DN 6000345

PROGRAMMING THE TELEMETRY DATA PROCESSOR ADVANCED PCM DECOMMUTATOR

Acroamatics, Inc.
70 South Kellogg Avenue
Goleta, CA 93117-3476

November 19, 2001

PROPRIETARY NOTICE

Information contained in this document is disclosed in confidence and may not be duplicated in full or in part by any person without prior written approval by Acroamatics, Inc., except as provided by separate contractual agreement. Its sole purpose is to provide the user with adequately detailed documentation in order to install, operate, maintain, and order spare parts for the equipment supplied. The use of this document for any other purpose is expressly prohibited.

ACROAMATICS DOCUMENT HISTORY

The following table indicates major changes made to *Programming the Telemetry Data Processor Advanced PCM Decommutator*, Acroamatics Document Number 6000345, released on June 13, 2000, and contains a record of all revisions made since that date.

DN6000345 CHANGE HISTORY			
Rev	**Date**	**Action**	**Name**
	6-13-00	Original Issue	BOBD
A	11-5-01	Added CCSDS Support Syntax	BOBD

PROGRAMMING THE TELEMETRY DATA PROCESSOR ADVANCED PCM DECOMMUTATOR

TABLE OF CONTENTS

CHAPTER 1 PROGRAMMING LANGUAGE STRUCTURE 1-1
INTRODUCTION 1-1
Other Documents 1-2
HOW THE DECOMMUTATOR WORKS 1-2
THE PROGRAMMING LANGUAGE 1-3
Language Structure 1-4
The Syntax and How It is Described 1-4
How Statements are Described in the Manuals 1-6
THE COMPILER OUTPUT LISTING 1-7
ERROR DETECTION AND ERROR CODES 1-9

CHAPTER 2 PCM DECOMMUTATOR PROGRAMMING 2-1
THE EXECUTABLE PCM DECOMMUTATOR PROGRAM 2-1
THE PCM STATEMENT 2-1

CHAPTER 3 PCM SYSTEM SETUP 3-1
THE SYSTEM SETUP (SYS) STATEMENT 3-1
Source Selection 3-1
Logical Stream Identifier 3-1
Three- and Five-Bit Window 3-2
Polarity Auto-correction 3-3
Parity Bits Included in Data 3-3
Leading or Trailing Parity 3-4
Minimum Error Detection 3-4
No Data in Check 3-4
Polarity Selection 3-4
Odd and Even Parity 3-5
Quick Auto Polarity 3-5
Reserved ID Tags Generated by the PCM 3-5
Quality Word Enable or Disable 3-6
Fixed ID 32-Bit Output Stream 3-7
Re-tag Subframe Data 3-7
THE TEST SIMULATOR 3-8

CHAPTER 4 THE MAINFRAME AND SUBFRAME SEGMENTS 4-1
THE MAINFRAME STATEMENT 4-1
Complementing Sync Patterns 4-3
THE SUBFRAME STATEMENT 4-3
The ID Subframe Mask Parameters 4-5
Mask Formation 4-5
The SIZE Parameter 4-8
Parity Bits 4-8
Time Ordering in the Mask 4-9

CHAPTER 5 THE PCM FRAME DESCRIPTION 5-1
DATA WORD DECLARATIONS 5-1
WORD Statement 5-3
WORDS Statement 5-4
CWORD Statement 5-4
RPT Statement 5-4
DUP Statement 5-6
LWORD and LWORDS Statements 5-6
SUP Statement 5-6
NAMES STATEMENT 5-7
WORD PROPERTIES 5-7
LEN - Change Word Length 5-9
MSB, LSB - Word Orientation 5-9
NPA, PA - Parity Control 5-9
NWD - Normal Word 5-9
OUTPUT DATA JUSTIFICATION 5-10
SYNC - A RECYCLING SYNC WORD 5-11
CHK - A CONDITIONAL SYNC WORD 5-12

CHAPTER 6 SUBCOMMUTATED WORDS 6-1
SFn - SUBCOMMUTATED POSITION DESCRIPTION 6-1
ID - the Position of an ID Sync Word 6-2
RCY - the Position of a Recycle Sync Word 6-2
ENTER - Entering an ID Subframe 6-3
RTn - RETURN TO SUBFRAME 6-3

CHAPTER 7 PROGRAM FLOW STATEMENTS 7-1
GOTO STATEMENT - UNCONDITIONAL TRANSFER 7-1
CMP, TST AND IFT - CONDITIONAL TRANSFERS 7-1
CALL STATEMENT - SUBROUTINE CALL 7-3
RET - RETURN FROM SUBROUTINE 7-3

CHAPTER 8 CCSDS PACKET DATA PROCESSING 8-1
PACKET DATA IN THE PCM FRAME 8-1
SUBFRAME SETUP 8-2
EXTRACT THE PACKET LENGTH 8-2
LOAD THE PACKET LENGTH 8-2
TEST THE PACKET LENGTH REGISTER 8-3
LOAD THE TAG REGISTER 8-3
OUTPUT FROM THE TAG REGISTER 8-4

CHAPTER 9 FORMAT PROGRAMMING 9-1
PROGRAMMING THE MAINFRAME 9-1
VARIABLE LENGTH MAINFRAMES 9-2
RECYCLE SUBFRAMES 9-4
ID SUBFRAMES 9-5
DEFAULT, NSYNC and FOR Statements 9-6
Multiple Entry Points - ENTRY 9-7
Example ID Subframe Programs 9-8
ESTABLISHING SUBFRAME WORD PROPERTIES 9-10
COMPLEMENT FRAME SUBFRAME SYNC 9-11
ASYNCHRONOUS SUBFRAMES 9-12

DATA MESSAGE FORMATS 9-13
Processing Chapter 8 1553 Data 9-14
Processing CCSDS Packet Data 9-20
The Mainframe Segment 9-20
Processing The Minor Frames 9-21
Processing The Packet Data 9-22
Extracting The Packet Length 9-24

APPENDIX A
INTRODUCTION A-1
PROGRAMMING EXAMPLE 1 .... A-1
PROGRAMMING EXAMPLE 2 .... A-2
COMPILER HELP A-3
UPDATING YOUR PROGRAMS A-4

PROGRAMMING THE
TELEMETRY DATA PROCESSOR
ADVANCED PCM DECOMMUTATOR

CHAPTER 1
PROGRAMMING LANGUAGE STRUCTURE

INTRODUCTION

This document is a user's reference manual describing the programming system used in the Acroamatics Telemetry Data Processors equipped with the Model 1502 Advanced PCM Decommutator for VME platforms, or the Model 1602 for PCI. It explains the programming language necessary to set up the Decommutators.

The Decommutator was originally designed as a 10 MHz stored program decommutator with multiple subframes. The design goal was to fully decommutate complex PCM formats so that data that would otherwise require post-flight computer analysis could be processed in real-time. Since then, it has been repeatedly redesigned and improved. We have carefully preserved its architecture through these redesigns, first, because it is highly successful, and second, so that, as much as possible, user software written for earlier models will run unchanged on current models. Acroamatics has always been committed to this idea.

While older Acroamatics decommutators had 16-bit instructions, the Advanced PCM Decommutator has a 32-bit instruction set. This allows the decommutator to operate at much higher speeds, processing at rates up to 32 MHz, and it also permits direct addressing of up to 1,048,576 words of instruction space. Extremely large format programs can be accommodated in a single, seamless setup.

The Advanced PCM Decommutator is designed to be a replacement for earlier frame synchronizers. The 32-bit instruction set of the 1502/1602 is largely compatible with the 16-bit equivalent in previous versions. Of course there are several new instructions in the advanced decommutator that add features and capabilities, but most setup programs for older machines will run on the 1502/1602 with few or no modifications.

Perhaps the most significant difference between the Advanced PCM Decommutator and earlier Frame Synchronizers is the way word properties are handled. In the advanced decommutators's predecessors, word properties are set by executing an instruction (op-code "**D**" or "**E**"). The word properties remain as set *until another word properties setting instruction is executed*. This means the word properties are set following the instruction flow, as the program is executed.

In contrast, in the 1502/1602 card the word properties of a PCM word are incorporated into the output instruction that assigns the ID tag for that word. When a program is compiled for the new card is compiled, a command such as

```
LEN=12
```

which would generate an instruction for earlier decommutators, simply tells the Compiler what the word properties are for subsequent `WORD` statements. In other words, for the 1502 or 1602, word properties are not set by instruction, but rather by commands that have their effect *as they are encountered during compilation.* The way these differences affect your programs will be discussed in greater detail in Section 5 of this manual.

Other Documents

Other pertinent reference documents are listed in the following tables. The documents listed here are those that we currently supply with TDP systems.

HARDWARE TECHNICAL DOCUMENTS	
Acroamatics Document	Document Title
6000329	Technical Manual - Model 502VB PCM Frame Synchronizer

PROGRAMMING AND USER DOCUMENTS	
Acroamatics Document	Document Title
6000252	User's Manual for the PCM Simulator
6000342	Users Manual - Real Time Display for Windows NT
6000343	Programming Manual for the VME Telemetry Data Processor Data Distribution System - 505VA SHARC Processor

HOW THE DECOMMUTATOR WORKS

The Decommutator treats PCM data as a stream of bits divided into packets in an arbitrary way. The packets represent data values that may be from 1 to 32 bits in length. The Decommutator uniquely tags every such sequence, regardless of where it may appear in the format. Synchronization patterns of one sort or another appear in the PCM stream, and the Decommutator uses these patterns to locate the data sequences.

The Decommutator has two major sections. The first section is a pattern correlator whose job is to locate the minor frame sync pattern. It generates control signals that enable the second major section to operate. The second section is a computer that operates synchronized to the data stream. The instructions for this computer are a map of the PCM format. It is these instructions that you are programming when you use the programming language to describe a PCM format. The instructions tell the computer how to dissect and identify the sequences of serial data that make up the PCM stream. They let you select the number of bits to extract from the stream, the LSB or MSB orientation of the

data in the stream, the MSB or LSB justification of the data in the output data words, and how to handle a parity bit. They let you assign the ID tag to the data. The Decommutator uniquely tags datum according to these instructions and outputs a data message containing the ID tag you choose and the data that comes from the stream. Another section of the TDP adds a time tag to this message and the result is what Acroamatics calls a *DIT*, which stands for "Data, ID, and Time."

Instructions let you identify the sequences of bits in the data stream that are counters or patterns use to identify sub-commutated data in the stream. They let you test the value of data in the stream and decide which sequence of instructions to follow to decommutate the data. This ability to test the date in the stream and remember the results of the test for use at some later time give the Decommutator its unique power to handle time varying PCM formats.

Since the Decommutator tracks the format by "following the map," dynamic variations in the format are easily managed. Because there is no bit counter, there is no maximum number of bits between repetitions of the sync pattern, nor are a fixed number of bits required, provided that the data contains flags to direct the Decommutator. Thus mainframe lengths can vary, lengths of subcommutated sequences can change, and positions of subcommutated words can change.

A special problem of subcommutated data is that the subframes may be asynchronous to each other. When a format contains only subcommutated data synchronous to a single additional commutator, it is always possible to write the frame out unfolded, as though it were simply a very long mainframe. In that instance a major cycle of the data is a fixed sequence. This is not true if there are two or more subframes. The Decommutator solves the problem of this asynchronism by providing separate tracking registers for the mainframe and each of the subframes. Since these are independently synchronized (by the patterns in the data), the TDP is able to follow maps that vary for the subframes as well as for the mainframe. Finally, the Synchronizer can transfer control to a point in the map offset by the value of a data word. This allows decommutation of subframes whose data is identified by an ID embedded in the data.

THE PROGRAMMING LANGUAGE

You program the Telemetry Data Processor using a language, developed by Acroamatics, and processed by a Compiler program that is resident in the TDP. You can use and control all the hardware capabilities of the TDP using this language. For the PCM Decommutator, the Compiler compiles statements you write into machine instructions that the Decommutator can follow to transform the serial PCM stream into parallel data messages.

The complexity of TDP programming is a function of the complexity of the format and what you wish to do with the data. For simple tasks such as decommutating a PCM stream into a number of fixed length data words, a few lines of setup code suffice. To describe a complicated format with elaborate computer generated subcommutation requires you describe the format in detail. You can often simplify TDP programming by processing only those words that are of interest, skipping over the other portions of the frame.

The language minimizes the number of statements required to define a format and its processing. You can define default values for the parameters that define data words in the PCM format, so you have to specify only those things that are different about a particular data word.

Language Structure

The setup language is a stream of text that defines the setup of the TDP. One line of this text is a *statement*. Sometimes more than one statement is required to specify a function; Such *statement groups* always begin with an introductory statement and conclude with an **END** statement. An **END** statement is a line whose first three non-blank characters are **END**. The introductory statement starts a separate processing routine in the Compiler. Subsequently only text recognized by that routine is legal. For example, a **TIME** statement to begin setup of the Time Port won't be recognized within the statement group that sets up a different port and results in an error message.

PCM stream decommutation instructions are the first part of a DATA PORT description the second part of which is a description of the processing of the data. Programming of the PCM Port converts the source program text into binary code that is loaded into the TDP memories.

The Compiler produces a listing in which the source program text is shown indented 16 character positions and the a hexadecimal program is shown at the left margin. If the indentation results in the line being too long to fit in 80 columns, the Compiler folds the line (between words, if possible), and the trailing fragment is printed on the next line, indented 48 characters.

The Syntax and How It is Described

A **program** is a series of lines of text containing program statements and comments.

Comments begin with a vertical bar symbol (|); all characters on a line following a vertical bar are commentary and the Compiler ignores them. So that you may conveniently use comments for program section headings the Compiler suppresses the 16 character indentation if the vertical bar is in column one.

Program **statements** are strings of Alphanumeric characters, including blanks. They end at the end of a line. White space divides statements into sub-strings called tokens. White space may be blank (<**sp**> or tab <**ht**>) Characters. The amount of white space between tokens is up to you. Position the tokens on the line to make your program easy to read.

Sometimes a statement may be too long to fit on an 80 character line. Then you will need continuation lines.

A **Continuation Line** follows a line that ends with a '\' character. It is processed as part of the previous line. You can describe the '\' character as "concealing the newline." You cannot, however, continue a comment line, since the Compiler ignores everything following the '|' character, including a '\' character. Continuation lines may also end with a '\' so you can continue statements indefinitely.

Tokens are strings of alphanumeric characters containing no blanks (white space). The case of alphabetic characters is not significant to the Compiler, and you may use upper or lower case to enhance readability. The Compiler processes statements from left to right, one token at a time. The '\' character, signaling a line continuation, ends a token. This means you must continue a line between tokens, never in the middle of a token.

There are several kinds of tokens.

A **Label** is a 1 to 6 character string, made up of the characters

```
A-Z 0-9 $ .
```

The first character must be Alphabetic (A-Z). A Label may be followed by a ':' and, if it is, it must be the first element on the line. There are two kinds of labels.

Statement Labels are used only in PCM decommutation programs to label the target of a transfer of control instruction. A Statement Label followed by a ':' identifies (labels) a particular program statement. Another statement would refer to that statement by using the same Statement Label. When used in that sense, a Statement Label is not followed by a ':' character. A Statement Label is *defined* when it appears with the ':' to mark a program statement. The Statement Label is *used* when it appears without the ':' in a transfer of control statement.

A **Data Identification Label** (always referred to as an *ID*) is associated with each measurement data source. ID Labels are distinct from Statement Labels, and you may use the same string both as a Statement Label and as an ID Label. ID Labels may be purely numeric. Then their value is the value of the numeric string. Numeric ID Labels must be less than 65536 (/10000). An ID Label is *defined* when it appears without the ':' in certain types of statements such as the PCM **WORD** statement. The ID Label is *used* when it is followed by a ':' in a Processing List, associating a DIT with its processing algorithms.

You may use the symbolic names to reference the DITs throughout the Data Port Description and in the Real-time Display system setup, if any. Symbolic names must begin with an Alphabetic character, and may be up to 72 characters in length. At the end of each Data Port Description the Compiler can print a table showing the hexadecimal numeric value of every symbol defined, sorted by symbol name and by numeric value. To get a symbol table listing, enter the command **PSY** following the **END** statement that closes the Data Port Description.

The Compiler assigns values to ID Labels relative to an offset value that is called the "Port Offset." For example, if you refer to ID 2 in the programming for the a PCM stream whose offset is set to /1000, you are really programming for DIT /1002.

The relative ID assignment can make programming of the Data Ports much easier than it would be using absolute IDs. For example, if you are using two PCM Decommutators to do the same PCM format, you can use the same setup program with the offset set to two different values. That separates the two streams by the amount of the offset. Of course, the ID tags are transmitted throughout the hardware system as their absolute values, and those are the values stored in the symbol table. They may range from 0 to 65535.

Section 2 of this manual explains how you set the Port Offset. Section 5 of this document explains how you program the ID assignments.

Keywords are reserved strings that the Compiler recognizes as having special meaning. Unless a statement begins with a Label, the first token on the line is always a Keyword. For some statements you use additional Keywords to define parameters. You usually don't have to worry about assigning symbolic names or labels that accidentally match Compiler keywords, since user assigned symbols and Compiler keywords are nearly always recognizable by context. Only three symbols are reserved. They are **SUP** and **RTM** which you may not use as names of variables (IDs), and **END**, which you may not use as the first three characters of a label.

Digits are the characters '0' through '9' and, when they are part of a hexadecimal number, the characters 'A' through 'F'.

Numbers are strings of digits, plus certain punctuation. For PCM setup you use only integer values. A number preceded by a '/' is interpreted as hexadecimal. For expressing sync pattern and mask values, the Compiler interprets the number as hexadecimal with or without the '/', except that it interprets a string of only ones and zeroes with no leading '/' as binary. This interpretation of numbers is unique to the PCM setup processor and does not apply to any other part of a TDP setup program.

Some tokens combine both a Keyword and a Number in the form

Keyword=Number
or
Keyword(Number)

When you use these constructions, do not embed any blanks in the token. For example

```
SYNC=/EB90       |Is a correct statement
SYNC = /EB90     |Will cause an error diagnostic
```

How Statements are Described in the Manuals

In this manual, and the other programming manuals, we show statements in two different ways.

1. A statement is sometimes represented schematically. The schematic representation of a statement defines the rules for construction of a real statement. The parameters of the statement are represented in a general way, using a conventionalized notation. The following conventions apply to schematic statements.

   The upper case strings in any schematic representation are **keywords** and must appear exactly as shown.

   The punctuation characters

   ```
   :   (   )   {   }
   ```

   must appear exactly as shown.

   The italic strings

n	A Digit
nn	A Number
label	A Statement Label referenced
label:	A Statement Label defined
id	A Data Identification Label defined
id:	A Data Identification Label used

represent variables that must conform to the formation rules described above.

Optional elements of a line are shown in square brackets. The square brackets do not appear in the statement when the optional element is used. For example the statement schematic

[*label*:] WORD *id* [RTM]

tells us that

```
JA11:       WORD    /202
            WORD    PB11
            WORD    PB21 RTM
```

are three valid **WORD** statements, since they all fit the schematic shown.

2. Example statements show arbitrary typical statements constructed from these rules, presented separated from the text and centered on the page and in typewriter font. In example statements no lower case characters appear, though the Compiler does not distinguish between upper and lower case. The presentation

```
MAINFRAME SYNC=/EB90 ERS(0,0,1)
```

shows how an example statement appears in the text.

THE COMPILER OUTPUT LISTING

The Compiler receives a stream of text from one of the I/O ports. It operates in a single pass mode on the incoming stream. It outputs diagnostics and a hexadecimal listing of the program as it stores it in the TDP memory. It also outputs tabular listings of the data ID label name definitions. Following is a program listing for a PCM format. This document has not yet explained the statements shown, but the listing is presented here to give you an idea of what it looks like.

```
| SETUP PCM DECOMMUTATOR (FORMAT SPECIFICATION)

|Hex list        User program
| output         input

                 PCM  /300
                 SYS  SRC=3  NOAUTO  REVPOL 3BIT
                 MAINFRAME LEN=12 LSB PA SYNC=/DF01CA MASK=/FFFFFE \\
                   ERS(2,2,3) INV=9 CS=2 CL=1 LS=10 ALTC
                         LEN=12
                   *=/400
 lsb par len=12          WORD    0
000040 00580300          (0)
 lsb par len=12          RPT     1
000041 C0001001
                         NWD
 lsb par len=12          WORD    MFW2
000042 00580300          (MFW2)
 lsb par len=12          WORD    SYNC1(/7F)
000043 0058037F          (SYNC1(/7F))
 lsb par len=8           SF1     RCY LEN=8
000044 86500000
000045 0081FFFF          SUP 50
000046 00A5FFFF
 lsb npa len=32          SYNC LEN=32
000047 A3C00000
 lsb npa len=32          WORD    SUP
000048 03C1FFFF          (SUP)
                 SUBFRAME 1 RCY LEN=8 LSB PA ERR=2 \\
                   CS=2  CL=2 LS=5 SYNC=/F6 MASK=/FE
 lsb par len=8           WORD    ABC(/100)  RTM
000049 24500400          (ABC(/100))
 lsb par len=8           WORD    *
00004A 04500700          (*)
 lsb par len=8           WORD    DEF
00004B 04500401          (DEF)
00004C 009FFFFF          SUP 15
00004D 00BDFFFF          SUP 30
00004E 0081FFFF          SUP 32
00004F 0081FFFF          SUP 64
000050 C0001001
 lsb par len=8           WORD    GHI(/801)
000051 04500B01          (GHI(/801))
 lsb par len=8           WORD    LMN(*)
000052 04500701          (LMN(*))
 lsb par len=8           RPT     4
000053 C0001004
 lsb npa len=8           SYNC
000054 A7D00000
 lsb npa len=8           WORD    RSYNWD
000055 04D00B02          (RSYNWD)
```

```
|Computer calculates setup hex output

PCM SETUP WORDS

000000  D0000001   000001  C2000007   000002  80000000   000003  80000000
000004  80000000   000005  80000000   000006  80000000   000007  00000049
000008  80000040   000009  C2000096   00000A  00000000   00000B  00DF01CA
00000C  00000000   00000D  00FFFFFE   00000E  80008000   00000F  80008000
000010  C2000081   000011  00000000   000012  C20000A1   000013  043382A1
000014  C20000B1   000015  00000046   000016  C2000013   000017  000000F6
000018  000000FE   000019  03338225   00001A  C20000C1   00001B  00000627
00001C  D0000040
                ENDFORMAT
                END PCM

|Computer outputs list of all symbolic names
|used in program, along with their ID numbers.

ID SYMBOL TABLE SORTED BY NAME

0400 ABC     0401 DEF     0b01 GHI     0701 LMN     0300 MFW2    0b02 RSYNWD
037f SYNC1

ID SYMBOL TABLE SORTED BY VALUE

0300 MFW2    037f SYNC1   0400 ABC     0401 DEF     0701 LMN     0b01 GHI
0b02 RSYNWD
                PSY
```

The Compiler can operate interactively on a terminal, although this method is only feasible for the simplest programs. The best way to program the unit is to prepare programming files using the editing and program construction facilities of the TDP. You can then compile the files you have created with the command:

```
tdpc file1 [file2 ... ]
```

The Compiler will generate the listing information (if any), and output it to "Standard Out", so you can redirect the listing to a file, to the printer, or pipe it to another program.

ERROR DETECTION AND ERROR CODES

The Compiler detects a number of syntactical and logical errors. It identifies the errors in the listing by an error message. The Compiler classifies errors as *Serious*Errors *Errors*, and *Warnings*. *Serious*Errors prevent the Compiler from continuing. *Errors* cause the compilation of the current line to stop. *Warnings* do not affect operation of the Compiler, but it is probable that you did not do what you intended.

The Compiler outputs error diagnostic messages in the form

```
***ERROR nnnn.
```

Warnings are marked by a message

```
***WARNING nnnn.
```

nnnn is a hexadecimal error code. When an error message (but not when a warning message has been printed) the Compiler prints the offending line with the error flag at the beginning. It underlines the part of the line that was correctly processed. It prints an exclamation point under the first character of the token that caused the error message. The line that contains the underline also contains a brief description of the error. For example

```
***ERROR 0014: WORD ACQ(#21)
Invalid Number -----!
```

Assigning an absolute ID to a label caused the error diagnostic. The error indicator points to the first character of the token ACQ(#21). The error in that token is the pound sign (#) character. The statement "WORD ACQ(21)" is perfectly legal.

When the Compiler prints a warning, it includes the token that caused the warning. For example, the PCM programming statement

```
WORDS  AA(8) BB(8)
```

causes the following listing output

```
0011  0008        WORDS AA(8) BB(8)
** WARNING 0105 ID relabeled  BB(8)
0012  0008        (BB(8))
```

The example assigns ID 8 to two different labels. The Compiler displays the offending token "BB(8)" with Warning 105 and the error message.

Compiler recovers gracefully from syntax errors that it detects. The state of the Compiler after discarding a line with an ERROR is just as though the line had never been read. This means that if you are typing the statement at a terminal, you may simply re-type the statement correctly.

CHAPTER 2
PCM DECOMMUTATOR PROGRAMMING

THE EXECUTABLE PCM DECOMMUTATOR PROGRAM

The object of writing a Compiler Source program is to create an executable program to load into the PCM Decommutator memory. The PCM Decommutator converts a stream of serial PCM to a parallel stream of ID and Time tagged data messages. It tracks the incoming PCM data stream with the program whose instructions are executed in synchronism with the data stream. These instructions set up the data output module, called the Word Processor, to extract and correctly tag the data words as they arrive. The executable PCM Decommutator program is an encoded list of the DITs (Data, ID tag, Time) to be output. This list is stored in the program memory of the Decommutator in the time order in which the data words appear in the PCM data stream. It contains interspersed special instructions that describe how to break up the bit stream and control the flow of the Decommutator program itself.

Once the pattern correlator recognizes the mainframe sync pattern, the Decommutator begins tracking the programmed PCM format.

This document uses the terms *mainframe* and *subframe*. The mainframe is a model of a minor frame and describes the sequence of words that occur during a minor frame, that is, between recirculations of the primary sync pattern. The mainframe may contain subcommutated words, and each independent synchronization pattern creates one additional subframe. A *subframe* and a *subcommutated word* are not the same thing. One subframe can include many subcommutated mainframe word positions.

The Compiler allows you to describe the data frame structure and the programming of operating parameters (such as synchronization word definitions and external device setups) in a readable, though abbreviated, form.

THE PCM STATEMENT

The PCM statement introduces the setup program for a PCM Decommutator. Its schematic is

PCM[*n*] *offset*

where *n* is the PCM Decommutator identifier, (there must be no space between PCM and the *n*), where *n* may be a number between 1 through 8. If *n* is omitted, the Compiler assumes you mean PCM1.

For systems predating VMEbus TDPs, when *n* is A through F it refers to decommutators installed in Model 2240 Programmable Format Synchronizers. So programs written for these older systems can be compiled on VMEbus TDPs, you can specify a "Stream Translation" parameter in **TDPC.CFG**, the TDP Configuration File. The Stream Translation determines how the Compiler translates "PCMA", for example. The default Stream Translation string is

```
STRTRANS=01223A4B5C6D7E8F
```

To interpret the string, you view it as positionally coded character pairs. Either of the first two characters will be translated to "PCM1", the second pair translate to "PCM2", and so on. Here is how the default string translates, then.

```
Input:         0 1  2 2  3 A  4 B  5 C  6 D  7 E  8 F
Translation:    1    2    3    4    5    6    7    8
```

This means, for example, that if your setup file specifies either "PCMA" or "PCM3", it will be interpreted the same way, as "PCM3".

The *offset* is the value that the Compiler must add to the ID tag values that it assigns. Every ID tag in the executable program is a literal numeric value stored in the program memory of the PCM Decommutator. When the Compiler processes your source program it generates these values and stores them into the memory. In your source program you use either symbolic IDs or numeric values relative to the offset. This lets you relocate the entire ID space of your decommutated data by changing only the offset value. If you do not give an *offset*, the ID space is not offset. An alternative way of separating the ID tag space between PCM streams is by assigning a physical stream to a "logical" stream, using the **SYS** command. The **SYS** command is discussed in Section 3 of this document. Regardless of how you manage your ID tag space, you should keep in mind that the TDP system reserves a number of ID tags for such special functions as the PCM data quality word and major and minor time words.

If you intend to record the data you may want to consider which output algorithm you will use when allocating ID tags. Some of the available standard output data stream algorithms for the TDP use bit 13 of the ID field as a time *tick*, and this format is incompatible with the use of bit 13 as part of the ID field. Other subroutines output only 12 bits of the ID tag field. *Programming Manual for the VME Telemetry Data Processor Data Distribution System - 505VA SHARC Processor*, Acroamatics Document Number 6000343, discusses the various output formats and show how many bits of the ID tag are output for each available output format.

Following the **PCM** statement are any number of lines of setup program for the PCM Decommutator.

A typical format specification has the schematic

PCM *offset*
SYS *PCM system setup*
MAINFRAME *mainframe setup and defaults*
:
mainframe description
:
SUBFRAME *n subframe setup and defaults*
:
subframe description
:
other subframes
:
ENDFORMAT
distribution programming
END

The piece of the source program that begins with a **MAINFRAME** or **SUBFRAME** *n* statement is called a "Program Segment" in this document. The program segment extends to the next **MAINFRAME**, **SUBFRAME** *n*, or **END** statement. The defaults you define at the beginning of a program segment apply throughout that segment. The PCM format programming ends with the first **END** statement. In the example above, this is written **ENDFORMAT**. The "FORMAT" tacked on to the **END** is noise that is ignored by the Compiler, and the word **END** alone would do as well, but it helps to document the end of the PCM *format* programming.

The Distribution System programming that follows the **ENDFORMAT** can use the symbolic IDs used in the preceding PCM format. Distribution System programming is described in *Programming Manual for the VME Telemetry Data Processor Data Distribution System - 505VA SHARC Processor*, Acroamatics Document Number 6000343.

CHAPTER 3
PCM SYSTEM SETUP

THE SYSTEM SETUP (SYS) STATEMENT

The **SYS** statement contains the programmable specifiers that affect overall operation of the Decommutator. These specifiers are listed in table 3-1 and are also described in the following sections. A typical System Setup line is

```
SYS  3BIT NOAUTO SRC=1
```

This enables the Three-Bit Window, disables Automatic Polarity Correction, and selects Input Source 1. The order of the specifiers after the introductory keyword (SYS) is immaterial.

Source Selection

The PCM Decommutator has four inputs; One selection is reserved for the 64-bit test simulator built into the correlator, and the remaining three are user inputs. The specifier is expressed

SRC=*n*

where *n* is SIM (or 0) for the test simulator, and 1, 2, or 3 for inputs 1, 2, and 3 respectively.

Logical Stream Identifier

You will probably use the Logical Stream Identifier in multiple stream PCM systems only. You need it to distinguish by stream number the reserved ID tags for the special words that the Decommutator generates. When you set the stream number, you also set the values of the reserved tags to unique values for each of the streams. Because the stream number can range from 0 to 7, you can separate up to eight data streams using the Stream Identifier. The Compiler also automatically applies an ID offset of /2000 times the logical stream number to all tags defined for that stream. This means, for example, that if you have assigned Logical Stream 3 to a setup, its tag space will begin at /6000. The Logical Stream offset is added to any "Port Offset" you specify on the **PCM** declaration line.

To set the value of the stream identifier, include the specifier

STR=*n*

where *n* is the stream number you want to assign, ranging from 0 to 7.

For multiple stream PCM you *must* set the stream identifier to separate the special words. Moreover, you may wish to use the stream identifier to make it easy to separate the data according to the stream it came from, as described above.

SYSTEM CONFIGURATION		
PARAMETER	DESCRIPTION	DEFAULT
SRC=*n*	Select input source as *n* (SIM,1,2 or 3)	1
STR	Program the logical stream number	Stream number=0
5BIT or 3BIT, or NO3BIT	Enable 3-bit or 5-bit window, or disable both	3-bit window enabled
AUTO or NOAUTO	Enable or disable Polarity Auto-correction in SEARCH.	Polarity Auto-correction enabled
DIP	Include parity bits in data	Parity not included
LPAR or TPAR	Specify leading or trailing parity (only applies to data words where parity-strip-and-check is specified)	Trailing parity
MED	Minimum error detection enable	Minimum error detection not enabled
NOCHEK	Do not output Data when the mainframe is in CHECK	Data is output in CHECK
NOPOL or REVPOL	Disable or enable reversal of incoming data polarity	Disable polarity reversal
OPAR or EPAR	Specify odd or even parity (only applies to data words where polarity-strip-and-check is specified)	Odd parity
QAUTO	Quick Auto polarity invert enabled	Quick Auto polarity not enabled
QWO or QWD	Enable (QWO) quality word output or disable it (QWD)	Quality Word output enabled
RAW	Always output raw data	Raw data is not output
RSD	Re-tag subframe data	Subframe data is tagged normally
RSY	Output synchronized data in raw form	Raw data is not output
RXS	Output raw data when frame is out-of-sync	Frame must be in sync to output data

TABLE 3-1

Three- and Five-Bit Window

When you enable the Three-bit Window, the correlator still detects the sync pattern if it arrives one bit early or one bit late of the expected position. When you enable the Five-bit Window, the correlator detects the sync pattern if it arrives up to two bits early or two bits late of the expected position. These features are controlled with the specifiers

```
3BIT      To enable the Three-bit Window
5BIT      To enable the Five-bit Window
NO3BIT    To disable both windows
```

The system default is the Three-bit Window enabled.

Polarity Auto-correction

If you enable auto-correction, the Decommutator inverts the bit stream polarity if it detects the complement of the sync word while in SEARCH. You can program the number of successive complemented patterns that the correlator must receive before the inversion occurs. You control this feature with the specifiers:

```
AUTO        Enable auto-correction
NOAUTO      Disable auto-correction
```

If you give the specifier **AUTO** to enable auto-correction, you should specify the number of complemented sync patterns that must occur before inversion takes place. You do that in the MAINFRAME line (See table 4-1) with the specifier:

`INV=`***nn***

where *nn* is used to specify a number, $1 \le nn \le 16$ (*nn* defaults to 8). Even when the auto-correction feature is enabled, it is best to program the most probable polarity of the data using the NOPOL or REVPOL (see below) specifications, since this minimizes the sync acquisition time if the polarity is initially correct.

Parity Bits Included in Data

The Decommutator can manage word parity in one of three programmable ways. Parity may be:

- Checked and the data parity bit discarded
- Checked and the data parity bit output in the data field
- Not checked, included in the data field

When the parity is checked, the result of the check appears in the *Parity Status Bit* in certain output algorithms (see *Programming Manual for the VME Telemetry Data Processor Data Distribution System - 505VA SHARC Processor*, Acroamatics Document Number 6000343). Do not confuse the parity bit which is a part of the PCM data with the Parity Status Bit.

Three sets of specifiers used with the **SYS** statement set properties for parity management that apply to the entire PCM format.

The first of these determines what happens to the parity bit in the data word itself. A data word is normally output without the parity bit (described as *stripped*). If you use the specifier **DIP**, you include the parity bit as part of the output data. Therefore, if 10-bit data is output with parity, the output word will be 11-bits long. An exception to this is that if the parity bit is leading and the data output mode is right-justified arithmetic (See Section 5) the parity bit would occupy the sign position. Since the Decommutator would extend that bit instead of the actual sign bit, it instead strips the parity bit for that special case.

The other two sets of properties set using the **SYS** statement are whether the parity bit is leading or trailing and whether it is even or odd.

You control whether there is a parity bit in the data separately for each data word in the PCM frame. The **PAR** and **NPAR** specifiers declare presence or absence of a parity bit. They are discussed in Section 5. If you declare the word to have a parity bit, the parity status bit indicates whether the word parity

including the parity bit matched the selected even or odd parity state. If it mis-matches, the parity status bit is set to 1 and that flags a parity error. If a word is declared to have no parity bit, the parity status bit is always 0. The **DIP** specifier has no effect on data words that have *no*parity selected.

Leading or Trailing Parity

Parity position is specified with the specifiers:

```
LPAR  Leading Parity
TPAR  Trailing Parity
```

The position of the parity bit refers to its position in time in the stream, and its position is not affected by the MSB/LSB orientation of the data words in the stream.

Minimum Error Detection

When minimum error detection sync strategy is selected, the synchronizer searches the bit stream for a pattern with the allowed number (or less) of bit mis-matches. It goes into CHECK when it detects a candidate pattern. While in CHECK, the synchronizer continues to look for another pattern with fewer errors than the current sync pattern. If it finds a pattern with fewer errors, it drops back into SEARCH and then goes right back into CHECK, synchronized on the new pattern. This process continues until the pattern with the smallest number of errors is found, at which time the system goes into LOCK.

To enable the minimum error detection function, use the specifier **MED**.

No Data in Check

Usually the system outputs data when the mainframe is in CHECK or LOCK. If you only want to output data when the stream is in LOCK, use the specifier **NOCHEK**.

Polarity Selection

The Decommutator can receive and invert a stream of complemented data. To select the parity, use the specifiers

NOPOL To specify normal (uncomplemented) data
REVPOL To specify that the data is complemented

Odd and Even Parity

Use the specifiers

EPAR Even Parity
OPAR Odd Parity

to select even or odd parity. If Odd Parity is selected words with an odd number of bits will have a zero in the parity bit, and words with an even number of bits will have a one in the parity bit. If Even Parity is selected words with an even number of bits will have a zero in the parity bit, and words with an odd number of bits will have a one in the parity bit.

Quick Auto Polarity

If you select Quick Auto Polarity, the synchronizer inverts (complement) the PCM stream instantly if it finds a complemented sync pattern (in CHECK or LOCK) at the expected sync pattern position. To enable Quick Auto Polarity use the specifier **QAUTO**. If you want to invert the polarity while in SEARCH, Polarity Auto Correction (explained in 3.1.4) must also be enabled.

Reserved ID Tags Generated by the PCM

The PCM has three programmably controlled features that generate data words with fixed ID tags. These ID tags are shown in Table 3-2 below.

SPECIAL ID's	
ID	**Reserved for**
1fe0	Quality word ID
1fe1	Fixed ID Data
1fe2	Subframe in Search

SPECIAL ID TAGS
TABLE 3-2

Each of the reserved ID's shown in Table 3-2 represent the ID's for Stream 0. The three most significant bits of the ID's are determined by the stream selected, therefore there can be eight sets of Quality word ID's, ranging from 1fe0 to ffe0. For example, the Q-Word ID for Logical Stream 4 is /9fe0. Use the **STR** specifier to set the "Logical Stream".

Quality Word Enable or Disable

The PCM Decommutator generates a Data Quality word, the *Q-word*, which reports the status of the synchronizers. It outputs it to the data stream at mainframe sync word time. The specifier **QWO** is supported for compatibility with programs written for earlier machines, but by default Q-Word output is enabled for the Advanced PCM Decommutator. You can disable it with the specifier **QWD**.

Table 3-3 shows the map of the Q-word.

QUALITY WORD FORMAT	
Bits	**Description**
31-25	Reserved
24	Parity Error Found in Frame
23	Pattern Found
22	Bit Slip Corrected
21-18	Number of Pattern Errors
17	Data Inverted
16	Return to Search
15-14	Subframe 6 Status
13-12	Subframe 5 Status
11-10	Subframe 4 Status
9-8	Subframe 3 Status
7-6	Subframe 2 Status
5-4	Subframe 1 Status
3-2	Mainframe Status
1-0	Bit Sync Status
Status = 3: Synchronizer Not Programmed	
Status = 2: Synchronizer In Search	
Status = 1: Synchronizer In Check	
Status = 0: Synchronizer In Lock	

Q-WORD MAP
TABLE 3-3

In Table 3-3, Bit 16, *Return to Search,* means that the strategy system is about to return to search, and that this is the last word which will be output until resynchronization occurs.

Bit 17, *Data Inverted,* means that the criterion for complementing the data has been met, and that the synchronizer has inverted the data.

Bits 21-18, *Number of Pattern Errors,* reports the actual number of conflicts (0-15) in the synchronization pattern just examined. If there are fifteen or more mismatches, this field will be set to 15.

Bit 22, *Bit Slip Corrected,* means that the sync windowing is enabled and the sync pattern arrived either early or late.

Bit 23, *Pattern Found,* means that the pattern just examined satisfied the programmed sync word criteria.

Fixed ID 32-Bit Output Stream

You can output data to the distribution as a series of 32 bit words with a Fixed ID (see Table 3-2). You use the **STR** specifier to set the stream number, and that modifies the Fixed ID as described in earlier in this chapter, under *Reserved ID Tags*.

You can select either one of three fixed formats described below.

1) **RAW** outputs the data to the distribution section as fixed 32-bit words when the TDP in in RUN, whether the synchronizer is in SEARCH, CHECK, or LOCK. When you output **RAW** data, processed or ID tag data is suppressed only to the PCM frame buffer memory.

2) **RSY** outputs the raw data as 32-bit words, synchronous to the frame, when the mainframe is in CHECK or LOCK. The first 32 bits following the sync pattern, regardless of how they are broken up in the decommutator, are packed into the first output word. The raw data continues to be output as 32-bit words, until the end of frame. If the frame length is not a multiple of 32 bits, the final word is zero-padded. When you output **RSY** data, processed or ID tag data is suppressed only to the PCM frame buffer memory.

3) **RXS** outputs the data to the distribution section as a stream of 32-bit words when the TDP is in RUN and the mainframe is in SEARCH. If you also select the **NOCHEK** option as part of the system configuration, the **RXS** function will occur when the mainframe synchronizer is in either SEARCH or CHECK. Otherwise it will output the raw data only when it is in SEARCH.

Re-tag Subframe Data

Once the mainframe has gone out of SEARCH the Decommutator normally outputs subcommutated data according to the decommutation program instructions regardless of the state of the subframe synchronizer. You can use the specifier **RSD** to make the Decommutator re-tag subcommutated data from a subframe whose synchronizer is not in LOCK. The data is tagged with the "Subframe in Search" tag shown in Table 3-2, modified by the **STR** specifier to set the stream number (see the description earlier in this chapter, under *Reserved ID Tags*). The **RSD** specifier applies to all subframes in use; the re-tagging, however, applies only to data from any synchronizer that is not in LOCK. If, for example, a program uses two subframes and Subframe 1 is in LOCK but Subframe 2 is not, the data from Subframe 1 will carry the correct (programmed) ID tags but the Decommutator will re-tag the other synchronizer's data as described above.

The Test Simulator

The PCM Correlator has a built in test generator that can generate a 64-bit stream stream of bits. You can use this to test the Decommutator or to generate test data to test the Distribution system, your distribution setup, or your data path to your host computer. It is this simulator that you use when you use the

```
SRC=SIM
```

specification in your SIM statement. You use the statement

```
SIM xxxx xxxx xxxx xxxx
```

to define the simulator's 64-bit pattern. xxxx xxxx xxxx xxxx represents four arbitrary 4 digit hexadecimal fields.

The **SIM** statement mus either immediately precede or immediately follow the **SYS** statement.

If you use the test simulator to generate data to test the Distribution system or the Host Computer interface, remember that it generates very high rate data. To test data recording you may want to use a data compression algorithm such as **OSN** (Output Sample N) to reduce the data.

CHAPTER 4
THE MAINFRAME AND SUBFRAME SEGMENTS

The format program has Mainframe and Subframe program segments. This organization lets you define separate default word properties for each segment, specifying the orientation and length for the typical words in that segment, and whether the data has parity. Although you can specify how bits are to be extracted from the stream for every word in the major frame, it is convenient to have default selections. You can set the Decommutator to the default properties with a special statement. Moreover, when you set particular properties for a word, properties that you don't specify are set to the default values.

Following the **MAINFRAME** or **SUBFRAME** statement that introduces each program segment is a list of default word property definitions. These definitions apply to the entire program segment. This list also defines the operation of the synchronizer, setting the sync pattern and mask and the synchronization strategy.

THE MAINFRAME STATEMENT

An example of a typical **MAINFRAME** statement is

```
MAINFRAME SYNC=FAF320 ERS(2,2,5) CS=2 CL=1 LS=5 LEN=12 LSB
```

This line establishes the sync word, sets up the strategy, and selects the default word length and orientation for the mainframe data. The definers that you may specify are shown in table 4-1. It also shows the default values that the Compiler assigns when if you don't specify them. The order in which the specifiers appear in the statement is immaterial.

Specify the Mainframe Sync word and Mask as two hexadecimal or binary numbers, up to 64 bits in length. The leftmost bit in the mask represents the position of the first bit in time order. You can most conveniently view the sync word as an MSB first digital word.

If you express the sync or mask word with only ones and zeroes, the Compiler assumes it is binary. Otherwise it treats it as hexadecimal. In the unlikely event that the word contains only *hexadecimal* ones and zeroes, precede the number with a "slash" character to force a hexadecimal interpretation.

If you don't supply a mask, the Compiler generates an all-ones mask beginning at the leftmost hexadecimal digit of the sync pattern, or if expressed in binary, in the left-most bit specified. Thus, if you express the sync word in hexadecimal but the sync word length is not a multiple of four bits, you *must* specify the mask. For instance, if you define 10-bit sync pattern with the statement

SYNC=2D3

you have to express the mask as 3FF. If you don't, the Compiler will generate a mask of FFF. That includes two bits in the pattern that don't belong in it. If you express the sync word in binary as 1011010011, the Compiler can generate the correct mask. Ones in the mask need not be contiguous. The rightmost bit expressed in the pattern must correspond to the last bit of the word

MAINFRAME SYNCHRONIZER CONTROLS		
DEFINER	DESCRIPTION	DEFAULT
SYNC=*nn*	Specify sync pattern, expressed as a binary or hexadecimal number	No default
MASK=*nn*	Set sync word mask (binary or hexadecimal)	Set to all ones with length equal to sync pattern length
ALTC	Specify that the Mainframe Sync pattern must be complemented on every other frame.	Normal sync
CSF	Specify that a complement Mainframe Sync pattern will occur to identify a major frame (Complement Sync Frame sync mode).	Normal sync
CL=*nn*	CHECK-to-LOCK count. Change status from CHECK to LOCK after *n* matched patterns	1
CS=*nn*	CHECK-to-SEARCH count. Change status from CHECK to SEARCH after *n* missed patterns. Cannot be zero	1
LS=*nn*	LOCK-to-SEARCH count. Change status from LOCK to SEARCH after *n* missed patterns. Cannot be zero	1
ERS(*nn*1,*nn*2,*nn*3)	Set allowable number of erroneous bits in the pattern, in SEARCH to *nn*1; in CHECK to *nn*2; in LOCK to *nn*3	0,1,1
INV=*nn*	When Polarity Auto-correction is enabled, complement the data after *nn* inverted sync patterns have been detected	8
MAINFRAME DEFAULT WORD PROPERTIES		
DEFINER	DESCRIPTION	DEFAULT
LEN=*nn*	Set Normal word length in bits	16
MSB or LSB	Specify data orientation	MSB first
PA or NPA	Parity or No Parity	NPA

TABLE 4-1

programmed in the format as the Mainframe Sync word. (See Section 5, **SYNC**.) The right way to think about the pattern and mask is to visualize them in the 64-bit register with the right-most bit of the register aligned to the last bit of the pattern as it arrives in time order. That is, the pattern shifts into the right-most bit, one bit at a time, from the PCM stream.

Complementing Sync Patterns

Complementing sync patterns are commonly used in PCM systems to perform two distinct functions. Alternate Complement Sync requires that the pattern alternate on successive frames. When you select the **ALTC** mode, the PCM Decommutator looks initially for either a complemented or non-complemented pattern before advancing to Check mode, then looks for alternating patterns thereafter. Beware of the polarity auto-correction feature of the Decommutator when you have complement patterns in the data stream. If the mode is alternating complement, you must not enable the auto-correction since there are as many complemented patterns in the stream as normal ones and no way to tell which is which.

When you use the **CSF** specifier you tell the synchronizer to accept a complemented pattern or an non-complemented pattern. The complemented pattern should occur infrequently, usually at the major frame rate. You may use the auto-correction for data of this kind, if you wish, but you must set the auto-correction frame count specified by the **INV** specifier to no less than 2. Then the synchronizer can only see two successive complemented patterns when the data is truly inverted. Be careful not to set the auto-correction frame count to greater than or equal to the number of minor frames in the major frame. If you do that, the auto-correction counter resets every time the complemented sync pattern occurs, and this prevents the auto-correction entirely. If you wish to synchronize to the major frame, use one of the subframe synchronizers in Recycle Mode to look for the complemented mainframe pattern in the mainframe sync word position. (Refer to Section 9 for a discussion of programming complement mainframe subframe sync.)

THE SUBFRAME STATEMENT

You introduce subframes in much the same way as you introduce the mainframe. The **SUBFRAME** statement defines the operating modes, operating parameters appropriate to the mode, and default values for word length and orientation for that subframe. The schematic of the statement is

SUBFRAME *n mode definers*

mode is a keyword that tells the Compiler what kind of subframe you want to set up.
The *mode* keywords for subframes are

MODE	SYNCHRONIZATION TYPE
ID	Binary ID Sync, up count
IDD	Binary ID Sync, down count
JAM	JAM ID Sync
RCY	Recycle Sync

Tables 4-2, 4-3, and 4-4 describe the definers that you may specify for ID, JAM ID, and Recycle subframes, respectively. The definers for ID type subframes pertain to ID or IDD modes.

ID SUBFRAME (ID or IDDWN)		
DEFINER	DESCRIPTION	DEFAULT
CL=*nn*	CHECK-to-LOCK count. Change status from CHECK to LOCK after *nn* matched patterns	1
FW=*nn*	Set flywheel count to *nn*	2
LEN=*nn*	Set default word length to *nn*	MAINFRAME setting
MIN=*nn*	Minimum ID=*n*, *n* may be 0 or 1	0
MAX=*nn*	Maximum ID=*nn*, *nn*<=65534	15
MASK=*nn*	Set sync word mask	Set to a field of binary ones appropriate to the Maximum ID (see description)
MSB or LSB	Specify data orientation	MSB first
PA or NPA	Parity or No Parity	NPA
SIZE=[±]*n*	Number of bits in ID	*n* set appropriate to the Maximum ID

TABLE 4-2

JAM ID SUBFRAME		
DEFINER	DESCRIPTION	DEFAULT
LEN=*nn*	Set default word length to n	MAINFRAME setting
MIN=*nn*	Minimum ID=*n*, *n* may be 0 or 1	0
MAX=*nn*	Maximum ID=*nn*, *nn*<=65535	15
MASK=*nn*	Set sync word mask	Set to a field of binary ones appropriate to the Maximum ID (see description)
MSB or LSB	Specify data orientation	MSB first
PA or NPA	Parity or No Parity	NPA
SIZE=[±]*n*	Number of bits in ID	*n* set appropriate to the Maximum ID

TABLE 4-3

A recycle subframe declaration must include the definition of the sync pattern. If you don't define the mask, the Compiler will generate it exactly as it does for the mainframe mask. The recycle synchronizer works the same as the mainframe synchronizer. As many bits as you specify in you format program are shifted into the right-most bits of the pattern correlator. The pattern and mask registers must be set up to match that alignment. If your data is transmitted in LSB order, and if the pattern is LSB as well, the LSB specifier will reorient the pattern word for the output data stream, but it won't reorient it for the synchronizer. You must bit reverse it when you program the sync word value. The Decommutator provides for separate error thresholds in SEARCH, CHECK and LOCK for recycle synchronized subframes.

The Advanced PCM Decommutator allows the sync and mask word size shown in the following table.

ID Counters	32 bits
Recycle Sync Words	32 bits
Mask Words	32 bits

A typical subframe introduction for an ID subframe might be

SUBFRAME 2 ID LEN=10 MSB MIN=1 MAX=10 FW=3 CL=2

The statement declares that subframe 2 is an ID subframe. It specifies the default word length and orientation for a typical subframe word. The example statement also sets the minimum and maximum values for the ID counter and the sync strategy. Because there is no mask specified, the Compiler will generate a 10-bit mask of 0000001111, binary.

Following the introducing line, the Mainframe and Subframe Description program segments define the format of each data word in the PCM stream, as well as the ID to be attached. The following section explains how you describe words in the PCM data stream.

The ID Subframe Mask Parameters

There is no sync pattern for an ID subframe, but there is a mask. The sync word mask for an ID subframe does something different from the mask for a recycle subframe. For a recycle frame, the mask describes "don't care" bits in the pattern when the pattern is matched to the bit stream. For an ID subframe the mask determines whether the corresponding bit is shifted into the ID register. If there is a '1' in the mask, the corresponding bit shifts in. If there is a '0' in the mask, no shift occurs. This packs non-adjacent bits in the bit stream. This is often useful when you build a JAM ID value from a number of non-adjacent bits in a word of the data stream. For example, if bits 13, 8, and 3 of a 16-bit word could represent a state variable with values ranging from 0 to 7, the mask '0010000100001000' would select those 3 bits and shift them alone into the ID register. The 32-bit mask allows you to select from 32 bits of the data stream, but there should be no more than 16 '1's in the mask. That's the width of the ID register.

Mask Formation

The PCM word that contains an ID counter often is wider than the counter and the position of the counting field may not be the obvious one of right justified in the field. The PCM Frame Synchronizer uses a mask to control which bits contain the counter. You can set the mask with a parameter of the *SUBFRAME n* setup command

MASK=*nnnnnnnnnn*

where *nnnnnnnnnn* is the mask value expressed either in hexadecimal or binary. The mask require some explaining because to get it right you need to understand how it works and how we generate it if you don't specify it.

RECYCLE SUBFRAME		
DEFINER	DESCRIPTION	DEFAULT
ASYNC	Specify Asynchronous Subframe mode	Not ASYNC mode
CL=*nn* CS=*nn* LS=*nn*	CHECK-to-LOCK count. Change status from CHECK to LOCK after *nn* matched patterns CHECK-to-SEARCH count. Change status from CHECK to SEARCH after *nn* missed patterns. Cannot be zero LOCK-to-SEARCH count. Change status from LOCK to SEARCH after *nn* missed patterns. Cannot be zero	1 1 1
ERR=*nn*	Set the allowable erroneous bits in the sync pattern for SEARCH, CHECK and LOCK, to *nn*	0
ERS(*nn*1,*nn*2,*nn*3)	Set allowable number of erroneous bits in the pattern, in SEARCH to *nn*; in CHECK to *nn*2; in LOCK to *nn*3	0,1,1
LEN=*nn*	Set default word length to n	MAINFRAME setting
MASK=*nn*	Set sync word mask	Set to all ones, length equal to sync pattern length
MSB or LSB	Specify data orientation	MSB first
PA or NPA	Parity or No Parity	NPA
SYNC	Specify sync pattern, expressed as a binary or hexadecimal number	No default.

TABLE 4-4

The mask is a map of the bits of the ID word or words. It is left justified in the mask field and each bit corresponds to one bit of the PCM stream that you direct to the subframe synchronizer with the

```
SFn ID
```

command. Thus if you program

```
SF1 ID LEN=20
```

The left-most 20 bits of the mask are the active bits. Bits to the right of those aren't used. If the actual ID field is the right-most 6 bits of the 20-bit word the mask has to be:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 ......

When the Frame Synchronizer executes the *SF1 ID* command it shifts the incoming data from the PCM stream and the mask word together. Where there is a zero in the mask it throws the bit away. Where there is a one in the mask it

shifts the incoming data left into an accumulators least significant bit. At the end of the input word it has accumulated the six bits of the ID value in the accumulator which then becomes the index value for the next minor frame. Of course the bits don't have to be contiguous and you can use the mask to select the bits from anywhere in the 20-bit (or whatever size) word. If you use the *SF1 SYL* instruction you can select bits from words scattered throughout the PCM stream. Then the total width of the mask is the sum of the syllable lengths. The mask is limited to thirty-two bits in length, no more than sixteen of which may be ones.

If the ID word is LSB first in the stream, the mask must be bit-reversed. Even if you program

```
SF1 ID LEN=20 LSB
```

you have to bit-reverse the mask, because the LSB specification controls only the Frame Synchronizer's Word Processor section, causing it to bit-reverse the output word. The incoming bit stream always shifts to the subframe synchronizer in time order. The correct mask for the LSB word is

1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ......

The TDP Compiler, tries to make it easier to set up subframes by generating a default mask for the ID word. The method it uses is more or less a guess based on incomplete information. It can't know when the ID value occupies bits other than the contiguous LSBs of the ID word but that is what it assumes. When its assumptions are wrong, you have to override them by supplying the correct mask using the rules described above.

To explain how the Compiler builds the mask, we will refer to the following example Subframe statement.

```
SUBFRAME 1 ID MIN=1 MAX=12 LEN=10 LSB
```

The Compiler uses the following rules.

1. The default word size of the Subframe statement determines the number of active bits of the mask; in the example, ten bits.
2. The default orientation determines the orientation of the ID word as well; in the example, LSB
3. The maximum value of the ID counter determines the number of ones in the mask. In the example that is twelve so the number of ones is four.

The example will generate the mask

0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ......

Given that, you must shift ten bits to the synchronizer using the *SF1* commands. Usually you will have

```
SF1 ID LEN=10 LSB
```

Suppose, however, that you have chosen to program the ID word as just the four bits that contain the counter field, that is, the *SF1* command is

```
SF1 ID LEN=4
```

The Compiler won't generate the correct mask for that unless you set the default

length in the *SUBFRAME* statement to *4*. That might not be very convenient since you would have to set the correct length for all the other words in the sub-frame. Instead you should define the mask as

```
MASK=1111
```

The SIZE Parameter

Ordinarily you will not need to use the **SIZE** specifier, because the Compiler will generate the appropriate size and mask based upon the **MAX** specifier. In older implementations of the PCM Frame Synchronizer the Compiler set the width of the ID field based upon the **SIZE** specifier. For the Model 502 and the Model 1502 the SIZE parameter is ignored by the Compiler It uses the Maximum value of the ID counter exclusively to set the width of the ID.

If, however, the orientation of the ID sync word is different than the orientation of the subcommutated words, you use the **SIZE** specifier to override the Compiler's defaults. When you express the size with an explicit sign ("+" or "–"), you are declaring the orientation of the sync word (MSB or LSB, respectively), irrespective of the orientation of the typical word in the subframe. The size is the number of bits needed to accommodate the maximum value of the ID; it is not the size of the sync word itself, unless the ID occupies the entire word. If you want the synchronizer to operate on an LSB-oriented word, you precede the size with a minus sign. If you want to force MSB-orientation, you precede the size with a plus. For example, if you have an frame counter on the mainframe that is transmitted MSB first, but the subcommutated data it synchronizes is delivered LSB first, your subframe declaration line might look like this

```
SUBFRAME 1 ID LEN=12 MIN=0 MAX=19 LSB SIZE=+5
```

Note that a "size" of five is correct for a maximum value of 19 (/13), and the plus sign indicates the sync word is MSB first, even though the subframe data is LSB.

If you do specify a mask or "size" in ID mode (JAM or counter) you should not program more '1's in the mask than the number of bits required to contain the value you program with the **MAX** specifier. The Compiler uses the maximum value to calculate the number of default program entries required. If you allow more bits into the ID register than the default entries accommodate, noise in the data stream may causes the program to mistrack.

Parity Bits

If the ID word has a parity bit, the *SF1* command would be, assuming an 8-bit word length and MSB data,

```
SF1 ID LEN=8 PA MSB
```

The **PA** and **MSB** parameters affect only the word processor. The Frame Synchronizer shifts not eight bits but nine bits to the ID Subframe Synchronizer. That doesn't matter if the parity bit trails the word (in time) because the trailing bits of the mask are 0 and the the parity bit will merely be thrown away. But if the parity bit is leading, it shifts to the synchronizer and it must be masked off. The Compiler does not deal with this situation, and you need to set the mask

yourself. For example, if the ID field is four bits wide, MSB first, in an 8-bit field with leading parity, the mask must be

0 0 0 0 0 1 1 1 1

If you express that in hexadecimal the *MASK* parameter would be

```
MASK=078
```

However, you can express the mask in binary and that is probably less error-prone and more readable.

```
MASK=000001111
```

If the ID word were LSB first, the mask would be

```
MASK=01111
```

Remember that you never have to express any bits of the mask to the right of the rightmost **1**.

Time Ordering in the Mask

The mask is always expressed in time order of the data stream. LSB data is assembled correctly in the ID register, but you must account for the time ordering of the mask. For example, suppose the ID value were bits 3, 2, and 1 of an eight bit word presented LSB first (bit 0 first in time order). You would need the mask 0111. (You could express it as 01110000 if you wanted to show all eight bits for documentary purposes.) The synchronizer would discard bit 0 and capture bits 1, 2, and 3 and load them into bits 0, 1, and 2 of the ID register.

You must consider the justification of the numbers when you express the pattern or mask in hexadecimal. You express the recycle sync and mask words *right justified* and express the ID mask word *left justified* in the 32-bit field. If, for example, a recycle subframe had a 10-bit sync pattern which was binary 1011010011, it would be expressed as 2D3. On the other hand if you want to express the same binary pattern as an ID sync word mask, you express it as B4C. We recommend that you express the sync and mask in binary. If you do that, the Compiler can justify the mask correctly without your having to think about. For the recycle mask, you don't express the leading zeroes, although they are harmless if you do express them. For the ID mask, you don't need the **trailing** zeroes. For either, you can express them if you wish, up to the 32-bit word size. Leading zeroes in the recycle sync pattern, however, will cause the default mask to have a '1' in every corresponding bit position.

CHAPTER 5
THE PCM FRAME DESCRIPTION

To describe a PCM format you construct a description list for the mainframe and a separate list for each of the subframes. The lists are sequences of lines of text that map the PCM format. You describe each word in the PCM format, word by word, in the time order of the serial data stream. It is easy to modify the decommutation of a specific word without disturbing the program for the surrounding words. If you omit a word in the frame, you can insert it without disturbing any of the surrounding programming.

The schematic of a line of in the list is

[*label*:] *statement*

The *label* is usually only used if the line is the target of a transfer of control statement, although you may want to use labels to help make the program self-documenting. An example of a line with a label is

```
SEQU19: LEN=12 MSB
```

A label may also appear alone on a line. Then it has the value of the next location generated by a code generating line.

You describe a data word in the PCM frame with statements that either declare the name or number of the data word to be output or define properties of the words to follow. Properties include such factors as length, orientation, that the word is subcommutated, or that the word is a sync word. You may also use control-of-flow statements, which will be described in Section 7. This section explains how you describe the properties of a data word and how you assign an ID tag to it.

DATA WORD DECLARATIONS

To define the position of a data word in the format, you declare the desired output ID for the decommutated word. The Compiler uses this declaration to generate an instruction that directs the Decommutator's Word Processor to construct the data word and transmit it to the Distribution System with the specified ID tag. You can output every data value with an arbitrary ID in the range 0-65,535.

You may declare data IDs by symbolic label or by numeric value. If you declare them by label you may define their numeric value or you may leave the definition to the Compiler. You may express an ID in one of the following forms.

nn	A decimal (or hexadecimal) integer which is the desired ID value.
*	The data is assigned an ID generated by the Compiler using an automatically incremented counter. Each time '*' is referenced it provides a value one greater than the previous reference to '*'. At any time the value of '*' can be assigned by a line that contains the statement *=*nn*, where *nn* is a number as defined above.
label(*nn*)	*label* is defined with the value *nn*.
label(*)	*label* is defined with the value of '*'. ('*' is incremented, as usual.)
label	If it is not already defined, *label* is given an automatically assigned value one greater than the output ID assigned to a previous label that was not defined using the '*' counter in the program list.

The Compiler always offsets the numeric values assigned to a label, adding the Port Offset that you specify on the line that introduces the programming for the port. For example, if the line

```
PCM /400
```

introduces the PCM port programming, then an ID expressed as

```
PRESS1(/20)
```

causes the label PRESS1 to have the value /420. Similarly the line

```
*=61
```

assigns the decimal value 1085 (/400+61) to '*'. This allows you to move the entire block of DITs assigned for a given port.

If you want to output an absolute ID, express the ID in the form

#*nn*

where *nn* is a number in the range 0-65,535. You cannot give a label an absolute value , nor can you assign an absolute value to '*'.

Once you define a label, you can use it throughout the remainder of the setup program, and it has the same value throughout. This is the easiest way to handle supercommutated data. If you always refer to a given measurement with the same label, it will always have the same numeric ID tag.

The Compiler distinguishes program labels from ID labels. A program label and an ID label may be the same character string.

Note that there are two separate conventions for generating automatic ID values. The first of these uses a counter called the ID counter. The Compiler sets the ID counter to a new value every time an ID is *assigned*. This happens when you mention a new ID symbol in the program. The new ID symbol has a numeric value one greater than the previously assigned one, unless you assign it a different value. Thus, unless you interfere, the counter counts by one for

each new symbol. (See the description of the **RPT** statement in this Section for exceptions to this rule.)

This allows you to assign all IDs, or the Compiler to assign all IDs, or you to assign the base ID of a block of IDs while the Compiler assigns the other IDs in the block.

The other assignment convention uses an independent counter, called the '*' counter. The Compiler increments that counter every time you reference it. If you use it to assign a value to a label, then the ID counter is *not* changed and only the '*' counter is incremented.

You can use this feature to assign a block of IDs to data from a particular data source. For example, you could give "Guidance Data" IDs in the 400's while giving "Flight Data" IDs in the 200's. To do this, you assign the first flight data label in the format description the ID value 200. Then you assign the '*' counter the value 400 with the statement

```
*=400
```

Then when you assign a Flight Data ID, you use the syntax *label*, and when you assign a Guidance Data ID you use the syntax k(*). If some attention is given to planning of the ID assignments, it may be easier to separate the data during analysis.

The Compiler outputs a Symbol Table listing of all ID labels and Program labels.

WORD Statement

You program data words of 32 or fewer bits in length with a WORD statement. The statement defines the position of the word in the format and assigns its output ID tag. The WORD statement has the schematic

WORD *id* [RTM]

where *id* is expressed in one of the forms described above.

Use the optional specifier, **RTM** meaning "Return To Mainframe," to select the mainframe program counter in the most efficient way. You need to do that when you program a subcommutated word where the next word in the minor frame is a mainframe word. Frequently in a highly subcommutated format adjacent words are subcommutated. The word descriptions in the subframe list for these words should not have the RTM specifier, except for the last one. When mainframe words lie between words in the subframe list, any word that is followed by a mainframe word should have the RTM specifier. Programming of subframes is described in more detail in sections to follow.

When you want to suppress the output of a data word, use the statement

WORD SUP [RTM]

It is equivalent to "WORD #131,071." ID 131,071 signals the Decommutator hardware to reject the data word, that is, not output it to the data stream at all. You also can use the **SUP** statement described in this Section of the manual to reject many sequential *bits* from the bit stream.

WORDS Statement

You can use the WORDS statement, which has the schematic:

WORDS *label*1 [RTM] *label*2 [RTM] *label*3 [RTM]... ..*labelnn* [RTM]

to declare a list of words on a single line. There is no difference in function between a **WORDS** statement with *nn* labels and *nn* separate **WORD** statements. Use whichever programming style seems better to you. You can specify Return To Mainframe (RTM) as often as necessary within the **WORDS** statement. For example, you may program

```
WORDS AAA BBB RTM CCC DDD EEE RTM FFF RTM GGG HHH III JJJ RTM
```

That sequence might describe a given minor frame's subcommutated words that occupy first two consecutive slots (AAA and BBB), then later three slots (CCC, DDD, and EEE), and still later, one slot, and finally, six slots.

CWORD Statement

You program conditional data word output with a CWORD statement. The statement defines the position of the word in the format and assigns its output ID tag based upon the state of one of the Compare Flags. (See Section 9 for a description of how the CMP and TST statements are used to set the Compare Flags.) The CWORD statement has the schematic

CWORD *cf id1* [*id2*] [RTM]

where parameter *cf* selects the Compare Flag (0-15). The data word is output with ID tag *id1* if the Compare Flag is true, otherwise *id2* is used as the tag. If you omit *id2,* the data is suppressed if the Compare Flag is false.

You can use the optional specifier, **RTM** meaning "Return To Mainframe," as described above. The RTM applies to *id1* and, if it is given *id2*. If *id2* is omitted and the Compare Flag is false, the data word will be suppressed, then the "Return To Mainframe" will select the mainframe program counter.

RPT Statement

The PCM Decommutator has a special facility to generate consecutive data words of the same form (the same word properties) with an incrementing ID. It does this with a single instruction that can generate up to 4095 sequential words. The schematic of the statement that programs this feature is

RPT *nn* [RTM] [*label*1 .... *labelnn*]

where *nn* (the repeat count) is the number of output words to generate. Although the hardware RPT instruction can repeat only 4095 times, you may specify any number less than 65,536. The Compiler generates the necessary code to produce the required number of words.

You *must* precede the **RPT** statement with a **WORD** statement that establishes the initial ID. If the initial ID is labeled, you may later reference the data generated by the **RPT** statement with a label of the form *label*#*nn*. A reference to "*label*#1" refers to the first word generated by the **RPT** statement. The **RTM** specifier causes the word following the last word generated by the **RPT** to be taken from the mainframe. Be sure to use this feature when a sequence of

consecutive words in the minor frame is subcommutated by the same subframe synchronizer. It saves both space in the memory and program execution time. That can be especially important when bit rates are high.

You may want to assign arbitrary labels to ID numbers created by a **RPT** statement. The Compiler assigns Sequential IDs to the list of labels that optionally appears in the statement. An **RTM** specifier can appear anywhere in the list; its position does not affect how the instruction behaves. An example of an **RPT** statement that assigns labels is

RPT 4 PRES1 VALV3A PRES2 VALV3B

The label list need not define all the ID values, but the Compiler assigns values starting with the first ID output by the RPT instruction. Of course it is illegal to attempt to assign a new ID value to one of the labels, that is, the form

label(*id*)

cannot appear in list.

If the **RPT** statement follows a **WORD SUP** statement, the Compiler generates code to suppress the number of additional words specified by the repeat count. A label list is illegal in that kind of **RPT** statement.

For certain types of subframes you must return to the mainframe following each subcommutated word. You can use the **RPT** statement to do this, also. The hardware RPT instruction can't do it, but the Compiler generates the code you need to perform the function. You express this with the program schematic

WORD id RTM
RPT *nn* [*label*1 .... *labelnn*]

The Compiler generates the required number of WORD instructions, each with the RTM bit set. This syntax does not conserve PCM memory as other uses of the **RPT** statement do, but it is syntactically condensed, compared to writing out *nn* **WORD** statements.

When you use the **RPT** statement, the ID generating mechanism in the Compiler increments the ID assignment counter by the number of IDs generated. Which counter it increments depends on the form of the **WORD** statement that is repeated. If it is

`WORD *`
or
`WORD` *label*(`*`)

the Compiler increments the '`*`' counter. If it is

`WORD` *label*
or
`WORD` *label*(*nn*)

it increments the ID counter. If it is

`WORD` *nn*

neither counter is incremented.

DUP Statement

The **DUP** statement is similar to the **RPT** statement. Like the "repeat" instruction, the Advanced PCM Decommutator has a special facility to generate consecutive data words of the same form (the same word properties), but with the same ID instead of incrementing IDs. It does this with a single instruction that can generate up to 4095 sequential words. The schematic of the statement that programs this feature is

DUP *nn* [RTM]

where *nn* (the duplicate count) is the number of output words to generate. Although the hardware DUP instruction can repeat only 4095 times, you may specify any number less than 65,536. The Compiler generates the necessary code to produce the required number of words.

You *must* precede the **DUP** statement with a **WORD** statement that establishes the ID of the word to be replicated.

LWORD and LWORDS Statements

In older TDPs, those that predate the VME version, the Word Processor output Words longer than 16 bits as two 16-bit syllables. You used the **LWORD** statement for output of long words so the Compiler would allocate two IDs for output. The VME PCM Decommutator can output 32 bits in a single transfer, so there is no longer any need to differentiate between PCM words that are 16 bits or less, and those that are longer. To maintain compatibility with old setup programs, the TDP Compiler continues to recognize statements, but treats them no differently than a **WORD** statement.

SUP Statement

The **SUP** statement efficiently suppresses a sequence of undesired data in the stream. The statement has the schematic

SUP *nn* [RTM]

where *nn* is the number of *bits* to reject from the data stream. Use the **RTM** specifier to return to the mainframe. You *must* reset the word length after the **SUP** statement. (Use, for example, the **NWD** statement, as explained in this Section of the manual). Accordingly, if you use the **RTM** specifier, you must set the correct word length at the point returned to on the mainframe.

If, for example, the data stream had 317 consecutive bits of data that were of no interest, the statement

SUP 317

causes them to be thrown away entirely. The statement may be used at the subframe level as well as on the mainframe. Remember, however, that the statement works on contiguous bits of the stream. On a subframe you must account for interspersed mainframe words. That is, a "SUP 32" cannot suppress two 16-bit words that are not adjacent in time, even though they may be adjacent in the subframe list.

NAMES STATEMENT

If a format is likely to change frequently, the ID assignments that the Compiler makes based on where the symbol occurs in the program may not be convenient. Instead, you can use a declared list of symbols to assign the ID tag values. The statement schematic is

$$\text{NAMES } id_1\ id_2 \ldots ..id_n$$

where id_j is any valid label definition

label
label(*n*)
k(*)

The Compiler assigns the ID values in ascending order as listed. For example:

```
NAMES XY(5) XZ AAA BB(15) CC(/23)
```

assigns the values as follows:

SYMBOLIC NAME	ID VALUE
XY	/5
XZ	/6
AAA	/7
BB	/f
CC	/23

Any number of **NAMES** statements may be used. You may place them anywhere in the Format program or in the Distribution program, however, a **NAMES** statement that defines a symbolic ID must precede the first use of that ID in a **WORD** statement. The recommended program organization puts all **NAMES** statements at the beginning of the program immediately following the **PCM** statement.

WORD PROPERTIES

The properties of a PCM data word include word length, orientation, and presence or absence of a parity bit. These properties are individually programmable for each data word in the format, regardless of commutation level. A description line that sets the properties precedes the declaration of the word or words that it controls. The symbols that describe these properties are called "definers." The definers are

LEN=*nn*	defines the data word length.
LSB	defines the orientation as Least Significant Bit first.
MSB	defines the orientation as Most Significant Bit first.
PA	defines the data word to include a parity bit.
NPA	defines the data word to include no parity bit.
NWD	defines the data word to have default properties.

On older Acroamatics PCM Decommutators, word properties were set with a hardware instruction. The remained set until another instruction changed them. This meant that word property settings followed the execution flow of the program. With the Advanced PCM Decommutator, however, the word properties are part of every output instruction, so when you declare the properties in your

setup program, those properties apply to the following **WORD** statements. They remain in effect until the next properties definer *is compiled.* If you have programs written for older PCM Decommutators, you may need to make some modifications to them so they will run on the new machine. The differences between the 1502/1602 card and earlier ones, and the changes you may need to make in existing programs are discussed in detail in Appendix A of this manual.

To set the properties for the data output words which follow in your program, you declare them all on a single line of setup text. A typical line might be

```
LEN=8 MSB
```

This says that the words following are 8-bit MSB oriented words. For each WORD instruction that follows, the Word Processor takes 8 bits from the PCM stream, orients them as MSB first, and outputs them in a data word.

The Compiler uses the defaults established at the beginning of the current program segment to supply any of the properties that you don't explicitly mention in a statement. For example, suppose you were describing a word in the mainframe list and on the **MAINFRAME** line default values of LEN=16, MSB, NPA had been given. Then when you write

```
LEN=6
```

the Compiler compiles it as though you had written

```
LEN=6 MSB NPA
```

For that reason you must take care when you program any atypical word properties that all the properties are set on a single line. If, for example, the **MAINFRAME** defaults were as described above, but a word in the frame is LSB oriented and 8 bits long, the correct program for that word is

```
LEN=8 LSB     |Correct
```

not

```
LEN=8         |Not Correct
LSB
```

The incorrect example compiles as if written

```
LEN=8 MSB NPA
LSB LEN=16 NPA
```

That leaves the Decommutator set according to the last instruction executed, and that's the wrong word length.

The following sections describe the various word property definers.

LEN - Change Word Length

The definer

LEN=*nn*

establishes the word length for succeeding words. The value of *nn* may be in the range 1 to 32.

MSB, LSB - Word Orientation

The orientation definers establish the orientation for succeeding words. They are, of course, mutually exclusive, but the Compiler doesn't detect an error if you try to use them both. It uses the last one on the line.

NPA, PA - Parity Control

You can specify the presence or absence of a parity bit individually for each word in the format. When you program the word the have a parity bit the Word Processor strips from the data and checks it. (You must also specify even or odd parity in the SYS statement.) If you specify **DIP** in the **SYS** statement (see Section 3), the Word Processor will not strip the parity bit. Instead, it outputs it with the data. If the data has such inconveniences as 16-bit words with byte parity, you can use the syllable assembly features of the Distribution system to reassemble the bytes into words. Note that you don't include the parity bit in the length specified for the word. Thus both

```
LEN=8 PA
```

and

```
LEN=9 NPA
```

process 9 bits of the data stream.

NWD - Normal Word

The NWD definer reestablishes the default word length, orientation, and parity selection without having to write them out explicitly. For example

```
LEN=9 LSB
WORD    COUNT1
LEN=3 MSB
WORD    MOD1
NWD
WORD    FFLOW
```

The NWD definer restores the normal word default length and orientation that you defined at the beginning of the program segment.

OUTPUT DATA JUSTIFICATION

You can program the Word Processor to either right- or left-justify the data in the 32-bit output data field. Words that are 16 or fewer bits in length are justified in the 16 LSBs of the 32-bit output word, with the sign extended appropriately into the high half of the word. Words that are longer than 16 bits are justified within the 32-bit word.

The PCM Decommutator always starts with left-justified output. It must execute an instruction to change the justification. Setting the output data justification is separate from setting the word properties described above, and one does not affect the other. Unlike setting the word properties, however, setting the output data justification is an actual machine instruction. It establishes the operating mode of the machine until another instruction changes it.

There are two forms of right-justification supported:

RJL Right Justify Logical
RJA Right Justify Arithmetic

The **RJL** statement makes the Word Processor shift the data to the right end of the 32-bit output word, zero filling from the left. The **RJA** statement makes the Word Processor propagate the most significant bit of the output word as the data is shifted right. This is the appropriate treatment for 2's complement data.

When you use a Right Justify Arithmetic, if you specify the parameter **SIN**, the sign bit will be inverted before the data is shifted right. This is the appropriate treatment for offset binary data.

If you have used either of the right-justification output modes for part of the frame, and you want to return the output to the left-justified mode, use **LJ** statement.

You can see the effect of these modes in the following example of a 12-bit data word. If its data value in binary were "1001 0101 0111," the output from the four different modes would be

MODE	OUTPUT WORD
LJ	1111 1111 1111 1111 1001 0101 0111 0000
RJL	0000 0000 0000 0000 0000 1001 0101 0111
RJA	1111 1111 1111 1111 1111 1001 0101 0111
RJA SIN	0000 0000 0000 0000 0000 0001 0101 0111

Observe that **RJA** extends the sign bit to the left and that **RJL** does not. **RJA SIN** inverts the sign bit (to a zero, in this case) before it is extended.

If you had a 24-bit data word with the value "1111 1010 1111 0011 0010 0001," the output from the different modes would be

MODE	OUTPUT WORD
LJ	1111 1010 1111 0011 0010 0001 0000 0000
RJL	0000 0000 1111 1010 1111 0011 0010 0001
RJA	1111 1111 1111 1010 1111 0011 0010 0001
RJA SIN	0000 0000 0111 1010 1111 0011 0010 0001

SYNC - A RECYCLING SYNC WORD

SYNC means that the next word output is the last (or perhaps only) syllable of the sync pattern. You use this specifier for a recycle sync word as well as for the mainframe sync word. An example of its use might be

```
SYNC  LEN=24
WORD MFSW
```

This example defines and outputs a 24-bit sync word. The instructions that the Compiler generates from this program tell the Decommutator that the current position in the PCM stream should contain the programmed sync pattern.

Note that the example assigns the sync word an arbitrary symbolic ID (MFSW). If it would be more useful to break the sync word into two 12-bit words, the statements

```
LEN=12
WORD MFSW1
SYNC LEN=12
WORD MFSW2
```

would accomplish that. The effect on the synchronizer is the same in both cases, however, with only the output format changed. If the sync word is longer than 32 bits, you must break it up into syllables, as shown above.

Note that a WORD statement is *required* after the sync word declaration, to tag and output the sync word. Of course the statement can be

```
WORD SUP
```

if you don't want to output the sync word.

The **SYNC** statement is another word properties-setting command, and the rules discussed in that section all apply to it. The Compiler obtains any properties not explicitly specified in the statement from the defaults for the current program segment. For example, suppose a format were transmitted as 8-bit words plus parity, and those were the defaults declared in the MAINFRAME line. Further suppose that the sync word were defined as an 18-bit word (two words of the frame, including their parity bits). Then the correct programming would be

```
SYNC LEN=18 NPA    |Correct
WORD MFSYNC
```

not

```
LEN=18 NPA         |Not correct
SYNC
WORD MFSYNC
```

The **SYNC** statement in the incorrect example resets the length to 8 bits plus the parity bit (the defaults). Remember, also, that although you can specify the orientation of the sync word that changes only the orientation of the output word. It does not affect what the synchronizer sees which is the stream of bits with the most significant bit first in time-order.

Because the program specifies the sync word position, different paths through the decommutation sequence can allow the minor frame length to vary without loss of sync, should a format require that.

You also mark the position of the recycle sync word for recycle subframes with **SYNC**. This is discussed in Section 9, *RECYCLE SUBFRAMES*.

CHK - A CONDITIONAL SYNC WORD

CHK means that the next word output may be the sync word, but it does not have to be. You can use this for special formats in which the length of the minor frame varies. This command tells the synchronizer that *if* the current word matches the pattern, it should reset to the beginning of the frame program, that is, to the *base*. If it does not match, the program continues. Section 9 explains how you can use the **CHK** specifier to program a variable length minor frame.

In the next section you can read about the **RCY** specifier that you use with subcommutated words to tell the subframe synchronizer that the next word position *may* contain the pattern for a recycle subframe. There you will also read about the **SYL** specifier that declares that the next word is a piece of the pattern. Note that you cannot use that specifier with the mainframe pattern. The mainframe pattern *must* lie within a range of 64 bits so that the mainframe correlator can find it initially. On the mainframe you use the **CHK** specifier rather than the **RCY** specifier because we thought it made the program more understandable. The two specifiers set the same bit in the PCM instruction.

CHAPTER 6
SUBCOMMUTATED WORDS

The Decommutator has the equivalent of separate processors for subcommutated words. You program up to six of these processors, each with a separate program segment. In your program you assign a subframe synchronizer to decommutate all the words that a particular subcommutator controls. You make a separate list of the subcommutated words for each subframe synchronizer. Each subframe list contains only the data words identified by reference to that subframe's synchronizer. You use an **SF***n* statement in the mainframe list (or in other subframe lists) to declare the position of series of one or more subcommutated words. When you execute the compiled program, separate program counters keep track of where the Decommutator is in each program segment. The instruction compiled from the **SF***n* statement directs the Decommutator to use program counter *n*. Then the Decommutator executes instruction from the corresponding list until an RTM instruction returns it to the mainframe counter or another SF instruction changes to a different subframe counter.

SF*n* - SUBCOMMUTATED POSITION DESCRIPTION

Use the **SF***n* declaration when the next word (or series of words) in the format is subcommutated, and the subcommutation program is in the list for subframe *n*. The schematic of the statement is

SF*n* [*modifiers*] [*definers*]

where *n* is the subframe number, *modifiers* define the nature of the next word in the frame, and *definers* specify the word properties of the next word of the current program segment. If the **SF***n* declaration transfers control to another program segment, the word properties *are not* necessarily set according to the defines of this declaration. When given without modifiers, the statement simply selects the specified subframe, usually to continue an already established subframe sequence.

The *modifiers* specify that the next word in the frame is a sync word, part of a sync word, or is the beginning of an ID synchronized subcommutation sequence. The *modifiers* and what they identify are

ID	An ID sync word (or its final syllable)
RCY	A recycle sync word (or its final syllable)
ID SYL	A syllable of an ID sync word
RCY SYL	A syllable of a recycle sync word
ENTER	The first entry to an ID subframe
ENTER AT	The first entry to ID subframe having multiple entry lists.

They are discussed below.

The **SF***n* statement, except when the **ENTER** modifier is used, is a word properties-setting statement. As always with such statements, the Compiler takes any properties not specified from the defaults of the *current program segment*

(not the program segment of the selected subframe). The modifiers, except for **ENTER**, tell the synchronizer to look for its pattern or ID counter value in the current frame position. When you use the **SF*n*** statement to do that, the **LEN** definer (whether expressed or defaulted) tells the subframe synchronizer how many bits are used in the sync pattern or ID counter.

ID - the Position of an ID Sync Word

The **ID** modifier says that the following word is an ID sync word. Since the ID sync word is not itself a subcommutated word, the **SF*n*** statement does not select the subframe, and the ID sync word is output on the current program level. A typical definition of an ID sync word position might be

```
SF2 ID LEN=6
WORD  SUB2ID
```

These statements prepare subframe synchronizer 2 to verify the ID sync word using the next six bits of data, then to output the data with the symbolic ID "SUB2ID."

Sometimes the sync word is split, scattered among words that are not consecutive in the frame. Use the **SYL** modifier to send a syllable of the sync word to the appropriate synchronizer, where it is held until the final syllable arrives. For example, if three bits of an 8-bit sync word were in one word of the frame and the remaining five bits in a later word, the statements

```
SF1 ID SYL LEN=3
WORD  SFSYL1
     .
     .
     .
SF1 ID LEN=5
WORD  SFSYL2
```

would concatenate the sync syllables correctly. The synchronizer would not verify the ID value until the output of "SFSYL2."

RCY - the Position of a Recycle Sync Word

The **RCY** modifier declares the subcommutated word position where the recycle sync word will appear in some minor frame. For example, if word positions 114 through 120 are subcommutated, the mainframe program might contain the statements

```
WORD  113
SF2 RCY
WORD  121
```

That says that the word slot following word 113 on the mainframe is subcommutated *and* is the position in which the recycle word for subframe 2 will appear. If the format is viewed as a matrix with a minor frame occupying every row, the column representing word position 114 contains the sync word in the last minor frame. The specific definition of the actual sync word position occurs in the subframe program segment with a **SYNC** statement.

When the subframe is in SEARCH, the Decommutator cannot track the subcommutation program because it can't determine the current minor frame until the sync pattern is seen the first time. The **SF*n* RCY** statement tells the synchronizer where to look for the pattern when the subframe is still in SEARCH. When the pattern is found the synchronizer leaves SEARCH and begins tracking the program. Then the **SYNC** statement in the subframe list tells the synchronizer when to expect the sync pattern again.

A recycle sync word may also be split into syllables that are not consecutive in the frame. Use the **SYL** modifier to send a syllable of the sync word to the appropriate synchronizer, where it is held until the final syllable arrives.

Because the recycle sync word (or a syllable of it) is a subcommutated word, when you use the **RCY** or **RCY SYL** modifier the subframe is selected, and the output of the word occurs according to the subframe list.

ENTER - Entering an ID Subframe

To enter an ID subframe in the correct minor frame requires a transfer to a subframe program at a location determined by the frame number contained in the PCM data. This data may be a counting ID that the synchronizer tracks and corrects, if necessary, or it may be a jammed frame number such as a computer generated format might contain. This entry normally occurs at one point in the format, usually at the first ID subcommutated word position in the minor frame. Once the subframe has been entered at that point, other subcommutated words are programmed in the same way that a recycle subframe is programmed.

Use the modifier **ENTER** to effect this entry. The schematic of a description that enters an ID subframe under control of the ID value is

SF*n* ENTER [AT *label*]

The optional **AT** *label* specification lets you enter the subframe list one of several labeled entry points. This feature allows different decommutation sequences for an ID subframe, based on dynamically determined conditions in the data. For example, if a word in the format indicates which phase of a test is in progress, compare instructions (**CMP**) and conditional jumps (**IFT**) could jump to different **SF ENTER AT** statements to decommutate the stream differently for each test phase.

RT*n* - **RETURN TO SUBFRAME**

When a subframe list is executed, the Decommutator must return to the Mainframe whenever a mainframe word or words lie between two words on the subframe. When there are deep levels of commutation, such as for sub-subcommutated data, either mainframe words or words in the frame one level higher may follow a sub-subcommutated word. To return control to a specified frame you use the **RT** statement whose schematic is

RT*n* [*definers*]

In this declaration, *n* is a number between 1 and 6, specifying the subframe level to return to or is an **M** to return to the mainframe level. *definers* set the word properties of the next word at the current commutation level. The Compiler sets any properties you don't specify to the default properties of the *current* program

segment, not those of the segment to which you are returning.

Because return to the mainframe from a subframe is the most common requirement, the decommutation instruction that assigns ID tags has a flag bit that causes an RTM after the associated data word is output. This saves an instruction when you need to do an RTM. How you do this has already been described in Section 5, but to review, you add the modifier word **RTM** to a **WORD** statement. For example

```
WORD 222 RTM
```

There is no operational difference between this statement and the statement pair

```
WORD 222
RTM
```

except that the latter example is a two-word instruction, and, of course, it takes longer to execute.

If you are returning to a subcommutated word position from a sub-subcommutated position, you use an **RT** statement that specifies which subframe you are returning to. For example, in a sub-subframe:

```
LEN=8
WORD 641
WORD 659
RT2
```

would do two words of 8 bits each, then returns to subframe 2.

CHAPTER 7
PROGRAM FLOW STATEMENTS

Program flow statements let you to introduce decision and control functions into the frame description. You can transfer control unconditionally to a different part of the format program, and you can call and return from program subroutines. You can test the value of the last word output and, based upon the results, you can transfer to a different path in the format description. This section explains the statements you use to do these things.

GOTO STATEMENT - UNCONDITIONAL TRANSFER

Use the **GOTO** statement to transfer control to a specified location in the format program. A typical use for this statement is to transfer to a common sequence after following a sequence entered conditionally. The statement schematic is

GOTO *label*

where *label* is a program label in the current program segment.

CMP, TST AND IFT - CONDITIONAL TRANSFERS

These statements allow you to test a value in the data stream and make a real-time decommutation format decision.

The **CMP** and **TST** statements both serve the same function: to test the value of the last word output by the Word Processor, and to set a Flag if the tested conditions are met. Then you use a conditional branch to transfer control to a different decommutation sequence.

The **CMP** statement generates an instruction that matches a 16-bit field in the PCM data word to the value specified in the statement (you can test the high or low half of a word that is longer than 16 bits). If it matches, the Decommutator sets the Compare Flag "True". You use the **TST** statement to check the state of a single bit in the data word. It sets the Compare Flag if the bit is 1. The Compare Flag remains set until the next **CMP** (or **TST**) statement for that Flag occurs. This means that you can test the value of a data word and use the results of that test repeatedly throughout the format to make path decisions. There are sixteen independent Compare Flags that can retain the results of sixteen separate tests.

The comparison field is 16 bits wide. The statement schematic is

CMP[*hw*][*cf*] *nn*

where *nn* is the value that the Decommutator matches to the 16-bit output word. The optional parameter *cf* selects the Compare Flag (0-15) to be set. If you don't specify *cf*, the Compiler selects Compare Flag 0. The option character *hw* specifies whether you are testing the high or low sixteen bits ("H" or "L") of a PCM word that is longer than 16 bits. If you omit *hw*, the comparison filed is matched with the low half, and that is also the correct way to compare data

words 16 bits or less in length.

When you compare left-justified data where the data word size is smaller than the comparison field width (16 bits), add trailing zeros to the comparison value to make 16 bits. For example, if you compare an 8-bit left-justified word, and you want to know when it is /1B, you should compare it to /1B00, not /001B.

The **TST** statement is similar to the Compare statement. Its schematic is

TST[*hw*][*cf*] *nn*

where *nn* is the bit number being tested (0-15, bit 15 referring to the most significant bit), and the optional parameter *cf* selects the Compare Flag (0-15) to be set. The option character *hw* specifies whether you are testing the high or low sixteen bits ("H" or "L") of a PCM word that is longer than 16 bits. You can perform More than one comparison or bit test on a single word and save the results in different Compare Flags. For example,

```
LEN=5
WORD     TSTWRD
CMP0     /F800
TST1     11
TST2     15
```

tests a 5-bit word, 'TSTWRD', checking for

CMP0: all bits on,
TST1: the least significant bit on, and
TST2: the most significant bit on,

and saves the results in, respectively, Compare Flags 0, 1, and 2.

To use the results of a data test (**CMP** or **TST**) you use a conditional branch statement. If the state of the Compare Flag is True, the conditional branch statement behaves exactly like a **GOTO** statement. If the Compare Flag is False, the Decommutator executes the next sequential instruction.

The simplest of the conditional branch statements is the **IFT** statement (IFT is mnemonic for "IF True"). The statement schematic is

IFT *label*

where *label* must be a program label in the current segment. The **IFT** statement tests the condition of the *last Compare Flag selected* and is therefore most useful for immediately testing the results of a **CMP** or **TST**, or when a decommutation program uses only one Compare Flag. The "last compare flag selected" refers to the compilation sequence, not the execution sequence. When you refer to a compare flag in a **CMP** or **TST** statement, the Compiler remembers the flag number. Then, when it compiles an **IFT** statement with no flag number specified, it supplies the remembered flag number. The scope of this remembered value is until another **CMP** or **TST** statement changes it.

You may also address a specific flag by using the **IF***cf* statement. The schematic for this statement is

IF*cf label*

where *cf* specifies the Compare Flag (0-15) which determines the branch decision. Using the results of the tests on 'TSTWRD' shown in the example above,

you could make format decisions using the following typical statements.

```
IF0     ALLON
IF1     LSB1
IF2     MSB1
```

When these are executed, the Decommutator branches to the label ‘ALLON’ if all bits of the tested word are on. It branches to ‘LSB1’ if the least significant bit is set, and it branches to ‘MSB1’ if the most significant bit is set. If none of these conditions are met, it executes the instruction following the **IF2** statement.

You can also use the **CWORD** statement to do a conditional data word output based upon the state of a Compare Flag. See Section 5 to learn how to use this statement.

CALL STATEMENT - SUBROUTINE CALL

Sometimes sequences of commutation are repeated within a format. You can save programming and Decommutator memory by treating these sequences as subroutines. Subroutines may be used whenever you need an identical sequence at more than one place on a frame. You can nest subroutine calls to a depth of four (a pushdown stack is used). You can use subroutines in subframe lists as well as on the mainframe. A typical example is an ID subframe in which an entire line is supercommutated at several values of the ID. Use the **CALL** statement to transfer control to a subroutine. The schematic of the statement is

CALL *label*

where *label* must be a program label in the current segment.

It is best that subroutine sequences called on subcommutated levels not span mainframe words. Similarly subroutines called on sub-subcommutated levels should not span subcommutated words or mainframe words. This means that you will not normally ever use an **RTM** or **RT***n* specifier in a subroutine. If you follow this rule, the subroutines may contain subcommutated words without danger of tangling the subroutine calling pushdown stack.

RET - RETURN FROM SUBROUTINE

Use the **RET** statement to conclude a subroutine reached by the **CALL** statement. The statement is simply

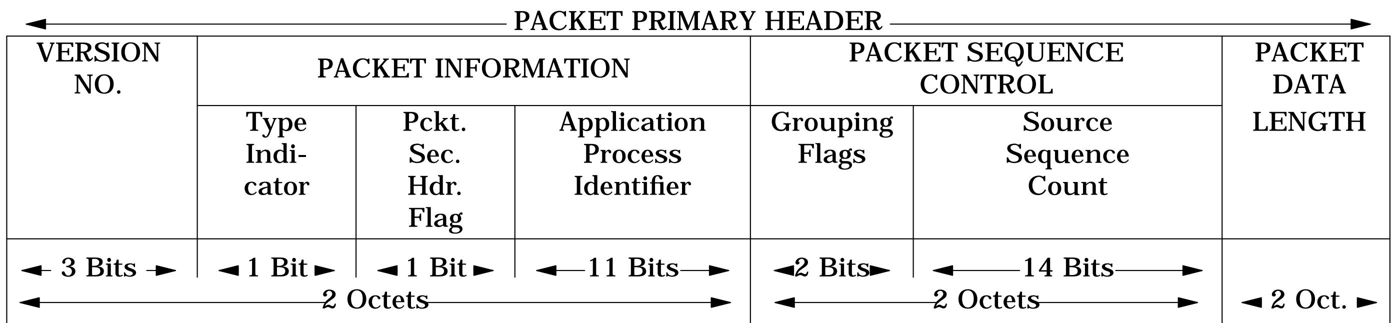
RET

The "Return" transfers control to the first instruction following the **CALL** statement that took the program to the subroutine.

CHAPTER 8
CCSDS PACKET DATA PROCESSING

To aid processing CCSDS packet data, several new instructions have been added to the 1502V and 1602P PCM decommutator cards. These instructions allow you to track the packets through the PCM frame, identify the packet messages, and to uniquely ID tag the data words from the packets of interest.

CCSDS packets consist of a Packet Primary Header followed by a variable number of data words.



The header is made up of three 2-octet words (an octet is an 8-bit data element). The 11-bit Application Process Identifier (which we will call the "Message ID"), in the first header word identifies the packet (and consequently its contents). The third header word has the 16-bit "Packet Data Length" field. According to the CCSDS Recommendations for Space Data System Standards on Packet Telemetry (CCSDS 102.0-B-5 Blue Book), the Packet Data Length contains the "number of octets of Packet Data Field minus 1".

PACKET DATA IN THE PCM FRAME

Typically certain words within a PCM format are allocated for packet data words, with some method provided for identifying the start of a Packet Primary Header (*e.g.*, a certain word in the frame will be always contain the first word of a packet header). The packet data, then, is a sequence of words: header, data words, header, data words, and so on.

After synchronizing to a header by whatever method is provided in the format, processing packet data normally consists of the following steps:

1. Identify the packet, usually with a "JAM ID" subframe. If it is a packet to be processed, load one of the CCSDS ID Tag Registers with the base ID of a block of tags allocated for that packet's data.
2. Extract the packet length with a subframe synchronizer, and load one of the Packet Length Registers with the subframe synchronizer's data.

3. Count the packet data words using the Packet Length Register. Tag and output the data words with the CCSDS ID Tag Register.
4. When the Packet Length Register has counted down to zero, process the next Packet Frame Primary Header (Step 1).

SUBFRAME SETUP

To use a subframe synchronizer to extract a packet length from a word in the format, you set up the synchronizer in "EXT" mode. When used to set up a synchronizer to extract the packet length, the only parameters that mean anything are the mask and orientation. The simplified subframe declaration syntax for this mode is

SUBFRAME *n* EXT MASK=*mmmm* [LSB]

No subframe processing is allowed for subframes used in "extract" mode.

EXTRACT THE PACKET LENGTH

You use a "Reference Subframe" instruction to extract the packet length from a word in the format. You could use the "`SF`***n*** `ID`" syntax to accomplish this, but you may prefer the new syntax,

SF***n*** LD

instead, as a more clear representation of what the program is doing. That statement indicates that the next word output will be loaded into the specified subframe synchronizer.

LOAD THE PACKET LENGTH

Having extracted the packet length with one of the subframe synchronizers, you need to load the value into one of the Packet Length Registers. You can load the register with the value, or with the value plus one. The syntax you use is

LPL*r* SF*n*[=]

where *r* is the length register you are using (0-15), and *n* is the number of the subframe synchronizer you used to extract the value. The optional "=" indicates that the length register is loaded with the exact value extracted. If the "=" is omitted, the value loaded is the extracted value plus one. The 16-bit "Packet Data Length" field is specified to contain the "number of octets of Packet Data Field minus 1", so you will usually load the length register with one plus the value.

TEST THE PACKET LENGTH REGISTER

You use a statement similar to the **IF** statement, discussed in Chapter 7 of this document, to test whether or not a Packet Length Register has been decremented to zero. The schematic for the statement for the "Test Length Zero and Branch" statement is

TLZ*r label*

where *r* is the length register you are testing (0-15), and *label* is a program label in the current segment.

The **TLZ** statement tests a length register for zero, and if true, transfers control to the program label you specify. If the length register is not zero it is decremented by one, and the following instruction is executed.

LOAD THE TAG REGISTER

The ID Tag Register can be used to generate a series of sequential ID tags when you output packet data words. You load one of the tag registers with a base ID, then the output instruction determines whether or not the register is incremented. The schematic for the "Load Tag Register" statement is

LTR*r tag* [*count*]

where *r* is the tag register you are using (0-30), and *tag* is the base ID to be loaded into the register. You can express the tag as a numeric ID, or use a symbolic label, as described in Chapter 5 of this document. We encourage you to use symbolic labels.

Normally you will use the tag register to output packet data words with sequential tags. This means that you will use a block of ID tags for that packet. You can declare all the symbols for that block with a **NAMES** statement. For example, if a packet contains four data words, the following code fragment could be used to reserve the block of tags, give each packet word a label, and initialize the tag register.

```
NAMES   P27W1(/400) P27W2 P27W3 P27W4

LTR1    P27W1
```

The symbolic label **P27W1** is assigned a numeric ID of hexadecimal 400, with the other three symbols assigned the next sequential ID values. Tag register **1** is loaded with the "base" ID of the block (/400, in this case).

Another way of accomplishing the same thing is to allow the Compiler to create the block of symbols for you. You do that with the *count* parameter of the **LTR** statement. You express the *label* parameter in the usual way, followed by a count. The Compiler will generate "count" labels, with the numbers 1 through *count* appended to the end of the label. The ID tags would be assigned sequentially. Using our earlier example, if you wrote the **LTR** statement as follows,

```
LTR1    P27W(/400) 4
```

you would get exactly the same results as before, when you used the **NAMES** statement.

If you want labels where the "count number" is somewhere other than the end of the label, you can use a percent character within the symbolic label, and that character will be replace by the numbers 1 through *count*, (only one percent character is allowed in a label for purposes of count substitution). For example, the statement

LTR1 W%P27(/400) 4

would generate symbolic labels **W1P27**, **W2P27**, **W3P27**, and **W4P27**, with numeric ID tags hexadecimal 400 through 403 assigned to them.

If you do not give the *count* parameter, a symbolic label is treated normally. The Compiler looks the label up in the symbol table and uses its numeric ID if has been defined, otherwise it defines it. It is the programmers responsibility to make sure if a block of tags is required that the sequence available from the starting ID is adequate.

OUTPUT FROM THE TAG REGISTER

When you are processing packet data words, instead of using a **WORD** statement to output the data, you use a **TAG** statement. Its schematic is

TAG *r* [HOLD] [RTM]

where *r* is the tag register you are referencing (0-30), and the option **RTM** parameter means Return to Mainframe after the output. If you do not give the **HOLD** parameter, the tag register value is incremented after the data word is output, otherwise it is held at its current value. Usually you will want to assign incrementing consecutive tags to the packet data words.

CHAPTER 9
FORMAT PROGRAMMING

This section explains how you use the language elements described in the previous sections to build a description of a PCM Format. It also describes some additional terms you need to program ID Subframes.

PROGRAMMING THE MAINFRAME

In the format programming examples in the sections that follow, we often use symbolic names that are related to word and frame numbers. These are simply arbitrary names, which we only use as examples rather than inventing "Measurement Names." You are free to use any ID tag values or names you want in your programs.

An example mainframe might be a 66-word format of 12-bit data words, including a 24-bit sync pattern. The program could be:

```
SYS 3BIT AUTO
MAINFRAME LEN=12 SYNC=/FAF320
WORD W1
RPT 63
SYNC LEN=24
WORD W66
```

> Note that, unlike earlier model PCM Decommutators, you *do not* have to explicitly establish the word properties of the first word in the frame. If you do not specify otherwise, the first word will have the default properties established for the typical mainframe word.

If a recycle subframe were added to the format, occupying words 9, 15, and 23 - 27, with the recycle sync located in subcommutated word 15, the mainframe part of the program would then be:

```
MAINFRAME LEN=12 SYNC=/FAF320
NWD
WORD W1
RPT 7
SF1                   | WORD 9
NWD
WORD W10
RPT 4
SF1 RCY               | WORD 15
NWD
WORD W16
RPT 6
SF1                   | WORDS 23 - 27
NWD
WORD W28
RPT 36
SYNC LEN=24          |Next word is SYNC
WORD W66             |Sync word symbolic ID
   (subframe description)
    .
    .
    .
```

VARIABLE LENGTH MAINFRAMES

Occasionally you may want to process a PCM format where the minor frame length is not a fixed number of bits. The TDP's PCM Decommutator can handle this kind of format because it does not have a fixed frame length counter as most PCM decommutators do. By controlling the flow of the program you can shorten or lengthen the number of bits before the sync pattern recurs. Some variable length formats have an indicator word in the frame that tells you what the format of the frame following it is. You can test this word using the Compare instruction to set flags corresponding to each of the formats. Then you branch on the flag state to the appropriate sequence to decommutate the upcoming segment of the frame. Another way to do this is to use the JAM ID mode of a subframe synchronizer. You load the synchronizer with the Format ID word then treat the variable part of the frame as sub-commutated by the Format ID. Each of the subcommutated sequences can be of different length, and that accounts for the format length changes.

A more inconvenient type of variable length frame has no Format ID. Instead it has fill-words, or even data words that may or may not be present, up to some maximum number. A typical example might have from 0 to 6 fill-words. You use the **CHK** specifier to tell the synchronizer that a word might be the pattern, but not to go to SEARCH if it isn't. For the example format you would program that specifier for the six possible fill-words:

```
   .
   .
   .
CHK LEN=24        |Assuming a 24-bit pattern
WORD    FILLW
CHK LEN=24
WORD    FILLW
CHK LEN=24
WORD    FILLW
CHK LEN=24
WORD    FILLW
CHK LEN=24
WORD    FILLW
CHK LEN=24
WORD    FILLW
SYNC LEN=24
WORD    SYW
```

At each 24-bit word that can be the pattern, the synchronizer checks to see if it *is* the pattern and resets the program if it is. At the last word position, the **SYNC** specifier requires the pattern, so, if it doesn't find the pattern here it will count the strategy counter If that has reached 0, the synchronizer will go to search. Flywheeling in this kind of PCM frame doesn't make much sense, so you will probably want to set the lock-to-search count to 1.

An even worse format can have an unknown number of fill-words or even an unknown number of fill bits. You can handle this with a loop at the end of the frame program:

```
     .
     .
     .
SYNWT:  CHK LEN=16    |Assuming 16-bit fill-words
        WORD SUP
        GOTO SYNWT
```

The Decommutator hangs in this loop until it finds the pattern. Then it resets and does not execute the GOTO.

For yet another example, suppose the frame had 16-bit fill-words but a 24-bit pattern. You can use the compare instruction to see if the first 16 bits match the pattern and, if they do, you go on to the sync word. If they don't, you stay in the fill-word loop.

```
     .
     .
     .
SYNWT: WORD FILWD   |Output 16 bits as fill
       CMP /FAF3    |16 bits of pattern?
       IFT DOSYN    |If it matches, go to the sync word
       GOTO SYNWT   |Else loop
DOSYN: SYNC LEN=8   |Tell the correlator
       WORD SYNW2   |Output the last 8 bits of the sync
```

The one disadvantage of this program is that it requires an exact match to the first 16 bits of the pattern so you may have problems with noisy data. On the other hand, you are lucky to be able to do this one at all.

The first three of these example programs must tag a sync word found in a fill-word position with the fill-word ID. When that happens, the sync word ID won't appear in the output stream. If that is a problem, the method shown in the last example ensures that the sync word data message is always output.

RECYCLE SUBFRAMES

Recycle subframe lists differ very little from mainframe lists. You list the subcommutated frame in time order beginning with the first word of the minor frame that follows the minor frame containing the sync word. Program the sync word in the *last* minor frame of the major frame. Use the statement

SYNC [*definers*]

to mark it, exactly as you do for the mainframe. As an example, refer to the mainframe with subcommutation shown in this Section of the manual. There are three places on the minor frame where there are subcommutated words. If the frame were three minor frames long the decommutation list might be:

```
SUBFRAME 1 RCY SYNC=D60
| Here is the first minor frame
WORD F1W09 RTM
WORD F1W15 RTM
WORD F1W23
RPT 4 RTM

| Here is the second minor frame
WORD F2W09 RTM
WORD F2W15 RTM
WORD F2W23
RPT 4 RTM

| Here is the third minor frame
WORD F3W09 RTM
SYNC            | SYNC MF3, W15
WORD F3W15 RTM
WORD F3W23
RPT 4 RTM
```

The notable features of the sequence are:

a. There is an **RTM** whenever mainframe words lie on the format between adjacent subcommutated words.

b. You can use the **RPT** statement if subcommutated words are adjacent.

c. The recycle sync word is not the last word in the frame, because there are words in the frame following the word that contains the sync pattern. The Subframe Synchronizer does not reset the program counter until it reaches the end of the last minor frame.

ID SUBFRAMES

There are two types of ID subframes: Counting frames and JAM ID frames. Decommutating both types depends on the value of the ID sync word.

For counting frames, a word in the format contains a binary sequence counter that numbers each minor frame. For counting frames the Subframe Synchronizers have a complete synchronization strategy with flywheeling through non-sequential ID values. The synchronizer expects the IDs to count according to the programmed rules (count-up or count down, between specified starting and ending values). Flywheeling occurs when the synchronizer receives an unexpected ID. The synchronizer counts the flywheel counter down, and if the counter has not gone to zero, the synchronizer outputs the expected ID value instead of the one it sees. That allows correct decommutation even with noisy data. You can program the number of unexpected ID values to allow (the "Flywheel Count") before returning to SEARCH. After the synchronizer the number of consecutive erroneous values you specified, it returns to SEARCH. If a valid ID count is found before the counter reaches zero, the flywheel counter is reset.

JAM IDs are common in computer generated formats, where the ID is an arbitrary identifier for a data sequence. JAM IDs come in random sequence, so no synchronization strategy is possible, but they can solve a variety of decommutation problems.

- Construct an artificial ID word. Use the **SYL** specifier to collect the "mode" bits that occur scattered throughout the data stream. You accumulate these flag bits in a Subframe Synchronizer's ID word register. Then use the Synchronizer in JAM ID mode to direct the Decommutator program through the sequence that corresponds the correct current mode.
- Use the same Synchronizer at several points in a complex computer generated format to perform multi-way branches based on more than one mode word in a frame. You can change the ID value stored in the synchronizer any number of times in the minor frame. The Phoenix missile format is a good example of this application. The guidance frame repeats two times in the mainframe. Each repetition begins with a mode word. You don't need to use two synchronizers to decommutate this format. Just use the mode words as the JAM IDs for the same synchronizer.
- Use the synchronizers to decode embedded avionics bus data recorded in the way described in Chapter 8 of the IRIG Standard 106 *Telemetry Standards.* You can use more than one synchronizer to analyze the message identifiers. An example of this type of format is shown in this Section of the manual, *Data Message Formats*.

DEFAULT, NSYNC and FOR Statements

You use a special syntax for ID subframes. You need this so that you can program entries to the subframe decommutation at points that correspond to the individual ID values. The schematic of an ID subframe description is

SUBFRAME *n ID-mode setup-parameters*
[DEFAULT
    decommutation]
[NSYNC
    decommutation]
"FOR" statements
    decommutation
[*subroutines and other labeled sequences*]

When the program arrives at the subcommutated position in the frame where the statement

SF*n* ENTER

appears, The Decommutator transfers control to the sequence corresponding to the current value of the ID sync word.

You use a **FOR** statement to define the decommutation sequence for specific values of the ID sync word. A **FOR** statement has the schematic

FOR *i1* [*i2* ....]
    decommutation

It says that *for* ID values "*i1* ...." perform the decommutation shown. A typical **FOR** statement might be

```
FOR 3 7 9 25
        LEN=12
        WORD PT2
        RPT 2
        WORD PT11 RTM
```

This statement defines the decommutation when the ID sync has the values 3, 7, 9 and 25. You may account for all the possible sync word values with similar programming in their own **FOR** statements.

The **DEFAULT** statement has the same form as the **FOR** statement. It defines the decommutation for all values of the ID sync word *not* listed in the **FOR** statements that follow it and must be stated first.

The **DEFAULT** statement is particularly useful to define the processing when you want to discard most of the subcommutated data in an ID subframe and output data for only a few values of the ID. Computer generated JAM IDs are often unary encoded, having, in an eight bit sync word for example, only 1, 2, 4, and so on to 128, as valid values. You can also use the **DEFAULT** to provide a decommutation path for invalid states (caused perhaps by noise), where you can either suppress the data or give it special ID tags. Either way, you must account for the correct number of bits in the subcommutated words so that mainframe sync is not lost.

The **NSYNC** statement is similar to the **DEFAULT** statement. (NSYNC is mnemonic for "Not in SYNC.") It has the same form as the **DEFAULT** and **FOR** statements, and you use it to describe what to do with the subcommutated data

words when the subframe synchronizer is in SEARCH mode. You can give invalid data special ID tags or suppress it altogether to avoid contamination of valid data.

When an ID mode synchronizer is in SEARCH the ID value it transmits to the Decommutator is undefined. In some formats the ID sync word occurs near the end of a minor frame, with subcommutated data based upon that sync word positioned earlier in the frame. This presents a problem because when the mainframe first leaves SEARCH the subframe synchronizer has not seen an ID sync word, yet the program must execute a **SF*n* ENTER** statement. When you enter an ID subframe while the synchronizer is in SEARCH, the synchronizer will jam the frame count to the *maximum ID plus 1.* Even if the ID word precedes the first subcommutated data, the first value seen may lie outside the programmed bounds (because of noise contamination). Processing for these states can only be programmed with a **DEFAULT** or **NSYNC** statement. You should program all ID subframes with one or the other of these statements, or both.

The **DEFAULT** statement provides processing instructions for *all* possible values of the ID sync word that the synchronizer programming allows. The **NSYNC** statement applies only to the "maximum ID plus 1" value through the maximum value that can be contained in a word the size of the ID field. For example, if you have an ID word that counts from 1 to 10, the values 0 and 11 through 15 are invalid, but could occur within the 4-bit ID field; the **NSYNC** statement would apply to those values.

If you want to process all invalid states, including not-in-sync, in the same way you can use a **DEFAULT** statement alone. If, however, there needs to be default processing for in-range IDs that differs from the out-of-range processing, use the **NSYNC** statement to describe the latter. If you use both types of statements, give the **DEFAULT** programming first, then the **NSYNC** programming, followed by any **FOR** statements you need.

Multiple Entry Points - ENTRY

Sometimes you need to have more than one possible set of entries to an ID subframe. Typical uses are for formats where the data sequence changes signaled by values in the data stream. You use labeled entry ID subframes to manage this type of format.

Use the statement

SF*n* ENTER AT *label*

to enter a complete and independent set of **FOR** statements. Using labeled entry points, a Subframe decommutation program has the following schematic:

SUBFRAME *n ID-mode setup-parameters*
ENTRY *label*1 [.... *labelnn*]
*label*1:
[DEFAULT
decommutation]
[NSYNC
decommutation]
"FOR" statements
.
.
.
labelnn:
[DEFAULT
decommutation]
[NSYNC
decommutation]
"FOR" statements
[*subroutines*and

You must use an **ENTRY** statement to list the labels that you intend to use as entry points in the decommutation sequences. The Compiler uses the list in the **ENTRY** statement to construct linkage code that the program needs to connect your **ENTER AT** statements to your decommutation sequences.

You must use the same labels that you use in the **ENTRY** statement to label the **DEFAULT** statement, if it is present, or else the first **FOR** statement in each labeled sequence. For multiple entry ID subframe programs there is no unlabeled entry; every sequence must have a label.

Each entry point should have either a **DEFAULT** or **NSYNC** statement, or both.

Example ID Subframe Programs

An example of a simple ID subframe description with a single entry point might be:

```
SUBFRAME 1 ID LEN=12 MIN=1 MAX=8
DEFAULT
        SUP  24 RTM
FOR 1 3 4
        WORD  VAL1
        WORD  VAL2 RTM
FOR 2
        WORD VAL3
        GOTO IDF7
FOR 7
        WORD VAL5
IDF7:   LEN=8
        WORD VAL61
        WORD VAL62 RTM
```

This example defines 8 data frames. Data from frames 5, 6, and 8 are suppressed (the **DEFAULT** statement). Note the use of the **GOTO** in the sequence for ID value 2 that goes to a label in the sequence for ID value 7.

The following example illustrates the structure of multiple entry ID subframe definitions.

```
SUBFRAME 1 ID LEN=12 MIN=0 MAX=3
ENTRY JT1 JT2
JT1:
DEFAULT
        SUP  48 RTM
FOR 1
        WORD  VAL1
        WORD  VAL2
        WORD  VAL3
        WORD  VAL4 RTM
FOR 2
        CALL  W2SPEC
        VAL5  RTM

JT2:
DEFAULT
        SUP 4 RTM
FOR 0 2 3
        WORD  DEF1
        WORD  DEF2
        WORD  DEF3
        WORD  DEF4 RTM
FOR 1
        CALL  W2SPEC
        WORD  DEF5 RTM

|  SUBROUTINE TO DO ONE BIT WORDS

W2SPEC: LEN=1
        WORD  BIT1A
        RPT  7
        LEN=4
        WORD  SUP
        LEN=1
        WORD   BIT1B
        RPT  11
        LEN=12
        WORD   SUP
        RET
```

In this example there are two entry lists, JT1 and JT2. To enter the subframe at JT1, for example, the mainframe might have the statement

```
SF1 ENTER AT JT1
```

The example also illustrates the use of a decommutation subroutine, "W2SPEC," that follows all the **FOR** statements, and which is called when the ID value is 2 in list JT1, and in list JT2 when the ID value is 1. The entry at JT2 has **FOR** statements account for all four possible states of the ID tag, but it still needs an **DEFAULT** statement to account for the not-in-sync data. The **DEFAULT** statement supplies that for the entry at JT1, plus processing for the values 0 and 3.

ESTABLISHING SUBFRAME WORD PROPERTIES

For the Advanced PCM Decommutator, word properties (length, orientation, parity bit, and justification) are set during compilation (see Section 5, *Word Properties*). As the program is being compiled, for the most part properties are set as they are encountered in the program, and applied to the **WORD** statements that follow. The exception to this is when the Compiler finds a **CALL**, **GOTO**, or **IF***n* referencing an undefined label. The software remembers the word properties in effect at the transfer statement and sets those when the label is defined later in the program. See Appendix A for additional information about word properties and the new decommutator cards.

Unless you start a program section, MAINFRAME and SUBFRAME, with an explicit properties setting command. the Compiler will use the properties you have selected for the typical words for that section. This means that the transition from MAINFRAME to SUBFRAME and back is simpler than it was on earlier machines. For example, if you were programming a format where the mainframe words are 12 bits long, but typically the subcommutated data is 16 bits, you could use the following sequence.

```
MAINFRAME SYNC=FAF320 LEN=12
        SF1     ID
        WORD    SFID            | 12 Bits
        WORD    W2              | 12 Bits
        SF1     ENTER           | Words 3 & 4
        WORD    W5              | 12 Bits
  .
  .
  .
SUBFRAME 1 ID MIN=0 MAX=7 LEN=16
DEFAULT
  .
  .
  .
FOR 0
        WORD    F0W3            | 16 Bits
        WORD    F0W4 RTM        | 16 Bits
FOR 1
        WORD    F1W3            | 16 Bits
        LEN=4
        WORD    F1W4A           | 4 Bits
        WORD    F1W4B           | 4 Bits
        WORD    F1W4C           | 4 Bits
        WORD    F1W4D RTM       | 4 Bits
FOR 2
        NWD
        WORD    F2W3            | 16 Bits
  .
  .
  .
```

Notice that it is not necessary to reset the word properties in the mainframe following the subframe reference. This is because the Compiler remembers what the settings were prior to the subframe enter, and they remain that way. The

NWD command *is* necessary after the **FOR 2** statement, however. If it had been omitted, the Compiler would have remembered the settings for the preceding 4-bit words, and applied them to F2W3, and that would be wrong.

You can usually avoid word properties problems if you always set the required properties as soon as you enter a frame segment at every transition into and on return from another frame segment. Remember that with the Advanced PCM Decommutator there is no penalty for declaring word properties. The properties are incorporated into every output (**WORD**) statement; declarations do not themselves generate any code.

COMPLEMENT FRAME SUBFRAME SYNC

When a complementing mainframe sync pattern flags the end of a major frame, it is simply a recycle subframe in which the mainframe sync word is a subcommutated word that contains the recycle subframe pattern. An example of the programming for this is

```
MAINFRAME CSF SYNC=EB90
   .
   .
   .
SF1 LEN=16             |The last subcommutated data word
LEN=8
WORDS AAA BBB CCC      |Just the last few mainframe words
SF1 RCY LEN=16         |This is the sync word for SF1
SYNC                   |And it's the minor frame sync, too
WORD SYWD

SUBFRAME 1 RCY SYNC=146F  |The complement of EB90
   .
   .
   .
WORD SFWDA RTM         |Next to last subcom word in frame 1
RTM                    |Mainframe Sync word frame 1 - do nothing
   .
   .
   .
WORD SFWDB RTM         |Next to last subcom word in frame 1
RTM                    |Mainframe Sync word frame 2 - do nothing
   .
   .
   .
WORD SFWDC RTM         |Next to last subcom word last frame
SYNC                   |Tell the subframe sync its the sync
RTM                    |But don't do anything else
```

ASYNCHRONOUS SUBFRAMES

Asynchronous Subframes occur when the position of the recycle sync word in a recycle subframe is not fixed in a particular mainframe word position. This occurs if the subframe does not recycle at a multiple of the minor frame rate. The resulting frame of subcommutated words precesses through the Mainframe words that they occupy. A similar problem occurs if the recycle frame is a proper multiple of the minor frame rate, but the initial position of the commutator is not synchronized to the minor frame word counter.

The Decommutator can decommutate frames of this kind. To do this, the you program the Mainframe to test for the recycle sync pattern at every occurrence of the subcommutated word position. A good example is the Cruise Missile Format that has an interlaced guidance frame of 16-bit words occupying 48 successive bits of the mainframe. The fragment of the Mainframe program for these 48 bits is

```
   .
   .
   .
WORD    W401         | PREVIOUS ANALOG (12 BIT WORD)
SF1 RCY LEN=16
SF1 RCY LEN=16
SF1 RCY LEN=16
NWD
WORD    W406         | NEXT ANALOG WORD
   .
   .
   .
```

This sequence says that following Mainframe word 401 are three 16-bit subcommutated words and that the recycle sync pattern for the subframe may appear in any of the three. After it finds the pattern, the Decommutator ignores the RCY modifier in following SF1 instructions and allows the subframe program to track the format.

In older Acroamatics decommutators, you needed a WFD (Wait For Data) statement following each of the SF1 declarations You needed the **WFD** statement because of the way in which the Decommutator executes the instruction in your program. After a **WORD**, the Decommutator reads and executes all the instructions up to the next **WORD**, regardless of whether the data they describe has arrived from the PCM stream. The instruction look-ahead substantially increases the data throughput. When processing formats in which an asynchronous secondary format is inserted into consecutive words of a primary format, you must test for the secondary format sync pattern at every position of the primary format where it can occur. Since the path the program takes depends on the data in the PCM stream, you must prevent the instruction look-ahead. The "Wait For Data" instruction causes the program advance to delay until the Word Processor completes assembly of the previous word. With the 1502/1602 Decommutator, the Wait For Data is built-in when an asynchronous subframe is selected, and the explicit instruction is not necessary. Then the data has also been processed by the subframe synchronizer.

The decommutation program for the subframe requires special treatment in that a mainframe word may or may not follow every word in the subframe. This means that, in our example where the embedded format is made of 16-bit words, the subframe list *must* RTM after every 16 bits. When the program is

running, if the RTM returns to the Mainframe at one of the SF1 statements, the SF1 will re-enter the subframe to do the next subframe word. Otherwise the Decommutator processes the next mainframe word.

An example of the subframe decommutation for an asynchronous subframe might be

```
SUBFRAME 2 RCY ASYNC SYNC=/EB90 LEN=16 LSB
        WORD  W501 RTM
        RPT 2           | Generates 2 WORD xx RTM
        LEN=1
        WORD  W641
        RPT   7         | 8 one-bit words
        SUP   8 RTM     | 8 more bits finishes the word
        LEN=8
        WORD  W519      | two 8-bit words
        WORD  VAL7 RTM
  .
  .
  .
```

Note particularly that you program an **RTM** at 16-bit intervals because any number of bits from the mainframe may intervene between any two words of the subframe. Observe how the **RPT** is expressed. The **RTM** appears on the base word, word W501. This causes the Compiler to generate the sequence:

```
WORD  W501 RTM
WORD  W502 RTM
WORD  W503 RTM
```

If you had written

```
WORD  W501
RPT   2 RTM
```

the result would be incorrect for this application. You would not return to the Mainframe until after 48 bits had been processed. You can also write the correct sequence:

```
WORD  W501 RTM
RPT   2 RTM
```

The Compiler ignores **RTM** attached to the **RPT** since it is already doing **RTM**s because of the one on the **WORD** statement, but it reminds you that an **RTM** is happening at each word time.

DATA MESSAGE FORMATS

Modern PCM formats often contain data from sources that cannot be made to generate their data synchronously to the PCM commutator. The previous section discussed a conventional frame structure embedded asynchronously in locations of another PCM format. Message telemetry adds more complexity to this kind of PCM. Messages, for example, avionics bus data, can be inserted into portions of a PCM format. The 1553 Bus encoding described in Chapter 8 of the IRIG Standard 106, *Telemetry Standards*, is an example of this kind of data. Another example of this kind of data is CCSDS Packet Telemetry data.

For many formats that use this type of data encoding, the position of the data messages is not fixed in the mainframe words. A message may span several minor frames and may end part way through a minor frame. Then another message may or may not begin immediately.

For the Decommutator these format are simply another form of asynchronous PCM formats. The Decommutator can decode and ID tag individual words in the data messages. You can process and display this data in real-time, if you wish.

Processing Chapter 8 1553 Data

The following example shows one kind of 1553 message encoding. Unfortunately, the message encoding shown in IRIG 106-8 is only one of several methods by which avionics bus data is inserted to PCM streams. The system used in this example follows the Chapter 8 method only approximately. In Acroamatics experience, no two formats use the same method to embed avionics bus data. The example here shows the techniques that you can use in the PCM Decommutator to solve these difficult decoding problems.

The mainframe contains 1553 words in locations 15, 17, 19-23 The following fragment of the mainframe program shows how to call a subroutine, B1553, to process the 24-bit data slots.

```
PCM
MAINFRAME SYNC=FAF320 LEN=24
  .
  .
  .
     LEN=24              |For the 1553 Bus data
     CALL B1553          |W15
     LEN=8
     WORDS XY1 XY2 XY3   |3 8-bit words
     CALL B1553          |W17
     LEN=10
     WORDS ZZZ1A ZZZ1B   |Two 10-Bit words
     SUP  4              |And four bits to discard
     CALL B1553          |W19
     CALL B1553          |W20
     CALL B1553          |W21
     CALL B1553          |W22
     CALL B1553          |W23
     LEN=8               |And so on ....
  .
  .
  .
     SYNC
     WORD MFSYNC         |W128
```

We use a subroutine that processes every 24-bit 1553 word position. The first 8 bits of each word are decoded by Subframe Synchronizer 1, so the routine sends them there. The 1553 encoder fills empty message slots with a word with bit 15 set. The format designer calls them "pseudo words," and we use a test to discard them. If it is not a pseudo word, we enter the subframe to decode the message.

```
| PROCESS 24 BIT 1553 BUS WORDS.
| FOUR BITS OF THE 8 MSBS OF THE
| 24 BIT WORD CONTAIN THE "DATA FIELD ID" AND THE "RT TO RT"
| TRANSFER FLAG THAT DEFINES THE WORD TYPE OF THE REMAINING
| 16 BITS (THE "DATA FIELD").

B1553:  SF1     ID LEN=8   |DATA FIELD ID
        WORD    WRDID
        LEN=16             |SET THE LENGTH FOR "DATA FIELD"
        TST1    15         |WAS IT A PSEUDO WORD?
        IFT     PSEUDO     |IF SO, JUST GET RID OF IT
        SF1     ENTER      |ELSE GO FIND OUT WHAT IT WAS
        RET
PSEUDO: WORD    PSEUDO
        RET
```

We use two subframe synchronizers to decode the 1553 data. The first one decodes the word identifier. The identifier is a 4-bit field that identifies the kind of data: Time Word, Status Word, Command Word, or Data Word. The subroutine, above, B1553, sets the word length to 16 bits before it selects the subframe. Only four states of the identifier are valid. The **DEFAULT** statement discards a word with an invalid identifier. There is not much else you can do with it. The **FOR** statements are the processing for each message type.

- A Time Message,
  We test bits in the identifier to tell which of the Time Words it is. Remember that the test instructions test the last word output, which was the identifier output in B1553.

- A Data Message
  We can process a data message only if we have received at least one Command Message. The Command Message processor sets up Subframe 2 to process the data message. Until that happens, Subframe 2's program counter is not set correctly, so we can't use it. Processing the first command message sets Flag 4. If it is not set, we discard the data. Notice that when we want to discard any data, we just output it and return to the mainframe. That returns us to where we are ready to do another 1553 word or a mainframe word, depending on where we are in the mainframe program. If we have seen a command, Subframe 2 processes the words in the data message. We select the subframe and that processes one more word of the data message. Subframe 2 also has a loop that discards extraneous data words. We will discuss that later.

- A Status Message
  A status message is just a 16-bit word and we tag and output it.

- A Command Message
  A command must set up Subframe 2 to process the data message that follows the command word. To force Flag 4 on, we test a bit of the word identifier that we know is set. The Data word message processor tests that flag to determine that a command has been received.

The next step illustrates an important principle for processing this type of data. The 16-bit command word is sent to the subframe synchronizer and output. Then we enter the subframe, even though we have no word to output from

the next slot, and we don't even know if it is another 1553 word. When we look at the Subframe 2 code, we will see that we don't process a word at the entry point. We make the entry to set Subframe 2's program counter to process the correct upcoming data message.

```
| JAM ON WORD IDENTIFIER IN BITS 18-21

SUBFRAME 1 JAM MIN=0 MAX=15 MASK=00111100 LEN=16

DEFAULT                  |INVALID VALUES
        WORD   INVID RTM |DISCARD

| FOR TIME PROCESSING WE USE THE "BUS ID" (BIT 17)
| AND THE "MUX BUS PARITY" (BIT 16) TO IDENTIFY
| WHICH TIME WORD IS BEING PROCESSED.

FOR /3                          |TIME WORD
        NWD                     |SET LENGTH FOR TIME WORD
        TST2    8               |CHECK FOR TIME WORD 2
        IFT     TW2             |IF IT IS ...
        TST2    9               |OR FOR TIME WORD 3
        IFT     TW3             |IF SO, GO GET IT
        WORD    TW1 RTM         |SEND THE WORD
TW2:    WORD    TW2 RTM         |SEND THE WORD
TW3:    WORD    TW3 RTM         |AND SEND THE WORD

FOR /4                          |DATA WORD
        IF4     DODATA          |HAS A COMMAND BEEN SEEN?
        WORD    SUP RTM         |IF NOT, DISCARD THE DATA
DODATA: SF2                     |ELSE SELECT SF2 FOR PROCESSING

FOR /5                          |STATUS
        WORD    STATW RTM       |OUTPUT IT
| IF THE IDENTIFIER IS "COMMAND", THE
| "DATA FIELD" CONTAINS THE FOLLOWING:
|
|      1
| BIT   5              0
|       RRRRRDSSSSSCCCCC
|
| WHERE:        RRRRR IS THE REMOTE TERMINAL ADDRESS
|               D     IS THE DIRECTION (NOT PROCESSED)
|               SSSSS IS THE SUBADDRESS (1-30)
|               CCCCC IS THE MESSAGE WORD COUNT
|
| THE GENERAL FORM FOR COMMAND MESSAGES IS "COMMAND/RTA,SUB-ADDR"
| FOLLOWED BY ONE OR MORE DATA WORDS, THE LENGTH DETERMINED BY THE
| RTA/SUBADDRESS.  THE "DATA FIELD" IS SENT TO SYNCHRONIZER 2 FOR
| DECODING, THEN SF2 IS ENTERED TO SET ITS PROGRAM COUNTER SO THE
| DATA WORDS WHICH FOLLOW WILL BE UNIQUELY ID TAGGED WHEN SF2 IS
| SELECTED UPON DETECTION OF A "DATA WORD IDENTIFIER" (ABOVE).
|
FOR /6                          |COMMAND
        TST4    11              |SET TOGGLE FOR "COMMAND SEEN"
        SF2     ID              |FOR ADDRESS/SUBADDRESS
        WORD    INFOW           |OUTPUT THE WORD
        SF2     ENTER           |SET FOR PROCESSING THE MESSAGE
```

For Subframe 2 we show the processing for the invalid command messages, and for three of the valid messages. The subframe mask selects 10 bits of the 16-bit command word do construct a message identifier. Each valid message has as **FOR** statement that processes it uniquely, based on the definition of that particular message.

The DSCRD loop is particularly interesting. Once we begin processing a command, we will keep on processing that command as long the 1553 encoder claims it is sending us data messages. If those messages are off the end of the message type we are processing or are part of an invalid command, we want to throw them away. The Subframe 2 program idles in the DSCRD loop until a new 1553 command reenters the subframe to process a new message. Hence, the **DEFAULT** entry falls into the DSCRD loop to throw away any message that is not defined. The **FOR** statements process the messages. Each message word receives a unique ID tag. After the program outputs the correct number of words for each message, each message processor jumps to DSCRD. If the message has been correctly transmitted by the 1553 system, this jump will never happen. It keeps the decoding from getting lost if an extra data word is output or if noise garbles the command type.

Be sure to notice the **RTM** statement that starts every processing sequence. The command processing code in Subframe 1 has already output the command word, so we just return to the mainframe. That leaves the Subframe 2 program counter in the right place to process the first data word.

```
| PROCESS DATA WORDS BASED ON RTA/SUBADDR.

SUBFRAME 2 JAM MASK=1111101111100000 MAX=1023 LEN=16
DEFAULT                         |INVALID MESSAGES
        RTM                     |ALREADY PROCESSED INFO. FIELD
        WORD    INVMSG RTM      |MARK INVALID IDENTIFIERS
DSCRD:  WORD    SUP RTM         |DISCARD THE DATA
        GOTO    DSCRD           |AS MANY TIMES AS NECESSARY

| THE "FOR" VALUES BELOW ARE THE CONCATENATED RTA & SUB-ADDR.

FOR /195                        |RTA 0C, SUB-ADDR 15
        RTM                     |ALREADY PROCESSED INFO. FIELD
        WORD    QC1501 RTM
        WORD    QC1502 RTM
        GOTO    DSCRD

FOR /182                        |RTA 0C, SUB-ADDR 02
        RTM                     |ALREADY PROCESSED INFO. FIELD
        WORD    QC0201 RTM
        GOTO    DSCRD

FOR /216                        |RTA 10, SUB-ADDR 16
        RTM                     |ALREADY PROCESSED INFO. FIELD
        WORD    I01601 RTM
        WORD    I01602 RTM
        WORD    I01603 RTM
        WORD    I01604 RTM
        WORD    I01605 RTM
        WORD    I01606 RTM
        WORD    I01607 RTM
        GOTO    DSCRD
ENDFORMAT
```

You can express the longer sequences with the **RPT** statement to make the source program more compact. For example message /216, above, can be expressed:

```
FOR /216                        |RTA 10, SUB-ADDR 16
        RTM                     |ALREADY PROCESSED INFO. FIELD
        WORD    I01601 RTM
        RPT 6
        GOTO    DSCRD
```

The Compiler will generate the necessary sequence of **WORD .. RTM** statements.

Processing CCSDS Packet Data

In this example, the PCM format has 12 minor frames per major frame, and the word length is 8 bits. Scattered throughout the frame are pairs of words allocated to 16-bit CCSDS packet data. A packet header is always located beginning at word positions 15 and 16 in Minor Frame 0. That header, followed by its packet data words, make up the first packet, which, in turn, is followed by another packet, continuing on through the remainder of the major frame. The last packet in the frame is usually "fill" data, but that is a packet like any other so no special programming is required to handle it.

The Mainframe Segment

The Mainframe program for this example is not unusual, except for setting a flag to indicate that Minor Frame 0 has been processed. The reason this is done is so the program does not attempt to process packet data if the decommutator comes into sync in the middle of the major frame; the only time we can reliably identify a header is in the first minor frame, where its location is known.

```
PCM
SYS SRC=1
MAINFRAME SYNC=/FAF320 ERS(0,1,2) LEN=8 MSB
        NWD
        SF1     ID LEN=8
        WORD    SFID            | W1
        IF1     ONESET          | DON'T TEST IF WE'VE SEEN
        CMP1    0               | MINOR FRAME 0, ELSE TEST
ONESET: SF1     ENTER           | W2-4
         .
         .
        SF1                     | W11-19
         .
         .
        WORDS   SYNC1 SYNC2
        SYNC    LEN=8
        WORD    SYNC3
```

After the frame counter, **SFID**, has been output, the **IF1** statement branches around the Compare statement if Compare Flag 1 has been set. That is done to prevent the flag from being reset after it has once been set. If the flag is not "True", that means the program has never seen minor frame 0, so the **CMP1** is executed to test whether the value of **SFID** just output is zero. If it is, the compare flag will be set, and the test won't be made again. If not, the program will check again at the next minor frame.

Processing The Minor Frames

Subframe 1 is used to process the subcommutated data, including identifying the packet data locations in the frame. The first 16-bit header word, which begins in word position 15 of Minor Frame 0, is sent to Subframe 2 for identification (the Message ID). The next word in the minor frame is another packet word, the second word of the 3-word header. The program enters Subframe 2 to process that word, thereby establishing the program path for the remainder of that first packet. All other packet words are processed by the **CCSDS** subroutine for the remainder of the major frame, after which the packet data processing will be resynchronized in Minor Frame 0.

```
SUBFRAME 1 ID MASK=/0F00 MIN=0 MAX=11 LEN=8

DEFAULT
         .
         .
FOR 0
        NWD
        WORD    F                | W2F0
        LEN=16
        WORD    TFPH1 RTM        | W3F0-W4F0
        LEN=16
        WORD    TFPH2            | W11F0-W12F0
        WORD    TFPH3            | W13F0-W14F0
        SF2     LD LEN=16        | FIRST PACKET HEADER WORD
        WORD    PKTW1            | W15F0-W16F0
        SF2     ENTER            | PROCESS THE REST OF THE HEADER
        NWD
        WORD    A35 RTM          | W19F0
        WORD    A4               | W26F0
        CALL    CCSDS            | 3RD WORD OF PKT HDR (W27F0-W28F0)
        CALL    CCSDS            | W29F0-W30F0 (PACKET DATA)
        NWD
        WORD    A36              | W31F0
        WORDS   G0 G1 RTM        | W32F0-W34F0 ~ G0 G1
        CALL    CCSDS            | W41F0-W42F0 (PACKET DATA)
        CALL    CCSDS            | W43F0-W44F0 (PACKET DATA)
        CALL    CCSDS            | W45F0-W46F0 (PACKET DATA)
        CALL    CCSDS            | W47F0-W48F0 (PACKET DATA)
        NWD
        WORD    A5 RTM           | W49F0
         .       .
         .       .
        CALL    CCSDS            | W405F11-W406F11 (PACKET DATA)
        CALL    CCSDS            | W407F11-W408F11 (PACKET DATA)
        NWD
        WORD    A44 RTM          | W409F11
        WORD    S                | W416F11
        WORD    S RTM            | W417F11
```

After the first header word in the major frame has been processed, and Subframe 2 has been entered to process its message, all other words in the major frame are processed by the subroutine shown below. The subroutine is called from Subframe 1 in each location where there is a packet data word. It either suppresses a 16-bit word if Compare Flag 1 is not set (because Minor Frame 0 has not yet been processed), or it selects Subframe 2 to output a 16-bit packet word. Either way, only one 16-bit word is handled by each call to the **CCSDS** subroutine.

```
CCSDS:  IF1     DOMSG          | IF MF0 HASN'T BEEN SEEN
        LEN=16                 | JUST SUPPRESS 16 BITS
        WORD    SUP
        RET
DOMSG:  SF2     LEN=16         | ELSE PROCESS PACKET MSG
        RET
```

Processing The Packet Data

Subframe 2, programmed in "JAM ID" mode, decodes the packet identifier (Message ID) in the 11 LSBs of the first word of a packet header. The program then *enters* the subframe, based upon the value of that identifier. Once Subframe 2 has been entered, the Subframe 2 "Select" instructions in Subframe 1 keep the process going.

You write a **FOR** statement corresponding the Message ID of a packet that contains data you want to uniquely tag in order to process or display. Any other packets are handled by the **DEFAULT** processing. In this example, we only show the processing for data in Packets 60 and 125.

```
SUBFRAME 2 JAM MIN=0 MAX=2047 MASK=0000011111111111 LEN=16
DEFAULT
        WORD    PKTW2           | FLAGS & SEQ. CNT
        RT1
        SF3     LD              | EXTRACT THE PACKET LENGTH
        WORD    PKTW3           | HEADER
        LPL1    SF3             | LOAD LENGTH INTO REGISTER 1
        RT1
DSCRD:  TLZ1    PRHDR           | FINISHED WITH THIS PACKET?
        WORD    SUP             | IF NOT, DISCARD A DATA WORD
        RT1
        GOTO    DSCRD

FOR 60                          | PACKET 60 - 32 WORDS
        WORD    PKTW2           | FLAGS & SEQ. CNT
        RT1
        SF3     LD              | EXTRACT THE PACKET LENGTH
        WORD    PKTW3           | HEADER
        LPL1    SF3             | LOAD LENGTH INTO REGISTER 1
        LTR1    P60W(/400) 32   | BASE TAG FOR PACKET 60 DATA
        RT1
        GOTO    DODATA

FOR 125                         | PACKET 125 - 5 WORDS
        WORD    PKTW2           | FLAGS & SEQ. CNT
        RT1
        SF3     LD              | EXTRACT THE PACKET LENGTH
        WORD    PKTW3           | HEADER
        LPL1    SF3             | LOAD LENGTH INTO REGISTER 1
        LTR1    P125W(/440) 5   | BASE TAG FOR PACKET 125 DATA
        RT1
```

Any path taken when Subframe 2 is entered does essentially the same thing. The first word processed upon entry is always the second header word, after which the program returns to Subframe 1 (**RT1**). Notice that Subframe 2 always returns to Subframe 1 after each 16-bit packet word has been processed. It is necessary to do that because there is no synchronous connection between the two subframes.

The next packet word processed has the Packet Data Length, limited in this example to a maximum value of 2047 octets (8 bits) Subframe 3 extracts the packet length from the data length field, then loads the packet length *plus one* into Packet Length Register 1.

Because the program will suppress the data from packets not being processed, it simply returns to Subframe 1 in the **DEFAULT** code section. In the **FOR** sections, however, the program loads Tag Register 1 with the base ID of a block of tags for the packet data words before returning. It also has the Compiler generate an appropriate number of symbolic labels for each word.

Having processed all three header words, the program uses the **TLZ1** statement to count the number of data words processed. The **DEFAULT** processing suppresses the words, while the packets of interest are processed in the common code sequence at **DODATA**, where the **TAG** statement outputs each data word with a consecutive numeric ID tag.

When a packet is complete and every data word output, the common code sequence at **PRHDR** starts the sequence over again. The next packet word is assumed to be the first word of a header, and it is sent to Subframe Synchronizer 2, just as it was in Minor Frame 0, then the program returns to Subframe 1. The next time Subframe 2 is selected, it "enters" itself recursively to process the next packet.

```
DODATA: TLZ1     PRHDR          | FINISHED WITH THIS PACKET?
        TAG      1              | IF NOT, TAG & OUTPUT DATA WORD
        RT1
        GOTO     DODATA         | UNTIL PACKET COMPLETE

PRHDR:  SF2      LD LEN=16      | PACKET HEADER WORD 1
        WORD     PKTW1          | MESSAGE ID
        RT1
        SF2      ENTER          | PROCESS THE REST OF THE HEADER
```

Extracting The Packet Length

Subframe Synchronizer 3 is used to extract the data octet count from the Packet Data Length word (the third header word).

```
SUBFRAME 3 EXT MASK=0000011111111110
```

In our example, there are 1024 16-bit words allocated for packet data, so Subframe 3 extracts 10 bits of the data length field, leaving off the LSB because in this format packet data is always a 2-octet (16-bit) word. The result is the number of 16-bit data words in the packet.

APPENDIX A
WORD PROPERTIES
AND THE
ADVANCED PCM DECOMMUTATOR

APPENDIX A
WORD PROPERTIES
AND THE
ADVANCED PCM DECOMMUTATOR

INTRODUCTION

The Models 1502V and 1602P Advanced PCM Decommutators are designed to be a replacement for the earlier 502 Frame Synchronizer. The 32-bit instruction set of the new decommutator is largely compatible with the 16-bit equivalent in the 502. Of course there are several new instructions that add features and capabilities, but most setup programs for the 502 card will run on the 1502 or 1602 with few or no modifications.

Perhaps the most significant difference between the Advanced PCM Decommutator and the 502 Frame Synchronizer is the way word properties are handled. In the 502 card, word properties are set by executing an instruction (op-code "**D**" or "**E**"). The word properties remain as set *until another word properties setting instruction is executed*. This means the word properties are set following the instruction flow, as the program is executed.

In contrast, in the new card the word properties of a PCM word are incorporated into the output instruction that assigns the ID tag for that word. When a program is compiled for the 1502 or 1602 card, a command such as

```
LEN=12
```

which would generate an instruction for the 502, simply tells the Compiler what the word properties are for subsequent `WORD` statements. In other words, for the Advanced PCM Decommutator, word properties are not set by instruction, but rather by commands that have their effect *as they are encountered during compilation*.

PROGRAMMING EXAMPLE 1

To see how this affects your programming, let us look at an example format where the typical word is eight bits long. Suppose there are three eight bit words that repeat several times in the frame, and they are always followed by a word that is 16 bits in length. Following good programming practice, you might have written a subroutine to output the three words, then set the word properties to 16 bits before returning. That way, for the 502 card, the word properties will always be set correctly after the subroutine is called. Your program fragment might look like this:

```
          .         .
          .         .
        WORD    SOMLEN
        CALL    DO3WDS
        WORD    A16B1
          .         .
          .         .
        CALL    DO3WDS
        WORD    A16B2
          .         .
          .         .
DO3WDS: LEN=8
        WORD    A8B1
        WORD    A8B2
        WORD    A8B3
        LEN=16
        RET
```

When this program is compiled for a new card, however, the Compiler is operating on a line at a time (single-pass). It generates the output instruction for **SOMLEN**, with its word properties, and then generates a forward-reference CALL to the subroutine **DO3WDS**, which it has not encountered yet. When it next compiles the output instruction for **A16B1**, the Compiler can only set the word properties as they are currently within the compilation — in this case, the same as those of **SOMLEN**. By the time the subroutine is encountered it is too late, the output instructions for what are supposed to be 16-bit words have already been compiled.

PROGRAMMING EXAMPLE 2

A similar problem may occur if you have a word position that is subcommutated, and has an atypical but consistent word property in every minor frame. If the typical Mainframe word is eight bits, for example, but in a certain word position the subcommutated data is always sixteen bits, you might well combine the selection of the subframe in that position with setting the length to 16. For the 502 card, then, you would not need to set the word properties on the subframe level at those word positions, since you (the programmer) know they will be set by the subframe select instruction. For the new card, however, the word properties for the WORD statements corresponding to that Mainframe position depend entirely on what they are set to at that point in the compilation. For example:

```
LEN=8
WORD    MFWD22
SF1     LEN=16
  .       .
  .       .
| Now in the subframe
LEN=12
WORD    SFW20 RTM
| The MF select comes here
WORD    SFW23
```

In this example, compiling for a 1502 or 1602 will generate an output

instruction for **SFW23** with the length set to the typical word length for the subframe program section, rather than the "special" length set in the Mainframe program.

COMPILER HELP

Modifications to the TDP Compiler help minimize the changes you will need to make to compile old programs for the Advanced PCM Decommutator. For example, when you write a subroutine call or a branch statement that transfers control to a label later in the program, the Compiler will record what the word properties are at that point. Then, when the label is encountered, the word properties are set to the remembered values, rather than the properties set by whatever happens to precede the label. This means that a program such as the fragment that follows will work correctly.

```
        LEN=8
        WORD    PARAM1
        CALL    DOWRK
         .        .
         .        .
        LEN=32
        WORD    FLWD RTM

DOWRK:  WORD    BYTE1
        WORD    BYTE2
        RET
```

In this example, because the Compiler records the 8-bit length that is in effect when the `CALL` is compiled, the words **BYTE1** and **BYTE2** will be that long, rather than the 32 bits set on the lines just before the label.

The Compiler handles word properties in subframes differently when generating code for the 1502 or 1602 card. The Compiler resets the word properties to those of the typical word in the current subframe section following a `DEFAULT`, or `FOR` statement, after a `WORD` or `RPT` statement with an `RTM` specifier, and also when the previous command was a "Return to Mainframe" (`RTM`) or "Return to Subframe" (`RT`*n*). If, for example, the last telemetry word before returning to the Mainframe was treated as a number of one-bit words, the next word in the program will be given the typical word properties, rather than inheriting the one-bit length. If in the following example the typical word in the subframe is 8 bits, the word **TYPWRD** will be that length, rather than set to a length of 1 bit as were the previous words were before returning to the Mainframe.

```
        LEN=1
        WORD    DSCRT1
        RPT     7 RTM
        WORD    TYPWRD
```

This is almost always correct. Of course if the next word's characteristics are different than the typical word, you can always explicitly set the properties.

Because the TDP is not restricted to the rigid matrix view of PCM, this "help" cannot be applied in situations such as the one shown in "Example 2".

UPDATING YOUR PROGRAMS

Because a word properties setting command does not generate any actual code in the 1502 or the PCM Decommutator, there is no penalty in explicitly setting the properties whenever there is any doubt about how the program will compile. The Compiler listing will help you make sure the settings are correct, because the word properties are shown for every `WORD` and `RPT` instruction when the Compiler is generating code for the 1502/1602 card.