1. INTRODUCTION

Data compression methods vary considerably in approach and in effectiveness. No single, practical method is optimal for all types of data. It is always possible to allot more processing and memory resources to obtain asymptotically better results. In general, the available technology limits the compromises that are sensible.

Independent of any specific application area, it always can be said that data compression products are sold on the basis of cost-justification arguments. Cost-effectiveness is calculated based on the incremental cost of performing compression, and on the resulting benefit at the system level. The system-level benefit relates directly to compression performance, which usually is expressed as a compression ratio:

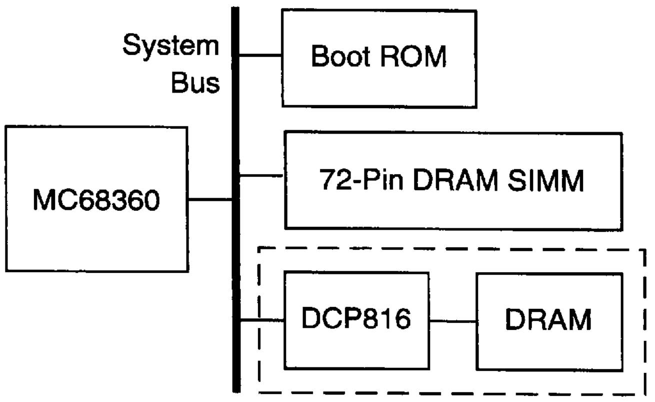
$$\text{compression ratio} = \frac{\text{plaintext size}}{\text{compressed size}}$$

DCP Research Corp. has pioneered the development of byte-string oriented data compression methods which make efficient use of high-density DRAM memory and of low-cost ASIC technology, to achieve near-optimal compression.

The DCP816 is the preferred solution for the majority of lossless data-compression applications, due in part to the broad speed-range of the part, and due as well to the low cost of the DCP and its associated DRAM module.

The DCP816 architecture incorporates a simple, rational software interface, minimizing system integration effort for both hardware and software. DMA controllers may be used with the DCP, but are not required. The glueless interfaces of Figure 21 can be applied as indicated in Figure 1.

Figure 1. Typical DCP816 Application



1.1 Notation Conventions

Within this manual, active-low signals are indicated by a bar, e.g. $\overline{\text{CS}}$. Square brackets denote bit indices or bit strings, e.g. WR3[4], D[7:0]. Bit 0 is the least significant bit. Bit indexing is used both for bus symbols and for register descriptions.

2. DCP816 FEATURES

As depicted in Figure 1, the DCP816 can be incorporated as a bus device into existing or projected, microprocessor-based products. Since the DCP is bus-oriented, it is natural to imagine systems which incorporate multiple DCPs, potentially operating in parallel. Multi-DCP systems can offer the following benefits:

- open-ended compression bandwidth
- open-ended dictionary/buffer memory configuration
- increased availability due to redundant DCPs

Examples of microprocessor-based systems that beneficially could employ one or more DCPs, include:

- telecommunication accelerators and data compressors
- modems
- statistical multiplexers
- bridges, routers, intelligent switches
- packet assembly/disassembly (PAD, FRAD) devices
- ISDN terminal adapters
- PCs and workstations

The features of the DCP include:

- high performance, real-time, lossless compression
- plaintext compression speeds of 210 Kbytes/s (40 MHz)
- compress/expand symmetry supports efficient anti-expansion encoding
- microprocessor-oriented 8-bit bus interface
- configurable handshake support for DMA controllers
- buffered or unbuffered data transfer
- parity-checked DRAM module (15-bit word + parity bit)
- 12 DRAM address lines for 32-Mbyte addressability
- configurable 8/16-bit data path to the dictionary/buffer DRAM module
- larger DRAM configurations and multi-DCP configurations facilitate multi-channel communication, e.g. multi-drop or packet-switched applications, where a separate dictionary table is required for each logical channel
- low-power operation using CMOS technology for both the DCP and its dictionary/buffer DRAM module
- fully static operation

3. DCP816 ARCHITECTURE

The DCP816 combines an application-specific, RISC architecture with an advanced Genetic Compression Algorithm. The data compression firmware is contained within an on-chip program ROM. The RISC processor includes a high-speed register set and an ALU that is specialized for compression operations.

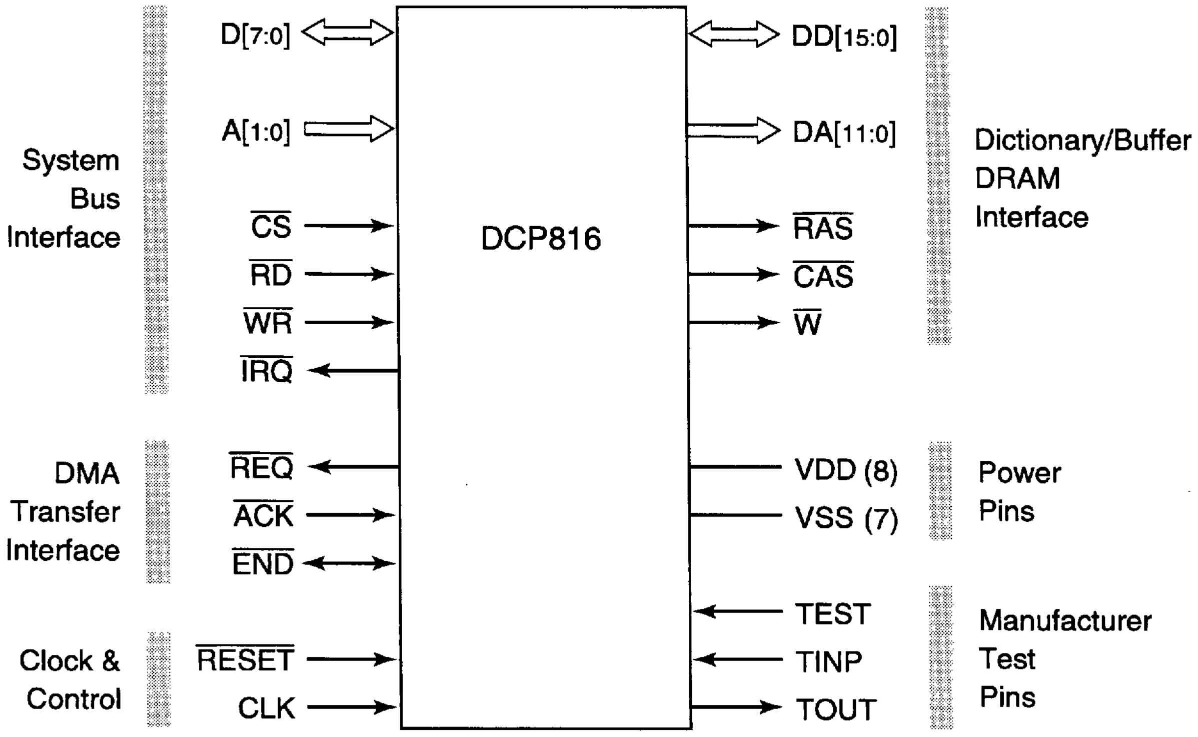
As shown in Figure 1, the DCP attaches as a peripheral device to the system bus of a microprocessor. The data path to the system bus is 8 bits wide. The system-bus interface consists of four 8-bit register ports, which are used to transfer data, commands, status, and configuration information.

Data transfers between the DCP and other devices on the system bus can be mediated optionally by a DMA controller. The DCP incorporates a DMA transfer interface with configurable handshake lines. In Figure 21 for example, the DMA controller connects directly to $\overline{\text{REQ}}$ and $\overline{\text{END}}$.

For systems that provide multiple DMA channels, the DCP can be configured for unbuffered data transfers. In unbuffered mode, the DCP overlaps plaintext transfers with compression operations to minimize DCP processing time. Cycle-steal DMA requests for plaintext bytes are generated during the course of DCP processing.

Where a single DMA channel must be shared by a number of devices, including one or more DCPs, the DCP(s) can be configured for buffered data transfer. Plaintext transfers are buffered in the DCP dictionary/buffer DRAM, permitting high-speed burst transfer and subsequent quick release of the DMA controller. In buffered mode, the DCP does not overlap plaintext transfers with compression operations.

Figure 2. DCP816 Interfaces



The dictionary tables and the buffers required by the Genetic Compression Algorithm are maintained within a DRAM module external to the DCP chip. The dictionary/buffer DRAM is local to the DCP, in the sense that the dictionary/buffer memory is not accessible from the host system bus. The DCP816 includes an integrated DRAM controller, and connects directly to the DRAM chips or SIMMs which comprise the dictionary/buffer memory.

The DCP RISC processor uses 15-bit logical words, and 16-bit physical words. The extra bit is used for parity checking. Note that the 16-bit physical word is well-adapted to the use of either nibble-wide DRAM chips or byte-wide SIMMs.

The DCP can be configured to operate with DRAM modules that are either 8 bits wide or 16 bits wide, hence the "8 16" designation. Where an 8-bit module is employed, it can be connected to either DD[15:8] or DD[7:0]. The 8-bit DRAM configuration slows the DCP to half speed.

The access time requirements (t_{RAC} in ns) for the DRAM module can be calculated from the clock frequency (f_{CLK} in Hz) of the DCP, according to the following formula:

$$t_{RAC} \le \frac{3 \times 10^9}{f_{CLK}} - 15$$

The DRAM selection formula includes provision for adequate timing margins over a wide range of f_{CLK} speeds.

DCP Research supplies portable C Language library functions to simplify software integration. The principal five functions of DCPLIB are enumerated in Figure 3.

Figure 3. Principal DCP Library Functions

DCPLIB Function	Description
`DCPresetPowerUp`	Initialize and configure the DCP
`DCPsenseDtable`	Obtain memory configuration info
`DCPinitialize`	Initialize a dictionary table
`DCPcompress`	Compress a block of text
`DCPexpand`	Inverse of `DCPcompress`

The `DCPresetPowerUp` function senses the DRAM configuration (8-bit or 16-bit data path) and prepares the DCP for operation. `DCPresetPowerUp` is performed once, after power is applied.

The `DCPsenseDtable` function senses the size of the DRAM and constructs the set of distinct dictionary indices. Usually `DCPsenseDtable` is called once, right after `DCPresetPowerUp`.

The remaining three functions are used to encode and decode a compressed channel. Each of the last three functions is parameterized by a dictionary table which is stored in the dictionary/buffer DRAM module. Speaking in terms of the Genetic Compression Algorithm, a dictionary is really a population of model organisms called *recognizers*. Successful recognizers breed new recognizers, and the fittest are favored with survival. The Genetic Compression Algorithm defines fitness as a function of compression ratio. Thus as the population adapts, compression ratio is optimized.

The DCPLIB functions are described in §4.

3.1 Signal Descriptions

TTL thresholds are employed for all input buffers on the DCP816. The output buffers are CMOS, and therefore are compatible with external 74LS or 74HCT circuitry.

As depicted in Figure 2, the signal descriptions can be divided into six functional groups:

3.1.1 Power Pins

VDD	Power. There are eight +5V power pins.
VSS	Ground. There are seven ground pins.

3.1.2 Clock and Control Pins

CLK	DCP Clock. (Input clock driver) The DCP clock controls internal operations and the timing of external data transfers.
$\overline{\text{RESET}}$	DCP Reset. (Input buffer, active low) The reset pin forces the DCP into a defined state. DCP $\overline{\text{RESET}}$ should be asserted for not less than 4 CLK periods when power is first applied.

3.1.3 System Bus Interface

D[7:0]	Data Bus. (Bidirectional buffer, active high) The data lines connect the DCP to the system bus of the host microprocessor.
A[1:0]	Address Bus. (Input buffer, active high) The address lines are decoded within the DCP to access the RRn and WRn register ports. The address lines are not used for DMA acknowledgment cycles.
$\overline{CS}$	Chip Select. (Input buffer, active low) The chip select is used to select the DCP as an I/O device, and to indicate that the address lines should be decoded.
$\overline{RD}$	Read Strobe. (Input buffer, active low) The read strobe signals a read access to the DCP. This signal is employed for both $\overline{CS}$ and $\overline{ACK}$ read cycles. Note that $\overline{RD}$ and $\overline{WR}$ should not be asserted simultaneously.
$\overline{WR}$	Write Strobe. (Input buffer, active low) The write strobe signals a write access to the DCP. This signal is employed for both $\overline{CS}$ and $\overline{ACK}$ write cycles. Note that the $\overline{RD}$ and $\overline{WR}$ strobes are mutually exclusive. Data should be stable for the duration of the $\overline{WR}$ strobe.
$\overline{IRQ}$	Interrupt Request. (Open drain output buffer, active low) The DCP can be configured to request interrupt service for 0-to-1 transitions of status bits. The $\overline{IRQ}$ line is enabled by WR3[0].

3.1.4 DMA Transfer Interface

$\overline{REQ}$	DMA Request. (3-state output buffer, active low) DMA request is used by the DCP to signal to a DMA controller that it is ready to transfer a byte of data. The $\overline{REQ}$ line is enabled by WR3[3]. $\overline{REQ}$ is a level-sensitive signal, i.e. $\overline{REQ}$ will remain asserted during a DMA acknowledgment cycle if there is more data to be transferred.
$\overline{ACK}$	DMA Acknowledge. (Input buffer, active low) The DMA acknowledge signal indicates to the DCP that a DMA cycle is in progress, and that the address lines should be ignored. During DMA cycles, only the data ports RR0 and WR0, are accessible. The $\overline{ACK}$ cycle is enabled by WR3[2]. Normally, $\overline{ACK}$ and $\overline{CS}$ will not be asserted simultaneously. If both $\overline{ACK}$ and $\overline{CS}$ are asserted and WR3[2]==1, then $\overline{CS}$ and A[1:0] are ignored.
$\overline{END}$	DMA End-of-block. (Open drain bidirectional buffer, active low) The DMA end-of-block signal can be passed from the DMA controller to the DCP, or in the reverse direction. The $\overline{END}$ line is enabled by WR3[1]. The $\overline{END}$ input can be used by the DMA controller to signal end-of-block to the DCP. $\overline{END}$ is latched during the $\overline{WR}$ strobe, along with D[7:0]. The DCP never asserts $\overline{END}$ for a $\overline{WR}$ cycle. $\overline{END}$ can be asserted by the DCP during the $\overline{RD}$ strobe. If the DMA controller asserts $\overline{END}$ during the $\overline{RD}$ strobe, the DCP will ignore the signal.

3.1.5 Dictionary/Buffer DRAM Interface

The integrated DRAM controller generates signals appropriate for use with page-mode DRAM. Virtually all DRAMs support page-mode operation, hence DRAM selection is kept simple. If f_{CLK} is the clock frequency of the DCP in Hz, the access time t_{RAC} in ns is given by:

$$t_{RAC} \le \frac{3 \times 10^9}{f_{CLK}} - 15$$

The DCP816 connects directly to CMOS DRAM. The driver circuitry in the DRAM interface incorporates slew-rate controls to avoid voltage undershoot problems. To avoid reflections, minimize the lengths of DRAM address and control wires. Total capacitive load on each address or control line should be limited to 140 pF.

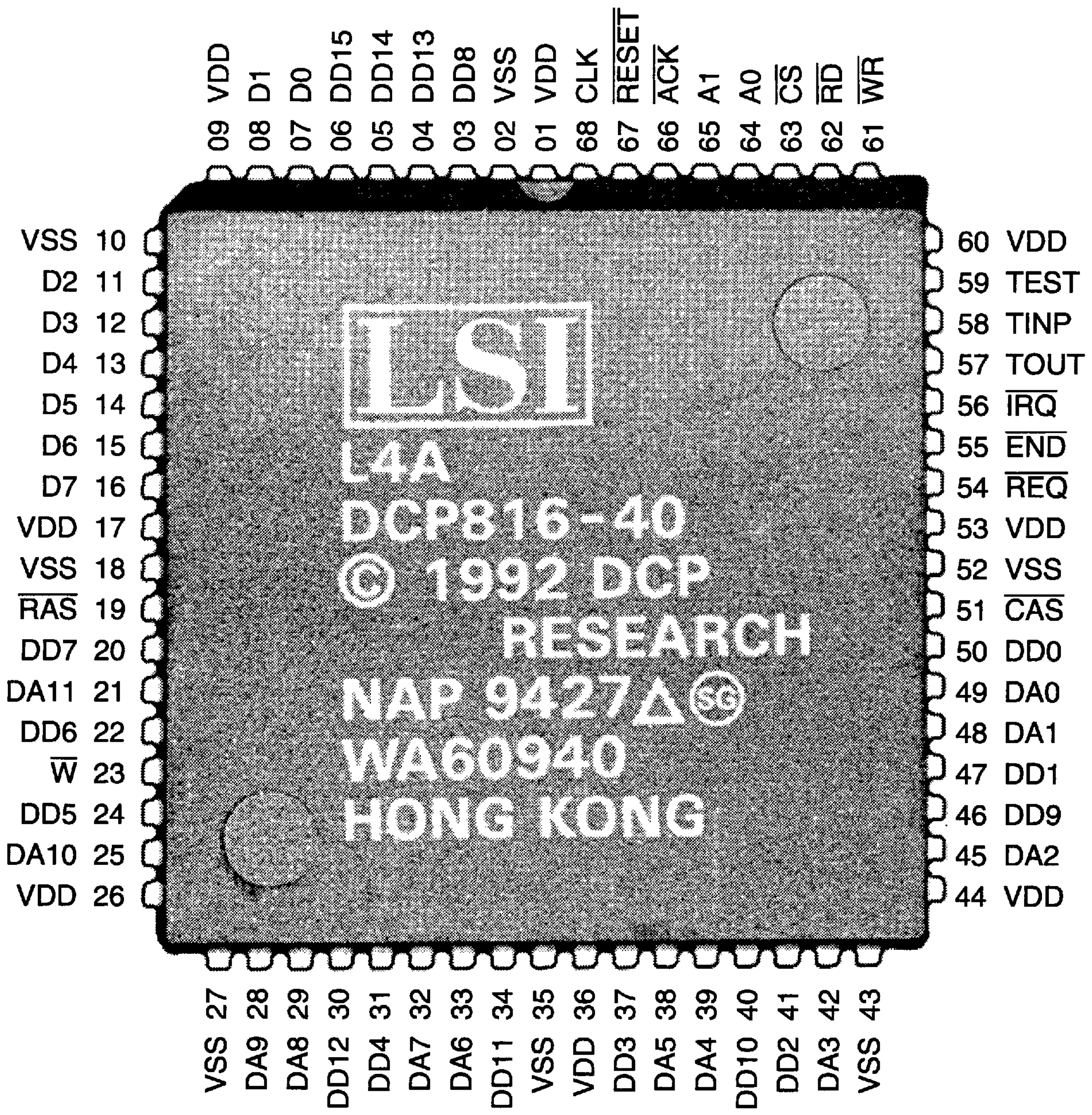
DD[15:0]	DRAM Data Bus. (Bidirectional buffer, active high) The DRAM data bus is either 16 bits wide or 8 bits wide. WR3[7:6] controls the width. For the 8-bit configuration, the DRAM data pins connect to either DD[15:8] or DD[7:0].
DA[11:0]	DRAM Address Bus. (Output buffer, active high) The DCP integrated DRAM controller multiplexes the row and column address bits on the DRAM address lines. Figure 17 describes the address multiplexer.
$\overline{\text{RAS}}$	Row Address Strobe. (Output buffer, active low) The row address strobe is produced by the DCP's integrated DRAM controller.
$\overline{\text{CAS}}$	Column Address Strobe. (Output buffer, active low) The column address strobe is produced by the DCP's integrated DRAM controller.
$\overline{\text{W}}$	Write Enable. (Output buffer, active low) The write enable signal is produced by the DCP's integrated DRAM controller.

3.1.6 Manufacturer Test Pins

These pins are reserved for manufacturing purposes. TEST and TINP should be connected to VSS. TOUT should be left open (not connected).

TEST	Test Control. (Input buffer, active high)
TINP	Test Input. (Input buffer, active high)
TOUT	Test Output. (Output buffer, active high)

Figure 4. DCP816 68-Pin PLCC Package



Integrated Circuit Topography

and/or Copyright Protected

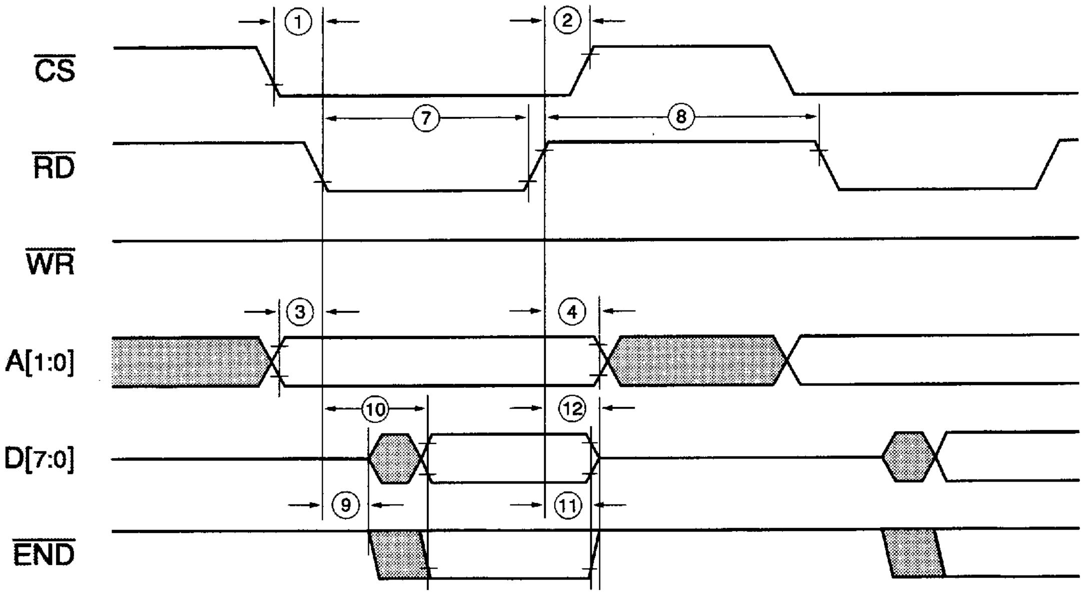
1992 by DCP Research Corp.

3.2 Signal Timing

PARAMETER		DCP816-40		
		Min	Max	Unit
	CLK period (== 1 cp)	25		ns
	CLK width high	10		ns
	CLK width low	10		ns
	$\overline{\text{RESET}}$ width asserted	4		cp
1	$\overline{\text{CS}}$ asserted to $\overline{\text{RD}}$,$\overline{\text{WR}}$ asserted	0		ns
2	$\overline{\text{RD}}$,$\overline{\text{WR}}$ negated to $\overline{\text{CS}}$ negated	0		ns
3	A[1:0] valid to $\overline{\text{RD}}$,$\overline{\text{WR}}$ asserted	5		ns
4	$\overline{\text{RD}}$,$\overline{\text{WR}}$ negated to A[1:0] invalid	5		ns
5	$\overline{\text{ACK}}$ asserted to $\overline{\text{RD}}$,$\overline{\text{WR}}$ asserted	0		ns
6	$\overline{\text{RD}}$,$\overline{\text{WR}}$ negated to $\overline{\text{ACK}}$ negated	0		ns
7	$\overline{\text{RD}}$,$\overline{\text{WR}}$ width asserted	3.5		cp
8	$\overline{\text{RD}}$,$\overline{\text{WR}}$ width negated	1.5		cp
9	$\overline{\text{RD}}$ asserted to D[7:0],$\overline{\text{END}}$ on	0		ns
10	$\overline{\text{RD}}$ asserted to D[7:0],$\overline{\text{END}}$ valid		3.5	cp
11	$\overline{\text{RD}}$ negated to D[7:0],$\overline{\text{END}}$ invalid	0		ns
12	$\overline{\text{RD}}$ negated to D[7:0],$\overline{\text{END}}$ off		20	ns
13	D[7:0],$\overline{\text{END}}$ valid to $\overline{\text{WR}}$ asserted	5		ns
14	$\overline{\text{WR}}$ negated to D[7:0],$\overline{\text{END}}$ invalid	5		ns
	WR0 data port only:			
15	$\overline{\text{WR}}$ width asserted	1.5		cp
16	$\overline{\text{WR}}$ width negated	1.5		cp
	DRAM selection criterion:			
17	$\overline{\text{RAS}}$ asserted to DD[15:0] valid		t_{RAC}	ns
	Initialize operation: total processing time (double for 8-bit DRAM path)		1000000	cp
	Compress operation: average processing time per byte (double for 8-bit DRAM path)		185	cp
	Expand operation: average processing time per byte (double for 8-bit DRAM path)		175	cp

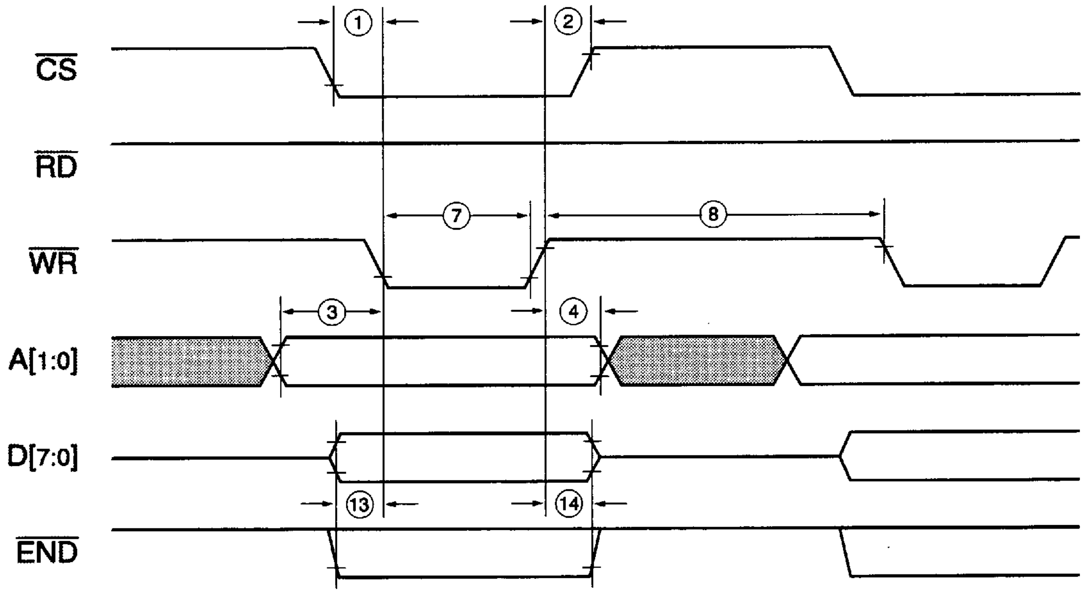
9002890 0000013 175

Figure 5. Register Read Cycle



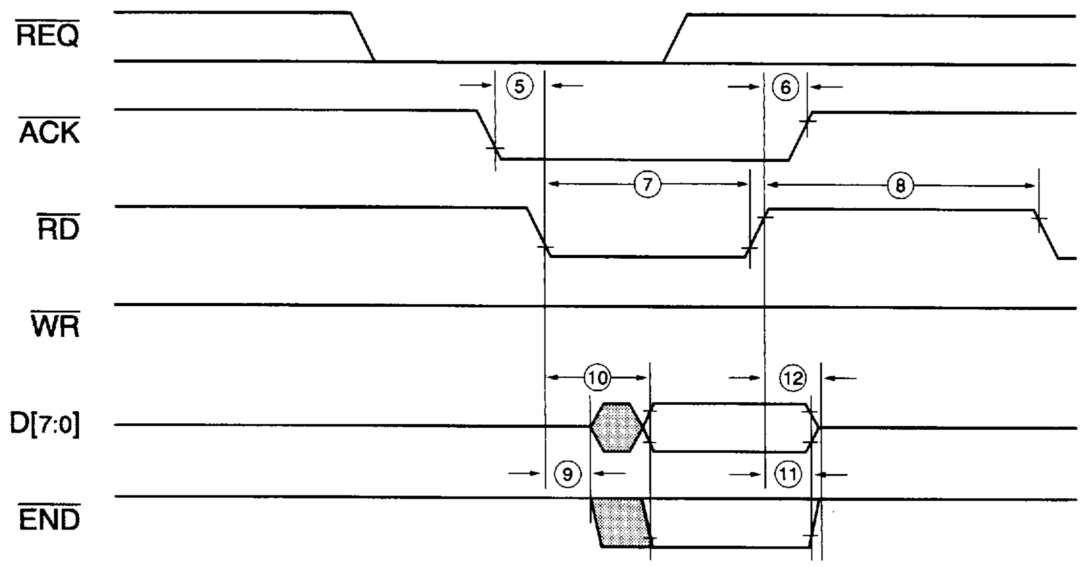
Notes: 1. If WR3[2]==1, $\overline{\text{ACK}}$ must remain negated
2. $\overline{\text{CS}}$ and A[1:0] are stable during $\overline{\text{RD}}$ pulse
3. If WR3[1]==1, the DCP may assert $\overline{\text{END}}$

Figure 6. Register Write Cycle



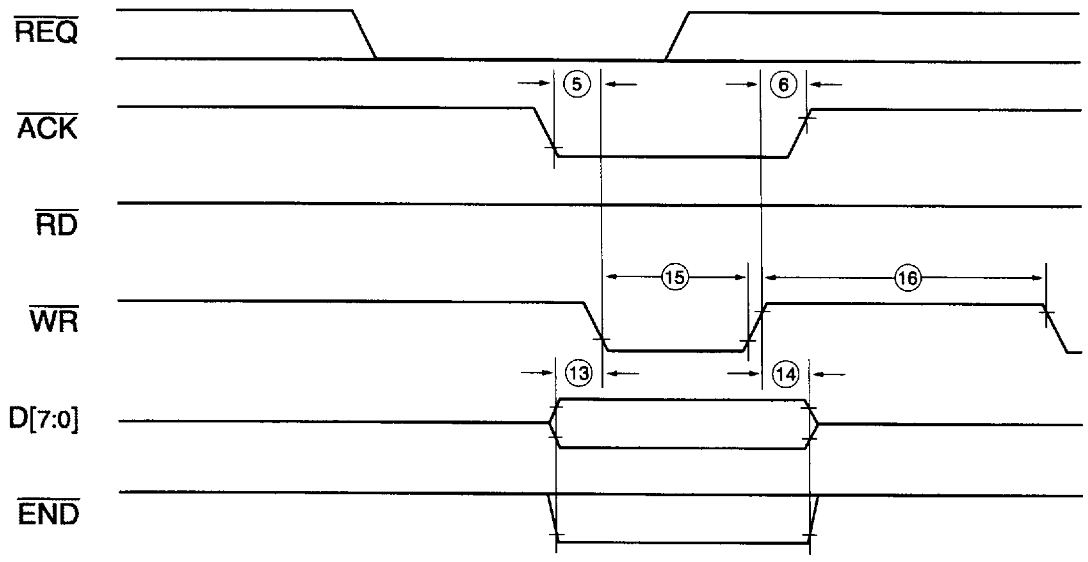
Notes: 1. If WR3[2]==1, $\overline{\text{ACK}}$ must remain negated
2. $\overline{\text{END}}$ may be asserted by the bus master
3. $\overline{\text{CS}}$, A[1:0], D[7:0], and $\overline{\text{END}}$ are stable during $\overline{\text{WR}}$ pulse
4. The DCP samples D[7:0] and $\overline{\text{END}}$ during the $\overline{\text{WR}}$ pulse

Figure 7. DMA Read Cycle



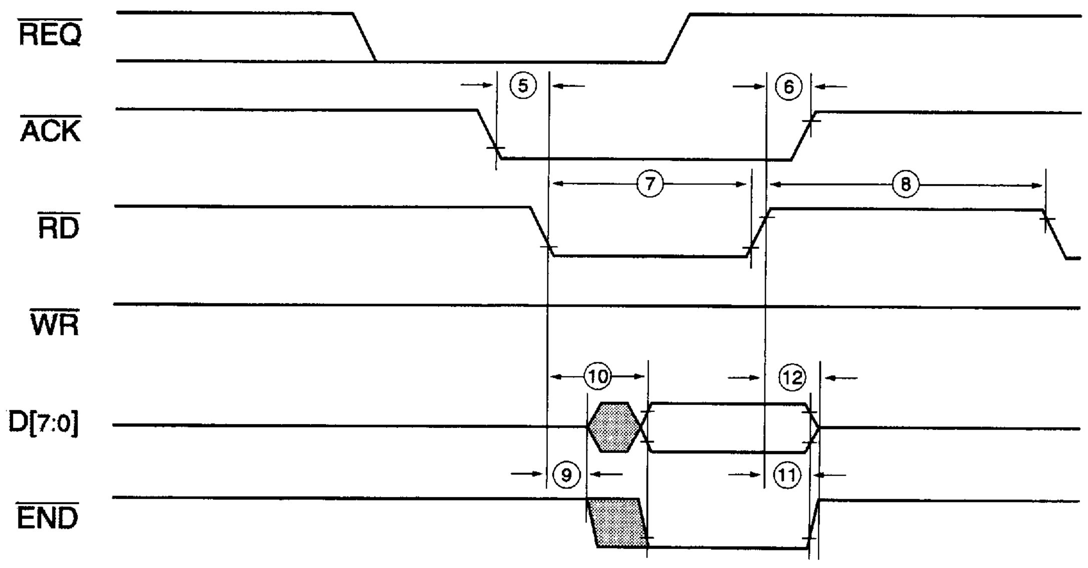
Notes: 1. This example assumes WR3[3:2]==11_2
2. If WR3[1]==1, the DCP may assert $\overline{END}$
3. The DMA read cycle accesses RR0

Figure 8. DMA Write Cycle



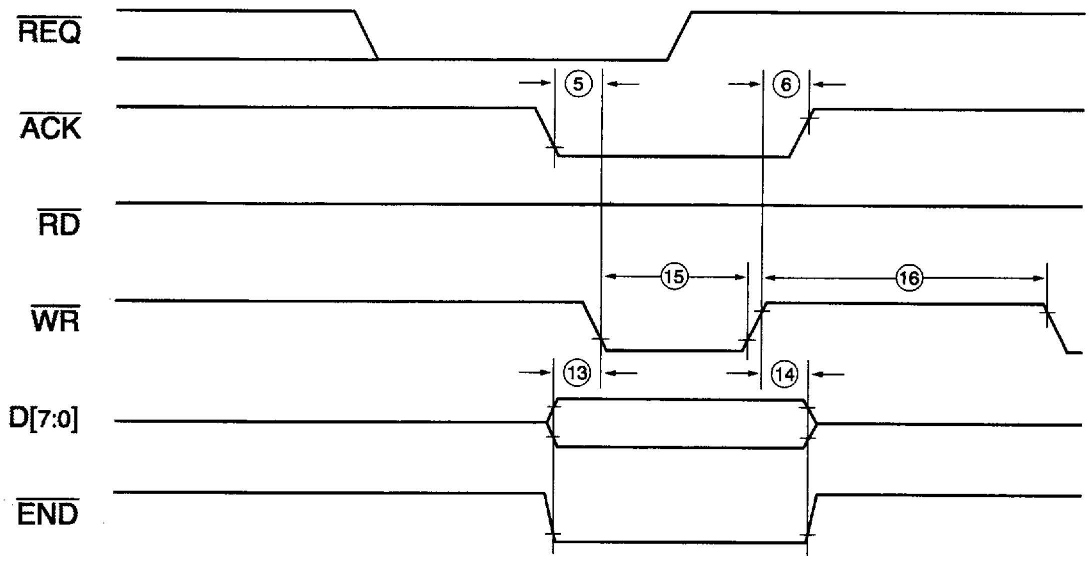
Notes: 1. This example assumes WR3[3:2]==11_2
2. $\overline{END}$ may be asserted by the bus master
3. The DCP samples D[7:0] and $\overline{END}$ during the $\overline{WR}$ pulse
4. The DMA write cycle accesses WR0

Figure 9. DMA Read Cycle (end-of-block)



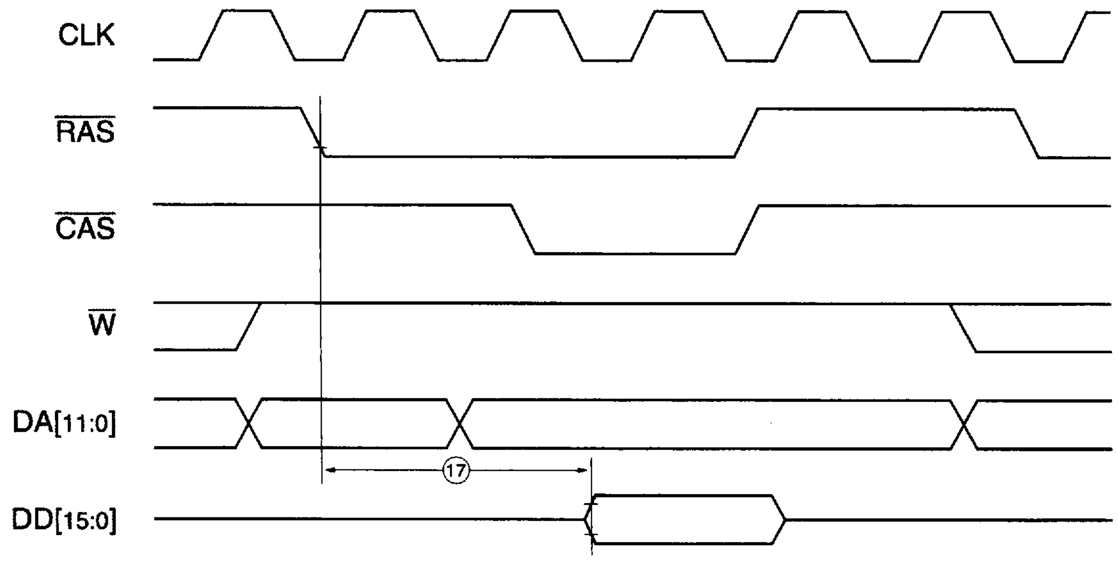
Notes: 1. This example assumes WR3[3:1]==111_2
2. $\overline{END}$ is asserted by the DCP
3. The bus master should discard the accompanying data byte

Figure 10. DMA Write Cycle (end-of-block)



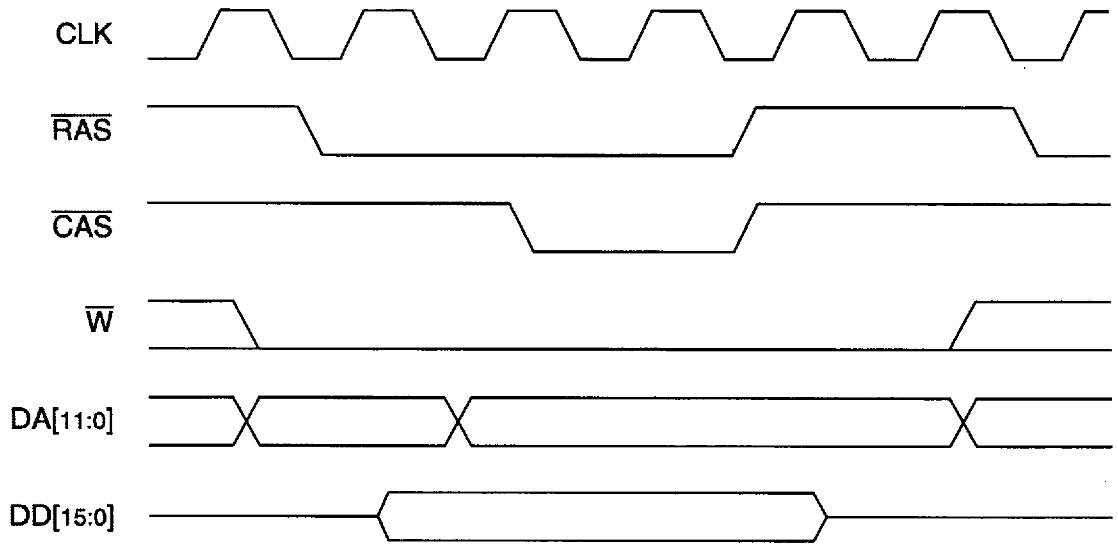
Notes: 1. This example assumes WR3[3:1]==111_2
2. $\overline{END}$ is asserted by the bus master
3. The DCP discards the accompanying data byte

Figure 11. DRAM Read Cycle



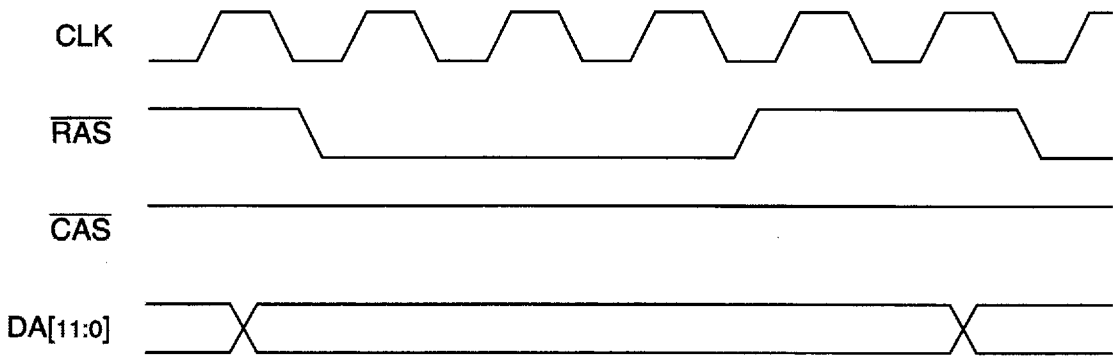
Notes: 1. Parameter 17 is the DRAM selection criterion, i.e. the access time
2. Timing for $\overline{RAS}$, $\overline{CAS}$, $\overline{W}$, and DA[11:0] is derived from CLK
3. The integrated DRAM controller uses both rising and falling edges of CLK

Figure 12. DRAM Write Cycle



Notes: 1. Timing for $\overline{RAS}$, $\overline{CAS}$, $\overline{W}$, DA[11:0], and DD[15:0] is derived from CLK
2. The integrated DRAM controller uses both rising and falling edges of CLK

Figure 13. DRAM Refresh Cycle



Notes: 1. The DCP816 performs $\overline{\text{RAS}}$-only refresh cycles
2. Timing for $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and DA[11:0] is derived from CLK
3. The integrated DRAM controller uses both rising and falling edges of CLK

3.3 System Bus Interface

The DCP816 is intended for use within systems based upon standard, commodity microprocessors. The interface to the host system bus is simple and versatile.

The DCP can be either memory-mapped or I/O-mapped, depending upon the norm for the host microprocessor. Two address lines A[1:0], are decoded within the DCP. As Figure 14 depicts, four 8-bit register ports are used to transfer data, commands, status and configuration information between the host system bus and the DCP:

Figure 14. DCP Register Port Summary

A[1:0]	Read Reg	Read Op	Write Reg	Write Op
00	RR0	Data	WR0	Data
01	RR1	Status	WR1	Divisor
10	RR2	Eob	WR2	Command
11	RR3	Config	WR3	Config

Some DMA mechanisms will not provide address bits to the DCP during DMA cycles, typically because the address bus is being used to address memory during the same cycle. The DCP can be configured to ignore A[1:0] during bus cycles in which the $\overline{\text{ACK}}$ signal is asserted. $\overline{\text{ACK}}$ cycles invariably are directed to the data port registers, RR0 and WR0.

Examples of typical DCP transactions follow. Since the DCP is a block-oriented device, it is natural to use a DMA controller to mediate block transfers to and from the data port. Note that DMA is not required, as indicated below by parenthesized instances of the acronym (DMA).

Initialization example:

1. write an Initialize command to WR2
2. read RR2 to signal end-of-block
3. read RR1 to check for completion and for errors

Compression example:

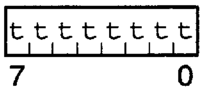
1. write a Compress command to WR2
2. (DMA) transfer a plaintext block to WR0
3. read RR2 to signal end-of-block
4. (DMA) transfer the compressed block from RR0
5. read RR1 to check for errors

Expansion example:

1. write an Expand command to WR2
2. (DMA) transfer a compressed block to WR0
3. read RR2 to signal end-of-block
4. (DMA) transfer the plaintext block from RR0
5. read RR1 to check for errors

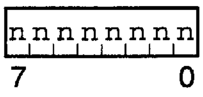
Figure 15. Write Port Summary

Data Port:

WR0


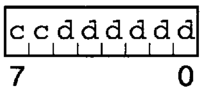
tttttttt = input-text byte

Divisor Port:

WR1


nnnnnnnn = refresh divisor

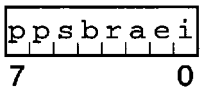
Command Port:

WR2


cc = operation code
dddddd = dictionary index

cc	Operation
00	Compress
01	Expand
10	Initialize
11	Reserved

Configuration Port:

WR3


pp = data path configuration
s = start-of-block interrupt
b = buffered mode
r = enable $\overline{\text{REQ}}$ line
a = enable $\overline{\text{ACK}}$ cycles
e = enable $\overline{\text{END}}$ cycles
i = enable $\overline{\text{IRQ}}$ line

pp	Data Path	Width
00	Disabled	–
01	DD[7:0]	8
10	DD[15:8]	8
11	DD[15:0]	16

Figure 16. Read Port Summary

Data Port:

```
RR0  |t t t t t t t t|      tttttttt = output-text byte
      7             0
```

Status Port:

```
RR1  |r e d m f p t v|
      7             0
```

r	= transfer request
e	= end-of-operation
d	= transfer direction
m	= memory refresh status
f	= firmware check
p	= parity check
t	= format check
v	= overrun/underrun check

Reading RR1 has the side-effect of signaling an interrupt-acknowledgement to the DCP. Note that clearing WR3[0] is an alternate method of acknowledging an interrupt request.

Eob Port:

```
RR2  |c c d d d d d d|      ccdddddd = WR2 image
      7             0
```

Reading RR2 has the side-effect of signaling an end-of-block to the DCP. Note that end-of-block also can be signaled by writing to WR0 with $\overline{\text{END}}$ asserted, provided that WR3[1]==1.

Configuration Port:

```
RR3  |p p s b r a e i|      ppsbraei = WR3 image
      7             0
```

Because the configuration port is read/write, bit-set and bit-clear operations can be used to alter the configuration.

3.4 Write Ports

Figure 15 summarizes the functions of the write ports. Write cycles are depicted in Figures 6, 8 and 10.

3.4.1 WR0 Data Port

WR0 is used in connection with Compress and Expand commands. For the Compress command, the plaintext block is written to WR0, byte by byte. For the Expand command, the compressed block is written to WR0, byte by byte.

The DCP indicates that it is ready to accept data via WR0 in two ways:

- the status port transfer-request bit RR1[7]==1
- if enabled by WR3[3]==1, the $\overline{\text{REQ}}$ line is asserted

If data is written to WR0 prior to the transfer-request indication, the DCP will set the overrun check bit RR1[0].

Following the transmission of a block, end-of-block can be indicated to the DCP in either of two ways: by a read from RR2 or by the $\overline{\text{END}}$ cycle. If WR3[1]==1, a DMA controller can assert $\overline{\text{END}}$ during an extra WR0 cycle. The byte that is transmitted during the $\overline{\text{END}}$ cycle is not part of the block, and is discarded by the DCP.

3.4.2 WR1 Refresh Port

WR1 controls the refresh rate for the dictionary/buffer DRAM module. The refresh timing is derived from the DCP CLK frequency, prescaled by a factor of 8.

The refresh divisor n is given by the following formula, where f_{CLK} is the DCP clock frequency and f_{REF} is the required refresh frequency:

$$n = \frac{f_{CLK} \div 8}{f_{REF}}$$

WR1 is encoded as a binary integer n – 1. For example, assume that a 40 MHz DCP is coupled to a DRAM module requiring 512 refresh cycles every 8 ms (125 Hz).

$$f_{CLK} = 40 \times 10^6$$
$$f_{REF} = 125 \times 512$$
$$n = 78.125$$

It follows that WR1 should be assigned 01001101_2 the binary equivalent of 78 – 1.

If WR3[7:6]==00_2 the integrated DRAM controller is disabled. $\overline{\text{RAS}}$ remains inactive whenever the DRAM controller is disabled. Refresh cycles are performed when the DRAM controller is enabled.

3.4.3 WR2 Command Port

Commands are issued to the DCP via WR2. The operation (Initialize, Compress, Expand, or Reserved) is encoded in WR2[7:6], while the dictionary index is encoded in WR2[5:0].

```
WR2  |c c d d d d d d|        cc     = operation code
      7             0         dddddd = dictionary index
```

Note that WR2[7:6]==11_2 is reserved. If the Reserved operation code is issued, the DCP will set the firmware check bit RR1[3].

The format of the dictionary index field depends upon the dictionary/buffer DRAM configuration. For all but the largest memory configurations, some bits of WR2[5:0] will be unused.

Figure 17 depicts the mapping from the dictionary index field WR2[5:0] to the multiplexed DRAM address DA[11:9]. When the memory is configured for an 8-bit data path, WR2[0] is not used. Other unused WR2[5:0] bits can be inferred from unused bits of DA[11:9], using Figure 17.

Figure 17. Multiplexer Mapping from WR2[5:0] to DA[11:9]

	WR2[...]	
DA[...]	row	column
9	1	0
10	3	2
11	5	4

Notes: 1. The address multiplexer maps either a row or a column address to DA[11:9]
2. Odd-numbered WR2 bits form the row address, even bits form the column address
3. Figures 11 and 12 depict $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and address multiplexer timing

Normally commands are issued while the DCP is idle, i.e. there is no active operation. However, the DCP is always ready and willing to accept a new command. When a command is issued while the DCP is busy, the active operation is terminated and the new operation is initiated.

The DCP indicates the completion of an operation by setting the end-of-operation bit RR1[6]. Once an operation is completed, the DCP idles, waiting for the next command.

When a command is issued via WR2, the status register RR1 and any pending interrupt conditions are cleared. The check bits of RR1 therefore describe conditions that have arisen during the course of the most recent DCP operation.

The dictionary tables required by the Compress and Expand operations must be validated by an Initialize command prior to use. The result of a Compress or Expand command that indexes an invalid dictionary table, is undefined. Note that undefined DCP behavior may include hung firmware.

Whenever an operation is terminated, or whenever any of the check bits RR1[3:0] are set, the indexed dictionary table is no longer valid.

3.4.4 WR3 Configuration Port

WR3[7:6] configures the data path to the dictionary/buffer DRAM module. WR3[7]==1 enables DD[15:8], while WR3[6]==1 enables DD[7:0]. When both bits are 1, the data path is 16 bits wide. When both bits are 0, the DCP is disabled. Otherwise, the data path is 8 bits wide. Changes to WR3[7:6] invalidate the entire contents of the dictionary/buffer DRAM.

Within a Compress or Expand operation, there can be a substantial delay between the last write to WR0 and the first read from RR0. If WR3[5]==1, the DCP will signal a start-of-block interrupt as soon as RR0 data is ready for transfer. The interrupt handler should check the status port RR1 for errors, and then initiate the block transfer from RR0.

WR3[4] controls plaintext buffering. If WR3[4]==0, plaintext transfers are overlapped with Compress and Expand operations. Otherwise WR3[4]==1, and plaintext transfers are buffered in the dictionary/buffer DRAM. Changes to WR3[4] become effective for subsequent DCP commands.

The $\overline{\text{REQ}}$ line is enabled by WR3[3]==1. When WR3[3]==0, the $\overline{\text{REQ}}$ driver is 3-state off.

The $\overline{\text{ACK}}$ cycle is enabled by WR3[2]==1. When WR3[2]==0, the DCP ignores the $\overline{\text{ACK}}$ input.

The $\overline{\text{END}}$ cycle is enabled by WR3[1]==1. When WR3[1]==0, the DCP ignores the $\overline{\text{END}}$ input, and the $\overline{\text{END}}$ open-drain driver is negated.

The $\overline{\text{IRQ}}$ line is enabled by WR3[0]==1. If WR3[0]==0, the DCP clears any pending interrupt conditions, and the $\overline{\text{IRQ}}$ open-drain driver is negated. Note that reading RR1 is an alternate method of acknowledging interrupts. When the DCP signals an interrupt request, it is asking that the status port RR1 be examined.

The conditions that can generate interrupts are 0-to-1 transitions for any of the bits RR1[3:0] or RR1[6]. When WR3[5]==1, the first 0-to-1 transition of RR1[7] following an end-of-block indication also will generate an interrupt. Interrupt conditions are ignored if WR3[0]==0.

While the DCP is busy, the result of a change to WR3[7:4] is undefined. Note that undefined DCP behavior may include hung firmware. A more direct way to state the rule might be: always call `DCPterminateOperation` before modifying the most significant nibble of WR3. `DCPterminateOperation` is a DCPLIB ancillary function (§4.1.6), and is the simplest way to force the DCP into the idle state.

Changes to WR3[7:4] are safe anytime the DCP is idle. Modification of WR3[3:0] is safe at any time, whether or not the DCP is busy.

```
WR3  |p p s b r a e i|
      7             0
```

pp	= data path configuration
s	= start-of-block interrupt
b	= buffered mode
r	= enable $\overline{\text{REQ}}$ line
a	= enable $\overline{\text{ACK}}$ cycles
e	= enable $\overline{\text{END}}$ cycles
i	= enable $\overline{\text{IRQ}}$ line

3.5 Read Ports

Figure 16 summarizes the functions of the read ports. Read cycles are depicted in Figures 5, 7 and 9.

3.5.1 RR0 Data Port

RR0 is used in connection with Compress and Expand commands. For the Compress command, the compressed block is read from RR0, byte by byte. For the Expand command, the plaintext block is read from RR0, byte by byte.

The DCP indicates that it is ready to transmit data via RR0 in two ways:

- the status port transfer-request bit RR1[7]==1
- if enabled by WR3[3]==1, the $\overline{\text{REQ}}$ line is asserted

If the RR0 data port is read prior to the transfer-request indication, the DCP will set the underrun check bit RR1[0].

Following the transmission of a block, RR1[6] is set, indicating end-of-operation. If WR3[1]==1, the DCP will assert $\overline{\text{END}}$ for a subsequent RR0 cycle. The byte that is transmitted during the $\overline{\text{END}}$ cycle is not part of the block, and should be discarded.

3.5.2 RR1 Status Port

Reading RR1 clears all pending DCP interrupt conditions. The DCP requests interrupt service for 0-to-1 transitions of RR1 status bits, therefore it is sensible that reading RR1 has the side-effect of signaling interrupt acknowledgment to the DCP.

RR1[7] indicates a transfer request. During the course of either a Compress or Expand operation, a block of text is transferred to the DCP via WR0, and then the result block is transferred from the DCP via RR0. Following the final WR0 cycle, end-of-block is indicated either by an RR2 cycle or by an $\overline{\text{END}}$ cycle. Because there is never ambiguity as to the direction of data transfer, RR1[7] is used for both directions.

RR1[6] indicates end-of-operation and end-of-transfer. The DCP sets RR1[6] following the last data byte read from RR0.

The DCP uses RR1[5] to indicate the direction of data transfer. If RR1[5]==0, the DCP expects transfers to WR0. When RR1[5]==1, the DCP expects transfers from RR0.

```
RR1  |r|e|d|m|f|p|t|v|
      7             0
```

r	= transfer request
e	= end-of-operation
d	= transfer direction
m	= memory refresh status
f	= firmware check
p	= parity check
t	= format check
v	= overrun/underrun check

The DRAM refresh controller includes a 12-bit row address counter. The most significant bit of the row address counter is observable as RR1[4]. Each refresh cycle increments the 12-bit counter.

The DCP check bits RR1[3:0] are associated with four categories of failure:

- RR1[3] firmware check
- RR1[2] DRAM parity check
- RR1[1] compressed-text format check
- RR1[0] overrun/underrun check

The firmware check bit RR1[3] is set for any of the following conditions:

- reserved operation code
- buffer overflow
- invalid dictionary table
- invalid compressed-text

The parity check bit RR1[2] is set when a DRAM failure is detected. Any attempt to use an uninitialized dictionary table also can cause a parity check indication.

Compressed-text that is written to WR0 is checked by the interface hardware for well-formedness. The format check bit RR1[1] is set when the DCP encounters a compressed-text pattern that could never have been produced by a properly functioning DCP. The most likely cause of a format check is corrupted compressed-text.

Data transfers to WR0 and from RR0 generally require flow control. If either WR0 or RR0 is referenced when there is no transfer request outstanding, the overrun/underrun check bit RR1[0] is set. The most likely cause of an overrun or underrun is software failure.

3.5.3 RR2 Eob Port

The WR2 command can be read back via RR2, however the principal use of RR2 is to signal end-of-block in lieu of the DMA $\overline{\text{END}}$ indication. In response to the end-of-block signal, the DCP will set the transfer-direction bit RR1[5].

For the Compress and Expand commands, the end-of-block signal is required following the transfer of a text block to WR0. The Initialize and Reserved commands should be followed by the end-of-block signal, even though no data is transferred to WR0 for those commands.

The command image is useful for tests of the system data bus. Data patterns written to WR2 can be checked via RR2. Whenever such testing is performed, the dictionary/buffer DRAM is invalidated and subsequently must be initialized.

3.5.4 RR3 Configuration Port

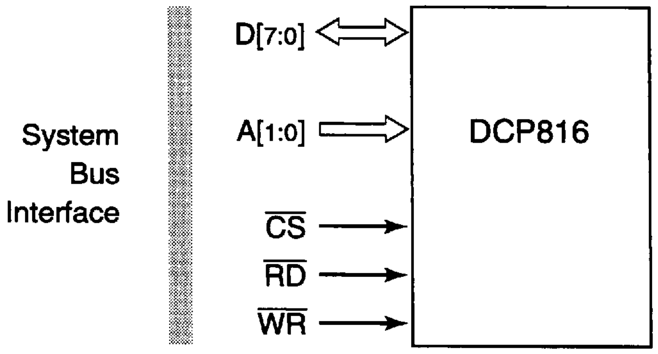
The WR3 configuration can be read back via RR3. Therefore where the DCP register ports are memory-mapped, bit-set and bit-clear instructions can be employed to adjust the various enable bits.

4. PROGRAMMING THE DCP

§3 introduced the DCP816 as a novel peripheral device, suitable for use within microprocessor-based systems. Software engineers imagine microperipherals in terms of their control and status registers. Peripherals such as UARTs, PIAs, and DCPs share similar device images and are programmed using similar techniques.

The complexity of integrating a microperipheral can be estimated from the number of signal pins, together with the number of register locations. With 4 locations and 13 pins visible to the host system bus, the DCP is among the most tractable of peripherals.

Figure 18. DCP816 Minimal Device Image



Borland C for the IBM PC affords a simple and elegant means to access peripheral devices: the `inportb` and `outportb` macros. Low-level operations on the DCP can be expressed within the C Language, without sacrificing execution efficiency.

The IBM PC is based on the Intel '86 architecture, which places most peripherals in an I/O space that is separate from the memory space. The Borland C versions of `inportb` and `outportb` are expanded in-line into '86 `IN` and `OUT` instructions.

Since `inportb` and `outportb` are macros, they can be defined for architectures that provide a single address space to memory and devices. For example, the following definitions would be appropriate for the Motorola 68K architecture:

```
#define inportb(port)              (*(char *)(port))
#define outportb(port,data)        (*(char *)(port)=(data))
```

The C Language codes given in the remainder of this Section will define several useful higher-level functions. The form of the definitions is such that they can be ported to many different architectures.

For some architectures it will be advantageous to employ more sophisticated DCP features, such as the DMA Transfer Interface and the $\overline{\text{IRQ}}$ line. The elementary C library is intended to guide the software engineers who ultimately must deal with architecture-dependent issues.

The elementary C library provides access to a single DCP, with up to 64 compression channels. Concurrent operation of multiple DCPs is discussed briefly in §4.4.

4.1 Software Conventions

The principal five functions from Figure 3 are declared in the C header file `DCPLIB.h`, as given in §4.2. The C Language definitions of `DCPLIB.c` are given in §4.3.

The remainder of §4.1 is numbered in correspondence with §4.3. For example, §4.1.2 corresponds to §4.3.2, and both relate to the `DCPsenseDtable` function.

When the DCP is configured with more than 512 Kbytes of DRAM, it becomes a multi-channel device. Each DCP channel is associated with a dictionary table in DRAM. A DCP can address enough DRAM to accommodate a maximum of 64 tables, or 64 unidirectional channels. The actual number of channels for a given DRAM configuration can be determined using the `DCPsenseDtable` function.

The dictionary tables are updated by the Genetic Compression Algorithm, to reflect the history of the channel. Each call to the `DCPcompress` function produces compressed text, and also updates the specified dictionary table.

For the reconstruction process, each call to the `DCPexpand` function reproduces the original plaintext. Each call applies the same updates to the specified dictionary as did the corresponding call to `DCPcompress`.

Provided that the compression and expansion processes each start with identical dictionary tables, the two dictionaries will remain synchronized and will conclude in identical states.

Note that the DCP is block-oriented, in the sense that `DCPcompress` operates on a block of plaintext to produce the corresponding block of compressed text. The reconstruction process must pass the compressed blocks to `DCPexpand`, in sequence, with block boundaries preserved. Plaintext and compressed blocks are always byte-string data structures.

The `DCPinitialize` function builds a valid dictionary table, initialized to the standard starting state, as specified by the Genetic Compression Algorithm. The dictionary table is a relatively large data structure: assuming that the data path to DRAM is 16 bits wide, `DCPinitialize` takes just under one million DCP CLK ticks to complete. With a 40-MHz CLK frequency, that translates into approximately 25 ms.

Any action, event, or error that could cause the dictionary tables of the compression and expansion processes to lose synchronization, is said to invalidate the affected dictionaries.

The following actions invalidate the entire DRAM module:

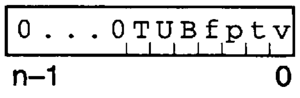
- hardware $\overline{\text{RESET}}$
- `DCPresetPowerUp` function
- `DCPsenseDtable` function

The following events invalidate a single dictionary:

- channel establishment
- unrecoverable loss of data in compressed channel
- non-zero result code from `DCPinitialize`, `DCPcompress`, or `DCPexpand`

`DCPinitialize`, `DCPcompress`, and `DCPexpand` share a common convention for result codes. Each of the three returns result code 0 if the requested operation was completed without incident. Otherwise, the result code is a vector of error flags packed into an `int`, as shown in Figure 19.

Figure 19. DCPLIB Result Codes



The `int` word is n bits wide

`T`	= time-out check
`U`	= unspecified check
`B`	= buffer overflow check
`f`	= firmware check
`p`	= parity check
`t`	= format check
`v`	= overrun/underrun check

The `fptv` bits reflect the RR1 status of the DCP, as defined in §3.5.2. The `TUB` bits are provided for reporting software-detected conditions.

The other two principal functions, `DCPresetPowerUp` and `DCPsenseDtable`, return `void`.

4.1.1 DCPresetPowerUp Function

The `DCPresetPowerUp` function prepares the DCP for operation.

`DCPresetPowerUp` is performed once after DCP hardware initialization. Note that the DCP requires initialization via the $\overline{\text{RESET}}$ line when power is first applied.

Because this function alters WR3[7:6] as described in §3.4.4, `DCPresetPowerUp` invalidates all dictionary tables. The `DCPsenseDataPath` function is called to set the data path configuration WR3[7:6] automatically.

4.1.2 DCPsenseDtable Function

As described in §3.4.3, address wrapping occurs in dictionary/buffer DRAM configurations smaller than 32 Mbytes. `DCPsenseDtable` detects the effects of address wrapping, and eliminates aliases. What remains after aliases have been eliminated, is a set of distinct dictionary indices.

The `DCPsenseDtable` function senses the DRAM size and constructs the set **D** of distinct dictionary indices. The set **D** is represented by a Boolean array `Dtable[64]`. Each element of the array is an `int`, the value of which is either 0 or 1. The size of the DRAM can be determined by counting the 1s in `Dtable`, and multiplying by 512 Kbytes per dictionary.

Membership of an element `d` in the set **D** can be tested with the C expression `Dtable[d]`.

Usually `DCPsenseDtable` is called once, immediately following `DCPresetPowerUp`. `DCPsenseDtable` invalidates all dictionary tables.

The `Dtable` array normally will be declared with global scope so that it can be consulted whenever a new dictionary index is required. The `DCPinitialize`, `DCPcompress`, and `DCPexpand` functions all require distinct dictionary indices for correct operation. Never use a dictionary index `d` unless `Dtable[d]`==1.

Example:

```
int Dtable[64];
DCPresetPowerUp ();
DCPsenseDtable (Dtable);
```

Following execution of the program fragment above, each element of `Dtable` is set to either 0 or 1, depending on whether the corresponding dictionary index is distinct.

4.1.3 DCPinitialize Function

The `DCPinitialize` function builds a valid dictionary table. Speaking in terms of the Genetic Compression Algorithm, the dictionary is a population of recognizers. Starting with a large population of standard recognizers reduces the training time for the genetic algorithm.

Example:

```
int rc, d;
d = 0;
rc = DCPinitialize (d);
if (rc) printf ("DCPinitialize (%d) == %d\n", d, rc);
```

The argument `d` should be a distinct dictionary index, as described in §4.1.2. Following execution of the program fragment, the dictionary `d` is valid if the result code `rc`==0. Otherwise the result code indicates failure, and dictionary `d` is invalidated.

4.1.4 DCPcompress Function

The `DCPcompress` function produces a compact representation for a block of plaintext. Both plaintext and compressed blocks are byte-string data structures.

The history of the plaintext channel affects the makeup of the specified dictionary table. Speaking in terms of the Genetic Compression Algorithm, the population of recognizers adapts in real time, to the characteristics of the plaintext channel. Fitness is a function of the compression ratio.

Example:

```
int rc, d, len1, len2;
char buf1[100], buf2[100];
d = 0;  /* DCPcompress will need a valid dictionary */
DCPinitialize (d);
strcpy (buf1, "Compression == Competitive Advantage\n");
len1 = strlen (buf1);
len2 = 100;
rc = DCPcompress (d, buf1, len1, buf2, &len2);
if (rc) printf ("DCPcompress (%d, ...) == %d\n", d, rc);
```

The argument `d` should be a distinct dictionary index, as described in §4.1.2. Dictionary `d` also must be valid, otherwise the result of the `DCPcompress` function is undefined (§3.4.3).

Prior to the call to `DCPcompress`, `len1` is set to the size of the plaintext in `buf1`, and `len2` is set to the size of the buffer `buf2`. If `DCPcompress` returns result code `rc`==0, then `len2` will reflect the size of the compressed text. All sizes are given in bytes.

The plaintext length `len1` should not exceed 16000 bytes. If the `len1` restriction is exceeded, `DCPcompress` may or may not fail, depending on the channel history.

For some data, the Compress operation produces a compressed result that is actually larger than the corresponding plaintext. Swelling is particularly apparent for random data. In the worst case, the Compress operation may produce a "compressed" result 9/8 times the plaintext size, for a 12.5% bloat. Note that worst-case behavior is unlikely. Usual swelling for random data hovers around 3%. §4.1.5 describes an efficient anti-expansion encoding technique.

4.1.5 DCPexpand Function

The `DCPexpand` function is the inverse of `DCPcompress`, in the sense that plaintext blocks are reconstructed from the compressed representation. However, `DCPcompress` also modifies the specified dictionary table to reflect the history of the plaintext channel. `DCPexpand` must mimic the modifications exactly to keep its working dictionary synchronized and up to date.

Corresponding calls to the `DCPcompress` and `DCPexpand` functions perform identical actions with respect to the specified dictionaries. This symmetry between the Compress and Expand operations can be exploited for an efficient anti-expansion encoding. With 1 bit of overhead per frame, an encoder can indicate whether it is transmitting plaintext or compressed text. The next paragraph describes the technique:

The encoder calls `DCPcompress`, which updates the encoder dictionary. The encoder then transmits either plaintext or compressed text, whichever is more compact, together with the 1-bit mark. If the decoder receives a frame that is marked *compressed*, it uses `DCPexpand` to recover the plaintext and to update the decoder dictionary. If the received frame is marked *plaintext*, the decoder uses `DCPcompress` to update the decoder dictionary and simply discards the compressed result. Either way, the encoder and decoder dictionaries remain synchronized.

Using the above technique, compression can be switched off and on as needed, on a frame-by-frame basis. The overhead bit often can be located in encapsulation headers, without added cost.

Example: (continuing the example of §4.1.4)

```
d = 2;   /* use a different table for reconstruction */
DCPinitialize (d);
strcpy (buf1, "Clobber the message from our Sponsor\n");
len1 = 100;   /* reconstruct buf1 from buf2, len2 */
rc = DCPexpand (d, buf2, len2, buf1, &len1);
if (rc) printf ("DCPexpand (%d, ...) == %d\n", d, rc);
else printf (buf1);
```

The argument `d` should be a distinct dictionary index, as described in §4.1.2. Dictionary `d` also must be valid, otherwise the result of the `DCPexpand` function is undefined (§3.4.3).

In the example of §4.1.4, `DCPcompress` assigned to `len2` the size of the compressed text in `buf2`. For the current example, assume that the assigned values for `buf2` and `len2` are carried over.

Prior to the call to `DCPexpand`, `len1` is set to the size of the buffer `buf1`. If `DCPexpand` returns result code `rc`==0, then `len1` will reflect the size of the reconstructed block in `buf1`. All sizes are given in bytes.

Following the call to `DCPexpand`, `buf1` should contain the original message from §4.1.4.

4.1.6 DCPterminateOperation Function

The `DCPterminateOperation` function ensures orderly termination of any active operation, and forces the DCP into the idle state. The termination process drains the DCP instruction pipeline and quiesces the integrated DRAM controller.

Some of the DCPLIB functions call `DCPterminateOperation` only when necessary. For example, the `DCPcompress` function (§4.3.4) examines the end-of-operation bit before it issues a command. If the end-of-operation bit RR1[6]==1, then the DCP already is idle.

`DCPterminateOperation` is the only DCPLIB function to use the Reserved command, described in Figure 15. The DCP processes the Reserved command by setting both firmware check RR1[3] and end-of-operation RR1[6].

`DCPterminateOperation` is an ancillary function of DCPLIB, and is intended for exclusive use by other DCPLIB functions.

4.1.7 DCPsenseDataPath Function

The `DCPsenseDataPath` function senses the data path to DRAM, and establishes the data path configuration.

The WR3[7:6] configuration bits are set as a side effect to reflect the actual width of the DRAM module. `DCPsenseDataPath` returns an `int` result code, defined in Figure 20.

Figure 20. DCPsenseDataPath Result Codes

Result	WR3[7:6]	Description
0	xx	DCP non-operational
1	01	Using DD[7:0]
2	10	Using DD[15:8]
3	11	Using DD[15:0]

The `DCPsenseDataPath` function conducts a systematic search for usable dictionary memory. The breadth of the search is controlled by the `Dmax` parameter. For example, if `Dmax`==4 then `DCPsenseDataPath` will test at most 4 dictionary indices before giving up. A smaller value for `Dmax` will make the function more sensitive to bad cells within the DRAM module.

Note that the `Dmax` parameter must not exceed 64.

`DCPsenseDataPath` is an ancillary function of DCPLIB, and is called exclusively by the `DCPresetPowerUp` function. Because this function alters WR3[7:6] as described in §3.4.4, all dictionary tables in the DRAM are invalidated.

4.1.8 DCPlclPutBlk Function

The `DCPlclPutBlk` function copies a byte string from a specified buffer, to the WR0 data port of the DCP, and then signals end-of-block via RR2.

The elementary DCPLIB uses the signal pins of Figure 18. Flow-control is accomplished by polling the status port, and checking the transfer request bit RR1[7]. Data transfer continues until either the buffer is exhausted or a check bit is set by the DCP.

The `DCPlclPutBlk` function is modified easily to use a DMA controller. The signal pins of the DMA Transfer Interface are depicted in Figures 2 and 21. The hardware flow-control method is faster and uses far fewer bus cycles, as compared with software flow-control.

`DCPlclPutBlk` is an ancillary function of DCPLIB, and is intended for exclusive use by the `DCPcompress` and `DCPexpand` functions.

4.1.9 DCPlclGetBlk Function

The `DCPlclGetBlk` function copies a byte string from the RR0 data port of the DCP, to a specified buffer. After the last byte is transferred, the DCP indicates end-of-operation. The size of the byte string is returned as an `int`.

Note that a special return value is used to indicate failure. If `DCPlclGetBlk` detects buffer overflow, the returned result is -1.

The elementary DCPLIB uses the signal pins of Figure 18. Flow-control is accomplished by polling the status port, and checking the transfer request bit RR1[7]. Data transfer continues until the end-of-operation bit is set, a check bit is set, or the specified buffer is overflowed.

Like `DCPlclPutBlk`, the `DCPlclGetBlk` function is modified easily to use a DMA controller. The preprocessor symbol `DCPppsbraei` should be redefined to enable one or more of the DMA handshake lines. For the example of Figure 21, WR3[3] and WR3[1] would be set to enable the $\overline{\text{REQ}}$ and $\overline{\text{END}}$ lines. If the driver software uses $\overline{\text{IRQ}}$, WR3[0] also would be set.

`DCPlclGetBlk` is an ancillary function of DCPLIB, and is intended for exclusive use by the `DCPcompress` and `DCPexpand` functions.

The C Language code of §4.3.4 and §4.3.5 can be improved with the addition of time-out checks. As described in §3.4.3, a damaged or invalid dictionary table can cause hung firmware. Some sort of timer should be employed for all calls to `DCPlclPutBlk` and `DCPlclGetBlk`.

DCP time-out failures are unlikely. For many applications, a global watchdog timer is sufficient to ensure robust system operation. Recovery from a time-out check always will include a call to the `DCPterminateOperation` function.

4.2 DCPLIB.h Header File

```
/* Five principal functions: */

void DCPresetPowerUp (void);

void DCPsenseDtable (int Dtable[/*64*/]);

int DCPinitialize (int D);

int DCPcompress (int D, char *from, int flen,
                        char *to, int *tlen);

int DCPexpand (int D, char *from, int flen,
                      char *to, int *tlen);

/* Ancillary functions: */

void DCPterminateOperation (void);

int DCPsenseDataPath (int Dmax);

void DCPlclPutBlk (char *buf, int len);

int DCPlclGetBlk (char *buf, int len);
```

4.3 DCPLIB.c Source Listings

```
/* Memory Refresh Controller Parameters */
#define DCPclkFreq   40000000
#define DCPrefRate   512 * 125

/* Platform Address Map */
#define DCPbase      0x03AC  /* IBM PC I/O */
#define DCPdata      0       /*  RR0  WR0  */
#define DCPdivisor   1       /*       WR1  */
#define DCPstatus    1       /*  RR1       */
#define DCPcommand   2       /*       WR2  */
#define DCPeob       2       /*  RR2       */
#define DCPconfig    3       /*  RR3  WR3  */

/* Commands */
#define DCPcomp      0x00    /* Compress   */
#define DCPexpa      0x40    /* Expand     */
#define DCPinit      0x80    /* Initialize */
#define DCPrsrv      0xC0    /* Reserved   */

/* Basic Configuration Image */
#define DCPppsbraei  0xC0

volatile int DCPdummyVar;   /* force side effects */
```

4.3.1 DCP and DRAM Initialization

```
void DCPresetPowerUp (void)
{
  int i;
  /* calculate refresh divisor n */
  int n = DCPclkFreq / 8 / ((long) DCPrefRate);
  if (n < 1) n = 1; if (n > 256) n = 256;
  /* orderly termination of current operation */
  DCPterminateOperation ();
  /* start full-speed memory refresh process */
  outportb (DCPbase+DCPdivisor, 0);
  outportb (DCPbase+DCPconfig,  DCPppsbraei);
  /* pump up internal DRAM nodes */
  outportb (DCPbase+DCPcommand, DCPinit+0);
  DCPdummyVar = inportb (DCPbase+DCPeob);
  /* wait at least 100 DCP CLK ticks */
  for (i=0; i<100; ++i)
    DCPdummyVar = inportb (DCPbase+DCPstatus);
  /* done pumping internal DRAM nodes */
  DCPterminateOperation ();
  /* back off to specified memory refresh rate */
  outportb (DCPbase+DCPdivisor, n-1);
  /* automatic data path configuration */
  DCPsenseDataPath (4);
}
```

4.3.2 Obtain Memory Configuration

```
void DCPsenseDtable (int Dtable[/*64*/])
{
  int d;
  int ppsbraei; /* RR3 config */
  int len2;
  char buf1[8], buf2[8];
  /* obtain current pp bits */
  ppsbraei = inportb (DCPbase+DCPconfig);
  /* initialize dictionaries [0:63] */
  for (d=0; d<64; ++d) {
    Dtable[d] = 0;
    /* no odd indices in 8-bit mode (Section 3.4.3) */
    if ((ppsbraei&0xC0)!=0xC0 && (d&1)) continue;
    DCPinitialize (d);
  }
  /* this pattern compresses to 3 bytes on first pass */
  /* on subsequent passes, this compresses to 2 bytes */
  buf1[0] = 0x7F; buf1[1] = 0x80;
  /* find lowest aliased address for each dictionary */
  for (d=0; d<64; ++d) {
    if ((ppsbraei&0xC0)!=0xC0 && (d&1)) continue;
    /* set target buffer length and compress buf1 */
    len2 = 8;
    DCPcompress (d, buf1, 2, buf2, &len2);
    /* compressed length is returned in len2 */
    if (len2 == 3) Dtable[d] = 1;
  }
}
```

4.3.3 Initialize Dictionary

```
int DCPinitialize (int D)
{
  long timer = 0;
  int redmfptv; /* RR1 status */
  /* check for DCP busy, force end-of-operation */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (!(redmfptv & 0x40)) DCPterminateOperation ();
  /* issue Initialize command */
  outportb (DCPbase+DCPcommand, DCPinit+D);
  /* signal end-of-block, null operand */
  DCPdummyVar = inportb (DCPbase+DCPeob);
  /* wait for end-of-operation */
  redmfptv = 0;
  while (!(redmfptv & 0x4F)) {
    /* should take less than 2000000 DCP CLK ticks */
    if ((++timer) == 2000000L) return (0x40);
    /* polling status */
    redmfptv = inportb (DCPbase+DCPstatus);
  }
  /* result is error flags */
  return (redmfptv & 0x0F);
}
```

4.3.4 Compress Block

```
int DCPcompress (int D, char *from, int flen,
                           char *to, int *tlen)
{
  int k;
  int redmfptv; /* RR1 status */
  /* check for DCP busy, force end-of-operation */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (!(redmfptv & 0x40)) DCPterminateOperation ();
  /* issue Compress command */
  outportb (DCPbase+DCPcommand, DCPcomp+D);
  /* transfer plaintext operand to DCP */
  DCPlclPutBlk (from, flen);
  /* check for errors */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (redmfptv&0x0F) return (redmfptv&0x0F);
  /* transfer compressed result from DCP */
  k = DCPlclGetBlk (to, *tlen);
  /* k<0 means the result is too large for buffer */
  /* check for errors */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (k < 0) return (0x10 | (redmfptv&0x0F));
  if (redmfptv&0x0F) return (redmfptv&0x0F);
  /* trim trailing 0's from compressed block */
  while (k>0 && to[k-1]==0) --k;
  /* length of result */
  *tlen = k;
  /* indicate no errors */
  return (0);
}
```

4.3.5 *Expand Block*

```
int DCPexpand (int D, char *from, int flen,
                              char *to, int *tlen)
{
  int k;
  int redmfptv; /* RR1 status */
  /* check for DCP busy, force end-of-operation */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (!(redmfptv & 0x40)) DCPterminateOperation ();
  /* issue Expand command */
  outportb (DCPbase+DCPcommand, DCPexpa+D);
  /* transfer compressed operand to DCP */
  DCPlclPutBlk (from, flen);
  /* check for errors */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (redmfptv&0x0F) return (redmfptv&0x0F);
  /* transfer plaintext result from DCP */
  k = DCPlclGetBlk (to, *tlen);
  /* k<0 means the result is too large for buffer */
  /* check for errors */
  redmfptv = inportb (DCPbase+DCPstatus);
  if (k < 0) return (0x10 | (redmfptv&0x0F));
  if (redmfptv&0x0F) return (redmfptv&0x0F);
  /* length of result */
  *tlen = k;
  /* indicate no errors */
  return (0);
}
```

4.3.6 *Orderly Termination*

```
void DCPterminateOperation (void)
{
  int i;
  int ccdddddd; /* RR2 command */
  /* read previous command image */
  ccdddddd = inportb (DCPbase+DCPcommand);
  /* issue Reserved command with previous dict index */
  /* this will drain internal DCP pipelines without   */
  /* messing up the integrated DRAM controller        */
  outportb (DCPbase+DCPcommand, DCPrsrv+(ccdddddd&0x3F));
  /* signal end-of-block, null operand */
  DCPdummyVar = inportb (DCPbase+DCPeob);
  /* wait at least 20 DCP CLK ticks */
  for (i=0; i<20; ++i)
    DCPdummyVar = inportb (DCPbase+DCPstatus);
  /* issue Reserved command, setting GCA level 0 */
  outportb (DCPbase+DCPcommand, DCPrsrv+0);
  /* signal end-of-block, null operand */
  DCPdummyVar = inportb (DCPbase+DCPeob);
  /* wait at least 10 DCP CLK ticks */
  for (i=0; i<10; ++i)
    DCPdummyVar = inportb (DCPbase+DCPstatus);
}
```

4.3.7 Data Path Configuration

```
int DCPsenseDataPath (int Dmax)
{
  int d;
  int ppsbraei; /* RR3 config */
  int len2;
  char buf1[8], buf2[8];
  /* this pattern compresses to 3 bytes on first pass */
  /* on subsequent passes, this compresses to 2 bytes */
  buf1[0] = 0x7F; buf1[1] = 0x80;
  /* obtain current pp bits */
  ppsbraei = inportb (DCPbase+DCPconfig);
  /* force end-of-operation */
  DCPterminateOperation ();
  /* configure for 16-bit data path DD[15:0] */
  outportb (DCPbase+DCPconfig, 0xC0|(ppsbraei&0x3F));
  /* attempt to operate the DCP */
  for (d=0; d<Dmax; ++d) {
    /* try to find a dictionary that is OK */
    if (DCPinitialize (d) == 0) {
      /* set target buffer length and compress buf1 */
      len2 = 8;
      DCPcompress (d, buf1, 2, buf2, &len2);
      /* compressed length is returned in len2 */
      if (len2 == 3) return (3); /* pp == 11 */
    }
  }
  /* tidy up after failure */
  DCPterminateOperation ();
  /* next, try 8-bit data path DD[15:8] */
  outportb (DCPbase+DCPconfig, 0x80|(ppsbraei&0x3F));
  for (d=0; d<Dmax; d+=2) {
    if (DCPinitialize (d) == 0) {
      len2 = 8;
      DCPcompress (d, buf1, 2, buf2, &len2);
      if (len2 == 3) return (2); /* pp == 10 */
    }
  }
  DCPterminateOperation ();
  /* next, try 8-bit data path DD[7:0] */
  outportb (DCPbase+DCPconfig, 0x40|(ppsbraei&0x3F));
  for (d=0; d<Dmax; d+=2) {
    if (DCPinitialize (d) == 0) {
      len2 = 8;
      DCPcompress (d, buf1, 2, buf2, &len2);
      if (len2 == 3) return (1); /* pp == 01 */
    }
  }
  DCPterminateOperation ();
  /* restore starting pp configuration */
  outportb (DCPbase+DCPconfig, ppsbraei);
  /* indicate DCP non-operational */
  return (0);
}
```

4.3.8 Transfer Block to DCP

```
void DCPlclPutBlk (char *buf, int len)
{
  int redmfptv; /* RR1 status */
  /* transfer operand bytes to DCP WR0 */
  while (len--) {
    redmfptv = 0;
    /* wait for transfer request */
    while (!(redmfptv & 0x80)) {
      redmfptv = inportb (DCPbase+DCPstatus);
      /* check for errors */
      if (redmfptv & 0x0F) return;
    }
    /* transfer 1 byte to WR0 */
    outportb (DCPbase+DCPdata, *buf++);
  }
  /* signal end-of-block */
  DCPdummyVar = inportb (DCPbase+DCPeob);
}
```

4.3.9 Transfer Block from DCP

```
int DCPlclGetBlk (char *buf, int len)
{
  int redmfptv; /* RR1 status */
  int k;  /* length of result */
  /* transfer result bytes from DCP RR0 */
  for (k=0; k<len; ++k) {
    redmfptv = 0;
    /* wait for transfer request */
    while (!(redmfptv & 0x80)) {
      redmfptv = inportb (DCPbase+DCPstatus);
      /* check for errors or end-of-operation */
      if (redmfptv & 0x4F) return (k);
    }
    /* transfer 1 byte from RR0 */
    *buf++ = inportb (DCPbase+DCPdata);
  }
  /* indicate the result is too large for buffer */
  return (-1);
}
```

4.4 Advanced DCP816 Applications

The Motorola MC68360 combines a host of hardware assists for telecommunication input/output and control. It is popular, and it represents a trend toward greater integration in telecommunication processors. Figure 1 depicts a typical application of the MC68360 and the DCP816.

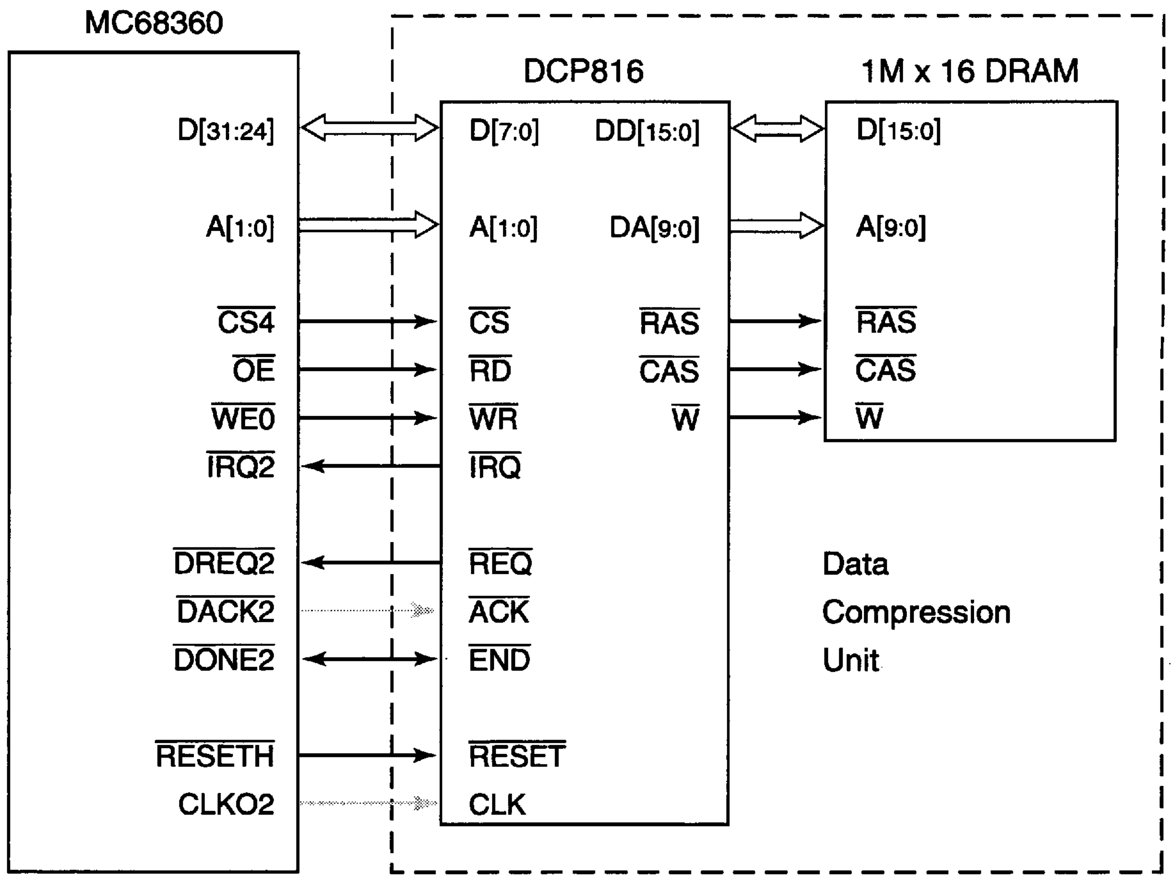
Figure 21 details the glueless interface between the MC68360 and a single DCP816. The $\overline{\text{ACK}}$ signal is not needed for the dual-address DMA facility of the MC68360. The CLKO2 driver on the MC68360 might be used in an ultra-low-cost design, but it is more usual to provide a separate oscillator for the DCP CLK input.

While the DCP throughput is around 210 Kbytes/s, the MC68360 DMA controller can transfer to the DCP at over 4 Mbytes/s. It is natural to imagine a single DMA controller driving burst transfers for multiple DCPs. With carefully constructed driver software, an array of DCPs can serve parallel communication processes, and can achieve high aggregate throughput.

The MC68360 serial communication channels support frame- or buffer-oriented processing. Like the MC68360, the DCP816 is block-oriented. Software engineers will recognize the advantages of the close architectural compatibility between the DCP816 and the MC68360.

Note finally that the DCP816 is optimized for shared use of a single DMA controller, whereas competing products use complicated, dual-port, stream-oriented architectures.

Figure 21. Glueless DMA Interface from MC68360 to DCP816



5. STATISTICAL PROPERTIES OF COMPRESSED TEXT

All 256 possible 8-bit patterns may appear within the compressed stream. The pattern frequencies are approximately uniform. Typically correlations within a plain stream are reflected in the compressed stream, although such correlations may be difficult to discern.

The redundancy due to encoding overheads within DCP816 compressed text can be estimated by compressing a random stream, and comparing the plain size against the compressed size. A random stream is used because the information content is known. Since the DCP816 swells random data by 3 to 4%, the encoding overhead for the random stream is also 3 or 4%.

Zero-bytes (00000000_2) never occur in runs within the compressed text. `DCPcompress` trims trailing zero-bytes, so the final byte of a block is never zero.

The following C Language code performs a more precise check for well-formed compressed text, and could be useful for debugging purposes:

```
int DCPtextWellFormed (char *from, int flen)
{
  int i, k;
  long v;
  if (flen>0 && from[flen-1]==0) return (0);
  i = v = 0;
  while (1) {
    while (i < 16) {
      if (flen > 0) {k = *from++; --flen;}
      else k = 0;
      v = (v << 8) | (k & 0xFF);
      i += 8;
    }
    --i;
    if (((v >> i) & 1) == 1) i -= 8;
    else {
      i -= 15;
      k = (v >> i) & 0x7FFF;
      if (k == 0) break;
      if (k >= 0x7F00) return(0);
    }
  }
  if (flen || (v & 0x7FFF)) return (0);
  return (1); /* compressed text is well-formed */
}
```

`DCPtextWellFormed` would be called prior to `DCPexpand`, to ensure that the latter call will not fail due to malformed compressed text. The arguments `from` and `flen` would be the same for both calls.

Previous versions of this manual were released on the following dates:

23 Nov 92

25 Feb 94