Chapter 1

Chapter 1: QED-Flash Board and Wildcard Carrier Board Memory Maps

The QED-4 (QED-Flash) Controller's Memory Map

The QED-4 Controller (also called the QED-Flash Board) uses a paged memory system to expand the processor's 64Kbyte address space to 8 Megabytes of addressable memory. The top half (32 Kbytes) of the address space (at addresses 0x8000 to 0xFFFF) addresses a common memory page that is always visible (i.e., accessible using standard 16-bit addresses) to any code running, no matter where it resides in the memory space. The bottom half (32 Kbytes) of the address space (at addresses 0x0000 to 0x7FFF) is duplicated many times and addressed through the processor's 16-bit address bus augmented by an 8-bit page address. Together the address and page are held in a 32-bit data type, an *xaddress*.

A subroutine on any page can fetch or store to any address on the same page or in the common memory, or transfer control to another routine there. It "sees" a 64K address space comprising its own page at addresses from 0x0000 to 0x7FFF and the common memory at addresses 0x8000 to 0xFFFF. To address memory on another page, or to call a routine on another page, special memory access routines are used to change the page. The heap memory manager and array routines allow you to think of the paged memory as contiguous memory for data storage. The operating system automatically handles function calls and returns among the pages.

The operating system is designed so that there is very little speed penalty associated with changing pages. The QED-Forth operating system automatically and transparently handles page changes.

Figure 1-1 illustrates the memory map of the QED Board. Briefly, the upper 32K of the 68HC11's address space, the *common memory*, is always accessible without a page change. In the lower 32K of the processor's address space, the operating system creates 256 pages of memory selected by an 8 bit on-chip port, with each page containing 32 Kbytes. The 32K of common memory at addresses 0x8000 to 0xFFFF (the upper half of the processor's memory space) is always accessible without a page change. Up to 256 pages (32K per page) occupy the paged memory at addresses 0x0000 to 0x7FFF. All pages of the QED-4 Controller's installed memory are shown in the figure.

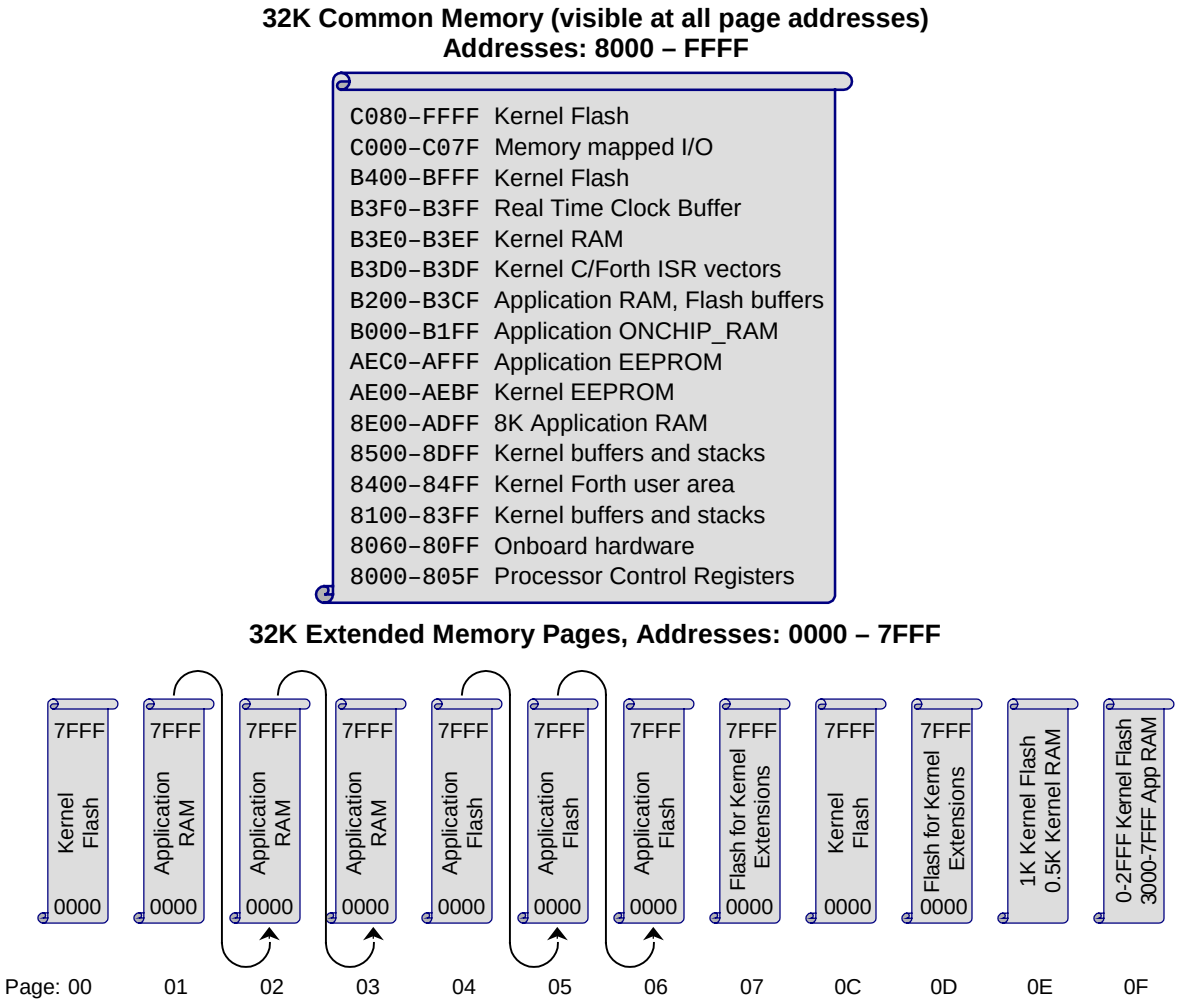

Figure 1-1 The paged memory space of the QED-Flash Board.

Common Memory

The *common memory* is addressed at locations 0x8000 – 0xFFFF. Some of it is used by the 68HC11 processor, some by the Forth kernel, and some is available for your programs to use. The processor's registers are located at 0x8000 – 0x805F, onboard hardware occupies addresses through 0x80FF, and the operating system reserves memory through location 0x8DFF for user areas, buffers, and stacks. For example, the default user area that runs the interactive Forth interpreter occupies 0x8400 – 0x84FF.

The 8 Kbytes at locations 0x8E00 – 0xADFF are available RAM for the user. The Control-C compiler uses this area for static variables, arrays, task areas, etc. The Forth memory map routine USE.PAGE locates the variable area starting at address 8E00 in common memory. If you need more than 8K for variables you will need to move the variable pointer, VP, to another section of RAM. RAM pages 01-03 are available for application variables, buffers or other storage.

The processor's on-chip EEPROM (Electrically Erasable Programmable Read Only Memory) is located at 0xAE00 – 0xAFFF. The first few locations, at 0xAE00 – 0xAEBF, are reserved by the

operating system for use by the **SAVE** and **RESTORE** utilities, and for interrupt vectors. The EEPROM locations at 0xAEC0 – 0xAFFF is available to your programs.

Table 1-1 Partition of Flash and RAM among Kernel and Application Functions on the QED-Flash Board

Function	Size	Memory Page	Memory Address	Physical Location
Kernel Flash (96K)				
Kernel	64K	00, 0C	0000 – 7FFF	QED Board S1
Kernel	1K	0E	various	QED Board S1
Kernel	19K	common	B400 – FFFF	QED Board S1
Kernel	12K	0F	0000 – 2FFF	QED Board S1
Flash for Kernel Extensions (64K)				
Kernel Extensions	64K	07, 0D	0000 –7FFF	QED Board S1
Kernel RAM (4K)				
Kernel	3.5K	common	8000 – 8DFF	QED Board S2
Kernel	0.5K	0E	various	QED Board S2
RTC	48 bytes		B3D0– B3FF	68HC11
Kernel EEPROM (192 bytes)				
Kernel	192 bytes	common	AE00 – AEBF	68HC11
Application Flash (96K)				
Application code and graphic images	96K	04 – 06 (01– 03)	0000 – 7FFF	QED Board S1
Application RAM (125K)				
C Variables	512 bytes	common	B000 – B1FF	68HC11
C Variables, available at runtime but used during download as a Flash write buffer	464 bytes	common	B200 – B3CF	68HC11
C Variables and task area, optionally battery-backed	8K	common	8E00 – ADFF	QED Board S2
Arrays and heap memory, optionally battery-backed	20K	0F	3000 – 7FFF	QED Board S2
Arrays and heap memory, optionally battery-backed	96K	01 – 03 (04 – 06)	0000 – 7FFF	QED Board S2
Application EEPROM				
EEPROM Variables	320 bytes	common	AEC0 – AFFF	68HC11

Notes:

1. Pages not enclosed in parentheses indicate the *standard*, or run-time memory map; pages in parentheses indicate the addressing of the memory during program download, i.e., the *download* memory map.
2. The 128K RAM in socket S2 on the QED-Flash Board can be optionally battery-backed.
3. Pages 00, 0C, 0E and a portion of 0F are reserved to the operating system (the Kernel).
4. Pages 07 and 0D are flash that is available for downloading kernel extensions.
5. Your application code is free to reside on pages 04-06 and you may use pages 01-03 for variables and buffers.
6. Addresses from 8000 through FFFF comprise common memory that is visible to code on all pages.

Table 1-2 Locations of Flash and RAM on a Wildcard Carrier Board

Function	Size	Memory Page	Memory Address
Application Flash (128K or 512K)			
Application code and graphic images	128K	50 – 53 (40 – 43)	0000 – 7FFF
Application code and graphic images, available with extended memory option	384K	54 –5F (44 – 4F)	0000 – 7FFF
Application RAM (128K or 512K)			
Variables, arrays and heap memory	128K	40 – 43 (50 – 53)	0000 – 7FFF
Variables, arrays and heap memory, available with extended memory option	384K	44 – 4F (54 – 5F)	0000 – 7FFF

Notes:

1. Pages not enclosed in parentheses indicate the *standard*, or run-time memory map; pages in parentheses indicate the addressing of the memory during program download, i.e., the *download* memory map.

The 68HC11's 1 Kbyte of on-chip RAM is located at 0xB000 – 0xB3FF. Locations 0xB3F0 – 0xB3FF are reserved for the real-time clock buffers. Locations 0xB3D0-0xB3DF are reserved for support of Forth interrupt service routines called from C-compiled programs. Locations 0xB200 – 0xB3CF are reserved for the flash programming routines. Locations 0xB000 – 0xB1FF are always available to the programmer (this area is named `ONCHIP_RAM` in the C linker command file; C programmers can locate data in this area using a `#pragma` directive).

Locations 0xB400 – 0xBFFF and 0xC100 – 0xFFFF contain kernel code. A *notch* at 0xC000 – 0xC07F is not decoded by any onboard devices, and provides a convenient place for the user to memory map I/O that must be accessed quickly (that is, without requiring a page change). Of course, an almost limitless amount of I/O can be mapped onto pages in the controller's 8 Megabyte address space.

Paged Memory

Occupying the paged memory space are 256K Flash and 128K RAM. Of the board's 256K of Flash, 96K is available for your application program and data storage. The remainder is used by the QED Forth Kernel for its multitasking operating system, debugger, interactive Forth compiler, assembler, and hundreds of pre-coded device driver functions. Of the 128K of RAM, 96K of contiguous memory is available for application program use (for variables, buffers, or heaps), a 20K portion on page 0x0F is useful for placing a heap (but can be used for any other purpose too), and smaller portions may be used for variables. The entire RAM can be optionally battery backed.

Table 1-1 illustrates the partitioning of the onboard memory between the operating system (Kernel) and your application functions.

Frequently, a QED-Flash Board is used in conjunction with a Wildcard Carrier Board, which provides additional Flash and RAM. Table 1-2 shows the page locations of the RAM and Flash available on a Wildcard Carrier Board. The standard version of the WCB provides 128K each of RAM and Flash, while an extended memory version provides 512K of each.

Most of the Flash memory is available as blocks of contiguously addressable memory on pages 04-06 of the QED-Flash Board and pages 50-53 on the Wildcard Carrier Board. RAM for your application program is also available in the paged memory in contiguously addressable chunks, filling pages 01-03 on the QED-Flash Board and 40-43 on the Wildcard Carrier Board. There is also 20K available RAM on page 0F, and approximately 9K in the common memory. This 9K is particularly important because it is used to hold C variables and task space for each separate task your application program sets up.

Table 1-3 Partition of the Common Memory

Address	Size (bytes)	Type	Function
C100 – FFFF	16128	Flash	Kernel – code
C000 – C0FF	256	I/O	Memory mapped I/O
B400 – BFFF	3072	Flash	Kernel – code
B3F0 – B3FF	16	RAM	Kernel – Real Time Clock Buffer
B3E0 – B3EF	16	RAM	Kernel
B3D0 – B3DF	16	RAM	Kernel – C/Forth ISR vectors
B200 – B3CF	**464**	**RAM**	**Application – C variables at runtime, Flash write buffer during program download**
B000 – B1FF	**512**	**ONCHIP_RAM**	**Application – C variables**
AEC0 – AFFF	**320**	**EEPROM**	**Application – nonvolatile storage**
AE00 – AEBF	192	EEPROM	Kernel
8E00 – ADFF	**8192**	**RAM**	**Application – C Variables and multitasking task areas, optionally battery-backed**
8500 – 8DFF	2304	RAM	Kernel – buffers and stacks
8400 – 84FF	256	RAM	Kernel – Forth user area
8060 – 83FF	928	RAM	Kernel – buffers and stacks
8000 – 805F	96	RAM	Kernel – processor Control Registers

Shaded entries indicate memory available for application programs.

The common memory is also partitioned between the operating system and application program. Table 1-3 shows the addresses used by the operating system, and in boldfaced type those addresses available to the application program.

Addressing Memory in C

Although 8-bits are sufficient to address the 256 possible pages, the page is padded out to a more standard 16-bit date type so that the full address, lower 16 bits plus 16-bit page, occupies 32 bits. We'll refer to this full address as an *xaddress* (32-bit extended addresses). Three macros are available in the **\mosaic\fabius\include\mosaic\types.h** file to simplify the manipulation of xaddresses and their constituent 16-bit addresses and pages. These C macros are:

```
TO_XADDR
XADDR_TO_ADDR
XADDR_TO_PAGE
```

Multi-page C programs rely on a "page change" routine in the common kernel memory to call functions on other pages. Unlike the Forth compiler, the C compiler is not "page smart", and does not know at compile time whether a page change is needed. In fact, page changes are rarely needed,

because most functions call other functions that are located on the same page or in common memory. Calls to functions on the same page or to common memory take only 11.5 or 13.75 microseconds, respectively, while function calls to other pages require just under 49 microseconds. Because page changes are rare, the average execution speed of multi-page C applications is not significantly impacted by the need for page changes.

Addressing Flash

Flash memory is nonvolatile, like PROM. Thus it retains its contents even when power is removed, and provides an excellent location for storing program code. Simple write cycles to the device do not modify the memory contents, so the program code is fairly safe even if the processor "gets lost". But flash memory is also re-programmable, and the flash programming functions are present right in the QED-Flash Board's onboard software library. These functions invoke a special memory access sequence to program the flash memory contents "on the fly". This allows you to modify your operating software (for example, to perform system upgrades). You can also store data in the flash device. You can program from 1 byte up to 65,535 bytes with a single function call using the pre-coded flash programming routine. Programming time is approximately 60 milliseconds per kilobyte.

Six special functions facilitate access to Flash memory. Their function names in C are:

```
DownloadMap()  PageToFlash()  PageToRam()
StandardMap()  ToFlash()      WhichMap()
```

and their names in Forth are:

```
DOWNLOAD.MAP   PAGE.TO.FLASH  PAGE.TO.RAM
STANDARD.MAP   TO.FLASH WHICH.MAP
```

The FLASH programming functions use a buffer in the 68HC11's on-chip RAM starting at hex addresses B200-B3CF. The remaining on-chip RAM at B000 to B1FF is available to you. Also, because FLASH programming is generally not done at run-time, you can still use the Flash buffer for run-time variables.

Software Development Using Flash Memory

Because code cannot be downloaded or compiled directly into flash memory, the flash memory map implements *page swapping* to provide a mechanism for getting the compiled code into the flash memory. There are two page-swap modes: one is called the Standard Map and the other is called the Download Map. As the names suggest, the Standard Map is used during run-time, and the Download Map is used during downloading and compilation of Forth source code or C-compiler S-records from the PC to the QED Board. The two maps are very similar; the effect of changing from the Standard to the Download map is to swap the locations of pages between the flash and the RAM.

Table 1-4 Addressing the Flash and RAM in *Standard* and *Download* Memory Maps

	Flash Pages		RAM Pages	
Standard Address Map	04 – 06	50 – 5F	01 – 03	40 – 4F
Download Address Map	01 – 03	40 – 4F	04 – 06	50 – 5F

In normal operation the Flash memory is addressed on pages 04-06 and 50-5F, and the RAM is addressed on pages 01-03 and 40-4F. During download their addresses are swapped, so that the Flash is addressed at pages 01-03 and 40-4F and the RAM at pages 04-06 and 50-5F.

To see how it works let's consider a hypothetical download. Suppose you have compiled code intended to load into the Flash and run from it at addresses on page 04. Automated commands contained in the download file establish the download map, load the code into RAM, transfer the code to flash, and re-establish the standard map. In this case, the download file would:

1. Swap the addresses of the RAM and Flash (by executing the command **DOWNLOAD.MAP**) so that the RAM is now addressed on page 04;
2. Download the code to its proper addresses on page 04;
3. Copy the code (using the command **PAGE.TO.FLASH**) from page 04 into the Flash addressed on page 01; then,
4. Swap (by executing the command **STANDAD.MAP**) the RAM and Flash addresses back so that the Flash is now addressed on page 04, and the RAM on page 01 is available for run-time use by your program.

The Control-C download file does all this for you so you don't need to worry about the details. But if you're interested, just peruse the download file in the editor where you'll see the commands it uses to manage memory during the download process.

Locating Nonvolatile Data in EEPROM

The QED Board's built-in EEPROM provides an ideal place to store calibration constants or other data that must be changed from time to time, but that must be retained even when power is removed. The EEPROM (Electrically Erasable Programmable Read-Only Memory) can be modified up to 10,000 times before it loses its ability to retain data. The **ANALOGIO.C** file presents an example of how to locate a static "variable" in EEPROM.

The EEPROM variable should be declared as an un-initialized static variable; these are located by the linker in the "data" section, which normally points to system RAM where normal variables are stored. By following the syntax presented here, you can relocate the data section to point to EEPROM while defining the EEPROM variables, and then restore the data section to its standard RAM location. To define an EEPROM variable, use the following code:

```
#pragma  option data=.eeprom  // put the following variables in eeprom
static uchar numsamples;
static int  nonvolatile_int;
```

```
static float calibration_value;
#pragma  option data=.data  // restore the data area to RAM
```

The **#pragma** statements are pre-processor directives that are interpreted by the linker. In the code fragment above, we located three nonvolatile variables in EEPROM; note that we did NOT include initializers in the declaration statements. Initializers don't make any sense for EEPROM variables, because special functions must be called to store values into EEPROM, so initialization can't be accomplished by placing initialization data in the download file. These EEPROM variables must be initialized programmatically at run-time.

To store data into the EEPROM variables, use the following functions which are declared in the **XMEM.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory:

```
void StoreEEChar(char value, char* addr)
void StoreEEInt(int value, int* addr)
void StoreEELong(long value, long* addr)
void StoreEEFloat(float value, float* addr)
```

To learn how to interactively modify the contents of EEPROM variables, read the glossary entries for these functions in the Control-C Glossary.

These EEPROM storage functions are easy to use. For example, to store the value 123 into the character variable **numsamples**, you would place the following statement in your program:

```
StoreEEChar( 123, &numsamples);
```

To avoid wearing out the EEPROM by executing unneeded write cycles, these functions check whether each EEPROM byte already holds its specified contents. If so, the write is not performed. Thus there is no penalty for redundant execution of commands that initialize particular locations in EEPROM.

While EEPROM variables *must* initialized programmatically at run-time the first time they are used, they don't need to be re-initialized each time the processor starts up because the nonvolatile EEPROM retains the data. Even so, initializations can be performed every time the processor starts up, with no adverse effects on the life span of the EEPROM. For example, initialization code in an autostart routine could execute ATTACH functions to ensure that all needed interrupt vectors are properly initialized each time the processor restarts. If the EEPROM cells have been corrupted for some reason, the ATTACH command installs the correct contents, but if the specified interrupt vector information is already in the EEPROM, the memory cells are not needlessly rewritten.

Interrupts are disabled during writes to EEPROM

All of the EEPROM storage routines globally disable interrupts while each EEPROM byte is being programmed, and it takes 20 milliseconds to program each byte. Thus you should avoid storing values in EEPROM while time-critical events are being serviced by interrupts.

For experts and the curious: Interrupts are disabled during stores to EEPROM because QED-Forth vectors all interrupts via the EEPROM, and the 68HC11 hardware does not allow any EEPROM cells to be read while a single EEPROM cell is being written to. Thus if an interrupt occurs while one of the EEPROM storage functions is writing to EEPROM, the interrupt will not be able to read the instruction code in the interrupt vector. Disabling interrupts prevents this error, but the interrupt service is delayed until the EEPROM write is finished.

Write-Protecting EEPROM

It is possible to write-protect locations within the EEPROM to ensure the integrity of calibration constants or other vital information. This is done using the EEPROM block protect register named **BPROT** (MC68HC11F1 Technical Data Manual, p.8-2). Four blocks of size 32, 64, 128, and 288 bytes may be individually protected by storing an appropriate configuration value to **BPROT**. The contents of the **BPROT** register may be changed using the C function **InstallRegisterInits** or the QED-Forth word **INSTALL.REGISTER.INITS**; please consult its glossary entry for details.

To make a turnkeyed application maximally "bullet-proof" and fail-safe, consider using the **BPROT** register to protect the first three blocks in the EEPROM totaling 224 bytes. This protects the on-board kernel's configuration region (the first 32 bytes in EEPROM) plus the interrupt vectors (the next 160 bytes in EEPROM) plus an additional 32 bytes available the programmer. The remaining 288 bytes of EEPROM then remain available for modification by the application program.

Once values have been stored in the EEPROM, they may be read using the conventional memory fetch operations such as C@, @, and 2@.

Using C Arrays and Forth (Kernel) Arrays

Storing Data Acquisition Results in C Arrays and Forth Arrays

Programs written in Control-C use space in common memory to store variables. You may store simple variables or arrays of variables there using standard C syntax. However, common memory is a limited to approximately 9K. It can get used quickly in multitasking systems because each task requires a task area of about 1K. Consequently, the programmer may require access to additional RAM. Access is provided through the use of Kernel Arrays, also called Forth Arrays. Using Forth Arrays you may dynamically dimension arrays of virtually any size in the extended address space – and their memory allocation is automatically handled by the kernel's heap memory manager.

The code presented in the sample program **ANALOGIO.C** uses a C array and a **FORTH_ARRAY** to store the results of multiple A/D conversions. This section uses that code as an example to discuss some interesting features of both C Arrays and **FORTH_ARRAY**s.

Declaring a C Array

The use of C arrays is discussed in detail in all standard C texts. In this program, the one-dimensional 16-element character array named **results_8** is declared and allocated in RAM using the statement:

```
uchar results_8[DEFAULT_NUMSAMPLES];
```

where **DEFAULT_NUMSAMPLES** is a constant equal to 16. The arrays are easy to use. For example, the following C statement assigns the last element in the array to a static variable named **my_variable**:

```
my_variable = results_8[15];
```

To see another simple example that demonstrates how C arrays are accessed, look at the **InitAnalog()** function in the **ANALOGIO.C** file. The **results_8** array is zeroed by executing the following statement:

```
for(i=0; i< DEFAULT_NUMSAMPLES; i++)
results_8[i] = 0;          // zero the array
```

Note that this array is dimensioned and allocated by the compiler and linker. In contrast, **FORTH_ARRAYS** are dimensioned and allocated dynamically by the run-time program itself.

Converting a 16 Bit Address to a 32 Bit xaddress

The **AD8ToCArray()** function that we just used provides an interesting example of type conversion. The definition of the function is:

```
_Q void AD8ToCArray( int channel)
{  EXTENDED_ADDR buffer;
buffer.sixteen_bit.addr16 = results_8;
buffer.sixteen_bit.page16 = 0;
AD8Multiple(buffer.addr32,0,DEFAULT_NUMSAMPLES,channel);
}
```

The purpose of **AD8ToCArray()** is to properly call **AD8Multiple()** which is defined in the **ANALOG.H** file. **AD8Multiple()** is optimized to use a **FORTH_ARRAY** buffer, and so expects a 32 bit buffer xaddress instead of a simple 16 bit buffer address. To convert the simple 16 bit address returned by **results_8** into a 32 bit extended address, we take advantage of the **EXTENDED_ADDR** union defined in the **TYPES.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory. The union is defined as:

```
typedef union{    xaddr addr32;
struct{ uint  page16;
char* addr16;
} sixteen_bit;
}  EXTENDED_ADDR;
```

To convert a 16 bit address into a 32 bit xaddress, we use the **EXTENDED_ADDR** typedef to declare an instance of the union (named "**buffer**" in this example), store the 16 bit address into the **buffer.sixteen_bit.addr16** element, and store 0 (the default page) into the **buffer.sixteen_bit.page16** element. Then we reference the corresponding 32 bit xaddress via the **buffer.addr32** element of the union. While it is rare that you will have to convert from 16 bit to 32 bit address types, this example provides a template for how to do it.

A Review of FORTH ARRAYs

FORTH_ARRAYs have two key advantages. First, they are allocated in paged memory, so they allow your program to access the large 8 Megabyte memory space of the QED Board. In contrast, C arrays must reside in the available common RAM which is limited to approximately 9 kilobytes on the QVGA Controller. Second, they can be dynamically dimensioned, re-dimensioned and de-allocated (deleted) while your program is running; this boosts efficiency by maximizing the use of the available memory.

To define a new Forth Array, simply use the **FORTH_ARRAY** typedef followed by a name of your choice. For example, in the **ANALOGIO.C** file the following declaration appears:

```
FORTH_ARRAY  results_12;
```

Before the **FORTH_ARRAY** can be accessed at runtime, it must be dimensioned. This is typically accomplished by calling the **DIM()** macro defined in the **ARRAY.H** header file. For example, to dimension the **results_12** array to have 10 rows and 1 column of integer data, we would execute:

```
DIM(int, 10, 1, results_12);
```

In the **ANALOGIO.C** file, the pre-defined macro named **DIM_AD12_BUFFER()** invokes the **DIM()** routine for us (its definition is in the **ANALOG.H** file in the **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory).

After the **FORTH_ARRAY** is dimensioned, it can be accessed by a family of macros and functions that are defined in the **ARRAY.H** header file and are described in the Control-C Glossary. These include functions that fetch from, store to, and calculate the address of individual elements, swap and copy entire arrays, fill an array with a specified character, and delete the array so that it no longer requires memory in the heap. The **PrintForthArray()** and **InitAnalog()** functions in **ANALOGIO.C** provide examples of how to call a few of these functions.

Printing the Contents of a FORTH ARRAY

The **PrintForthArray()** function presented in **ANALOGIO.C** is a more general version of the **PrintFPArray()** function in **GETSTART.C** as discussed in an earlier chapter. The function is defined as follows:

```
_Q void PrintForthArray(int float_flag, FORTH_ARRAY* array_ptr)
// works for FORTH_ARRAYS dimensioned using the standard DIM() macro.
// float_flag is true if array holds float numbers, false otherwise.
{ int r, c;
putchar('\n');
for (r = 0; r < NUMROWS(array_ptr); r++)         // for each row
{   for  (c = 0; c < NUMCOLUMNS(array_ptr); c++) // for each column
if(float_flag)
```

```
printf("%9.4g  ",FARRAYFETCH(float,r,c,array_ptr));
else
printf("%9ld  ",ARRAYFETCH(long,r,c,array_ptr));
putchar('\n');        // newline after each row is printed
PauseOnKey();         // implement xon/xoff output flow control
}
}
```

After calling **putchar()** to output a newline character, we enter nested **for()** statements that print the contents of each element. Because the compiler treats floating point numbers differently than numbers stored in other formats, we use **FARRAYFETCH()** to access floating point arrays, and **ARRAYFETCH()** to access **char**, **int** or **long** arrays. The **PauseOnKey()** function is called once per row to suspend the QVGA Controller's printed output if the terminal program has sent the **XOFF** handshake character; the printout resumes when the terminal sends the **XON** character. **PauseOnKey()** also gives the user the ability to terminate the printout by typing a carriage return character from the terminal.

This function can be tailored to meet the detailed needs of your application. You can change the **printf()** formatting, or insert extra carriage returns to confine the printout to one screen width.

Access Routines for Arrays and Matrices

The routines **2ARRAY.STORE** and **2ARRAY.FETCH** have been added to the kernel to enable you to store to and fetch from 2-dimensional arrays or matrices. The unique aspect of these routines is that they properly handle data of different sizes (1 byte, 2 bytes, or 4 bytes) depending upon the way that the array or matrix is dimensioned. For example, if an array named **CHAR.ARRAY** is dimensioned (using the **DIMENSIONED** routine) to hold 1 byte per element, then executing

```
0  0  ' CHAR.ARRAY  2ARRAY.FETCH
```

will return the first 8-bit character in the array. If another array named **DATA.ARRAY** is dimensioned to hold 16-bit data, then executing

```
0  0  '  DATA.ARRAY  2ARRAY.FETCH
```

will return the first 2-byte element in the array. These routines help support access to paged memory when programming in Control-C.

```
Wildcard Carrier Board Drivers
\
********************************************************************************
********
\ FILE NAME:  To_xFlash.4th
\ Kernel extension for supporting memory management and flash programming for the
\ Wildcard Carrier Board and QVGA Controller Board
\ Copyright 2002-2003 Mosaic Industries, Inc.  All rights reserved.
\ ------------------------------------------------------------------
\ DATE:       2/06/2003
\ VERSION:    1.2
\ SUPPORTED HARDWARE:
\      QED-4 (FLASH) Board
\      QVGA Controller Board
\      WildCard Carrier Board
\ ------------------------------------------------------------------
\ Please see the Wildcard Carrier Board User Guide for detailed instructions.
\
\ These are the User Functions defined by this driver:
\   To_xFlash      ( xsource_addr\xdestination_addr\numbytes -- success )
\   Query_xMemory  ( page -- device.size&type )
\   Is_xRAM        ( page -- flag )
\   Is_xFlash      ( page -- flag )
\   Page_To_xFlash ( source.page -- )
\   Download_Map_QVGA ( -- )
\   Download_Map_WCB  ( -- )
\   Standard_Map_QVGA ( -- )
\   Standard_Map_WCB  ( -- )
\
\ To_xFlash allows writes to the extra flash chip on the WCB, the QVGA Controller,
\ and/or the kernel flash chip. This function has the same behavior as To_Flash
\ (see its glossary entry in the QED-FLASH update notice), but uses the ID of the
\ flash device to determine how to perform the write.
\
\ Query_xMemory determines the size and type of an extra memory device. The value
\ returned is a signed integer of the size in kilobytes. The sign represents the
\ device type, a positive size for RAM, and negative for flash. A value of zero
\ means either no device resides at the base page, or a flash chip is installed with
\ either an unknown ID, or is write protected.
\ Two notes on Query_xMemory:
\   The base page of the device must equal the bitwise-AND of the page and 0xF0.
\   Zero is returned if the base is zero
\
\ Is_xRAM and Is_xFlash return true is the given page resides on the coorepsonding
\ memory device. Note, these functions are wrappers for Query_xMemory.
\
```

```
\ Page_To_xFlash is the To_xFlash replacement for PAGE.TO.FLASH. It is aware of the
\ RAM and flash devices available for the WCB and QVGA Controller, and provides
\ download friendly error reporting.
\ Note, it relies on Is_xRAM and Is_xFlash for parameter checking
\
\ Note, the above functions use the same buffer areas as the kernel routines
\ that they supercede (ie To.Flash and Page.To.Flash).
\ (buffers are located in the top half of onchip RAM from 0xB200-0xB3E0).
\ These routines are not re-entrant with respect to multitasking.
\ This means that a multitasking application cannot support simultaneous
\ flash programming by separate tasks unless a resource variable is defined
\ and GET and RELEASE are used.
\
\ Download_Map_QVGA, Download_Map_WCB, Standard_Map_QVGA, Standard_Map_WCB
provide a
\ simple way to set the memory map of a board while clearly stating what is being done.
\
\      Standard Map       Download Map
\      RAM     Flash      RAM     Flash
\ QED   01-03   04-06      04-06   01-03
\ WCB   40-4F   50-5F      50-5F   40-4F
\ QVGA  60-6F   70-7F      70-7F   60-6F
\
\ NOTE: The memory map of the QED Board is maintained across power cycles and resets,
\   but is cleared by a factory cleanup. However, the WCB and QVGA, revert to standard
\   map at powerup, but maintain their state across resets or factory cleanups.
\
\ The above functions should provide simple means to compile code for storage in the
\ flash devices of the QED Board, the Wildcard Carrier Board, or the QVGA Controller.
\
\ Top Level C Functions:
\   int To_xFlash ( xaddr source, xaddr destination, uint numbytes )
\   int Query_xMemory ( int page )
\   bool Is_xRAM ( int page )
\   bool Is_xFlash ( int page )
\   void Page_To_xFlash ( int source_page )
\   void Download_Map_QVGA ( void )
\   void Download_Map_WCB ( void )
\   void Standard_Map_QVGA ( void )
\   void Standard_Map_WCB ( void )
```

Special Memory Areas for Forth Programmers

The Kinds of Memory Available on the QED Board

[[Change to following to describe Flash program development instead of ROM program development]]

RAM (Random Access Memory) can be both read and written to by the processor. Standard RAM loses its contents when power is removed; it is "volatile". On the other hand, ROM (Read-Only Memory) and PROM (Programmable Read-Only Memory) can be read by the processor but cannot be modified by the processor. PROM can be programmed in a PROM burner which applies programming voltages in a specified pattern to initialize the contents of the memory. In this document we use the terms "ROM" and "PROM" interchangeably. PROMs retain their contents when power is removed; for this reason they are called "nonvolatile". A small amount of EEPROM (Electrically Erasable PROM) is available on the QED Board. EEPROM is nonvolatile and can be read by the processor. It can also be programmed by the processor, but each byte takes 20 milliseconds to modify, or about 10,000 times longer than it takes to write to a byte of RAM. Thus EEPROM is useful for nonvolatile storage of quantities that are only rarely changed or updated, such as calibration constants.

Like PROM, battery-backed RAM retains its contents when power is removed, but unlike PROM, it can be modified by the processor. Battery-backed RAM behaves exactly like PROM when it is write-protected: it retains its contents when the board is powered down, and its contents cannot be modified by the processor. The QED Board allows you to "emulate" the behavior of PROM by flipping a switch to write-protect pages of battery-backed RAM. This feature speeds debugging by obviating the need for PROM burning during the development of your application. You can see exactly how your application will work once it is burned into PROM without actually having to burn a PROM.

Overview of the Memory Areas

As described in the prior chapters of this manual, the QED Board uses a paged memory architecture that yields an ample 8 Megabyte addressable memory space. Up to 384 Kbytes of memory (128K in each of 3 sockets) can be accommodated on the QED Board. The "memory map" describes how the available memory is allocated to specific uses by an application program. For example, Appendix A details how memory is allocated after a COLD startup.

Before you start programming your application, you should decide upon an appropriate memory map. This involves allocating the ten memory areas listed in Figure 3-1. A short summary of these areas may help to clarify their functions. The user area is a region of up to 256 bytes that contains the pointers and "user variables" that the QED-Forth task needs. In fact, the user area contains all of the pointers and base addresses that define the memory map. Each task in an application has its own user area, so each task can have a distinct memory map specified by its own user variables. The location of the start of the user area is specified by the contents of a variable named UP (user pointer).

The return stack holds the return addresses of subroutine calls, and the data stack holds parameters passed among QED-Forth routines. The Terminal Input Buffer (TIB) holds the last line of serial input. The POCKET buffer holds the last word interpreted by QED-Forth. The PAD buffer is a scratchpad available to the user; number-to-string conversion is performed in the area below the PAD buffer. The heap is a block of RAM available for dimensioning data structures such as arrays and matrices. The variable area holds variable contents and parameter fields, the name area holds the headers of the words in the dictionary, and the definitions area holds the object code that defines the action of each word.

Configuring an appropriate memory map is not difficult. Fortunately, the first six of these ten memory areas are small buffers or regions occupying less than 1 Kbyte each, and often they can be left in the default positions that they occupy after a COLD restart. The remaining four memory regions (the heap, variable area, name area, and definitions area) should be initialized to appropriate locations by the programmer at the start of the programming session. The pre-defined utility USE.PAGE described below helps you to locate these four memory areas.

The Default Memory Map in Common RAM

Figure 3.1 lists the ten key areas in the memory map, the user variables that set their locations, the command that places the location on the data stack, the default address (in hex) in common memory after a COLD restart, and their default size (in decimal). A “negative size” means that the area grows downward in memory.

Table 1-5 Ten memory areas that specify an application’s memory map.

Memory Area	User Variable Name	Contents	Default Location	Default Size
User area[1]	UP	UP @	8400H	256
Return stack	R0	R0 @	8800H	-768
Data stack	S0	S0 @	8B00H	-768
Terminal Input Buf	UTIB	TIB	8B00H	96
Pocket	UPOCKET	POCKET	8B60H	36
Pad	UPAD	PAD	8BA8H	+88, -36
Heap end	CURRENT.HEAP	CURRENT.HEAP X@	8DFFH	512
Variable area	VP	VHERE	8E00H	512
Name area	NP	NHERE	9000H	512
Definitions area	DP	HERE	9200H	1024

1. UP is a variable, not a user variable.
2. Negative sizes indicate that memory use grows downward, towards lower addresses.

A COLD restart initializes the system so that all of the memory areas are in the common RAM. The programmer can move the memory areas by modifying the appropriate user variables.

The stacks, user area, and POCKET must reside in common RAM. Recall that the common memory includes all addresses above 8000H (that is, the top 32K of memory). QED-Forth reserves the

1K of RAM from 8000H through 83FFH for system needs, 1/2K EEPROM is located at AE00H through AFFFH, and the kernel occupies B400H through FFFFH. The common RAM areas from 8400H through ADFFH (10.5K) and B000H through B3FFH (1K RAM on the 68HC11 chip) are available for the programmer's use.

The User Area

The user area is a 256-byte block of memory which contains all of the user variables that set the memory map and facilitate compilation and execution of QED-Forth code. Each user variable returns an extended address that is calculated relative to the contents of the user pointer UP. When executed, a user variable adds its unique offset to the contents of UP and places this address (under 0, the default page) on the stack. For example, to find the current base address of the user area, execute

```
HEX UP←
ok [ 2 ] \ 8200 \ 0
?←
8400  ok
```

Thus the current user area starts at 8400H. The definitions pointer DP is a user variable that leaves its address on the stack when executed:

```
DP←
ok [ 2 ] \ 8408 \ 0
```

DP is defined with the word USER as a user variable that adds the offset 8 to the current value of UP. The user area is described in detail in Chapter 15.

The user area facilitates multitasking, as the multitasking executive only needs to change the contents of the single variable UP to perform a context switch to a new task's user area. Each task can have its own memory map, stacks, and task-private variables. These topics are covered in detail in the "Multitasking" Chapter.

The following sections describe each of the memory areas listed in Figure 3.1. The location of each area may be changed by storing an address in the appropriate user variable.

Return and Data Stacks

The user variables R0 (R-zero) and S0 (S-zero) hold the 16-bit addresses of the bottoms of the return and data stacks, respectively. Both stacks must be in common ram. The default locations for the return and data stacks are 8B00H and 8800H. Each stack is allocated 3/4K of space, and grows downward in memory. The first item on the stack is stored at the two bytes below the value in R0 or S0. For example, the default value in R0 is 8800H, so the first item on the stack is stored with its most significant byte at 87FE and its least significant byte at 87FF. The S register of the 68HC11 is the return stack pointer. The kernel word (RP) returns the address of the most significant byte of the top item on the return stack. The Y register is used as the data stack pointer; it points to the most significant byte of the top item on the data stack. The kernel word (SP) places on the data stack the value of the Y register that existed just before (SP) was executed.

Terminal Input Buffer

The terminal input buffer (TIB) holds an ascii representation of the last line of input from the serial port. The default location of the TIB is at 8B00H, and its default size is 96 characters. The kernel word QUERY, a key routine in the QED-Forth interpreter, executes

```
TIB CHARS/LINE @ EXPECT
```

to get the next line of input into the terminal input buffer, and then loads the number of characters received into the user variable #TIB. The default value of the user variable CHARS/LINE is 96, which is the size of the TIB. Unless you move the TIB and/or the POCKET buffer just above it in memory, you should not set the user variable CHARS/LINE to a value greater than 96. Doing so could cause a long command line to overwrite the memory area above the TIB.

The TIB may be placed at any RAM location. It need not be in the common memory; however, the buffer may not cross a page boundary. To change the location of the TIB, use X! to store a 32-bit starting address into the user variable UTIB. Executing the word TIB places the 32-bit starting address of the buffer on the data stack.

POCKET

POCKET is a buffer used by the WORD routine in QED-Forth's interpreter. The interpreter parses each blank-delimited word in the input stream, clamps its size to a maximum of 31 characters, and moves the counted ascii string to POCKET, which must be in the common RAM. (For experts: Larger strings may be parsed using the word PARSE). To change the location of POCKET, use ! to store a 16-bit starting address (which must be in the common RAM) into the user variable UPOCKET. Executing the kernel word POCKET leaves the starting address of the pocket buffer under a 0 (the default page) on the data stack.

PAD

PAD is a scratchpad buffer. The memory area above PAD is available for the programmer, and the memory below PAD is used by QED-Forth's number conversion routines. PAD can be in RAM on any page or in the common area, as long as the buffer does not cross a page boundary. At least 32 bytes below PAD (starting at PAD - 1) must be available for the integer and floating point number-to-string conversion routines. The memory above PAD is used to hold incoming characters by the words RECEIVE.HEX, INPUT.STRING, ASK.NUMBER, and ASK.FNUMBER. The location of PAD may be altered by using X! to store a 32-bit base address into the user variable UPAD. Executing PAD leaves the base address of the buffer on the stack. The default memory map allocates 36 bytes below PAD for number conversion, and 88 bytes above PAD for use by the programmer.

Heap

The user variable CURRENT.HEAP holds the 32-bit extended address of the end of the heap (actually, it points 1 byte above the end of the heap). The other variables that configure the heap are stored near the top of the heap, just below the address in CURRENT.HEAP. This makes it easy to switch among multiple heaps, if needed, by changing only a single 32-bit user variable. The size of the heap is limited only by the amount of available contiguous RAM, and the heap can occupy mul-

tiple contiguous pages. The default heap is only 1/2K in size; it can be resized and/or relocated using the command

```
IS.HEAP  ( start.xaddr\end.xaddr -- )
```

which sets CURRENT.HEAP and initializes the other heap configuration variables.

The heap must always be in RAM, even after the application is finished and the dictionary areas are write-protected or ROMmed. Chapter 5 describes the heap memory manager in detail.

Variable Area

The variable area holds the parameter fields of variables, self-fetching variables, arrays, and matrices. The parameter fields of variables hold the contents of the variables, and the parameter fields of arrays, matrices, and other heap items hold handles to the heap items and dimensioning information. The variable area must be in RAM, and can be on any page. To modify its location, use X! to store a 32-bit extended starting address into the user variable VP (variable pointer).

Executing VHERE places the current contents of VP on the data stack. VALLOT increments the contents of VP, and aborts if a page boundary is crossed. In general, the parameter field of a data structure such as an array or a matrix cannot cross a page boundary.

The programmer must guarantee that the variable area will always be in RAM, even after the application is finished and the dictionary is write-protected or ROMmed.

Name Area

The name area holds the headers of the words in the dictionary. Each header contains the count and first several characters (determined by WIDTH) in the name, a pointer to the code field of the word, and a link to the previous name in the dictionary. (Appendix C describes the header format in detail). The name area must be in modifiable RAM during compilation, but sections of the name area containing the headers of defined words can be write-protected, and the entire name area may be write-protected or burned into PROM when the application program is finished.

The name area can be spread throughout the memory space. For example, a portion of the name area will be in a block of memory specified by the programmer, and another portion of the name area is in a separate block on page 15 where the headers for the kernel routines are stored. The interpreter automatically links each new header to the previous header to maintain the integrity of the name list.

The user variable NP (name pointer) points to the next available location in the name area. Executing NHERE (name here) places the contents of NP on the data stack. To initialize the location of the name area, use X! to store a 32-bit extended address into NP. NP is automatically incremented each time a new word is defined. If the creation of a new word causes NP to cross a page boundary, an error message is issued. An individual header cannot cross a page boundary. In response to such an error message, you should store a new value into NP to reset the name area to a new page before compiling more definitions.

Definitions Area

The definitions area holds the code that defines the behavior of the words in the dictionary. The user variable DP (definitions pointer) points to the next available location in the definitions area. The area must be in RAM during compilation, but sections of the definitions area containing already defined words can be write-protected, and the entire definitions area is typically write-protected or burned into PROM when the application program is finished.

The user variable DP (definitions pointer) points to the next available location in the definitions area. DP is initialized using an X! command, and HERE leaves the contents of DP on the data stack. ALLOT advances the contents of DP, and issues an error message if DP crosses a page boundary during a definition. In general, the definition code of a single word cannot cross a page boundary. In response to such an error message, you should store a new value into DP to reset the definitions area to a new page before compiling more definitions.

Customizing the Memory Map

After a COLD restart, all of the memory areas just described are located in the common RAM. While the location and sizes of the user area, return and data stacks, TIB, POCKET, and PAD are probably adequate for most purposes, the heap, name and definitions areas are too small for all but the shortest development sessions. If you do not modify NP and DP, you may find that after defining several dozen words, the name area will start to over-run the definitions area, giving unpredictable results.

It is recommended that you set the memory map immediately after a COLD restart. For all but the shortest debugging sessions, it is recommended that the name and definitions areas and the heap be moved and resized.

The command

```
USE.PAGE    ( page -- )
```

has been included in the kernel to make it easy to set up a memory map after a COLD restart. It lets you specify a 32K page that will accommodate the name and definitions areas, and sets up the following memory map:

20 Kbyte definitions area starting at 0000H on the specified page

12K name area starting at 5000H on the specified page

4K variable area starting at 3000H on page 15 (0FH)

16K heap occupying 4000H to 7FFFH on page 15 (0FH)

The return and data stacks, TIB, POCKET, and PAD are left at their prior locations. Thus the variable and heap areas are placed on page 15 which is guaranteed to be non-write-protected RAM so the variable and heap areas will never be mistakenly ROMmed or write-protected. The definitions and names areas are placed on the specified page and may be write-protectable; recall that pages 4, 5, 6, and 7 on the QED Board can be write-protected by flipping the on-board DIP switches.

An equivalent definition of USE.PAGE is:

```
HEX
: USE.PAGE  ( page -- )
  >R                      \ keep the page on the return stack
  0000 R@ DP X!           \ set definitions pointer
  5000 R> NP X!           \ set the name pointer
  3000 0F VP X!           \ set variable pointer
  4000 0F 7FFF 0F IS.HEAP \ set heap
;
```

In keeping with the discussion earlier in this chapter, it is recommended that an ANEW command be executed at the start of each debugging session immediately after changing the memory map.

For example, suppose that a 20K definitions area, 12K names area, 4K variable area, and 16K heap are adequate for your application. The following commands would correctly configure your memory map:

```
4 USE.PAGE
ANEW  APPLICATION.NAME  \ choose any name you like here
```

The resulting memory map places the definition and name areas in page 4 where they can be write-protected during development by turning DIP switch #1 ON. As described in the next section, this memory map is also compatible with a production system which contains the 64K QED-Forth PROM in socket S1, a 32K RAM in socket S2, and a 32K PROM in socket S3.

The memory map can be customized to meet the needs of each application. Some applications may need a large heap, while others may need little or no heap space because they do not use arrays, matrices, or other dynamic data structures. Several example memory maps are presented in the next section.

Summary

In summary, remember that the user area, stacks and POCKET must be in the common memory, and the other sections of the memory map may be placed anywhere in memory. Stacks, heap, variable area, TIB, POCKET, and PAD should always be in modifiable memory (RAM) even after the name and definitions areas have been write-protected or PROMmed. The name and definitions area must be in RAM during compilation, but can be write-protected during debugging and burned into PROM when the application is finished. USE.PAGE is a handy utility that sets up a default memory map, and it is strongly recommended that you execute an ANEW command after changing the memory map.

Designing a Memory Map for a “Turnkeyed” Application Program

Most users of the QED Board want to create a “turnkeyed” application program that is burned into PROM and is set up to autostart each time the board is powered up. These users want to design their application program to control a production instrument as opposed to a one-of-a-kind prototype. To accomplish this goal, the memory map must be configured to properly allocate the memory areas between the available RAM and PROM. This section describes how to perform this allocation. For a detailed and very informative example of a turnkeyed application program, please see Chapter 17.

RAM vs. ROM

The first step in programming an application is assigning the memory areas that will be occupied by the definitions, name area, variable area, and heap. The definitions area of a turnkeyed program must be in PROM (or some type of write-protected nonvolatile memory) for the application to run properly. If the definitions are not write-protected or in PROM, any "crash" could cause part of the program to become corrupted. The name area must be present during development and debugging so that you can compile and execute words. If names are required in the final application (for example, if you want to give the end user the ability to execute commands from a terminal), then the name area should be included in PROM or nonvolatile memory in the final turnkeyed system. The name area can be left out of the final turnkeyed application if the end user need not execute Forth commands to use the final instrument. Removing the names saves PROM space in the production system, and can improve the "security" of the code by keeping informative English-sounding names from would-be pirates who might try to decipher the code.

While the definitions and name areas should end up as PROM in the final system, the variable area and heap must always be located in RAM. Both variables and the contents of the heap must be subject to change while the application is running, so they can never be ROMmed.

Other Memory Areas

In addition to the four main memory areas, memory must be allocated in the common RAM for data and return stacks, and for task areas. For most applications, the default data and return stacks of 768 bytes each in common RAM are adequate, and no further allocation is needed. In multitasking applications, each task typically requires the default 1K task area in common RAM as described in the glossary entry for BUILD.STANDARD.TASK. This task area includes the data stack, return stack, PAD, POCKET, TIB, and user area for the task. The 6K of common RAM starting at 9600H and labeled as "6K available RAM" in Appendix A is a convenient place to allocate task areas for up to 6 standard tasks.

Required buffers such as the PAD (scratchpad area) and TIB (terminal input buffer, for receiving a line from the serial port) may be allocated in common RAM or paged RAM; the POCKET buffer must be allocated in common RAM. For the great majority of applications, the default locations of these buffers are adequate and the programmer need not take any action to re-locate them (the memory map in Appendix A describes the default locations).

While compiling an application, all four main memory areas (definitions, names, variables, and heap) are located in RAM. During debugging, the definitions and name areas may be optionally write-protected. As described in the next section of this chapter, proper use of write-protection can save you from having to re-download all of your code after a crash or mistaken command.

Available PROM and RAM on the QED Board

The QED Board has 3 memory sockets labeled S1, S2, and S3. Each can accommodate a memory device ranging from 32 Kbytes to 128 Kbytes. Appendix A specifies the memory map in detail, and the following brief summary may be helpful.

Socket S1 accommodates the QED-Forth ROM which holds the development system as well as the runtime libraries. This ROM must be present for the QED Board to function.

Socket S2 holds a 32K or 128K RAM, and at least 32K of RAM must be present in this socket for the QED Board to operate. This memory cannot be write-protected. If a 32K RAM is installed, the onboard logic places 20K of it on page fifteen (0FH) at addresses 3000H to 7FFFH and the remainder in common memory at addresses 8100H-ADFFH (but 8100H-83FFH is reserved for use by QED-Forth). An additional 1K of common memory is available at addresses B000H-B3FFH; this is the 68HC11's on-chip RAM. The total amount of common RAM available to the programmer (including the 68HC11 on-chip RAM and excluding the common RAM reserved by QED-Forth) is 11.5K. If a 128K RAM is installed in socket S2, the three 32K pages numbered 1, 2, and 3 are also available.

Socket S3 is called the "RAM/ROM socket". It accommodates either RAM (typically during program development) or PROM (typically in your final product). During program development and debugging, a 128K battery-backed RAM is typically installed in this socket. The memory is addressed at pages 4, 5, 6, and 7, and each page contains 32 Kbytes. Turning onboard DIP switch #1 ON write-protects pages 4 and 5, and turning DIP switch #2 ON write-protects pages 6 and 7.

This socket is an ideal location for the memory areas such as the definitions and name area that will end up as PROM in the final system. During program development, code can be compiled into the RAM in this socket, and the memory can be write-protected to emulate a PROM. When the application is completely debugged, the contents can be burned into a PROM, and the PROM can be plugged into the socket. Turning dip switch #3 ON configures the socket for a PROM, and the system is complete. The PROM size can be 32K (page 4 only), 64K (pages 4 and 5), or 128K (pages 4, 5, 6, and 7).

Setting Up the Memory Map

To set up the memory map for a turnkeyed application program, the following factors should be taken into account:

1. The amount of memory space required by the application for each of the four main memory areas (definitions, names, variables, and heap).
2. Whether or not the name area must be present in the final application.
3. Memory requirements for task areas in common RAM (see the "Multitasking" Chapter).
4. The amount and location of available PROM on the final production board which must accommodate the definitions and name area (if present).
5. The amount and location of available RAM on the final production board which must accommodate the variable area, heap area, task areas, stacks and buffers.

If a 20K definitions area and 12K name area in PROM and a 4K variable area and 16K heap in RAM are adequate, then executing the simple commands

```
4 USE.PAGE
ANEW  APPLICATION.NAME  \ choose any name you like here
```

after a COLD restart conveniently initializes the memory map. The resulting memory map is consistent with the minimum onboard memory configuration which is:

QED kernel ROM in socket S1

32K RAM in socket S2

32K application PROM in socket S3

The name area can be present in the final 32K PROM. If the names of the routines are not needed in the final production instrument, we can simply neglect to transfer the names area to the PROM. If the name area is not needed in the final application and we would like to use an entire 32K definitions area, the following code could be used after a COLD restart to set up the memory map:

```
HEX
0000 4 DP X!   \ set 32K definitions area on page 4
0000 5 NP X!  \ set 32K name area on page 5
3000 F VP X!   \ 4K variable area on page fifteen
4000 F 7FFF F IS.HEAP \ 16K heap on page fifteen
ANEW  APPLICATION.NAME  \ choose any name you like here
```

Notice that we execute the ANEW command after we have set up the memory map but before we have defined the first entry in the dictionary.

To gain a comprehensive understanding of the creation of a turnkeyed program, it is strongly recommended that you read the final chapter of this manual titled “Putting It All Together: A Turnkeyed Application Program”.

Splitting the Dictionary to Write-Protect Debugged Code

As with all computers, executing “buggy” code can cause the QED Board to “crash”, which means it executes a set of instructions that it is not supposed to. This may cause the processor to write over and corrupt memory locations that are not write-protected. If the dictionary is in RAM, definitions may be corrupted. In some cases executing a buggy program can corrupt a small portion of the dictionary and lead to hard-to-diagnose problems. Write-protecting the dictionary can minimize these difficulties.

If you have spent several days debugging a portion of your program, a single crash can corrupt all of your code if the full dictionary is in modifiable memory (RAM). After such a crash, you would be forced to download all of your code to re-establish the dictionary.

The split-dictionary memory map solves this problem. The memory map can be configured so that the portion of the dictionary that has been tested and debugged resides in a write-protected memory page that cannot be corrupted by a crash. The variable and heap areas plus the portion of the dictionary that is now being debugged are placed in a non-write-protected page of RAM. Then a crash can corrupt only the portion of the dictionary that you are debugging, and cannot harm the write-protected code. As more words are debugged, they can be transferred to the write-protected page of memory.

Using SAVE and RESTORE to Recover from a Crash

If a crash results in a cold restart, you'll want to regain access to the words in the write-protected dictionary. The kernel words SAVE and RESTORE provide this capability. SAVE stores in EEPROM the contents of 5 key user variables: DP, NP, VP, CURRENT.HEAP, and the top link in the FORTH vocabulary. The latter typically specifies the last name added to the dictionary.

Execute SAVE after you have compiled your definitions into RAM. Then you can flip the DIP switch to write protect the definitions. This saves the key pointers in EEPROM. After a crash, executing RESTORE will fetch the saved pointers and put them in their proper places in the user area. This restores the memory map and dictionary to the state they were in when SAVE was executed, giving you access to the write-protected portion of the dictionary. You can continue debugging without having to re-download all of your code.

An Example of Dictionary Splitting

Let's start with a clean slate by executing

```
COLD
```

to initialize the user area and memory map. Let's say we need a 16K heap and a 4K variable area, and we can leave the return and data stacks, TIB, POCKET, and PAD at their default locations in common memory. In addition, assume that we want to be able to write protect page 4 after a number of words have been defined and debugged, and then continue compiling definitions in RAM on page 6. To set up a useful memory map we can execute

```
4 USE.PAGE
```

which allocates a 4K variable area and 16K heap on page fifteen, and a 20K definitions area and 12K name area on page 4. Recall that DIP switch#1 controls the write-protection of pages 4 and 5; it should be OFF to enable compilation of new definitions on page 4. Then we can start compiling code into page 4, starting with an ANEW statement, as usual. For example,

```
ANEW SPLIT.DICTIONARY

MATRIX: 1MAT
INTEGER: 1INT

: 1ST.WORD  ( -- )
  ." I'm the first word"
;
```

and so on. The code and names of these words are being compiled on page 4. The parameter fields of 1MAT and 1INT are in the variable area in page fifteen. Now assume that we've compiled and debugged enough words on page 4, and we want to write protect them so a crash can't corrupt the definitions. Simply execute

```
SAVE
```

to save the state so that it can be restored after a crash. Then dip switch #1 can be flipped to the "on" position to write-protect pages 4 and 5.

To move the dictionary pointers to page 6, execute

```
HEX
0000 6 DP X!      \ start of the definitions area; 20K
5000 6 NP X!      \ start of the name area; 12K
```

and then some words can be defined on page 6:

```
ANEW PAGE6.WORDS

: CRASH  ( -- )
  UP @ 0 100 ERASE
;
```

But when CRASH is executed, the system crashes and forces a cold restart (try it). This happened because the user area that holds all of QED-Forth's key variables was over-written with zeros. Now neither CRASH nor any of the words we defined on page 4 is in the dictionary. This can be verified by executing WORDS which lists the words in the dictionary starting at the last one defined. Type a carriage return to abort the listing of the WORDS. But all is not lost. The definitions on page 1 are intact. To regain access to them, execute

```
RESTORE
```

which restores the memory map and the vocabulary link to the states they had when SAVE was executed. Typing WORDS again will confirm that the page 4 definitions are present. Now we can continue with the debugging session where we left off, starting with the commands

```
HEX
0000 6 DP X!      \ start of the definitions area; 20K
5000 6 NP X!      \ start of the name area; 12K
ANEW PAGE6.WORDS
```

If another crash occurs, simply execute RESTORE again to restore the state that existed at the last execution of SAVE.

When a fair number of words on page 6 have been debugged you may want to write protect them also by compiling them on page 4 after the words that are already there. Turn DIP switch#1 OFF to make page 4 write-able again. Execute RESTORE to restore the machine to the state it had just after the last word was defined on page 4. Then download the source code for the additional debugged words; they will be compiled onto page 4. Execute SAVE to save this new state. If your application is finished, you may want to set up the top level word as a PRIORITY.AUTOSTART program. Flip DIP switch#1 ON to write-protect your code. If you are not finished and want to debug more words, you would again initialize NP and DP to page 6 locations, execute ANEW, and start defining new words on page 6.

Summary of the Split-Dictionary and SAVE/RESTORE Technique

In summary, to use the split dictionary scheme, assuming for simplicity that page 4 is the write-protectable page and page 6 is RAM that can accommodate additional definitions during debugging:

1. Decide how much memory is needed for heap, variables, names, etc.
2. Set the memory map, making sure that the heap and variable area are in un-write-protected RAM.

3. Compile definitions in page 4.
4. When ready to change dictionary pages, execute SAVE, write-protect page 4, and set the definitions and name pointers to page 6.
5. Execute ANEW and define additional words on page 6.
6. To recover from a crash, execute RESTORE, set the definitions and name pointers to page 6, then repeat step 5.

The Heap Memory Manager

Like many workstations but unlike most microcontroller systems, QED-Forth provides a heap memory management system. QED-Forth's advanced array and matrix math routines make extensive use of the heap memory manager, but they do it transparently. To use these features you need only a cursory familiarity with the information in this chapter. On the other hand, if you want to create your own sophisticated dynamically allocated data structures, you should read this chapter carefully.

A "heap" is a pool of memory from which smaller blocks of memory of variable size are allocated. The heap manager allocates blocks of memory requested by the user's program from this pool. When the program is finished with the block of memory, it can return it to the heap. This results in efficient use of memory, especially for data structures which can vary in size and for temporary data structures which are used and then de-allocated. The de-allocated memory becomes available for use by other segments of the program. QED-Forth can maintain any number of heaps simultaneously, each containing numerous data structures.

Those interested in a the implementation of a heap manager may consult the final chapter in Object Oriented Forth--Implementation of Data Structures, by Dick Pountain, Academic Press, 1987. QED-Forth's implementation of the heap manager is similar to that proposed by Pountain. It is also similar to that used by the Apple Macintosh computer as described in Apple's Inside Macintosh publications.

Heap Compacting

The heap manager works by allocating blocks of memory beginning at the designated starting address of the heap. When a heap item is de-allocated and returned to the heap, its memory is free for use by other programs. A simple-minded heap manager which simply keeps track of which areas of the heap are used and which are free encounters the problem of heap fragmentation. For example, if heap items #1, #2, and #3 are sequentially allotted memory, they will occupy blocks of memory starting at the bottom of the heap. Then if item #2 is de-allocated, the heap's free memory will consist of two disjoint sections: the memory that was occupied by item #2, and the memory above item #3. As this process of allocation and de-allocation continues, the heap gets fragmented into smaller available pieces so that less of the heap consists of contiguous memory. This makes it difficult or impossible for the heap manager to respond to requests for large blocks of contiguous memory.

To solve this problem, the QED-Forth heap manager "compacts the heap" each time a heap item is de-allocated. This involves re-arranging the blocks in the heap so that all of the free heap memory is available in one contiguous block above the allocated heap items.

Handles to Heap Items

The problem then becomes one of keeping track of where the items are in the heap. The process of compaction moves the heap items so that the base addresses of the heap items change. But the program that requests a heap allocation must have a reliable address that can be used to refer to the heap item. The heap manager solves this problem by adding a level of indirection in the addressing. Instead of informing the requesting program of the actual base address of the heap item when it is allocated, the heap manager returns to the requesting program a "handle" (also called an "xhandle") which is an extended address that contains the extended base address of the heap item. As long as a particular heap item is allocated, its handle does not change and can be used by the requesting program. The heap manager changes the handle's contents appropriately as the heap is compacted, and the requesting program can fetch the contents of the handle at any time to obtain a valid base xaddress for the heap item.

Thus the problems of heap fragmentation and the changing of base addresses are solved by compacting the heap and using handles instead of direct addresses.

Heap Size

In QED-Forth, the maximum size of a heap is limited only by the amount of available contiguous RAM. A heap can flow over page boundaries. Likewise, the size of a data structure in the heap is limited only by the available memory in the heap.

The only heap limitation is that the list of handles which is maintained near the top of the heap must be on the last page of the heap, and the handle list cannot flow over a page boundary. Because each handle requires only 4 bytes, this poses very little limitation for most practical applications.

The following implementation details clarify how the heap works. The next chapter discusses how array and matrix data structures use the heap to facilitate dynamic dimensioning and memory allocation.

Initializing the Heap

To create a heap, specify a starting extended address and an ending extended address and execute IS.HEAP. For example, let's designate the 20 Kbyte RAM area from 3000H to 7FFFH on page 15 as the heap by executing

```
HEX  3000 F  7FFF F  IS.HEAP  ok
```

This command sets the user variable CURRENT.HEAP in the user area to 7FFF\FH, and initializes the heap management variables on the heap page to indicate that START.HEAP is at 3000\FH and

there are no allocated heap items. The hexadecimal number base is more convenient than decimal for specifying memory map address locations, so it will be used for the remainder of the chapter.

The user variable called CURRENT.HEAP contains an extended address that specifies the address and page of the top byte + 1 in the heap. The heap allocation begins in low memory at the extended address pointed to by START.HEAP; a variable named HEAP.PTR holds the extended address of the next byte available for allocation. The contents of START.HEAP and HEAP.PTR as well as 2 other variables that manage the heap (HANDLE.PTR and FREE.HANDLE) are kept in the heap area, just below the specified end of the heap. The handles are kept below these variables in the heap.

The kernel word ROOM calculates the amount of space available in the current heap and returns it as a 32-bit number on the stack:

```
ROOM D.  4FE7  ok
```

which is nearly 20 Kbytes, as expected. When all available heap memory has been allocated and no more items can be added to the heap, ROOM returns 0\0.

Allocating Heap Items

To request a block of 1FEH bytes from the heap execute

```
DIN 1FE  FROM.HEAP  ok  [ 2 ] \ 7FEF \ F
```

which expects a double number size in bytes and returns the extended address of the handle for the heap item. A non-zero handle indicates that the heap manager successfully allocated the memory. The requesting program should save this handle because its contents contain the base address needed to access the data in the heap item. It can be saved in a self-fetching variable whose name corresponds to the function of the heap item. For example,

```
XADDR: 1ST.DATA.BLOCK  ok  [ 2 ] \ 7FEF \ F

TO 1ST.DATA.BLOCK  ok  \ save the handle in the variable
```

A handle of 0\0 is returned if there is not enough memory in the heap to allocate the item. None of the heap manager words ABORT if an error is detected; rather, they signal the error by passing a zero handle or an error flag to the calling program. It is the calling program's responsibility to take the appropriate response in case of an error.

If the requested size passed to FROM.HEAP is not an even multiple of 4 bytes, the heap manager rounds the size up to the next 4-byte multiple. In this case, the size is rounded up to 200H which is 2 bytes more than the requested size. In addition, the heap manager ensures that the base address of each heap item is an even multiple of 4 bytes. These steps speed heap compaction and also assure proper operation of the fast vector and matrix arithmetic functions which require all elements to be aligned on 4-byte boundaries.

To obtain the base address of the heap item, fetch the extended base address from the handle:

```
1ST.DATA.BLOCK X@.    ok  ( 2 ) \ 3004 \ F

SP!.    ok                          \ clear the data stack
```

The contents, 3004H on page FH, specify the base address of the heap item at this time. The heap manager saves the 32-bit size of the heap item in the four bytes below the base address. The word ?HANDLE.SIZE expects the extended handle address on the stack and returns the 32-bit size of the heap item:

```
1ST.DATA.BLOCK  ?HANDLE.SIZE  D.    200   ok
```

which prints 200H as expected.

The word .HANDLES prints the status of the current heap:

```
.HANDLES

Size      Addr        Handle

200    3004\ F    7FEF\ F

ok
```

This shows that one handle (7FEF\FH) has been allocated with current base address 3004\FH and size 200H. .HANDLES always prints in hexadecimal base, regardless of the current number conversion base.

Another heap item can be allocated as

```
XADDR: 2ND.DATA.BLOCK    ok

DIN 400 FROM.HEAP  ok  [ 2 ] \ 7FEB \ F

TO 2ND.DATA.BLOCK    ok
```

The heap manager returns a handle 7FEB\FH and again the allocation is successful.

De-allocating Heap Items

To de-allocate the first heap item heap item execute

```
1ST.DATA.BLOCK   TO.HEAP .    FFFF   ok
```

TO.HEAP expects a handle xaddress on the stack, and returns a success flag. A true flag indicates that the handle is valid and that the heap item has been successfully de-allocated, and a false flag indicates that the handle was invalid and no action was taken. Execute .HANDLES to see the effect of the de-allocation:

```
.HANDLES

Size      Addr        Handle
```

```
400    3004\ F    7FEB\ F
*              0\ 0    7FEF\ F
ok
```

The handle of 1ST.DATA.BLOCK that was returned to the heap is shown with an asterisk to indicate that it is not in use (because it has been de-allocated) and that the listed base address and size are not significant. The handle of 2ND.DATA.BLOCK has not changed, but its base address has been changed to 3004\FH. It was moved to the bottom of the heap during the heap compaction that occurred when 1ST.DATA.BLOCK was de-allocated. The heap manager moved the memory and adjusted the contents of the handle to implement the heap compaction. All of the available memory is maintained as one contiguous block above the last allocated heap item.

Maximizing the Efficiency of Heap Compaction

Heap compaction occurs each time an item is returned to the heap. All of the bytes above the de-allocated item are moved down in memory. This move requires 17 microseconds per byte, or 17 milliseconds per Kbyte. Notice, however, that compacting the heap to de-allocate the last item allocated is accomplished by simply adjusting the heap pointer; there is no need to move a block of memory in the heap. When possible, programs should de-allocate heap items in the proper order so that the last item allocated is the first item de-allocated; this maximizes execution speed. For example, when allocating several temporary matrices to hold the intermediate results of a calculation, de-allocating them in the proper order improves performance.

Resizing and Copying Heap Items

RESIZE.HANDLE expects a valid 32-bit handle under a 32-bit number of bytes on the stack, and tries to resize the heap item to the specified size, preserving as much of the item's data as possible. The heap must have enough room to copy the heap item. A flag is returned to indicate whether the resizing was successful.

DUP.HEAP.ITEM expects a handle xaddress on the stack, and creates a copy of the specified item in the current heap. The copy's handle is returned if the duplication was successful; otherwise 0\0 is returned. For example, to duplicate 2ND.DATA.BLOCK, execute

```
XADDR: 2ND.COPY␣    ok      \ create a name for the new heap item
2ND.DATA.BLOCK DUP.HEAP.ITEM␣  ok  [ 2 ] \ 7FEF \ F
TO 2ND.COPY␣    ok              \ save the handle
```

To verify that the heap item has been duplicated, execute .HANDLES

```
.HANDLES␣
Size      Addr       Handle
400    3004\ F    7FEB\ F
```

```
400   3408\ F   7FEF\ F

ok
```

The heap manager has re-used the handle from the recently de-allocated word 1ST.DATA.BLOCK and assigned it to the newly allocated item.

Multiple Heaps

In a timesliced multitasking environment where several tasks are using heap memory, each task must have its own separate heap. If tasks share a single heap, an interrupting task could compact the heap and change the contents of a handle that is about to be used by another task. Thus multitasking environments may require multiple heaps. There may be other cases where it is advantageous for a single task to have multiple heaps.

To use multiple heaps, the programmer must manage the user variable CURRENT.HEAP. To see how this works, let's set up a second heap with a size of 8 Kbytes that starts on page 6 at address 7000H and ends on page 7 at 1000H. Check to be sure that a 128K RAM is installed in the RAM/ROM socket of your board so that this memory area is available, and turn DIP switch #2 OFF to make sure that pages 6 and 7 are not write protected. Then execute:

```
7000 6  1000 7 IS.HEAP□    ok \ set up new heap

XADDR:  1ST.HEAP□    ok

XADDR:  2ND.HEAP□    ok    \ define save variables for CURRENT.HEAP

7FFF F TO 1ST.HEAP□    ok

1000 7 TO 2ND.HEAP□    ok    \ initialize the save variables
```

The self-fetching variables 1ST.HEAP and 2ND.HEAP hold the values of CURRENT.HEAP that specify the two heaps. Recall that the contents of the 32-bit user variable CURRENT.HEAP specifies the heap in which heap items are to be allocated and de-allocated. Thus manipulating the single extended address in CURRENT.HEAP enables multiple heaps to be managed. For example, to make the heap on page F the current heap, execute

```
1ST.HEAP CURRENT.HEAP X!
```

and to make the heap on pages 6 and 7 the current heap, execute

```
2ND.HEAP CURRENT.HEAP X!
```

Transferring Heap Items

TRANSFER.HEAP.ITEM copies a heap item into any specified heap on any page. For example, to transfer a heap item from the 1ST.HEAP into 2ND.HEAP, put the handle of the source heap item and the destination CURRENT.HEAP on the stack and execute TRANSFER.HEAP.ITEM as

```
XADDR: 3RD.DATA.BLOCK .  ok

2ND.DATA.BLOCK 2ND.HEAP TRANSFER.HEAP.ITEM .  ok [ 2 ] \ FF0 \ 7

TO 3RD.DATA.BLOCK .  ok            \ save the new handle
```

The new item resides in 2ND.HEAP and its handle is saved in 3RD.DATA.BLOCK. The transfer was successful; otherwise, TRANSFER.HEAP.ITEM would have returned a handle of 0\0 .

To verify the status of 2ND.HEAP, execute

```
2ND.HEAP END.HEAP X!       \ set current heap

.HANDLES .

Size      Addr       Handle

400    7004\ 6    FF0\ 7

ok
```

which shows that the heap item was copied to 2ND.HEAP .

Heap-Based Data Structures

Data structures that are maintained in the heap are usually created by "defining words" such as ARRAY: and MATRIX:. These defining words create and initialize a parameter field in the variable area, as described in the next chapter. The contents of the parameter field are filled in when the data structure is dimensioned or allocated. When the data structure is dimensioned it is assigned to the heap specified by CURRENT.HEAP. At this same time the dimensioning information (e.g., number of rows and columns), the handle (which contains the base xaddress of the data) and the value of CURRENT.HEAP are saved in the parameter field associated with the data structure. Based on the information in the parameter field, the memory manager can resize, copy, or de-allocate the heap item irrespective of the current contents of the user variable CURRENT.HEAP.

For example, the word DELETED which de-allocates arrays and matrices automatically saves CURRENT.HEAP, looks at the parameter field of the item to be deleted, puts its heap specification into the user variable CURRENT.HEAP, executes TO.HEAP to de-allocate the item, and restores the original value of CURRENT.HEAP. Thus arrays and matrices in any heap can be deleted. Other array and matrix operations perform similar user-transparent manipulations of CURRENT.HEAP to make the words powerful and remove book-keeping chores from the programmer.

Because both the heap and the variable area where the parameter field is stored are in modifiable memory (RAM), the data structure can be allocated, re-dimensioned, or de-allocated "on the fly" as the program executes. This is a powerful programming capability. It is the basis for the comprehensive matrix mathematics package which is discussed in detail in the next chapter.

Using Arrays

QED-Forth provides a set of data structures and associated operations that simplify and streamline sophisticated processing tasks. Arrays and matrices can be easily created, dynamically dimensioned, deleted, initialized, and copied. Matrices, which are 2-dimensional arrays of floating point numbers, support a comprehensive set of pre-programmed data editing and matrix algebra operations. This chapter describes how to define and use Arrays. Matrices are discussed in a following chapter.

An array is a data structure stored in memory whose elements can be individually addressed and accessed. QED-Forth supports dynamically dimensioned arrays whose elements are stored in heap. The array can be redimensioned as needed under program control and the array's storage space can be dynamically re-allocated. Consult the "Array" section of the Categorized Word List in the QED-Forth Glossary for a list of available routines in the array library.

Array Size and Dimensions

The number of elements per dimension must be between 1 and 16,383, and the number of bytes per element must be between 1 and 65,535. The size of an array is limited by the amount of space available in the current heap. The size of the heap, in turn, is limited only by the amount of available contiguous RAM. The heap may occupy multiple contiguous pages.

The value stored in the user variable MAX#DIMENSIONS determines the maximum number of dimensions in an array; the default is 4, but you can specify any number up to a maximum of 255. The higher the contents of MAX#DIMENSIONS, the more memory is required in the array's parameter field which holds the dimensioning information of each array. Each additional dimension requires 2 additional bytes in the parameter field.

If you wish to change the default value of MAX#DIMENSIONS, it is recommended that you do so before any arrays are defined. Unpredictable behavior may result if arrays are defined while MAX#DIMENSIONS is small, and then it is increased. This is because the value of MAX#DIMENSIONS determines the size of the array's parameter field when the array is created. If MAX#DIMENSIONS is later increased, operations on the array will assume a larger size parameter field, perhaps writing over memory that is associated with the parameter field of another array.

Creating Arrays

To create a named array execute ARRAY: followed by a name of your choice, as

```
ARRAY: ARRAY.NAME  ok
```

The array is created but no heap memory is assigned to it yet. Memory is allocated and the array is dimensioned using the command DIMENSIONED which requires the limits of the indices under the number of indices under the number of bytes per element under the array's extended parameter field address (xpfa). There can be up to MAX#DIMENSIONS indices (the default is 4). For example to

dimension an already named array to have 2 dimensions, 5 rows and 10 columns, with 2 bytes per element, execute

```
DECIMAL␣  ok                    \ set base to decimal

5 10 2 2  ‘ ARRAY.NAME   DIMENSIONED␣  ok
```

where ‘ (pronounced “tick”) places the extended parameter field address on the stack; the xpfa is used to refer to the array as a whole. Memory for the array is allocated in the current heap. If the heap has insufficient room for the specified array, QED-Forth will ABORT and print the message:

```
ARRAY.NAME is too large. Not enough memory !
```

Addressing Array Elements

Individual elements are referred to by placing the element’s indices on the stack and stating the array’s name; the element’s extended address is then left on the stack. The element size and all indices are 16-bit unsigned integers, and indices are zero-based. This means that for a two dimensional array with 5 rows and 10 columns the rows are numbered 0 through 4 and the columns are numbered 0 through 9. To store the integer 104 at the second row (row#1) and seventh column (column#6) you can say

```
104  1 6 ARRAY.NAME !␣  ok
```

and to fetch and print the value from the same location you can execute,

```
1 6 ARRAY.NAME @ .␣  104  ok
```

If DEBUG is ON, indices are checked to insure that they are within their limits. Out-of-range indices cause an ABORT and an error message. For example,

```
5 6 ARRAY.NAME @␣

Error at []  ARRA_____  5 is an out-of-range index !  ok
```

The error message reports the routine that detected the error (which is [], a word that calculates an address given the indices), the name of the array, the offending index, and the reason for the ABORT. Note that only the first 4 letters of the array’s name were printed. This is because we assume that the user variable WIDTH equals its default value of 4. The remainder of the name is represented by the underbar character. (To improve the reporting of names in error messages, you can increase WIDTH with a simple ! command).

If DEBUG is OFF, error checking is not done, and array element addressing occurs more rapidly.

References to Entire Arrays

An entire array can be referred to, as opposed to referring to only an individual element, by using the kernel word ‘ (tick) before the name. Executing ‘ followed by the array’s name leaves the 32-bit extended parameter field address (xpfa) of the array on the stack; this address must be used when-

ever an entire array is treated as a single object. It is useful when you need to pass an entire array to a FORTH word which then operates on all elements of the array rather than on just a single element. References to arrays can be passed from one word to another without the called word needing prior knowledge of the array's name.

Operations for Entire Arrays

An entire array can be initialized to zero as

```
‘ ARRAY.NAME  ZERO.ARRAY  ok
```

Suppose a character array is needed to hold a character string. To define and dimension a 1-dimensional array with 80 single-byte elements, execute

```
ARRAY: CHARACTER.ARRAY  80 1 1 ‘ CHARACTER.ARRAY DIMENSIONED  ok
```

and then initialize it to contain ascii blanks as

```
‘ CHARACTER.ARRAY  BLANK.ARRAY  ok
```

The array can be initialized to contain any specified character using FILL.ARRAY. For example, to initialize an array to hold ascii zeros (whose code is 48), execute

```
‘ CHARACTER.ARRAY  ASCII 0  FILL.ARRAY  ok
```

Try fetching and printing the first element of the array to verify that the operation worked:

```
0 CHARACTER.ARRAY  C@ .  48  ok
```

Note that C@ was used to retrieve a single byte from the array; @ would have retrieved two successive bytes.

The word SWAP.ARRAYS expects the xpfa's of two arrays on the stack. It swaps the contents of the parameter fields of the arrays (which contain dimensioning information and an indirect pointer to the contents) without moving the contents in the heap. Thus the swap is accomplished very rapidly.

The contents of one array can be copied to another array, irrespective of the prior dimensions or contents of the destination array, using COPY.ARRAY. For example, to copy ARRAY.NAME to CHARACTER.ARRAY, execute

```
‘ ARRAY.NAME  ‘ CHARACTER.ARRAY  COPY.ARRAY  ok
```

Deleting an array de-allocates its memory in heap, undimensions it by clearing its parameter field, and leaves only its name remaining. It is accomplished by executing

```
‘ CHARACTER.ARRAY  DELETED  ok
```

Determining a Pre-existing Array's Dimensions

?ARRAY.SIZE returns the 32-bit number of elements (not number of bytes!) of an array:

```
‘ ARRAY.NAME  ?ARRAY.SIZE  D.  50  ok
```

and the limits of all its indices are returned by

```
‘ ARRAY.NAME  ?DIMENSIONS  [ 4 ] \ 5 \ 10 \ 2 \ 2  ok
```

which are the dimensions we specified. The stack may be cleared using

```
SP!  ok
```

Internal Representation of Arrays

This section is provided for those interested in understanding the definitions of array defining words. You need not read or understand this section to use arrays.

An array has four parts:

Name: Stored in the name area of the dictionary.

Action: Performed by the FORTH word [] which is automatically called when the array's name is executed with indices on the stack; the extended address of the specified array element is placed on the stack. Error checking is done if DEBUG is ON.

Parameter field: Holds the limits for each of the array indices, the number of dimensions, the element size, a handle to the array's storage space in the heap, and a reference to CURRENT.HEAP. The parameter field is stored in the variable area in RAM.

Contents: Stored in the heap as a relocatable block of memory.

After an array name is created, the array dimensions (index limits) and element size are stored in the parameter field and the array elements are stored in the heap.

The contents of the parameter field are as follows:

Address	Description
pfa+0	32-bit extended handle to the array's contents in heap
pfa+4	addr of CURRENT.HEAP (page is same as handle's)
pfa+6	size in bytes of each element
pfa+8	number of dimensions; 255 is maximum allowed
pfa+10	limit of first index (#cols); for 0,1,2,...n : limit=n+1
pfa+12	limit of second index (#rows)

pfa+14 limit of third index, etc....

...

pfa+10+2*MAX#DIMENSIONS limit of last index

If the handle is zero then the array is undimensioned. The element size and all indices are 16-bit unsigned integers. Indices are all zero-based. The command

ARRAY: <name>

creates a dictionary entry for <name> and initializes its dimensions to 0. It does not allocate heap space. Memory is allocated when arrays are dimensioned or redimensioned using DIMENSIONED. For example:

#index.n #index.n-1 ... #index.1 n element.size ‘ <name> DIMENSIONED

allocates sufficient memory from the heap for the array and stores the handle and dimensions in the parameter field of <name>. Any prior contents of the array are deleted. The command

‘ <name> DELETED

de-allocates memory for the array and returns its handle to the heap manager.

Note that executing an X@ from the xpfa of the array returns the extended address of the handle to the heap item; executing an X@ from this handle returns the base xaddress of the heap item. As discussed in the heap chapter, the handle remains valid as long as the array is dimensioned, but the base xaddress (the contents of the handle) may change every time the heap is compacted. The array words take care of address calculation for you so you don’t have to perform explicit memory fetches to find the address of an array element. Just place the indices on the stack and state the array’s name to obtain an element’s address.

Using Structures

A “structure” is an object that groups different types of data according to a template designed by the programmer, and allows the programmer to designate names that can be used to store and retrieve the data items in the structure. While arrays hold data that is all one size and refer to the data elements using index numbers, structures let you group data of different sizes and refer to them using names of your choice. The structure defining words declare the type of the data (integer, real, address, extended address, etc.) and this enhances the clarity of the program.

Using structures involves defining them, instantiating them, and addressing their elements which are called “members”. Defining a structure can be thought of as setting up a template for the data. The template comprises a set of named offsets that specify the position of each member relative to the base of the structure. Instantiating a structure creates an instance of the structure and assigns it to a particular named memory location, either in the variable area, the definitions area of the dictionary, or in the heap. Addressing a member in a structure is accomplished by stating the name of the structure instance followed by the name of the structure member. Executing the name of the structure instance leaves its base xaddress on the stack, and then executing the name of the structure

member automatically adds an offset to the base xaddress to produce the extended address of the member. Standard fetch and store operations can then be used to access the member.

Defining Structures

To define a structure, execute STRUCTURE.BEGIN: followed by the name of the structure. Then state the appropriate member defining commands followed by member names that you choose. STRUCTURE.END finishes the definition.

For example, the following commands create a structure to hold customer information:

```
STRUCTURE.BEGIN: CUSTOMER.INFO      \ name the structure as a whole

INT->   +ACCOUNT#   \ now name each of the members

40 STRING->   +CUSTOMER.NAME

50 STRING->   +STREET

35 STRING->   +CITY&STATE

DOUBLE->      +ZIP.CODE

REAL->        +ACCT.BALANCE

STRUCTURE.END      \ end the structure definition
```

This creates a template for the data. If you try this example on your computer, you will notice that while the structure is being compiled, items are temporarily placed on the data stack by the compiler. These stack items should not be altered; they are used to initialize the size of the structure and the offset of the members.

The member defining word INT-> removes the next name from the input stream, +ACCOUNT#, and creates a header in the name area of the dictionary. The action of +ACCOUNT# is to add an offset to an extended address on the top of the stack. Because +ACCOUNT# is the first member in the structure, it adds an offset of 0. (Actually, members with offsets of 0 are smart enough to do nothing at run time, thereby saving execution time). Note that the member defining words end with -> to suggest that they are defining the name that follows. Note also that we start each member name with + to suggest that it adds an offset to the quantity on the stack. The defining word STRING-> creates a header and action for CUSTOMER.NAME and, based on the quantity on the stack, reserves space in the structure for the string. It reserves 1 byte more than the number on the stack to allow for a count byte at the start of the string. In the example structure two more string members are defined, then a double number field is reserved for the zip code, and a floating point field is reserved for the customer's account balance.

Executing CUSTOMER.INFO leaves the size of the entire structure (in bytes) on the stack; this structure takes 138 bytes.

Here is a list of the available member defining words:

n RESERVED reserves unnamed space within a structure of n bytes in size

n MEMBER-> defines a member word with n bytes reserved

BYTE-> defines a single byte item

n BYTES-> defines an item of n bytes in size

INT-> defines a 2-byte (16-bit) integer

n INTS-> defines n 2-byte integer numbers

DOUBLE-> defines a 4-byte double number

n DOUBLES-> defines a collection of n 4-byte double numbers

REAL-> defines a 4-byte floating point number

n REALS-> defines a collection of n 4-byte floating point numbers

ADDR-> defines a 2-byte (16-bit) address

n ADDRS-> defines n 2-byte addresses

PAGE-> defines a 2-byte page, only the lower order byte is significant

XADDR-> defines a 4-byte extended address

n XADDRS-> defines a collection of n 4-byte extended address

XHNDL-> defines a 4-byte xaddr whose contents are a handle

n STRING-> defines a counted string n+1 bytes long, n <= 255

n STRUCT-> defines a member word with n bytes reserved

n1 n2 STRUCTS-> defines a collection of n1 structures each n2 bytes in size

Most of these words call the kernel word FIELD which creates a named offset that, when executed, calls XU+ to add the offset to the extended address on the top of the stack.

Creating Instances of Structures

To create a structure instance in the variable area for a particular customer named BURGER.WORLD, use V.INSTANCE: as

```
CUSTOMER.INFO   V.INSTANCE:   BURGER.WORLD
```

CUSTOMER.INFO leaves the size of the structure on the stack. V.INSTANCE: (where the “V” indicates the variable area) reserves room in the variable area for the structure. It creates a definition for BURGER.WORLD that, when executed, leaves the extended address of the structure instance on the stack.

To initialize the BURGER.WORLD's account number to 1234, we can execute

```
1234 BURGER.WORLD +ACCOUNT#  !
```

BURGER.WORLD leaves the extended address of the structure instance on the stack, +ACCOUNT# adds its offset to yield the extended address of the member, and ! saves the value 1234 into the specified member. Similarly, CMOVE can be used to initialize the name, street, and city strings, 2! can initialize the double number zip code, and F! can set the floating point account balance.

If we want a structure to eventually reside in write-protected or PROM memory in the definitions area instead of in the RAM variable area, we execute

```
CUSTOMER.INFO   D.INSTANCE:   MARTHA'S.PIES
```

where the "D" in D.INSTANCE means that space for the structure is reserved in the definitions area.

Executing the word SIZE.OF followed by the name of a structure instance leaves the size of the instance on the stack. For example

```
SIZE.OF   BURGER.WORLD  .   ok  138
```

which is the size of the CUSTOMER.INFO structure.

Note that structure instances are allowed to cross page boundaries, and executing V.INSTANCE: or D.INSTANCE: may cause the variable pointer VP or the dictionary pointer DP, respectively, to be advanced to a new page.

Heap Instances

It is sometimes useful to instantiate structures in the heap, and this involves a slightly different procedure. To create a named instance of the structure, execute

```
CUSTOMER.INFO   H.INSTANCE:   JOE'S.PIZZA
```

Like the previous commands, this creates a definition for JOE'S.PIZZA that, when executed, leaves the base address of the instance on the stack. And SIZE.OF can be used to retrieve the size of the heap instance. However, unlike the previous commands, no memory has yet been assigned. Instead, a parameter field for JOE'S.PIZZA has been created in the variable area, and the size of the structure has been saved with the definition of JOE'S.PIZZA. To allocate memory in heap, we can execute

```
SIZE.OF JOE'S.PIZZA  ' JOE'S.PIZZA   ALLOCATED
```

The word SIZE.OF removes the next word from the input stream, examines its definition to recover the size (which was put there by H.INSTANCE:) and leaves the size on the stack. ' JOE'S.PIZZA leaves the extended parameter field address on the stack, and ALLOCATED converts the size to a

double number and calls FROM.HEAP to allocate memory for the structure. Note that we could have replaced the command SIZE.OF JOE'S.PIZZA with CUSTOMER.INFO, as both leave the size of the structure on the stack.

To de-allocate the heap instance, simply execute

```
‘ JOE’S.PIZZA DEALLOCATED
```

which returns the memory to the heap manager and clears the parameter field. The definition of the structure remains intact for future use.

Nested Structure Definitions

Structures can be nested. For example, the following is a valid structure definition:

```
STRUCTURE.BEGIN: ADDRESS.LIST
BYTE->          +FLAG
STRUCTURE.BEGIN: SUB
20 STRING->   +NAME
20 STRING->   +ADDRESS
INT->   +ID#
STRUCTURE.END
2 SUB STRUCTS->      +NAME&ADDRESSES
STRUCTURE.END
```

Embedded within the definition of ADDRESS.LIST is another structure definition for SUB. The member defining word STRUCTS expects on the stack the number of structures and the size of the structures. It names a member with the appropriate offset, and reserves space for the sub-structures in the main structures. Strict hierarchy must be maintained, with each STRUCTURE.END matching its corresponding STRUCTURE.BEGIN command.

Note that SUB could also have been defined outside of ADDRESS.LIST. The following pair of structures behave identically to those defined above:

```
STRUCTURE.BEGIN: SUB
20 STRING->   +NAME
20 STRING->   +ADDRESS
INT->           +ID#
STRUCTURE.END
```

```
STRUCTURE.BEGIN: ADDRESS.LIST

BYTE->          +FLAG

3 SUB STRUCTS->     +NAME&ADDRESSES

STRUCTURE.END
```

If the base xaddress of a particular instance of ADDRESS.LIST is on the stack and we wish to fetch and print the ID# in the first SUB structure, we execute

```
( base.xaddr -- )   +NAME&ADDRESSES +ID# @ .
```

That is, +NAME&ADDRESSES adjusts the base address to point to the first SUB structure, and +ID# further adjusts the base address to point to the identification number field within the SUB structure.

Multiform Structures

Multiform structures (called “unions” in C and “variant records” in Pascal) are used to define structures whose members may hold different types of data at different times, or whose data may be referred to in several equivalent forms.

For example, the following structure describes the first 6 bytes of the parameter field of a heap object:

```
STRUCTURE.BEGIN: HEAP.STRUCTURE.PF

TYPE.OF:

XHNDL->         +xHANDLE      \ xaddress of the handle to the heap block

\ containing the data

OR.TYPE.OF:

PAGE->          +HNDL.PAGE  \ handle can be referred to as page...

ADDR->          +HNDL.ADDR\ ... and 16-bit address

OR.TYPE.OF:

PAGE->          +HEAP.PAGE   \ page of handle is also page of heap

TYPE.END

ADDR->          +CURRENT.HEAP       \ address of the heap used. Its page

\ is the same as the handle’s page.

STRUCTURE.END
```

The TYPE.OF: ... OR.TYPE.OF ... TYPE.END syntax is used to define the multiform structure which in this case allows us to refer to a 32-bit quantity in one of three ways, depending on the context of its use.

Example: Structure of a Matrix Parameter Field

The following structure defines the layout of a matrix parameter field:

```
STRUCTURE.BEGIN: MATRIX.PF

HEAP.STRUCTURE.PF

STRUCT->      +HEAP.STRUCTURE.PF        \ xHandle and end.heap

INT->   +ELEMENT.SIZE                  \ #Bytes per element

INT->   +DIMENSIONS                    \ #dimensions

INT->   +#COLS                         \ #columns

INT->   +#ROWS                         \ #rows

STRUCTURE.END
```

Note that the HEAP.STRUCTURE.PF appears as a sub-structure. MATRIX.PF and its analog, ARRAY.PF are kernel words that leave the size of their respective parameter fields on the stack. They are useful when creating stack frames to hold the parameter fields of temporary matrices and arrays. This is a key technique for writing re-entrant code as explained in the next chapter.