operator, used in expressions, gives absolute value of the argument appearing to its right, e.g., ABS (Y), ABS (X-Y)
ACOs	Function used in expressions, giving the arc cosine of the argument
AND	Operator, used in logical expressions, gives bitwise conjunction of the argument appearing to its right with the argument to its left; both must be boolean expressions or integer expressions
ANGle	Function used with a Z-plane pole or zero argument; returns the angle of the object; e.g., ANG (POL 2)
ASIn	Function used in expressions, giving the arc sine of the argument
ATAn	Function used in expressions, giving the arc tangent of the argument
COS	Function used in arithmetic expressions, gives cosine of the argument appearing to its right
EXP	Function used in arithmetic expressions, gives powers of e (=2.718281)
HPI	Constant, value 3.1415926/2 = 1.57079633
IMAg	Function used with a S-plane pole or zero argument; returns the imaginary part of the object; e.g., IMA (ZER 12)
LOG	Function used in arithmetic expressions, gives natural log to the base e (2.718728)
MASk	Operator, used in arithmetic expressions, gives bitwise conjunction of the argument appearing to its right with the argument to its left; unrestricted e.g. PI MASK 0.FFFFH = .14159, PI MASK 2 = 2.00
MOD	Function used in arithmetic expressions, gives the remainder from dividing the argument to its left with the argument appearing to its right
NOT	Operator, used in logical expressions, gives the negation of the argument appearing to its right
OR	Operator, used in logical expressions, gives the inclusive or (disjunction) of the argument appearing to its right with the argument to its left. Each must be a boolean or an integer expression
PI	Constant, value 3.141592653
RADius	Function used with a Z-plane pole or zero argument; returns the magnitude of the object; e.g., RAD (POL 217)
REAl	Function used with a S-plane pole or zero argument; returns the real part of the object; e.g., REAL (ZER 6)
SIN	Function used in arithmetic expressions, gives sine of the argument appearing to its right
SQR	Function used in arithmetic expressions, gives square root of the argument appearing to its right
TAN	Function used in arithmetic expressions, gives tangent of argument appearing to its right

TPI	Constant, value 3.14159265*2 = 6.2831852
XOR	Operator used in logical expressions, gives exclusive or (disjunction) of the argument appearing to its right with the argument to its left. Each must be a boolean or an integer expression

Commands

APPend	File command, writes out the specified (or default) objects to the specified file, either creating a new file or adding to the end of an existing file (Chapter 7)
CODe	Creates AS2920 assembly language code for the current poles and zeros or for equations; also abbreviated C, CO
COUnt	Compound command, establishes maximum number of times command block is to be executed
DEFine	For symbols, creates an entry in a table and attaches a numeric value to it; for poles or zeros, the value is the coordinates and plane of that object; for macros, it is a pointer to the macro's block of commands
DIR	Used only in DIR MACRO command; lists all the names of macros currently available
DISplay	File command; copies the named file to the console
EVAluate	Gives the decimal numeric value of the argument appearing to its right
EXIt	Terminates the current SPAC20 Compiler session
GRAph	Entered alone, this displays the last curve plotted; if a filter response is supplied as an object, e.g. GRAPH PHASE, this displays the graph of the values of the object specified, using the latest appropriate scales; also abbreviated GR If the object is LBO or UBO, the lower or upper bounds are graphed
HELp	Types explanatory message about the argument appearing to its right; if the item is *, types all such messages
HOLd	Command to correct attenuation due to sample-and-hold distortion: if ON, corrects AGAIN by multiplying by | sin(X)/X |, where X = TS*FREQ*PI, and corrects PHASE by adding X, and GROUP by subtracting PI*TS; also abbreviated H, HO
IF	Compound command, often used in macros, establishes conditional flow of control within a command block
INClude	File command, executes contents of specified file as if typed as commands at the console
LISt	File command, establishes file copy of all console interactions
MOVe	Command to change location or plane (or both) for one or more poles or zeros by specifying an increment or final value for each coordinate; also abbreviated M, MO
OGRaph	Displays the graph of the values for the filter response entered as its argument, simultaneously superimposing the last curve plotted; also abbreviated OG If the object is LBO or UBO, the lower or upper bounds are graphed

PUT	File command, writes out the specified (or default) objects to the specified file, either creating a new file or writing over an existing file of the same name (Chapter 8)
REMove	Deletes from a table one, several, or all entries: poles, zeros, macros, symbols (Chapters 2, 3, 9)
REPeat	Compound command, establishes unlimited repetition of commands block (subject to optional WHILE or UNTIL clauses) (Chapter 9)
WRIte	File command, puts out one line to the LIST file and console; usually used in compound commands

Objects

AGAin	Absolute gain, expressed as a multiplier, due to all existing poles and zeros; also used as a function with an expression (as its argument), giving the absolute gain at that (expression's) frequency; also abbreviated AG
BOUnds	Represents the piecewise linear bounds, in PUT or APPEND commands, or for display of LBO and UBO
ERRor	Absolute error in multiplier from last CODE command
FSCale	Frequency scale for computing and graphing filter responses, initially 10, 10000; establishes the range for the specific points (up to 69) of evaluation
GAIn	Gain in decibels due to all existing poles and zeros, normalized by the current setting of GREF; also used as a function with an expression (as its argument), giving the gain at that (expression's) frequency; also abbreviated G, GA
GERror	Deviation of the gain response from the bounds; also used as a function with an expression (as its argument), giving the gain error at that (expression's) frequency
GREf	Reference gain, expression AT expression, meaning a gain of expression1 AT frequency expression2
GROup	Group delay of the filter (= the negative of derivative of the phase with respect to the frequency); also used as a function with an expression (as its argument), giving the group response at that (expression's) frequency
IMPulse	Filter output in reaction to a unit up-impulse at time zero (i.e. an instantaneous jump from 0 to 1 and return to zero)
INSt	Number of AS2920 assembly language instructions created by latest CODE command
LBOund	The lower of the bounds on gain, defined as piece-wise linear regions; intially −1000000 AT 1; also abbreviated LB
MACro	Entered alone, an object keyword to display all macro command blocks; when one or more macro names follow it, only the named macros command blocks are displayed; this word can also be a modifier keyword to qualify the effect of DEFINe or REMOVE, and it appears as a necessary part of the DIR MACRO command.
MAGain	Object keyword giving the maximum absolute gain taken over the frequencies in FSCALE
MERror	Maximum absolute error in gain relative to the bounds, considered over the frequencies in FSCALE

MSQe	Mean square error in gain as compared to bounds, considered over the frequencies in FSCALE
PERror	Object keyword giving the allowable change in coordinates of the pole or zero to be CODEd; used only in CODE command, to specify a limit (constraint) on this movement, as in CODE POLE 12 PERROR < 3,4
PHAse	Object keyword giving the phase delay response of the filter; also used as a function with an expression (as its argument), giving the phase delay response at that (expression's) frequency; also abbreviated PH
POLe	Used to display the pole whose number-label is the argument appearing to its right; also used as modifier to DEFINE or REMOVE to add or delete POLE entries (one, several, or all) in the table of poles and zeros; also abbreviated P, PO
PZ	Designates the entire set of poles and zeros, for display or as object to REMOVE, PUT, or APPEND
STEp	Filter output in reaction to a unit up-step at time zero (i.e. an instantaneous jump from 0 to 1)
SYMbol	Designates entire set of numeric-valued user-symbols in that symbol table, for display or as modifier to REMOVE, PUT or APPEND
TS	Sample interval for sampled S-plane
UBOund	The upper of the bounds on gain, defined as piece-wise linear regions; initially 1000000 AT 1; also abbreviated UB
XSIze	Number of vertical columns defining entire graphics screen area, up to 79; i.e., maximum number of characters displayable per horizontal line; the area for curves being plotted is 10 less to allow for labeling the axis
YSCale	Specific range for vertical scale on graphs, by giving minimum and maximum values; if AUTO is specified, the min and max values of the curve being plotted are used
YSIze	Number of horizontal rows defining graphics screen area, up to 25; i.e., maximum number of characters displayable per vertical column is 25; the area for curves being plotted is 3 less to allow for labels
ZERo	Used to display the zero whose number-label is the argument appearing to its right; also used as modifier to DEFINE or REMOVE to add or delete ZERO entries (one, several, or all) in the table of poles and zeros; also abbreviated Z, ZE

Modifiers

AT	Used in setting GREF, LBOUND, and UBOUND to specify frequencies, e.g. AT 0 meaning DC, or AT 249 meaning Hertz
AUTo	Used in setting YSCALE, indicating full screen vertical scale for the actual range of the object being graphed
BY	In MOVE commands, tells the increments (in a coordinate pair) for the movement of one or more poles or zeros in the original plane of definition
CONtinuous	Designates continuous S-plane for pole or zero definition or movement

ELSe	Identifies that block of commands (in an IF statement) which is to be executed if all earlier if-expressions proved FALSE
EM	Required end-statement for a macro definition
END	Required to end compound commands, i.e. REPEAT, COUNT, IF
OFF	Indicates there is to be no correction for sample-and-hold distortion; see HOLD
ON	Turns on correction for sample-and-hold distortion; see HOLD
ORIf	Identifies an alternate test expression and block of commands in an IF statement
THEn	Optional entry after the first test expression of an IF statement (and before the first block of commands)
THRough	Identifies the range of a partition, as in POLE 1 THROUGH 9
TO	In MOVE, tells the desired location of the object(s) being MOVEd, in(to) any plane
UNTil	A loop-exit in a compound command, causing execution to skip all commands between it and the next END statement when the expression after the UNTIL is TRUE.
WHILe	A loop-exit; when the expression following is FALSE, execution skips to the next END statement in the compound command
Z	Designates sampled Z-plane for pole or zero definition or movement

List of all Keywords

ABS	ERROR	MACRO	REMOVE
ACOS	EVALUATE	MAGAIN	REPEAT
AGAIN	EXIT	MASK	SIN
AND	EXP	MERROR	SQR
ANGLE	FSCALE	MOD	STEP
APPEND	GAIN	MOVE	SYMBOL
ASIN	GERROR	MSQE	TAN
AT	GRAPH	NOT	THEN
ATAN	GREF	OFF	THROUGH
AUTO	GROUP	OGRAPH	TO
BOUNDS	HELP	ON	TPI
BY	HOLD	OR	TS
CODE	HPI	ORIF	UBOUND
COS	IF	PERROR	UNTIL
COUNT	IMAG	PHASE	WHILE
DEFINE	IMPULSE	PI	WRITE
DIR	INCLUDE	POLE	XOR
DISPLAY	INST	PUT	XSIZE
ELSE	LBOUND	PZ	YSCALE
EM	LIST	RADIUS	YSIZE
END	LOG	REAL	Z
			ZERO

APPENDIX C
NOTES AND CAUTIONS

While the following do not seriously affect the usability of the SPAC20 Compiler, they should be noted as areas for possible macro development by those users who find them inconvenient:

There is no direct command for specifying an S-plane to Z-plane transform different from the matched-Z transform. Other transforms, if desired, must be implemented via macros. (See, for example, Chapter 10.)

The SPAC20 Compiler produces IIR (infinite impulse response) digital filters. There is no facility to interactively design FIR (finite impulse response) filters.

Step and impulse time responses are available but only over 64 or so equal time intervals, starting at zero. Ideally, the time response to a larger variety of inputs, over a larger variety of time scales, should be available. The computational complexity involved inhibits this for the present.

In any digital filter implementation, anomalous behavior will occur when the input signal is small compared to the digital precision. Analysis of this dead band (i.e. region of signal amplitude causing misbehavior) and limit cycles (i.e. self-sustaining low amplitude oscillation) is in general difficult and is not undertaken in this product.

Some calculations performed by the Compiler may press or exceed the limits of its floating point package. One such limit is the 24 bit precision of the numbers. If a pole and zero are superimposed, for example, the gain is almost, but not exactly, zero. Graphing this gain with YSCALE = AUTO can yield unexpected curves. Another limit is the 7 bit exponent. If many more poles than zeroes are defined, for example, underflow or overflow may occur and may distort the expected filter response. Alternately defining poles and then zeroes may ameliorate this problem.

Because of the interactive nature of the SPAC20 Compiler, and because of the extensive floating point calculations, a high-speed math-board (iSBC-310) is highly recommended.

APPENDIX D
BNF SYNTAX SUMMARY

This appendix summarizes the syntax for the SPAC20 Compiler using Backus Naur Form (BNF). The vertical bar means a choice among alternatives. Asterisk means the optional item (in square brackets) may be repeated any number of times.

Command Summary

<top-level comnd> ::= <define macro comnd>

| <remove macro comnd>
| <comnd>

<comnd> ::= <compound comnd> | <simple comnd>

<compound comnd> ::= <if comnd>

| <repeat comnd>

| <count comnd>

<simple comnd> ::= <display pole/zero comnd>

| <define pole/zero comnd>

| <move pole/zero comnd>

| <remove pole/zero comnd>

| <list filter response comnd>

| <graph filter response comnd>

| <display gref comnd>

|<change gref comnd>

| <display bounds comnd>

| <set bounds comnd>

| <display filter response function comnd>

| <display scale comnd>

| <set scale comnd>

| <display code comnd>

| <code comnd>

| <display comnd>

| <change comnd>

| <define symbol comnd>

| <display symbols comnd>

| <remove symbols comnd>

| <evaluate comnd>

| <display file comnd>

| <put file comnd>

| <append file comnd>

| <include comnd>

| <list comnd>

| <write comnd>

| <help comnd>

| <exit comnd>

| <macro invocation comnd>
| <display macro comnd>
| <dir macro comnd>

Expressions

<exp> ::= <boolean term> [<or op> <boolean term>]*
<or op> ::= OR | XOR
<boolean term> ::= <boolean factor> [AND <boolean factor>]*
<boolean factor> ::= [NOT] <boolean primary>
<boolean primary> ::= <arith exp> [<rel op> <arith exp>]
<rel op> ::= < | > | <= | >= | = | <>
<arith exp> ::= <arith term> [MASK <arith term>]*
<arith term> ::= <term> [<plus op> <term>]*
<plus op> ::= + | -
<term> ::= <factor> [<mult op> <factor>]*
<mult op> ::= * | / | MOD
<factor> ::= [<plus op>] <secondary>
<secondary> ::= <primary> [** <primary>]
<primary> ::= (<exp>)
| <function> (<exp>)
| PI | TPI | HPI
| <numeric constant>
| <symbolic ref>
| <keyword reference>
| <filter response function>
| <pole/zero reference>
<function> ::=SIN | COS | EXP | LOG | SQR | ABS | TAN | ASIN | ACOS | ATAN
<numeric constant> ::= <digit> [<digit>]* [<radix>]
| [<digit>]* . <digit> [<digit>]* [<radix>]
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| A | B | C | D | E | F
<radix> ::= B | D | H
<symbolic ref> ::= <symbol>
<symbol> ::= . <identifier>
<partition> ::= <arith exp> [THROUGH <arith exp>]

Keyword References

<keyword reference> ::= TS | XSIZE | YSIZE | MAGAIN | MSQE | MERROR | INST
| ERROR

Filter Design Commands

<display pole/zero comnd> ::= <poles and zeroes>
<poles and zeroes> ::= <pole/zero> [<partition>]
| PZ
<pole/zero> ::= POLE | ZERO

<define pole/zero comnd> ::= DEFINE <pole/zero> <arith exp> = <exp> , <exp> [, <plane>]

<plane> = CONTINUOUS | TS | Z

<move pole/zero comnd> ::= MOVE <poles and zeroes> <movement>

<movement> ::= TO <exp> , <exp> [, <plane>]

| TO <plane>

| BY <exp> , <exp>

<remove pole/zero comnd> ::= REMOVE <poles and zeroes>

<pole/zero reference> ::= REAL (<pole/zero> <arith exp>) | IMAG (<pole/zero> <arith exp>) | RADIUS (<pole/zero> <arith exp>) | ANGLE (<pole/zero> <arith exp>)

<list filter response comnd> ::= <filter response>

<graph filter response comnd> ::= <graph/ograph> <filter response>

| GRAPH

<graph/ograph> ::= GRAPH | OGRAPH

<graph/ograph bounds comnd> ::= <graph/ograph> LBOUND | <graph/ograph> UBOUND

<filter response> ::= GAIN

| AGAIN

| GERROR

| PHASE

| GROUP

| STEP

| IMPULSE

<display gref command> ::= GREF

<change gref command> ::= GREF = <exp> AT <exp>

<display hold comnd> ::= HOLD

<change hold comnd> ::= HOLD ON | HOLD OFF

<display bounds comnd> ::= LBOUND | UBOUND | BOUNDS

<set bounds comnd> ::= LBOUND = <piecewise linear bound>

| UBOUND = <piecewise linear bound>

<piecewise linear bound> ::= <bound> [, <piecewise linear bound>]

| <bound> , , <piecewise linear bound>

<bound> ::= <exp> AT <exp>

<display filter response function comnd> ::= <filter response function>

<filter response function> ::= GAIN (<exp>)

| AGAIN (<exp>)

| GERROR (<exp>)

| PHASE (<exp>)

| GROUP (<exp>)

<display scale comnd> ::= FSCALE | YSCALE

<set scale comnd> ::= FSCALE = <exp> , <exp> [, <exp>]*

| YSCALE = <yscale setting>

<yscale setting> ::= <exp> , <exp> | AUTO

<display code comnd> ::= CODE

<code comnd> ::= CODE <pole/zero> <arith exp> <pz constraint>

| CODE <multiplication> <multiplication constraint>

<multiplication> ::= <y identifier> = <primary> * <x identifier> [+ <y identifier>]

<x identifier> ::= <identifier>

<y identifier> ::= <identifier>

<pz contraint> ::= [INST < <exp>] | MSQE < <exp> [, INST < <exp>]

| MERROR < <exp> [, INST < <exp>]

| PERROR < <exp>, <exp> [, INST < <exp>]

<multiplication constraint> ::= [INST < <exp>]

| ERROR < <exp> [, INST < <exp>]

Interrogation and Utility Commands

<display comnd> ::= <keyword reference>

| <symbolic reference>

<change comnd> ::= <keyword reference> = <exp>

| <symbolic reference> = <exp>

<define symbol comnd> ::= DEFINE <symbol> = <exp>

<display symbols comnd> ::= SYMBOL

<remove symbols comnd> ::= REMOVE <symbolic ref list>

| REMOVE SYMBOL

<symbolic ref list> ::= <symbolic ref> [, <symbolic ref>]*

<evaluate comnd> ::= EVALUATE <exp>

<display file comnd> ::= DISPLAY <path name>

<put file comnd> ::= PUT <path name> [<file object>] [,<file object>]*

<append file comnd> ::= APPEND <path name> [<file object>] [,<file object>]*

<file object> ::= PZ | BOUNDS | SYMBOLS | MACROS | CODE

| <strings and exps>

<strings and exps> ::= <string or expression> [, <string or expression>]*

<string or expression> ::= <string> | <exp>

<include comnd> ::= INCLUDE <path name>

<list comnd> ::= LIST <path name>

<write comnd> ::= WRITE <strings and exps>

<help comnd> ::= HELP [<help request>]

<help request> ::= <help item>

| *

<help item> ::= <identifier>

<exit comnd> ::= EXIT

Compound Commands and Macros

<if comnd> ::= IF <exp> [THEN] <cr> <true list> [ORIF <exp> <cr> <true list>]* [ELSE <cr> <false list>] END

<true list> ::= [<command> <cr>]*

<false list> ::= [<command> <cr>]*

<cr> ::= carriage-return | line-feed

<repeat comnd> ::= REPEAT <cr> <loop list> END

<count comnd> ::= COUNT <exp> <cr> <loop list> END

<loop list> ::= [<loop element> <cr>]*

<loop element> ::= <command> | <loop exit>

<loop exit> ::= WHILE <exp> | UNTIL <exp>

<define macro comnd> ::= DEFINE MACRO <macro name> <cr> <macro body> EM

<macro name> ::= <identifier>

<macro body> ::= [<command> <cr<]*

<macro invocation comnd> ::= :: <macro name> [<actual parameter list>]

<actual parameter list> ::= <actual parameter> [, <actual parameter>]*

<actual parameter> ::= <limited token> * | <string>

<limited token> ::= any token except <cr>, <string>, or ","

<remove macro comnd> ::= REMOVE MACRO [<macro list>]

<macro list> ::= <macro name> [, <macro name>]*

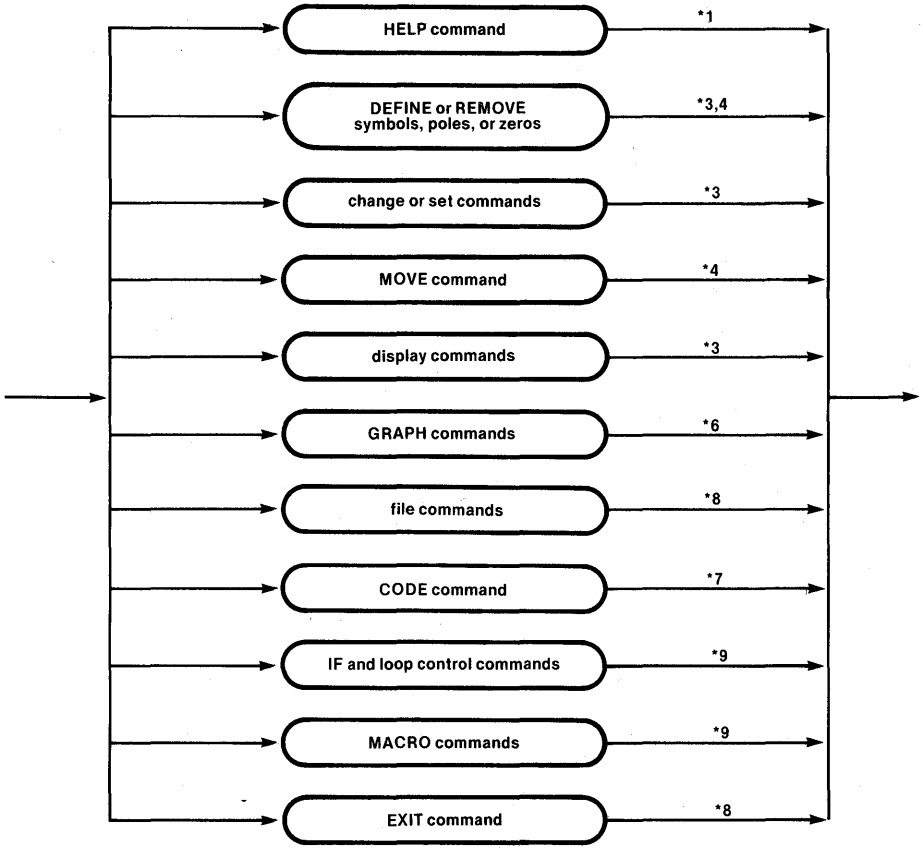
<display macro comnd> ::= MACRO [<macro list>]

<dir macro comnd> ::= DIR MACRO

APPENDIX E
SYNTAX CHARTS

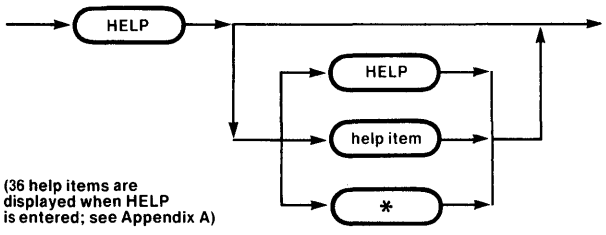
Table of Sample Commands to Define, Display, or Remove Objects from Compiler Tables

	Defining an Object into a Table	Displaying Part or All of a Table	Removing an Object from a Table
One Symbol	DEFINE .NAME_1 = 3 DEFINE .NAME_2 = .NAME_1 + 1 DEF .NAME_3=.NAME_1*.NAME_2	.NAME_1 .NAME_2	REMOVE .NAME_1 REMOVE .NAME_2
One pole or zeros	DEF POLE 2 = -5, 450, TS DEF ZERO 2 = 1/2, 100, Z	POLE 2 ZERO 2	REMOVE POLE 2 REM ZERO 2
One Macro	DEFINE MACRO JP xoxoxoxox xoxox EM	MACRO JP	REM MACRO JP
Several Symbols	Requires multiple commands	Requires multiple commands	REM .NAME_1,.NAME_2, .NAME_3
Several poles or zeros	Requires multiple commands	POLE 1 THROUGH 5	REM POLE 1 THROUGH 3
Several Macros	Requires multiple commands	MACROS JP, RQ, TEN	REM MAC RQ, TEN, FEEDER
ALL Symbols	———	SYMBOLS	REMOVE SYMBOLS
ALL poles	———	POL	REM P
ALL zeros	———	ZER	REM Z
ALL poles and zeros	———	PZ	REMOVE PZ
All Macro Names	———	DIR MACROS	———
ALL Macros	———	MACROS	REMOVE MACROS

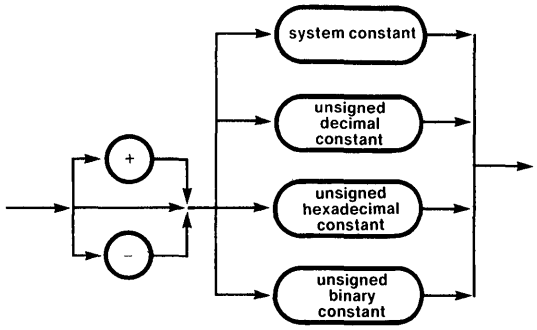


* chapter where discussed

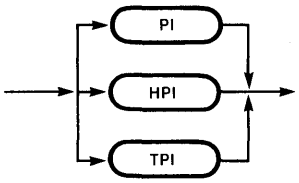
Commands 121533-01



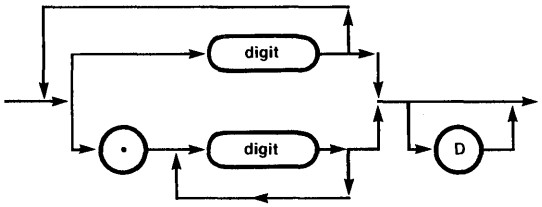
HELP Command 121533-27



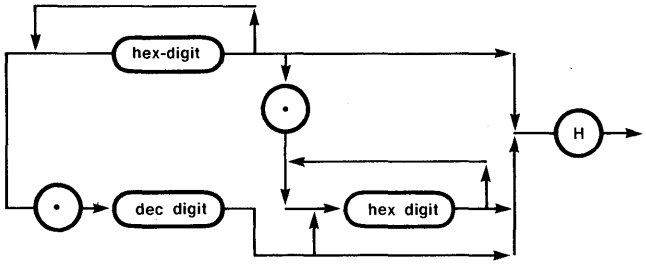
Numeric Constant



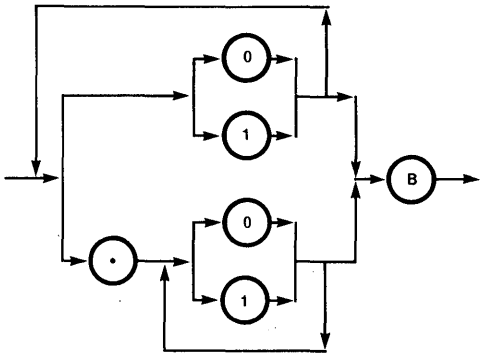
System Constant



Unsigned Decimal Constant

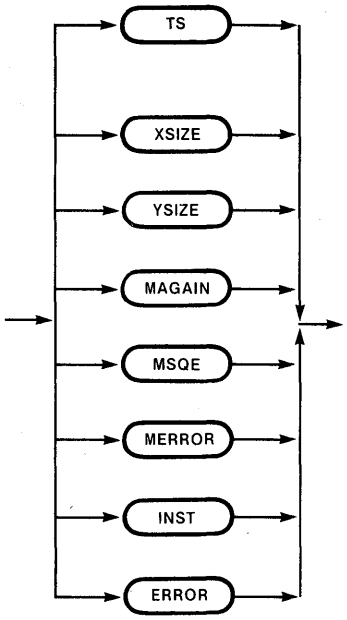


Unsigned Hexadecimal Constant

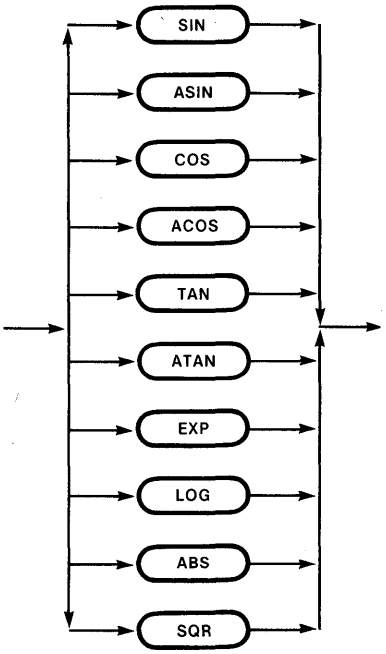


Unsigned Binary Constant

121533-06

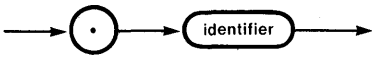


Keyword References

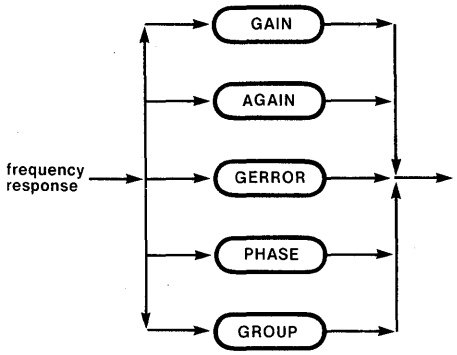


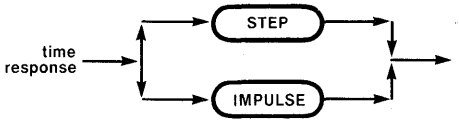
EXPonentiation and LOGarithms
to the base e = 2.718281

Functions 121533-28

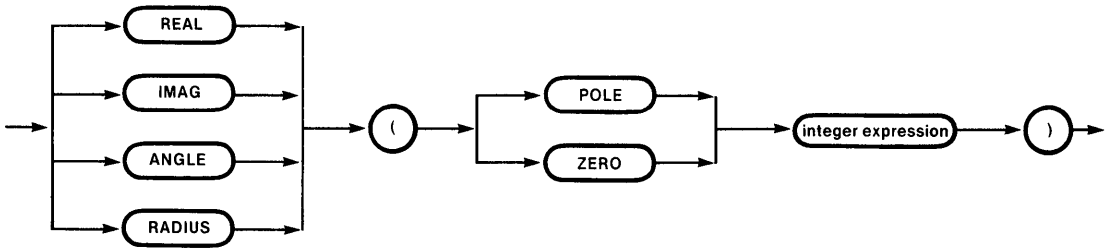


Symbolic References 121533-08

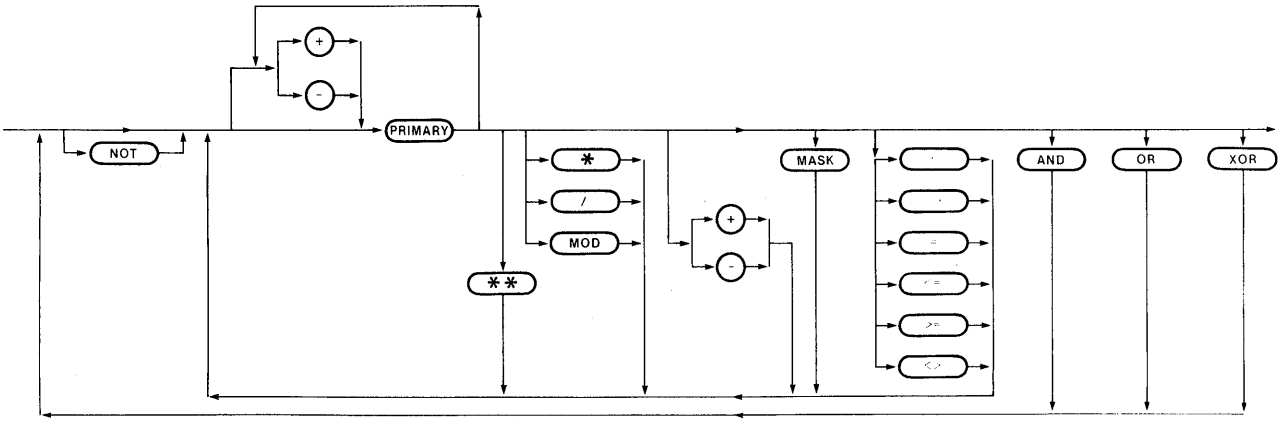




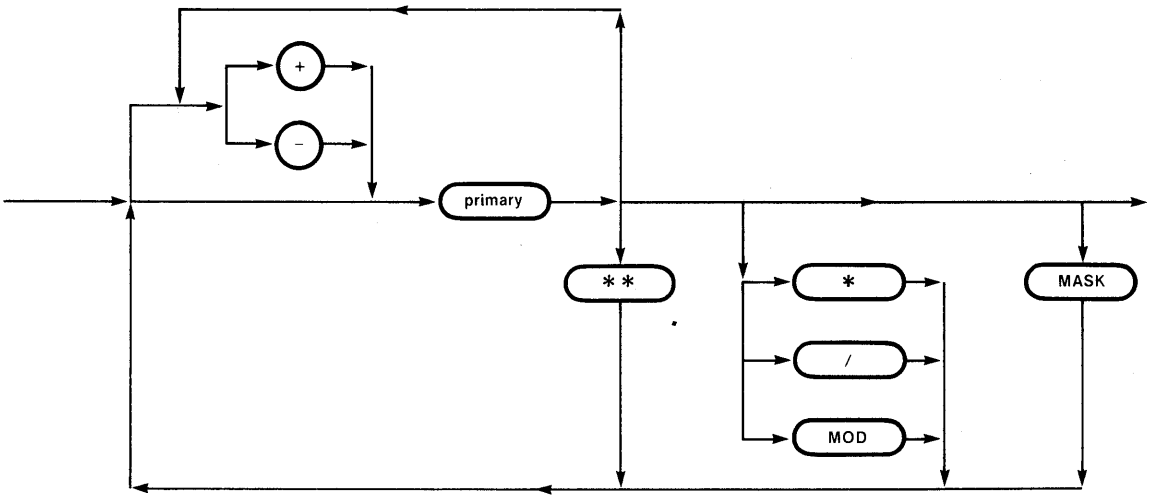
Frequency Responses, Time Responses 121533-09



Coordinate (Pole/Zero Expression) 121533-43

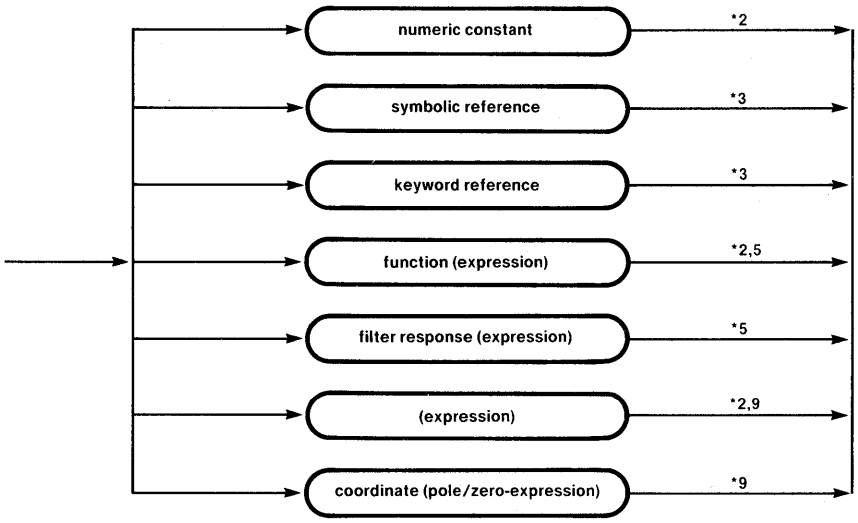


Expression 121533-26



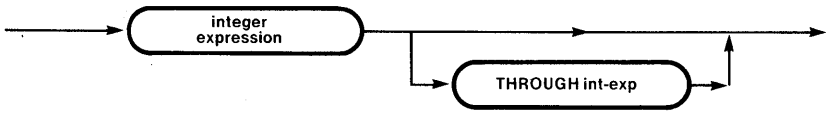
An integer-expression is an arithmetic-expression which evaluates to an integer.

Arithmetic Expression 121533-07

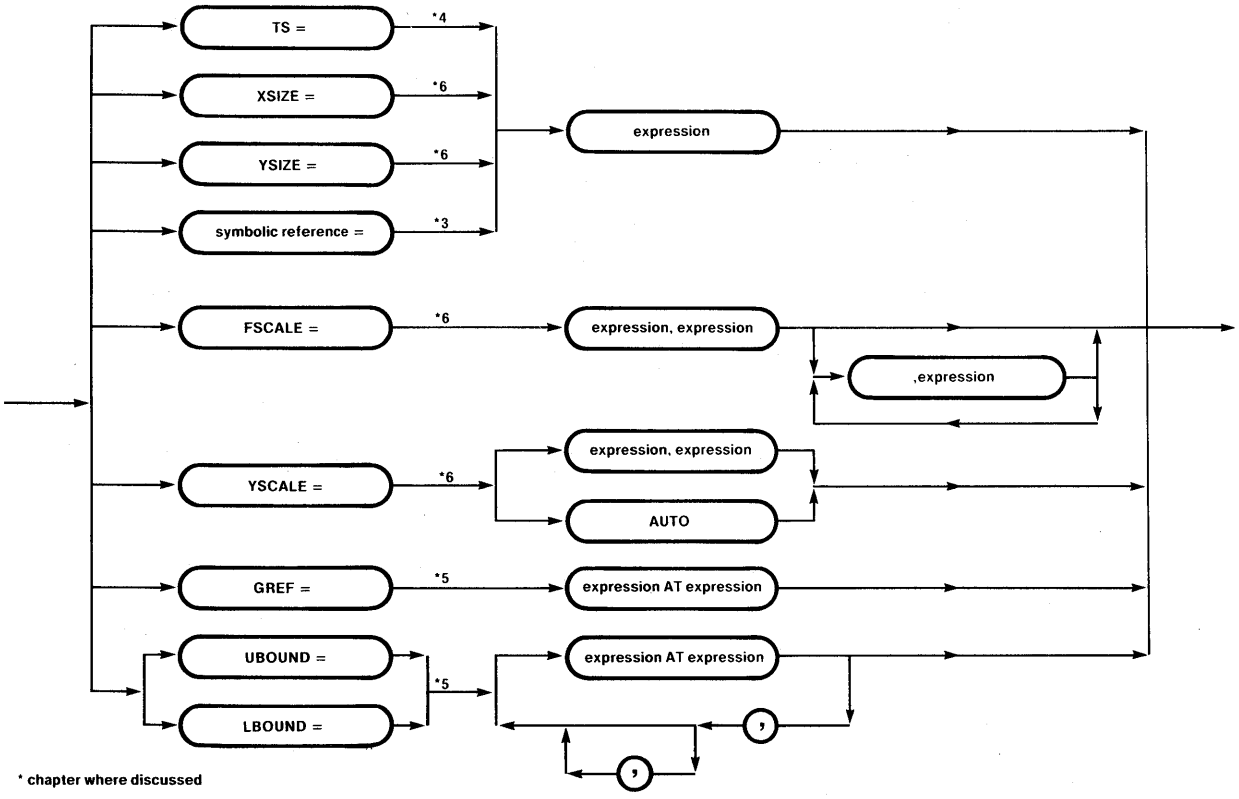


* chapter where discussed

Primaries 121533-04

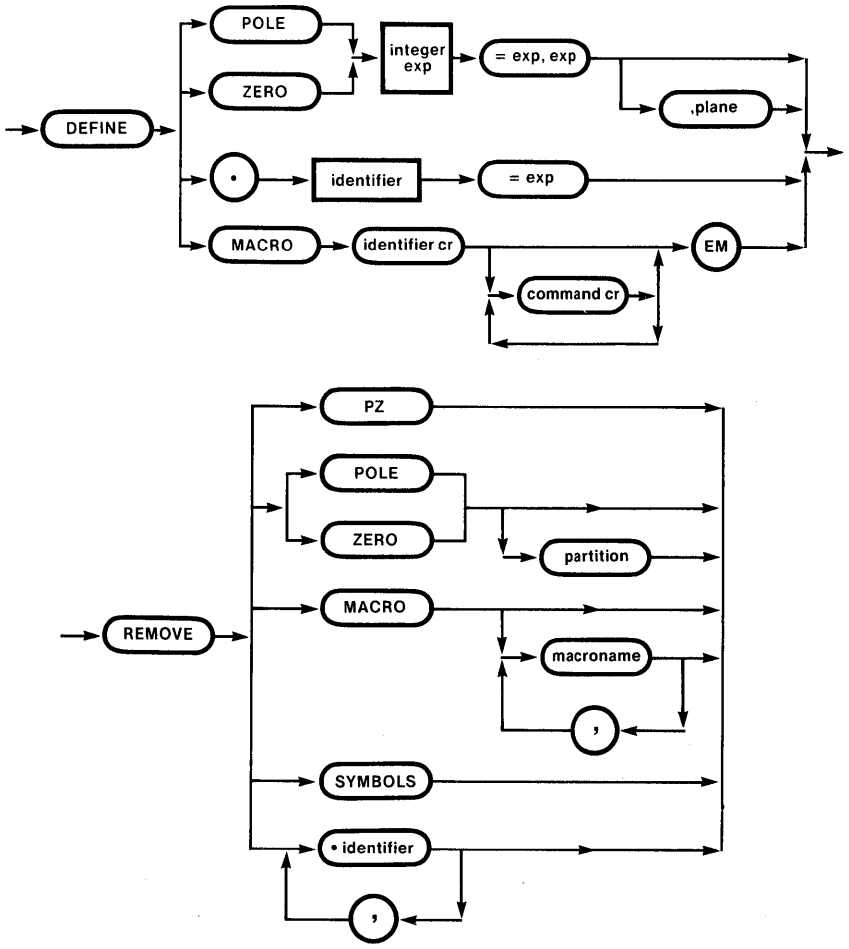


Partition 121533-05

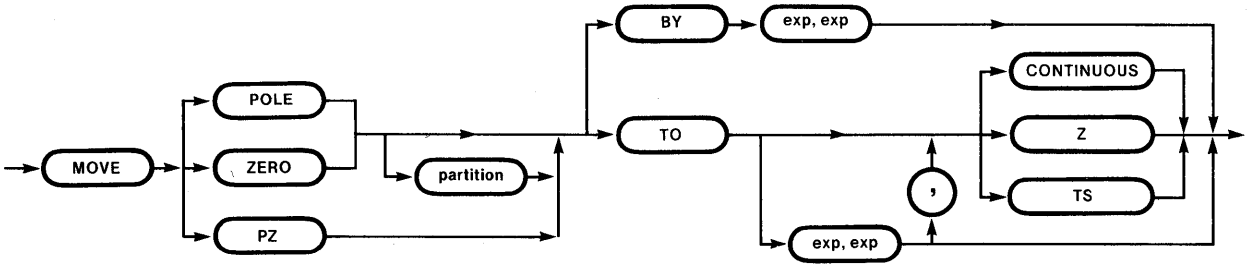


* chapter where discussed

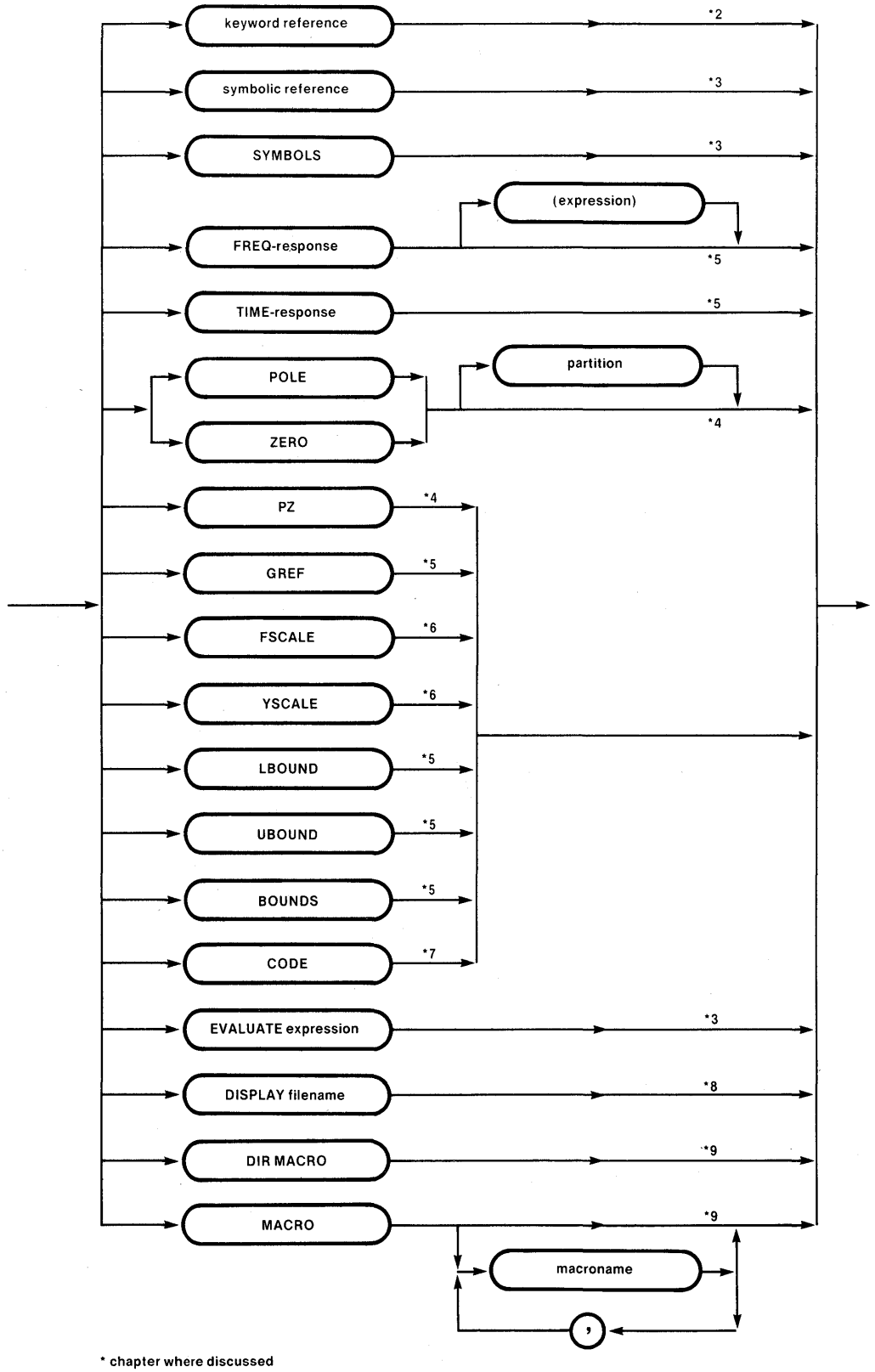
The Change Commands 121533-02



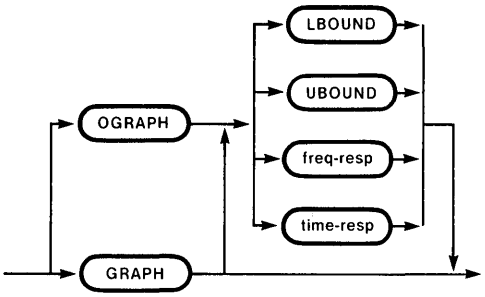
Full DEFINE and REMOVE 121533-10



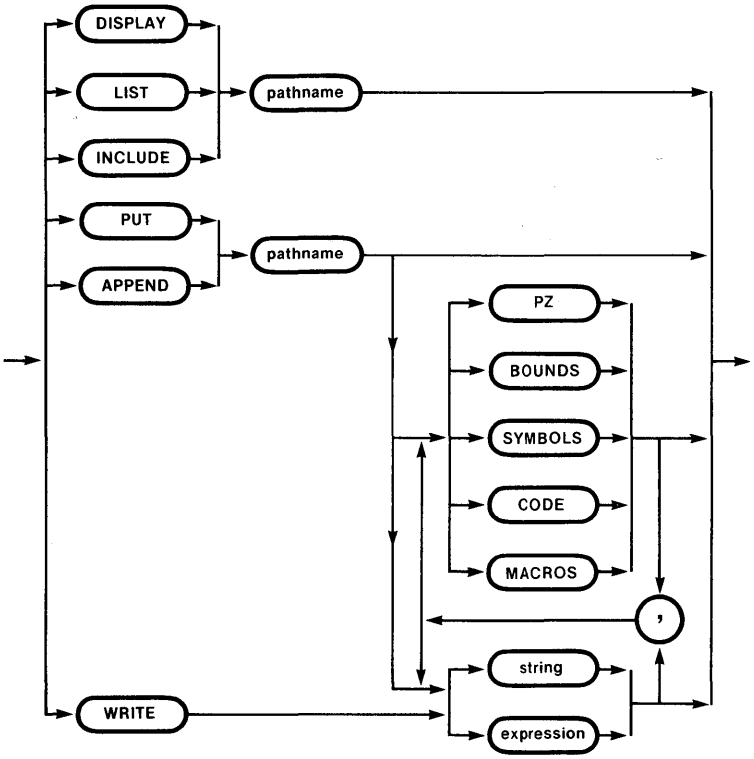
MOVE Command 121533-15



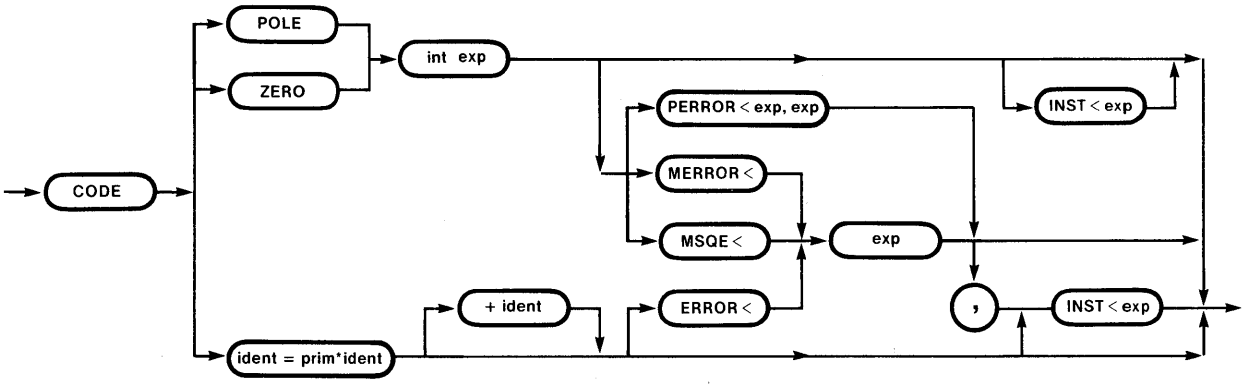
Display Commands 121533-03



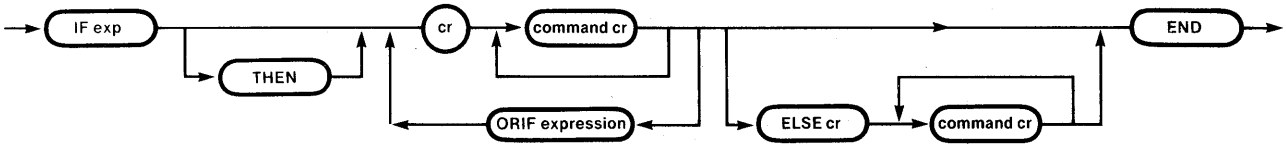
GRAPH/OGRAPH 121533-25



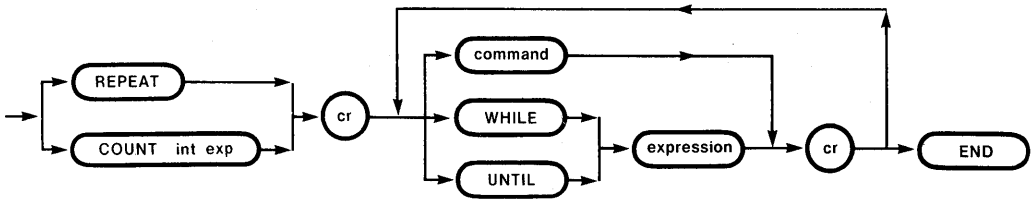
File Commands 121533-24



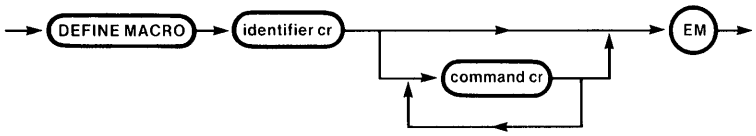
CODE Command 121533-16



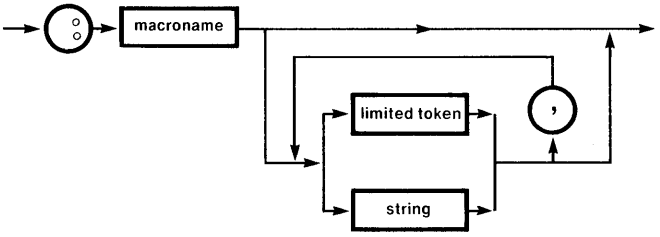
IF Command 121533-22



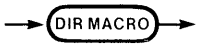
REPEAT/COUNT 121533-23



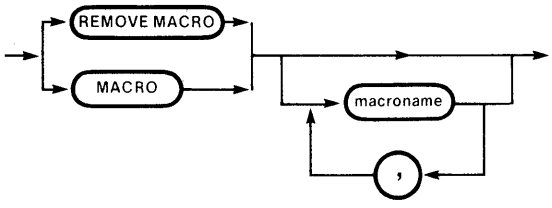
DEFINE Command for Macros 121533-19



Invoking Macros 121533-20



Macro Directory 121533-21



macroname := an identifier appearing as above in a legal define-macro command

Remove or Display Macros 121533-21

APPENDIX F
SOFTWARE INSTALLATION PROCEDURE

Software Installation Procedure

The two modules of system software that support interactive development of signal processing programs for the Intel 2920 chip are the Intel Systems Implementation Supervisor (ISIS-II) and the SPAC20 software module. If the iSBC 310 math board is to be used, it should be installed as if to run FORTAN. (See the manuals listed in the Preface.)

SPAC20 Compiler Files

The SPAC20 Compiler consists of ten files, SPAC20.SFT, SPAC20.HRD, SPAC20.OVH, SPAC20.OV0 through SPAC20.OV5, and SPAC20.OVE. The first two represent different versions of the Compiler: SPAC20.HRD provides faster computations by taking advantage of the iSBC-310 math board, which must be present if this version is used. SPAC20.SFT performs the math functions in software. One of these two files should be renamed to SPAC20, using ISIS-II, e.g.

```
RENAME :F1:SPAC20.SFT to :F1:SPAC20
```

The file SPAC20.OVH contains the help messages, which allows the Compiler to interactively give information about individual command syntax and options interactively, complementing the data in this manual. The help messages are described below and are reproduced in full in Appendix A. This file should be accessed only by the Compiler.

The file SPAC20.OVE contains the error messages. SPAC20.OV0 through SPAC20.OV5 are overlays called by SPAC20 (SFT or HRD). There is also a macro file on the disk, SPAC20.MAC, containing the example macros shown in Chapters 10 and 11. If you wish to copy all the files to a backup diskette, you can use the term SPAC20.* (see ISIS manual). Specific macros can be edited into separate files for later INCLUDE commands.

The Compiler can be invoked by typing the drive and file name after the ISIS prompt. For example:

```
-:F1:SPAC20
```

The Compiler will sign on with the message:

```
ISIS-II 2920 Signal Processing Applications Compiler, V1.0
*
```

You enter your commands to the Compiler one per line, and terminate each with a carriage-return. You may include comments by preceding them with a semicolon. Commands may be continued by typing an ampersand prior to both the carriage-return and the comment field (if any) of the line to be continued. Characters between the ampersand and the line terminator (carriage-return or line-feed) are ignored, and the ampersand is treated as a space. Each input line may contain no more than 120 characters before the line terminator.

The ESCape key cancels the current activity and returns control of the Compiler to you. This applies to partially typed commands or commands in progress, including a macro or compound command.

All command keywords can be abbreviated to their first three characters, e.g., APP for append, or DEF for define. Many frequently used command keywords have single character abbreviations, such as P for pole, or Z for zero. Appendix B shows all keywords and their legal abbreviations.

The EXIT command returns control to ISIS, ending the present use of the SPAC20 module:

```
*EXIT
```

ISIS-II and SPAC20 Diskette

The Intellec system uses the diskette hardware and ISIS to provide a powerful, convenient microcomputer development tool. ISIS interfaces to the diskette hardware and to any other standard peripheral device. You communicate with ISIS-II by entering commands on the system console or by embedding system calls in programs that will run in an ISIS environment. ISIS enables rapid storage and retrieval of files on diskette.

Diskette files containing SPAC20 parameters or 2920 assembly language source code can be read or written during a SPAC20 design session. The main purpose of ISIS during setup and interactive development is to provide the I/O interface to the console and files on diskette. To begin a SPAC20 session, you must first boot ISIS and then load the SPAC20 software module. The process for booting ISIS is fully described in the *ISIS-II System User's Guide* and summarized below.

Loading ISIS and the SPAC20 Software Module

After installing all hardware and turning on power to the console, disk drives, and Intellec Series I, the following steps will load ISIS and SPAC20 (the full procedure and notes are in Chapter 2 of the *ISIS-II User's Guide*:

a. Place system diskette in drive 0 and close drive door.

b. Press top half of boot switch on Intellec front panel.

c. Press top half of reset switch on that panel.

d. After interrupt light 2 goes on, press space bar.

e. After the light goes off, press bottom half of boot switch.

f. After receiving the ISIS sign-on message and prompt

```
—ISIS-II Vn.n
```

you type:

```
—SPAC20
```

If the diskette containing the SPAC20 module is not a system diskette, mount it in drive 1 instead of drive 0 and boot ISIS from a separate system diskette in drive 0. Then, at this point, type :F1:SPAC20 (to load the SPAC20 module from drive 1) followed by a carriage return. The SPAC20 module will sign on and you are ready to continue.

If ISIS fails to sign on, recheck that all boards and cables are correctly installed and firmly seated, then perform the above procedure again.

If you have an Intellec Series II, there are slight variations in the above procedure. This is what you must do:

a. Turn on power to Intellec Series II and to disk drives. Press square white on/off button in the lower right-hand corner of front panel. It is a two-position switch; it lights up if you have pressed it correctly.

b. Note that this message appears:

```
SERIES II, MONITOR, Vn.n
```

c. Place system diskette in drive 0 and close door.

d. Press reset button that is to left of the on/off button. Note that drive light is on to indicate that information is being accessed.

e. After receiving the ISIS sign-on message and prompt

```
-ISIS-II Vn.n
```

you type:

```
-SPAC20
```

If the diskette containing the SPAC20 module is not a system diskette, mount it in drive 1 instead of drive 0 and boot ISIS from a separate system diskette in drive 0. Then, at this point, type :F1:SPAC20 (to load the SPAC20 module from drive 1) followed by a carriage return. The SPAC20 module will sign on and you are ready to continue.

If ISIS fails to sign on, recheck that all boards and cables are correctly installed and firmly seated, then perform the above procedure again.

APPENDIX G
CODE SUBMISSION TO THE AS2920 ASSEMBLER

Since the Assembler uses the ISIS-II keyboard and file capabilities, ISIS-II must be loaded before invoking the Assembler. The full procedure for this is given in the ISIS manual named in the Preface. Once ISIS-II is present, you can enter the Editor to key-in or modify the source text of your Assembler program.

You need to finish five tasks before invoking the assembler:

1. create the front-end analog-to-digital signal acquisition code, using the Editor. For this you need to learn the commands and rules for editing and file manipulation under ISIS-II, and the analog instructions of the AS2920 Assembler.
2. scale the signals entering each stage of the filter. For this you need to have a fairly clear idea of the expected original input signal, and to understand the techniques and warnings in Appendix J on scaling. Again you use the Editor (or APPEND commands) judiciously to create the required code (usually simply right shifts of already coded instructions) to precede successive stages.
3. create the digital-to-analog code to output the results of the filter's manipulations, again via the Editor and appropriate Assembler commands.
4. review the code files you now have. You may see an opportunity to combine analog instructions with arithmetic operations, as described in the Assembler manual.
5. create a single program file for submission to the assembler, by copying each existing file in order to the new master. If there were nine such code files, this step might use two ISIS-II COPY commands to effect its purpose:

```
COPY A_TO_D.INP, SCALED.ST1, STAGE2.SCA, ST3SCA.LED, &
STAGE4.COD TO TEMP

COPY TEMP, ST5.COD, ST6.SCA, STAGE7, D_TO_A.OUT TO MASTER
```

(see ISIS-II manual for complete discussion of the COPY command.)

After developing and editing the program into a form ready to test, you can invoke the Assembler as described below.

The AS2920 Assembler may reside on the ISIS-II system diskette or on a non-system diskette. You load the assembler by entering a command that names the Assembler and specifies the source file. You may also name the list and object files, but you don't have to. Control options may also be specified as part of the command.

After the assembler goes into execution, all assembler operations specified are performed without further intervention. If the invocation line has an error, the error is reported and you must retype the commands. You may use upper or lower case indiscriminately. The assembler converts all to upper case for its own use, except for echoing back what you typed exactly as it was.

Example:

```
-AS2920 FILTER.COD
```

(After an ISIS-II prompt, shown here as a dash, you type the command as shown to assemble your source program, which is here assumed to be in the file named FILTER.COD. Assembly listing and object code files will be output to

FILTER.LST and FILTER.HEX, respectively. In addition, a symbol table listing will be suppled, and the symbol table debugging output to the object file is suppressed. These defaults are automatic when you do not specify any controls. It is exactly as if you had typed (on one line only)

```
-AS2920 FILTER.COD PRINT(FILTER.LST) LIST OBJECT(FILTER.HEX) SYMBOLS
        NODEBUG PAGING PAGEWIDTH(120) PAGELENGTH(66)
```

All but the last two options have opposites beginning with NO, like NODEBUG, whose opposite (however) is DEBUG. So you can say NOPRINT, NOLIST, NOOBJECT, NOSYMBOLS, or NOPAGING.

All options are discussed in the AS2920 Assembler manual. Briefly, options beginning with NO suppress the indicated action. PRINT establishes a file for the listing of your program. LIST creates that listing. OBJECT creates the 2920 machine code for your program and stores it in a file. SYMBOLS lists the names you used and their established RAM locations. DEBUG puts the symbol table out to the object file for your use in debugging via the Simulator. The page-related options specify how you'd like to see the listing output. Pages of 66 lines of 120 characters are the default format.

After running the one assembler pass and completing assembly listing and object output, the assembler outputs a sign-off message and summary:

```
ASSEMBLY COMPLETE
ERRORS    = XXXX
WARNINGS  = XXXX
RAMSIZE   = XXXX
ROMSIZE   = XXXX
```

APPENDIX H
DESIGN OF COMPLEX DIGITAL FILTERS USED IN THE 2920

Review of Continuous Analog Filters

Analog filters have been in use for many years for a variety of signal conditioning and modifying operations. Originally, most filters were realized with resistors, inductors, and capacitors. More recently active circuit techniques have allowed elimination of most inductors, which tend to be large and may have linearity and coupling problems. Digital filters, such as realized with the 2920, have characteristics which resemble their analog counterparts. As a result a review of analog filter design and analysis may be of assistance to the reader.

Complex networks of R-L-C (resistor-inductor-capacitor) elements are usually analyzed using complex variable techniques. A complex variable S is used to describe frequency, with a pure sinusoidal waveform of frequency f corresponding to

$$s = j * 2 * \pi * f, \text{ where } j = \sqrt{-1}$$

Each of the elementary RLC components has a voltage-current relationship which can be described by a simple equation:

$v = Ri$; for a resistor of resistance R
$v = sLi$; for an inductor of inductance L
$v = (1/sC)i$; for a capacitor of capacitance C

The complex network is analyzed by using these relationships and the fact that the sum of all currents into a node must be zero, and the sum of voltages around any loop must add to zero. A set of equations is derived from the network topology, and solved to relate output voltage to input voltage, or some other relationship of interest. When such equations are solved for networks with finite numbers of elements, the result will take the form of the ratio of two polynomials in complex frequency s:

$$\frac{v_{out}}{v_{in}} = H(s) = \frac{Z(s)}{P(s)} = A \frac{(s-z_1)(s-z_2)(s-z_3)\cdots(s-z_m)}{(s-p_1)(s-p_2)(s-p_3)\cdots(s-p_n)} \quad \text{EQ.1}$$

The transfer characteristic H(s) is the ratio of network output voltage v_{out} to input voltage v_{in}, and the two polynomials are designated Z(s) and P(s) respectively. Once the transfer characteristic has been found as the ratio of these two polynomials, they may be factored into the form above. In Equation 1, the coefficients designatged z_1, z_2,..., z_m are called the zeros of the transfer characteristic and the coefficients p_1, p_2,..., p_n are known as the poles of the transfer characteristic. The letter m literally designates the number of zeros, and n designates the number of poles, or, equivalently, m represents the order of the polynomial Z(s) and n represents the order of polynomial P(s).

The coefficients of the original unfactored polynomials Z and P must be real if the filter is to be made from "real" components. This also means that if any of the zeros z_i or poles p_i is complex, then there will be another zero or pole present, representing the complex conjugate of z_i or p_i. Each pair of factors corresponding to a complex conjugate pair of poles or zeros may be combined to result in a quadratic term with real coefficients. The frequency response of such a filter may be found by substituting the value $j2\pi f$ (where j is the square root of -1) for the value of s. The complex value of the gain expression contains both amplitude and phase information.

The development of operational amplifiers has made possible filter realizations which use only resistors and capacitors. These filters usually consist of a cascade of stages, with each stage realizing a single real pole or a complex conjugate pole pair. Zeros are realized by interconnections between the stages realizing the poles. Such a network may be designed by factoring the polynomials describing the desired transfer characteristic, and then putting the poles into an order that groups the complex conjugate pairs separately from the real poles. Each complex conjugate pair is realized by a separate stage. Real poles may or may not need to use a separate stage.

Impulse Response Analysis

The description above mentioned how the frequency response of a filter may be determined by solving for the polynomials in complex frequency s. However, an alternate description of a filter is its impulse response, i.e. its response to a single impulse stimulation.

The impulse response of a filter may be used to determine the response to a more complicated wave form by treating that wave form as a sequence of impulses of varying amplitude. The individual responses are accumulated, a process known as convolution, which is described by the equation:

$$y(t) = \int_0^t h(\tau)\, x(t-\tau)\, d\tau \quad \text{EQ.2}$$

where $h(\tau)$ is the impulse response of the filter, x(t) is the input, and y(t) the output of the filter at time t.

The impulse response of a filter may be found from the complex frequency polynomial ratio using Laplace transforms.

Using Partial Fractions

One convenient method for finding impulse response consists of first expanding the polynomials as a series of partial fractions, P(s) is first factored into quadratic terms, corresponding to complex conjugate pole pairs, and first order terms, corresponding to real poles. The expansion is then developed using the factors, i.e.

$$P(s) = (s^2 + a_1 s + b_1)(s^2 + a_2 s + b_2)(\ldots)\ldots(s + A_1)(s + A_2)\ldots(\ldots) \quad \text{EQ.3}$$

$$\frac{Z(s)}{P(s)} = A_0 + \frac{A_1 s + B_1}{s^2 + a_2 s + b_1} + \frac{A_2 s + B_2}{s^2 + a_2 s + b_2} + \ldots + \frac{R_1}{s + r_1} + \frac{R_2}{s + r_2} \quad \text{EQ.4}$$

NOTE

Multiple poles with the same value require a somewhat different form of expansion.

Each term in equation 4 is then replaced by its transform, often drawn from a table such as Table I:

Table H-1. Laplace Transforms

Frequency domain term F(s)	Time (impulse) domain term f(t)
A	Impulse of weight A
$\frac{A}{s}$	Step of amplitude A
$\frac{R}{s+r}$	Re^{-rt}
$\frac{As+B}{s^2+as+b}$	$e^{-at/2}\left\{A\cos\left(\sqrt{b-a^2/4}\ t\right) + \frac{B-aA/2}{\sqrt{b^2-a^2/4}}\ \sin\left(\sqrt{b^2-a^2/4}\ t\right)\right\}$

The overall impulse response of the filter is the sum of the impulse responses represented by each of the individual terms in equation 4. As a result, the impulse response of any filter consisting of a finite number of RLC components will normally consist of a sum of exponentials and exponentially decaying sinusoids.

Canonical Forms of Digital Filters

A band-limited signal may be completely reconstructed from discrete samples of its values. As long as a signal is maintained in a band-limited form, it is possible to perform arithmetic operations on samples of the signal yielding results equivalent to arithmetic operations performed on the continuous signal.

The processed samples may then be used to reconstruct the equivalent modified continuous signal. As long as the operations performed are linear, i.e.

$$F(x+y) = F(x) + F(y)\ ;\ \text{where } F \text{ is the operation}$$

then a band-limited signal will retain its band-limited nature throughout the processing. Digital filtering consists of processing digitized samples of signals in a manner similar to the methods for realizing continuous analog filters.

Figure 1 is a block diagram of a digital filter module. Each block labeled z^{-1} is a unit delay, i.e. a delay of one inter-sample interval. The other blocks are multipliers (X) and adders (Σ). The values A_0, A_1, A_2, B_1, and B_2 are coefficients which determine the behavior of the module.

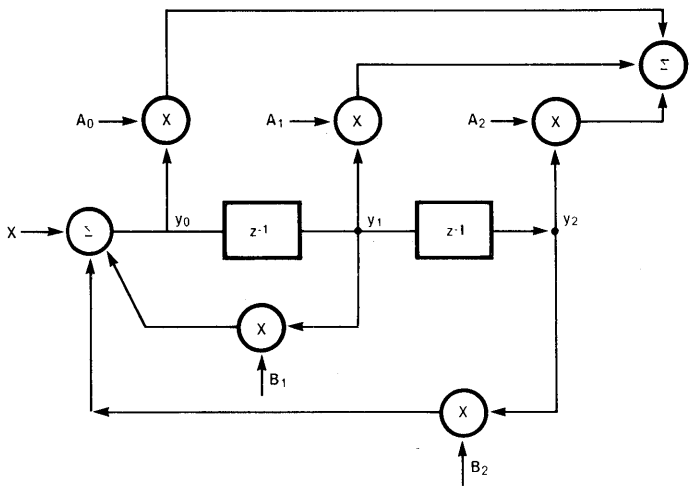


Figure H-1. Digital Filter Module (Second Order Section) 121533-34

The stage shown in Figure 1 behaves in a manner analogous to a continuous analog stage which realizes a complex conjugate pair of poles. For example, if the structure initially has values Y1 and Y2 equal to zero and is excited by a single impulse (i.e. one sample of unit value followed by zero-valued samples), the output may take the form of samples of an exponentially decaying sinusoid. The impulse response of the stage may be expressed as:

$$h(0) = D+A$$

$$h(iT) = e^{-\alpha iT}\, A\cos(\beta iT) + B\sin(\beta iT) \quad \text{for } i > 0$$

when

$$B_1 = 2e^{-\alpha T}\cos\beta T$$

$$B_2 = -e^{-2\alpha T}$$

$$A_0 = D+A$$

$$A_1 = -(2D+A)e^{-\alpha T}\cos\beta T + Be^{-\alpha T}\sin\beta T$$

$$A_2 = De^{-2\alpha T}$$

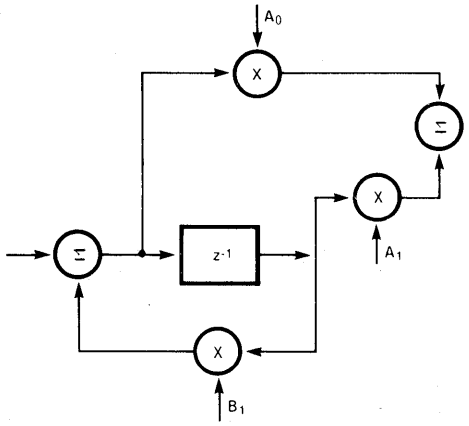


Figure H-2. Digital Filter Module (First Order Section) 121533-35

The diagram of Figure 2 corresponds to a stage realizing a single real pole. Its impulse response takes the form:

$$h(0) = D+A$$
$$h(iT) = Ae^{-\alpha iT}$$

when

$$B_1 = e^{-\alpha T}$$
$$A_0 = D+A$$
$$A_1 = -De^{-\alpha T}$$

From the equations, it can be seen the impulse responses consist of (optional) initial delta functions, followed by a series of samples which are equivalent to having sampled an exponential decay, or an exponentially decaying sinusoid.

Therefore, if we have a *continuous* filter F1 that has an impulse response which consists of a sum of decaying exponentials or esponentially decaying sinusoids, we can realize a *digital* filter F2 that has an impulse response whose values at each sample time are identical to those we would expect from F1. This impulse response may be achieved by building a network of the structures shown in Figures 1 and 2, and summing their outputs.

This procedure defines a type of transform from the continuous domain to the sampled domain, that is, the sampled domain structure implements an impulse response equivalent to having sampled the impulse response of the corresponding continuous filter. This transform is known as the "impulse invariant" transform, and is one of several which may be used to relate the sampled world and the continuous world.

Because of the nature of the sampling process and the corresponding frequency folding about the sample rate, it is not possible for a digital filter to duplicate exactly the characteristics of a continuous analog filter. As the frequencies of interest approach and exceed half the sample rate, the frequency characteristics of the digital filter differ radically from those of its continuous counterpart. These differences may be shown by solving for the frequency response of the second order digital filter section as shown below:

$$F(jw) = \frac{A_0 + A_1(\cos wT - j \sin wT) + A_2(\cos 2wT - j \sin 2wT)}{1 - B_1(\cos wT - j \sin wT) - B_2(\cos 2wT - j \sin 2wT)}$$

Note that a periodic function of frequency results, unlike the continuous case.

Sampled systems can be described as functions of a complex variable z, where $z=e^{sT}$ and T is the inter-sample interval. In Figure 1, each of the blocks labeled z^{-1} corresponds to a unit delay of time T. It is possible to describe the characteristics of the block diagram of Figure 1 as a ratio of polynomials in z or z^{-1}.

Consider the case of a continuous analog filter where one stage realizes a single exponentially decaying sinusoid. Just as such a structure corresponds to a single pair of complex conjugate poles, the diagram shown in Figure 1 is capable of realizing a single exponentially decaying sinusoid and corresponds to a single complex conjugate pair of poles in the complex z plane. Figure 3 shows a plot of the frequency response of the typical second order continuous section, and, for comparison, that of a second order sampled section, for the case where the impulse invariant transform described above was used.

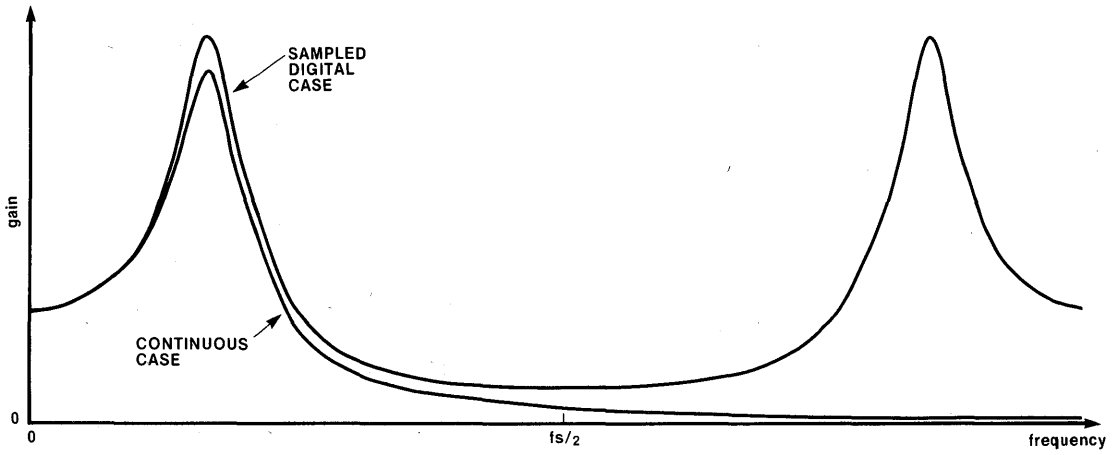


Figure H-3. Comparison of Digital and Continuous Frequency Response 121533-36

Matched Z Transform

Another method for converting from the s-plane to the z-plane is known as the matched z transform. This method is simply a technique for mapping each pole or zero of the s-plane to a corresponding pole or zero in the z-plane. A pole or zero at a + jb on the s-plane is transferred to a pole or zero at $e^{(a+jb)T}$ on the z-plane, where T represents the sample interval in seconds. In polar coordinates, this z-plane location is (e^{aT}, bT). The equations for the coefficients are shown below:

Second order sections for a continuous pole pair $-a \pm jb$ in the s-plane

$$B_1 = 2\, e^{-aT} \cos\ bT$$

$$B_2 = -e^{-2aT}$$

for a continuous zero pair at $-a \pm jb$

$$A_1 = 2\, A_0 e^{-aT} \cos\ bT$$

$$A_2 = A_0 e^{-2aT}$$

First order section

for a real pole at $-a$

$$B_1 = e^{-aT}$$

for a real zero at $-a$

$$A_1 = -A_0 e^{-aT}$$

This transform is not guaranteed equivalence in either frequency or time domains, although pole positions correspond to the impulse invariant transform. The transform is sometimes useful for conceptually estimating the influence, on the resulting filter characteristic, of moving the poles or zeros. In general, it is easier to predict the impact on frequency response of moving a pole or zero in the s-plane than in the z-plane, because the s-plane axes are more directly related to frequency.

The matched z transform allows a one-to-one correspondence of poles and zeros in the s-plane to poles and zeros in the z-plane. One use of this transform is therefore to aid manipulation of the positions of poles and zeros in the z-plane in order to achieve some desired frequency response.

Rather than attempt to do the complete design on the s-plane and then transform to the z-plane to achieve the desired filter, the designer manipulates the poles and zeros in the s-plane while observing the frequency response of the digital filter resulting from the matched z transform. Once the desired characteristic is obtained, the coefficients of the filter are determined by using the transform. This technique has been implemented in the SPAC20 Compiler, and aids the empirical design of filters when mixtures of continuous and digital filters are used.

NOTE

When dealing with complex frequencies in the s-plane or "TS" plane, the SPAC20 Compiler accepts and displays values in Hertz, rather than the traditional radians/sec of the s-plane. The equations shown here utilize the radian/sec representation of frequency. If the frequencies are given in Hertz, they must be multiplied by 2π to connect them to the radian/sec form before use in the equations above

Bilinear Transform

This transform is a method for mapping the s-plane (jw) frequency axis into the z-plane unit circle, such that the continuous s-plane frequency scale from DC to infinity is mapped into a corresponding frequency range of DC to one-half of the sample rate. Therefore, this transform distorts the frequency axis or the frequency characteristics of the filter.

However, the transform does have the property that the shape of the frequency characteristics of the analog filter is preserved with the exception of the frequency distortion. It is common to pre-distort the characteristics of a continuous filter to compensate for the transform's distortions, and thereby implement a sampled filter with a frequency response very closely resembling that of its continuous counterpart. The equations for the bilinear transform are shown below. (A macro implementing this transform, is available for use with the SPAC20 Compiler. It appears in Chapter 10.)

Bilinear Transform Equations

The equations for the Bilinear Transform are:

$$S \rightarrow \frac{2\ (1-Z^{-1})}{T\ (1+Z^{-1})} \qquad Z \rightarrow \frac{(2/T+S)}{(2/T-S)}$$

where T is the sampling interval.

That is, given a polynomial expression (in s) for the transfer characteristic of a continuous filter, a corresponding digital filter may be found by substituting

$$\frac{2}{T}\frac{(1-z^{-1})}{(1+z^{-1})}$$

for each occurrence of s, and then converting the resulting expression to a ratio to two polynomials in z.

These functions map the jw axis of the s-plane onto the unit circle of the z-plane.

i.e. when

$$S = j\Omega$$

where Ω is the analog frequency (in radians/sec)

$$Z = \frac{(2/T + j\Omega)}{(2/T - j\Omega)} \quad \text{or} \quad |Z| = 1$$

The Bilinear Transform maps the point

$$\Omega = 0 \text{ to } Z = 1$$
$$\Omega = \infty \text{ to } Z = -1$$

and the entire left half plane into the unit circle.

A nonlinear distortion is produced by the mapping of the analog $j\Omega$ axis onto the z- plane unit circle. This distortion is given by the mapping

$$\Omega = 2/T \tan WT/2 \qquad W = \frac{2}{T}\tan^{-1}(\Omega T/2)$$

where Ω is an analog frequency and W is a corresponding digital frequency in radians/sec

As an example of using the Bilinear Transform, consider the design of a lowpass digital filter with a cutoff frequency of f_c (in Hz):

1) Convert f_c to radians/sec and find the proper prewarping for the equivalent analog filter:

$$\Omega_c = \frac{2}{T}\tan\frac{W_c T}{2} \qquad W_c = 2\pi f_c$$

2) Design an analog filter that will satisfy the given specification with a lowpass cutoff frequency of Ω_c in radians/sec or $\Omega_c/2\pi$ Hz. Express the transfer function as a ratio of polynomials in s.

3) Use the bilinear transform on the transfer function in s (obtained in step 2) to obtain a transfer function in z, i.e., replace each occurrence of s with

$$\frac{2}{T}\frac{(1-z^{-1})}{(1+z^{-1})}$$

The digital filter which corresponds to the z-plane expression from step 3 (figure 4) will now have the desired cutoff characteristic.

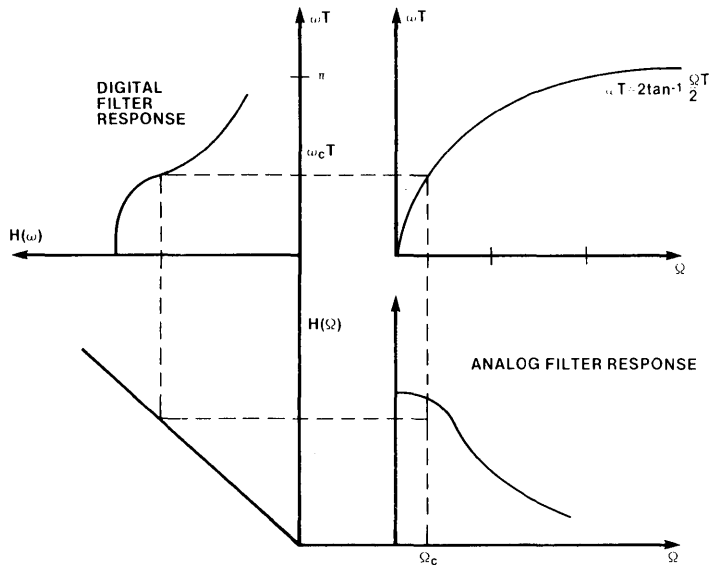


Figure H-4. Transfer Function From Ω to w 121533-37

Note that this transform may alter the number of poles and zeros involved. If poles and zeros are independently transformed, redundant poles or zeros may occur. Using this transform requires careful elimination of such redundancies.

Implementing Filters with the 2920

Once you have determined the locations of your filter's poles and zeros in the z-plane, converting this structure into 2920 code is relatively straightforward. In the blocks of Figures 1 and 2, there are three basic operations performed to achieve digital filtering action: a unit delay represented by the symbol z^{-1}, and addition and multiplication.

For time invariant filters, i.e. those for which the R's, L's, and C's used are fixed and stable with time, the multiplications performed will be of some variable Yi by a constant represented by the values A_0, A_1, A_2, B_1, or B_2. The goal of the 2920 programmer is to implement these functions in a minimum of 2920 instructions.

The blocks labeled z^{-1} correspond to unit delays, i.e. delays of one sample interval. The sample interval is the time it takes for the 2920 to make one pass through its program. The value on the output side of a delay block represents the value computed at the block's input on the previous pass through the program.

The delay can be realized by a RAM location which retains the data from the previous pass until it is needed. A single LDA instruction of the 2920 is sufficient to implement a unit delay block. Figure 1 shows two delay blocks; thus two LDA instructions and two RAM locations are required. These instruction have the form shown below:

```
LDA Y2, Y1, R00
LDA Y1, Y0, R00
```

After executing these two instructions, the RAM location designated Y2 contains the value of Y1 from the previous pass, and Y1 contains the value of Y0 from the previous pass. To complete the filter realization, it is sufficient to complete the calculations of the new value of Y0 from the current values of input, Y1, and Y2, and then compute the output from Y0, Y1, and Y2. The new value of Y0 involves multiplication of Y1 and Y2 by the constants B1 and B2. The instruction set of the 2920 permits implementing these multiplications-by-constants as a series of addition and subtraction steps.

In general, the coefficients are not realized exactly, but rather are approximated as closely as necessary to meet the filtr specifications. This permits minimizing the number of 2920 program steps required to realize the multiplications.

Each ADD or SUB instruction of the 2920 can be thought of as adding a value to (or subtracting it from) the destination operand (e.g. Y1 in the last instruction above). The value used in that operation is the product of some power of two and the source operand (e.g. Y0 in the last instruction above). There is a simple algorithm for converting a multiplication by a constant into a series of additions and subtractions. It consists of choosing, at each step, the particular power of two and the specific addition or subtraction operation which will minimize the error, i.e. produce the closest approximation to the desired value.

For example, consider the coefficient B1 = 1.8. The power of two that would most closely approximate this value would be 2^1, or 2. This value may be realized with a single 2920 instruction:

```
ADD Y0, Y1, L01
```

The error in realizing B1, after this step, would be 2−1.8=+0.2. If such an error is too large, another 2920 instruction step is added. To reduce an error of +0.2, the programmer subtracts the value 2^{-2} or 0.25 from the approximation, giving a net approximation of 1.75 and an error of −0.05. If −0.05 is still too large an error, an additional 2920 step equivalent to adding the source operand multiplied by 2^{-4} = 0.0625 can be added. A net approximation of 1.8125 results, with an error of +0.0125. This process can be repeated until the coefficient is realized with adequate accuracy for the filter requirements. A more powerful version of this algorithm is used in the 2920 Signal Processing Applications Compiler's CODE generation command.

Because there are two coefficients in the filter, two sequences of operations must be defined as described above. As the procedure described performs an addition to the destination location, it is necessary to initialize the destination location. This can be done by clearing the location (e.g. by subtracting the location from itself) or by converting an addition operation to an LDA and placing it as the first step of the sequence. The last steps to realize the filter involve adding the weighted input variable and computing the output. Procedures similar to those above are used for the multiplications and additions needed for these operations.

Some Practical Considerations

The procedures described above show how second order filter sections can be realized. In selecting the gain for the filter, the user should consider the scaling of the variables within the filter. Improper scaling can result in a number of problems.

If the variables are very small, it is possible that the 25-bit word width will not provide enough resolution, and significant truncation noise will be introduced. Because a second order filter of this type may perform the equivalent of integrations in which results are obtained by summing many small values, roundoff error can occur in unexpected ways.

If the variables are scaled too large, overflow saturation may result, with behavior very similar to that occurring in an analog circuit when the signals exceed the dynamic range of the amplifiers. However, an additional consideration may be important in 2920 realizations of second order sections. As coefficient products are developed by series of additions and subtractions, intermediate values may be larger than those finally obtained.

In general, it is necessary to provide sufficient margins when scaling input variables to ensure that overflow saturation does not occur for intermediate values. Sometimes the sequence of calculations can be ordered to minimize potential overflow saturation.

A third method to prevent intermediate overflow saturation is to compute some fraction of Y0, restoring it to full value when it is transferred to Y1, such as shown in Figure 5. This of course adds some noise to the final output, lowering the accuracy somewhat.

The coding generated by the SPAC20 Compiler is already ordered and scaled in this manner to minimize overflow. The user must still address the issue of scaling for input and for signals propagated from earlier stages.

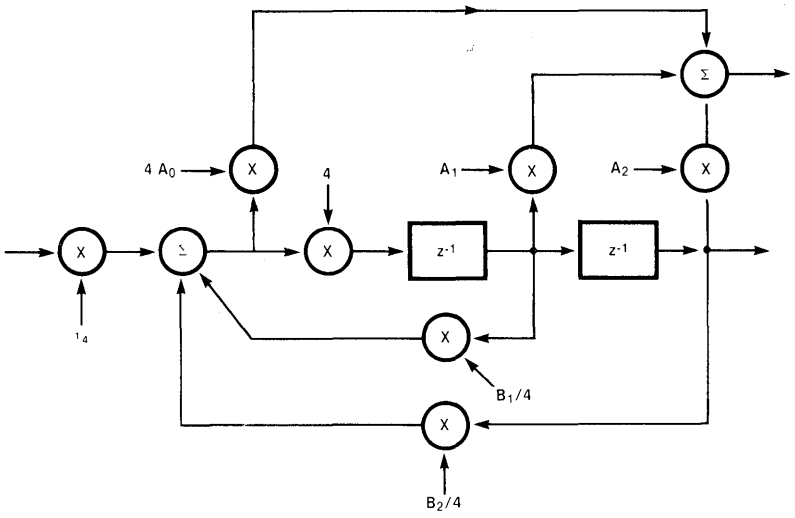


Figure H-5. Method for Preventing Intermediate Overflow 121533-38

(If overflow occurs, it will be when Y0 is increased and loaded to Y1.)

No additional instructions are necessary in general, because the extra multiplications shown in Figure 5 can be performed by modifying the instructions of the original realization.

When a filter consists of a cascade of second order sections, code can be saved by performing any gain trimming calculations at just one point in the cascade. However, to maintain properly scaled variables, the gain for the inputs to each stage should be adjusted by the appropriate power of two. The proper scaling factor can be determined by evaluating the maximum gain from the input to each point in the cascade, starting with the first stage. The gain for the input to that stage is adjusted to ensure that the overall gain does not exceed unity at any frequency. After each stage is adjusted, the process is repeated for the next stage. See Appendix J for more details.

Very Low Frequency Filters

As mentioned above, the processes occurring in the recursive second order section are equivalent to integration. When very low frequency filters or filters with very high Q's must be realized, even the 25-bit word width of the 2920 may not provide adequate protection from truncation error. In some cases it may be possible to reduce the clock rate (and therefore sample rate) which will reduce coefficient precision requirements.

When other functions prevent reduction of the sample rate, or when the predicted value of clock rate must be lower than the minimum permitted by the 2920, alternate programming techniques must be used. (The 2920 word size and the dynamic range of the variables being processed establish a maximum ratio of sample rate to frequencies of interest.)

For very low frequency filters, the effective sampling rate must be reduced or the effective precision of the processor must be increased. One approach, extended precision arithmetic, appears possible but cumbersome. When very low frequencies are being used, the coefficients B_1 and B_2 approach very closely to the values +2 and −1 respectively. By realizing the filter as shown in Figure 6, the small terms B_1-2 and B_2+1 are isolated from the large terms and scaled upwards by some power of two. The equivalent multiplications may then be done using single precision, which is converted back to extended precision by a 2^{-n} scaling.

Extended precision arithmetic may be executed using masks derived from the constants, or by conditional additions. In either case, carries generated by the low order word are added to the high order word to maintain carry propagation. The carries may be simulated in one of the high order bits of the low order word, tested via conditional operations or masking, and then removed by masking or conditional addition of a negative constant. Table II shows an extended precision add routine.

Table H-2. Extended Precision Add Routine (48 Bit Precision) Technique Uses Simulated Carry at 2nd Bit From Left of Low Order Word

```
ADD   YL,   XL,   R00   ; add low order word (25 bits + carry)
LDA   TMP,  YL,   R00   ; copy word to temporary location
AND   TMP,  KP4,  R00   ; mask off simulated carry bit
SUB   YL,   TMP,  R00   ; clear carry from low order word
ADD   YH,   XH,   R00   ; add high order words
LDA   TMP,  TMP,  R13   ; move carry to right
ADD   YH,   TMP,  R10   ; add carry to high order word
```

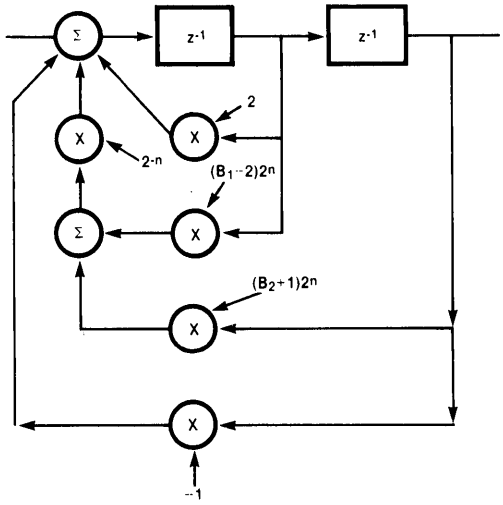


Figure H-6. Very Low Frequency Filter 121533-39

When low frequency filters must be realized, it is in general more convenient to reduce the sample rate rather than attempt to extend the precision of the variables. The sample rate may effectively be reduced by using the conditional load operation triggered by an oscillator run at a submultiple of the sample rate. The filter calculations go to completion every nth cycle. Such an oscillator can be realized by the program shown in Table III.

Table H-3

```
; Oscillator

  SUB OSC, KP1, R05        ; subtract constant KP1 from OSC
  LDA DAR, OSC, R00        ; move to DAR for sign test
  LDA OSC, KP3, R00, CNDS  ; re-initialize if negative to
  ADD OSC, KP3, R05, CNDS  ; 99 times KP1

; conditional filter implementation

  LDA Y2,  Y1,  R00, CNDS  ; delay occurs only on cycling
  LDA Y1,  Y0,  R00, CNDS  ; of oscillator

; remainder of filter calculations are done unconditionally - result is valid
; only on cycling of oscillator
```

The filter code generation may be done with the SPAC20 Compiler by using the *effective* sample rate. To use this filter at the normal sample rate, the output code must be edited to add the CNDS operations to the delay realization.

A constant value is subtracted from a RAM location on each pass through the program. If (and only if) that operation causes the result to be negative, the condition for re-initializing the oscillator is met. A conditional load operation restores the oscillator to a positive value. Thus the oscillator cycles at a submultiple of a sample rate (at 1/100 in the Table III example.)

The filter itself is realized using the same equations as are used in any second order section, with the exception that the delay realization operations i.e. loading Y1 to Y2 and Y0 and Y1, are performed only on those program passes which re-initialize the oscillator. Because the oscillator calculations only produce re-initialization every nth cycle, a sample rate has been achieved equal to the 2920 sample rate divided by n.

On occasion, it may be desirable to operate one or more stages of the filter at a higher sample rate than that of the 2920. For example, it may be possible to use a lower cost external anti-aliasing filter by sampling the inputs at a higher than normal rate, and performing some of the anti-aliasing using a digital filter stage operating at this higher rate. Subsequent processing of the data is performed at the nominal rate of the 2920.

One means for achieving the higher sample rate it to use two copies each of the sampling routine and the anti-alias digital filter section. Figure 7 shows the impact on the external anti-alias requirements obtained by using the double sample rate technique. External anti-alias requirements may also be reduced for 2920 outputs by the use of interpolating digital filters, i.e. filters which compute values between successive samples.

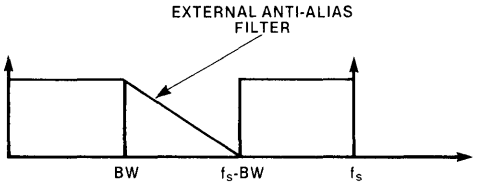
Interpolating filters may also be realized by operating a filter stage at twice the sample rate by using two copies of the program withinin the 2920. There are two options for the input of such a filter operating at twice the sample rate. The same input sample may be used for both copies of the program, or one copy may use a zero-valued input. The latter case resembles using an impulse source where the former case is more like a sampled and held source. The methods produce somewhat different frequency responses.

The SPAC20 Compiler can be coerced to produce code for this mixed sample rate implementation. To accomplish this, set the TS to the faster rate (say 3 times the 2920 program loop rate) and, using the CODE command, generate code for the anti-alias (low-pass) stages of the filter. Three copies of this code must appear in the final program.

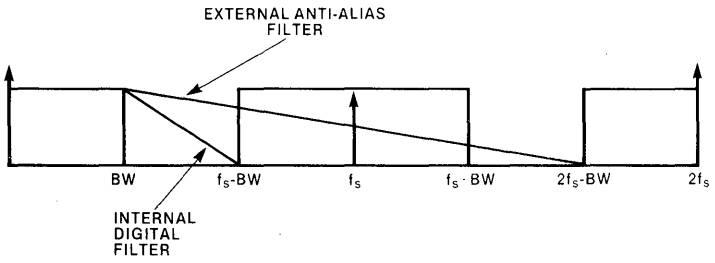
Then set the TS down to the 2920 program loop rate, and generate code for the remaining stages of the filter. One copy of this code must appear in the final program.

At this point, the filter responses which can be graphed and otherwise examined are relatively accurate reflections of the true behavior, at least below half the slower sample frequency. This assumes that the signals are transformed between the stages using the impulse method (either true input or zero) as opposed to the hold method (either true input or held true input).

In most of the examples described above, a cascade of filter stages has been assumed. However, when the impulse invariant transform is used, an alternate realization could be found by expanding into a sum of partial fractions, evaluating the impulse response associated with each fraction, and realizing the output of the filter as the sum of the section outputs. The resulting realization is shown in Figure 8b as opposed to the cascade structure of Figure 8a. In some cases, the parallel structure may be less sensitive to variable scaling than the cascade structure.



a. Original spectrum showing bandwidth of digital processing.
External anti-alias filter must pass below BW, stop beyond f_S−BW



b. Spectrum using double rate sampling.
External filter passes BW, stops beyond $2f_S$−BW, internal digital filter performs rest of anti-alias function.

Figure H-7. Effects of Double Rate Input Sampling 121533-40

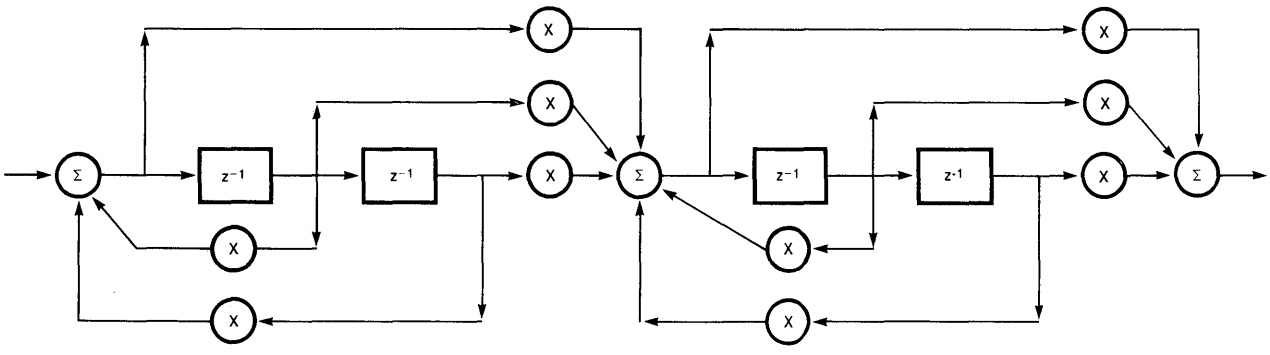


Figure H-8a. Cascade Structure for Complex Filter (Directly Derived From Matched Z or Bilinear Transform) 121533-41

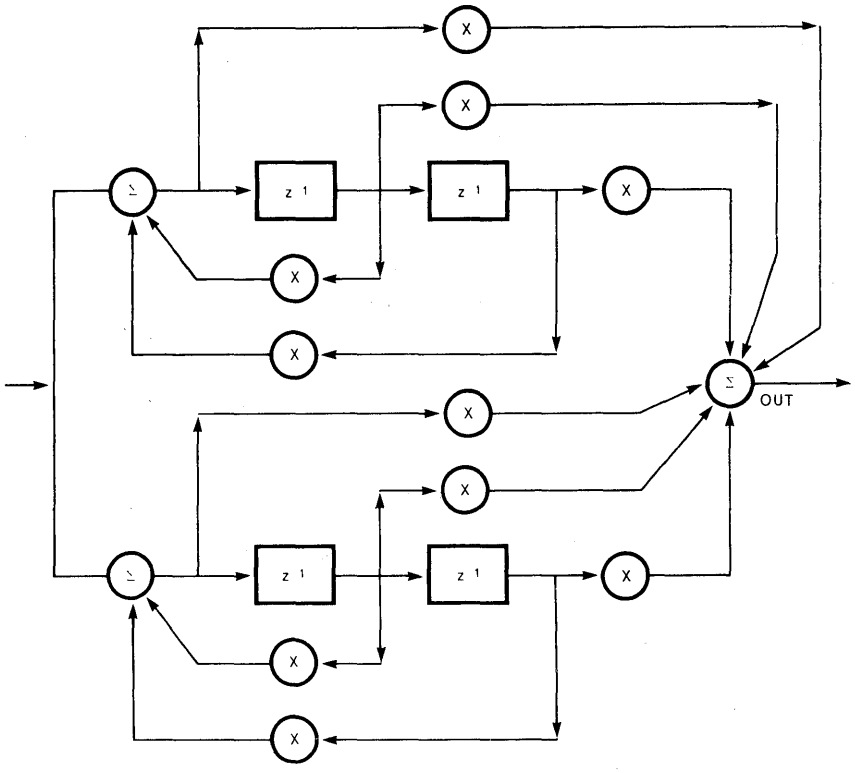


Figure H-8b. Parallel Structure for Complex Filters (May Result From Impulse Invariant Transform)

121533-42

APPENDIX I
FORMULAS USED BY THE SPAC20 COMPILER

The formulas by which the filter response keywords are calculated are given below. They depend upon s-plane or z-plane representation of the locations for poles and zeros. Three distinct graphs are used to indicate the quantities named in the formulas for AGAIN, and four additional graphs are referred to by the formulas for PHASE. Poles are indicated by the character X, zeros by 0. The character a shows the object's real part (or projection), b shows its imaginary part, and f indicates the varying frequency of interest. These letters then appear in the formulas. For z-plane graphs, R indicates the length of the vector from the origin to the pole or zero, and theta (Θ) the vector's angle.

GAIN, MAGAIN, GERROR, and MSQE are defined in terms of AGAIN. GROUP is the negative of derivative of PHASE with respect to frequency. The formulas are shown in the simplest relation to the graphs. Simplification, grouping, and recombination of terms would in some cases produce more compact formulas, but their meaning and relation to the positions of poles and zeros would be obscure. In some cases, the result of such manipulations is in fact much more complex than the original formulation, though it can have computational benefits for the efficiency of a tool such as the 2920 Signal Processing Applications Compiler.

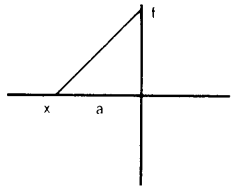
AGAIN

$$AGAIN = \frac{\prod_{Zi} DIST_i}{\prod_{Pi} DIST_i}$$

AGAIN is the ratio of two products: the product of all the distances of the zeros of the filter divided by the product of all the distances of the poles of the filter, where distance means the vector distance from the frequency in question (on the vertical j axis) to the position of the zero or pole (or complex conjugate of a pole).

S-PLANE

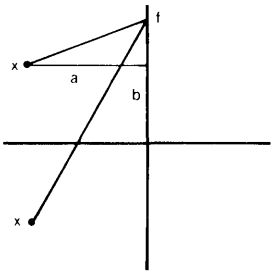
REAL POLE OR ZERO at −a



121533-29

$$DIST(f) = \frac{\sqrt{f^2+a^2}}{N} \quad \text{where } N = \text{normalization factor} = (1+a)$$

COMPLEX POLE OR ZERO at (−a+jb), (−a−jb)



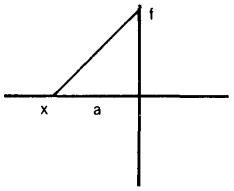
121533-30

$$DIST(f) = \frac{(\sqrt{a^2+(f-b)^2})\ (\sqrt{a^2+(f+b)^2})}{N}$$

$$\text{where the normalization factor } N = (1+\sqrt{a^2+b^2})^2$$

SAMPLED S-PLANE (sampled at T)

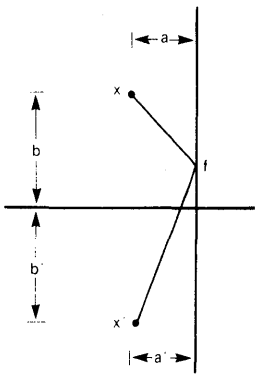
REAL POLE OR ZERO at −a



121533-29

$$DIST(f) = |1-e^{-2\pi aT} * e^{-j2\pi fT}|$$

COMPLEX POLE OR ZERO at (−a+jb),(−a−jb)

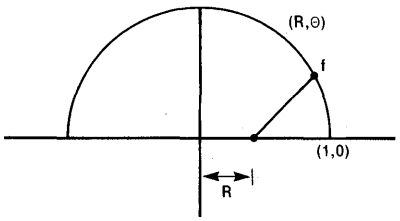


121533-31

$$DIST(f) = |1-(2e^{-2\pi aT}\cos 2\pi bT)\ e^{-2\pi jfT} + e^{-4\pi aT} * e^{-4\pi jfT}|$$

Z PLANE (sampled at T)

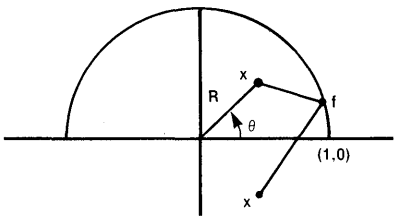
REAL POLE OR ZERO at R, 0



121533-32

$$DIST(f) = |1-Re^{-j2\pi fT}|$$

COMPLEX POLE OR ZERO at R,Θ



121533-33

$$DIST(f) = |(1-Re^{-j\Theta}e^{-j2\pi fT})\ (1-Re^{+j\Theta}e^{-j2\pi fT})|$$

$$GAIN(f) = 20 \log_{10} \left\{ \frac{AGAIN(f)}{GREF} \right\}$$

units in dB

GREF = gain at reference frequency specified by user

$$MAGAIN = \max \{ AGAIN(f_i) \}$$

f_i in FSCALE

$$GERROR(f) = \begin{cases} GAIN(f) - UBOUND(f) & \text{if } GAIN(f) > UBOUND(f) \\ GAIN(f) - LBOUND(f) & \text{if } GAIN(f) < LBOUND(f) \\ 0 & \text{otherwise} \end{cases}$$

$$MSQE = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (GERROR(f_i))^2}$$

N = number of points in FSCALE

f_i in FSCALE

PHASE $= \sum_{Z_i} \Theta_i - \sum_{P_i} \Theta_i$ **(units are radians)**

S-PLANE

REAL

$$\Theta = \tan^{-1} \frac{f}{a}$$

COMPLEX

$$\Theta = \tan^{-1}\left(\frac{f-b}{a}\right) + \tan^{-1}\left(\frac{f+b}{a}\right)$$

SAMPLED S-PLANE (sampled at T)

REAL

$$\Theta = \text{angle of } (1-e^{-2\pi aT} \cdot e^{-j2\pi fT})$$

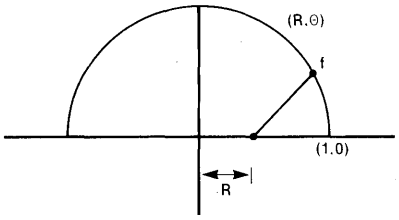
$$\text{angle of } a+jb = \tan^{-1}\frac{b}{a}$$

COMPLEX

$$\Theta = \text{angle of } 1-2e^{-2\pi aT}\cos 2\pi bT\, e^{-2\pi jfT} + e^{-4\pi aT}.e^{-j4\pi fT}$$

Z PLANE (sampled at T)

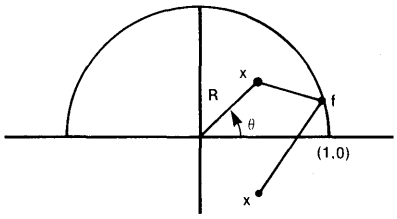
REAL



121533-32

$$\Theta = \text{angle of } 1-Re^{-2j\pi fT}$$

COMPLEX



121533-33

$$\Theta = \text{angle of } (1-Re^{-j\Theta}e^{-j2\pi fT})\,(1-Re^{+j\Theta}e^{-j2\pi fT})$$

GROUP DELAY:

$$\text{GROUP}(f) = \frac{-1}{2\pi}\cdot\frac{d\,\text{phase}}{df}$$

With HOLD ON, AGAIN is multiplied by | sin(x)/x |, where x=TS*FREQ*PI which causes GAIN to be corrected by adding 20 $\log_{10}$ | sin(x)/x | and PHASE to be corrected by adding x. GROUP is corrected by subtracting TS/2.

With HOLD OFF, the above corrections are omitted.

APPENDIX J
SCALING AND OTHER CONSIDERATIONS

Scaling

Each stage of a filter performs various arithmetic operations, which have the potential for causing overflow saturation if the numbers coming in are too large. Thus the issue arises of scaling down such input to avoid inadvertent overflow. If the signal input to the filter were a pure sine wave, then the peak AGAIN for each stage would indicate how incoming signals needed to be scaled in order to avoid saturation or overflow within that stage. That is, if the peak AGAIN under these circumstances were 50 for stage 1, the next higher power of two should be used to scale the input. Thus a right shift of six, equivalent to dividing by 64, would be the correct scaling, e.g.,

```
LDA INPUT, DAR, R06
```

If the first two stages taken together indicated an AGAIN peak of 250, the input to the second stage needs only an additional reduction factor of 4, i.e., rightshift 2, as part of the scaling has already been done at the input to the first stage. Successive stages repeat this reasoning, using the cumulative effect (product) of all earlier scaling factors to determine what, if any, additional scaling is needed. (Note: set HOLD OFF when using AGAIN to determine scale factors, since the HOLD compensation really does not affect the signal until it leaves the 2920 chip.)

Because inputs are limited to the range plus-or-minus 1.0, the largest possible instantaneous output of a filter may be found by integrating the absolute area over the impulse response of the filter. In general this value is unnecessarily conservative, and may result in excessive truncation error.

Even assuming this input scaling has been performed, it is possible that intermediate calculations within a filter stage will cause overflow. For example, warning messages produced by the Compiler in the code may say, e.g.,

```
'';NOTE: MAKE SURE SIGNAL IS <0.547''
```

indicating such intermediate overflow will occur unless the expected maximum output signal amplitude is less than 0.547.

If the input to a stage is scaled so that the expected maximum output signal is less than 0.25, such intermediate overflow cannot occur (for poles and zeros within the unit circle on the z-plane).

As a rule of thumb, it should be sufficient to use four times the MAGAIN (rounded up to the next power of 2) as the scaling needed for each stage. That is, the user should ensure that the scaling before any stage is at least four times the MAGAIN due to the combination of all previous stages and the present stage under consideration.

During the coding process, if β_1 is implemented first in the coding, a warning message will appear, advising the user to keep the signal below a certain level. If the above scaling factor of four times MAGAIN has already been performed, then the purpose of these messages has already been accomplished.

Signal Propagation

In the code output from the SPAC20 Compiler, you will find, usually once per module, an ADD or LDA instruction using names like IN0, IN1, followed by the stage label, e.g., IN0_Z1 for zero 1. These are the instructions which must be replaced using the correct scaling and appropriate input source, which depends on the sequence and combination of poles and zeros, as follows:

The table below indicates the number of inputs and outputs for the four kinds of poles and zeros. A complex pole takes in one signal and produces three output signals, whereas a complex zero uses three inputs and produces one output. A real pole has one input and two outputs, while a real zero takes in two signals and outputs one.

(using hypothetical poles and zeros labeled 3 (for complex) and 2 (for real).

	Input Signals	Signal Delay	Output Signals	Signal Delay
Complex Pole	IN0_P3	Signal Input, not delayed	OUT0_P3	Signal Input, not delayed
			OUT1_P3	Delayed 1 sample interval
			OUT2_P3	Delayed 2 sample intervals
Real Pole	IN0_P2	Signal Input, not delayed	OUT0_P2	Signal Input, not delayed
			OUT1_P2	Delayed 1 sample interval
Complex Zero	IN0_Z3	Signal Input, not delayed	OUT0_Z3	Signal Input, not delayed
	IN1_Z3	Delayed 1 sample interval		
	IN2_Z3	Delayed 2 sample intervals		
Real Zero	IN0_Z2	Signal Input, not delayed	OUT0_Z2	Signal Input, not delayed
	IN1_Z2	Delayed 1 sample interval		

Merging Code for Poles and Zeros

From the table it is easy to perceive the proper meshing of the code for a complex pole followed by that for a complex zero:

```
IN0_Z3 EQU OUT0_P3
IN1_Z3 EQU OUT1_P3
IN2_Z3 EQU OUT2_P3
```

This provides the correct, suitably delayed pole output signals to the appropriate zero inputs. Similarly, to merge the code for a real pole followed by a real zero, you can use

```
IN0_Z2 EQU OUT0_P2
IN1_Z2 EQU OUT1_P2
```

If, however, a complex pole is followed by a real zero, then as the table indicates you must select the pole output with the correct delay, i.e.,

```
IN0_Z2 EQU OUT0_P3 ; no delay
IN1_Z2 EQU OUT1_P3 ; delay
```

A complex pole followed by two real zeros cannot be directly merged. The first zero can be merged with the pole as above. Then the signal needs to be propagated to the second real zero (here labeled Z22). For example, the code below

```
IN0_Z22 EQU OUT0_P3
LDA IN1_Z22, IN0_Z22
```

will accomplish this for a real zero. An equivalent method in the real case is to create a pole at 0,0,Z and then merge it with the zero. For a complex zero, the code below should be used:

```
IN0_Z3 EQU OUT0_P3
LDA IN2_Z3, IN1_Z3
LDA IN1_Z3, IN0_Z3
```

Use of Temporary RAM Locations

The coding for equations of the form YY = C*YY is only optimal assuming no scratch RAM locations are to be used. You can often improve it by the simple expedient of coding in two steps, first saying

```
CODE XX = 1. * YY
```

and then, after saving that code, enter CODE YY = C * XX. Using XX as a scratch variable in this way can be a useful technique.

If more than 16 bits of precision is needed in constants, say in PERROR or ERROR constraints, the code produced with the standard SPAC20 algorithms may suffer.

```
CODE YY = (1 + 1/2**17) * XX ERR < 0
```

is an example. Five instructions are generated where 3 could suffice. One way around this is to code in several steps, e.g.,

```
CODE YY = C*XX ERR< (1/2**13)
DEFINE .ERROR$$SAVER = ERR
CODE XTMP = (1/2**13) * XX
CODE YY = (.ERROR$$SAVER * 2**13) * XTMP + YY   ERR <(1/2**10)
```

This effectively combines coding for the top 13 bits and for the least significant 13 bits.

APPENDIX K
ERROR MESSAGES AND CORRECTIVE ACTIONS

Error conditions encountered by the SPAC20 module cause a numbered error message to print on your console.

Since commands are read on a line-by-line basis, the Compiler will not flag any error until after an entire line has been entered. When the first command error is found, command processing stops and the offending line has no further effect on any internal variables.

If a syntax error is encountered by the Compiler during a multi-line compound command, the error is reported and the line is ignored. Whenever possible, the lines already successfully entered in the compound command are kept and input may resume. Sometimes the compiler finds it impossible to do so and in this case the entire compound command is lost. The prompt for the next input line indicates which of these options was selected: "....*" indicates continued compound command input; "*" indicates new command input.

Errors during compound command execution will terminate processing, leaving the compile-state intact as of the last successfully completed command.

Error numbers C0 to CF are warnings of conditions which are probably undesired. These warnings do not terminate compound command processing.

The following list of error messages does not include those which can come directly from ISIS-II. These appear in a separate list after the Compiler's messages.

In some rare cases, the error number may be printed without its associated message, as in

```
ERR 80: ?
```

This means an Error 80 was detected but some other (probably unrelated) problem prevented the Compiler from printing the message, e.g., error message file SPAC20.OVE is missing.

ERR 71:	ILLEGIBLE NUMBER	A floating point number input cannot be deciphered.
ERR 72:	HELP FILE MISSING	The help file SPAC20.0VH is missing.
ERR 73:	SAMPLE RATE UNDEFINED	TS has not yet been assigned. TS must be assigned a nonzero postive value before sampled poles and zeros can be created or before IMPULSE or STEP responses are examined.
ERR 74:	GREF AGAIN ZERO	The frequency specified in the GREF has absolute gain zero and this cannot be used as a reference level. Select a different GREF frequency.
ERR 75:	NEGATIVE RADIUS	Poles or zeros defined in the Z-plane must have positive radius. A negative radius is equivalent to a positive radius with an angle offset of 180° $=\pi$ radians.

ERR 76:	POLE/ZERO NOT SAMPLED	A pole or zero must be sampled before code can be generated. Move it to the TS or Z planes.
ERR 77:	CONSTRAINT TOO SEVERE	The pole or zero or multiplication cannot be coded within the instruction constraint specified. Try relaxing the INST constraint.
ERR 78:	ANGLE > PI OR <= −PI	Poles or zeros in the Z-plane must have angle between $\pm\pi$. Take the desired angle mod 2π to obtain this.
ERR 79:	EXTRA CONTINUOUS ZEROS	The STEP or IMPULSE time responses cannot be calculated because there are more continuous zeros than continuous poles. Such a combination cannot be physically manifested.
ERR 7A:	ILLEGAL CODE COMMAND	The code command issued does not exist, e.g., CODE Y = 1*X + Z.
ERR 7B:	INTEGER NEEDED	An integer valued expression is needed in context e.g., POLE 1.5.
ERR 80:	SYNTAX ERROR	The token flagged is not one that is allowed in the current context.
ERR 81:	INVALID TOKEN	The token flagged is illegal because it does not follow the rules for a well-formed token. The line is ignored and you must re-enter your intended command. Check the correctness of the syntax and variable-names used. A string longer than 255 characters can result in this error.
ERR 83:	INAPPROPRIATE NUMBER	The value printed on the preceding line is not appropriate in the current context. Some contexts allow only certain numbers, e.g., TS must be positive.
ERR 84:	PARTITION BOUNDS ERROR	The partition values entered in a command are not correct. Either the left part of the partition is greater than the right part, or the values of the partition extremes are out of range in the current context. For example, Poles 3 thru 2.
ERR 85:	ITEM ALREADY EXISTS	The symbol or macro entered in a define command is currently defined in the symbol or macro table. You may need to validate the current usage of this symbol or macro, or perhaps merely use a different spelling to maintain the distinction.
ERR 86:	ITEM DOES NOT EXIST	The item printed on the preceding line does not reside in the symbol or macro table. It may have been removed in an earlier test session, or it may be in a change you haven't inserted yet.

ERR 90:	MEMORY OVERFLOW	Either too many poles and zeros have been defined (more than 20), or too many macros or symbols have been defined or some other internal buffer size has been exceeded. If the message MEMORY RECLAIMED appears on the next line, success may be obtained by simply reissuing the command which caused the original overflow. Before doing so it is recommended to delete any unused symbols or macros first.
ERR 91:	STACK OVERFLOW	The capacity of a statically allocated internal stack has been exceeded. This is probably due to an excessively complicated command, e.g., one with 20 parenthesis pairs. An example would be DEFINE .DAR$SAVED = ((((((((((((((((DAR))))))))))))))))
ERR 92:	COMMAND TOO LONG	Too complicated a command due to number of operators, most probably, as in DEFINE .TEMPFUNC = 1+8*9-7/44*.... out to many operators. Break it up in several smaller commands.
ERR 94:	NON-CHANGEABLE ITEM	An attempt to alter a read-only item, e.g., INST.
ERR 9D;	LINE TOO LONG	Command line was longer than 122 characters.
ERR A0:	TOO MANY PARTITIONS	An fscale or lbound or ubound has been specified with more than the maximum number 10 of piecewise linear segments.
ERR B9:	NO HELP AVAILABLE	Help has been requested for a help item which has no help message.
WARN C8:	F.P. INVALID OPERAND	The program tried to use a value resulting from an underflow or overflow condition. If this message persists, try flushing the Compiler's internal storage with the command XSIZE = XSIZE.
WARN C9:	F.P. OVERFLOW	A value larger than 10 times the largest allowable number occurred in some expression.
WARN CA:	F.P. UNDERFLOW	A value smaller than the smallest allowable number occurred. One example is 1/largest#.
WAR CB:	F.P. ZERO-DIVIDE	Dividing by zero was attempted.
WARN CC:	F.P. DOMAIN ERROR	One example would be the square root of a negative number.

The last five warnings are flagged during command execution due to an inappropriate action or result for a floating point operation. See the documentation for the FORTRAN floating point libraries for further details.

ERR E7:	ILLEGAL FILENAME	The filename specified does not conform to a well-formed ISIS filename. See *ISIS Manual* for valid formulation and device labels.
ERR E8:	ILLEGAL DEVICE	Illegal or unrecognized device in filename. An invalid device label was used, e.g., :D0: instead of :C0, or something unrelated such as :PQ:. See *ISIS Manual* for valid list.
ERR E9:	FILE OPEN FOR INPUT	Attempt to write to a file open for input, e.g., PUT :CI:, a file predefined as console input.
ERR EF:	FILE ALREADY OPEN	Attempt to open a file that was already open.
ERR F0:	NO SUCH FILE	The file specified does not exist. Possibly a wrong or missing device label, as in typing :F2:FILE when you meant :F3:FILE, or a file missing due to forgetting to copy it onto a new disk.
ERR F1:	WRITE-PROTECTED FILE	The file named for output is write-protected and cannot be over-written.
ERR F3:	CHECKSUM ERROR	An overlay file cannot be loaded because it has become trashed.
ERR F6:	DISKETTE FILE REQUIRED	A file was referenced which needs a diskette.
ERR F9:	ILLEGAL ACCESS	Attempt to open a read-only file for the purpose of storing data (e.g., specifying :CI: as the list device) or to open a write-only file as a source of data (e.g., :LP: in an include command).
ERR FA:	NO FILE NAME	No filename specified for a diskette file (e.g., no filename following :F2:).
ERR FH:	NULL FILE EXTENSION	An expected filename extension was not found (e.g., :F2:FILT.).

INDEX

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

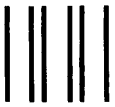
ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

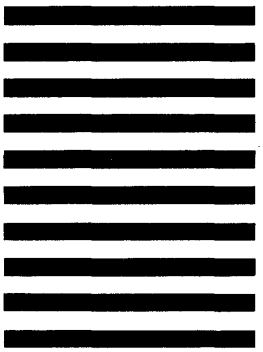
Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.



NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, CA 95051 (408) 987-8080

Printed in U.S.A.