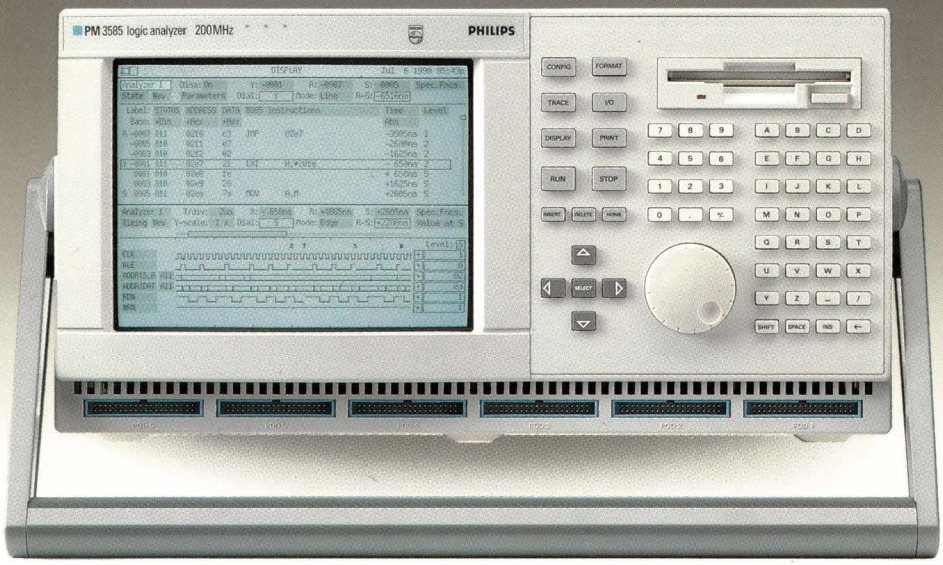
LOGIC ANALYZERS PM 3580/PM 3585

Custom disassembler



FLUKE AND PHILIPS - THE GLOBAL ALLIANCE IN TEST & MEASUREMENT

PHILIPS

PHILIPS

LOGIC ANALYZERS
PM 3580 / PM 3585

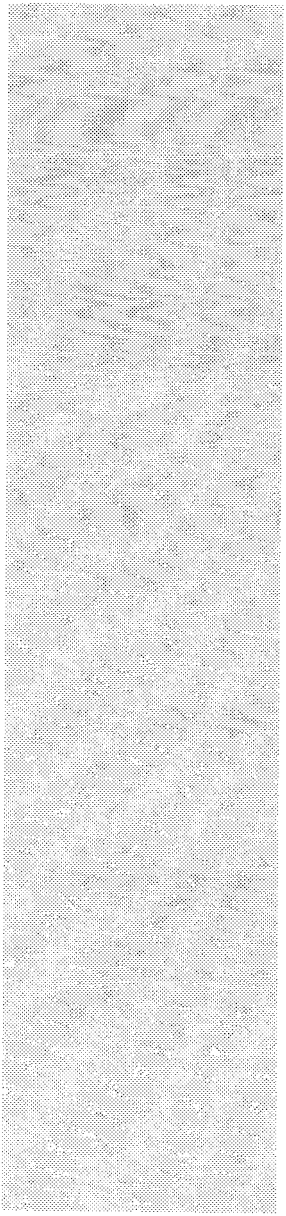
Custom disassembler
PF 8629/30

PHILIPS

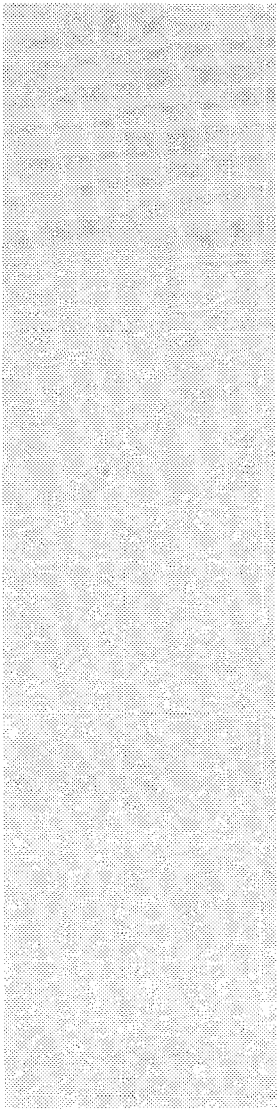
Dual logic analysis

PF8629/30 - Custom Disassembler
Software Version 2.0
IE, Test & Measurement

Publication Number 4022 104 90662



Guarantee Statement



This Philips guarantee is in addition to all rights which the buyer may have against his supplier under the sales agreement between the buyer and the supplier and according to local legislation.

Philips guarantees this product to be free from defects in material and workmanship under normal use and service for a period of one (1) year from the date of shipment. This guarantee does not cover possible required standard maintenance actions. This guarantee extends only to the original purchaser and does not apply to any product or part thereof that has been misused, altered or has been subjected to abnormal conditions of operation and handling.

Fluke/Philips-supplied software is guaranteed to be properly recorded on non-defective media. We will replace improperly recorded media without charge for 90 days after shipment upon receipt of the software. Our software is not guaranteed to be error free.

Philips' obligation under this guarantee is limited to have repaired or replace a product that is returned to an authorized Philips Service Centre within the guarantee period, provided that Philips determines that the product is defective and that the failure has not been caused by misuse, alteration or abnormal operation.

Guarantee service for products installed by Philips will be performed at the Buyer's facility at no charge within Philips' service travel area; outside this area guarantee service will be performed at the Buyer's facility only upon Philips' prior agreement and the Buyer shall pay Philips round trip travel expenses.

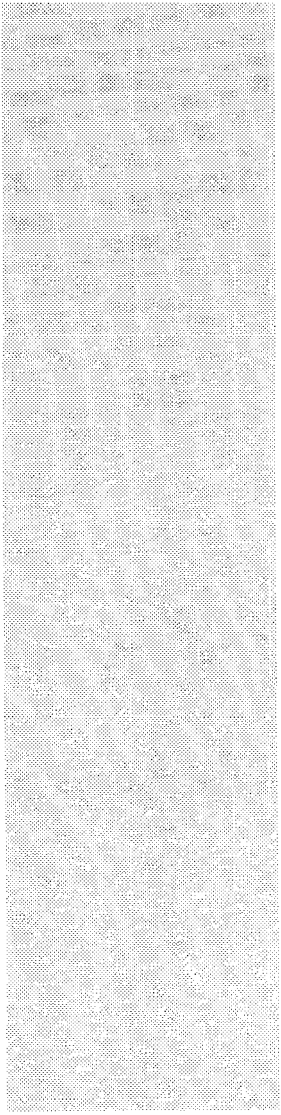
If a failure occurs, send the product, freight prepaid to the Service Centre designated by Philips with a description of the difficulty. At Philips' option, repairs will be made or the product will be replaced. Philips shall return the product, F.O.B. Repair Centre, transportation prepaid, unless the product is to be returned to another country, in which case the Buyer shall pay all shipping charges, duties and taxes. Philips assumes NO risk for damage in transit.

Disclaimer

The foregoing guarantee is exclusive and is in lieu of all other guarantees, expressed or implied, including but not limited to any implied guarantee of merchantability, fitness, or adequacy for any particular purpose or use. We shall not be liable for any direct, indirect, special incidental, or consequential damages, whether based on contract, tort, or otherwise.

Some countries or states do not allow the foregoing limitations. Other rights may also vary.

Preface



Thank you for purchasing the PF 8629/30 Custom Disassembler package.

Should you have any suggestions on how this product could be improved then please contact your local Fluke/Philips representative. Fluke/Philips addresses are listed in chapter 11 of the PM 3580/PM 3585 Logic Analyzers User Manual.

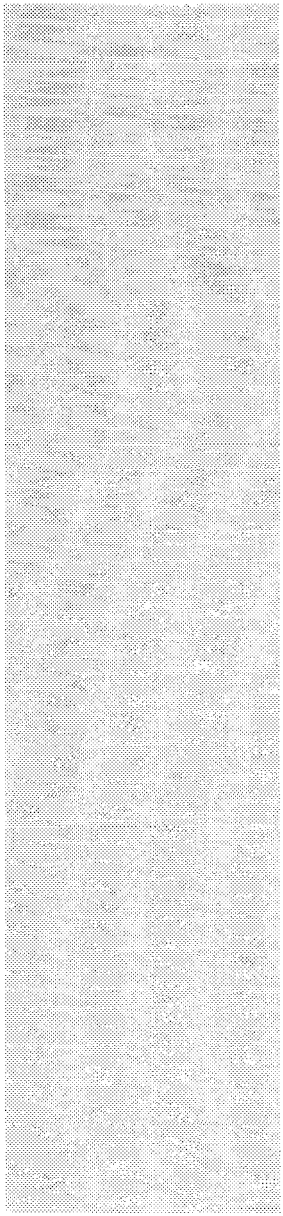
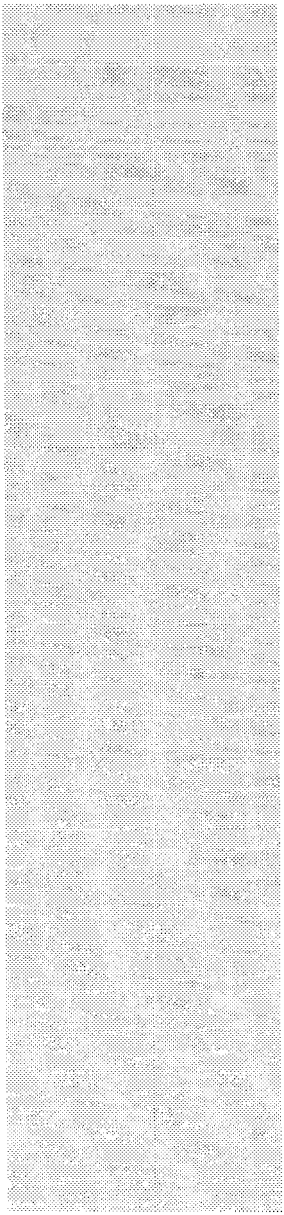


Table of Contents

Guarantee Statement iii
Disclaimer iv
Preface v
Introduction 1
Key Features 1-2
Microprocessor Adapters 1-2
Installation 1-3
Creating a Custom Disassembler 1-4
Disassembler Description File 1-4
Invocation of the Disassembler Compiler 1-4
Loading a Custom Disassembler 1-5
Disassembler Parameters 1-5
Display Options 1-6
Translation Options 1-6
Activating/Deactivating the Disassembler 1-7
About this Manual 1-7
Writing a Custom Disassembler 2
The Hypothetical Microprocessor 2-2
Disassembly Process 2-4
Lookup Table 2-5
Index Table with local variables 2-8
Local Variable 2-8
Print Sample Value 2-10
Detection of Illegal Opcodes 2-11
Global Variables 2-12
Display Section 2-13
Data Transfers 2-14
Completing the Disassembler Description File 2-16
Declaration of Tables and Variables 2-16
Label and Clock Definitions 2-17
Start Section 2-17
Complete Example 2-17
About Processing Speed 2-20
Rearrange Lookup Tables 2-20
Splitting up Lookup Tables 2-20
Usage of Index Tables 2-20
Disassembler Description Language Reference 3
Introduction 3-3
File Structure 3-5
The Declarative Part 3-6
The Tabular Part 3-7
%% DEF Section 3-8
Global Variables 3-8

Constants 3-9
%% EQU Section 3-9
%% FORMAT Section 3-11
Logo Definition 3-11
Header Definition 3-12
Pod Threshold Definition 3-13
Symbolic Output Control 3-14
Synchronization Blocksize Definition 3-14
Clock Definition 3-16
Label Definition 3-22
Clock Sequence Definition 3-29
Tab Settings 3-31
%% START Section 3-32
Tabular Section (%%<name>) 3-33
Lookup Tables (LT) 3-33
Index Tables (IT) 3-33
General Elements 3-35
Lines 3-35
Comments 3-35
Spaces and Tabs 3-36
Upper and Lower Case Characters 3-36
Conditions and Commands 3-37
Conditions 3-37
Pattern Conditions 3-37
Relational Conditions 3-40
Clock Sequence Conditions 3-41
AND-ing and OR-ing of Conditions 3-42
Pattern Expression 3-42
Commands 3-43
Display Selection Commands 3-44
PROG 3-45
Instructions 3-46
UNUSED 3-46
SKIP 3-46
MR 3-47
MW 3-48
IOR 3-48
IOW 3-48
Positioning in a Measurement 3-49
GOTO [i] 3-49
TELL 3-51
NEXT 3-51
PREV 3-52
GOTOPART[i] 3-52
TELLPART 3-53

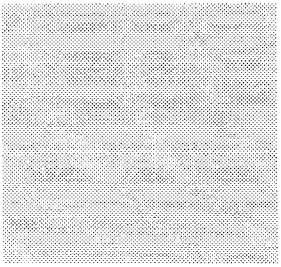
NEXTPART 3-53
PREVPART 3-54
UNGET 3-55
Instructions 3-56
Print Commands 3-57
Special Commands 3-59
UNPUT 3-60
ERROR 3-60
Transfer Control to other Tables 3-60
Writing a 68000 Disassembler 4
Disassembler labels 4-3
Disassembler Status Selection 4-4
Instruction Decoding 4-4
Finding additional opcodes 4-5
Display additional opcodes 4-7
Computing branch offsets 4-7
Sub values in the instruction 4-9
Computing return address 4-9
Searching datatransfers 4-9
Suppressing unused opcodes from display 4-10
Processing data transfers 4-13
Branch instruction 4-13
Conditional branches 4-14
Static char variable usage 4-15
Finishing the example disassembler 4-17
Writing a 68030 Disassembler 5
Disassembler labels 5-3
Disassembler part label 5-4
Positioning within a disassembler state 5-4
Processing data transfers 5-5
Search following opcode 5-6
Branch instructions 5-6
Unused opcode fetches 5-8
Using data transfer states 5-10
Error Messages 6
Warnings 6-2
Error Messages 6-3
Index



Chapter 1

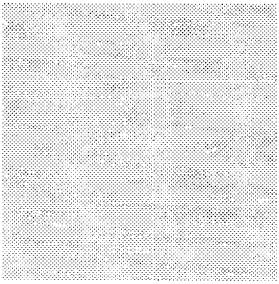
Introduction

Key Features 1-2
Microprocessor Adapters 1-2
Installation 1-3
Creating a Custom Disassembler 1-4
    Disassembler Description File 1-4
    Invocation of the Disassembler Compiler 1-4
Loading a Custom Disassembler 1-5
Disassembler Parameters 1-5
    Display Options 1-6
    Translation Options 1-6
Activating/Deactivating the Disassembler 1-7
About this Manual 1-7



The PF 8629/30 package allows you to develop your own disassemblers, referred to as Custom Disassemblers, that can be executed on the PM 3580/ PM 3585 family of logic analyzers.
This product can be used to develop disassemblers especially for those microprocessors or busses and their protocols for which a standard disassembler package* is not available such as for example your own company proprietary microprocessors.

Key Features



The key features of this package are as follows:

- Simple, yet powerful Disassembler Description Language.
- No knowledge of high-level-language programming required.
- Supports microprocessors as well as busses and their protocols.
- Supports up to 32-bit microprocessors.
- Disassembler compiler program runs under MS-DOS.

Microprocessor Adapters

To measure microprocessor signals you can use special microprocessor adapters. These adapters provide a convenient connection to all the signals of the specific microprocessor. The adapters that have been designed for the PM 3580/PM 3585 family of logic analyzers contain special RC connectors to which pod cables can be directly connected. If you need to develop your own adapter, you can use these special RC connectors in your own design. The RC connectors can be separately purchased from your local Fluke/Philips sales representative, and come in sets of ten connectors (order number: PF 8603/20).

* The number of microprocessors supported by the PM 3580/ PM 3585 family of logic analyzers is continuously growing. You can obtain an up-to-date list of all microprocessors supported from your local Fluke/Philips sales representative.

Installation

The disassembler compiler program runs under MS-DOS. Any IBM-type or IBM compatible PC can therefore be used to develop a Custom Disassembler. The program is not copy protected. However, copying is restricted to the terms indicated in the licence agreement.

To install the program in the current directory on your hard disk or another previously formatted diskette, type the following command after the system prompt:

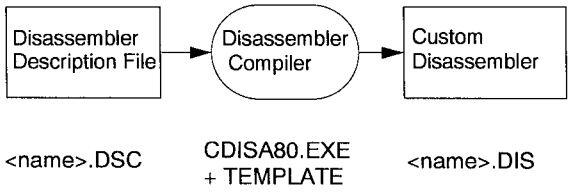
<drive> : INSTALL [RETURN]

<drive> should be replaced by the drive letter containing the custom disassembler package diskette. The installation program is now started. "Install" will copy the required files to the current directory. After these files have been copied "Install" will ask you whether the example files present on the distribution disk should also be copied. Installation is then completed.

The example files are described in chapter 2, "Writing a Custom Disassembler", chapter 4, "Writing a 68000 disassembler" and chapter 5, "Writing a 68030 disassembler" of this manual.

Creating a Custom Disassembler

A disassembler developed with the disassembler compiler program is written to a file with file extension .DIS. This file has a PM 3580/PM 3585 disassembler format and can be loaded as such.
The generation process is shown as follows:



Disassembler Description File

The disassembler description file is a DOS-text file that has to be created according to the syntax requirements outlined in this manual.
The default extension of this file is .DSC. You may use any editor or word processor that can produce DOS-text files.

Invocation of the Disassembler Compiler

The disassembler compiler program (CDISA80.EXE) is invoked by typing the following command after the system prompt:

[<drive>:]CDISA80 <name>.DSC

Two Floppy System or Hard Disk System

On a two floppy system or a hard disk system the resulting file <name>.DIS is placed in the current directory.

One Floppy System

On a one floppy system make sure the current drive is A: Suppose your description file is called MYDISA.DSC. Invoke the CDISA80.EXE program by typing:

A:\> **B:CDISA80 MYDISA.DSC [RETURN]**

You will be asked to swap disks so the CDISA80.EXE program can be started.

Loading a Custom Disassembler

The disassembler <name>.DIS created by the disassembler compiler program can be loaded in the same way as standard disassemblers are loaded on your PM 3580/ PM 3585 logic analyzer.

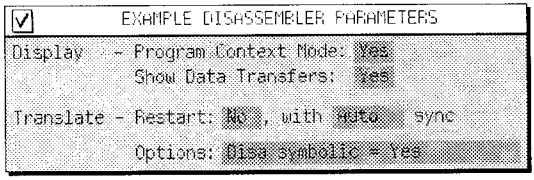
Copy the disassembler onto a 3.5" DOS diskette. Put this diskette in the disk drive of your logic analyzer. You can now load the disassembler using the Option field in the Configuration menu or the LOAD command in the I/O menu.

The disassembler software and incorporated Logic Analyzer settings are then loaded. Please also refer to chapter 7,"Disassemblers", of your PM 3580/PM 3585 User Manual.

Disassembler Parameters

After a disassembler has been loaded, an extra field, *Parameters*, is shown in the state list display.

Pressing *SELECT* on this field shows a popup menu via which different disassembler parameters can be set in order to control the disassembly process. An example of this disassembler parameter popup menu is shown below.



This disassembler parameter popup menu is the same as that for the standard microprocessor support packages. Please also refer to chapter 7, "Disassemblers", of your PM 3580/PM 3585 User Manual.

The fields on the Disassembler Parameters menu are grouped in two sections:

Display This controls which state samples are shown.

Translate This controls the disassembly process.

Display Options

The display options determine which and how disassembled instructions are displayed.

Program Context Mode

The *Program Context Mode* field determines which state samples selected by custom disassembler commands should be shown and whether state samples captured with external clocks that are not defined by the disassembler, should also be shown.

Show Data Transfers

The *Show Data Transfers* determines if state samples selected by the custom disassembler data transfer commands are shown.

Translation Options

The fields related to translation are *Restart* and *Synchronization* (with sync).

Restart

Restart determines whether a new translation (disassembly) should be performed on the current measurement as soon as the popup is closed.

Synchronization

The *Synchronization* field and other fields that may subsequently appear on that line, determine how the disassembler searches for proper instruction starting points.

Automatic Synchronization

For automatic synchronization, the disassembler starts at the earliest point in memory, and continues correcting itself until a properly synchronized disassembly is achieved.

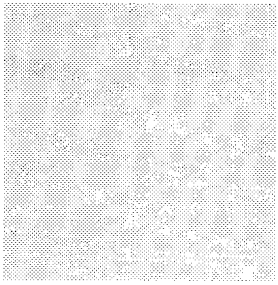
Manual Synchronization

For a manually synchronized disassembly, the disassembler starts at the position the Y cursor is set to. Dependent of the custom disassembler "at Y" fields may be present.

Option

The "Disa symbolic" field specifies whether the disassembler uses symbols on producing disassembler output. This options field is NOT effective for logic analyzer system software versions below 2.01.

Activating/ Deactivating the Disassembler

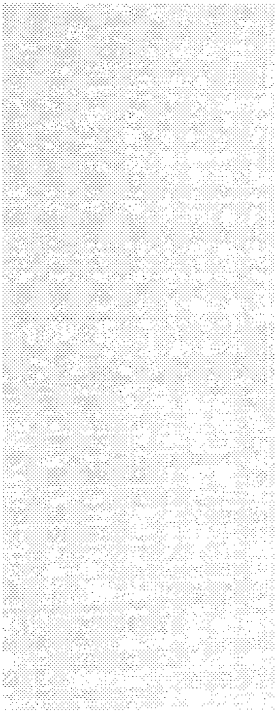


If a disassembler has been loaded, disassembly can be enabled or disabled using the field called *Disa* in the state data display menu.

When disassembly is enabled, the state display shows an additional column containing a translation of the data stored in the acquisition memory. When disassembly is disabled, the additional column is not shown.

Note: If no disassembler has been loaded, the *Disa* field shows "None" and is not selectable.

About this Manual



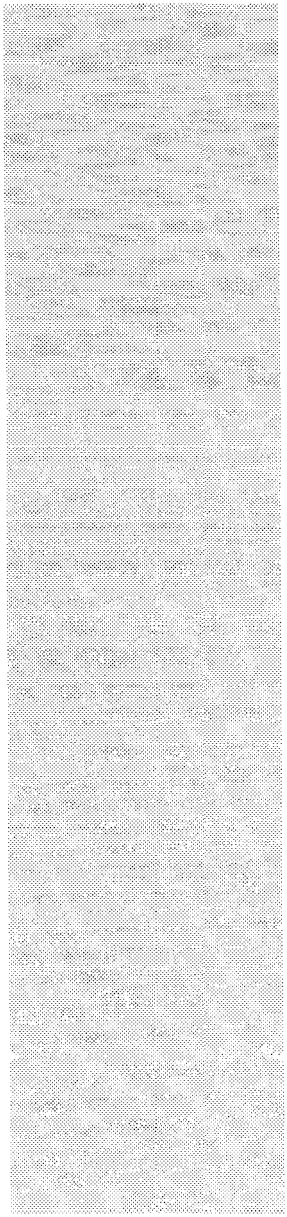
The next chapters contain language reference details and full blown examples of disassemblers covering different levels of complexity. A brief outline of these chapters follows.

Chapter 2, "Writing a Custom Disassembler", describes by means of a simple example how a custom disassembler is developed and how it is described in the Disassembler Description Language.

Chapter 3, "Disassembler Description Language Reference", describes the exact syntax for each of the language elements.

Chapter 4, "Writing a 68000 Disassembler" contains an example how to write a disassembler for an advanced pipelined microprocessor such as the 68000. This example shows how to handle pipeline compensation and how to synchronize the disassembler to find the instructions by using the microprocessor status lines and the information available on the ADDRESS lines of the microprocessor.

Chapter 5, "Writing a 68030 Disassembler" gives an example how to upgrade an existing 68000 disassembler to a 68030 disassembler. This example shows the handling of multiple instructions in one disassembler state.



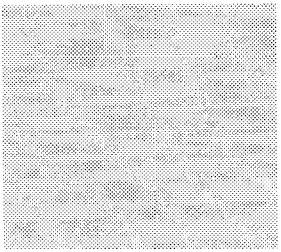
Chapter 6, "Error Messages", explains the error messages and warnings that may be generated by the Disassembler Compiler Program.

The last part of this manual contains the index.

Chapter 2

Writing a Custom Disassembler

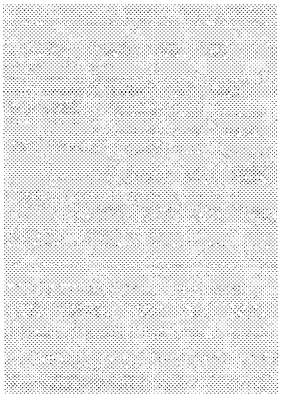
The Hypothetical Microprocessor 2-2
Disassembly Process 2-4
Lookup Table 2-5
Index Table and Local Variables 2-8
    Local Variable 2-8
    Print Sample Value 2-10
Detection of Illegal Opcodes 2-11
    Global Variables 2-12
    Display Section 2-13
    Data Transfers 2-14
Completing the Disassembler Description File 2-16
    Declaration of Tables and Variables 2-16
    Label and Clock Definitions 2-17
    Start Section 2-17
    Complete Example 2-17
About Processing Speed 2-20
    Rearrange Lookup Tables 2-20
    Splitting up Lookup Tables 2-20
    Usage of Index Tables 2-20



Writing a Custom Disassembler requires knowledge of the specific microprocessor as well as the Disassembler Description Language.

In this chapter we will develop a Custom Disassembler for a hypothetical microprocessor. While developing this disassembler we will also learn the basics of the Disassembler Description Language.

The Hypothetical Microprocessor



Let us assume this microprocessor has instructions of different lengths: 1, 2 or 3 bytes. The first byte of an instruction contains the opcode. In two-byte instructions the second byte is a databyte and in three-byte instructions the second and third byte form a 16-bit address item (high byte first).

The microprocessor has a separate R/WN line and OPC/DN line respectively indicating a read/write cycle and, for read cycles, whether an opcode or data is being fetched. The microprocessor has a single clock "CLK" and a special signal "QUAL".

For the purpose of this example we will only look at a part of the microprocessor's instruction set. This part is described below.

Object Code (Hex)	Instruction* Mnemonic	Operand	Operation performed
0i	LOAD	A,Ri	Load accumulator with the contents of register Ri
10 dd	LOAD	A,data	Load immediate data into accumulator
1i dd	LOAD	Ri,data	Load immediate data into register Ri
20 hh ll	LOAD	A,addr	Load accumulator from directly addressed memory location
2i hh ll	LOAD	Ri,addr	Load register Ri from directly addressed memory location
30 hh ll	STORE	A,addr	Store accumulator contents in directly addressed memory location
3i hh ll	STORE	Ri,addr	Store contents of register Ri in directly addressed memory location
4i	ADD	A,Ri	Add contents of register Ri to accumulator
7i	DECR	Ri	Decrement contents of register Ri
F0 hh ll	JUMP	addr	Jump to instruction at address indicated
F1 hh ll	JUMP	Z,addr	Jump to instruction at address indicated if zero flag is set
F2 hh ll	JUMP	NZ,addr	Jump to instruction at address indicated if zero flag is not set

* Ri (i =1, 2, 3, 4): General Purpose Registers

Now assume this microprocessor executes the following program part starting at address 0000H:

Address	Instruction
0000	LOAD R1,1200
0003	LOAD R2,2
0005	LOAD R4,1201
0008	LOAD A,R4
0009	ADD A,R1
000A	DECR R2
000B	JUMP NZ,0009
000E	STORE A,2000
0011	LOAD A,5000
0014	....

The logic analyzer has been set up to capture state data. The state data captured for the program part shown above looks as follows:

CAPS DISPLAY Feb 16 1993 11:02a

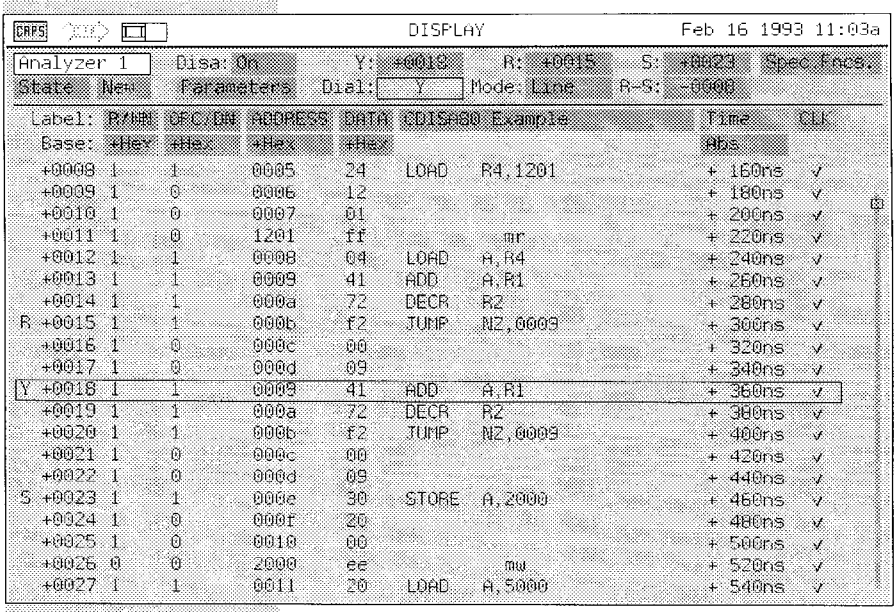
Analyzer 1 Disa: Off Y: +0018 R: +0015 S: +0023 Specifics

State Mem Parameters Dial: Y Mode: Line R-S: -0008

Label:	R/WN	OFF/ON	ADDRESS	DATA	Time	CLK
Base:	+Hex	+Hex	+Hex	+Hex	Abs	
+0008	1	1	0005	24	+ 160ns	v
+0009	1	0	0006	12	+ 180ns	v
+0010	1	0	0007	01	+ 200ns	v
+0011	1	0	1201	ff	+ 220ns	v
+0012	1	1	0008	04	+ 240ns	v
+0013	1	1	0009	41	+ 260ns	v
+0014	1	1	000a	72	+ 280ns	v
R +0015	1	1	000b	f2	+ 300ns	v
+0016	1	0	000c	00	+ 320ns	v
+0017	1	0	000d	09	+ 340ns	v
Y +0018	1	1	0009	41	+ 360ns	v
+0019	1	1	000a	72	+ 380ns	v
+0020	1	1	000b	f2	+ 400ns	v
+0021	1	0	000c	00	+ 420ns	v
+0022	1	0	000d	09	+ 440ns	v
S +0023	1	1	000e	30	+ 460ns	v
+0024	1	0	000f	20	+ 480ns	v
+0025	1	0	0010	00	+ 500ns	v
+0026	0	0	2000	ee	+ 520ns	v
+0027	1	1	0011	20	+ 540ns	v

This acquisition is available on disk with the file name EXAMPLE.NEW

With this package we will develop step by step a custom disassembler for the hypothetical microprocessor. The resulting display after disassembly will show:

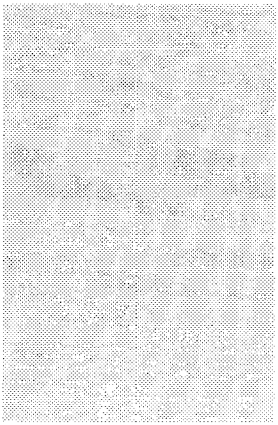


The disassembler description file is available on disk with the filename EXAMPLE.DSC.
Use the CDISA80 compiler to create a loadable disassembler EXAMPLE.DIS.

Disassembly Process

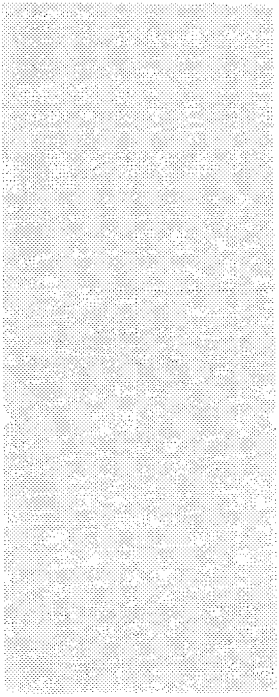
The disassembly process involves the following basic steps:

1. Initialize sample pointer. The disassembler uses a sample pointer to keep track of it's position in the measurement. The initialization of this pointer is done by the disassembler.



2. If the sample pointed to by the sample pointer is an opcode, then print the opcode in the disassembler output column and proceed with step 3.
   If the sample pointed to by the sample pointer is a memory read or write print "mr" or "mw" in the disassembler output column and proceed with step 4.
3. If the instruction, identified by the opcode, requires one or more additional bytes, then read those bytes (samples) and print them after the opcode. Proceed with step 4.
4. Increment the sample pointer. So it will point to the first sample of the next instruction.
5. If more samples are left, then go back to step 2. Else, stop.

Lookup Table



The whole disassembly process as described above can be controlled by means of a lookup table. The values of the opcodes are used as entry points in this table. The different entries in the table determine the string to be printed, and indicate whether additional bytes (samples) are required with this opcode to complete the instruction. The third step of the disassembly process indicated above is thus programmed as a command in the lookup table for those opcodes requiring additional bytes. If no additional bytes are required the disassembly for the instruction is finished.

The lines for our lookup table are built as follows:

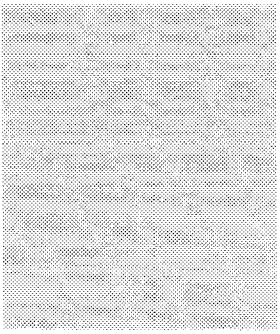
If ((OPC/DN = 1) AND (DATA = <code>)) DO <commands>

Using the Disassembler Description Language syntax this is written as:

(OPC/DN = 1, DATA = <code>) ! <commands> !

The brackets "(" and ")" relate to the word "If". The "," is used for "AND" and the pair of exclamation marks "!" replaces the word "DO".

The commands we need for our example are: "Print string", "Read next sample" and "Print sample value".



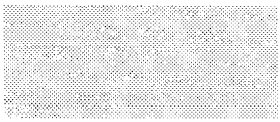
The "Print string" command is given by putting the string to be printed between quotes "".

The command "Read next sample" simply is "NEXT" or "next".

The command "Print sample value" is more complicated and requires some additional explanation. For now we assume this command is "V".

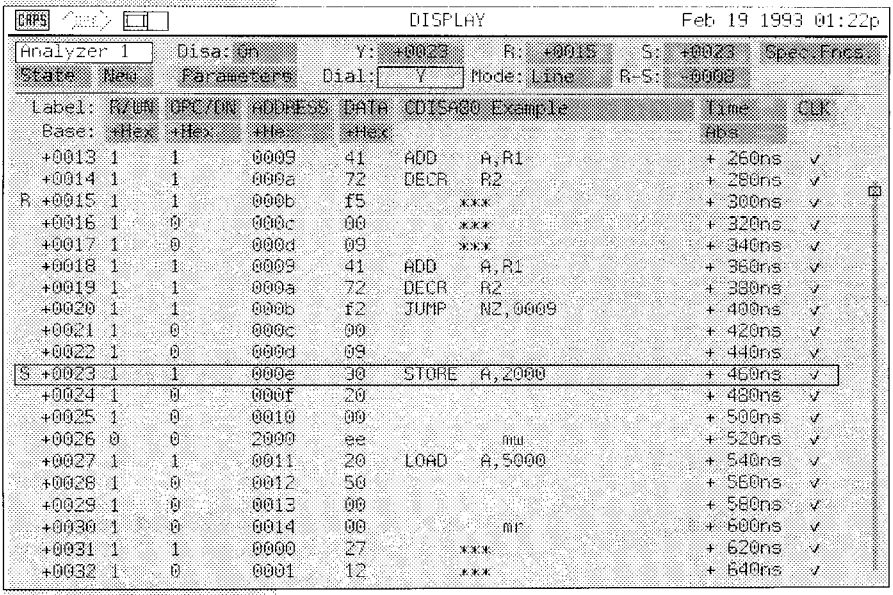
The lookup table sofar looks as follows (note that hexadecimal values are preceded by 0x; binary values are preceded by 0b):

```
(OPC/DN = 0b0,  R/WN = 0b0)   | "mw" !
(OPC/DN = 0b0,  R/WN = 0b1)   ! "mr" !
(OPC/DN = 0b1,  DATA = 0x01)  ! "LOAD     A,  R1" !
(OPC/DN = 0b1,  DATA = 0x02)  ! "LOAD     A,  R2" !
(OPC/DN = 0b1,  DATA = 0x03)  ! "LOAD     A,  R3" !
(OPC/DN = 0b1,  DATA = 0x04)  ! "LOAD     A,  R4" !
(OPC/DN = 0b1,  DATA = 0x10)  ! "LOAD     A," NEXT V !
(OPC/DN = 0b1,  DATA = 0x11)  ! "LOAD     R1," NEXT V !
(OPC/DN = 0b1,  DATA = 0x12)  ! "LOAD     R2,"NEXT V !
(OPC/DN = 0b1,  DATA = 0x13)  ! "LOAD     R3,"NEXT V !
(OPC/DN = 0b1,  DATA = 0x14)  ! "LOAD     R4,"NEXT V !
(OPC/DN = 0b1,  DATA = 0x20)  ! "LOAD     A," NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x21)  ! "LOAD     R1," NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x22)  ! "LOAD     R2,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x23)  ! "LOAD     R3,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x24)  ! "LOAD     R4,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x30)  ! "STORE    A," NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x31)  ! "STORE    R1," NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x32)  ! "STORE    R2,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x33)  ! "STORE    R3,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x34)  ! "STORE    R4,"NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0x41)  ! "ADD      A,R1" !
(OPC/DN = 0b1,  DATA = 0x42)  ! "ADD      A,R2" !
(OPC/DN = 0b1,  DATA = 0x43)  ! "ADD      A,R3" !
(OPC/DN = 0b1,  DATA = 0x44)  ! "ADD      A,R4" !
(OPC/DN = 0b1,  DATA = 0x71)  ! "DECR     R1" !
(OPC/DN = 0b1,  DATA = 0x72)  ! "DECR     R2" !
(OPC/DN = 0b1,  DATA = 0x73)  ! "DECR     R3" !
(OPC/DN = 0b1,  DATA = 0x74)  ! "DECR     R4" !
(OPC/DN = 0b1,  DATA = 0xF0)  ! "JUMP     "   NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0xF1)  ! "JUMP     Z," NEXT V NEXT V !
(OPC/DN = 0b1,  DATA = 0xF2)  ! "JUMP     NZ,"NEXT V NEXT V !
```



Note: A lookup table is scanned from top to bottom. If a value of a sample is not found in the lookup table the disassembler will print "***" in the disassembler out-

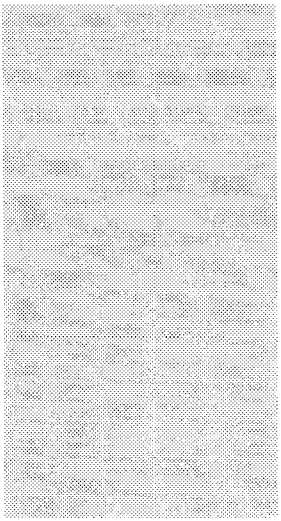
put column indicating that the disassembler lost synchronization status (See also chapter 7, "Disassemblers", of your PM 3580/PM 3585 User Manual). The disassembler will then proceed with the next sample.
This display below shows that two undefined opcodes (F5 and 27) are detected.



This acquisition is available on disk with the file name EXAMPLE.ERR

Before discussing how the "Print sample value" command is actually solved it is worthwhile to point out another basic element of the Disassembler Description Language: the Index Table. This table allows for a more compact notation of our lookup table.

Index Table and Local Variables



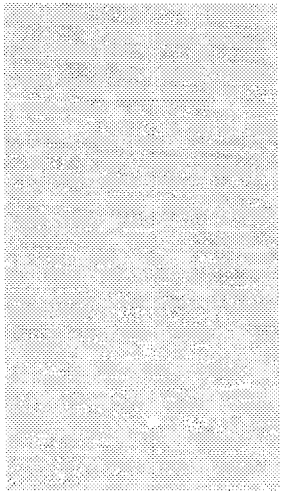
Our hypothetical microprocessor has its registers coded in consecutive numbers: 0 = A; i = Ri. This allows for a more compact notation of the lookup table. We would like to use the least significant part of the sample value (opcode) to select the text ("A" or "Ri") to be printed. This can be done using a local variable and an index table. The local variable should get the value of the least significant part of the sample. This variable is then used to select the proper string from the index table. The index table which we will give the name "R", looks as follows:

Index Table R:

```
! "A"    !
! "R1"   !
! "R2"   !
! "R3"   !
! "R4"   !
```

Thus R[0] relates to "A", R[1] to "R1", etc.

Local Variable

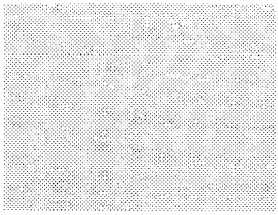


Local variables can be used to pass values from the acquired data (sample) to a command. The value assigned to a local variable is extracted from the sample value. A sample value can be seen as a bit pattern. To indicate which part of the bit pattern is to be assigned to a local variable, you have to place brackets "[" and "]" around positions in the pattern. More than one occurrence of a pair of brackets may be specified in a single pattern. Local variables corresponding with the identifiers $1, $2, $3, etc. are assigned from left to right. Up to nine local variables ($1 to $9) may be used.

The following notation:

```
(OPC/DN = 0b1,  DATA = 0x0[.])  ! "LOAD    A," R[$1] !
```

would assign the value 1 to $1 if DATA = 0x01, resulting in the print out of the string "LOAD A,R1". Likewise the val-



ue 2 would be assigned to $1 if DATA = 0x02, resulting in the print out of the string "LOAD A,R2".

The concept of using an index table in combination with a local variable for printing register names can also be used for "JUMP" instructions.
This leads to the following reduced lookup table which we will name 'Opcode':

Lookup Table Opcode:

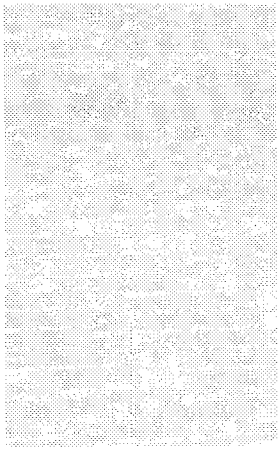
```
(OPC/DN = 0b0,  R/WN = 0b0)    ! "mw" !
(OPC/DN = 0b0,  R/WN = 0b1)    ! "mr" !
(OPC/DN = 0b1,  DATA = 0x0[.])  ! "LOAD      A,"  R[$1] !
(OPC/DN = 0b1,  DATA = 0x1[.])  ! "LOAD      "    R[$1] "," NEXT  V !
(OPC/DN = 0b1,  DATA = 0x2[.])  ! "LOAD      "    R[$1] "," NEXT  V NEXT  V !
(OPC/DN = 0b1,  DATA = 0x3[.])  ! "STORE     "    R[$1] "," NEXT  V NEXT  V !
(OPC/DN = 0b1,  DATA = 0x4[.])  ! "ADD       A,"  R[$1] !
(OPC/DN = 0b1,  DATA = 0x7[.])  ! "DECR      "    R[$1] !
(OPC/DN = 0b1,  DATA = 0xF[.])  ! "JUMP      "    C[$1] NEXT  V NEXT  V !
```

Index Table R:

```
! "A"   !
! "R1"  !
! "R2"  !
! "R3"  !
! "R4"  !
```

Index Table C:

```
! ""    !
! "Z,"  !
! "NZ," !
```



Note: If an index table is accessed outside its index boundaries the disassembler will print "***" in the disassembler output column indicating that the disassembler lost synchronization status. The disassembler will then proceed with the next sample.

Note: In the custom disassembler created so far opcode "00" will lead to the display of opcode "LOAD A, A" which is not defined for our microprocessor. Likewise illegal opcodes "40" and "70" are not detected.
To prevent this we need to check the value of the bitpattern before really using it as an index. Before describing how this can be done, we will first complete the lookup table commands; i.e. implement the "Print sample value" command.

Print Sample Value

The command NEXT causes the sample pointer to be incremented so that it points to the next sample. To print the value of (a part of) a sample a special print command, the format command, is available in the Disassembler Description Language. The syntax of this format command is:

```
<format command>    ::=  <format specifier>
<format specifier>  ::=  '%' <width and type>
<width and type>    ::=  'c' | <width> <type>
<width>             ::=  <decdigit>
                         | '0' <decdigit>
                         | <empty>
<type>              ::=  'b' | 'o' | 'd' | 'x'
```

In our example we will print the values in hexadecimal format using two positions with leading zeroes: %02x.

To indicate which part of the actual value in the bitpattern is to be printed, you have to place brackets "[" and "]" around positions in the bit pattern. The resulting command line now is:

```
(DATA = 0x[..])   ! "%02x" !
```

Because we should extract a local variable from this sample, another table is required to hold this command line. This could be a lookup table. The only purpose for the condition in the lookup would be to assign a value to a local variable. For this reason the index table may also contain an expression (like conditions in the lookup tables) before the commands. This expression is only used to describe which values must be assigned to local variables. Using the following index table 'V', the data can be printed.

```
Index Table V:
(DATA = 0x[..])       ! "%02x" !
(DATA = 0x[..])       ! "%02x" NEXT V[0] !
```

If we now replace "NEXT V" by "NEXT V[0]" and "NEXT V NEXT V" by "NEXT V[1]" in our lookup table "Opcode" we get the required result.

Together with the index table V described before our disassembler description so far looks as:

Lookup Table Opcode:

```
(OPC/DN = 0b0,  R/WN = 0b0)     ! "mw" !
(OPC/DN = 0b0,  R/WN = 0b1)     ! "mr" !
(OPC/DN = 0b1,  DATA = 0x0[.])  ! "LOAD   A," R[$1] !
(OPC/DN = 0b1,  DATA = 0x1[.])  ! "LOAD   "   R[$1] "," NEXT V[0] !
(OPC/DN = 0b1,  DATA = 0x2[.])  ! "LOAD   "   R[$1] "," NEXT V[1] !
(OPC/DN = 0b1,  DATA = 0x3[.])  ! "STORE  "   R[$1] "," NEXT V[1] !
(OPC/DN = 0b1,  DATA = 0x4[.] ) ! "ADD    A," R[$1] !
(OPC/DN = 0b1,  DATA = 0x7[.])  ! "DECR   "   R[$1] !
(OPC/DN = 0b1,  DATA = 0xF[.])  ! "JUMP   "   C[$1] NEXT V[1] !
```

Index Table R:

```
! "A"     !
! "R1"    !
! "R2"    !
! "R3"    !
! "R4"    !
```

Index Table C:

```
! ""      !
! "Z,"    !
! "NZ,"   !
```

Index Table V:

```
(DATA = 0x[..])     ! "%02x" !
(DATA = 0x[..])     ! "%02x" NEXT V[0] !
```

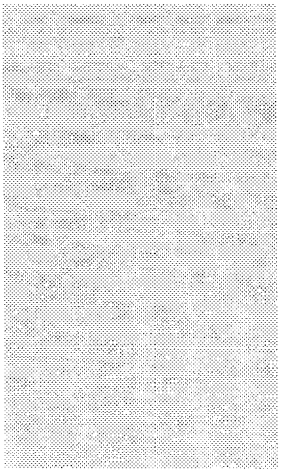
Detection of Illegal Opcodes

With the description file described above, opcode "00" results in 'LOAD A,A' which is an illegal instruction in our hypothetical microprocessor. Opcode "40" and "70" are incorrectly interpreted as well. To prevent this we need to check the register index ($1) before the index table R is accessed. To do this we need the following command line:

```
(reg_nr > 0)  ! R[reg_nr] !
```

Because we need a condition (reg_nr > 0), which may not be present in the command, another table is required. This table should be a lookup table which we will give the name "CheckR".

Global Variables



Local variables only exist on the current line of the table. In order to transfer the value of a local variable to another table or another line, it must be assigned to a global variable. Global variables are defined in the definition section (%% DEF) of the description file and can be used throughout the entire description file. For our purpose we will define a global variable "p".

Using the syntactical rules defined for the Disassembler Description Language we now get the tables described below. Keep again in mind that a lookup table is always scanned from top to bottom. If a value of a sample is not found in the lookup table the disassembler will print "***" in the disassembler output column indicating that the disassembler lost synchronization status (See chapter 7, "Disassemblers", of your PM 3580/PM 3585 User Manual). The disassembler will then proceed with the next sample.

Lookup Table Opcode:

```
(OPC/DN = 0b0,  R/WN = 0b0)     ! "mw" !
(OPC/DN = 0b0,  R/WN = 0b1)     ! "mr" !
(OPC/DN = 0b1, DATA = 0x0[.])   ! "LOAD   A,"   {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0x1[.])   ! "LOAD   "     R[$1] "," NEXT V[0]!
(OPC/DN = 0b1, DATA = 0x2[.])   ! "LOAD   "     R[$1] "," NEXT V[1]!
(OPC/DN = 0b1, DATA = 0x3[.])   ! "STORE  "     R[$1] "," NEXT V[1]!
(OPC/DN = 0b1, DATA = 0x4[.])   ! "ADD    A,"   {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0x7[.])   ! "DECR   "     {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0xF[.])   ! "JUMP   "     C[$1] NEXT V[1]!
```

Lookup Table CheckR:

```
(p > 0)  ! R[p] !
```

Index Table R:

```
! "A"  !
! "R1" !
! "R2" !
! "R3" !
! "R4" !
```

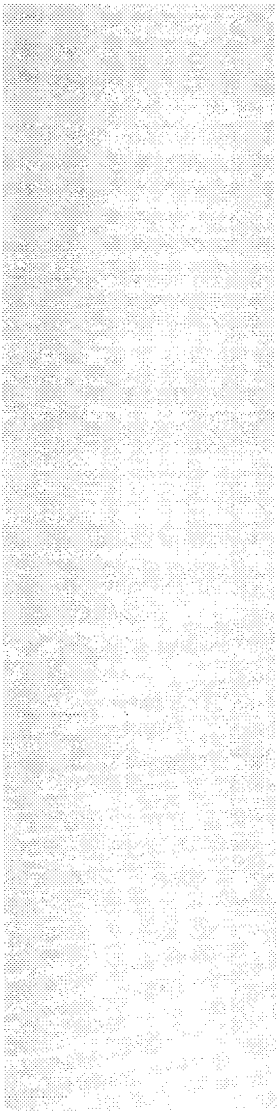
Index Table C:

```
! ""   !
! "Z," !
! "NZ," !
```

Index Table V:

```
(DATA = 0x[..])    ! "%02x" !
(DATA = 0x[..])    ! "%02x" NEXT V[0] !
```

Display Selection



The lookup and index tables described so far only contain commands for displaying strings and positioning the disassembler within the measurement. In order to support the display options available in the logic analyzer disassembler parameters popup menu, the custom disassembler provides commands to control the appearance of a sample in the display menu.

The custom disassembler has to know which samples belong to the current instruction; the program samples as well as data samples. This is done via display selection commands. To indicate which opcode samples belong to the instruction being decoded, the 'PROG' command has to be used. This command is required for all opcode samples except for the first one. Other display selection commands will be explained later. In our example the samples accessed are all part of the instruction. So after each 'NEXT' command the command 'PROG' should follow to indicate that the sample is a part of the instruction. This can be done within the 'V' index table commands because 'V' is called after each 'NEXT' command.

Note: 'NEXT' can not be done within the 'V' table because the local variable printed in the 'V' table should be extracted after the 'NEXT' command is executed. If 'NEXT' was placed in the 'V' table first the local variable was extracted and then the 'NEXT' command was executed resulting in an unwanted local variable value.

This results in the following table 'V':

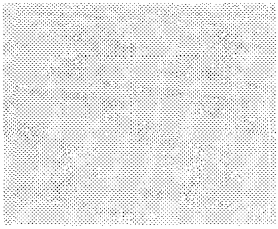
```
Index Table V:
(DATA = 0x[..])    ! PROG "%02x" !
(DATA = 0x[..])    ! PROG "%02x" NEXT V[0] !
```

Data Transfers

All samples accessed in the measurement will be displayed as an instruction opcode. However, the samples which meet one of the conditions in the first 2 lines of the lookup table 'Opcode' are no instruction opcodes but are data transfers according the 'Show Data Transfers' field in the disassembler parameters popup menu (See Chapter 7, "Disassembler display options", of your PM 3580/ PM 3585 User Manual). To treat these samples as data transfer instead of instruction opcodes the default ('PROG') must be replaced by other display selection commands. The display selection commands for these data transfers are 'MW' for memory write and 'MR' for memory read. The display selection commands for data transfers automatically result in a text in the disassembler column for the sample at which the command was given.
The first 2 lines in the lookup table now look like:

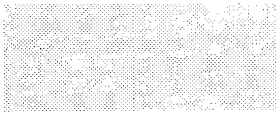
(OPC/DN = 0b0, R/WN = 0b0)! MW !
(OPC/DN = 0b0, R/WN = 0b1)! MR !

The difference is that the print commands "mw" and "mr" are replaced by display selection commands MW and MR which automatically result in the text 'mw' and 'mr' in the disassembler column.
The data transfers in our measurement (mr, mw) are a result of the execution of the preceding instruction. These data transfers are also a part of the instruction being decoded.
So we should add the data transfer samples to the instruction when we are decoding the instruction. The instructions which result in a data transfer are 'LOAD' and 'STORE' which have a directly addressed memory location as operand (instructions with opcode 0x2. and 0x3.).
The data transfers appear immediately after the instruction so a NEXT command must be done if we found such an instruction. Then the check if it really is a data transfer should be done in a separate lookup table.



For the LOAD instruction the resulting data transfer is a memory read. The STORE instruction results in a memory write. This results in two additional lookup tables: Check-Mw and CheckMr. One for checking memory write actions and one for checking memory read actions. The 2 lines in the lookup table 'Opcode' explained above are moved to their respectively tables: CheckMw or CheckMr.
The resulting disassembler description then looks as:

```
Lookup Table Opcode:
(OPC/DN = 0b1, DATA = 0x0[.])    ! "LOAD    A,"   {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0x1[.])    ! "LOAD     "    R[$1] "," NEXT V[0]!
(OPC/DN = 0b1, DATA = 0x2[.])    ! "LOAD     "    R[$1] "," NEXT V[1] NEXT CheckMr!
(OPC/DN = 0b1, DATA = 0x3[.])    ! "STORE    "    R[$1] "," NEXT V[1] NEXT CheckMw!
(OPC/DN = 0b1, DATA = 0x4[.])    ! "ADD     A,"   {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0x7[.])    ! "DECR     "    {p=$1} CheckR  !
(OPC/DN = 0b1, DATA = 0xF[.])    ! "JUMP     "    C[$1] NEXT V[1]!


Lookup Table CheckMr:
(OPC/DN = 0b0, R/WN = 0b1)    ! mr  !


Lookup Table CheckMw:
(OPC/DN = 0b0, R/WN = 0b0)    ! mw !


Lookup Table CheckR:
(p > 0)  ! R[p] !


Index Table R;
! "A"  !
! "R1" !
! "R2" !
! "R3" !
! "R4" !


Index Table C:
! ""   !
! "Z," !
! "NZ," !


Index Table V:
(DATA = 0x[..])    ! PROG "%02x" !
(DATA = 0x[..])    ! PROG "%02x" NEXT V[0] !
```



Note: If the lines in the CheckMr and CheckMw tables were placed in the lookup table "Opcode" the data transfers are not related to a 'PROG'-sample. The first sample on decoding an instruction, is a data

transfer sample. This would result in adding the suffix string "(unrel.)" after the data transfer text 'mw' or 'mr' (See Chapter 7, "Disassembler bus transfers and disassembler status" of your PM 3580/PM 3585 User Manual).

Completing the Disassembler Description File

The Disassembler Description File is now almost complete. Three topics are still missing:

- Declarations of tables and global variables
- Label and clock definitions
- Start command

These will be described below.

Declaration of Tables and Global Variables

Global variables used in the disassembler description file must be declared first in the so called DEF section of the file. Using the syntactical rules defined for the Disassembler Description Language for our example this looks as follows:

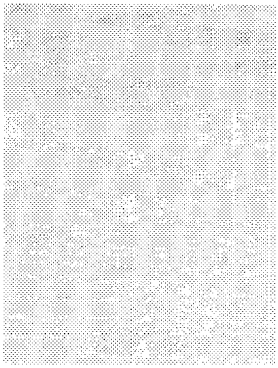
```
%%DEF
int   p;
```

Likewise tables used in the disassembler description file must be declared first in the so called EQU section of the file. For our example this looks as follows:

```
%%EQU

/* lookup tables */
LT:  Opcode, CheckMr, CheckMw, CheckR

/* index tables  */
IT:  R, C, V
```

Note: Any text enclosed with the /* and */ delimiters is regarded as comment. Comment may be placed anywhere in a description file. Also note that the local variables "$i" do not need to be separately declared.

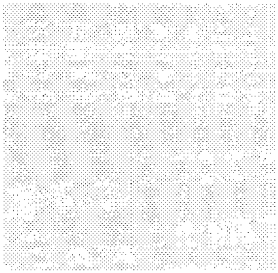
Label and Clock Definitions



Until now we have been using labels without describing which channels of the logic analyzer are assigned to these labels and what the attributes of these labels are. The same applies for the state clock definitions. In other words: we have not yet defined the settings of the Format menu.

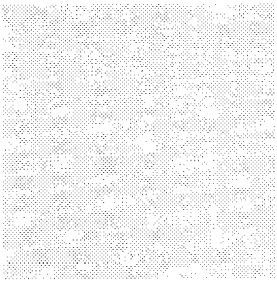
This is done using the assignment structures (threshold, clock, label and clock sequence) of the FORMAT section defined in the Disassembler Description Language. The FORMAT section for our example is shown with the complete description file below. Refer to the chapter "Disassembler Description Language Reference" of this manual for a description of the syntax.

Start Section



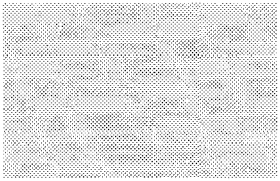
Within the disassembler description file a START section is required. The commands contained in this section are the commands the disassembly process starts with, in order to process a complete microprocessor instruction. In general only one command is given in the start section. This command is the name of the lookup table to start disassembly with. Note that all global variables except one (see global variables; static char. description in chapter 3) are set to zero each time the start table is entered.

Complete Example



Adding the declarations, definitions and start section as described above to our disassembler description results in the following completed disassembler description file for our hypothetical microprocessor.

Note: According to the Disassembler Description Language, label identifiers may not contain certain characters like for example "/" or spaces. Label names however, may contain these special characters. In the following description the "/" is therefore

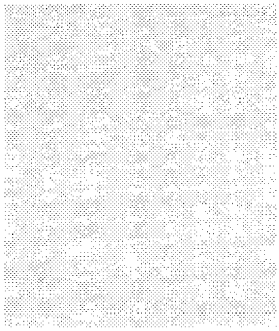


removed from the label identifiers (both in the declarations of the identifiers and references to them), but not from the label names.

Note: A table description starts with "%%<table name>" according to the syntactical rules rather than "Table <table name>:" as we did until now.

```
%% DEF
int     p;

%% EQU
LT :   Opcode, CheckMr, CheckMw, CheckR      /* lookup tables */
IT :   R, C, V                               /* index tables  */

%% FORMAT
logo: "CDISA80 Example"

head: "CDISA80 Example"  20   /* Label for disassembler output column and width of column */
pods: threshold =   { {TTL, TTL},
                      {TTL, TTL}
                    }

clock: clk =        { name = "CLK",
                      edge = rising,
                      channel = 31,
                      qualifier= { channels = { 30 },
                                   levels = { high }
                                 }
                    }

label: QUAL =       { name = "QUAL",
                      display = timing,
                      channels = { 30 }
                    }
label: RWN =        { name = "R/WN",
                      channels = { 24 }
                    }
 label: OPCDN =     { name = "OPC/DN",
                      channels = { 25 }
                    }
label: ADDRESS =    { name = "ADDRESS",
                      channels = {15, 14, 13, 12, 11, 10, 9, 8,
                                  7,  6,  5,  4,  3,  2, 1, 0 }
                    }
label: DATA  =      { name = "DATA",
                      channels = { 23, 22, 21, 20, 19, 18, 17, 16 }
                    }
```

```
clockseq: SEQ= {clk}

%% START
! Opcode !

%% Opcode
(OPCDN = 0b1, DATA = 0x0[.])    ! "LOAD   A,"    {p=$1} CheckR !
(OPCDN = 0b1, DATA = 0x1[.])    ! "LOAD   "      R[$1] "," NEXT V[0]!
(OPCDN = 0b1, DATA = 0x2[.])    ! "LOAD   "      R[$1] "," NEXT V[1] NEXT CheckMr!
(OPCDN = 0b1, DATA = 0x3[.])    ! "STORE  "      R[$1] "," NEXT V[1] NEXT CheckMw!
(OPCDN = 0b1, DATA = 0x4[.])    ! "ADD    A,"    {p=$1} CheckR !
(OPCDN = 0b1, DATA = 0x7[.])    ! "DECR   "      {p=$1} CheckR !
(OPCDN = 0b1, DATA = 0xF[.])    ! "JUMP   "      C[$1] NEXT V[1]!

%% CheckMw
(OPCDN = 0b0, RWN = 0b0)        ! mw !

%% CheckMr
(OPCDN = 0b0, RWN = 0b1)        ! mr !

%% CheckR
(p > 0)  ! R[p] !

%% R
! "A"     !
! "R1"    !
! "R2"    !
! "R3"    !
! "R4"    !

%% C
! ""      !
! "Z,"    !
! "NZ,"   !

%% V
(DATA = 0x[..])     !PROG "%02x" !
(DATA = 0x[..])     !PROG "%02x" NEXT V[0] !
```



About Processing Speed

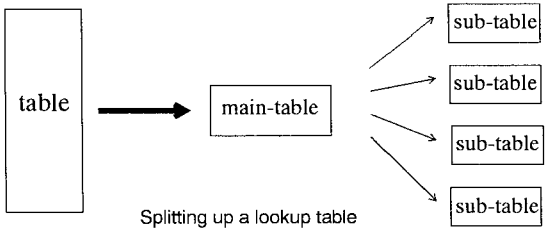
Creating a description file that will serve your purpose should be fairly easy. Almost every description file will have one main (lookup) table that is used to check the op-code and select the appropriate actions.
The performance of a custom disassembler can be improved in various ways. We will name a few.

Rearrange Lookup Tables

Sometimes lookup tables tend to get very large (more than 50 or 100 lines). Since the lookup tables are scanned every time from top to bottom you can improve the disassembler performance by putting often used instructions at the top of the table so they are recognized very quickly.

Splitting up Lookup Tables

Performance can also be improved by splitting a large lookup table into a few smaller (sub-)tables.
Then create a main-table which decides to what sub-table to jump to.



Splitting up a lookup table

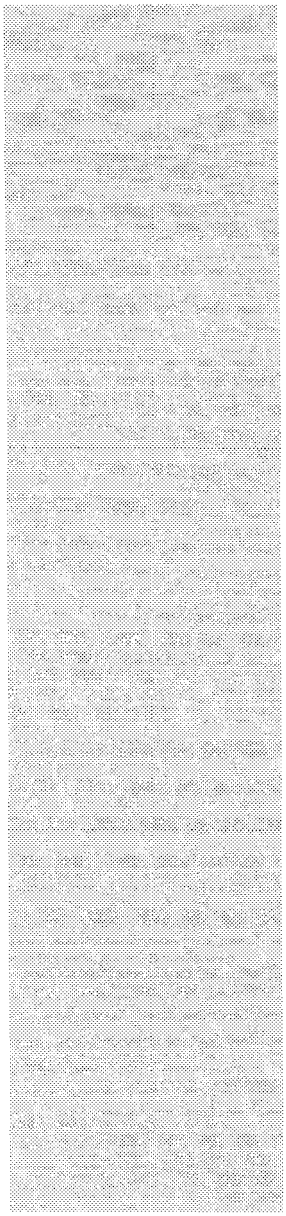
Usage of Index Tables

Since index tables are directly accessible, a disassembler will be faster when using these type of tables wherever possible.

Chapter 3

Disassembler Description Language Reference

Introduction 3-3
File Structure 3-5
  The Declarative Part 3-6
  The Tabular Part 3-7
%% DEF Section 3-8
  Global Variables 3-8
  Constants 3-9
%% EQU Section 3-9
%% FORMAT Section 3-11
  Logo Definition 3-11
  Header Definition 3-12
  Pod Threshold Definition 3-13
  Symbolic Output Control 3-14
  Synchronization Blocksize Definition 3-14
  Clock Definition 3-16
  Label Definition 3-22
  Clock Sequence Definition 3-29
  Tab Settings 3-31
%% START Section 3-32
Tabular Section (%%<name>) 3-33
  Lookup Tables (LT) 3-33
  Index Tables (IT) 3-33
General Elements 3-35
  Lines 3-35
  Comments 3-35
  Spaces and Tabs 3-36
  Upper and Lower Case Characters 3-36
  Conditions and Commands 3-37
Conditions 3-37
  Pattern Conditions 3-37
  Relational Conditions 3-40
  Clock Sequence Conditions 3-41
  AND-ing and OR-ing of Conditions 3-42
Pattern Expression 3-42



Commands 3-43
Display Selection Commands 3-44
PROG 3-45
Instructions 3-46
UNUSED 3-46
SKIP 3-46
MR 3-47
MW 3-48
IOR 3-48
IOW 3-48
Positioning in a Measurement 3-49
GOTO [i] 3-49
TELL 3-51
NEXT 3-51
PREV 3-52
GOTOPART[i] 3-52
TELLPART 3-53
NEXTPART 3-53
PREVPART 3-54
UNGET 3-55
Instructions 3-56
Print Commands 3-57
Special Commands 3-59
UNPUT 3-60
ERROR 3-60
Transfer Control to other Tables 3-60

Introduction

The disassembler description file describes the specific properties of the microprocessor or bus used.
It is created by the user as a DOS text file using an editor or word processor.

Any word processor capable of producing DOS text files such as PC-write, WordPerfect, MS-Word, PFS Write or Wordstar will do.

While reading through this chapter you will notice that some of the syntax used for a .DSC file resembles the C programming language. If you are familiar with programming in C, you will have little trouble learning to use this syntax. If you are not, don't worry. Only a very small set of commands from the C language is used so the learning curve will be very short.

BNF-Notation

In each section, the syntax of specific language elements will be defined using a Backus-Nauer-Format (BNF) notation. It is important that these definitions are interpreted correctly. If you are not familiar with this way of describing a syntax study the first examples below very carefully. Comments are added to the definitions below to help you get familiar with the notations used:

Syntax

<bindigit> ::= 0 | 1
A binary digit can be '0' or '1'.

<octdigit> ::= 0..7
An octal digit can be anything from'0 ' up to '7'.

<decdigit> ::= 0..9
A decimal digit can be anything from '0' up to '9'.

<hexdigit> ::= <decdigit> | a..f | A..F
A hexadecimal digit can be a decimal digit or anything from 'a' up to 'f' (upper or lower case).

<decnumber> ::= <optsign><decdigits>
A decimal number has an optional sign followed by a list of decimal digits.

<optsign> ::= <empty> | '+' | '-'
An optional sign is empty, a '+' or a '-'.

<decdigits> ::= <decdigit> | <decdigit> <decdigits>
A list of decimal digits consists of one decimal digit, or one decimal digit followed by a list of decimaldigits

<letter> ::= a..z | A..Z
A letter can be anything from 'a' up to 'z' (upper or lower case).

<name> ::= <letter><extname>
A name starts with a letter and is followed by an *extname*

<extname> ::= <empty>
| <letter><extname>
| <digit> <extname>

An *extname* is empty, or a letter or a digit followed by an *extname*.

<empty> ::= Empty is ... empty.

File Structure

The structure for a .DSC file is shown below.

```
%%DEF
  .....
  /* Declarations and definitions of global variables and
     constants */

%%EQU
  .....
  /* Allocation of symbolic names to tables */

%%FORMAT
  .....
  /* Analyzer channel allocation and grouping */

%%START
  .....
  /* Table section */

/* TABULAR SECTION */
%%.....
  .....
  /* User defined disassembler tables */
```

The structure can be divided in two parts:

- the **declarative** part containing the %%DEF, the %%EQU and the %%FORMAT sections
- the **tabular** part containing the %%START section and the user defined disassembler tables.

Each section of this file will be described in detail in this chapter.

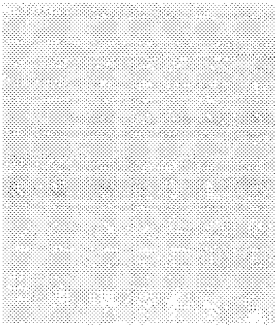
The declarative part of a description source file precedes the tabular part and contains general information that the CDISA80 compiler needs concerning the structure of the input data file (in particular the channel allocation and grouping). It is also used to declare variables and constants that will be used in the tabular part of the disassembler description file.

Header Lines

A declarative section and a disassembler table begin with a header line. A header line starts with a double percentage mark (%%) and is followed by an identifier. No leading spaces and/or tabs are allowed in header lines. The identifier is not case sensitive so it may consist of upper or lower case characters. Four predefined header lines are available:

%%DEF	followed by definitions and declarations of global variables and constants
%%EQU	followed by user defined names for disassembler tables
%%FORMAT	followed by channel allocation and grouping, disassembler name and general disassembler output control
%%START	followed by disassembler table part

Other identifiers can be defined by the user in the %%EQU section. These identifiers declare the tables defined in the tabular part. Also alternative label names (aliases) can be defined in this section. The predefined header lines must precede the user defined header lines of the tabular part. The START header line must be the first header line in the tabular part. The sequence of the other three predefined header lines in the first part as well as the sequence of the user defined header lines in the second tabular part, is free.



Syntax

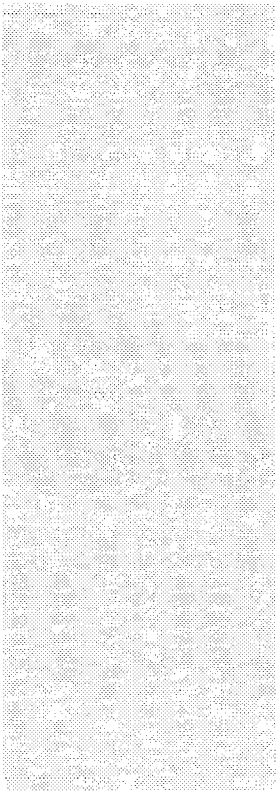
%%<identifier>
...

Examples

%%DEF
...
%%FORMAT
...

Note: No spaces or tabs preceding '%%' are allowed.

The Tabular Part



The tabular part contains the START table, lookup tables (LT) and index tables (IT). Each entry in a lookup table consists of a condition followed by a command chain. The Index table entries contain an optional expression (like a condition in lookup tables) followed by a command chain. The function of each of the lookup and index tables is explained in section "Tabular Section (%%<name>)".

%% *DEF Section*

The definition section is used to declare global variables and constants that are used for the remainder of the program. This section is preceded by the %%DEF" header line and concluded by the next header line. The declaration conforms to the C language syntax for variable types, however, only a restricted set of types is allowed.

Global Variables

Syntax

For global variables the permitted variable types are:

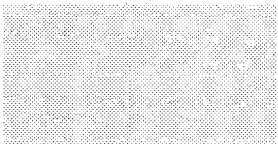
<var declaration> :: = <type> <names> ';'

<type> ::=	'char'	(8 bits)
	| 'int'	(16 bits)
	| 'long'	(32 bits)
	| 'unsigned'	(16 bits)
	| 'unsigned int'	(16 bits)
	| 'unsigned char'	(8 bits)
	| 'static char'	(8 bits)

<names> ::= <name> | <name>, <names>

All global variables except the 'static char' variable are set to zero each time the START table is entered. The 'static char' variable can therefore be used to report a status or a small value to a following instruction. The instruction which uses the value in the 'static char' variable should be located in the near distance of the instruction which sets the value. See also 'synchronization blocksize' if you are using the static char variable in instructions in the far distance of the instruction which sets the value.

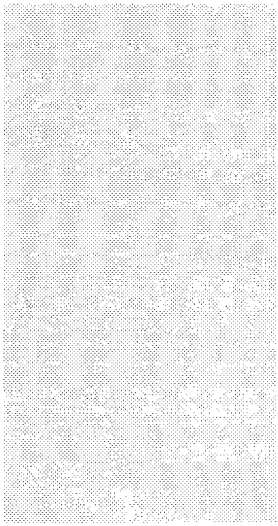
The number of global variables is limited to 32 variables and one 'static char' variable.



Examples

```
char c;
int offset, offset2;

static char status;
```

Constants



The syntax to declare constants is:

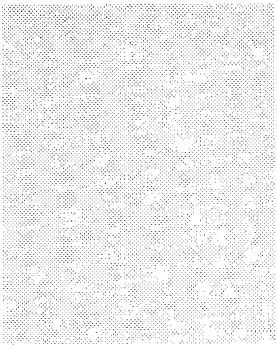
```
#define  <name>    <value>

<value>  ::=       <decdigits>
                 | '0x'  <hexdigits>
                 | '0X'  <hexdigits>
                 | '0b'  <bindigits>
                 | '0B'  <bindigits>
                 | '0o'  <octdigits>
                 | '0O'  <octdigits>
                 | '<ASCII-char>'
```

Examples

```
#define MAX      255
#define DUMMY    0
```

Note: Constants that are referred to with user defined symbolic names always are 16 bits. As a result no long type constant definitions are allowed.

%% *EQU Section*



The equate section is used to define aliases for user defined labels and to declare disassembly tables. Two different types of tables can be used in the tabular section.

Syntax

```
%%EQU

<equate definition> ::=
      <lookup table definition>
    | <index table definition>
    | <label alias definition>

<lookup table definition> ::=  'LT' ':' <name list>
```

<name list> ::= <name> | <name> ',' <name list>

<index table definition> ::= 'IT' ':' <name list>

<label alias definition> ::=
<label id> ':' <label alias id list>

<label alias id list> ::=
<label alias id>
| <label alias id> ',' <label alias id list>

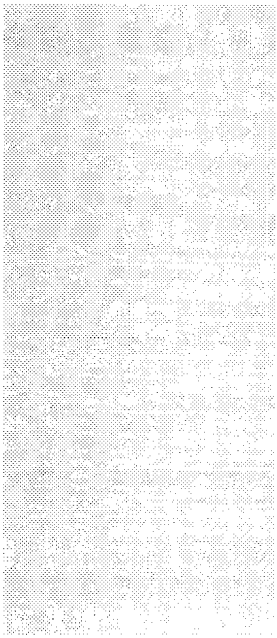
<label alias id> ::= <name>

Examples

```
%% EQU

LT      : MAIN, Commands, BUSCYCLES
          /* lookup tables */

IT      : ADDR[MODE, REGSET
          /* index tables */

BWE     : stat
          /* status lines */

ADR     : address
          /* address lines */

DATA    : opcode, byte
          /* data lines */
```

Note: More than one label can be used to denote the same group of channels. Based on the last equate statement above the data for a disassembler state can be accessed using either the label "DATA" or "opcode" or "byte".

%% FORMAT Section



The format section is preceded by the "%%FORMAT" header line and terminated by the next header line. This section in fact allows you to specify the fields of the FORMAT menu of the logic analyzer. It is used to inform the disassembler which channels were used to capture the data from the CPU and how these channels should be grouped and displayed. Furthermore, a disassembler name can be specified, which will appear at the top of the disassembler output column in the state display.

The format section allows you to define:

- Disassembler name (logo)
- Disassembler output column title (header)
- Pod thresholds
- Symbols
- Synchronization block size
- Tab settings
- Clocks and their attributes
- Labels and their attributes
- Valid clock sequences

Logo Definition

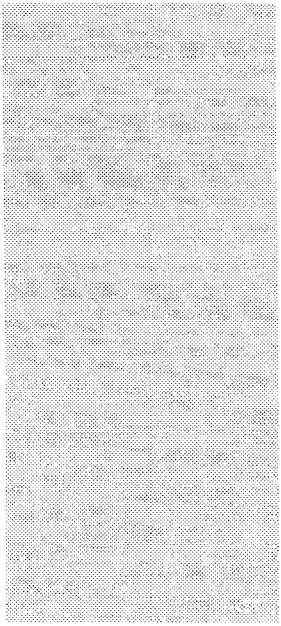
The logo definition allows you to specify a string that will be shown in the logo which pops up when the custom disassembler is loaded. The string is displayed on the first line in this pop up immediately following the standard text:

"PF8629/30 - Custom Disassembler".

Syntax

< logo definition> ::=
'logo' ':' <logo specifier>

<logo specifier> ::=
<string> (max. 37 characters)



<string> ::=
'"' <string symbol list> '"'

<string symbol list> ::=
<string symbol>
| <string symbol> <string symbol list>

<string symbol> ::=
<type1_string symbol>
| '*' | ',' | ';' | ':' | '#' | '<' | '>'
| '{' | '}' | '(' | ')' | '[' | ']' | '+'
| '-' | '=' | '\' | ''' | '"' | '"' | '/'
| '!' | '?' | '%' | '$' | '@' | '&' | '.'
| '~' | '^' | '|'

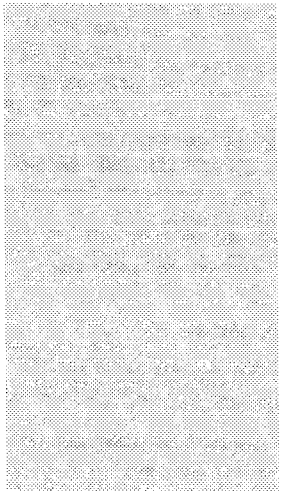
<type1_string symbol> ::=
'a' | 'b' | .. | 'z' | 'A' | .. | 'Z' | '_'

Example

logo: "CDISA80 Example"

Note: Specification of the logo is optional. If no logo is specified the logo field in the popup is left blank.

Header Definition



The header definition is used to specify the title for the disassembler output column and the width of that column. The width is specified as a number of characters.

Syntax

< header definition> ::= 'head' ':' <header specifier>

<header specifier> ::= <string> <column width>
(max. 64 characters)

<column width> ::= <decimal number>

Example

head: "CDISA80 Example" 20

Note: Specification of the header is optional. If no header is specified the default value: "Custom Disassembler" 25 is taken.

Pod Threshold Definition

The pod threshold definition is used to specify the threshold values for the pods. Thresholds specified are assigned to Pod (n) high, low; Pod (n-1) high, low etc.

Syntax

```
<threshold definition>          ::=
            'pods: threshold =' <threshold specifier>

<threshold specifier>           ::=
            '{' <threshold pod specifier list> '}'

<threshold pod specifier list>  ::=
             <threshold pod specifier>
            | <threshold pod specifier> ','
             <threshold pod specifier list>

<threshold pod specifier>       ::=
            '{' <threshold group> , <threshold group> '}'

<threshold group>               ::=
            TTL | ECL | <var> <threshold value>

<var>                           ::= VAR | <empty>

<threshold value>               ::= -3.0 .. 12.0 (unit = V)
```

Example

The following definition:

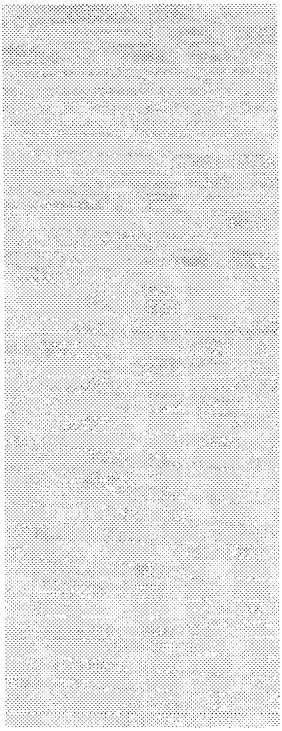
```
Pods: threshold = { {ECL, TTL}
                    { -2, +5}
                  }
```

assigns the following threshold values to the pods:

Pod 2	bit 15-8	:	ECL
Pod 2	bit 7-0	:	TTL
Pod 1	bit 15-8	:	-2V
Pod 1	bit 7-0	:	+5V

Note: Definition of pod threshold values is optional. If thresholds are not specified the default value TTL is taken.

Symbolic Output Control



The symbolic output control is used to specify if values selected for symbolic printout in the disassembler column should be displayed in symbolic format or not. This is a global switch for symbolic support of the disassembler. The symbolic output control is optional. If no symbolic output control is given the disassembler will not produce symbolic output. The disa parameters popup in the display menu will have an (additional) Options field. This Options field can be used to select if the disassembler output uses symbolic values or not.

Syntax

<symbolic output definition> ::=
'symbolic' ':' <symbolic output specifier>

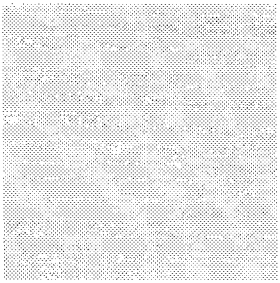
<symbolic output specifier> ::=
'yes' | 'no'

Note: If logic analyzer system software BEFORE 2.01 is used on the PM3580/PM3585 logic analyzer the custom disassembler does not support symbolic output.

Example

symbolic: yes

Synchronization Blocksize Definition



For synchronizing the disassembler the measurement is cut into parts. The size of the parts can be adjusted by changing the block-size definition of the disassembler. The synchronization blocksize definition is optional. The default blocksize is 32 disassembler states. The user can change it by specifying 'BLOCKSIZE: <blocksize>

Syntax

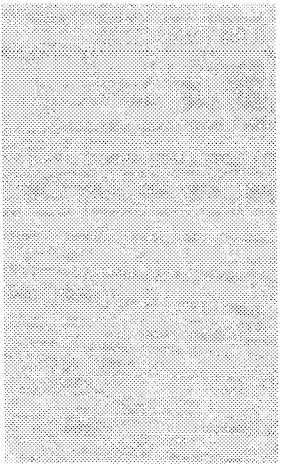
<block size definition> ::= 'BLOCKSIZE' ':' <block size>

<block size> ::= '16'
| '32'
| '64'
| '128'
| '256'
| 'max'
| 'min'
| 'maximum'
| 'minimum'

If the synchronization blocksize is increased the time to synchronize one block will increase. When max or maximum is used the disassembler will synchronize the complete measurement. Min or minimum will set the block size to 16.
Because of synchronizing the measurement in parts the user should guarantee a consistent value of the static char variable after some instructions (together at most 2 times BLOCKSIZE disassembler states). If this is not done the disassembler could give different output when moving from disassembler state 0 to a certain disassembler state and when moving from the last acquired disassembler-state back to the mentioned disassembler state.

Example:

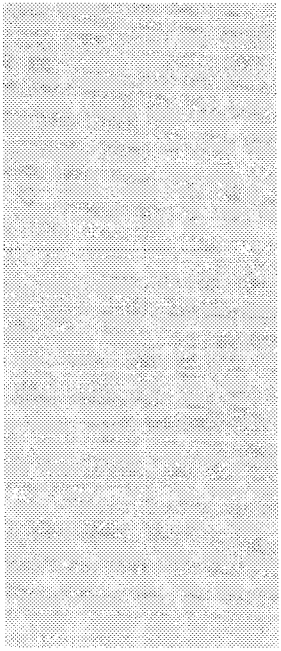
Suppose the disassembler is synchronizing a measurement for displaying state line 600. To achieve this the disassembler internally uses a part of the measurement. The static char variable is initially 0 at the first state line of this measurement part. Within the measurement part the value of the static char can be changed. To get a consistent disassembler output the value of the static char should have the same value at state line 600 regardless of the first state line number in the measurement part.
The result on the logic analyzer display menu could give different output for state line 600 in both cases. If the same value of the static char variable can not be guaranteed for



state line 600 the BLOCKSIZE parameter should be increased.
The static char variable should therefore best be used only to report a sort of status to an instruction in the near distance.
The synchronization BLOCKSIZE should have a value equal to or above the maximum value of:

a) The number of disassembler states which the disassembler needs to assure a consistent value of the static char variable (the global variable over the total measurement).
b) The average number of disassembler states between the lowest and the highest numbered disassembler state on the display divided by 2.
c) 16.

Clock Definition



The clock definition is used to specify which channel is used for a clock, what edge should be used and the polarity. It is also used to specify the clock qualifiers and clock attributes.

Syntax

```
<clock definition>          ::=
            'clock' ':' <clock id> '=' <clock specifier>

<clock id>                  ::= <identifier>

<identifier>                ::= <name>

<clock specifier>           ::=
            '{' <clock parameter list> '}'

<clock parameter list>      ::=
              <clock parameter>
            | <clock parameter> ',' <clock parameter list>

<clock parameter>           ::=
              <clock name definition>
            | <clock polarity definition>
            | <clock channel definition>
            | <clock edge definition>
```

| <clock qualifier definition>
| <clock merge definition>
| <clock timing definition>
| <clock required definition>
| <clock display definition>

Example

```
clock: clk ={  name = "CLK",
               edge = rising,
               channel = 31,
               timing = dataandtrigger,
               display = stateandtiming,
               polarity = +,
               required = yes,
               mergeclock = none,
               qualifier = { channels = { 30 },
                             levels = { high }
                             delays = { 0 },
                             required = yes
                           }
            }
```

Note: The clock definitions must precede the definitions of labels and clock sequences.

Note: The <clock id> must be used in clock sequence definitions; not the clock name specifier. This is because the clock name specifier may contain any character including for example "/", spaces, etc.

Note: Below the syntax for the parameters is defined. The specification of most parameters is optional. If a parameter is **optional** this is indicated in the left margin by "(**O**)". The default value for those optional parameters is also shown in the left margin.
If no additional description for a parameter is given its purpose can be derived from the PM 3580/ PM 3585 Reference Guide, chapters "Format Menu"" and "Clock Attributes Menu" respectively. A further explanation can also be found in the PM 3580/PM 3585 User Manual, chapter "State Clocks".

Clock Name

<clock name definition> ::=
'name' '=' <clock name specifier>

<clock name specifier> ::=
<type1_string>

<type1_string> ::=
'"' <type1_string symbol list> '"'

<type1_string symbol list> ::=
<type1_string symbol>
| <type1_string symbol> <type1_string symbol list>

<clock definition> ::=
'polarity' '=' <clock polarity specifier>

Clock Polarity (O)
Default: +

<clock polarity specifier> ::=
<polarity>

<polarity> ::=
'positive' | 'negative' | '+' | '-'

Clock Channel

<clock channel definition> ::=
'channel' '=' <clock channel specifier>

<clock channel specifier> ::=
<channr>

<channr> ::=
0..95

Clock Edge

<clock edge definition> ::=
'edge' '=' <clock edge specifier>

<clock edge specifier> ::=
<edge>

<edge> ::=
'rising' | 'falling' | 'any'

Clock Merge (O)
Default: none

The clock merge definition can be used to specify whether the samples captured with this clock should be displayed on the same line as the samples captured by another clock (compare the "*Display on same line as*" field in the Clock Attributes Menu).

Syntax

<clock merge definition> ::=
'mergeclock' '=' <clock merge specifier>

<clock merge specifier> ::=
<clock id> | 'none'

Clock Timing (O)
Default: dataandtrigger

<clock timing definition> ::=
'timing' '=' <clock timing specifier>

<clock timing specifier> ::=
'none' | ' trigger' | 'dataandtrigger'

Clock Display (O)
Default: stateandtiming

The "clock display definition" specifies whether the clock signal should be shown in the state display only, the timing display only, both displays or in neither of those two. Note that you can always add a label later in your display menu using the *INSERT* key on your logic analyzer.

Syntax

<clock display definition> ::=
'display' '=' <clock display specifier>

<clock display specifier> ::=
'none' | 'state' | 'timing' | 'stateandtiming'

Clock Required (O)
Default: yes

The "clock required definition" specifies whether this state clock is required for disassembly.

Syntax

<clock required definition> ::=
'required' '=' <clock required specifier>

<clock required specifier> ::=
'yes' | 'no'

Note: The clock definitions for clocks which are required should precede the clock definitions for the non-required clocks.

Note: When a disassembler is loaded it is checked whether sufficient resources are available. If only sufficient resources are available for those signals required by the disassembler, the disassembler is still loaded. Setups for the other microprocessor signals will then not be loaded. Please also refer to your PM 3580/PM 3585 User Manual, chapter "Disassemblers".

Clock Qualifier (O)
Default: no qualifier

<clock qualifier definition> ::=
'qualifier' '=' <clock qualifier specifier>

<clock qualifier specifier> ::=
'{' <clock qualifier parameter list> '}'

<clock qualifier parameter list> ::=
<clock qualifier parameter>
| <clock qualifier parameter> ','
<clock qualifier parameter list>

<clock qualifier parameter> ::=
<clock qualifier channels definition>
| <clock qualifier levels definition>
| <clock qualifier delays definition>
| <clock qualifier required definition>

Qualifier Channels

<clock qualifier channels definition> ::=
'channels' '='
<clock qualifier channels specifier>

<clock qualifier channels specifier> ::=
'{' <channel list> '}'
<channel list> ::=
<channr> | <channr> ',' <channel list>

Note: If the channel list contains more than one channel number (channr) the numbers should be specified in a descending order, i.e. highest channel number first.

Qualifier Levels

<clock qualifier levels definition> ::=
'levels' '='
<clock qualifier levels specifier>

<clock qualifier levels specifier> ::=
'{' <level list> '}'

<level list> ::=
<level> | <level> ',' <level list>

<level> ::=
'high' | 'low'

Qualifier Delays (O)
Default: 0

Immediately after the system software has been loaded a calibration procedure is executed. This procedure ensures that the propagation delay is the same on all channels and

that the set-up and hold times for the logic analyzer meet their specification. Please refer to your PM 3580/PM 3585 "Service Manual" for a detailed discussion about set-up and hold times.
Using the "clock qualifier delay definition" you can slightly shift the time window ($=t_{su}+t_h$) for a specific qualifier with respect to the edge of the clock. A positive delay value will increase the set-up time and decrease the hold time for that qualifier (shift the time window to the left). A negative delay will decrease the set-up time and increase the hold time for that qualifier (shift the time window to the right). One delay step represents a value between 0.5 ns min. and 2 ns max. For most microprocessors the default value (0) is required.

<clock qualifier delays definition> ::=
'delays' '='
<clock qualifier delays specifier>

<clock qualifier delays specifier> ::=
'{' <delay list> '}'

<delay list> ::=
<delay value>
| <delay value> ',' <delay list>

<delay value> ::=
-2 | -1 | 0 | 1 | 2

Qualifier Required (O)
Default: yes

<clock qualifier required definition> ::=
'required' '='
<clock qualifier required specifier>

<clock qualifier required specifier> ::=
'yes' | 'no'

Note: The qualifier definitions for the qualifiers which are required for a clock should precede the definitions for the qualifiers not required for that clock.

Label Definition

The label definition is used to specify which channel is used for a label, what edge should be used and the polarity. It is also used to specify the label qualifiers and label attributes and label symbols.

Syntax

```
<label definition> ::=
      'label' ':' <label id> '=' <label specifier>

<label specifier> ::=
      '{' <label parameter list> '}'

<label id> ::=
      <identifier>

<label parameter list> ::=
      <label parameter>
    | <label parameter>  ','  <label parameter list>

<label parameter> ::=
       <label name definition>
     | <label polarity definition>
     | <label channels definition>
     | <label delays definition>
     | <label radix definition>
     | <label clocks definition>
     | <label required definition>
     | <label timing definition>
     | <label display definition>
     | <label symbolic definition>
     | <label symbol viewsize definition>
     | <label symbol definition>
     | <label part definition>
     | <label type definition>
```

Example

```
label: ADDRESS = { name = "ADDRESS",
                   timing = dataandtrigger,
                   display = stateandtiming,
                polarity = +,
                channels = {15, 14, 13, 12, 11, 10, 9, 8,
                            7, 6, 5, 4, 3, 2, 1, 0 },
                delays = {0, 0, 0, 0, 0, 0, 0, 0,
                          0, 0, 0, 0, 0, 0, 0, 0 },
                radix = hex,
                required = yes,
                symbolic = yes,
                symbol viewsize = unique,
                symbol = { "stack", 0xf...},
                symbol = { "I/O Init", 0x430},
                symbol = { "RESET", 0x0, 0x7},
                clocks = { clk }
        }
```

Note: A maximum of 32 labels may be specified in a .DSC file.

Note: The <label id> must be used in label conditions; not the label name specifier. This is because the label name specifier may contain any character including for example "/", spaces, etc..

Note: Below the syntax for the parameters is defined. The specification of most parameters is **optional**. If a parameter is optional this is indicated in the left margin by "(**O**)". The default value for those optional parameters is also shown in the left margin. If no additional description for a parameter is given its purpose can be derived from the PM 3580/PM 3585 Reference Guide, chapters "Format Menu"" and "Label Attributes Menu" respectively. A further explanation can also be found in the PM 3580/PM 3585 User Manual, chapter "State Clocks".

Label Name

<label name definition> ::=
'name' '=' <label name specifier>

<label name specifier> ::=
<type1_string>

Label Polarity (O)
Default: +

<label polarity definition> ::=
'polarity' '=' <label polarity specifier>

<label polarity specifier> ::=
<polarity>

Label Channels

<label channels definition> ::=
'channels' '=' <label channels specifier>

<label channels specifier> ::=
'{' <channel list> '}'

Note: If the channel list contains more than one channel number (channr) the numbers should be specified in a descending order, i.e. highest channel number first.

Label Delays (O)
Default: 0

Immediately after the system software has been loaded a calibration procedure is executed. This procedure ensures that the propagation delay is the same on all channels and that the set-up and hold times for the logic analyzer meet their specification. Please refer to your PM 3580/PM 3585 "Service Manual" for a detailed discussion about set-up and hold times.
Using the "label delay definition" you can slightly shift the time window ($=t_{su}+t_h$) for a specific label with respect to the edge of the clocks. A positive delay value will increase the set-up time and decrease the hold time for that label (shift the time window to the left). A negative delay will decrease the set-up time and increase the hold time for that label (shift the time window to the right). One delay step represents a value between 0.5 ns min. and 2 ns max. For most microprocessors the default value (0) is required.

Syntax

<label delays definition> ::=
'delays' '=' <clock delays specifier>

<label delays specifier> ::= '{' <delay list> '}'

Label Radix (O)
Default: hex

The "label radix definition" specifies in which radix (base) the label data should be shown in the Trace Menu and the State Display.

Note that you can always change this base on the Trace Menu and the State Display.

Syntax

<label radix definition> ::=
'radix' '=' <label radix specifier>

<label radix specifier> ::= <radix>

<radix> ::= 'bin' | 'oct' | 'dec' | 'hex' | 'ascii'

Label Clocks (O)
Default: all clocks defined

<label clocks definition> ::=
'clocks' '=' <label clocks specifier>

<label clocks specifier> ::=
'{' <clock id list> '}' | '{' 'none' '}'

<clock id list> ::=
<clock id> | <clock id> ',' <clock id list>

The "label required definition" specifies whether this label is required for disassembly.

Syntax

Label Required (O)
Default: yes

<label required definition> ::=
'required' '=' <label required specifier>

<label required specifier> ::= 'yes' | 'no'

Note: The label definitions for labels which are required should precede the label definitions for the non-required labels.

Note: For only 16 labels the label required definition may be 'yes'.

Note: When a disassembler is loaded it is checked whether sufficient resources are available. If only sufficient resources are available for those signals required by the disassembler, the disassembler is still loaded. Setups for the other microprocessor signals will then not be loaded. Please also refer to your PM 3580/PM 3585 User Manual, chapter "Disassemblers".

Label Timing (O)
Default: dataandtrigger

<label timing definition> ::=
'timing' '=' <label timing specifier>

<label timing specifier> ::=
'none' | 'trigger' | 'dataandtrigger'

Label Display (O)
Default: stateandtiming

The "label display definition" specifies whether the label data should be shown in the state display only, the timing display only, both displays or in neither of those two. Note that you can always add a label later on to your display using the *INSERT* key on your logic analyzer.

Syntax

<label display definition> ::=
'display' '=' <label display specifier>

<label display specifier> ::=
'none' | 'state' | 'timing' | 'stateandtiming'

Label Part (O)
Default: 1

The "label part definition" defines the number of parts in which this label is split when used in the tabular section. The label parts definition should be used when more than one instruction can occur in one disassembler state. E.g. the 68030 microprocessor has a 32 bit data bus. The minimum instruction size is 16 bits and as such it is possible to have 2 instructions in one disassembler state. To handle this kind of microprocessors it is possible to define (only) one label which can be accessed in equally sized parts.

Syntax

<parts definition> ::= 'parts' '=' <number of parts>

<number of parts> ::= '1' | '2' | '4'

Example

parts = 2 /* for 68030 data label */

Label Type (O)
Default: big endian

The label type definition should be used together with the label parts definition to define the type of the parts label. Two types are possible:

- big endian format(e.g. Motorola 680x0 format)

The most significant part in the label is the first part used by the microprocessor and is part '0' for the disassembler.

- little endian format(e.g. Intel 80x86 format)

  The least significant part in the label is the first part used by the microprocessor and is part '0' for the disassembler.

If no label type definition is given the default label type is big endian.

Syntax

```
<label type definition> ::=
      'type' '=' <label type>

<label type> ::=
       'little'
      | 'little_endian'
      | 'big'
      | 'big_endian'
```

Example

```
type = little
```

Label Symbolic (O)
Default: no

The "label symbolic definition" specifies if the label should be displayed symbolic in the trace and display menus.

Syntax

```
<label symbolic definition> ::=
      'symbolic' '=' <symbolic output specifier>
```

Label Symbol Viewsize (O)
Default: maximum

The label symbol viewsize definition specifies the symbol viewsize for this label.

Syntax

```
<label symbol viewsize definition> ::=
        'viewsize' '=' <symbol viewsize specifier>
```

<symbol viewsize specifier> ::=
'max'
| 'maximum'
| 'uniq'
| 'unique'
| <decimal number>
(min.1, max. 32)

Example

viewsize = unique
viewsize = 12

Label Symbol (O)
Default: no symbols

The "label symbol definition" specifies the symbolic names for label values and ranges of label values. A label may have several label symbol definitions.

Syntax

<label symbol definition> ::=
'symbol' '=' <symbol specifier>

<symbol specifier> ::=
'{' <symbol name> ',' <symbol value or range> '}'

<symbol name> ::=
<type1_string>

<symbol value or range> ::=
<symbol value>
| <symbol range>

<symbol value> ::=
<bitpattern>
| <decimal number>

<symbol range> ::=
<symbol value> ',' <symbol value>

Examples

symbol = { "Super Data" , 5 }
symbol = { "stack" , 0xf... }
symbol = { "reset" , 0x0, 0x7 }

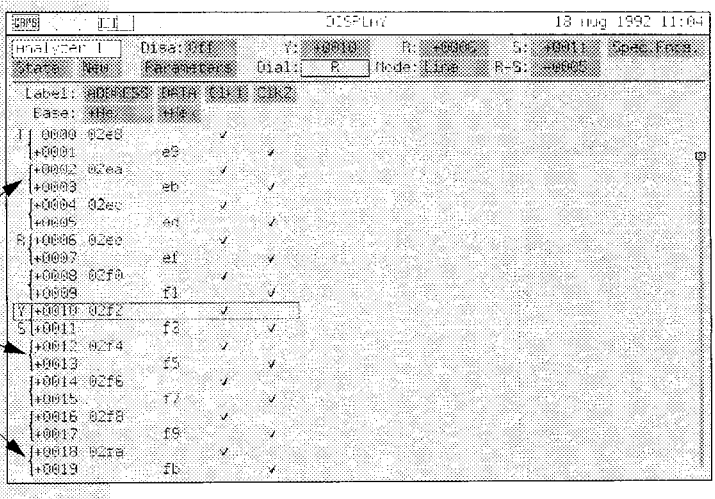
Clock Sequence Definition

Disassembler State

Depending on the microprocessor one or more state clocks may be required to capture the state information for the microprocessor. If more than one state clock is required it is possible that the samples captured by two or more clocks together form one logical state (referred to as *disassembler state*) provided that the samples were captured with a specific sequence of those clocks.

As an example consider a microprocessor having a multiplexed address/data bus. The addresses are valid for CLK1 and the data is valid for another clock, CLK2. The data is read from or written to the address immediately preceding the data on the multiplexed address/data bus. The samples captured in sequence by CLK1 and CLK2 respectively therefore together form one logical state (disassembler state).

In the state display below the disassembler states are indicated.



Clock Sequence

Clock sequence definitions are used to define specific sequences of clocks. Consecutive samples, together represent a valid disassembler state if and only if the sam-

ples were captured by a defined clock sequence. Only clocks that are specified in the disassembler description file are effective in the clock sequence.

In most cases one clock sequence is used for each clock. Such a clock sequence only contains one clock id:

clockseq: SEQ1 = { clka }

where SEQ1 is the clock sequence id, which can be used as a condition in look-up tables, and clka is the clock id of a previously defined clock. Each required clock id must be used at least in one clock sequence.

Clock sequences may be parts of each other. The disassembler always tries to recognize the longest sequence.

Syntax

<clock sequence definition> ::=
'clockseq' ':' <clock sequence id> '='
<clock sequence specifier>

<clock sequence id> ::=
<identifier>

<clock sequence specifier> ::=
'{' <clock id list> '}' | '{' 'none' '}'

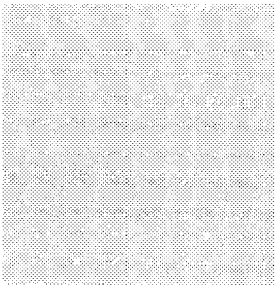
<clock id list> ::=
<clock id>
| <clock id> ',' <clock id list>

Examples

For the example given above the following clock sequence should be specified

clockseq: disastate = {CLK1, CLK2}

As another example consider the 8085 microprocessor from Intel. For this microprocessor three clock sequences need to be defined as can be derived from the description in the PM 3580/PM 3585 User Manual, chapter "State Clocks", section "Multiplexed Busses". These clock sequences are (it is assumed that ALE, RDN, WRN and INTAN are defined as clock id's, not only clock names):



clockseq: read = {ALE, RDN}

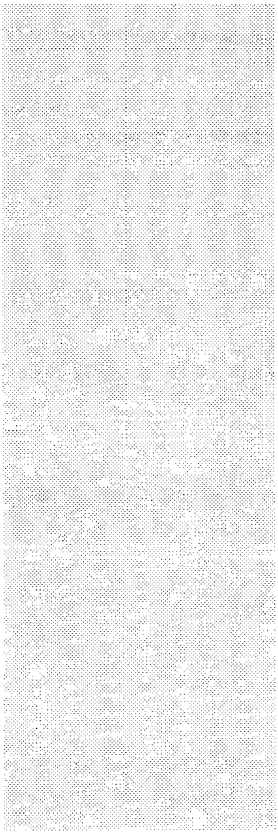
clockseq: write = {ALE, WRN}

clockseq: inta = {ALE, INTAN}

See also the "Clock Merge" section for displaying the 8085 disassembler states on one line instead of split over two lines: one line for each clock.

Note: No defaults are specified for clock sequences. At least one clock sequence should be defined.

Tab Settings



When some opcodes are longer than others a clean screen layout can still be created when tabs are used. For this purpose tab settings can be specified in the description file using the 'tab' command. The arguments the TAB command takes have two meanings. The first arguments separated by commas, specify absolute tab positions. The last argument, separated from the others by a space, specifies the tab spacing (number of characters) to the next position, starting from the last absolute tab position. The default first tab position is 7. The default tab space is 8 characters.

Syntax

<tab definition> :: =
'TAB' ':' < tab pos> ' ' <tab spacing>

<tab pos> :: = <decdigits> ',' tab pos
| <decdigits>

<tab spacing> :: = <empty>
| <decdigits>

Example

TAB : 10, 20 7
sets tab stops at positions 10, 20, 27, 34, etc.

%% START Section

The start section is the last predefined header line and contains the first phase of the disassembly process. This section is made up of an optional pattern condition and a mandatory command chain. The pattern condition qualifies each line for disassembly. If the current line that is disassembled does not meet this qualification, the disassembler looses instruction synchronization and will display "***" in the disassembly output column. Thus, the pattern condition can be used to synchronize the disassembly process. The second part consists of a command chain.

No relational conditions, clock sequence conditions or local variables are permitted in the START section.

The disassembler state accessed when entering the START-table is the first state of the instruction.

Syntax

%%START

<start definition>::=
    <opt pattern condition list> <command chain>

<opt pattern condition list>::=
    '(' <pattern condition list> ')' | <empty>

<pattern condition list>::=
    <pattern condition>
    |<pattern condition> ',' <pattern condition list>

Examples

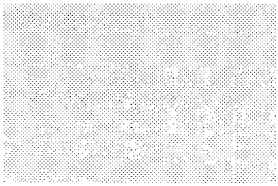
```
%%START
(status=0b0100....)    ! MAIN !
```

or

```
%%START
! LT0 !
```

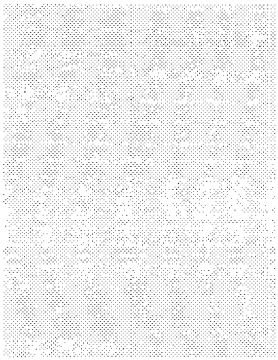
Note: No relational conditions, clock sequence conditions or local variables are permitted in the START section.

Tabular Section (%%<name>)



The tabular section contains lookup tables (LT) and/or index tables (IT). Each entry in a lookup table consists of a condition followed by a command chain. Index table entries consist of an optional pattern expression and a command chain. The function of these tables is explained in the following paragraphs.

Lookup Tables (LT)



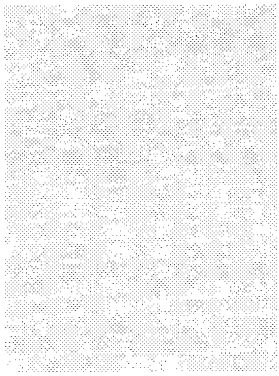
Lookup tables are used to scan for a specific condition. **These tables are scanned from top to bottom.** The first entry that contains a true condition will cause execution of the command chain for that entry.

Syntax

```
%%<name>
<condition>     <command chain>
<condition>     <command chain>
  ....
```

Note: The number of lookup tables cannot exceed 64.

Index Tables (IT)



Index tables are direct access tables. A call to an index table is made with an offset parameter which must be a variable (global or local) or a constant. The entry pointed to by this parameter is accessed and the command chain found at that position is processed.

The optional pattern expression at an index table entry is used only to obtain values from the current disassembler state or from global variables.

Conditions are not allowed in the index table.
Examples for the use of the pattern expression in an index table are:

- calculate destination addresses for branch instructions.
- calculate the result of mathematical instructions on immediate data.
- extract the index to be used for calling another index table.

Syntax

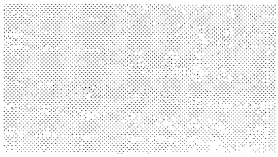
```
%%<name>
<index table line>
<index table line>
...

<index table line> ::=
        <command chain>
      | <pattern expression> <command chain>

<pattern expression> ::=
      '(' <pattern expression list> ')'

<pattern expression list> ::=
        <pattern>
      | <pattern> ',' <pattern expression list>

<pattern>                ::=
                    <label or variable> '=' <bitpattern>
                  | <label or variable> '==' <bitpattern>


<label or variable>      ::=
                      <label id>
                    |  <variable name>
```

<label id> is the label identifier.
<variable name> is the name of a global variable. Please refer to section "Label Definition" or "Global Variables" for more details

Note: A label-id as parameter for an index table call is not allowed.

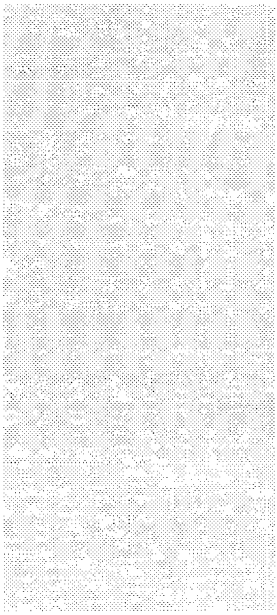
Note: The maximum number of index tables is 64.

General Elements



The CDISA80 compiler expects line oriented input files. This is one reason why you may want to use a programmers text editor instead of a word processor. Each language element or statement must be confined to one line.

Lines



Lines can be up to 255 characters long but it is good practice to restrict each line to what fits on the screen (80 characters).
Language elements can be extended over more than one line using the continuation character (backslash '\'). This allows for statements that exceed the screen limit (80 characters), the maximum line length limit (255 characters) or for formatting practices that make the source files more readable.

Syntax

<1..255 characters> CR/LF

continuation character : \

Examples

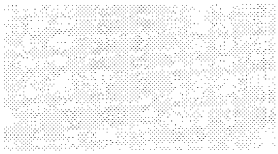
single line statement :

```
(opc=0xff)  !"ADD INT BCD\t"  PREV V[3] GOTO[5]!
```

multi line statement :

```
(opc=0x4.)  !"MOVE REG\t"  REGADR V[1] \
                next GOTO[2] "internal"!
```

Comments



Comments can be placed anywhere in a description file where spaces or tabs are permitted, provided they are enclosed with the /* and */ delimiters. The use of comments throughout the description file is highly recommended for

maintenance purposes. If in the future you ever have to upgrade to a microprocessor with an enhanced instruction set, it will be so much easier to modify a well documented source file.

Syntax

/* <any number of characters> */

Examples

/* This is a comment */

/* This comment extends over more than one line and
also has some tabs included */

Note: Continuation characters are not required for multi-line comments since anything between comment delimiters (/* ... */) is ignored by the compiler.

Spaces and Tabs

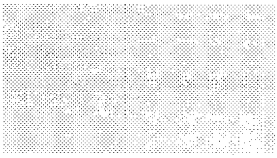
Like most compilers, the CDISA80 compiler program is not sensitive to spaces and tabs in source files. Consequently, spaces and tabs can be used freely throughout the source file for formatting purposes and to enhance legibility.

The only exception to this are the header lines (lines starting with %%..) which must start at the beginning of a line and strings where spaces and tabs are interpreted as "characters".

Upper and Lower Case Characters

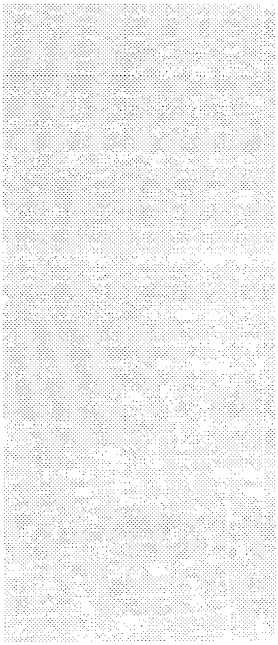
Commands may be entered in any mixture of upper and lower case. The compiler interprets upper and lower case characters literally in strings as well as variable names.

Conditions and Commands



In general, a description file consists of tables. Each table contains lines consisting of a condition or expression and a command chain. In the following paragraphs we will discuss the generics of the conditions, expressions and the command chains.

Conditions

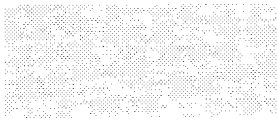


A condition is always enclosed in brackets "(.....)" and typically starts at the beginning of each line in a table. Leading spaces or tabs are allowed. A condition consists of a combination of pattern conditions, relational conditions, and/or clock sequence conditions, separated by commas. Pattern conditions, relational conditions and clock sequence conditions are introduced in the following paragraphs.
In lookup tables conditions are evaluated. When the condition evaluates to TRUE, the corresponding command chain is executed.

Syntax

```
<condition>              ::=
                    '(' <prc-condition list> ')'

<prc-condition list>     ::=
                    <prc-condition>
                  | <prc-condition> ',' <prc-condition list>

<prc-condition>          ::=
                    <pattern condition>
                  | <relational condition>
                  | <clock sequence condition>
```

Pattern Conditions



Pattern conditions have two functions. One is to compare individual bits to a value of 0, 1 or "don't care". The other is to extract values of one or more bit fields in a label or variable.

Syntax

```
<pattern condition>    ::=      <pattern>

<pattern>              ::=
                        <label or variable> '=' <bitpattern>
                      | <label or variable> '==' <bitpattern>

<label or variable>    ::=
                        <label id>
                      | <variable name>
```

Bit Patterns

'=' and '==' both mean 'equal to'.
A <bitpattern> is used to express a value in hex, binary or octal format. It must exactly match the length of the label or global variable. This bit pattern can contain any number of periods ('.') as don't cares.

Repetitions of two or more identical digits or periods in a bit pattern expression can be abbreviated with the "<i>n" structure, where n stands for the number of times the digit is to be repeated and "i" stands for the digit or period to be repeated. This allows for a more compact notation of long binary strings. The repetition factor has to be separated from any subsequent digits by means of the underscore character ('_'). The underscore can be used at any time in a bit pattern for spacing purposes. Spaces and tabs, however, are not allowed in bit patterns.

Syntax

```
<bitpattern>      ::=  '0x'<hexpattern>
                     | '0X'<hexpattern>
                     | '0b'<binpattern>
                     | '0B'<binpattern>
                     | '0o'<octpattern>
                     | '0O'<octpattern>

<hexpattern>      ::=  <hexdigit+>
                     | <hexdigit+>  <hexpattern>
                     | <hexpattern>  <hexdigit+>
                     | '[' <hexpattern> ']'
                     | <hexpattern> '_'
```

```
                        | '_' <hexpattern>
                        | '<' <hexdigit+> '>' <decdigits> '_'

<hexdigit+>     ::=  <hexdigit> | '.'

<binpattern>    ::=  <bindigit+>
                   | <bindigit+> <binpattern>
                   | <binpattern> <bindigit+>
                   | '[' <binpattern> ']'
                   | <binpattern> '_'
                   | '_' <binpattern>
                   | '<' <bindigit+> '>' <decdigits> '_'

<bindigit+>     ::=  <bindigit> | '.'

<octpattern>    ::=  <octdigit+>
                   | <octdigit+> <octpattern>
                   | <octpattern> <octdigit+>
                   | '[' <octpattern> ']'
                   | <octpattern> '_'
                   | '_' <octpattern>
                   | '<' <octdigit+> '>' <decdigits> '_'

<octdigit+>     ::=  <octdigit> | '.'
```

'[' <pattern> ']' are local variable assignments. They are described in the next paragraph.

Examples

```
(A = 0xff..)                /* 0b11111111xxxxxxxx */
(A = 0o3.14)                /* 0b011xxx001100     */
(A = 0b10<.>4_0.1<1>4. )    /* 0b10xxxx0x11111x   */
(A = 0b<1>11_0101)          /* 0b111111111110101  */
```

Local Variables

Local variables are used to pass values from the acquired data or global variables to the command chain. Values of bit fields in the bit pattern of the conditional section are assigned to local variables. To indicate what part of the actual value in a bit pattern is to be assigned to a local variable, brackets ('[' and ']') are placed around positions in the pattern. More than one occurrence of these bracket pairs can occur in a single string. Local variables are assigned

to identifiers $1, $2, $3 etc. from left to right. Up to 9 local variables ($1 to $9) may be used.

Local variables are useful for extracting addressing information that is embedded in instruction codes. Examples are internal register addresses that are commonly embedded in register move instructions. The local variable can be used as an offset in an index table containing the mnemonics for the various internal registers. Local variables are also used in tables for the output of acquisition data into the disassembled text. Examples are immediate data that is part of an opcode or offsets for branch instructions.

Examples

If label A contains 10101001 then

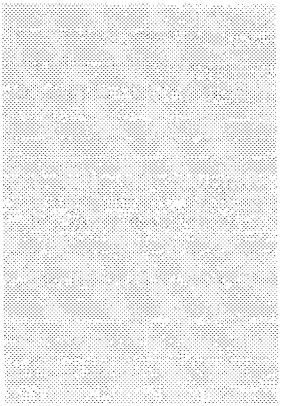
(A = 0b10[...][...]) /* $1 = 0b101 and $2 = 0b001 */

(A = 0b..[1..0]..) /* $1 = 0b1010 */

Note: The scope of local variables is one (possibly folded) line of a table. In order to transfer the value of a local variable to another line in the same table or to another table, it must be assigned to a global variable.

Note: Nesting of brackets, i.e. 0b.[.[..]].1 is not allowed.

Relational Conditions



Relational conditions are used to test the value of a global variable or label against another global variable or a constant value. Global variables are defined in the %%DEF section of the description file and can be used throughout the entire description file.

Syntax

<relational condition> ::=
<name> <operator> <value>
| <name> <operator> <name>
| <label_id> <operator> <value>
| <label_id> <operator> <name>

<operator> ::= '==' | '=' | '!='
| '>' | '<' | '>=' | '<='

'==' is the 'equal to' condition
'=' is the 'equal to' condition
'!=' is the 'not equal to' condition
'>' is the 'greater than' condition
'<' is the 'less than' condition
'>=' is the 'greater than or equal' condition
'<=' is the 'less than or equal' condition

<value> stands for integer values expressed in C notation or integer values declared in the %%DEF section of the declaration part of the .DSC file as either a variable or a constant.

Note: A label_id at the right hand side of a relational condition is not allowed.

Examples

(i < 3)

(mvr != 12)

(x >= y)

(ADDRESS <0x3f0)

Clock Sequence Conditions

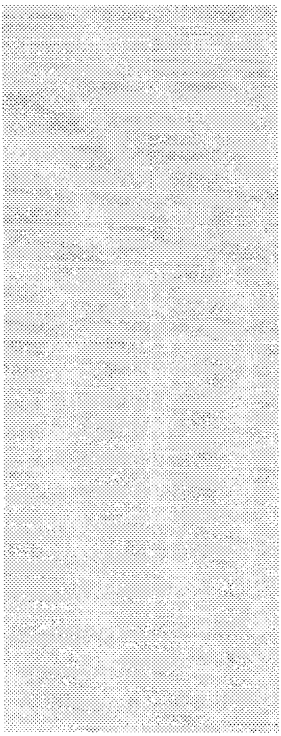
The clock sequence condition is true if the current sample is recognized as part of the specified clock sequence. Please refer to section "Clock Sequence Definition" for more details.

A Clock sequence condition is used to test if a clock sequence is valid for the current sample. This is useful in cases where two clock sequences are defined, for example one for a read cycle and one for a write cycle. In this case clock sequence conditions can be used to distinguish between a read or a write cycle.

Examples

(rdclkseq) /* read cycle */
(wrclkseq) /* write cycle */

AND-ing and OR-ing of Conditions



Multiple conditions can be used to qualify the same command chain for execution. These multiple conditions have to be AND-ed or OR-ed together for this purpose. To AND two or more conditions, they have to be enclosed by brackets ('(' and ')') and separated by commas (',').

In order to OR conditions together, the conditions must be positioned on subsequent lines, not followed by a command chain, except for the last conditional section of the OR-ed group. This effectively means that each conditional section has the same command chain associated with it.

Combinations of AND-ing and OR-ing is also possible.

Examples

Both conditions must be met:

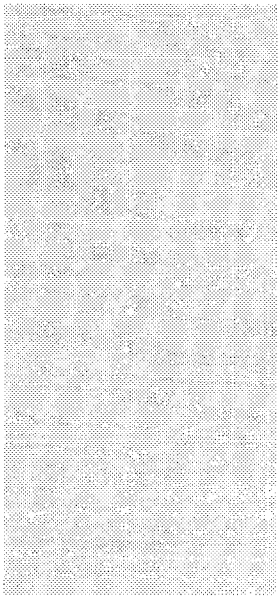
```
(status=0b0101, opcode=0x4f)
/*both status and opcode conditions must be satisfied. */
```

Either condition must be met:

```
(opc=0x45)
(opc=0x47)
(opc=0b01001...)     !"command chain for all conditions"!
    /* Either of these conditions has to be valid for the
    command chain to be executed. */
```

Pattern Expressions

Pattern expressions are optional expressions preceding the commands in index table lines. The pattern expressions are only used to assign values of one or more bitfields in a label or global variable to local variables. They look like pattern conditions in lookup tables with the exception that digits other than the don't care digit ('.') are not allowed for a bitpattern. See 'bit patterns' and 'local variables' in the Conditions section described before for the syntax description of pattern expressions.



Syntax

<pattern expression> ::=
'(' <pattern expression list> ')'

<pattern expression list> ::=
<pattern>
| <pattern> ',' <pattern expression list>

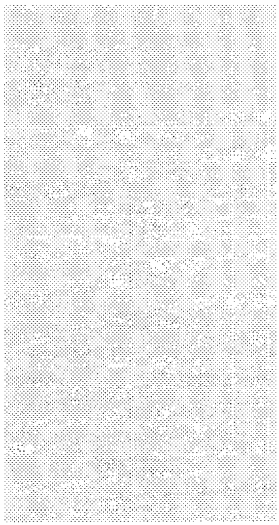
<pattern> ::=
<label or variable> '=' <bitpattern>
|<label or variable> '==' <bitpattern>

<label or variable> ::=
<label id>
| <variable name>

Examples

(DATA = 0x[....])
/* the value of the 4 digits is assigned to $1 */

Commands



Commands exist to perform all kinds of operations: output strings, interpret other tables, perform assignments to global variables, positioning in the measurements, display selection, etc.

Commands have to be enclosed in '!' signs. All commands together between two '!' signs form a command chain.

Tables of the tabular section can be used as procedures, i.e. when called from a command chain, control of the program is transferred to that table. Much like a subroutine call in a programming language when the end of a command chain is reached, control is passed back to the calling process. If the calling process was the %%START section and if the end of the command chain in the START-section is reached, then disassembly of one instruction is completed and disassembly of the next instruction is started.

The format of each command is explained in detail in the following paragraphs.

Syntax

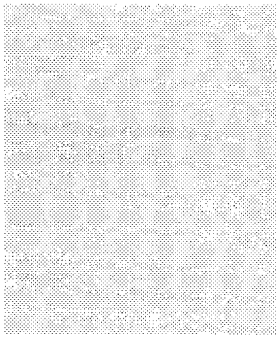
```
<command chain>  ::=
                      '!'  <commands> '!'

<commands>       ::=
                          <command>
                        |  <command> <commands>

<command>        ::=
                         <instruction_blk>
                        | <print_string>
                        | <acq_update>
                        | <table>
                        | <key_word>
                        | <display_sel>
```

Display Selection Commands

To tell the disassembler how to display a disassembler state or part of a disassembler state a number of commands are incorporated in the disassembler description language. These commands are:

- PROG
- UNUSED
- SKIP
- MR
- MW
- IOR
- IOW

These commands should be used to tell the disassembler in which manner the current disassembler state or the current part of the disassembler state has to be displayed in the logic analyzer display menu. The display selection commands give the user maximum flexibility in using the display part of the disassembler parameters popup menu of your PM 3580/PM 3585 logic analyzer.
For general information on instruction representation and

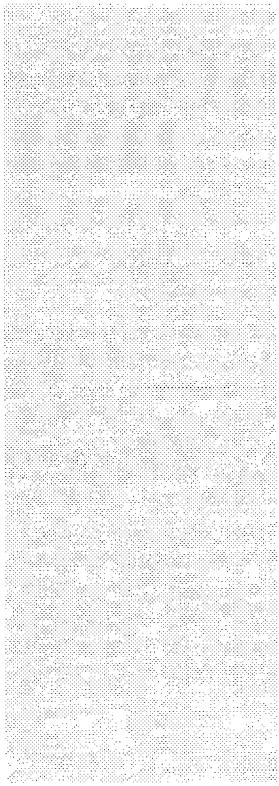


display selection see chapter 7 "Disassemblers" of your PM3580/PM3585 User Manual.

The first three commands (PROG, UNUSED and SKIP) are used the make a selection for instruction opcodes needed for 'Program Context Mode'. The last 4 commands (MR, MW, IOR, IOW) are used to treat disassembler states as data transfer according 'Show Data Transfers'.

Once a disassembler state is selected by one of the display selection commands it can not be altered, unless the UNGET command is used. If the display selection is altered the disassembler looses synchronization status.

PROG

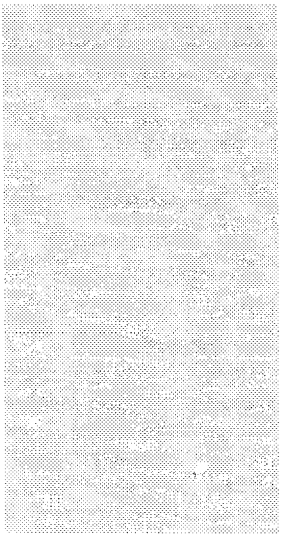


This command indicates that the current state or current part of a state has to be displayed with all disassembler display options. The generated text by the disassembler will be displayed with the first state selected with this 'PROG' command. The first state accessed in the START-section is default selected as PROG when reaching the end of the START table command chain. Other disassembler states are default selected as SKIP (described on the next page). It is allowed to overwrite this default. The next disassembler states selected with the 'PROG' display selection commands will produce an empty field in the disassembler column if program context mode is enabled. If program context mode is disabled a field with the text "opc" (opcode fetch) is displayed in the disassembler column.

Syntax

<key_word> ::= 'PROG' | 'prog'

UNUSED



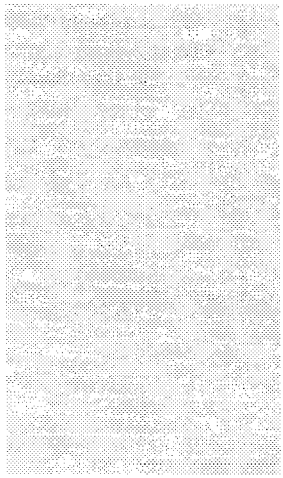
The command 'UNUSED' can be used to suppress the current state or part of a state from the display if program context mode is enabled.
The states or part of states for which the 'UNUSED' command is specified are only displayed when program context mode is disabled. The disassembler state or part of a state given the command 'UNUSED' is then displayed as "unused opc" in the disassembler column.

Syntax

<key_word> ::= 'UNUSED' | 'unused'

Note: Using a 'PROG' or 'UNUSED' command on a disassembler state results in treating all default states between the current and the first state as 'UNUSED' when reaching the end of the START table command chain. The first state or first state part is left unchanged (default PROG).

SKIP



If the current state or part of a state is to be used in the next 'START' the user is able to skip this state for usage in the current instruction. The command to do this is 'SKIP'. By using this command the current state or part of state will be made available for a next 'START'. After a state or part of a state for which the command 'SKIP' was used, only data transfer selection commands (MR, MW, IOR and IOW) are allowed. If no display selection command is specified for a disassembler state, 'SKIP' is the default for all disassembler states following the last UNUSED or PROG when reaching the end of the START table command chain. The default disassembler states preceding the last UNUSED or PROG are treated as if an UNUSED command was given.

Syntax

<key_word> ::= 'SKIP' | 'skip'

MR

By using the MR command, the current disassembler state is selected as a data transfer 'memory read' action according the disassembler parameters popup menu of your PM3580/PM3585 logic analyzer.

Concerning the data transfer display selection commands (MR, MW, IOR, IOW) some general remarks can be made. Where in the following part MR is used it can be replaced by other data transfer commands (MW, IOR or IOW) which will be described later.

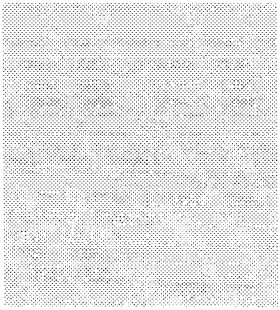
Two cases can be distinguished for data transfer commands.

1. The current state, for which a MR command is specified, **is** the first disassembler state after START. In this case the current state will be displayed as an unrelated data transfer: 'mr (unrel)'.
2. The current state, for which a MR command is specified, is **not** the first disassembler state after START. The first disassembler state is either default or a specified 'PROG' or 'UNUSED' command. In this case the disassembler state is displayed as 'mr' on your PM3580/PM3585 logic analyzer display menu.

   If the disassembler state is preceded by one or more disassembler states for which the 'PROG' display selection command is used the data transfer will be displayed immediately after the last 'PROG' disassembler state of the current 'START' sequence if program context mode is enabled.
   This gives the user the possibility to compensate microprocessor pipelines in which results of an instruction appear on the microprocessor bus after other instruction fetches.

When 'parts' (see 'Label parts definition') is used the disassembler state for which a data transfer display selection command is specified has only one part anymore. On accessing the disassembler state for getting values of the part-label only the value of the first part can be accessed.

Before using the MR command for a disassembler state



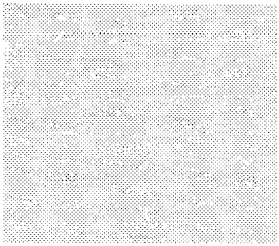
the user should save all requested values from the other parts in the disassembler state. The other parts can not be accessed after the MR command is given.

Data transfer commands are not allowed if another display selection command is already specified for any part of the disassembler state. The disassembler will then loose synchronization status.

Syntax

<key_word> ::= 'MR' | 'mr'

MW

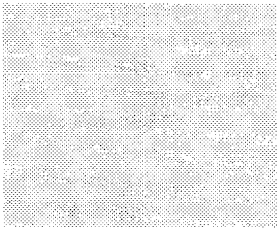


The current disassembler state is selected as a data transfer 'memory write' which is displayed as 'mw' on your PM3580/PM3585 logic analyzer display menu.
See the 'MR' display selection command for general remarks on data transfer selection commands.

Syntax

<key_word> ::= 'MW' | 'mw'

IOR

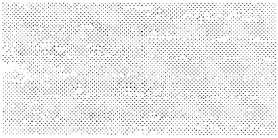


The current disassembler state is selected as a data transfer 'I/O read' which is displayed as 'ior' on your PM3580/PM3585 logic analyzer display menu.
See the 'MR' display selection command for general remarks on data transfer selection commands.

Syntax

<key_word> ::= 'IOR' | 'ior'

IOW



The current disassembler state is selected as a data transfer 'I/O write' which is displayed as 'iow' on your PM3580/PM3585 logic analyzer display menu.
See the 'MR' display selection command for general remarks on data transfer selection commands.

Syntax

<key_word> ::= 'IOW' | 'iow'

Positioning in a Measurement

For positioning in the measurement a number of commands are incorporated in the Disassembler Description Language:

- GOTO [i]
- TELL
- NEXT
- PREV
- GOTOPART [i]
- TELLPART
- NEXTPART
- PREVPART
- UNGET

The disassembler steps through a measurement by disassembler states. All positioning and access in the measurement is therefore done in complete disassembler states. Please refer to section "Clock Sequence Definition" for more details on disassembler states.
Some microprocessors can have more than one instruction in one disassembler state. Examples are 68020 or 80x86 microprocessors. Positioning only on disassembler states is not enough in such cases. For this reason a positioning within a disassembler state can be achieved by use of the commands 'GOTOPART', 'TELLPART', 'NEXTPART' and 'PREVPART'.

GOTO [i]

The GOTO command is used to instruct the disassembler to proceed to relative position i of the measurement. The first disassembler state used for disassembly of an instruction is the current state when the START-table is entered. This state is always state 1. The parameter i denotes the position relative to this first state. States for which a display

selection command other than SKIP was specified in the previous START are not considered within the current START command chain. As such positioning on these already selected states is impossible.
When the end of the command chain in the 'START' table is reached, the disassembler proceeds to:

- The first disassembler state or state part for which the 'SKIP' command was specified.
- The first default disassembler state or state part after the last 'UNUSED' or 'PROG' disassembler state.
- The second disassembler state or the next state part of the current first disassembler state, if the disassembler lost synchronization status.

This disassembler state becomes state 1 when entering the START table for decoding the next instruction.

If the GOTO[i] command results in a positioning to a disassembler state outside the measurement the disassembler looses synchronization status.

See also the TELL, NEXT and PREV command.

Syntax

<acq_update> ::= 'GOTO[' <name or constant> ']'

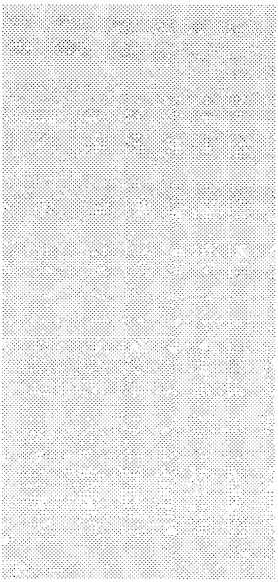
Examples

! GOTO[4] !

! GOTO[oldstate] !

Note: The maximum value of <name or constant> is 32. As such, the disassembler can look ahead 32 disassembler states. Negative values are not allowed so there is no look-back capability. To report a status of an instruction to a following instruction one can use the 'static char' global variable (See section 'Global Variables').

TELL



A command which is closely related to positioning in measurement with the GOTO command is the 'TELL'-command. This command returns the value of the relative position of the current disassembler state. The command can be used in relational conditions and in instructions in the command chain.

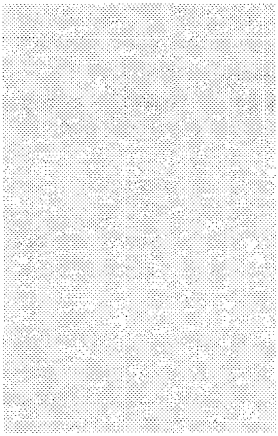
Syntax

<key_word> ::= 'TELL' | 'tell'

Examples

```
(TELL < 5)     /* is relative position < 5 */

! { oldstate = TELL } ! /*  the global variable oldstate gets
                            the value of the current relative
                            position. This variable can later
                            be used for repositioning with
                            the 'GOTO' command. */
```

NEXT



The next state from the measurement becomes current. If the current state is the last state of the measurement the command NEXT results in loosing synchronization status. States for which a display selection command other than SKIP was specified in the previous START are not considered within the current START command chain. As such positioning on these already selected states is impossible.

Syntax

<key_word> ::= 'NEXT' | 'next'

Examples

```
! GOTO[3] NEXT !/* After 'next' state 4 is current */
```

PREV

The previous state from the measurement becomes current. If the current state is the first state (GOTO[1]) the command PREV results in loosing synchronization status. States for which a display selection command other than SKIP was specified in the previous START are not considered within the current START command chain. As such positioning on these already selected states is impossible.

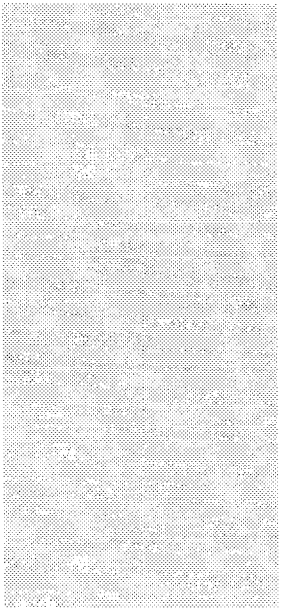
Syntax

<key_word> ::= 'PREV' | 'prev'

Examples

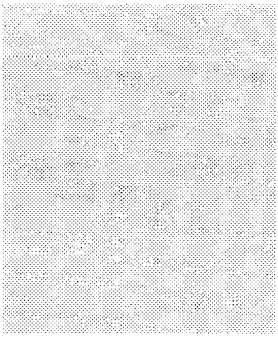
! GOTO[3] PREV !/* After 'prev' state 2 is current */

GOTOPART[i]



If the disassembler can have more than one instruction in one disassembler state the user has the ability to define one label which can be accessed in parts. For example a 68030 32-bit databus can contain 2 NOP instructions in one disassembler state (the NOP instruction is 16-bits). Because the databus is 32-bits wide and the minimum instruction size is 16-bits it is possible to have 2 instructions per disassembler state. To be able to properly handle the instructions the databus label can be accessed sequentially in parts of 16-bits. See section 'label-type' and 'label-parts' definition for defining such a label. If the databus label is accessible in 2 parts of 16 bits, 'GOTOPART' can be used to position on the proper part of the databus label. The label which can be accessed in parts may have up to 4 parts. GOTOPART may be called with part 0 to 3. GOTOPART to a part which is not valid will result in loosing instruction synchronization status.

The part number given as parameter in the GOTOPART command is absolute within the current disassembler state. At entering the START-table the current part number probably is not zero. This in contradiction to the GOTO[i]



command where the disassembler state at entering the START-table is always 1.

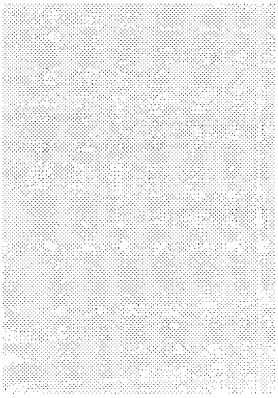
Syntax

<acq_update> ::= 'GOTOPART[' <name or constant> ']'

Examples

! GOTOPART[0] !
! GOTOPART[oldpart] !

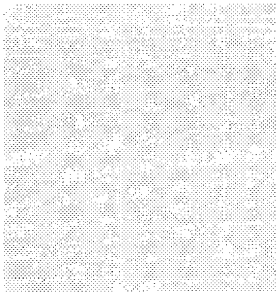
TELLPART



A command which is closely related to positioning in a disassembler state with the GOTOPART command is the TELLPART-command. This command returns the value of the current part in the current disassembler state. The returned value can be 0 to 3. The command can be used in relational conditions and in instructions in the command chain.

```
(TELLPART == 0)
      /*is current part in the current disassembler state 0 */
 ! { oldpart == TELLPART } !
      /* the global variable oldpart gets the value of the current relative part. This variable can later be used for repositioning with the 'GOTOPART' command. */
```

NEXTPART



The next part of the current disassembler state becomes current. If the current part is the last part of the current disassembler state or there is only one part per disassembler state the first part of the next disassembler state becomes current. In this case 'NEXTPART' behaves like 'NEXT'

Syntax

<key_word> ::= 'NEXTPART' | 'nextpart'

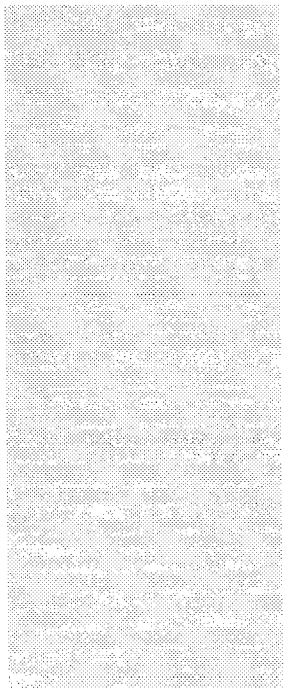
Examples

Suppose a label with 4 parts is defined in the FORMAT section.

```
! NEXT NEXTPART NEXTPART !
    /* after the next command the current part in the current disassembler state is 0. After 2 times 'NEXTPART' the part number 2 is the current part of the current disassembler state */
```

Note: If a label is defined which has more than one part the 'NEXT' command will proceed to the first part of the next disassembler state, regardless what the current part was before the 'NEXT' command.

PREVPART



The previous part of the current disassembler state becomes current. If the current part is already the first part of the current disassembler state, the last part of the previous disassembler state becomes current. In this case 'PREVPART' behaves line 'PREV'.

Syntax

<key_word> ::= 'PREVPART' | 'prevpart'

Examples

Suppose a label with 4 parts is defined in the FORMAT section.

```
! PREV PREVPART !
    /* after the prev command the current part in the current disassembler state is the last part (in this case 3). After 'PREVPART' the part number 2 is the current part of the current disassembler state */
```

Note: If a label is defined which has more than one part the 'PREV' command will proceed to the last part of the previous disassembler state, regardless what the current part was before the 'PREV' command.

UNGET

The Custom Disassembler keeps track of the disassembler states which were accessed since START and which of them were selected for displaying. (See section 'Display Selection'). The status of the last accessed entire disassembler state can be reset to the default status by using the 'UNGET'-command. From then on it is treated as not accessed. It could be regarded as a push back operation on the disassembler input.

Syntax

<key_word> ::= 'UNGET' | 'unget'

Example

```
(dt=0x..)        ! "opcode " NEXT UNUSED LX !
                 /* start line */

%% LX
(dt=0x00)        ! unget !
                 /* Pattern 00 is always the start of an
                    instruction */

(dt=0x[..])      ! "continuation" ... !
                 /* All other patterns must be the
                    continuation of an instruction */
```

Note: The highest disassembler state number is the highest number the disassembler reached while using the positioning commands. So in the command chain ! NEXT NEXT GOTO[7] PREV PREV ! the highest number is 7 even though two prev commands follow the command GOTO[7]. To decrement the highest number the unget command must be used. For each unget command the highest number is decreased by 1.

Note: If the highest number equals 1 the unget command has no effect.

Instructions

Instructions are operations upon local or global variables and are comparable to C program blocks. They should be enclosed in '{' and '}'. Multiple instructions can occur within one block, provided they are separated by a semi-colon (';').

Syntax

```
<instruction_blk>  ::=
                '{'<instructions> '}'

<instructions>     ::=
                <expression>
                | <expression> ';' <instructions>

<expression>       ::=
                 <name> '=' <name>
                | <name> '=' <name> <operator> <name>
                | <name> '=' <name> <operator> <constant>
                | <name> '=' <name> <operator> <value>
                | <name> '=' <instr cond>
                | <name> '=' <name> <operator> <instr cmd>

<instr cmd>        ::=

                    'TELL' | 'tell' | 'TELLPART' | 'tellpart'
<operator>         ::=
                    '+' | '-'  |  '*'  |  '/'  |  '%'
                       | '&' |    '|'  | '>>' |  '<<'
```

'+'	add
'-'	subtract
'*'	multiply
'/'	divide
'%'	remainder after division (mod).
'&'	bitwise AND
'|'	bitwise OR
'>>'	'shift right'
'<<'	'shift left'

Examples

{temp = max; $1 = temp - $1}

{offset = offset + 12; dest = $2 + offset}

{state =TELL}

Note: An instruction like for example:
{dest = $2 + offset + 12} is not allowed but should be done as in the second example above.

Print Commands

Print commands are used to output character strings. Typical examples are instruction mnemonics or names for registers. In general, anything that you want to appear on an output line is processed through print statements. Formatting of print statements conforms to the C language syntax and includes the use of the tabulation notation ('\t') and the new line notation ('\n'). Also refer to section Tab Settings of this chapter. Always enclose the string to be printed between quotes ("").

Data values can be printed by defining local variables. For formatting purposes, the C language style for hexadecimal, binary, octal, decimal and character output are available. An additional format modifier for symbolic output of a local variable is available. Values of the local variables $1, $2, $3 etc. are used for printing according the respectively format modifiers found within a command chain.

For the symbolic printout format the value of the local variable is matched against the defined symbols for the specified label in the logic analyzer. If a matching symbol can be found, the corresponding symbol text is printed instead of the value of the local variable. If no matching symbol can be found the value will be printed in the radix of the specified label. (Symbolic printout is only effective with logic analyzer system software version 2.01 or up).

If the output line exceeds the disassembler column width, defined in the header definition (see section 'Header Definition'), the disassembler will automatic fold the rest of the resulting disassembler text to a new disassembler line.

This folding will be done on the last spacing character in the output line before the column width is exceeded.

The total output line may not exceed 255 characters. If this maximum is exceeded remaining characters are silently truncated. Where local variables are used in a command chain they are inserted in the sequence $1, $2, ..., $9.

Syntax

```
<print string>          ::=
                '"'  <print commands> '"'
<print commands>   ::=
                <ASCII-character>
                | <format_command>
                | <ASCII-character>  <print_commands>
                | <format-command> <print_command>

<ASCII-character>   ::=
                printable character
                | \t        (tab)
                | \n (new disa output line)

<format_command>  ::=
                <format specifier>


 <format specifier>   ::=
                '%'    <width and type>

< width and type>     ::=
                'c'
                | <width> <type>
                | <symbol format>
<width>                 ::=
                <decdigits>          (field width equals n positions with leading blanks)
                | '0'  <decdigits>   (field width equals n positions with leading zeroes)
                | <empty>

<type>                  ::=
                'b' | 'o' | 'd' | 'x'     (binary, octal, decimal, hexadecimal)
```

<symbol format> ::=
'<' <label> '>' <symbol viewsize> 's'

<symbol viewsize> ::=
'.' <width>
| <empty>

Examples

If label A is 0b00100110 then

(A=0x..)	!"This text will be printed."! /* This text will be printed */
(A=0x[.].)	!"address = %d"! /* address = 2 */
(A=0x[..])	!"opcode=\t%04x\tin hex."! /* opcode=0026 in hex. */
(A=0x[.][.])	!"opc1=%02x opc2=%02x"! /* opc1=02 opc2=06 */
(A=0x..)	! "Percent sign: %%" ! /* Percent sign: % */
(variable = 0x[....])	! "%<ADDRESS>s" !
(ADDRESS = 0x[....])	! "%<ADDRESS>.10s" ! Will print the symbolic value of ADDRESS in exactly 10 characters.

Note: <symbol format> is only effective if logic analyzer system software version 2.01 or up is used

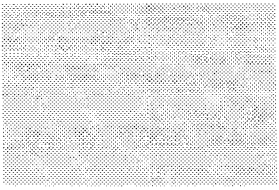
Note: Zero width can be specified to skip local variables. "%0x%02x" will print the value of $2.

Special Commands

Two special commands related to printing are available:

- UNPUT
- ERROR

UNPUT



The last character that was printed with a print-string command, is erased. The effect is similar to a backspace in a text editor.

Syntax

<key_word> ::= 'UNPUT' | 'unput'

ERROR

The error command is intended for debugging purposes while developing a disassembler.
The string "? ERROR: [i]" will be displayed to indicate the occurrence of an unknown condition or data value. The value "i" is the relative disassembler state according the GOTO and TELL command, within the current START-table command chain.
The error command could be placed at the end of a lookup table to signal a no-match situation.

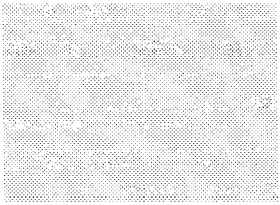
Syntax

<key_word> ::= 'ERROR' | 'error'

Examples

```
%%MAIN_LOOKUP
....
....
(opcode=0x..)     ! error !    /* No match found */
```

Transfer Control to other Tables



Calls to other lookup or index tables can be made from within a command chain. When a table call is encountered, control is transferred to this new table. If the table is an index table, an offset must be supplied between square brackets ('[' and ']'). This offset can be a constant, a variable or a local variable. No offset can be specified for

lookup tables as these are always scanned from top to bottom.

Calls to tables can be nested and any table can be called recursively. The names used to call other tables must have been declared in the %%EQU section of the disassembler description source file.

Note: The first entry position (index) in an index table is always zero (0).

Note: The total nesting of tables is limited to a depth of 128. The generated custom disassembler will automatically display the following error message in the disassembler output column if this number is exceeded: "? maximum table nesting exceeded: [i]", where "i" is the relative position in the measurement. When this error occurs you probably defined an infinite loop in accessing tables.

Note: If a value of a sample is not found in a lookup or index table it is accessed outside its boundaries. The disassembler will automatically print "***" in the disassembler output column indicating that the disassembler lost synchronization status. (Compare chapter 7, "Disassemblers", of your PM 3580/ PM 3585 User Manual). The disassembler will then automatically proceed with the disassembler state following the first accessed disassembler state.

Syntax

```
<table>          ::= <name-lookup>
                   | <name-index>   '['  <offset> ']'
<name-lookup>    ::=   <name> (lookup table name)

<name-index>     ::= <name> (index table name)
<offset>         ::= <name>
                   | $<lvdigit> (local variable)
                   | <value>
                   | <constant>

<lvdigit>        ::= 1 ..9
```

Examples

```
!"ld\t" REG[$1] GOTO[2] MAIN!
                    /* Print mnemonic then register type,then
                       process next line */
! IDx[6] !          /* Jump to index table entry */
! Ind[$3] !         /* Take local variable 3 as the offset in index
                       table Ind */
! it[dw] !          /* Take global variable dw as offset in index
                       table it*/
```

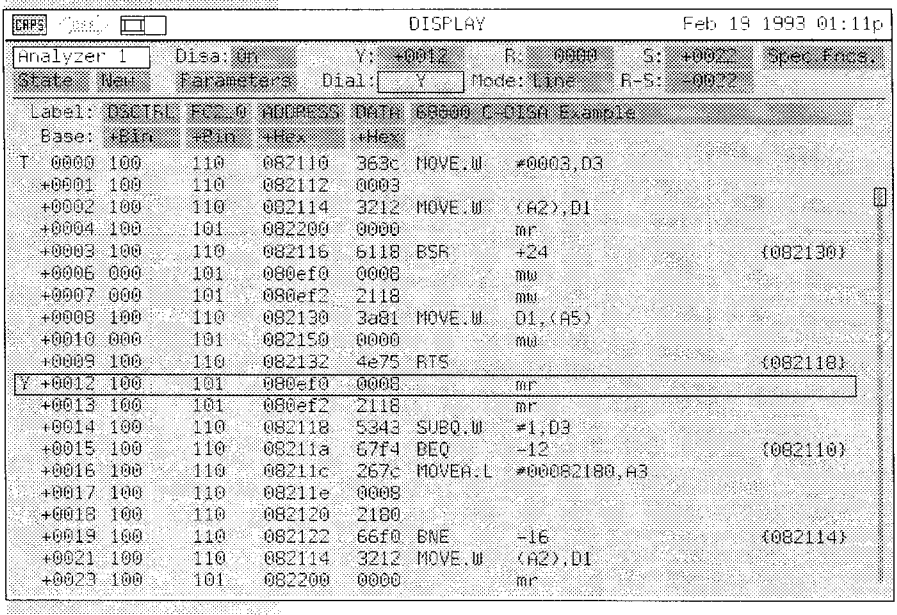
Chapter 4

Writing a 68000 Disassembler

Disassembler labels 4-3
Disassembler Status Selection 4-4
Instruction Decoding 4-4
Finding additional opcodes 4-5
Display additional opcodes 4-7
Computing branch offsets 4-7
Sub values in the instruction 4-9
Computing return address 4-9
Searching datatransfers 4-9
Suppressing unused opcodes from display 4-10
Processing data transfers 4-13
Branch instruction 4-13
Conditional branches 4-14
Static char variable usage 4-15
Finishing the example disassembler 4-17

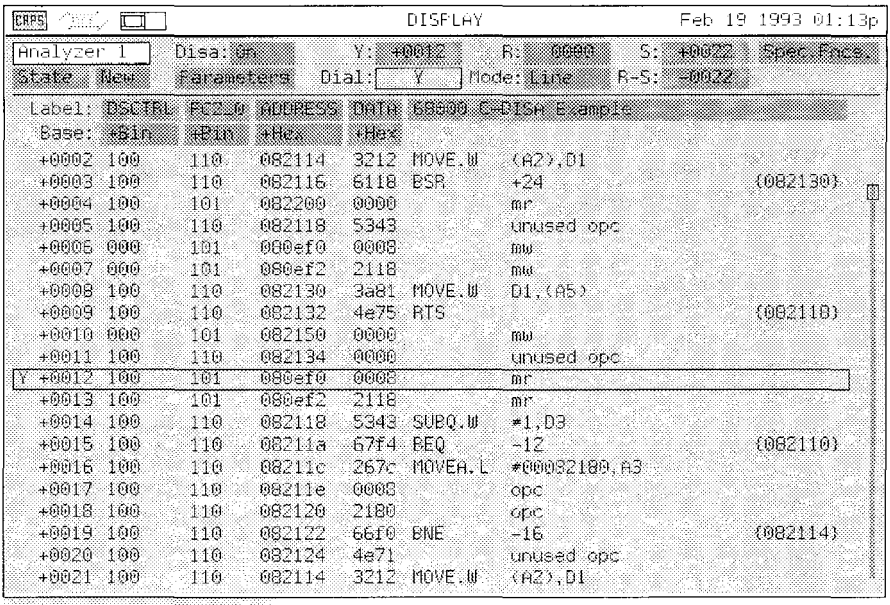
Suppose a 68000 disassembler has to be written to disassemble the DEM68000.NEW measurement delivered with your custom disassembler software package.

The disassembler should produce the next output in the display menu of your PM 3580/PM 3585 logic analyzer if program context mode is enabled.



The measurement file is available on disk with the filename DEM68000.NEW.
The disassembler description file is available on disk with the filename DEM68000.DSC. Use the CDISA80 compiler to create a loadable disassembler DEM68000.DIS.

If program context mode is disabled the following output should be produced. The differences can be seen in the order of disassembler states shown and at the unused opcode fetches on disassembler state +0005, +0011 and +0020.



The instructions which are shown in the measurement are:

- MOVE - general move instruction
- BSR - branch to subroutine
- RTS - return from subroutine
- SUBQ - subtract
- BEQ - branch on equal
- MOVEA - move to an address register
- BNE - branch on not equal

Disassembler labels

Before one can write such a disassembler we should know some information specific for the 68000.
The 68000 has labels 'DSCTRL', 'FC2_0', 'ADDRESS' and 'DATA' and the clocks 'UDSN' and 'LDSN'. These two clock labels contain no relevant information for the user. They are only relevant to the disassembly process because they identify applicable disassembly states. The 3 channel label 'DSCTRL' (data size control) contains the

following information
(the most significant channel is the R/WN channel):

0 0 0 : memory write state 16-bit databus.
1 0 0 : memory read state 16-bit databus.
others : not supported for this example.

The 3 channel label 'FC2_0' (function code) contains the following information:

1 0 1 : memory action in data area.
1 1 0 : memory action in program area.
others : not supported for this example.

The 'DATA'-label contains the opcodes of the instructions to be disassembled as well as the data transfers caused by the instruction. Each instruction has an opcode of 16-bits or a multiple of 16-bits. An instruction starts with a 16-bit databus memory read action in program area.

Disassembler Status Selection

The FC2_0 and DSCTRL labels contain information about the bus status. For an instruction opcode fetch state these labels should have the following values:
FC2_0 = 1 1 0 and DSCTRL = 1 0 0.
With this knowledge instruction states can now be distinguished in the disassembler from data transfer states by setting up the following table:

```
%% STATUS_TAB
(FC2_0 = 0b110, DSCTRL=0b100)       ! "instruction" !
(FC2_0 = 0b101, DSCTRL=0b1..)       ! mr ! /* memory read */
(FC2_0 = 0b101, DSCTRL=0b0..)       ! mw ! /* memory write */
(FC2_0 = 0b...)                     ! "reserved" ! /* not supported *
```

Instruction Decoding

A distinction is made between instruction and memory actions of the microprocessor by using the available status labels 'FC2_0' and 'DSCTRL'.
Below you see the binary format of the instructions in the measurement we want to disassemble:

```
MOVE.W      #<immediate>,D3   : 0011 0110 0011 1100 <16-bit immediate data>
MOVE.W      (A2),D1           : 0011 0010 0001 0010
BSR         <addr-offset>     : 0110 0001 <8-bit signed addr-offset>
MOVE.W      D1,(A5)           : 0011 1010 1000 0001
RTS                           : 0100 1110 0111 0101
SUBQ.W      #<value>,D3       : 0101 <3-bit value>1 0100 0011
BEQ.B       <addr-offset>     : 0110 0111 <8-bit signed addr-offset>
MOVEA.L     #<immediate>,A3   : 0010 0110 0111 1100 <32-bit immediate data>
BNE.B       <addr-offset>     : 0110 0110 <8-bit signed addr-offset>
```

Instead of printing "instruction" a sub-table for decoding the instructions contained in the 'DATA' label is now called. Some command lines in the 'decode' table contain the names of other, not yet defined, tables.

```
%% decode                    /* _ in a condition is only used for readability */
(DATA = 0b0011_0110_0011_1100)     ! "MOVE.W\t#" immediate16 ",D3" !
(DATA = 0b0011_0010_0001_0010)     ! "MOVE.W\t(A2),D1" !
(DATA = 0b0110_0001_...._....)     ! "BSR\t" addr_offset !
(DATA = 0b0011_1010_1000_0001)     ! "MOVE.W\tD1,(A5)" !
(DATA = 0b0100_1110_0111_0101)     ! "RTS\t" compute_addr !
(DATA = 0b0110_0111_...._....)     ! "BEQ\t" addr_offset !
(DATA = 0b0101_...1_0100_0011)     ! "SUBQ.W\t#" val3bit ",D3" !
(DATA = 0b0110_0110_...._....)     ! "BNE\t" addr_offset !
(DATA = 0b0010_0110_0111_1100)     ! "MOVEA.L\t#" immediate32 ",A3"!
```

Finding additional opcodes

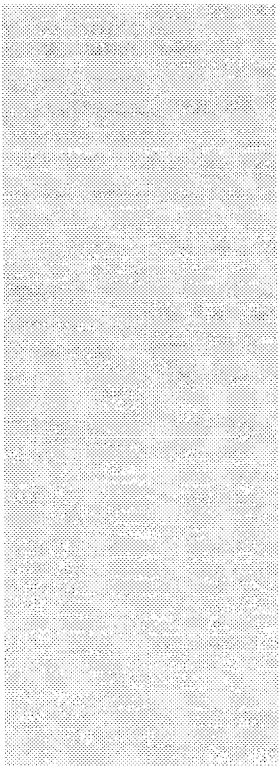
For the first MOVE immediate instruction the following disassembler state with a function-code in program area and a memory read of 16-bits contains the immediate data for this instruction. Now suppose that this state is always the next disassembler state in the measurement. The call to immediate16 could then be replaced by:

```
! "MOVE.W\t#" NEXT print[0] ",D3" !
```

in which print[0] is a call to the 'print' table with offset 0.

```
%% print
(DATA=0x[....])! "%04x" !
```

The 68000 is a pipelined micro-processor, which means the 68000 is already fetching another instruction opcode before the current instruction is completed. So the next disassembler state is probably not the disassembler state which contains the immediate data.

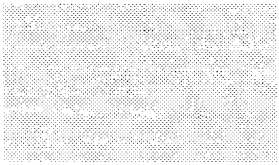


The next disassembler state could also be a data-transfer disassembler state which is a result of a previously executed instruction. Due to the microprocessor internal opcode pipeline it can appear now on the bus. To handle this case a search to the next instruction opcode should be done. The next instruction opcode state should meet the following conditions:

- The FC2_0 label function-code must be in program area
- The DSCTRL-label must be a memory read over 16-bits.
- The ADDRESS of the opcode-state must be 2 higher than the address of the first MOVE-opcode disassembler state.
- It must be the first disassembler state with function-code in program area and a16-bit memory-read action.

The call of command 'NEXT' must be replaced by a table which searches the next opcode state: 'FindNextOpc'.
This can be an index table. The call should then be 'FindNextOpc[0]'.
The next instruction opcode state should be present in the measurement. If it is not something is wrong and the disassembler should loose synchronization status.
This results in the following tables for finding an additional opcode for an instruction.

```
%% FindNextOpc                          /* index table, compute next opcode
                                          address, and search for it */
(ADDRESS=0x[......])                    ! { nopc_addr = $1 + 2 } NEXT ChkNextOpc !

%% ChkNextOpc                           /* lookup table */
(ADDRESS = nopc_addr, FC2_0 = 0b110,DSCTRL = 0b100)!/* found next opcode */!
(FC2_0 != 0b110)                        ! NEXT ChkNextOpc !
                                        /* next opcode not found yet, look forward
                                        Function-code FC2_0 = 0b110 is illegal.
                                        The disassembler looses synchronisation
                                        status */
```



Display additional opcodes

Assume the next opcode of the instruction is found. To display this opcode with the rest of the instruction a display selection command for this state has to be specified. Because it is an instruction opcode there are 3 possibilities:

- PROG
- UNUSED
- SKIP

PROG means display the current state in all possible display modes on your logic analyzer display menu.
UNUSED means display state only if program context mode is disabled.
SKIP means do not use this state now, use it in another START for decoding in a following instruction.

In this case the opcode belongs to the instruction, so it should be displayed: PROG is then the one to use. Which makes the handling of the MOVE instruction complete:

```
(DATA = 0b0011_0110_0011_1100)      ! "MOVE.W\t#" FindNextOpc[0] \
                                      PROG print[0] ",D3" !
```

The other MOVE instructions are already completely decoded.

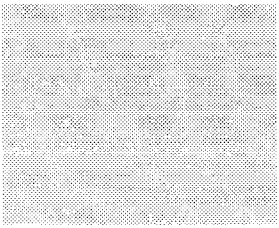
```
(DATA = 0b0011_0010_0001_0010)      ! "MOVE.W\t(A2),D1" !
(DATA = 0b0011_1010_1000_0001)      ! "MOVE.W\tD1,(A5)" !
```

The MOVEA instruction can be handled in a similar way as the MOVE immediate explained before. The only difference is that it requires two additional opcodes in which the immediate address is contained. This results in the following lines for this instruction:

```
(DATA = 0b0010_0110_0111_1100)      ! "MOVEA.L\t#" FindNextOpc[0] PROG \
                                    print[0] FindNextOpc[0] PROG \
                                    print[0] ",A3"!
```

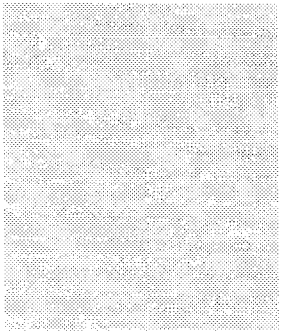
Computing branch offsets

The BSR instruction contains in the lower 8-bits a relative offset from the current address to the address at which the microprocessor will start executing the subroutine.
The relative offset can be used to compute the address at which the microprocessor will continue execution of instructions. The sign bit in the offset should be used to compute the correct address.



The computed address value should be printed which is done through a new entry in the 'print' index table. This new entry extracts the value of the computed address in a local variable and prints it in exactly 6 hexadecimal digits including leading zeros.
The resulting line in the 'decode' table and the new tables or table lines look like:

```
(DATA = 0b0110_0001_...._....)        ! "BSR\t" Compute8bitBrAddr print[1] !

%% Compute8bitBrAddr                  /* data label contains 8-bit signed off-
                                      set to ADDRESS */
(DATA=0b<.>8_0[<.>7], ADDRESS=0x[......])   ! /* positive offset */ \
                                      {dest_address = $2 + $1;\
                                      /* compute destination address */ \
                                      dest_address = dest_address + 2 }\
                                      /* adjust value */ \
                                      "+%d"    /* print offset */ !
(DATA=0b<.>8_1[<.>7], ADDRESS=0x[......])   ! /* negative offset */ \
                                      {$1 = $1 - 0x80;  /* sign extend $1 */ \
                                      dest_address = $2 + $1;\
                                      /* compute destination address */ \
                                      dest_address = dest_address + 2 }\
                                      /* adjust value */ \
                                      "%d"   /* print offset */ !

%% print
(DATA=0x[....])                    ! "%04x" !
(dest_address = 0x..[......])      ! "\t{%06x}" !
```



Note: In the first condition of the Compute8bitBrAddr table the notation "<.>8_" is a short hand notation for 8 don't care digits. So instead of 8 dots representing 8 don't care digits the user can also use the short hand syntax

"<" digit ">" decimal_number "_"

in which digit can be a dot (don't care) or any digit in the range of the given radix.

Two other instructions (BEQ and BNE) look almost the same as the BSR instruction. The entry for these instructions in the decode table looks like:

```
(DATA = 0b0110_0111_...._....)! "BEQ\t" Compute8bitBrAddr print[1]! !
(DATA = 0b0110_0110_...._....)! "BNE\t" Compute8bitBrAddr print[1]! !
```

Sub values in the instruction

The next instruction to disassemble is the SUBQ instruction. The SUBQ instruction is 16 bits long. The immediate substract operand is encoded in the instruction opcode as a 3-bit value. This value can be extracted directly from the DATA label and formatted in the final disassembly text by using a local variable. The decode line for the SUBQ instruction will then be:

```
(DATA = 0b0101_[...]1_0100_0011)  ! "SUBQ.W\t#%d,D3" !
```

Computing return address

There is only one instruction left which is not completely decoded yet: 'RTS'. For this instruction the return address is incorporated in 2 disassembler states with function code in data-area and read-actions of 16-bit. The return address should be printed the same way as is done for the BNE, BEQ and BSR instruction.
Because of the pipeline of the microprocessor another instruction opcode prefetch is done before the 2 disassembler states in data-area appear which contain the return address. So a search is started to find the data read. After this search the return address from the RTS instruction can be computed and printed the same way as is done for the BSR instruction. The decoding of the RTS instruction then looks as:

```
(DATA = 0b0100_1110_0111_0101)     ! "RTS\t"  FindDataRead \
                                   ComputeReturnAddr[0]  print[1] !

%% FindDataRead
(FC2_0 = 0b101, DSCTRL = 0b100)    ! /* found, stop searching */ !
(FC2_0 = 0b...)                    ! /* not found yet */ NEXT FindDataRead !

%% ComputeReturnAddr
(DATA=0x[....])          ! { dest_address = $1 << 16 } MR \
                         FindDataRead ComputeReturnAddr[1] !
(DATA=0x[....])          ! { dest_address = dest_address + $1 } MR !
```

Searching data transfers

After decoding the instruction the data transfer is searched by a call to the table 'FindDataRead. If the data read is found the address which will be next on the bus is computed in the table 'ComputeReturnAddr'. This return address is then printed.
A command already used in the example in chapter 2 is specified in the ComputeReturnAddr table: 'MR'. This

command is one of the 4 possible display selection commands to select a disassembler state as a data transfer. The possible data-transfer display commands are:

- MR : for memory read
- MW : for memory write
- IOR : for I/O read
- IOW : for I/O write

Data transfer states are displayed together with the instruction if program context mode is enabled.
If program context mode is disabled the disassembler states are displayed in chronological order. The data transfer states are suppressed from the display if show data transfer mode is disabled. These modes can be selected in the disassembler parameters popup menu on your PM 3580/PM 3585 logic analyzer.
Seven display selection commands are available to determine the way the display of states is affected by the LA's display modes.

These commands are:

For instruction opcode selection: PROG, UNUSED, SKIP.

For data transfer selection: MR, MW, IOR, IOW.

Using these commands the user has maximum flexibility to allow or suppress disassembler states from the PM 3580/ PM 3585 logic analyzer display menu by means of the disassembler parameters popup (See Chapter 7, "Disassemblers" of your PM 3580/PM 3585 User Manual).

Suppressing unused opcodes from display

The RTS instruction is decoded and the return address is computed and printed. The disassembler state with the computed return address in the ADDRESS-label contains the opcode which will be executed next after the RTS instruction. The opcodes fetched on other disassembler states after the RTS instruction and before the disassembler state with the computed return address in the ADDRESS-label are not executed opcodes.
If program context mode is enabled these not executed opcodes should not be displayed. To achieve this, the previous tables must be extended.

After the return address is computed the search for the disassembler state with the return address in the ADDRESS-label is started from the last opcode disassembler state of the RTS instruction. The computed address most likely is an address which is not in sequence with the value on the ADDRESS-label for the current RTS opcode disassembler state. A search is started to find an opcode fetch with a value on the ADDRESS-label which is not in sequence with the address value of the last opcode fetch.
If this so called sequence break can be found the address value is compared with the computed return address of the RTS instruction. If these are the same the opcode fetches found between the last opcode disassembler state of the RTS instruction and the current disassembler state are 'unused opcodes'.
The unused opcodes can be suppressed from the default logic analyzer display menu by the 'UNUSED' command.
The disassembler states will then only be shown if the program context mode is disabled in the disassembler parameters popup on your PM 3580/PM 3585 logic analyzer. With this knowledge the full decoding of the RTS instruction requires the following tables.

```
(DATA = 0b0100_1110_0111_0101)    ! "RTS\t" \
                                  { last_opc = TELL } \
                                  FindDataRead \
                                  ComputeReturnAddr[0] \
                                  print[1] \
                                  GOTO[last_opc] \
                                  FindNewOpcAddress[0] CheckBrAddress !

%% FindDataRead
(FC2_0 = 0b101, DSCTRL = 0b100)   ! /* found, stop searching */ !
(FC2_0 = 0b...)                   ! /* not found yet */ NEXT FindDataRead !

%% ComputeReturnAddr
(DATA = 0x[....]                  ! { dest_address = $1 << 16 } MR \
                                  FindDataRead ComputeReturnAddr[1] !
(DATA = 0x[....]                  ! { dest_address = dest_address + $1 } MR !

%% FindNewOpcAddress
! FindNextOpc[0] { state = TELL } FindAddrSeqBrk !

%% FindAddrSeqBrk
(pipeline > 2)                    ! /* not found, reposition */ GOTO[state] !
(ADDRESS = 0b[<.>24])             ! {nopc_addr = $1; \
                                  pipeline = pipeline + 1} \
                                  NEXT ChkAddrSeq !

%% ChkAddrSeq
(FC2_0 = 0b110, ADDRESS = nopc_addr, DSCTRL = 0b100) ! FindAddrSeqBrk !
(FC2_0 != 0b110)                  ! NEXT ChkAddrSeq !
(FC2_0 = 0b110)                   ! /* found a break in the opcode addr
                                  sequence, ready */ !


%% CheckBrAddress
(ADDRESS = dest_address /* dest_address is computed before */ )\
                                  /* branch executed */ \
                                  !{ new_state = TELL} GOTO[state] SelUnused !
(ADDRESS = 0x......)              ! /* branch not executed */ !


%% SelUnused
(state < new_state, FC2_0 = 0b110) ! UNUSED NEXT { state = TELL } SelUnused !
(state = new_state)               ! /* ready */ !
(state = 0x....)                  ! NEXT { state = TELL } SelUnused !
```

Processing data transfers

At this point all the instructions occurring in the measurement are decoded.
Some instructions are not handled completely yet.
A few instructions result in data transfers as e.g. the RTS instruction.
Other instructions result in 'unused opcodes'.

After decoding an instruction the number of data-transfers following the instruction is known. These data transfers should be treated as part of the instruction.
As an example how to handle this the decoding of the MOVE.W instruction (line +0002) is given in the next table line:

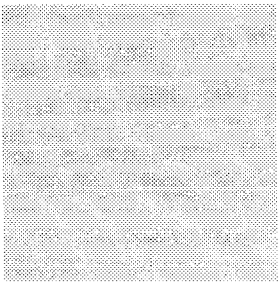
```
(DATA = 0b0011_0010_0001_0010)    ! "MOVE.W\t(A2),D1" FindDataRead MR !
```

The search for the data transfer is done by means of the FindDataRead table which is already described with the RTS instruction before.
The resulting data transfer of this instruction can be found on line +0004. Due to the microprocessor pipeline the next instruction opcode (line +0003) is fetched before the MOVE instruction data appears on the bus.
The next opcode is part of the following instruction and may not be treated as an unused prefetch (UNUSED) or part of this instruction (PROG).
It could be given the command 'SKIP' but because that is the default display selection command on all except the first disassembler state no display selection command at all is given when searching the data transfer.
The instruction in the subroutine (line +0008) is an instruction resulting in a memory write action. The disassembly procedure for this instruction is almost the same as for the previous MOVE-instruction.

```
(DATA = 0b0011_1010_1000_0001)    ! "MOVE.W\tD1,(A5)" FindDataWrite MW !

%% FindDataWrite
(FC2_0 = 0b101, DSCTRL = 0b000)   ! /* found, stop searching */ !
(FC2_0 = 0b...)                   ! /* not found yet */ NEXT FindDataWrite !
```

Branch instruction

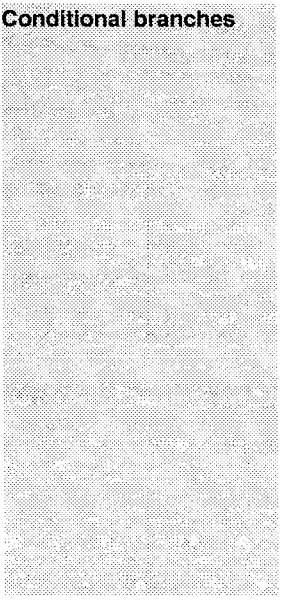
In this way the other instructions resulting in data transfers can also be handled. Now the handling of the possible 'unused opcodes' for the BSR, BNE and BEQ instructions has to be done.



The effect of the BSR instruction in this example is a branch to the computed address. How to compute the address is explained before. The effect of the RTS instruction explained before also has a branch to a computed address. The search for the computed address can be done the same way as it is done for the RTS instruction.
Additionally the BSR instruction causes two data transfer memory write actions in which the current program counter is written to the stack.
The disassembly for the BSR instruction then looks like:

```
(DATA = 0b0110_0001_...._....)    ! "BSR\t" \
                                  Compute8bitBrAddr print[1]  \
                                  { last_opc = TELL } \
                                  FindNewOpcAddress[0] CheckBrAddress \
                                  GOTO[last_opc] \
                                  FindDataWrite MW NEXT FindDataWrite MW !
```

Conditional branches



The conditional branches BEQ and BNE do not result in data transfer actions.
Because the BNE and BEQ instructions are conditional the next opcode is sometimes an unused opcode as seen with the BSR instruction and sometimes it is the first opcode of the following executed instruction.
The only possibility for the disassembler to distinguish these two cases is to investigate if the branch was taken or not by looking ahead and testing the ADDRESS label for appearance of the computed branch address.
If it appears the branch is assumed and the opcode after the instruction is an unused opcode. Otherwise the branch is not taken and the first opcode after the instruction is the beginning of the following instruction. The look ahead process is already used with the RTS instruction in the tables FindNewOpcAddress and CheckBrAddress together with their sub tables.
The CheckBrAddress will automatically select the possibly unused opcode prefetches with the 'UNUSED' commands if the branch is taken. This results in the following table lines:

```
(DATA = 0b0110_0111_...._....)    ! "BEQ\t" \
                                  Compute8bitBrAddr print[1] \
                                  FindNewOpcAddress[0] CheckBrAddress !
(DATA = 0b0110_0110_...._....)    ! "BNE\t" \
                                  Compute8bitBrAddr print[1] \
                                  FindNewOpcAddress[0] CheckBrAddress !
```

Static char variable usage

The instructions are all completely handled with the disassembler descriptions made sofar. The only thing to do yet is to report errors if the disassembler does not automatically loose instruction synchronization.
An example how to do this can best be illustrated for the unconditional branch instructions BSR and RTS. If the instruction is decoded the computed new opcode address must appear on the address label. If the computed address can not be found the disassembly is not ok and the disassembler should loose synchronization status. One way to loose instruction synchronization within the custom disassembler is calling an empty lookup table or an index table with a non existing index. This would result in displaying '***' when decoding the instruction. Because the instruction itself was decoded properly but the result of the instruction was not correct the disassembler should loose synchronization status at the start of the next instruction. This can be achieved by reporting a status from this instruction to the next by use of the 'static char' variable. Before starting the decoding of the next instruction the disassembler should check this status and loose instruction synchronization.

The static char variable can be declared in the 'DEF' section:

```
static char instr_status;
```

To detect if an illegal branch or return instruction is encountered a new table with the test on the required address is created. If the required address is not found the static char variable is set. When decoding the next instruction the value of the static char variable is checked on the first line in the STATUS_TAB.

If the static char variable is set a lookup table sync_lost is called which results in loosing synchronization status and printing '***' in the disassembler output column on your logic analyzer. For the RTS instruction this results in a additional call to the CheckSync table after the CheckBrAddress is done.
The BSR instruction will have a similar call to the CheckSync table after the CheckBrAddress. The BEQ and BNE instructions do not have the call to the CheckSync

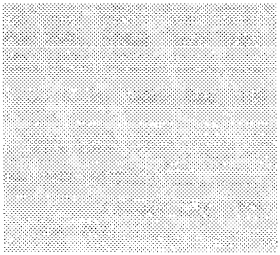
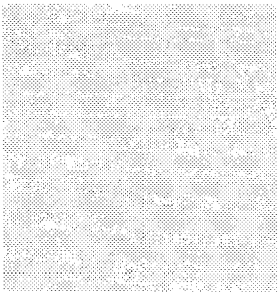


table because these are conditional branches.
For readability purposes the value for the static char variable to report the synchronization lost status is given in a constant definition 'SYNC_LOST'. Because the static char variable initially is 0, the constant definition for synchronization lost can be any value except 0. The value 0 is defined as SYNC_OK.
The resulting tables and the RTS instruction decoding then look like:

```
%% STATUS_TAB
(instr_status = SYNC_LOST)          ! sync_lost { instr_status = SYNC_OK }!
(FC2_0 = FC_PROGRAM, DSCTRL=0b100)  ! decode !
(FC2_0 = FC_DATA,    DSCTRL=0b1..)  ! MR !      /* memory read */
(FC2_0 = FC_DATA,    DSCTRL=0b0..)  ! MW !      /* memory write */
(FC2_0 = 0b...)                     ! "reserved function code" !

%% decode
......
(DATA = 0b0100_1110_0111_0101)      ! "RTS\t"      \
                                    { last_opc = TELL } \
                                    FindDataRead \
                                    ComputeReturnAddr[0] \
                                    print[1] \
                                    GOTO[last_opc] \
                                    FindNewOpcAddress[0] CheckBrAddress \
                                    CheckSync !
......

%% CheckSync
(ADDRESS = dest_address)            ! { instr_status = SYNC_OK } !
(ADDRESS != dest_address)           ! { instr_status = SYNC_LOST } !

%% sync_lost
(instr_status = SYNC_OK)            !{ instr_status = SYNC_LOST } sync_lost !
```



The BSR instruction decoding can be changed the same way as is done for the RTS instruction.

Finishinging the example disassembler

The complete description of the example 68000 disassembler is given below.
In this table you will notice a change of the FindDataRead and FindDataWrite tables. A check to assure the searched data transfer is found after the microprocessor opcode fetch pipeline is added. This is another possibility to synchronize the disassembler with the measurement.
The use of the tab-positions keyword 'tab' is shown in which two absolute tab-positions are given and an offset starting from the last absolute tab position to the next tab position.
For readability purposes a few constant definitions were added. For the FC2_0 label the constants can be used in the lookup tables The constants are FC_PROGRAM and FC_DATA. They are used in conditions instead of the values 0b110 and 0b101 which makes the condition for the FC2_0 label easier to understand.

```
%% DEF
long        nopc_addr, dest_address;
int         pipeline;
int         datatr_opcpipe;
int         last_opc;
int         state;
int         new_state;
static char instr_status;

#define     MAXPIPE                 3

#define     FC_DATA                 0b101
#define     FC_PROGRAM              0b110

#define     SYNC_OK                 0
#define     SYNC_LOST               1

%% EQU
/* lookup tables */
LT: STATUS_TAB, decode, ChkNextOpc
LT: Compute8bitBrAddr, FindDataRead, FindDataWrite
LT: CheckBrAddress
LT: ChkAddrSeq
LT: FindAddrSeqBrk
LT: SelUnused
LT: CheckSync
LT: sync_lost
```

```
/* index tables */
IT: FindNextOpc, print, ComputeReturnAddr
IT: FindNewOpcAddress

%% FORMAT

logo: "68000 C-DISA Example"

head: "68000 C-DISA Example"  39

tab: 10,32 7

pods: threshold = {{TTL, TTL},
                   {TTL, TTL},
                   {TTL, TTL},
                   {TTL, TTL}
                   }

clock: udsn ={ name = "UDSN",
               edge = rising,
               channel = 31,
               qualifier = { channels = { 52 },
                             required = no,
                             levels = { high }
                           }
             }

clock: ldsn ={ name = "LDSN",
               edge = rising,
               channel = 30,
               qualifier = { channels = { 52 },
                             required = no,
                             levels = { high }
                           }
             }
label: DSCTRL =  { name = "DSCTRL",
               radix = bin,
               channels = { 29, 28, 27 }
                }

label: FC2_0 ={ name = "FC2_0",
              radix = bin,
              channels = { 26, 25, 24 }
               }

label: ADDRESS ={ name = "ADDRESS",
                  channels = { 23, 22, 21, 20, 19, 18, 17, 16,
                               15, 14, 13, 12, 11, 10,  9,  8,
                                7,  6,  5,  4,  3,  2,  1,  0}
```

```
                }
label: DATA ={ name = "DATA",
               channels = { 47, 46, 45, 44, 43, 42, 41, 40,
                            39, 38, 37, 36, 35, 34, 33, 32}
                }


clockseq: SEQ1 = { udsn }
clockseq: SEQ2 = { ldsn }


%% START
! STATUS_TAB !


%% STATUS_TAB
(instr_status = SYNC_LOST)              ! sync_lost { instr_status = SYNC_OK } !
(FC2_0 = FC_PROGRAM, DSCTRL=0b100)      ! decode !
(FC2_0 = FC_DATA,    DSCTRL=0b1..)      ! MR !/* memory read */
(FC2_0 = FC_DATA,    DSCTRL=0b0..)      ! MW !/* memory write */
(FC2_0 = 0b...)                         ! "reserved function code" !

%% decode                       /* _ in a condition is only used for readability */
(DATA = 0b0011_0110_0011_1100)          ! "MOVE.W\t#" FindNextOpc[0] PROG \
                                        print[0] ",D3" !
(DATA = 0b0011_0010_0001_0010)          ! "MOVE.W\t(A2),D1" FindDataRead MR !
(DATA = 0b0011_1010_1000_0001)          ! "MOVE.W\tD1,(A5)" FindDataWrite MW !
(DATA = 0b0010_0110_0111_1100)          ! "MOVEA.L\t#" FindNextOpc[0] PROG \
                                        print[0] FindNextOpc[0] PROG \
                                        print[0] ",A3"!
(DATA = 0b0110_0001_...._....)          ! "BSR\t" \
                                        Compute8bitBrAddr print[1]  \
                                        { last_opc = TELL } \
                                        FindNewOpcAddress[0] CheckBrAddress \
                                        CheckSync \
                                        GOTO[last_opc] \
                                        FindDataWrite MW NEXT FindDataWrite MW !
(DATA = 0b0101_[...]1_0100_0011)        ! "SUBQ.W\t#%d,D3" !
(DATA = 0b0100_1110_0111_0101)          ! "RTS\t" \
                                        { last_opc = TELL } \
                                        FindDataRead \
                                        ComputeReturnAddr[0] \
                                        print[1] \
                                        GOTO[last_opc] \
                                        FindNewOpcAddress[0] CheckBrAddress \
                                        CheckSync !
(DATA = 0b0110_0111_...._....)          ! "BEQ\t" \  Compute8bitBrAddr \
                                        print[1] \
                                        FindNewOpcAddress[0] CheckBrAddress !
```

```
(DATA = 0b0110_0110_...._....)          ! "BNE\t" \
                                        Compute8bitBrAddr print[1] \
                                        FindNewOpcAddress[0] CheckBrAddress !

%% FindNextOpc
(ADDRESS=0x[......])                    ! { nopc_addr = $1 + 2 } NEXT ChkNextOpc !


%% ChkNextOpc                           /* lookup table */
(ADDRESS = nopc_addr, FC2_0 = FC_PROGRAM,   DSCTRL = 0b100)  ! /* found opc */ !
(FC2_0 != FC_PROGRAM)                   ! NEXT ChkNextOpc !
                                        /* if we didn't find the next   opcode,
                                        function-code 6 is illegal. The disassem-
                                        bler looses synchronisation status */


%% Compute8bitBrAddr  /* data label contains 8-bit signed offset to ADDRESS */
(DATA=0b<.>8_0[<.>7], ADDRESS=0x[......])! /* positive offset */ \
                                        {dest_address = $2 + $1;\
                                        /* compute destination address */ \
                                        dest_address = dest_address + 2 } \
                                        /* adjust ADDRESS value */ \
                                        "+%d"   /* print offset */ !
(DATA=0b<.>8_1[<.>7], ADDRESS=0x[......])! /* negative offset */ \
                                        {$1 = $1 - 0x80;   /* sign extend $1 */ \
                                        dest_address = $2 + $1; \
                                        /* compute destination address */ \
                                        dest_address = dest_address + 2 }\
                                        /* adjust ADDRESS value */ \
                                        "%d"  /* print offset */ !

%% print
(DATA=0x[....])                         ! "%04x" !
(dest_address = 0x..[......])           ! "\t{%06x}" !


%% FindDataRead
(FC2_0 = FC_DATA, DSCTRL = 0b1.., datatr_opcpipe > 1) ! /* found */ !
(FC2_0 = FC_PROGRAM)                    ! /* not found yet */ \
                                        { datatr_opcpipe = datatr_opcpipe + 1 } \
                                        NEXT FindDataRead !
(FC2_0 = 0b...)                         ! /* not found yet */ NEXT FindDataRead !


%% FindDataWrite
(FC2_0 = FC_DATA, DSCTRL = 0b0.., datatr_opcpipe > 1) ! /* found */ !
(FC2_0 = FC_PROGRAM)                    ! /* not found yet */ \
                                        { datatr_opcpipe = datatr_opcpipe + 1 } \
```

```
                                    NEXT FindDataWrite !
(FC2_0 = 0b...)                     ! /* not found */ NEXT FindDataWrite !

%% ComputeReturnAddr
(DATA=0x[....])                     ! { dest_address = $1 << 16 } MR NEXT \
                                    FindDataRead ComputeReturnAddr[1] !
(DATA=0x[....])                     !{ dest_address = dest_address + $1 } MR !


%% FindNewOpcAddress
! FindNextOpc[0] { state = TELL } FindAddrSeqBrk !


%% FindAddrSeqBrk
(pipeline > MAXPIPE)                !/* not found, reposition */ GOTO[state] !
(ADDRESS=0b[<.>24])                 ! { nopc_addr = $1; \
                                    pipeline = pipeline + 1 } \
                                    NEXT ChkAddrSeq !


%% ChkAddrSeq
(ADDRESS = nopc_addr, FC2_0 = FC_PROGRAM, DSCTRL = 0b1..) ! FindAddrSeqBrk !
(FC2_0 != FC_PROGRAM)               ! NEXT ChkAddrSeq !
(FC2_0 = FC_PROGRAM)                ! /* found addr sequence break, ready */ !


%% CheckBrAddress
(ADDRESS = dest_address /* dest_address computed before */ ) \
                                    /* branch executed */ \
                                    ! { new_state = TELL} GOTO[state] \
                                    SelUnused !
(ADDRESS = 0x......)                ! /* branch not executed */ !


%% SelUnused
(state < new_state, FC2_0 = FC_PROGRAM)! UNUSED NEXT {state = TELL} SelUnused !
(state = new_state)                 ! /* ready */ !
(state = 0x....)                    ! NEXT { state = TELL } SelUnused !

%% CheckSync
(ADDRESS = dest_address)            ! { instr_status = SYNC_OK } !
(ADDRESS != dest_address)           ! { instr_status = SYNC_LOST } !

%% sync_lost
(instr_status = SYNC_OK)            ! { instr_status = SYNC_LOST } sync_lost !
```

Chapter 5

Writing a 68030 Disassembler

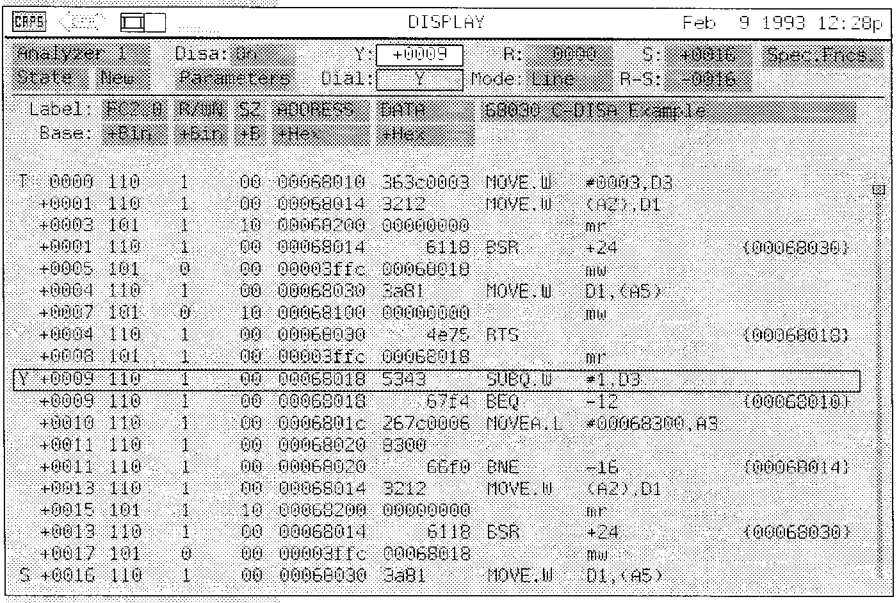
Disassembler labels 5-3
Disassembler part label 5-4
Positioning within a disassembler state 5-4
Processing data transfers 5-5
Search following opcode 5-6
Branch instructions 5-6
Unused opcode fetches 5-8
Using data transfer states 5-10

In the previous chapter an example was given how to write a 68000 disassembler. Suppose an extension this 68000 disassembler to a 68030 disassembler has to be made.

In this chapter references are made to the previous chapter 'Writing a 68000 disassembler'. It is assumed that the previous chapter is read before.

The opcodes for the instructions are the same as for the 68000 disassembler. The 68030 disassembler has a 32-bit databus DATA label. This data label can contain 2 instructions in one disassembler state. The measurement to disassemble contains the same instruction loop as the 68000 example in the previous chapter.

The disassembler should produce the next output in the display menu of your PM 3580/PM 3585 logic analyzer if program context mode is enabled.



The measurement file is available on disk with the filename DEM68030.NEW. The disassembler description file is available on disk with the filename DEM68030.DSC. Use the CDISA80 compiler to create a loadable disassembler DEM68030.DIS.

If program context is disabled the following output should be produced in the display menu. The differences can be seen in the order of disassembler states shown and in the appearance of unused opcode fetches on disassembler state +0002, +0006, +0012 and +0014.

```
CAPS                          DISPLAY                        Feb  9 1993 12:29p
Analyzer 1    Disa: On        Y: +0003     R:  0000   S: +0016   Spec.Fncs.
State  New    Parameters      Dial: Y      Mode: Line   R-S: -0016
  Label: FC2_0 R/WN SZ ADDRESS  DATA     68030 C-DISA Example
   Base: +Bin  +bin  +b +Hex    +Hex
T  0000 110    1    00 00068010 363c0003 MOVE.W   #0003,D3
  +0001 110    1    00 00068014 3212     MOVE.W   (A2),D1
  +0001 110    1    00 00068014     6118 BSR      +24              {00068030}
  +0002 110    1    00 00068018 534367f4          unused opc
  +0003 101    1    10 00068200 00000000          mr
  +0004 110    1    00 00068030 3a81     MOVE.W   D1,(A5)
  +0004 110    1    00 00068030     4e75 RTS                      {00068018}
  +0005 101    0    00 00003ffc 00068018          mw
  +0006 110    1    00 00068034 4e714e71          unused opc
  +0007 101    0    10 00068100 00000000          mw
Y +0008 101    1    00 00003ffc 00068018          mr
  +0009 110    1    00 00068018 5343     SUBQ.W   #1,D3
  +0009 110    1    00 00068018     67f4 BEQ      -12              {00068010}
  +0010 110    1    00 0006801c 267c0006 MOVEA.L  #00068300,A3
  +0011 110    1    00 00068020 8300              opc
  +0011 110    1    00 00068020     66f0 BNE      -16              {00068014}
  +0012 110    1    00 00068024 4e714e71          unused opc
  +0013 110    1    00 00068014 3212     MOVE.W   (A2),D1
  +0013 110    1    00 00068014     6118 BSR      +24              {00068030}
  +0014 110    1    00 00068018 534367f4          unused opc
```

Disassembler labels

As already mentioned the instruction opcodes are the same as for the 68000 example explained in the previous chapter. The 68030 also has a FC2_0 label containing information about the function code and a R/WN label with information about the Read or Write status. The ADDRESS and DATA label contain the microprocessors address-bus and data-bus respectively. The SZ label contains the 68030 SIZE1 and SIZE0 lines which reports the data bus size requested by the microprocessor for this transfer. The microprocessor asks for a 32-bit data bus if a value of 0b00 is available for the SZ label. With a value of 0b10 for the SZ label the requested size of the data bus is 16 bit.

With this basic 68030 information the extension of the 68000 disassembler to a 68030 disassembler can be started.

Disassembler part label

As already mentioned one disassembler state can contain 2 instructions for the 68030. The data-label should therefore be handled in 2 parts for decoding the instruction. This can be achieved by using the label definition element 'parts = 2' in the definition of the data label. Because there are 2 different ways of interpreting the numeric value on a data bus, known as little endian and big endian, this bus type also has to be defined in the data label definition. The definition of the DATA label in the FORMAT section of the description file for this example then looks like:

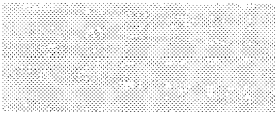
```
label: DATA =  { name = "DATA",
               parts = 2, /* label accessible for the disa in 2 separate parts */
               type = big, /* big endian bus type */
               channels = { 63, 62, 61, 60, 59, 58, 57, 56,
                            55, 54, 53, 52, 51, 50, 49, 48,
                            47, 46, 45, 44, 43, 42, 41, 40,
                            39, 38, 37, 36, 35, 34, 33, 32}
                  }
```

Positioning within a disassembler state

The use in the disassembly tables of he label DATA only represents one part of the entire measurement label. This means that only 16 bits at a time are used in table line conditions and other label references.
Because one disassembler state can contain 2 instructions, positioning within a disassembler state should be possible. This position is similar to the NEXT and PREV commands, that are used to position between disassembler states.
To change and retrieve the current position within a disassembler state the following commands are available:

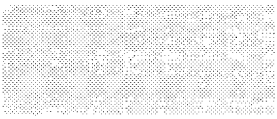
- NEXTPART
- PREVPART
- GOTOPART
- TELLPART

These commands are comparable to the commands available for positioning between disassembler states (NEXT, PREV, GOTO and TELL).
To start a 68030 instruction decoding the FC2_0 label should have value 0b110, the disassembler state should be a read action: RWN=0b1. The requested size should be



32 bits: SZ = 0b00. The slightly changed STATUS_TAB table in the 68030 description against the 68000 description is shown below.

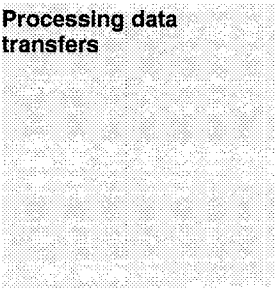
```
%% STATUS_TAB
(FC2_0 = 0b110, RWN=0b1, SZ=0b00)        ! decode !
(FC2_0 = 0b101, RWN=0b1)                 ! MR !  /* memory read */
(FC2_0 = 0b101, RWN=0b0)                 ! MW !  /* memory write */
(FC2_0 = 0b...)                          ! "reserved function code" !
```



The opcodes for the instructions are the same as for the 68000 disassembler in the previous chapter. The decoding line for the "MOVE.W (A2),D1" instruction then looks the same as for the 68000 disassembler:

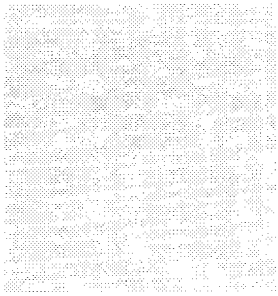
```
(DATA = 0b0011_0010_0001_0010)      ! "MOVE.W\t(A2),D1" FindDataRead MR !
```

Processing data transfers



Because the DATA label is accessed in 2 parts the DATA label size for a condition is 16 bits. For the 68000 the FindDataRead table counted the opcode fetches found after the instruction.
The opcode fetches for the 68030 should be handled part by part so positioning in parts should be done for counting the opcode fetches. The found opcode fetches are counted in the variable 'datatr_opcpipe'. This results in the following FindDataRead table:

```
%% FindDataRead
(FC2_0 = 0b101, RWN = 0b1, datatr_opcpipe > 2)! /* found */ !
(FC2_0 = 0b110)                          ! /* not found yet */ \
                                         { datatr_opcpipe = datatr_opcpipe + 1 } \
                                         NEXTPART FindDataRead !
(FC2_0 = 0b...)                          ! /* not found yet */ NEXT FindDataRead !
```



The differences with the 68000 FindDataRead are:

- The opcode pipeline is at least 2 opcodes
- The positioning command after finding an opcode is NEXTPART which means: go to the next part in the current disassembler state or if already positioned at the last part of the current disassembler state go to the next disassembler state.

Search following opcode

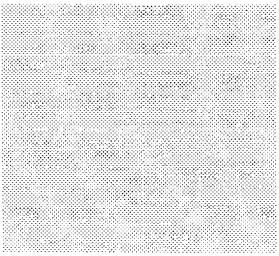
The 'MOVE.W #0003,D3' instruction decoding on line 0000 looks almost the same as the 68000 decoding. The call to the FindNextOpc table has no parameter anymore. This FindNextOpc table has changed into a Lookup table. The reason for this change is that the 'ADDRESS' label is used to find the next opcode. When positioning within a disassembler state for the 68030 the ADDRESS label is not changed. To handle this the part at which we are positioned within the disassembler state should also be checked for finding the next opcode. The current part (achieved with the command TELLPART) together with the ADDRESS label value is used in the ChkNextOpc table to determine if the next instruction opcode is found.
The resulting FindNextOpc and ChkNextOpc tables are shown below.

```
%% FindNextOpc
(ADDRESS=0b[<.>32], TELLPART = 0)     ! { nopc_addr = $1; nopc_part = 1 } \
                                       NEXTPART ChkNextOpc !
(ADDRESS=0b[<.>32], TELLPART = 1)     ! { nopc_addr = $1 + 4; nopc_part = 0 } \
                                       NEXTPART ChkNextOpc !


%% ChkNextOpc                         /* lookup table */
(ADDRESS = nopc_addr, TELLPART = nopc_part, FC2_0 = 0b110, \
RWN = 0b1, SZ = 0b00, DSACK = 0b11)    ! /* found next opcode */ !
(FC2_0 != 0b110)                       ! NEXT ChkNextOpc !
                                       /* if we didn't find the next opcode
                                       function-code 0b110 is illegal.
                                       The disassembler looses his synchroni-
                                       sation status */
```

Branch instructions

For the branch instructions the table Compute8bitBrAddr should also consider the address label value and the current disassembler state part for determining the new address value.
The computed dest_address is used for printing. This dest_address must be transferred to a value for the ADDRESS label and the part for the disassembler state containing this address value which is done in a new CompNewAddr table.
This gives the following tables:

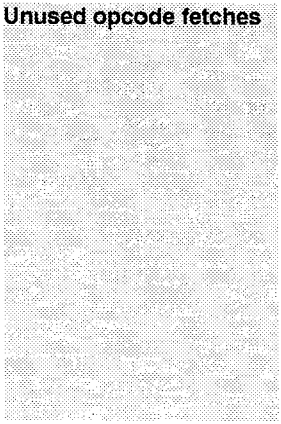
```
%% Compute8bitBrAddr  /* data label contains 8-bit signed offset to ADDRESS */
(DATA=0b<.>8_0[<.>7], ADDRESS=0x[........], TELLPART = 0)
                            ! /* compute destination address */ \
                            /* positive offset */ \
                            {dest_address = $2 + $1;\
                            dest_address = dest_address + 2 }\
                            /* adjust for ADDRESS value */ \
                            CompNewAddr /* adjust for ADDRESS bus */ \
                            "+%d"   /* print offset */ !

(DATA=0b<.>8_0[<.>7], ADDRESS=0x[........], TELLPART = 1)
                            ! /* positive offset */ \
                            {dest_address = $2 + $1;\
                            /* compute destination address */ \
                            dest_address = dest_address + 4 }\
                            /* adjust for ADDRESS value */ \
                            CompNewAddr /* adjust for ADDRESS bus */ \
                            "+%d"    /* print offset */ !

(DATA=0b<.>8_1[<.>7], ADDRESS=0x[........], TELLPART = 0)
                            ! /* negative offset */ \
                            {$1 = $1 - 0x80;    /* sign extend $1 */ \
                            dest_address = $2 + $1;\
                            /* compute destination address */ \
                            dest_address = dest_address + 2 }\
                            /* adjust for ADDRESS value */ \
                            CompNewAddr /* adjust for ADDRESS bus */ \
                            "%d"   /* print offset */ !

(DATA=0b<.>8_1[<.>7], ADDRESS=0x[........],TELLPART = 1)
                            !/* negative offset */ \
                            {$1 = $1 - 0x80;  /* sign extend $1 */ \
                            dest_address = $2 + $1;\
                            /* compute destination address */ \
                            dest_address = dest_address + 4 }\
                            /* adjust for ADDRESS value */ \
                            CompNewAddr /* adjust for ADDRESS bus */ \
                            "%d"  /* print offset */ !

%% CompNewAddr
(dest_address = 0b<.>30_10)  ! { new_address = dest_address - 2;\
                            new_part = 1 } !
(dest_address = 0b<.>30_00)  ! { new_address = dest_address ;\
                            new_part = 0 } !
```



The new value on the ADDRESS label is computed and the search for this value on the ADDRESS label can be started. This is done the same way as it is done for the 68000 disassembler.
The tables used to achieve this goal are changed only to support the multiple parts in the data label for one disassembler state in order to count the opcodes. The resulting tables to find the new address on the ADDRESS label look like:

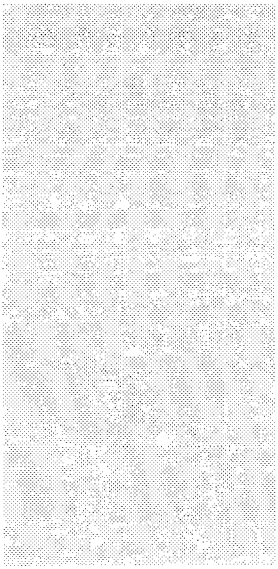
```
%% FindNewOpcAddress
! FindNextOpc { state = TELL; part = TELLPART } FindAddrSeqBrk !


%% FindAddrSeqBrk
(pipeline > MAXPIPE)                    ! /* not found, reposition */ \
                                        GOTO[state] GOTOPART[part] !
(ADDRESS=0b[<.>32], TELLPART = 0)       ! { nopc_addr = $1; nopc_part = 1 } \
                                        { pipeline = pipeline + 1 } \
                                        NEXTPART ChkAddrSeq !
(ADDRESS=0b[<.>32], TELLPART = 1)       ! { nopc_addr = $1 + 4; nopc_part = 0 } \
                                        { pipeline = pipeline + 1 } \
                                        NEXTPART ChkAddrSeq !


%% ChkAddrSeq
(ADDRESS = nopc_addr, TELLPART = nopc_part, FC2_0 = 0b110, \
 RWN = 0b1, SZ = 0b00)                  ! FindAddrSeqBrk !
(FC2_0 != 0b110)                        ! NEXT ChkAddrSeq !
(FC2_0 = 0b110)                         ! /* found addr sequence break, ready */ !
```

Unused opcode fetches



If a possible new address is found a check is made if it is the correct address. If so the fetched opcodes between the last instruction opcode and the new disassembler state are selected as UNUSED.
These CheckBrAddr and SelUnused tables are also almost the same as for the 68000 disassembler example with changes made for the multiple parts in the data label.

```
%% CheckBrAddr
(ADDRESS = new_address /* new_address computed in CompNewAddr table */ )\
                                    /* branch executed */ \
                                    ! { new_state = tell }\
                                    GOTO[state] GOTOPART[part] SelUnused !
(ADDRESS = 0x........)              ! /* branch not executed */ !


%% SelUnused
(state < new_state, FC2_0 = 0b110)  ! UNUSED NEXTPART { state = TELL }\
                                    SelUnused !
(state = new_state, TELLPART < new_part )! UNUSED NEXTPART SelUnused !
(state = new_state)                 ! /* ready */ !
(state = 0x....)                    ! NEXTPART { state = TELL} SelUnused !
```



The RTS instruction results in a branch to a new address. The branch to the computed address can be handled the same way as is done for the other branch instructions BSR, BNE and BEQ. Determining the branch address needs an explanation. In the decode table the disassembly of the RTS instruction is almost the same as for the 68000 disassembler:

- Print the instruction text: "RTS\t".
- Save the last instruction opcode disassembler state: TELL and TELLPART.
- Search the stack read: FindDataRead
- Determine the return address: ComputeReturnAddr.
- Print the return address: print.
- Reposition to the last instruction opcode: GOTO and GOTOPART.
- Search the return address: FindNewOpcAddress and CheckBrAddr.

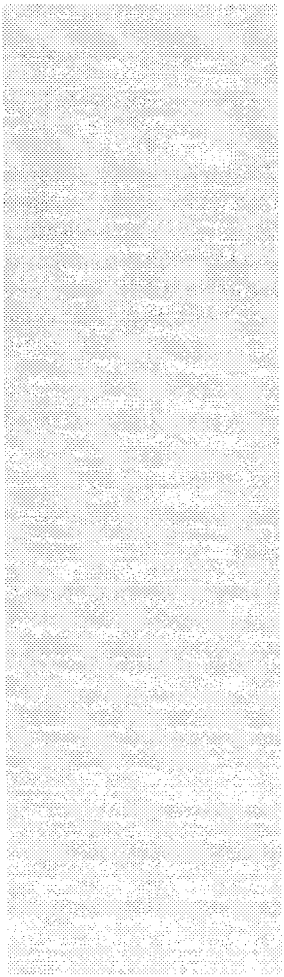
```
(DATA = 0b0100_1110_0111_0101) ! "RTS\t" \
                               { last_opc = TELL; last_opc_part = TELLPART } \
                               FindDataRead \
                               ComputeReturnAddr[0] print[1] \
                               GOTO[last_opc] GOTOPART[last_opc_part] \
                               FindNewOpcAddress[0] CheckBrAddr !
```

Using data transfer states

The table ComputeReturnAddr is different against the 68000 table because of the multiple parts in the DATA label.

```
%% ComputeReturnAddr
(DATA=0x[....])

(DATA=0x[....])
```

```
! { dest_address = $1 << 16 } NEXTPART \
ComputeReturnAddr[1] !
! { dest_address = dest_address + $1 } MR \
CompNewAddr /* adjust for ADDRESS bus */ !
```



The return address of the RTS instruction is contained in one disassembler state.

In the example on line +0008. The total data label value (32-bit) has to be read in 2 phases because the DATA label can only be accessed in parts of 16-bit. First the high part of the data label, being the first part accessed, is read in the dest_address variable.

It is not allowed to already select this disassembler state as a memory read data transfer (MR) yet, because then this disassembler state only has one part anymore and only the first part of the DATA label can be accessed. Before selecting this disassembler state as data transfer the low part of the DATA label should be accessed.

This is done by proceeding to the next part in the disassembler state and adding the low part of the DATA label to the dest_address variable. All the information needed from the DATA label in this disassembler state is available and the disassembler state can be specified for the display menu as a data transfer. The return address from the RTS instruction is computed and can be used for the search to the next executed instruction opcode as is done for the branch instructions BSR, BNE and BEQ.

The complete description for the 68030 disassembler which is able to disassemble the EXAMPLE3.NEW measurement is given on the following pages.

```
%% DEF
long        nopc_addr;
long        dest_address;
long        new_address;
int         nopc_part;
int         new_part;
int         pipeline;
int         datatr_opcpipe;
int         state;
int         part;
int         new_state;
int         last_opc;
int         last_opc_part;

#define     MAXPIPE                 8

#define     FC_DATA                 0b101
#define     FC_PROGRAM              0b110

%% EQU
/* lookup tables */
LT: STATUS_TAB, decode, ChkNextOpc
LT: Compute8bitBrAddr, FindDataRead, FindDataWrite
LT: CheckBrAddr, FindNextOpc, CompNewAddr
LT: FindAddrSeqBrk
LT: ChkAddrSeq
LT: SelUnused

/* index tables */
IT: print, ComputeReturnAddr
IT: FindNewOpcAddress

%% FORMAT

logo: "68030 C-DISA Example"

head: "68030 C-DISA Example"  35

Pods: threshold = {{TTL, TTL},
                   {TTL, TTL},
                   {TTL, TTL},
                   {TTL, TTL},
                   {TTL, TTL}
                   }

tab: 10,26 7
```

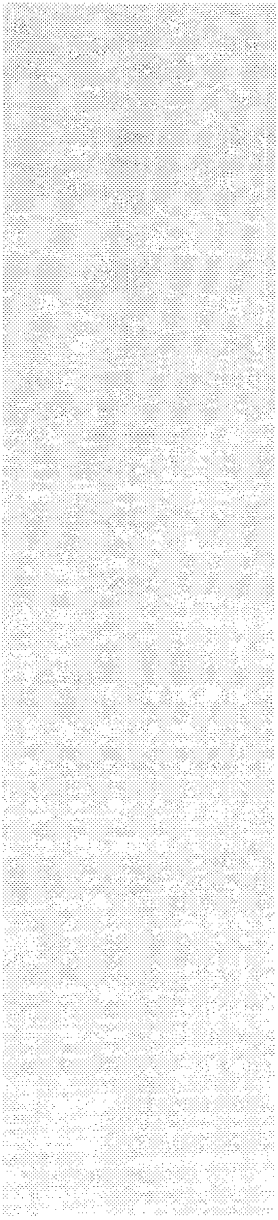
```
clock: dsn = { name = "DSN",
          edge = rising,
          channel = 77,
          qualifier = { channels = { 78 },
                   required = no,
                   levels = { high }
                          }
            }
clock: asn = { name = "ASN",
          edge = rising,
          channel = 76,
          qualifier = { channels = { 78 },
                   required = no,
                   levels = { high }
                          }
            }
label: FC2_0 =    { name = "FC",
              radix = bin,
              channels = { 66, 65, 64 }
               }

label: RWN =  { name = "R/WN",
           radix = bin,
           channels = { 67 }
              }

label: SZ =  { name = "SZ",
          radix = bin,
           channels = { 69, 68 }
              }

label: ADDRESS ={ name = "ADDRESS",
                                    channels = { 31, 30, 29, 28, 27, 26, 25,
                                                 24,23, 22, 21, 20, 19, 18,
                                                 17, 16,15, 14, 13, 12, 11,
                                                 10,  9,  8,  7,  6,  5,  4,
                                                 3,  2,  1,  0}
              }

label: DATA =  { name = "DATA",
                                    parts = 2,
                                    type = big,
                                    channels = { 63, 62, 61, 60, 59, 58, 57,
                                                 56,55, 54, 53, 52, 51, 50,
                                                 49, 48, 47, 46, 45, 44, 43,
                                                 42, 41, 40, 39, 38, 37, 36,
                                                 35, 34, 33, 32}
              }
```

```
clockseq: SEQ1 = { dsn }
clockseq: SEQ2 = { asn }


%% START
! STATUS_TAB !


%% STATUS_TAB
(FC2_0 = FC_PROGRAM, RWN=0b1, SZ=0b00) ! decode !
(FC2_0 = FC_DATA, RWN=0b1)             ! mr !/* memory read */
(FC2_0 = FC_DATA, RWN=0b0)             ! mw !/* memory write */
(FC2_0 = 0b...)                        ! "reserved function code" !

% decode                        /* _ in a condition is only used for readability */
(DATA = 0b0011_0110_0011_1100)         ! "MOVE.W\t#" FindNextOpc PROG \
                                       print[0] ",D3" !
(DATA = 0b0011_0010_0001_0010)         ! "MOVE.W\t(A2),D1" FindDataRead MR !
(DATA = 0b0011_1010_1000_0001)         ! "MOVE.W\tD1,(A5)" FindDataWrite MW !
(DATA = 0b0010_0110_0111_1100)         ! "MOVEA.L\t#" FindNextOpc PROG \
                                       print[0] FindNextOpc PROG \
                                       print[0 ",A3"!
(DATA = 0b0110_0001_...._....)         ! "BSR\t" \
                                       Compute8bitBrAddr print[1]  \
                                       { last_opc = TELL;\
                                       last_opc_part = TELLPART } \
                                       FindNewOpcAddress[0] CheckBrAddr \
                                       GOTO[last_opc] GOTOPART[last_opc_part] \
                                       FindDataWrite MW !
(DATA = 0b0101_[...]1_0100_0011)       ! "SUBQ.W\t#%d,D3" !
(DATA = 0b0100_1110_0111_0101)         ! "RTS\t" \
                                       { last_opc = TELL;\
                                       last_opc_part = TELLPART } \
                                       FindDataRead \
                                       ComputeReturnAddr[0] print[1] \
                                       GOTO[last_opc] GOTOPART[last_opc_part] \
                                       FindNewOpcAddress[0] CheckBrAddr !
(DATA = 0b0110_0111_...._....)         ! "BEQ\t" Compute8bitBrAddr print[1] \
                                       FindNewOpcAddress[0] CheckBrAddr !
(DATA = 0b0110_0110_...._....)         ! "BNE\t" Compute8bitBrAddr print[1] \
                                       FindNewOpcAddress[0] CheckBrAddr !


%% FindNextOpc
(ADDRESS=0b[<.>32], TELLPART = 0)      ! { nopc_addr = $1; nopc_part = 1 } \
                                       NEXTPART ChkNextOpc !
(ADDRESS=0b[<.>32], TELLPART = 1)      ! { nopc_addr = $1 + 4; nopc_part = 0 } \
                                       NEXTPART ChkNextOpc !
```

```
%% ChkNextOpc                          /* lookup table */
(ADDRESS = nopc_addr, TELLPART = nopc_part, FC2_0 = FC_PROGRAM, \
RWN = 0b1, SZ = 0b00)                 ! /* found next opcode */ !
(FC2_0 != FC_PROGRAM)                 ! NEXT ChkNextOpc !
                                      /* if we didn't find the next opcode func-
                                      tion-code FC_PROGRAM is illegal
                                      The disassembler looses his synchronisa-
                                      tion status */

%% FindDataRead
(FC2_0 = FC_DATA, RWN = 0b1, datatr_opcpipe > 2)! /* found */ !
(FC2_0 = FC_PROGRAM)                  ! /* not found yet */ \
                                      { datatr_opcpipe = datatr_opcpipe + 1 } \
                                      NEXTPART FindDataRead !
(FC2_0 = 0b...)                       ! /* not found yet */ NEXT FindDataRead !

 %% FindDataWrite
(FC2_0 = FC_DATA, RWN = 0b0, datatr_opcpipe > 2)! /* found */ !
(FC2_0 = FC_PROGRAM)                  ! /* not found yet */ \
                                      { datatr_opcpipe = datatr_opcpipe + 1 } \
                                      NEXTPART FindDataWrite !
(FC2_0 = 0b...)                       ! /* not found yet */ NEXT FindDataWrite !


%% FindNewOpcAddress
! FindNextOpc { state = TELL; part = TELLPART } FindAddrSeqBrk !


%% FindAddrSeqBrk
(pipeline > MAXPIPE)                  ! /* not found, reposition */ \
                                      GOTO[state] GOTOPART[part] !
(ADDRESS=0b[<.>32], TELLPART = 0)     ! { nopc_addr = $1; nopc_part = 1 } \
                                      { pipeline = pipeline + 1 } \
                                      NEXTPART ChkAddrSeq !
(ADDRESS=0b[<.>32], TELLPART = 1)     ! { nopc_addr = $1 + 4; nopc_part = 0 } \
                                      { pipeline = pipeline + 1 } \
                                      NEXTPART ChkAddrSeq !


%% ChkAddrSeq
(ADDRESS = nopc_addr, TELLPART = nopc_part, FC2_0 = FC_PROGRAM, \
RWN = 0b1, SZ = 0b00)                 ! FindAddrSeqBrk !
(FC2_0 != FC_PROGRAM)                 ! NEXT ChkAddrSeq !
(FC2_0 = FC_PROGRAM)                  ! /* found addr sequence break, ready */ !

%% print
(DATA=0x[....])                 ! "%04x"      !
(dest_address = 0x[........])   ! "\t{%08x}" !
```

```
%% Compute8bitBrAddr    /* data label contains 8-bit signed offset to ADDRESS */
(DATA=0b<.>8_0[<.>7], ADDRESS=0x[........], TELLPART = 0) \
                              ! /* positive offset */ \
                              {dest_address = $2 + $1;\
                              /* compute destination address */ \
                              dest_address = dest_address + 2 } \
                              /* adjust for ADDRESS value */ \
                              CompNewAddr /* adjust for ADDRESS bus */ \
                              "+%d"       /* print offset */ !
(DATA=0b<.>8_0[<.>7], ADDRESS=0x[........], TELLPART = 1) \
                              ! /* positive offset */ \
                              {dest_address = $2 + $1; \
                              /* compute destination address */ \
                              dest_address = dest_address + 4 }\
                              /* adjust for ADDRESS value */ \
                              CompNewAddr /* adjust for ADDRESS bus */ \
                              "+%d"       /* print offset */ !
(DATA=0b<.>8_1[<.>7], ADDRESS=0x[........], TELLPART = 0) \
                              ! /* negative offset */ \
                              {$1 = $1 - 0x80; \
                              /* sign extend $1 */ \
                              dest_address = $2 + $1;\
                              /* compute destination address */ \
                              dest_address = dest_address + 2 } \
                              /* adjust for ADDRESS value */ \
                              CompNewAddr /* adjust for ADDRESS bus */ \
                              "%d"        /* print offset */ !
(DATA=0b<.>8_1[<.>7], ADDRESS=0x[........], TELLPART = 1) \
                              ! /* negative offset */ \
                              {$1 = $1 - 0x80; \
                              /* sign extend $1 */ \
                              dest_address = $2 + $1; \
                              /* compute destination address */ \
                              dest_address = dest_address + 4 }\
                              /* adjust for ADDRESS value */ \
                              CompNewAddr /* adjust for ADDRESS bus */ \
                              "%d"        /* print offset */ !

%% ComputeReturnAddr
(DATA=0x[....])               ! { dest_address = $1 << 16 } NEXTPART \
                              ComputeReturnAddr[1] !
(DATA=0x[....])               ! { dest_address = dest_address + $1 } MR \
                              CompNewAddr /* adjust for ADDRESS bus */ !


%% CompNewAddr
(dest_address = 0b<.>30_10)  ! { new_address = dest_address - 2; new_part = 1 } !
(dest_address = 0b<.>30_00)  ! { new_address = dest_address ; new_part = 0 } !
```

```
%% CheckBrAddr
(ADDRESS = new_address /* new_address computed in CompNewAddr table */ )\
                            /* branch executed */ \
                            ! { new_state = TELL } GOTO[state]\
                            GOTOPART[part] SelUnused !
(ADDRESS = 0x........)      ! /* branch not executed */ !


%% SelUnused
(state < new_state, FC2_0 = 0b110) ! UNUSED NEXTPART { state = TELL } SelUnused !
(state = new_state, TELLPART < new_part ) ! UNUSED NEXTPART SelUnused !
(state = new_state)         ! /* ready */ !
(state = 0x....)            ! NEXTPART { state = TELL} SelUnused !
```



Chapter 6

Error Messages

Warnings 6-2
Error Messages 6-3

Each error or warning message is preceded by the description file name and the line number within the file. Thus, one can easily trace back where the (syntax) error occurred.

Below the error messages and warnings of the CDISA80 program are given together with a hint on how to solve the problem.

Warnings

Warnings of the CDISA80 program:

warning 1: duplicate conditions in lines ... and ... of ...
The specified lookup table contains two entries with the same condition.

warning 2: undefined symbol(s) after command chain ignored
The character after the trailing '!' of the command chain was not a carriage return.

warning 3: [$...] is not defined in condition
A local variable was used that was not defined in the condition.

warning 4: lookup table declared but not defined: ...
The specified lookup table was defined in the header but not described in the tabular section.

warning 5: index table declared but not defined: ...
The specified index table was declared in the header but not defined in the tabular section.

warning 6: no index tables defined
Index tables were declared in the header section but none were defined in the tabular section.

warning 7: missing command chain in value table
An entry in the value table consists of a condition and a command chain. Here the command chain wasn't specified.

warning 8: '=' after '!' assumed
The logical operator 'not equal' is '!='. The '!' was not immediately followed by '='.

Error Messages

Error messages of the CDISA80 program:

fatal error: error count exceeds ...; program terminated
The number of errors is too big. You will have to solve some of the problems first and than run CDISA80 again.

error 1: cannot find input file: ...
The specified description file could not be found. Possibly, the file resides in another directory or you made a typing error.

error 2: cannot read file: ...
The template input file could not be found in the same directory as the file CDISA80.EXE. The template file is needed for making a disassembler. Re-install the custom disassembler compiler program.

error 3: cannot create output file: ...
The output file(....DIS) can not be created.

error 4: resulting table too large: ..%
The total table space is exhausted. Remove some symbol definitions or rearrange some tables.

error 5: START table multiple defined
More than one table with the name 'START' are defined. Remove or rename all but one of the 'START' tables.

error 6: local variables in START table not allowed
The usage of local variables in the START table is not allowed.

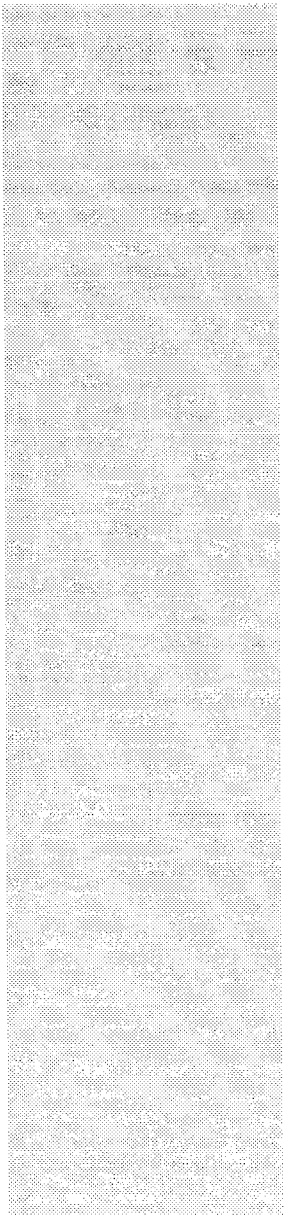
error 7: START table not first table
The START table must be the first table specified. The START table is not the first table or is not specified at all.

error 8: missing START table
At least one table named START must be used in the description.

error 9: unexpected input line: ...
An unexpected line was encountered.

error 10: missing FORMAT section
A FORMAT section is required.

error 11: not enough value table lines
An index in the value table occurs that is larger than the number of lines in the value table. Possibly a value table



command is erroneous or the value table does not have enough entries.

error 12: syntax error: illegal table line
A syntax error is present at the start of a table definition.

error 13: missing definition for label: ...
Label identifier used in alias definition never defined.

error 14: syntax error: missing '}'
At the end of a global variable command a ';' or a '}' is expected.

error 15: syntax error: missing '!' or '\' for next line
An unexpected end of the command chain is encountered. Possibly a '!' marking the end of the command chain is forgotten, or a '\' marking the continuation on the next line is missing.

error 16: syntax error: missing '[' in GOTO command
The GOTO command was not followed by an opening bracket '[' indicating the relative disassembler state.

error 18: syntax error: missing ']' in GOTO command
The command GOTO[.. was not followed by a closing bracket ']'.

error 19: undeclared table name: ...
All tables should be declared in the %%EQU-section. The table mentioned in the error message was not declared.

error 20: syntax error: missing '[' after index table name
In the command chain, the index table name was not followed by a '[' indicating the index in the table.

error 21: undefined variable or number
The offset of an index table must be a local or global variable, a constant or a number.

error 22: syntax error: missing ']' after index table name
The closing bracket ']' in the index table command is missing.

error 24: syntax error: missing '[' after value table name
The opening bracket '[' in the value table command is missing.

error 25: syntax error: missing ']' after value table name
The closing bracket ']' in the value table command is missing.

error 26: syntax error: undefined symbol after condition: '...'
The first character after a condition must be a '!' denoting the start of a command chain.

error 27: maximum number of index tables (...) exceeded
You should combine two or more index tables into one.

error 28: illegal operation
An illegal global variable instruction is detected. This is caused by:
1.The variable name is not followed by a '=' sign, or
*2.The operation is not +, -, *, /, %, &, /, >> or <<.*

error 29: unexpected operation: '...'
If the right hand side of an expression has two operands, the first must be a variable.

error 30: syntax error: missing "'"
A character constant must be enclosed by single quotes. For this constant, the closing quote is missing.

error 31: maximum number of commands (...) exceeded
Try to make as much use of nested lookup tables as possible to avoid long command chains or look up tables containing command chains that look very much alike.

error 32: maximum number of global variables (...) exceeded
Try to use less global variables and more tables to obtain a correct disassembly or try to combine the function of two small sized variables (e.g. char) into one larger variable (e.g. int or long).

error 33: maximum number of print strings (...) exceeded
Try to combine small strings into larger ones.

error 34: maximum number of global variable commands (...) exceeded
Try to use less global variables and more tables to obtain a correct disassembly or try to combine the function of two small sized variables (e.g. char) into one larger variable (e.g. int or long).

error 35: maximum number of entries in lookup tables (...) exceeded
The total number of lookup table lines may not exceed the maximum number of entries in lookup tables minus the number of lookup tables defined.

error 36: maximum number of value table lines (...) exceeded
Use index tables.

error 37: maximum number of entries in index tables (...) exceeded
The total number of index table lines may not exceed the maximum number of entries in index tables minus the number of index tables defined. Try to make use of lookup tables, and global variables.

error 38: maximum number of relational conditions (...) exceeded
Try to use more pattern conditions.

error 40: maximum number of local variable usage (...) exceeded
Try to use more tables so that local variables are not so intensely used.

error 41: maximum number of constants (...) exceeded
Use more global variable mathematics to decrease the number of constants.

error 42: maximum number of text symbols in description file (...) exceeded
Use shorter names or shorter abbreviations.

error 43: maximum number of relational condition bytes (...) exceeded
Use less or smaller relational conditions.

error 44: maximum number of local variable definitions (...) exceeded
Use shorter or less local variable definitions.

error 45: syntax error: missing condition
In a lookup table, lines consist of a condition-commands pair. The condition is mandatory. Possibly you forgot to continue the previous line so CDISA80 thinks that this line starts a new condition-commands pair.

error 47: syntax error: undefined or missing operand
A constant or variable is expected. Possibly an undefined symbol is used in the global variable command.

error 48: illegal TAB definition
The definition of the tab-settings is incorrect.

error 49: syntax error: illegal FORMAT definition: ...
String not recognized as valid keyword in the FORMAT section.

error 50: illegal variable type: ...
In the %%DEF-section, only a limited number of variable types may be used. The syntax used for the definition of the variables is the same as in the programming language C.

error 51: missing clock(s) in FORMAT section
At least one clock must be specified in the FORMAT section.

error 52: missing label(s) in FORMAT section
At least one label must be specified in the FORMAT section.

error 53: missing clock sequence(s) in FORMAT section
At least one clock sequence must be specified in the FORMAT section.

error 54: required clock not used in any clock sequence: ...
Each required clock must appear in at least one clock sequence.

error 55: illegal unsigned variable type: ...
In the %%DEF-section, only a limited number of variable types may be used. The syntax used for the definition of the variables is the same as in the programming language C.

error 56: global variable '...' multiple declared
Each global variable may only be defined once. Use unique names for global variables and constants.

error 57: syntax error: missing ';'
One of the global variables was not correctly defined. The line should end with a semi-colon (';') if global variables are defined.

error 58: value table '...' multiple declared
Only one value table can be used in the description file. Use index tables.

error 59: syntax error: illegal table type
In the EQU-section, tables are defined using LT or IT only.

error 60: syntax error: missing ':'
In the declaration of tables in the %%EQU-section, the ':' after LT, IT or VT was omitted.

error 61: table '...' multiple declared
Each table may only be declared once. Use unique names for tables.

error 62: reserved table name: '...'
The specified name may not be used as a table name. It is a reserved keyword.

error 63: maximum number of table names (...) exceeded
You should combine two or more tables into one.

error 64: maximum number of lookup tables (...) exceeded
You should combine two or more lookup tables into one.

error 65: maximum number of labels (...) exceeded
Remove or combine labels.

error 66: syntax error: missing ':'
In the declaration of labels in the %%EQU-section, the ':' after the label name was omitted.

error 67: label '...' multiple declared
The specified name already exists as the name of a different item.

error 68: syntax error: missing 'define'
In the declaration of constants in the %%DEF-section, the keyword 'define' was omitted or not correctly spelled.

error 69: constant '...' multiple declared
The specified name already exists as the name of a different item.

error 70: number too big
In the declaration of constants in the %%DEF-section, the resulting number was too big. It should have a value between -32768 and 32767.

error 71: syntax error: missing number
A syntactically incorrect number was detected.

error 72: illegal polarity
specified value not recognized as polarity

error 73: syntax error: missing ':'
In the %% FORMAT-section an expected ':' is missing.

error 74: not required label may not be used in tables: '...'
Label defined as NOT required may not be used in the condition part of a table command line

error 77: maximum number of local variables (...) exceeded
Combine local variables and use global variables or the index table to decrease the number of defined local variables in the condition.

error 78: syntax error: missing '['
In a condition, a ']' (indicating the end of a local variable definition) was encountered without a matching '['.

error 79: syntax error: missing '>'
In a condition, a '>' was expected. Possibly more than one character was enclosed in the <>-pair.

error 80: syntax error: missing bit pattern
In a condition or symbol definition, a ',' or ')' (end of condition) was encountered before a bitpattern was specified.

error 81: syntax error: illegal end of bitpattern
In a condition, a syntactically incorrect bitpattern was spec ified.

error 82: syntax error: not enough positions in bitpattern
In a pattern condition, the number of positions in a bitpattern must exactly match the number of positions in the label or variable. When for example a label consists of 5 channels, every bitpattern in a pattern condition must contain 5 bits (which means you can only use binary bitpatterns).

error 83: illegal binary digit: ...
A binary number is expected. Correct digits are '0', '1' or '.'.

error 84: illegal octal digit: ...
An octal number is expected. Correct digits are '0'..'7' '.'.

error 85: illegal hexadecimal digit: ...
A hexadecimal number is expected. Correct digits are '0'..'9', 'A'..'F' or '.'.

error 86: maximum number of channels (...) exceeded
In a label condition more channels are specified than allowed.

error 87: syntax error: too many digits in bitpattern
In a pattern condition, the number of digits in a bitpattern must exactly match the number of digits in the label or variable. When for example a label consists of 5 channels, every bitpattern in a pattern condition must contain 5 bits (which means you can only use binary bitpatterns).

error 88: syntax error: undefined or missing operand
In a condition, no or an undefined variable was specified on the left hand side of a relational condition.

error 89: illegal condition
An incorrect operator was specified in a relational condition. Allowed are =, !=, <, >, <=, and >=.

error 90: syntax error: undefined or missing operand
In a relational condition, after the operator, no or an undefined identifier was specified.

error 91: illegal condition in table line
An illegal condition in a table was encountered. Possibly an undefined label or undefined global variable name is used in the condition.

error 92: syntax error: missing end of condition
The trailing ')' in the condition was not encountered.

error 93: syntax error: illegal space in bitpattern
There is no white space allowed within a pattern value of a condition.

error 94: syntax error: missing '"'
A string was started but no terminating '"' was found.

error 95: illegal string symbol: '...'
Symbol not allowed for this string.

error 96: string too large (maximum ... characters)
Only a limited number of characters is allowed for this string.

error 97: string too small (minimum ... characters)
A certain number of characters is required for this string.

error 98: syntax error: number expected: '...'
The syntax for a number was not correctly used.

error 99: missing identifier
An identifier name is expected, but not specified.

error 100: input line too long
The line contains more than 255 characters. Use continuation lines (lines ending with '\') to shorten the length of long lines.

error 101: syntax error: illegal section header
A line started with one percent-sign not immediately followed by another.

error 102: missing command chain in: '...'
The lookup table mentioned in the error message did not contain a command chain.

error 103: syntax error: unexpected EOF

error 104: syntax error: last '{', '!', '"' or '/*' was here
An end of file was encountered while handling a comment, a string, a command chain or a set of global variable commands. Possibly a trailing end-of-comment (/), trailing end of command chain (!), end-of-global-variables-commands (}) or trailing string-quote (") was omitted.*
Error message 104 gives the line number of the corresponding 'opening mark.'

error 105: syntax error: missing '\'
The end of a line encountered while still some items expected on the line.

error 106: syntax error: missing '{'
Missing '{' in channel delay definition.

error 107: syntax error: missing ','
Missing ',' in channel delay definition.

error 108: illegal delay value (value between -2 and 2)
The specified delay value is not valid. A delay value should be in the range from -2 to 2.

error 109: syntax error: illegal channel number order
Channel numbers for a qualifier or a label must always be specified in decreasing order.

error 110: illegal channel number (0..95)
A channel number should be in the range from 0 to 95. The specified number is not in that range.

error 111: syntax error: missing '{'
Missing '{' in qualifier level definitions or in label clocks definition.

error 112: 'none' not last item in list: ...
In the label clocks definition 'none' or a list of clock identifiers are allowed. It is not allowed to have both in one label clocks definition.

error 113: 'none' not first item in list: ...
In the label clocks definition 'none' or a list of clock identifiers are allowed. It is not allowed to have both in one label clocks definition.

error 114: illegal identifier: ...
The given identifier is not defined or is not allowed.

error 115: syntax error: missing '='
Missing '=' after pods threshold.

error 116: syntax error: missing '{'
Missing '{' in the pods threshold definition.

error 117: syntax error: missing ','
Missing ',' between the threshold definitions for two pods.

error 118: syntax error: missing '}'
Missing '}' at the end of pods threshold definition.

error 119: syntax error: missing '{'
Missing '{' in the threshold definition for a pod.

error 120: syntax error: missing ','
Missing ',' between the threshold definitions for one pod.

error 121: syntax error: missing '}'
Missing '}' at the end of the threshold definition for a pod.

error 122: syntax error: illegal threshold type: ...
The specified type is not a legal threshold type. Only TTL, ECL or a variable threshold are allowed.

error 123: syntax error: illegal threshold value
Variable threshold values between -3.0 and 12.0 are allowed.

error 124: maximum number of clocks (4) exceeded
The total number of clocks you may use is 4.

error 125: identifier name already in use for label: ...
The identifier name for this clock is already in use for a label.

error 126: identifier name already in use for clock sequence: ...
The identifier name for this clock is already in use for a clock sequence.

error 127: identifier name not allowed for clock: ...
The identifier name for this clock is a reserved word and may not be used as a clock identifier name.

error 128: clock identifier already in use: ...
The identifier name for this clock is already in use for another clock.

error 129: syntax error: missing '='
Expected '=' after clock identifier name is missing.

error 130: syntax error: missing '{'
Expected '{' before clock parameter list is missing.

error 131: syntax error: missing ','
Expected ',' after clock parameter is missing.

error 132: illegal clock specifier: ...
The specifier is not recognized as a valid keyword for a clock definition.

error 133: syntax error: missing '='
Expected '=' after a clock definition keyword is missing.

error 134: clock name for '...' multiple declared
The name for a clock may only be declared once.

error 135: clock polarity for '...' multiple declared
The polarity for a clock may only be declared once.

error 136: clock timing definition for '...' multiple declared
The timing definition for a clock may only be declared once.

error 137: clock-required definition for '...' multiple declared
The required definition for a clock may only be declared once.

error 138: clock display definition for '...' multiple declared
The display definition for a clock may only be declared once.

error 139: clock edge for '...' multiple declared
The edge for a clock may only be declared once.

error 140: clock channel for '...' multiple declared
The channel number for a clock may only be declared once.

error 141: syntax error: missing ','
Missing ',' in qualifier levels definition or in label clocks definition

error 142: qualifier channel already in use for clock: ...
The specified channel number is already in use as clock channel.

error 143: clock merge-clock for '...' multiple declared
The merge-clock definition for a clock may only be declared once.

error 144: missing clock name for: ...
The name for this clock is missing. The name for a clock is required.

error 145: missing clock channel number for: ...
The channel for this clock is missing. The channel for a clock is required.

error 146: missing clock edge for: ...
The clock edge for this clock is missing. The edge for a clock is required.

error 147: syntax error: missing '}'
Expected '}' is missing at the end of a clock definition.

error 148: required clock not allowed after not required clock
The clocks which are required for the disassembler should all be declared before the clocks which aren't required.

error 149: clock name already in use: ...
The name for the clock is already in use for another clock or for a label.

error 150: syntax error: illegal clock name
An illegal name is given for this clock.

error 151: illegal clock timing definition: ...
Keyword is not recognized as a valid specifier for clock timing attribute.

error 152: illegal clock-required definition: ...
Keyword is not recognized as valid specifier for clock required attribute.

error 153: illegal clock display definition: ...
Keyword is not recognized as valid specifier for clock display attribute.

error 154: clock channel already in use: ...
The channel number is already in use for another clock or for a qualifier.

error 155: illegal clock edge: ...
Keyword is not recognized as a valid specifier for a clock edge.

error 156: syntax error: missing '{'
Expected '{' is missing in a clock qualifier declaration.

error 157: maximum number of clock qualifiers (4) exceeded
The total number of clock qualifiers is 4. Each clock uses at least one qualifier.

error 158: syntax error: missing '}'
Expected '}' is missing in a clock qualifier declaration.

error 159: illegal clock identifier: ...
An unrecognized or illegal clock identifier is given as merge-clock. Only already declared clock identifiers or 'none' are allowed.

error 160: syntax error: missing ','
Expected ',' is missing between the attributes for a qualifier.

error 161: illegal qualifier identifier: ...
Keyword is not recognized as valid specifier for a qualifier definition.

error 162: syntax error: missing '='
Expected '=' after a qualifier definition keyword is missing.

error 163: qualifier channels multiple declared
The channels for a qualifier may only be declared once.

error 164: conflicting number of channels
The number of channels given for this attribute doesn't match an already specified or implied number of channels for this qualifier.

error 165: qualifier channel delay values multiple declared
The channel delay values for a qualifier may only be declared once.

error 166: qualifier levels multiple declared
The levels for the qualifier channels may only be declared once.

error 167: qualifier-required definition multiple declared
The required definition for a qualifier may only be declared once.

error 168: missing qualifier levels
The levels for the qualifier channels are missing. The levels for the qualifier channels are required.

error 169: missing qualifier channels
The channels for the qualifier are missing. The qualifier channels are required

error 170: head width too small (minimum ... characters)
Specify larger width for header.

error 171: head width too big (maximum ... characters)
Specify smaller width for header.

error 172: required qualifier for clock not allowed after not required qualifier
The qualifiers for a clock which are required should all be declared before the qualifiers which aren't required.

error 173: illegal qualifier-required definition
Keyword is not recognized as valid specifier for qualifier required attribute.

error 174: maximum number of elements in list (...) exceeded
In the list too many elements were specified.

error 175: maximum number of clock sequences (...) exceeded
Too many clock sequences are defined.

error 176: clock sequence defined before clock
No clocks are declared yet. The clock declarations should precede the clock sequence declarations.

error 177: identifier name already in use for label: ...
The identifier name for this clock sequence is already in use for a label.

error 178: identifier name already in use for clock: ...
The identifier name for this clock sequence is already in use for a clock.

error 179: identifier name not allowed for clock sequence: ...
The identifier name is a reserved word and may not be used as a clock sequence identifier name.

error 180: identifier name already in use: ...
The identifier name for this clock sequence is already in use for a clock sequence, global variable or constant

error 181: syntax error: missing '='
Expected '=' after clock sequence identifier name is missing.

error 182: missing clock identifier names in clock sequence: ...
Clock identifier names are expected in a clock sequence declaration.

error 183: maximum number of defined labels (...) exceeded.
Remove one or combine two defined labels

error 184: label defined before clock
Clocks must be defined before labels.

error 185: identifier name already in use for clock: ...
Identifiers must have unique names.

error 186: identifier already in use for clock sequence: ...
The specified identifier is already in use for a clock sequence

error 187: identifier name not allowed for label: ...
The identifier name is a reserved word and may not be used as a label identifier name.

error 188: label identifier already in use: ...
The specified identifier is already in use for a label, a global variable or constant.

error 189: syntax error: missing '='
Missing '=' after label identifier.

error 190: syntax error: missing '{'
Missing '{' in label definition.

error 191: syntax error: missing ','
Missing ',' between two label parameters.

error 192: syntax error: illegal label specifier: ...
Wrong keyword used in label parameter specifier.

error 193: syntax error: missing '='
Missing '=' in label parameter specifier.

error 194: label name for '...' multiple declared
The name for a label may only be declared once.

error 195: label polarity for '...' multiple declared.
The polarity for a label may only be declared once.

error 196: label timing definition for '...' multiple declared.
The timing definition for a label may only be declared once.

error 197: label-required for '...' multiple declared.
The label-required for a label may only be declared once.

error 198: label display definition for '...' multiple declared.
The label display definition for a label may only be declared once.

error 199: label channels for '...' multiple declared.
The channels for a label may only be declared once.

error 200: syntax error: conflicting number of channels
The number of channels given for this attribute doesn't match an already specified or implied number of channels for this label.

error 201: label delay values for '...' multiple declared
The delay values for a label may only be declared once.

error 202: syntax error: conflicting number of channels
The number of channels given for this attribute doesn't match an already specified or implied number of channels for this label.

error 203: label clocks for '...' multiple declared
The clocks for a label may only be declared once.

error 204: label radix for '...' multiple declared
The radix for a label may only be declared once.

error 205: missing label name for: ...
Name for label must be specified.

error 206: missing label channel numbers for: ...
Label channel numbers for a label must be specified.

error 207: syntax error: missing '}'
Missing '}' in label definition.

error 208: required label not allowed after not-required label
A non-required-label definition may not appear before any required-label definition.

error 209: label name already in use
Name is already in use for a clock or a different label definition.

error 210: syntax error: illegal label name
The specified label name is not allowed.

error 211: illegal label timing definition
The specified label timing definition is not allowed.

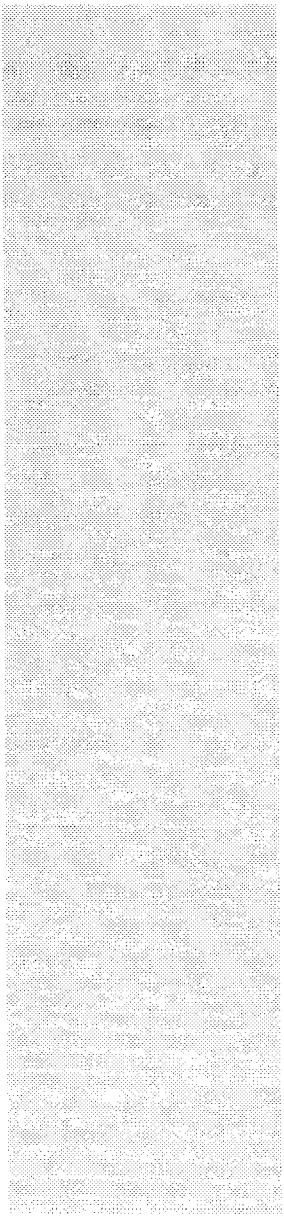
error 212: illegal label required definition
The specified label required definition is not allowed.

error 213: illegal label display definition
The specified label display definition is not allowed.

error 214: label channels missing
No channels are specified for this label.

error 215: label channel already in use for clock: '...'
Specified channel number already in use by a defined clock.

error 216: label delay values missing
No delay values are specified for this label.



error 217: clock identifiers missing
No clock identifiers are specified for this label.

error 218: illegal label radix definition
The specified label radix is not allowed.

error 219: maximum number of channels in label (...) exceeded
More channels are specified then allowed for a label.

error 220: illegal channel number
The specified number is not in the permitted channel number range.

error 221: maximum number of pattern condition expression bytes (...) exceeded.
The total space reserved for label conditions is exhausted.

error 222: maximum number of pattern conditions (...) exceeded.
The total number of allowed different condition expressions is expired.

error 223: maximum number of local variable expression bytes (...) exceeded.
The total space reserved for local variables is exhausted.

error 224: maximum number of local variables (...) exceeded.
The total number of allowed different local variable expressions is expired.

error 225: maximum number of required labels (...) exceeded.
Redefine any label, not being used in the tables as not required, otherwise reduce the number of (required) labels or combine some (required) labels into one label..

error 226: syntax error: missing ','
Missing ',' after channel number.

error 227: syntax error: missing '{'
Missing ',' between two elements in a list.

error 228: different edges for clock channel: ...
Clocks with the same clock channel must have equal edges.

error 229: different polarities for clock channel: ...
Clocks with the same clock channel must have equal polarities. If a polarity definition is omitted, a positive polarity is assumed.

error 230: different delay values for channel: ...
Delay values specified for one channel must have the same value. If the delay value definition is omitted a delay value '0' is assumed.

error 231: number too big
The specified number is not allowed.

error 232: syntax error: signal number not allowed
Un unsigned number is expected.

error 233: syntax error: illegal local variable: $...
Only local variables $1 to $9 are allowed.

error 234: syntax error: missing ']' for local variable
Expected ']' for definition of a local variable in a condition is missing

error 235: tab settings multiple declared
The tab settings definition in the FORMAT section may only be declared once

error 236: label type for '...' multiple declared
The label type definition for a label may only be declared once

error 237: syntax error: illegal label type definition: ...
The specified type is not a legal label type definition.
Only 'little_endian' or 'big_endian' or their short hand notations 'little' or 'big' are allowed.

error 238: label parts already in use for other label
The label parts definition for a label may only be used for one label. More than one label with parts are not allowed.

error 239: illegal number of parts defined for label: ...
The specified number of parts defined for a label is not allowed. Only 1,2 or 4 are allowed.

error 240: conflicting number of channels and parts for label: ...
The number of channels for a label divided by the number of parts for the label must result in an integer value.

error 241: label parts for '...' multiple declared
The label parts definition for a label may only be declared once

error 243: static char multiple declared
Only one static char variable is allowed.

error 244: syntax error: illegal tab position number order
Absolute tab positions defined with the 'tab' command must always be specified in increasing order.

error 247: blocksize multiple declared
The blocksize definition may only be declared once

error 248: illegal synchronization block size
The specified blocksize definition is not allowed. Allowed are 16, 32, 64, 128, 256, total, minimum (min) and maximum (max).

error 249: relational condition not allowed in index table
In an index table only pattern conditions are allowed to extract local variables. Relational conditions are not allowed in index tables.

error 250: illegal symbol print format: ...
The specified symbol print format in the print string command is illegal.

error 251: illegal print format: ...
The specified print format in the print string command is illegal.

error 252: symbolics definition multiple declared
The symbolics definition may only be declared once

error 253: syntax error: illegal symbolics definition: ...
The specified symbolics definition is illegal. Only 'yes' or 'no' are allowed.

error 254: label symbolic definition for '...' multiple declared
The label symbolic definition may only be declared once

error 255: syntax error: illegal label symbolic definition: ...
The specified label symbolic definition is illegal. Only 'yes' or 'no' are allowed.

error 256: reserved variable name: ...
The specified name may not be used as a global variable name or constant name It is a reserved keyword.

error 257: illegal variable symbol viewsize (1..32)
A variable label symbol viewsize should be in the range from 1 to 32. The specified number is not in that range.

error 258: label symbol viewsize for '...' multiple declared
The label symbol viewsize definition may only be declared once

error 260: syntax error: missing '{'
Missing '{' in a symbol definition for a label.

error 261: syntax error: missing ','
Missing ',' in a symbol definition for a label.

error 262: syntax error: illegal symbol name
An illegal name is given for a label symbol.

error 263: head multiple declared
The custom disassembler head definition may only be declared once

error 264: pod thresholds multiple declared
The pod thresholds definition may only be declared once

error 265: logo multiple declared
The disassembler logo definition may only be declared once

error 266: illegal symbol viewsize: ...
The specified symbol viewsize definition is illegal. Only 'unique' (uniq), 'maximum' (max) or a number in the range from 1 to 32 are allowed.

error 267: symbol first value too big: ...
The specified first value for the symbol is too big for this label.

error 268: symbol second value too big: ...
The specified second value for the symbol is too big for this label.

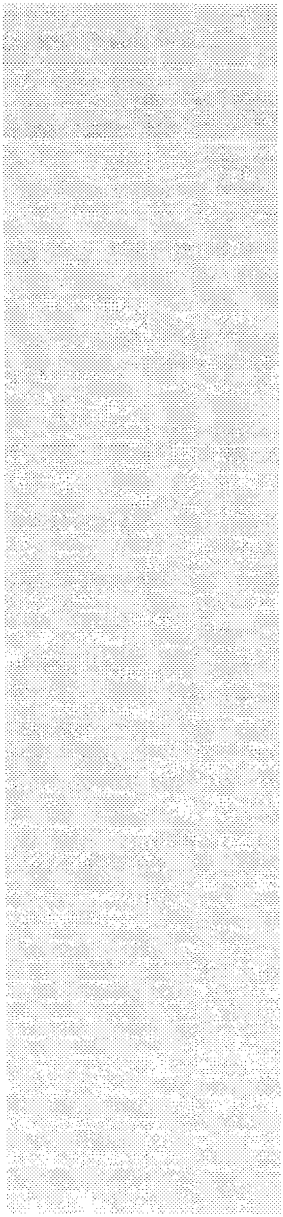
error 269: maximum number of symbols (...) exceeded
Delete some symbol definitions.

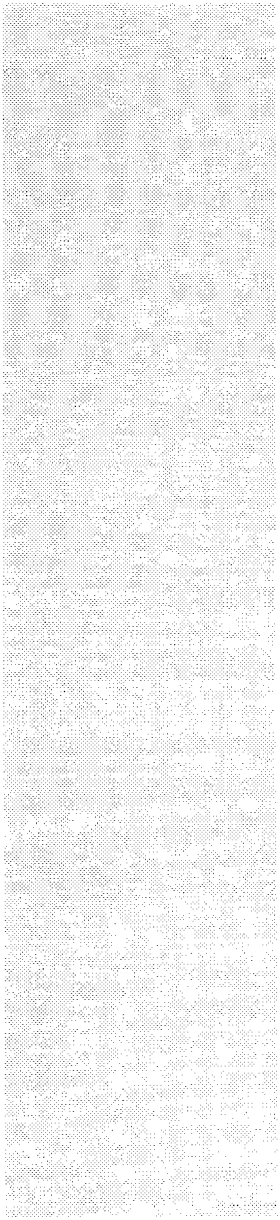
error 270: maximum number of symbol name bytes (...) exceeded
Try to use smaller symbol names.

error 271: multiple command chains in START table
The START table may only have one command chain.

error 272: illegal digit in index table pattern expression
Pattern expressions in index tables may only contain don't care digits ('.'). The pattern expression is only used to assign values to local variables.

error 273: name already in use: ...
The specified name is already used for a label identifier or clock sequence identifier.



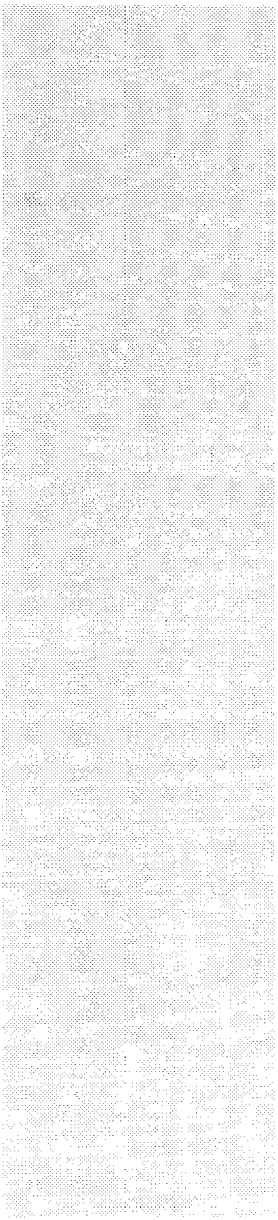


Index

\ 3-35, 3-41, 3-56, 3-57
- 3-56
\t 3-57
\n 3-57
! sign 2-5, 3-44
!= 3-41
"" 3-57
$1, $2, $3, etc. 2-8, 3-40
% 3-56
%% 3-6
  name 3-33
  table name 2-18
  DEF 2-16, 3-8
  EQU 2-16, 3-9
  FORMAT 3-11
  START 3-32
& 3-56
( ) 2-5, 3-37, 3-43
; 3-56
* 3-56
*** 2-6, 2-9, 3-61
*/ 2-16, 3-35
+ 3-56
. operator 3-38
/ 3-56
/* 2-16, 3-35
= 3-41
== 3-41
< 3-41
> 3-41
>= 3-41
>> 3-56
[2-8
] 2-8
[ ] 3-39
_ 3-38
{ } 3-56
| 3-56
` 3-39
0b 2-6, 3-38
0o 3-38
0x 2-6, 3-38
68000
  bus status 4-4
  data transfer 4-4
  disassembler 4-2
  label DSCTRL 4-3
  label FC2_0 4-4
  opcode fetch 4-4
  R/WN 4-4
68030
  label FC2_0 5-3
  label R/WN 5-3
  label SZ 5-3

A

acq_update 3-50
activating disassembler 1-7
adapter 1-2
add 3-56
address sequence break 4-11
AND-ing of conditions 3-42
arithmetic operators 3-56
ASCII-character 3-58



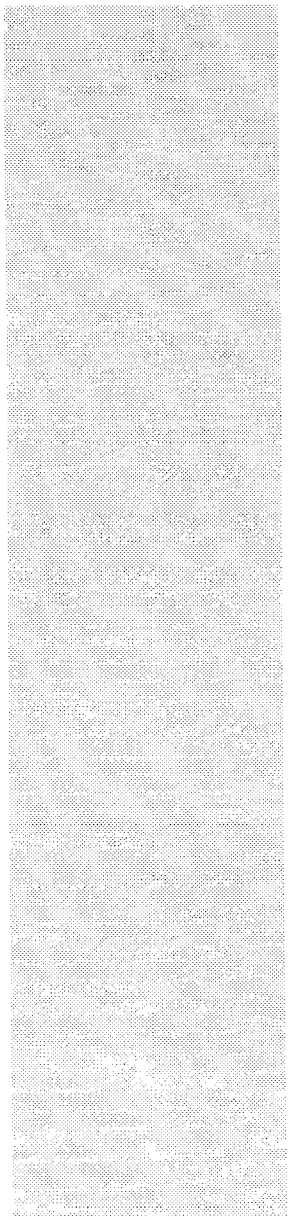
assignment structures 2-17
automatic synchronization 1-6

B

backslash 3-35, 3-57
backspace 3-60
Backus-Nauer-Format 3-3
base 3-24
big endian 3-26, 5-4
binary 3-58
  digit 3-3
bindigit 3-3, 3-39
binpattern 3-39
bitpattern 2-8, 3-38
  don't care 3-38
  repetition factor 3-38
  syntax 3-38
  underscore character 3-38
bitwise AND, OR 3-56
BLOCKSIZE 3-14
BNF 3-3
boundaries
  access table 3-61
braces 3-56
brackets 2-5, 3-37, 3-39
  nesting of 3-40
branch instruction 4-13, 5-6
bus
  status 4-4
  type 5-4

C

calls to tables 3-61
channel
  list 3-20
  number 3-18, 3-20, 3-24
  qualifier 3-20
character 3-8
  lower case 3-36
  upper case 3-36
  unsigned 3-8
clock 3-16, 3-30
  channel, definition, specifier 3-18
  definition 2-17, 3-16
  display on same line as 3-18
  display, definition, specifier 3-19
  edge 3-18
  edge definition 3-18
  id 3-16
  id list 3-25, 3-30
  merge 3-18
  merge definition, specifier 3-18
  name 3-17
  name definition 3-17, 3-26
  parameter list 3-16
  polarity, definition, specifier 3-18
  sequence 3-41
  specifier 3-16
  timing, definition, specifier 3-19
clock definition 3-16
  example 3-17
clock qualifier 3-19
  channels definition, specifier 3-20
  definition 3-19



delays definition, specifier 3-21
levels definition, specifier 3-20
parameter list 3-20
required definition, specifier 3-21
specifier 3-20
clock sequence 3-29, 3-30
condition 3-37, 3-41
definition 3-29, 3-30
id 3-30
specifier 3-30
column width 3-12
combination of conditions 3-42
command 2-5, 3-37, 3-43
chain 3-44
display selection 3-44
positioning 3-49
in relational conditions 3-51, 3-53
special 3-59
commands 2-13, 3-43
comment 2-16, 3-35
compensate microprocessor pipelines 3-47
compiler 1-4
program 1-3
compute
return address 4-9
return address on 32-bit bus 5-10
computed address 4-8
computing branch offsets 4-7
condition 3-37, 4-17
combinations 3-42
constant
definition 3-9, 4-16
size of bits 3-9
constants 3-9
in conditions 4-17
continuation character 3-35
creating a custom disassembler 1-4
current part 3-53
at entering START-table 3-52
custom disassembler 1-2
creating of 1-4
deactivating of 1-7
loading of 1-5

D

data
label value 5-10
samples 2-13
data transfer 4-4
display selection commands 3-47
not related 2-15, 3-47
unrelated 3-47
data transfers 2-14
and label parts 3-47
as part of the instruction 2-14, 4-13
deactivating disassembler 1-7
debugging 3-60
decdigit 3-3, 3-4
decimal 3-58
declaration of
constants 3-9
digits 3-3
numbers 3-3
tables 2-16

variables 2-16
declarative
part 3-6
section 3-6
decnumber 3-3
decrease
hold time 3-21, 3-24
set-up time 3-24
DEF section 2-16
default
tab, space position 3-31
value 3-13
define constant 3-9
delay 3-21
definition, specifier 3-24
qualifier 3-20
DEM68000
.DIS 4-2
.DSC 4-2
.NEW 4-2
DEM68030
.DIS 5-2
.NEW 5-2
description
file 1-4
language 3-1
detection of illegal opcodes 2-11
development process 1-4
digit repetition abbriviation 3-38
direct access tables 3-33
disassembler
activating of 1-7
compiler DIS, DSC 1-4
compiler program 1-3
deactivating of 1-7
description file 2-17, 3-3
description language 3-1, 4-1, 5-1
example microprocessor file 2-17
output column 2-5, 3-12
package 1-2
parameter 1-5
part label 3-26, 5-4
process 2-4
state 3-29, 3-49
state with 2 instructions 5-4
display
additional opcodes 3-45, 4-7
data transfers 3-47
disassembler state 3-44
menu 3-44
options 1-6, 2-13
display selection 2-13
commands 3-44
default 3-45
reset 3-55
displayed with all display options 3-45
divide 3-56
don't cares in bitpattern 3-38
DSC, DIS file 1-4

E

edge 3-18
effective clocks 3-30
elements 3-35

empty 3-4
end command chain START 3-43
EQU section 2-16, 3-9
equal to 3-41
equate
  definition 3-9
  section 3-9
erase last character 3-60
ERROR 3-60
error messages 6-3
EXAMPLE
  .DIS 2-4
  .DSC 2-4
  .ERR 2-7
  .NEW 2-3
example microprocessor
  description file 2-2
  instruction set 2-2
expression 3-56
extname 3-4

F
file
  DEM68000.DSC 4-2
  DEM68000.NEW 4-2
  DEM68030.DSC 5-2
  DEM68030.NEW 5-2
  EXAMPLE.DIS 2-4
  EXAMPLE.DSC 2-4
  EXAMPLE.ERR 2-7
  EXAMPLE.NEW 2-3
  structure 3-5
first
  disassembler state 3-49
  disassembler state is data transfer 3-47
  instruction state 3-32
  phase disassembly process 3-32
  state for next START table 3-50
folding output line 3-58
format
  command 2-10, 3-58
  specifier 2-10, 3-58
FORMAT
  menu of the logic analyzer 3-11
  section 3-11
formatting of print statements 3-57

G
general elements 3-35
global variable 2-12, 3-8
GOTO 3-49
GOTOPART 3-52, 5-4
greater than 3-41
  or equal 3-41

H
head 3-12
header 3-12
  definition, specifier 3-12
  line 3-6
header lines, sequence of 3-6
hexadecimal 3-58
hexdigit 3-3, 3-39
hexpattern 3-38

highest disassembler state number 3-55
hold time for
  label increase, decrease 3-24
  qualifier increase, decrease 3-21

I
I/O read 3-48, 4-10
I/O write 3-48, 4-10
identifier 3-16
illegal opcodes
  detection of 2-11
immediate data 4-5
increase
  set-up time 3-21, 3-24
index 3-61
  table 2-8, 3-10, 3-33
  table definition 3-10
index table 3-33
  definition 3-9
  line 3-34
installation 1-3
instr command 3-56
instruction
  blk 3-56
  length 3-49
  separation 3-56
  set of example microprocessor 2-2
instructions 3-56
int 3-8
integer unsigned 3-8
invocation disassembler compiler 1-4
IOR 3-48, 4-10
IOW 3-48, 4-10
IT 3-33

K
keyword 3-51, 3-60

L
label 3-24
  alias 3-10
  channels, definition, specifier 3-24
  clocks, definition, specifier 3-25
  definition 2-17, 3-22
  delays 3-24
  display , definition, specifier 3-26
  example 3-23
  holdtime 3-24
  id 3-22
  name, definition, specifier 3-23
  non-required 3-25
  or variable 3-38
  parameter,list 3-22
  part 3-26
  parts and data transfers 3-47
  polarity, definition, specifier 3-24
  required, definition, specifier 3-25
  specifier 3-22
  symbol 3-28
  symbol viewsize 3-27
  symbolic 3-27
  timing, definition, specifier 3-26, 3-27, 3-28
  type 3-26
leading

blanks 3-58
zeroes 3-58
less than 3-41
or equal 3-41
letter 3-4
level 3-20
list 3-20
qualifier 3-20
line 3-35
continuation 3-35
maximum length 3-35, 3-58
reshuffling 3-47
little endian 3-27, 5-4
loading a custom disassembler 1-5
local variable 2-8, 3-40
assignment 3-39
from global variables 3-39
in command chain 3-58
in index tables 2-10
transfer of value of 3-40
logo 3-11
logo definition,specifier 3-11
long 3-8
look ahead 3-50
lookup 3-61
lookup table 2-5, 2-20, 3-9, 3-33
definition 3-9
loose synchronization 4-15
lower case character 3-36
LT 3-33
lvdigit 3-61

M

manual synchronization 1-6
maximum
line length 3-35, 3-58
table nesting 3-61
measurement positioning 3-49
memory
read 2-14, 3-47, 4-10
write 2-14, 3-48, 4-10
microprocessor
adapter 1-2
pipeline 4-9
mod 3-56
MR 2-14, 3-47, 4-10
MR (unrel) 3-47
multi-line comments 3-36
multiple
conditions 3-42
instructions 3-56
multiplexed busses 3-30
multiply 3-56
MW 2-14, 3-48, 4-10

N

name 3-4
clock 3-18
index 3-61
label 3-23
list 3-10
specifier 3-17
nesting of
brackets 3-40

tables 3-61
new disa output line 3-58
NEXT 3-51
next
sample 2-5
state 3-51
NEXTPART 3-53, 5-4, 5-5
non-required label 3-25
not
equal to 3-41
executed opcodes 4-10
related data transfer 2-15, 3-47
notation Backus-Nauer 3-3
number of
global variables 3-8
index tables 3-34
lookup tables 3-34

O
octal 3-58
octdigit 3-3, 3-39
octpattern 3-39
offset in an index table 3-60
opc 3-45
opcode
detection 2-11
fetch 4-4
illegal 2-11
part of the following instruction 4-13
pipeline 4-6
operator 3-41, 3-56
opt pattern condition list 3-32
optional pattern expression 3-33
optsign 3-4
or condition 3-42, 3-56
OR-ing of conditions 3-42
other format type 3-58
output
column 2-5, 3-12
column title 3-11, 3-12
column width 3-12
output line
exceeds column width 3-57
line maximum 3-58

P
part
label and data transfers 5-10
number 3-52
pattern 3-34, 3-38, 3-43
condition 3-38
condition list 3-32
expression 3-34, 3-42
expression list 3-34, 3-43
performance 2-20
period operator 3-38
pipeline 4-9
pipelined micro-processor 4-5
pod threshold definition 3-13
polarity clock,label 3-18
positioning
in the measurement 3-49
outside the measurement 3-50
within a disassembler state 3-49, 5-4

predefined header line 3-6
PREV 3-52
previous state 3-52
PREVPART 3-54, 5-4
print
  command 2-10, 3-57
  computed address 4-8
  sample value 2-5, 2-10
  statements 3-57
  string 2-5, 3-58
  symbols 3-59
  value 2-10, 3-57
processing
  data transfers 4-13, 5-5
  speed 2-15, 2-16, 2-20
PROG 2-13, 3-45, 4-7
program
  context mode field 1-6
  samples 2-13

Q

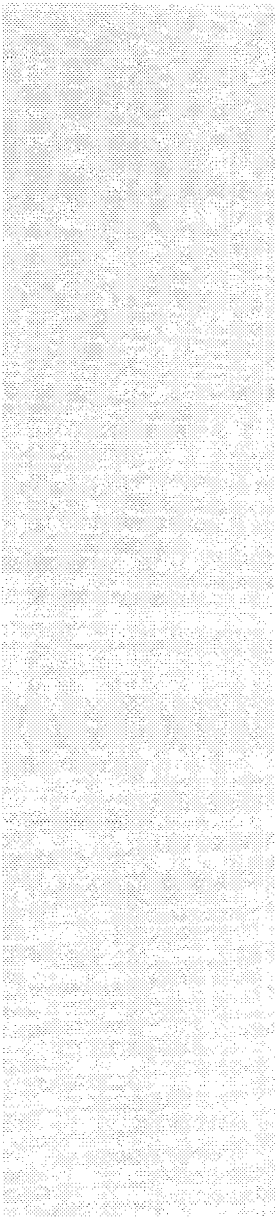
qualifier
  channels 3-20
  delays 3-20
  holdtime 3-21
  levels 3-20
  required 3-21

R

radix 3-25
RC connector 1-2
read next sample 2-5
rearrange lookup tables 2-20
relational
  condition 3-37, 3-40
  operator 3-41
relative
  branch offset 4-7
  position 3-49
remainder 3-56
repetition factor in bitpattern 3-38
report status to next instruction 4-15
required
  labels 3-25
  qualifier 3-21
reset display selection 3-55
restart 1-6
return
  address 5-10
  current part 3-53
running disassembler compiler 1-4

S

sample pointer 2-4
search following opcode 5-6
searching
  additional opcode 4-5
  data transfers 4-9
semi-colon 3-56
separation of
  instruction 3-56
  lookup tables 2-20
sequence
  break 4-11



of header lines 3-6
set-up time
for label increase, decrease 3-24
for qualifier increase, decrease 3-21
shift left/right 3-56
show data transfers 1-6, 2-14
sign extension 4-8
SKIP 3-46, 4-7, 4-13
skip local variables 3-59
spaces 3-31, 3-36
splitting up lookup tables 2-20
standard disassembler package 1-2
start
definition 3-32
section 2-17, 3-32
state 3-51
1 for next instruction 3-50
for next START 3-46
statements for print 3-57
static char 3-8
and synchronization blocksize 3-15
variable 3-8, 4-15
step delay 3-21
string 3-12, 3-57
structure for assignment 2-17
sub values in the instruction 4-9
subtract 3-56
suppress
data transfers 4-10
state from display 3-46
unused opcodes 4-10
symbol
format 3-59
viewsize 3-59
symbolic
printout 3-14
value 3-57
symbolic output
control 3-14
definition 3-14
format 3-57
synchronization
automatic 1-6
blocksize 3-14
blocksize and static char 3-15
disassembly process 3-32
status 2-9
status lost 4-15
status lost at next instruction 4-15
syntax 3-3
bitpattern 3-38

T
tab
command 3-31
position, settings, space 3-31
settings 3-31
table 3-61
access outside boundaries 3-61
boundaries 3-61
description 2-18
index (IT) 3-33
lookup (LT) 3-33
nesting 3-61

recursion 3-61
transfer control 3-61
tables scanned from top to bottom 3-33
tabular
part 3-7
section 3-33
TELL 3-51
TELLPART 3-53, 5-4, 5-6
threshold
definition 3-13
group 3-13
pod specifier 3-13
pod specifier list 3-13
specifier 3-13
value 3-13
time window 3-21, 3-24
total output line 3-58
transfer
control to other tables 3-60
value of local variable 3-40
translation options 1-6
two additional opcodes 4-7
type 2-10, 3-8
and width 3-58
type1
string 3-18
string symbol 3-12
string symbol list 3-18

U

unconditional branch 4-15
underscore 3-38
UNGET 3-55
UNPUT 3-60
unrelated data transfer 3-47
unsigned
character 3-8
integer 3-8
UNUSED 3-46, 4-7, 4-11
unused opc 3-46
upper case character 3-36
use of tables 2-20
using data transfer states 5-10

V

valid clock sequence 3-41
value 3-9, 3-41
delay 3-21
symbolic 3-57
var 3-13
declaration 3-8
variable
names 3-8
types 3-8

W

warnings 6-2
width 2-10, 3-58
and type 2-10, 3-58

Z

zero width 3-59