MOTOROLA

MC6829
(1.0 MHz)
MC68A29
(1.5 MHz)
MC68B29
(2.0 MHz)

Advance Information

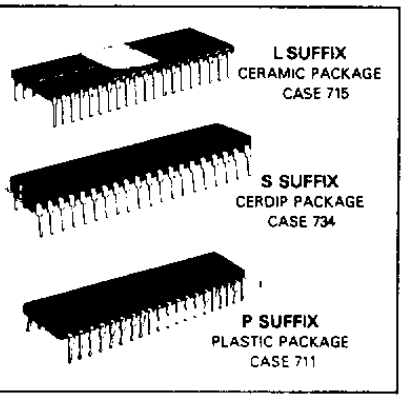
HMOS
(HIGH DENSITY N-CHANNEL, SILICON-GATE)

MEMORY MANAGEMENT UNIT (MMU)

MEMORY MANAGEMENT UNIT

The principle function of the MC6829 Memory Management Unit (MMU) is to expand the address space of the MC6809 from 64K bytes to a maximum of 2 Megabytes. Each MMU is capable of handling four different concurrent tasks including DMA. The MMU can also protect the address space of one task from modification by another task. Memory address space expansion is accomplished by applying the upper five address lines of the processor (A11-A15) along with the contents of a 5-bit task register to an internal high-speed mapping RAM. The MMU output consists of ten physical address lines (PA11-PA20) which, when combined with the eleven lower address lines of the processor (A0-A10), forms a physical address space of 2 Megabytes. Each task is assigned memory in increments of 2K bytes up to a total of 64K bytes. In this manner, the address spaces of different tasks can be kept separate from one another. The resulting simplification of the address space programming model will increase the software reliability of a complex multiprocess system.

- Expands Memory Address Space from 64K to 2 Megabytes
- Each MMU is Capable of Handling Four Separate Tasks
- Up to Eight MMUs can be Used in a System
- Provides Task Isolation and Write Protection
- Provides Efficient Memory Allocation; 1024 Pages of 2K Bytes Each
- Designed for Efficient Use with DMA
- Fast, Automatic On-Chip Task Switching
- Allows Inter-Process Communication Through Shared Resources
- Simplifies Programming Model of Address Space
- Increases System Software Reliability
- MC6809/MC6800 Bus Compatible
- Single 5-Volt Power Supply



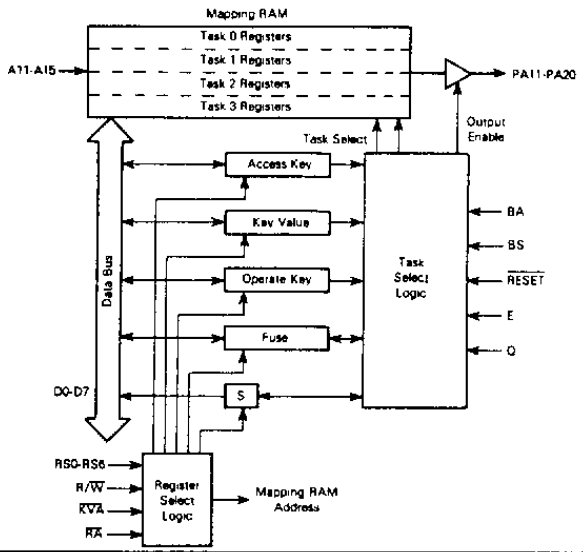
L SUFFIX
CERAMIC PACKAGE
CASE 715

S SUFFIX
CERDIP PACKAGE
CASE 734

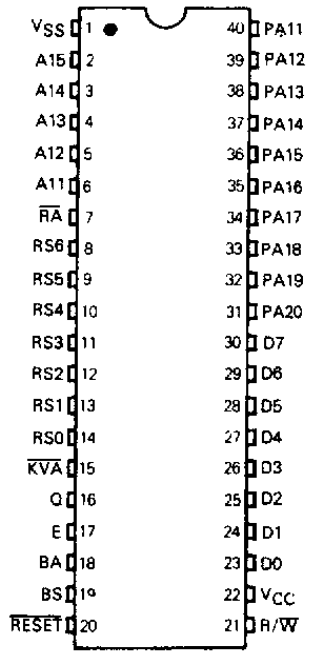
P SUFFIX
PLASTIC PACKAGE
CASE 711

4

BLOCK DIAGRAM



PIN ASSIGNMENT



MAXIMUM RATINGS

Characteristics	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.3 to +7.0	V
Operating Temperature Range MC6829, MC68A29, MC68B29 MC6829C, MC68A29C, MC68B29C	T_A	T_L to T_H 0 to 70 −40 to +85	°C
Storage Temperature Range	T_{stg}	−55 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either V_{SS} or V_{CC}).

THERMAL CHARACTERISTICS

	Symbol	Value	Rating
Thermal Resistance Plastic Cerdip Ceramic	θ_{JA}	 100 60 50	°C/W

4

POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from:

$$T_J = T_A + (P_D \bullet \theta_{JA}) \quad (1)$$

Where:

T_A = Ambient Temperature, °C

θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W

$P_D = P_{INT} + P_{PORT}$

$P_{INT} = I_{CC} \times V_{CC}$, Watts — Chip Internal Power

P_{PORT} = Port Power Dissipation, Watts — User Determined

For most applications $P_{PORT} \ll P_{INT}$ and can be neglected. P_{PORT} may become significant if the device is configured to drive Darlington bases or sink LED loads.

An approximate relationship between P_D and T_J (if P_{PORT} is neglected) is:

$$P_D = K \div (T_J + 273°C) \quad (2)$$

Solving equations 1 and 2 for K gives:

$$K = P_D \bullet (T_A + 273°C) + \theta_{JA} \bullet P_D^2 \quad (3)$$

Where K is a constant pertaining to the particular part. K can be determined from equation 3 by measuring P_D (at equilibrium) for a known T_A. Using this value of K the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

DC ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 Vdc ±5%, V_{SS} = 0, $T_A = T_L$ to T_H unless otherwise noted)

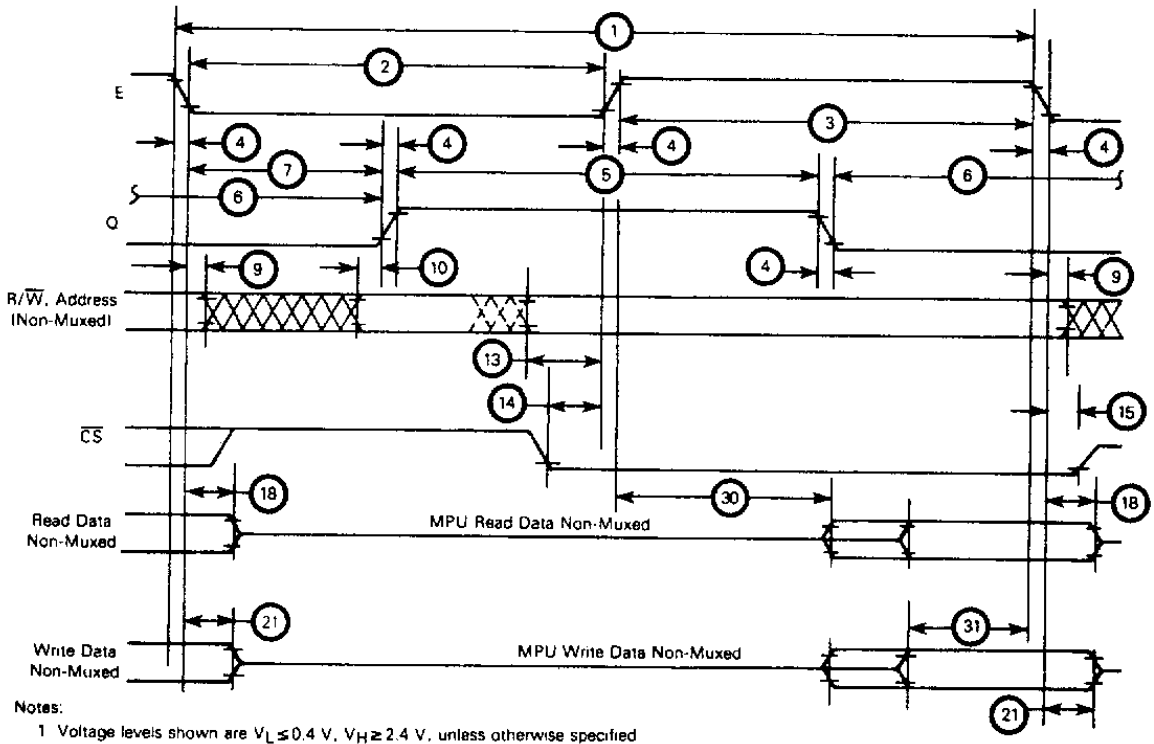
Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage	All Inputs	V_{IH}	V_{SS}+2.0	–	V_{CC}	V
Input Low Voltage	All Inputs	V_{IL}	V_{SS}−0.3	–	V_{SS}+0.8	V
Input Leakage Current (V_{in} = 0 to 5.25 V)	V_{CC} = Max	I_{in}	–	1.0	2.5	μA
Three-State (Off State) Input Current (V_{in} = 0.4 to 2.4 V)	D0-D7	I_{IZ}	–	2.0	10	μA
Output High Voltage (I_{load} = −145 μA) V_{CC} = min	 D0-D7 PA11-PA20	V_{OH}	 V_{SS}+2.4 V_{SS}+2.4	 – –	 – –	V
Output Low Voltage (I_{load} = 2.0 mA) V_{CC} = max	 D0-D7 PA11-PA20	V_{OL}	 – –	 – –	 V_{SS}+0.5 V_{SS}+0.5	V
Internal Power Dissipation (Measured at $T_A = T_L$)		P_{INT}	–	–	800	mW
Input Capacitance (V_{in} = 0, T_A = 25°C, f = 1.5 MHz)	All Inputs	C_{in}	–	10.0	12.0	pF
Output Capacitance (V_{in} = 0, T_A = 25°C, f = 1.5 MHz)	All Outputs	C_{out}	–	–	12.0	pF

BUS TIMING CHARACTERISTICS (See Notes 1 and 2)

Ident. Number	Characteristic	Symbol	MC6829		MC68A29		MC68B29		Unit
			Min	Max	Min	Max	Min	Max	
1	Cycle Time	t_{cyc}	1.0	10	0.667	10	0.5	10	μs
2	Pulse Width, E Low	PW_{EL}	430	9500	280	9500	210	9700	ns
3	Pulse Width, E High	PW_{EH}	450	9500	280	9500	220	9700	ns
4	Clock Rise and Fall Time	t_r, t_f	–	25	–	25	–	20	ns
5	Pulse Width, Q High	PW_{QH}	430	5000	280	5000	210	5000	ns
6	Pulse Width, Q Low	PW_{QL}	450	9500	280	9500	220	9500	ns
7	E to Q Rise Delay Time*	t_{AVQ}	–	250	–	165	–	125	ns
9	Address Hold Time	t_{AH}	10	–	10	–	10	–	ns
13	Address Setup Time Before E	t_{AS}	80	–	60	–	40	–	ns
14	Chip Select Setup Time Before E	t_{CS}	80	–	60	–	40	–	ns
15	Chip Select Hold Time	t_{CH}	10	–	10	–	10	–	ns
18	Read Data Hold Time	t_{DHR}	20	100	20	100	20	100	ns
21	Write Data Hold Time	t_{DHW}	10	–	10	–	10	–	ns
30	Output Data Delay Time	t_{DDR}	–	290	–	180	–	150	ns
31	Input Data Setup Time	t_{DSW}	165	–	80	–	60	–	ns
See Figures 2 and 3	Three-State Address Delay	t_{TAD}	–	90	–	80	–	60	ns
See Figure 2	Mapped Address Delay	t_{MAD}	–	200	–	145	–	110	ns

*At specified cycle time.

FIGURE 1 — BUS TIMING



Notes:
1. Voltage levels shown are $V_L \leq 0.4$ V, $V_H \geq 2.4$ V, unless otherwise specified
2. Measurement points shown are 0.8 V and 2.0 V, unless otherwise specified

FIGURE 2 — MAP SWITCHING, ADDRESS MAPPING

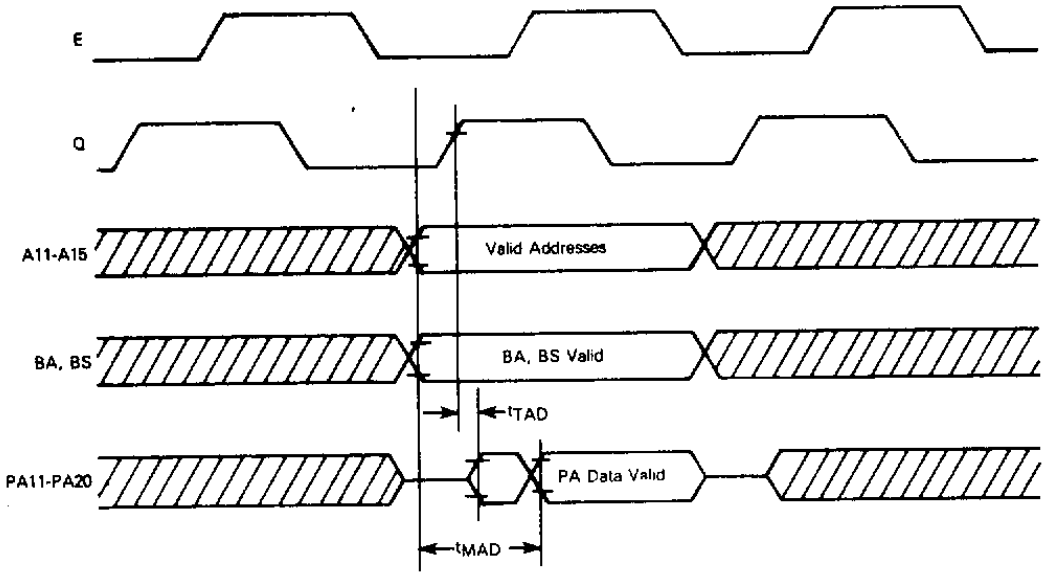
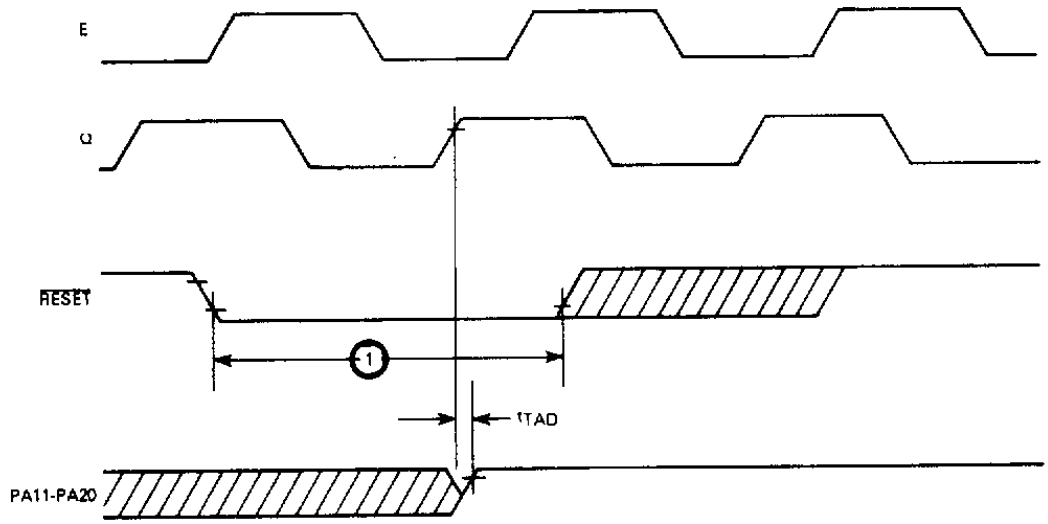


FIGURE 3 — $\overline{\text{RESET}}$ TIMING



Note: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

PIN DESCRIPTION

The following section describes each pin of the MMU in detail.

V_{CC}, V_{SS} – Supplies power to the MC6829. V_{CC} is +5 volts and V_{SS} is ground.

E – Input E clock (from MC6809).

Q – Input Q clock (from MC6809).

R/$\overline{W}$ – Read/Write Line Input; 1 = Read, 0 = Write.

D0-D7 – Bi-directional Data Bus. The data bus is used when the MMU registers are to be read or written.

PA11-PA20 – Physical Address Lines (Output from MMU). The physical address lines are generated by the MMU for every bus cycle. When multiple MMUs are present in a system, only one MMU will output a physical address. Each physical address line will drive one Schottky TTL load or four LS TTL loads and a maximum of 90 pF.

RS0-RS6 – Register Select Lines (Access to MMU Registers). When accessing the MMU registers, the register select lines determine which byte of information is being referenced within the MMU. Valid addresses are detailed in the Register Select Truth Table.

BA, BS – Bus Available and Bus State (Inputs). These inputs are directly connected from the BA, BS lines of the MC6809. They provide the MMU with information about the class of bus operation for each cycle. Note that when coming out of a DMA cycle, the MC6809 BA, BS pins change back from DMA acknowledge (BA = 1, BS = 1) to running (BA = 0, BS = 0) one cycle before the end of the DMA.

$\overline{RA}$ – Register Access (Chip Select for MMU Registers). This active low input determines the location of the MMU registers. Since the MMU registers are only accessible from the last page of task #0 ($F800-$FFFF), this signal can be derived from address lines A10-A7 of the processor. When RA is asserted low, the MMU registers are selected if the current task number is zero and A15-A11 are all 1's.

$\overline{KVA}$ – Key Value Access select line (Input). This active low input enables access to the 3-bit Key Value register on the MMU. Reading the Key Value Register is allowed only when the current task is zero, address lines A11-A15 are all ones, $\overline{RA}$ = 0 (asserted), RS6-RS0 are within the range $40-$47 and $\overline{KVA}$ = 0 (also asserted). Writing the Key Value Register has the additional requirement of having the S-bit set.

$\overline{RESET}$ – $\overline{RESET}$ (Input). A low level on this input causes the MMU to initialize its registers to a known state. An internal flag is also set which forces $3FF onto the physical address lines until the Key Value Register is written. $\overline{RESET}$ must be low for at least one cycle.

MMU OPERATION

For every processor cycle, the MMU supplies a mapped address based on the processor address and the current task number (refer to Figure 4). The current task number is kept in an on-chip register called the OPERATE KEY. Changing the value of the operate key causes a new map to be selected.* The MMU also contains automatic task switching logic to cause pre-defined task numbers to override the task number in the operate key for certain events (Interrupts, Direct Memory Access, Reset).

The MMU registers always appear as a block of 64 bytes located on the last page of task #0 (refer to Figure 5). When the registers are accessed, the MMU outputs a physical address of $3FF (PA11-PA20 all high). This is necessary since the mapping RAM of the MMU cannot map an address and be modified at the same time.

The exact location of the MMU registers within the last page of physical memory is determined by the REGISTER ACCESS ($\overline{RA}$) signal which is similar to a chip select line. The $\overline{RA}$ signal will normally be derived from processor address lines A7-A10 using a simple 4-input gate. For example, a 4-input OR gate would place the MMU registers at $F800 to $F87F. In systems using DMA, the $\overline{RA}$ input must include the externally derived DMA/VMA signal to prevent dead bus cycles from affecting the MMU. Refer to Programming Considerations.

Inputs RS0-RS6 to the MMU are the register select lines. These lines are normally connected to the low order address lines A0-A6 from the processor. The MMU registers are only accessible if:

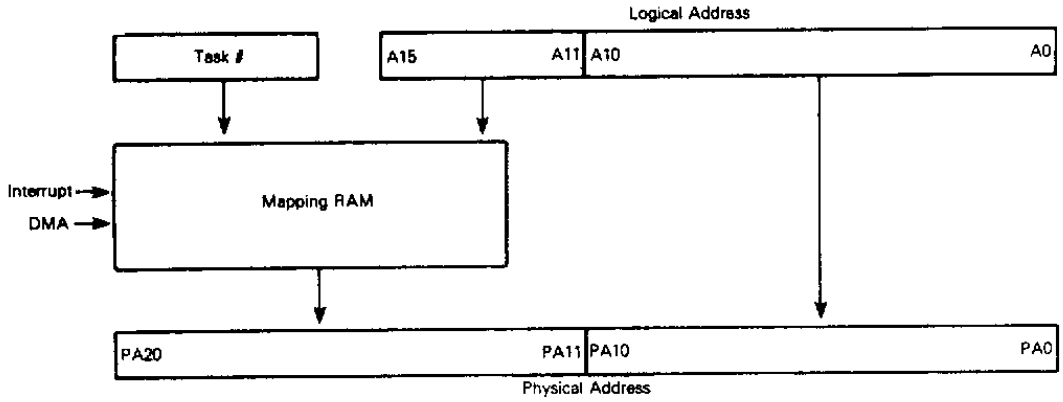
1. the current task number is zero;
2. processor address lines A11-A15 are all 1's;
3. the Register Access line ($\overline{RA}$) is asserted low;
4. Register Select lines (RS0-RS6) contain a defined register address; and
5. the System Bit (S-bit) is set (for a write operation only).

As a result of the above restrictions on accessing the MMU registers, the portion of the software that sets up and maintains the memory maps for all tasks must run as task zero.

The first 64 bytes of the MMU's register area comprise a "window" through which any one of the 4 maps may be viewed or changed. The task number to be viewed through this "window" is written into a read/write register called the ACCESS KEY. Thus, to examine or change the map for any task, the processor must first write the task number into the Access Key. Once set, the Access Key will retain its value until explicitly changed.

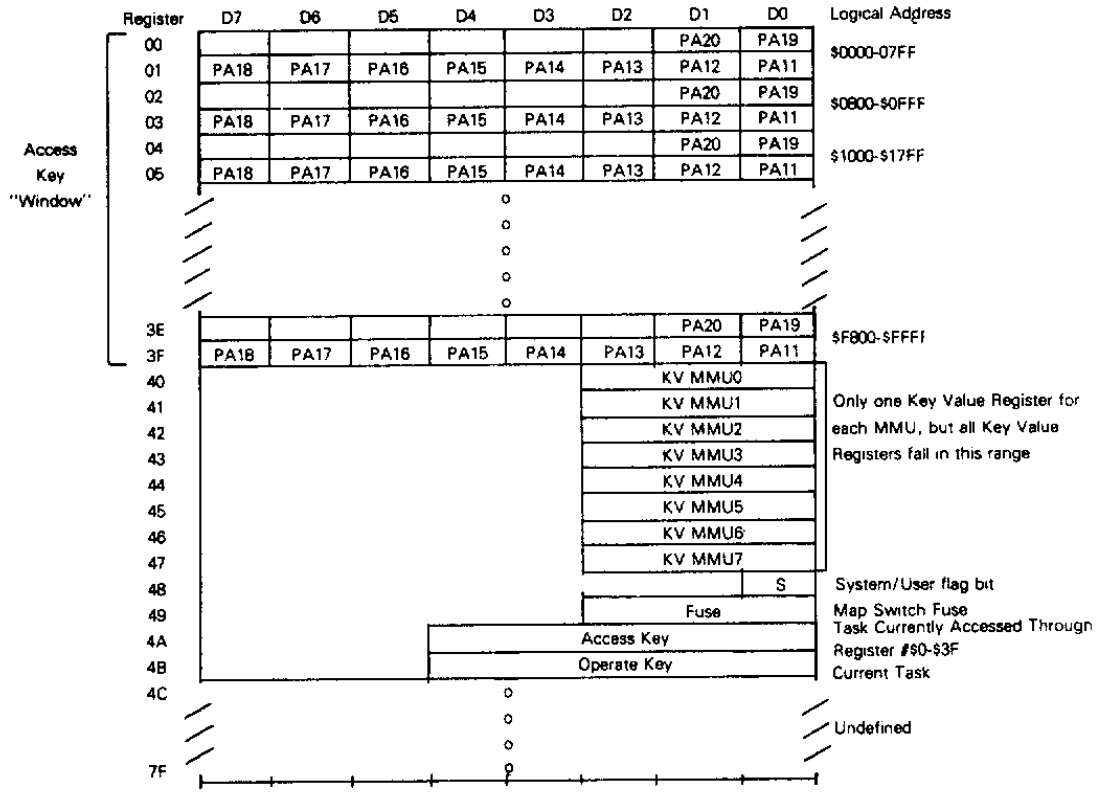
*Refer to Register Select Truth Table for exact procedure to change this register.

FIGURE 4 — LOGIC-TO-PHYSICAL ADDRESS TRANSLATION DIAGRAM



4

FIGURE 5 — MMU REGISTER MODEL



Notes:

1. The contents of bytes $4C through $7F are undefined and do not respond to any reads or writes.
2. The Access, Operate and Key Value Registers are cleared on reset. The S-bit is set.
3. Unused bits of defined registers always read zeros.
4. Locations $40-$47 are accessible only when $\overline{KVA} = 0$.
5. In multiple MMU configurations, the MMU whose Key Value Register matches the upper three bits of the access key will respond to a processor read of locations $48-$4B. Processor writes to these registers will cause the data to be written to all MMUs simultaneously.

Pages in physical memory require 10 bits to define their location (refer to Figure 5). These 10 bits are arranged as a pair of bytes in the MMU in order to allow the use of double byte instructions (e.g., LDD) in manipulating the MMU registers. These first 64 bytes of the register area are then accessed as 32 pairs of bytes with each pair describing the logical-to-physical mapping for one 2K page. Registers 0 and 1 contain the page number for logical addresses $0000-$07FF, register 2 and 3 control logical addresses $0800-$0FFF, etc.

Each MMU has a 3-bit register called the KEY VALUE REGISTER. This register determines the range of task numbers an MMU controls. The top three bits of the Operate Key must match the Key Value Register for that task to be active. Similarly, the Key Value Register must match the top three bits of the Access Key to change or view registers #0 through #$3F. Each MMU must receive a unique key value when the system is initialized to guarantee that no two MMUs control the same range of tasks. To be able to write to each MMU's Key Value Register separately, an external decoder must be provided. This decode function can be derived from address lines A0, A1 and A2 using a 3-to-8 line decoder. Writing to locations $40-$47 will cause the Key Value of the MMU to be updated only if the $\overline{KVA}$ input is low. In systems using a single MMU, the $\overline{KVA}$ input may be wired low.

BUILDING AN MMU SYSTEM

Up to 8 chips may be connected in parallel to create a maximum of 32 tasks. All MMU pins except one ($\overline{KVA}$) may be wired in parallel. Each MMU chip contains 1280 bits of fast on-chip lookup RAM. This RAM is accessible 10 bits at a time for mapping purposes, and as 2 and 8 bits at a time when the Operating System OS is changing the contents of the RAM. In addition to the lookup RAM, each MMU contains a separate copy of the Access Key, Operate Key, Fuse Register, Key Value Register, and S-bit. A CPU write to the Access, Operate, or Fuse Register causes all registers on all MMUs to be updated. In contrast, the lookup RAM for each chip is updated only when the top three bits of the Access Key match the Key Value Register for that chip. During mapping operations, each MMU compares the value in its Operate Key (top three bits) with its Key Value Register and responds only if a match is found. Similarly, when the processor reads the RAM, each MMU compares its Key value with the Access Key (Figure 6).

REGISTER SELECT TRUTH TABLE

Table 1 shows how the MMU registers are accessed by the processor. It is assumed that the current task is zero and that the processor address lines A11-A15 are all ones. If the S-bit is not set, the registers are still readable, but cannot be modified.

4

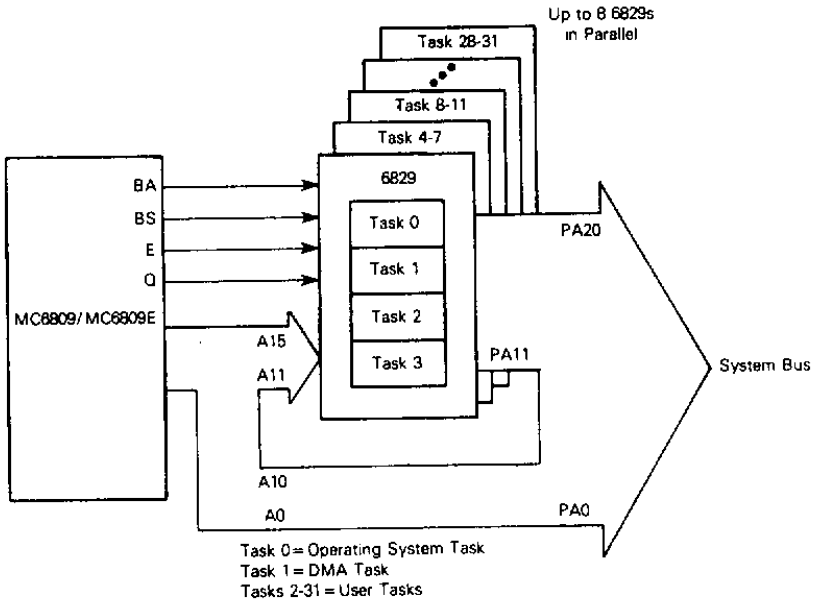
TABLE 1 — REGISTER SELECT TRUTH TABLE

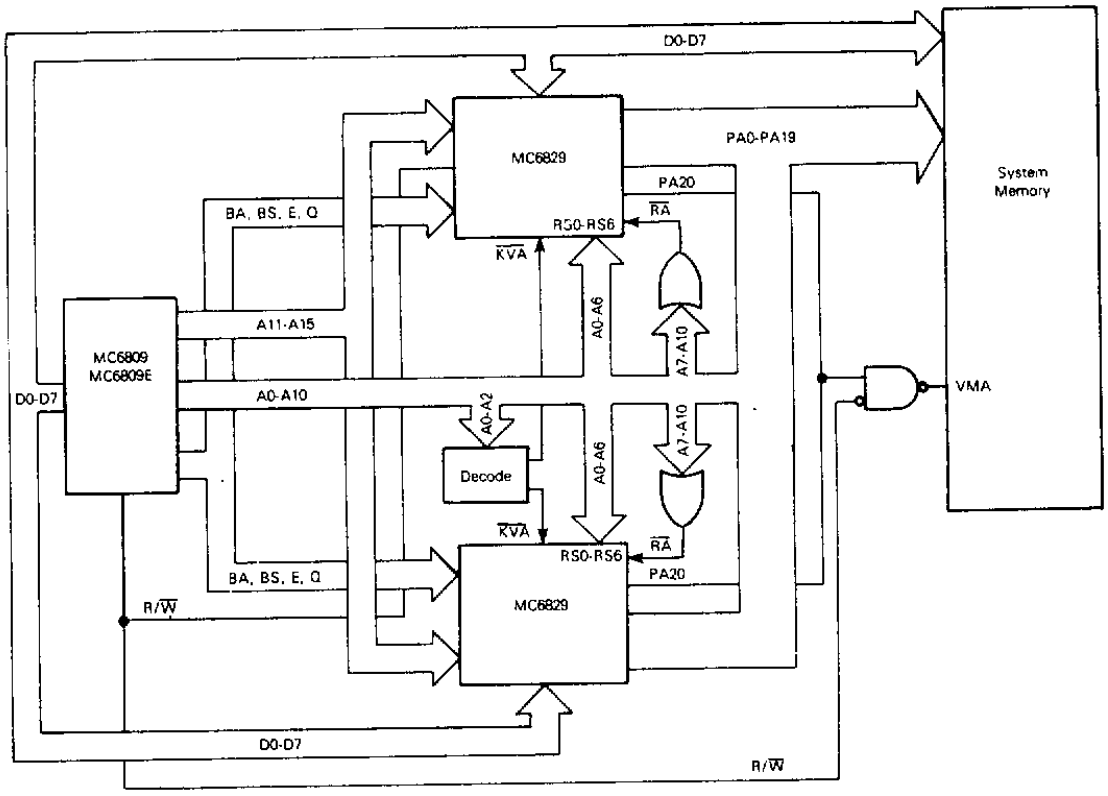
$\overline{RA}$	R/$\overline{W}$	$\overline{KVA}$	RS6	RS5	RS4	RS3	RS2	RS1	RS0	register addressed
1	X	X	X	X	X	X	X	X	X	none
0	X	1	1	0	0	0	X	X	X	none
0	1	0	1	0	0	0	X	X	X	read Key Value Register
0	0	0	1	0	0	0	X	X	X	write Key Value Register
0	X	X	0	n	n	n	n	n	n	byte nnnnnn of MMU RAM (Note 1)
0	0	X	1	0	0	1	0	0	0	none (Note 2)
0	0	X	1	0	0	1	0	0	1	write Fuse Register
0	0	X	1	0	0	1	0	1	0	write Access Key
0	0	X	1	0	0	1	0	1	1	write Operate Key
0	1	X	1	0	0	1	0	0	0	read S-bit (Note 3)
0	1	X	1	0	0	1	0	0	1	read Fuse Register (Note 3)
0	1	X	1	0	0	1	0	1	0	read Access Key (Note 3)
0	1	X	1	0	0	1	0	1	1	read Operate Key (Note 3)
0	X	X	1	0	0	1	1	X	X	none
0	X	X	1	0	1	X	X	X	X	none
0	X	X	1	1	X	X	X	X	X	none

Notes:

1. The MMU RAM is accessible only if the Key Value Register is equal to the top 3 bits of the Access Key Register. The lower two bits of the Access Key Register then determines which task is to be accessed (R/$\overline{W}$).
2. The S-bit is read-only.
3. The S-bit, Fuse, Access or Operate registers are readable only if the Key Value Register is equal to the top 3 bits of the Access Key Register. This insures that only one MMU will respond to a read request of these locations.

FIGURE 6 — MMU SYSTEMS CONFIGURATION





REGISTER DESCRIPTION

System Bit (S-bit) – Read-only bit that must be set (S = 1) to write MMU registers. Reset and Interrupts set the S-bit. Refer to Fuse Register for clearing the S-bit.

Operate Key – 5-bit R/$\overline{W}$ register that contains the current task number. The operate key retains its value until explicitly changed. During DMA transfers, the MMU overrides the value in the operate key and forces task #1 to be the active task. When the S-bit is set, the operate key is also overridden, and task #0 is forced to be the active key.

Key Value – 3-bit R/$\overline{W}$ register that contains the range of tasks an MMU controls. The Key Value Register must match the top three bits of the Operate Key for a task to be active. The $\overline{KVA}$ signal must be low for an access of this register.

Access Key – 5-bit R/$\overline{W}$ register that contains the task number of a task to be viewed or changed. This register retains its value until explicitly changed.

Register #0 to #3F – 64 bytes accessed as 32 pairs of bytes with each pair describing the logical to physical mapping for one 2K page. Refer to Figure 5.

Fuse Register – 3-bit count down register used to change from task #0 to a user task. When a write to this register is detected, the value written is loaded into the counter and it begins to decrement by one for every processor cycle. When the counter underflows, the S-bit is cleared and the next processor cycle will be mapped using the task number in the operate key.

RESET OPERATION

When reset, the MMU performs the following operations:

1. The Key Value Register is cleared;
2. The Fuse Register is disabled;
3. The System bit (S-bit) is set,
4. The Operate Key Register is cleared;
5. The Access Key Register is cleared,
6. An internal reset flag is set.

Reset causes the MC6829 to automatically switch the memory map to task #0. An internal flag is set causing all bus cycles to access physical addresses $1FFB00-$1FFFFF (PA11 to PA20 all high, page $3FF). This flag is cleared when the Key Value Register is first written. While the internal reset flag is set, each MMU in the system will be actively driving the address bus. An orderly start up procedure must assign each MMU a key value before individual task allocations are made.

FUSE REGISTER OPERATION

The Fuse Register is a 3-bit register used to switch from task #0 to any other task. A write to this register causes an internal 3-bit counter to be loaded with the data. On each successive valid (non-DMA) processor cycle the internal counter is decremented once. When the counter reaches zero, the task number in the Operate Key will be the active task, mapping logical to physical address. The value written into the Fuse Register must be the number of cycles it takes to transfer program control from the store to Fuse Register instruction. It is the responsibility of the Operating System (task #0) to make sure the processor will execute code from the new task properly by changing the Program Counter the same cycle that the Fuse Register reaches zero (see following example).

Change from Task #0 to Task n

```
LDA #n
STA OPERATE
LDA #4
STA FUSE
JMP $XXXX
```

Cycle by Cycle Operation	Write #4 to Fuse Register	JMP	Address High	Address Low	$\overline{VMA}$	Task N Opcode
Fuse Register Contents	0	4	3	2	1	0

Refer to Section *MMU in a MC6809 System* for Fuse Register use in returning from an interrupt.

MMU INITIALIZATION PROCEDURE

The following steps should be followed to initialize a multiple MMU system. (Refer to Hardware/Programming Considerations; Programming Examples section.)

1. Out of Reset, all MMUs are driving the address lines, PA11 to PA20, high. This requires the initialization program to be located in this 2K byte page of physical memory. Each MMU must be deselected by writing a unique value to its Key Value Register except for the MMU that will run task #0 (MMU0). MMU0's Key Value Register must not be written to until task #0 registers $00 to $3F are programmed, specifying the logical to physical mapping of memory. In addition, if MMU0 Key Value Register is also initialized with a non-zero value at this time the entire memory space is deselected and the operating system (task #0) cannot be accessed (Example 1).
2. Only one MMU is now driving the address bus. Task #0 memory pages (2K per page) must be assigned by writing the corresponding values into registers $00 to $3F (Example 2).
3. The Key Value Register must be written to MMU0's key value to allow initialization of all other tasks by removal of automatic mapping of PA11 to PA20 high (Example 2).
4. At this time, each MMU has a unique key value, Task #0 has a specified memory map, and Task #0 is operating. Tasks can now be started by writing the task number to be specified in the Access Key Register, writing registers $00 to $3F to the memory map desired, loading the program into memory and causing a task switch by a correct use of the Fuse Register.

INTERRUPTS/MAP SWITCHING

The MC6829 monitors the Bus Available (BA) and Bus Status (BS) lines from the processor to determine what type of bus operation is occurring. When an interrupt is detected, the current task is overridden by Task #0. The map switch occurs during the processor vector fetch (BA = 0, BS = 1) so that Task #0 supplies the interrupt vector address. Detecting an interrupt also sets the S-bit within the MMU allowing Task #0 to be the operating task while the interrupt is serviced.

DMA OPERATION

For a DMA transfer, the memory map is switched to Task #1. This allows transfers of up to 64K bytes without processor intervention and without interfering with any other task. (An external DMA/$\overline{VMA}$ signal should be included in the decode circuitry for the $\overline{BA}$ input to prevent dead bus cycles from affecting the MMU). At the end of the DMA transfer, the MC6829 returns to the task being used before the transfer began (refer to Programming Considerations).

MMU IN A MC6809 SYSTEM

The MC6829 is designed to work directly with the MC6809 processor. Other 8-bit microcomputers may also use the MMU by generating the appropriate inputs to the MMU. The crucial area for interfacing the computer to the MMU is the design of the map switching hardware.

For the MC6809, the BA and BS signals are extremely useful for this function. Decoding these two signals provides the following information:

BA	BS	MC6809 State
0	0	Normal (running) mode
0	1	Interrupt Acknowledge (IACK)
1	0	SYNC Acknowledge
1	1	HALT or Bus Grant

The MMU uses these two signals directly from the processor to determine what action to take for every bus cycle.

The MMU, unlike other M6800 peripherals, introduces an additional delay (t_{MAD}) in the system configuration as it accepts address signals from the MPU and maps the MC6809 logical address to the system physical address. When a system is constructed this additional delay must be considered.

The system clock frequency is determined by these address timing delays. Figure 7 shows this data. The System Cycle time may be determined by adding:

1. the MPU E to Q rise delay t_{AVQ} (max)
2. the MPU address valid to Q rise to t_{AQ} (min)
3. the MMU mapping delay t_{MAD} (max)
4. the system decode and buffer time t_B (this is the delay due to bus buffers and decoding circuitry)
5. the address setup time required by peripherals t_{AS} (note the setup time is required for the peripheral to determine if it is selected as well as deselected during every bus cycle).
6. the MPU pulse width high t_{PWEH}.

NOTE

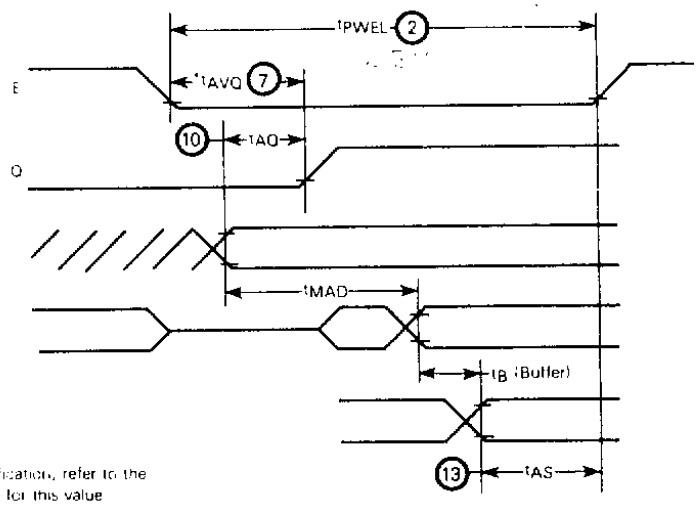
This equation must be satisfied:

$t_{PWEL} \geq t_{AVQ} - t_{AQ} + t_{MAD} + t_B + t_{AS}$

DMA OPERATION — By decoding the bus grant signal (BA = 1, BS = 1), the MMU will automatically switch to Task #1. Even when the MC6809 occasionally steals back a cycle to refresh its internal buses, this is reflected by a change in the bus grant signal which causes the map to temporarily switch back to the normal running mode.

Note that the bus grant status is identical to the Halt status and is thus indistinguishable from a HALT. This should not cause a problem since halting the processor will simply cause the MMU to switch to Task #1. When the MC6809 starts to run again, the status lines will change and cause the MMU to switch to the proper map.

FIGURE 7 — ADDRESS DELAY



*As per MPU specification, refer to the MC6809 Data Sheet for this value

CHANGING TASK TO OPERATING SYSTEM (OS) – The OS map (Task #0) is automatically selected to service all interrupts. The Interrupt Acknowledge (IACK; BA = 0, BS = 1) signal is used to determine when an interrupt vector is being fetched. The map is switched at this time in order to supply the processor with an interrupt vector from the OS address space, not the user's. At the time IACK is asserted, all of the registers have been stacked for the interrupt in the user's address map. This means that the only information the OS needs to save concerning the running process is its stack pointer. All other information about the task is saved on the user's stack and in the MMU registers. The map switch is latched since IACK will only be present for two machine cycles, yet the OS must retain control until the interrupt is serviced. This latched information is kept in a flag register called the S-bit. This bit is set on any IACK and remains set until cleared by software. The first thing the OS must do is save the interrupted task's stack pointer in a table and load the stack pointer with the current top of stack in the OS map. This is a critical section of code and must not be interrupted. For this reason, an MMU system cannot accept two interrupts in a row. The first interrupt causes the map to switch to task zero. The second interrupt would stack the machine state at the wrong address in the operating system. As a consequence of this, Non-Maskable Interrupts (NMI) must be forbidden in multi-tasking systems since an NMI is possible at any time (even during another interrupt). Similarly, normal interrupts (IRQ) do not set the Fast Interrupt (FIRQ), bit F of the status register, in the processor and, thus, potentially allow another interrupt before the processor has a chance to switch stack pointers. Simple external hardware can be used to disable FIRQ when IRQ is pending. Unlike the NMI input, the FIRQ input is level sensitive and may be masked with external hardware during IRQ operations.

A typical interrupt service routine begins like this:

```
ORCC    #1+F
STS     SAVESP
LDS     OSSP
```

RETURNING FROM THE OS TO TASK N – The OS must execute an RTI instruction to get the processor to reload the user registers. The map switch must occur after the opcode for the RTI is fetched and before the first register is pulled from the stack. Prior to the RTI, the OS must reload the stack pointer from the one that corresponds to the task about to run. There must be no interrupts from the time the stack pointer is reloaded until the RTI is executed. The signal to the MMU that the map should be returned to the user task is noted by a write to a 3-bit down counter called the FUSE REGISTER. When a write to this register is detected, the value written is loaded into the counter and it begins to decrement by one for every processor cycle. When the counter under flows, the S-bit is cleared and the next processor cycle will be mapped using the task number in the Operate Key. For most systems, a 1 would be written to the Fuse Register immediately before the RTI opcode is executed. Note that DMA operations are still possible within this critical section. The Fuse Register counts only non-DMA cycles after the write to the Fuse Register in order to be sure of when to switch the map. Bus dead cycles are also excluded when clocking the Fuse Register. Thus, the Fuse Register is inhibited from counting whenever BA is high, and for the cycle after BA transitions from high to low. The common exit point for all OS functions looks something like this:

```
EXIT    LDA     TASK    GET NEXT TASK TO RUN
        STA     OPERAT  AND PLACE IT IN THE OPERATE KEY
        STS     OSSP    SAVE CURRENT STACK POINTER
        ORCC    #F+I    SET F AND I (ENTER CRITICAL SECTION)
        LDS     SAVESP  RESTORE USER'S STACK POINTER
        LDA     #1      CAUSE MAP SWITCH 1 CYCLE AFTER
        STA     FUSE    WRITE TO FUSE REGISTER
        RTI             RETURN TO USER TASK
        •
        •
        •               MAP SWITCH OCCURS, USER TASK RESUMES
```

USING THE MC6800

When using a MC6800 processor external logic is required to determine when to switch maps. The MMU is controlled by its BA, BS inputs, the S-bit and the Operate Key. For example, decoding any references to the interrupt vectors and generating IACK as a result will work as long as each task references these locations only when the processor is fetching an interrupt vector. Another possibility is to monitor the processor R/W line. For the MC6800, the only time seven writes occur in a row is during an interrupt sequence. Thus the external logic that generates BA and BS must wait until it sees the seven writes and then assert IACK for the next two cycles.

A MC6800 processor interface to the MMU must also include logic to generate the Q bus signal.

HARDWARE/PROGRAMMING CONSIDERATIONS

4

The following sections contain examples and suggestions on how to apply the MMU in a system.

MEMORY PROTECTION – The MMU can provide memory protection on a per page basis by defining the high order physical address line (PA20) as a write access line. If write protection is desired, this signal can be gated with the read/write line, from the processor, to generate a disable signal. This can be used to inhibit the memory chip select logic or generate an interrupt to signal a violation of a write protected area. The write protect line can also be combined with the DMA/VMA logic that is necessary in systems using DMA. In this case, writes to protected memory would appear as dead cycles to the main memory. Note that the designtion of the write protect line is purely arbitrary. The MMU simply combines the incoming address with the current task number to determine a 10-bit result. If no write protection is needed, PA20 can be used as a 21st address line, giving a total addressing range of 2 Megabyte. This scheme can be reversed if desired and additional output lines from the MMU can be used to specify more attributes of the physical pages at the expense of reducing the number of pages in physical memory.

MANAGING INTERRUPTS – An interrupt causes the processor to suspend the current running task and perform a service routine for the interrupting device. User programs should not have to handle interrupts directly. Thus on interrupts, the MMU (the operating system OS) must switch from the current map to task 0 so that it can handle the interrupt. (The OS may of course elect to pass the work of handling a specific interrupt to a task that is expecting it.) The map switching is latched (indicated by the S-bit) so that the processor has as much time as it needs to service the interrupt. After the interrupt has been processed, the OS can then look at the current process priorities and determine the next process to run. If, after the interrupt service, the task that was running before the interrupt is to continue to run, the OS causes the map to switch back to that task. If, however, another task is to start running, the OS can simply write the new task number into the Operate Key Register and then cause the map switch. Returning to the normal map clears the S-bit and allows the user process to continue. By supplying a source of periodic interrupts, the OS can regain control of the processor and reschedule running processes.

Operating system requests for privileged operations by running tasks are ideally handled using the SWI instruction. This causes a map switch to task zero (IACK is asserted on SWI) which then processes the request and eventually returns control to the requesting task. Note that SWI sets the I and F bits during execution of the instruction so that when the OS is entered, the critical section of saving the user task pointer and reloading the OS stack pointer can be safely executed. Note that SWI2 and SWI3 do not have this property and therefore require special handling. To safely use SWI2 or SWI3, the programmer must explicitly mask hardware interrupts.

```
ORCC    #I+F        DISABLE INTERRUPTS
SWI2/3              CALL OS
```

MANAGING NON-EXISTENT MEMORY ACCESSES – Memory accesses to non-existent memory requires careful consideration. Once an instruction has begun execution, there is no way to stop it from completing. Thus, an instruction may reference a non-existent memory location, or an interrupt may cause the machine state to be stacked into non-existent memory. Once this has occurred, there is not always enough information available to backtrack the last instruction.

One solution to this problem is a hardware FIFO. When a task is initialized, a certain number of pages will be assigned from available memory. For example, a ROM program could be placed in a task's map along with RAM for stack and variable data areas. The remaining pages in the task's map are unassigned and references to these unassigned areas require special handling. These gaps in the memory map of a task may be filled by constructing a "FIFO page" that returns a known value when read (zero) and when written saves the (logical) address and the data written to it. If at any time the FIFO is not empty, the FIFO causes an interrupt at the end of the current instruction. The processor then examines the contents of the FIFO and allocates real pages where there were none before. The data in the FIFO is then placed in real memory and the task may resume execution. Thus, the program is stopped at the end of the instruction that causes a page fault, and all writes to non-existent memory are captured in the FIFO.

The maximum number of new pages that may be required after any page fault is four. Consider the following instruction sequence. A task has just started running and has only one page allocated to it ($0000-$1FFF). The program to be executed is as follows:

```
ORG     $0000       PROGRAM START ADDRESS
LDS     #$8000      INITIALIZE STACK
LDX     #$3FFF      POINT TO DATA AREA
LDD     #$1234
STD     ,X          INITIALIZE VARIABLE
```

Execution then proceeds as follows. Upon executing the fourth instruction, two bytes are written, one at location $3FFF and the other at $4000. Since neither of these two pages actually exist, the FIFO catches the address and data written and pulls the IRQ line to signal a page fault. At the

end of the STD instruction, the processor will stack the machine registers which causes two further page faults since the stacking operation writes data to locations $7FF5-$8000. The FIFO must also catch these references since they contain the machine state at the time of the original interrupt. When task zero gains control, the FIFO data must be cleared before any attempt is made to reference the task's memory map. If there are no available pages, the task may be made inactive until sufficient space exists to allow the program to continue.

The maximum number of bytes that may be written to non-existent memory before task zero gains control is 24. This occurs when the task pushes all of its registers onto the stack when the stack points to an uninitialized page. Pushing all registers requires 12 bytes. At the end of the instruction, an interrupt will be generated which again pushes the entire machine state. Thus, the FIFO must be 24 bits wide (16 address + 8 data lines) and 24 words deep.

The primary benefit of this scheme is to allow the MC6809 stack to grow dynamically. When a task starts to run, the stack could be initialized to $FFFF with no real memory at that location. When the task did its first subroutine call or stack push, the FIFO interrupt would catch the information and the operating system would then allocate memory. If the task never used this area, it would remain unallocated and thus be available for other uses. Note that this approach provides for dynamic memory expansion of growing data areas. If the size of the static data areas is known at load-time, then memory can be allocated to a task as needed. Heap management (such as for an editor buffer) can be handled by task resident memory allocation routines which make operating system calls to obtain more heap space.

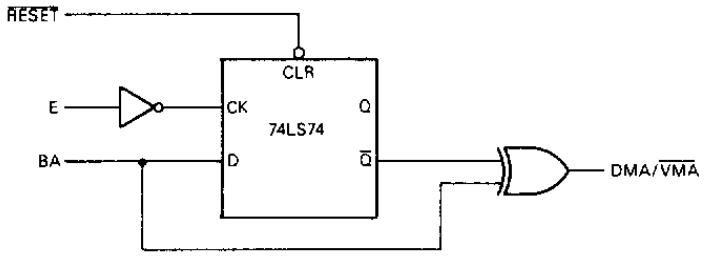
The FIFO scheme does not implement a demand paging system. It is assumed that once a page has been assigned to a task the page remains assigned until the task ends execution or possibly gives it back (via a system call) to the operating system.

DMA/$\overline{\text{VMA}}$ CIRCUIT

The following circuit, Figure 8 , is suggested to keep the MC6829 deselected during dead bus cycles of DMA. This circuit will also work in a non-MMU system.

4

FIGURE 8 — M6809 DMA/$\overline{\text{VMA}}$ LOGIC



COMMON MMU EQUATES

Here is a list of assembler equates that are used in the following examples:

```
MMU      EQU    $F800       START OF MMU REGISTERS (IN TASK 0)
MMU0     EQU    MMU+$40     FIRST MMU'S KEY VALUE REGISTER
MMU7     EQU    MMU+$47     LAST MMU'S KEY VALUE REGISTER
SBIT     EQU    MMU+$48     SYSTEM/USER FLAG BIT
FUSE     EQU    MMU+$49     MAP SWITCH COUNT-DOWN REGISTER
ACCESS   EQU    MMU+$4A     ACCESS.KEY
OPERAT   EQU    MMU+$4B     OPERATE KEY
NTASK    EQU    32          NUMBER OF TASKS IN SYSTEM
NPAGE    EQU    32          NUMBER OF PAGES PER TASK
MAXPGE   EQU    $400        MAXIMUM NUMBER OF PAGES IN SYSTEM
PSIZE    EQU    2048        NUMBER OF BYTES IN A PAGE
```

Programming Examples

Example #1 — **Only works if $\overline{KVA}$ is fully decoded**

Write a program to initialize all MMU Key Value Registers except MMU0.

```
*
*           RESET ENTRY POINT FOR MMU SYSTEM
*
            LDX     #MMU7+1         POINT TO LAST MMU KEY VALUE REGISTER +1
            LDA     #7              INITIALIZE VALUE
KVINIT      STA     ,-X
            DECA
            BNE     KVINIT
            *
            *       CONTINUE INITIALIZATION
            *
```

At this point, each MMU will have a unique key value. Note that the Key Value Register for MMU0 has not yet been written so that page $3FF is still on the physical address bus. The difference is that now only one MMU is driving the address bus.

4

Example #2 —

Write an initialization program that sets up the pages of Task #0 so that an address $XXXX in Task #0 corresponds to physical address $1FXXXX.

```
            *
            *       FROM KEY VALUE INITIALIZATION
            *
*           NOW INITIALIZE IDENTITY MAP FOR TASK 0
*
            CLR     ACCESS      TALK TO TASK 0 (ALREADY ZERO ANYWAY)
            LDX     #MMU
            LDD     #$3E0       LAST PAGE -32
MOINIT      STD     ,X++
            INCB                QUIT WHEN D=$200
            BNE     MOINIT
            CLR     MMU0        LET MMU #0 GO
            JMP     EXBUG       TRANSFER TO MONITOR (EXBUG09)
```

Example #3 —

Give task #9 physical page #88 and place it in the task's address space so that #9 refers to this page with addresses $1000-$17FF. Write protect this page for this task. (The write protect bit is defined as PA20 of the MMU.)

```
PROTEC      EQU     $200        WRITE PROTECT BIT POSITION (PA20)
            *
            *
            *
            LDA     #9          SELECT TASK #9 FOR
            STA     ACCESS      MODIFICATION
            LDX     #88+PROTEC  WRITE PHYSICAL PAGE INTO
            STX     MMU+4       THE APPROPRIATE REGISTER
            *
            *
```

Example #4 —

Write a subroutine that reads a byte from any task. On entry, the A register contains the task number, and the X register contains the address of that task to read. Assume that the OS task has its third page free for this use. The byte that is read is returned in A.

```
FPAGE       EQU     $1000       DEDICATED FREE PAGE
FREE        EQU     4           OFFSET INTO MMU OF FPAGE
*
*           FUBYTE — FETCH USER BYTE
*
FUBYTE      LBSR    GETPAGE     POINT TO PAGE
            LDA     ,X          PICKUP BYTE
            RTS
```

Example #5 –

Write a subroutine that writes a byte to any task. On entry the A register contains the task number and the X register contains the address of that task to read. The B register contains the byte to place in the task's memory. Assume that the OS task has its third page free for this use.

```
*
*           SUBYTE – SET USER BYTE
*
SUBYTE      LBSR        GETPAGE PLACE USER PAGE IN FPAGE
            STB         ,X
            RTS
```

Example #6 –

Write a subroutine to be given a task number and memory address that returns a pointer to that byte of the named task. On entry, the A register contains the task number and the X register contains the task address.

```
* GET PAGE – POINT TO USER BYTE
* Given a task number in A and a task address in X,
* return with X pointing to that byte in task 0.
* This subroutine assumes that task 0 has a free
* page (FPAGE) that it uses to map a page of the
* specified task into task 0's map.
*
*
GETPAGE     PSHS        D, Y            SAVE SOME REGISTERS
            STA         ACCESS          SETUP WINDOW TO TASK
            TFR         X, D            MOVE POINTER INTO ACCUMULATOR
            ASRA                        FIND PHYSICAL PAGE #
            ASRA
            ANDA        #%00111110      MASK ALL BUT PAGE #
            LDY         #MMU
            LDY         A, Y            PICKUP PAGE
            CLR         ACCESS          NOW TALK TO OS MAP
            STY         MMU+FREE        'FREE' OS PAGE
            TFR         X, D            NOW POINT TO OFFSET
            ANDA        #%111           MASK HIGH BITS OF ADDRESS
            LDX         #FPAGE          POINT TO PAGE START
            LEAX        D, X            ADD OFFSET
            PULS        D, Y, PC        RESTORE AND RETURN
```

The above method of fetching bytes from other tasks is appropriate where only a few bytes of memory are to be transferred. When larger amounts of memory are to be moved, a more general subroutine can be written that transfers up to 2K bytes (one page) before the MMU registers need to be changed.