Glossary I

Glossary I: QCard/QScreen/Handheld Control-C Glossary

This Glossary provides detailed descriptions of all of the library routines that customize Control-C for use with the QCard, QScreen Controller, and QED Handheld. These C library routines are defined and declared in a set of header files in the compiler's \FABIUS\INCLUDE\MOSAIC directory. These products carry an operating system firmware version number of the form V4.4x, where x is a numeric value. This Glossary contains the following:

- ⇨ *An introduction presents an overview of the header files and explains how to interpret the information that is presented in the Main Control-C Glossary and the Interactive Debugger Glossary section.*
- ⇨ *A Categorized List of All Control-C Library Functions;*
- ⇨ *A Categorized List of QED-Forth Debugger and System Configuration Functions;*
- ⇨ *A Listing of Library Functions That Disable Interrupts;*
- ⇨ *A Comparison of Operating System Firmware V4.4x versus Prior V4.08 Firmware;*
- ⇨ *A Main Glossary of the Control-C Library Functions; and,*
- ⇨ *A Glossary of Interactive Debugger Functions.*

Introduction to the Control-C Glossary

Library Routines Are Defined and Declared in the Header Files

The names and contents of the header files are summarized in the following table. To access the five new C-callable functions in V4.40, you must #include V4_4Update.c and its companion header file V4_4Update.h as explained below.

Header File	Contents
ANALOG.H	Device drivers for the 8-bit A/D and SPI
ARRAY.H	Routines that dimension, access and manipulate Forth Arrays in paged memory

Header File	Contents
COMM.H	Serial communications driver functions
FLASH.H	Flash memory functions
HEAP.H	Heap memory manager functions
INTERUPT.H	Interrupt identifiers and functions to facilitate posting interrupt handlers
INTRFACE.H	Device drivers for the keypad and LCD display interfaces
MTASKER.H	Multitasking executive and elapsed-time clock routines
NUMBERS.H	Formatted output for QED floating point numbers, and ANSI/QED conversion
QEDREGS.H	Macro definitions for all of the 68HC11F1 registers
QEDSYS.H	Operating system functions for initialization, autostarting, and error handling
STEPPER.H	Stepper motor control primitives
TYPES.H	Useful type definitions
USER.H	Declarations of USER_AREA and TASK structures, and user variable definitions
UTILITY.H	Defines macros such as MIN(), MAX(), ABS(), TRUE and FALSE
V4_4UPDATE.H	BufferToSPI, BytesToDisplay, CalcChecksum, ClearBootVector, SetBootVector
WATCH.H	Routines that set and read the battery-backed real-time “smart watch”
XMEM.H	Fetch, store and bit manipulation functions for extended memory and EEPROM

The “Categorized List of QED Library Functions” section of this glossary is organized according to these header files. To examine these files, use TextPad (the Mosaic IDE Editor) to open the files in the compiler's **\MOSAIC\FABIUS\INCLUDE\MOSAIC** directory.

How To Interpret the Glossary Definitions

Each entry in the Main Glossary of Control-C Library Functions includes the following elements:

1. A declaration of the routine. If it is a constant or a macro that does not require any arguments, then the name of the routine is simply presented. If it is a function, or a macro that behaves like a function, then the declaration looks like an ANSI-C function prototype, such as:

   ```
   void Emit(uchar c)
   ```

   where **void** tells us that there is no return value, and **uchar c** tells us that there is a single input parameter that is an unsigned character.

2. A detailed definition of what the routine does.

3. A “Type:” field that specifies whether the routine is a **_forth** function, a C function, a macro, a constant, or a **typedef**.

4. A “Related QED-Forth name” field that specifies the name of a closely related function or constant that is accessible via the QED-Forth interpreter/compiler using QED-Forth syntax.

5. A “Header file” field that specifies where the routine is defined or declared.

Type Abbreviations Used in Function Declarations

Standard type specifiers such as char, int, long, and float are used in the glossary declarations. In addition, we use four convenient typedefs that are defined in the TYPES.H header file:

```
typedef unsigned char     uchar;
typedef unsigned int      uint;
typedef unsigned long     ulong;
typedef unsigned long     xaddr;
```

The meanings of the first three typedefs are obvious; they are abbreviations for unsigned types. The "xaddr" typedef stands for "extended address", and is used when a 32-bit address must be passed as a parameter.

Routines That Disable Interrupts

This section summarizes the functions and macros that temporarily disable interrupts. These routines are summarized to assist you in planning the time-critical aspects of your application.

Comparison with Prior QED-Forth Firmware

This section summarizes the changes to the V4.4x operating system firmware compared to the prior V4.08 firmware.

The Main Glossary of Control-C Library Functions

The Main Glossary section of this document presents definitions of all of the functions, constants and macros that are defined and declared in the \FABIUS\INCLUDE\ MOSAIC header files. This glossary does not describe standard ANSI-C library functions (such as functions in stdio.h); consult any ANSI-C text for descriptions of the ANSI standard library functions.

The routines described in the Main Glossary facilitate control of all of the hardware on the QED Board, including the analog and digital I/O lines, serial communications ports, keypad scanner, character and graphics display interfaces, and battery-backed real-time clock. In addition, the routines provide complete control over the built-in multitasking executive, heap memory manager, interrupt handling capabilities, and operating system features including Autostarting of your application program.

The Interactive Debugger Glossary

The Interactive Debugger Glossary describes a set of useful routines that allow you to interactively call C functions and manipulate variables and FORTH_ARRAY data. These versatile routines make it easy to thoroughly test each function in your program over its full range of allowed input parameters.

The environment for the interactive debugger is the QED-Forth interpreter. Thus, while the syntax of the debugger commands is similar to C function prototypes and assignment statements, you will

in fact be “talking to” the debugger using the QED-Forth language. The Debugging chapter of the QVGA Manual explains how to use QED-Forth and the debugger routines.

In addition to the debugger keywords that were created for debugging C programs, the Debugger Glossary also describes some QED-Forth functions that can be interactively called. These useful functions let you write to EEPROM, view memory contents using a DUMP command, capture your application in a Motorola S-record file to facilitate program transfer to flash memory on production boards, change the baud rates of the serial ports, and setup the MAIN program to AUTOSTART each time the QED Board starts up.

The “Introductory Notes” section at the top of the Debugger Glossary provides further information.

Categorized List of Control-C Library Functions

ANALOG.H

AD8Multiple()
AD8Off()
AD8On()
AD8Sample()
AD8ToBuffer()
DIM_AD8_BUFFER()
FastAD8Multiple()
FastAD8Sample()
InitSPI()
SPIOff()

ARRAY.H

ARRAYBASE()
ArrayBase()
ARRAYFETCH()
ArrayFetch()
ARRAYMEMBER()
ArrayMember()
ARRAYSIZE()
ARRAYSTORE()
ArrayStore()
COPYARRAY()
CopyArray()
DELETED()
Deleted()
DIM()
Dimensioned()
FARRAYFETCH()
FArrayFetch()
FARRAYSTORE()
FArrayStore()
FILLARRAY()
FillArray()
FORTH_ARRAY
NUMCOLUMNS()
NUMDIMENSIONS()
NUMROWS()
SIZEOFMEMBER()
SWAPARRAYS()
SwapArrays()

COMM.H

AskKey()
AskKey1()
AskKey2()
Baud1AtStartup()
Baud2()
Beep()
Cr()
DisableSerial2()
Emit()
Emit1()
Emit2()
ForthAskKey()
ForthEmit()
ForthKey()
InitRS485()
InitSerial2()
Key()
Key1()
Key2()
NumInputChars()
NumOutputChars()
PARITY
PARITY_IN
PARITY_OUT
PauseOnKey()
RS485Receive()
RS485Transmit()
Serial1AtStartup()
Serial2AtStartup()
TRANSMITTING
UseSerial1()
UseSerial2()
_peekTerminal()
_readChar()
_readTerminal()
_writeChar()

FLASH.H

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

HEAP.H

DEFAULT_HEAPEND
DEFAULT_HEAPSTART
DupHeapItem()
FromHeap()
INIT_DEFAULT_HEAP()
IsHeap()
Room()
ToHeap()
TransferHeapItem()

INTERUPT.H

ATTACH()
Attach()
CLOCK_MONITOR_ID
COP_ID
DISABLE_INTERRUPTS()
ENABLE_INTERRUPTS()
IC1_ID
IC2_ID
IC3_ID
IC4_OC5_ID
ILLEGAL_OPCODE_ID
IRQ_ID
OC1_ID
OC2_ID
OC3_ID
OC4_ID
PULSE_EDGE_ID
PULSE_OVERFLOW_ID
RTI_ID
SCI_ID
SPI_ID
SWI_ID
TIMER_OVERFLOW_ID
XIRQ_ID

INTRFACE.H

AskKeypad()
AskKeypress()
BufferPosition()
CharsPerDisplayLine()
CharToDisplay()
ClearDisplay()
CommandToDisplay()
DisplayBuffer()
DisplayOptions()
DISPLAY_HEAP
GARRAY_XPFA
InitDisplay()
IsDisplay()
IsDisplayAddress()
Keypad()
LinesPerDisplay()
PutCursor ()
ScanKeypad()
ScanKeypress()
StringToDisplay()
STRING_TO_DISPLAY()
UpdateDisplay()
UpdateDisplayLine()
UpdateDisplayRam()

MTASKER.H

Activate()
ACTIVATE()
AD8_RESOURCE
ASLEEP
AWAKE
BuildTask()
Halt ()
InitElapsedTime()
InstallMultitasker()
Kill ()
MAILBOX
MicrosecDelay()
SEND()
Send()
SERIAL
SERIAL1_RESOURCE
SERIAL2_RESOURCE
SPI_RESOURCE

MTASKER.H

BUILD_C_TASK()
ChangeTaskerPeriod()
FMAILBOX
FORTH_TASK
FRECEIVE()
FReceive()
FSEND()
FSend()
GET()
Get ()
Pause()
ReadElapsedSeconds()
RECEIVE()
Receive()
RELEASE()
Release()
RELEASE_AFTER_LINE
RELEASE_ALWAYS
RELEASE_NEVER
RESOURCE
StartTimeslicer()
StopTimeslicer()
TIMESLICE_COUNT
TryToFSend()
TryToGet()
TryToSend()
TRY_TO_FSEND()
TRY_TO_GET()
TRY_TO_SEND()

NUMBERS.H

FILL_FIELD
FIXED()
FLOATING()
FPtoString()
FP_CtoQ()
FP_FORMAT
FP_QtoC()
LEFT_PLACES
MANTISSA_PLACES
NO_SPACES
PrintFP()
Random()
RANDOM_SEED
RIGHT_PLACES
SCIENTIFIC()
TRAILING_ZEROS

QEDREGS.H

DDRA
DDRD
PORTA
PORTD
PORTE

See the QEDREGS.H file. All registers are named using the standard names as described in the Motorola 68HC11F1 documentation.

QEDSYS.H

Abort()
Autostart()
Breakpoint()
Cold()
ColdOnReset()
DefaultRegisterInits()
Execute()
InitVitalIRQsOnCold()
InstallRegisterInits()
NoAutostart()
NoVitalIRQInit()
PriorityAutostart()
StandardReset()
SysAbort ()
Warm()

STEPPER.H

CreateRamp()
SpeedToDuty()
StepManager()

TYPES.H

EXTENDED_ADDR
PAGE_LATCH
THIS_PAGE
TWO_INTS
_Q

USER.H

CURRENT_HEAP
CUSTOM_ABORT
CUSTOM_ERROR
NEXT_TASK
PAD
SERIAL_ACCESS
STATUS
TASK
TASKBASE
TIB
UABORT
UASK_KEY
UDEBUG
UEMIT
UERROR
UKEY
UP
UPAD
USER_AREA
UTIB

UTILITY.H

ABS()
FALSE
MAX()
MIN()
TRUE

WATCH.H

CALENDAR_TIME
ReadWatch()
SetWatch()
WATCH_DATE
WATCH_DAY
WATCH_HOUR
WATCH_HUNDREDTH_SECONDS
WATCH_MINUTE
WATCH_MONTH
watch_results
WATCH_SECONDS
WATCH_YEAR

V4_4UPDATE.H

BufferToSPI()
BytesToDisplay()
CalcChecksum()
ClearBootVector()
SetBootVector()
PPB_

XMEM.H

AddXaddrOffset()
ChangeBits()
ClearBits()
CmoveMany()
CountedString()
FetchChar()
FetchLong()
FetchLongProtected()
FillMany()
SetBits()
StoreChar()
StoreEEChar()
StoreFloat()
StoreFloatProtected()
StoreInt()
StoreLong()
StoreLongProtected()
StringMove()

XMEM.H

FetchFloat()
FetchFloatProtected()
FetchInt ()
StoreEEFloat()
StoreEEInt ()
StoreEELong()
ToggleBits()
XaddrDifference()

Categorized List of QED-Forth Debugger and System Configuration Functions

Assignment, Fetching and Function Calling

=CHAR	CHAR	INT
=FLOAT	CHAR*	INT*
=INT	DO[]	LONG
=LONG	FLOAT	LONG*
CALL.CFN	FLOAT*	

EEPROM Storage

(EE!)	(EEC!)
(EE2!)	(EEF!)

Hex Dump and Receive

DUMP	DUMP.S1	RECEIVE.HEX
DUMP.INTEL	DUMP.S2	

Numeric I/O

.	FP_CtoQ	PrintFP
D.	FP_QtoC	U.
DECIMAL	HEX	

Operating System and Memory Management

ABORT	ENABLE.DOWNLOAD	SERIAL2.AT.STARTUP
ALL.TO.FLASH	INIT.VITAL.IRQS.ON.COLD	SET.BOOT.VECTOR
AUTOSTART	INSTALL.REGISTER.INITS	SP!
BAUD1.AT.STARTUP	MAIN	STANDARD.MAP
BAUD2	NO.AUTOSTART	STANDARD.RESET
CALC.CHECKSUM	NO.VITAL.IRQ.INIT	TO.FLASH
CFA.FOR	PAGE.TO.FLASH	USE.SERIAL1
CLEAR.BOOT.VECTOR	PAGE.TO.RAM	USE.SERIAL2
COLD	PRIORITY.AUTOSTART	WARM
COLD.ON.RESET	RESTORE	WHICH.MAP
DEFAULT.REGISTER.INITS	SAVE	WORDS
DOWNLOAD.MAP	SERIAL1.AT.STARTUP	

Library Functions That Disable Interrupts

Certain Control-C library functions temporarily disable interrupts by setting the I bit in the condition code register. The glossary entries for these words detail the length of time that interrupts are disabled. These routines are summarized here to assist you in planning the time-critical aspects of your application.

The library provides a set of uninterruptable memory operators that disable interrupts for a few microseconds during the memory access. These are very useful in applications where several tasks or interrupt routines must access a shared memory location:

ChangeBits()	ClearBits()	FetchFloatProtected()
FetchLongProtected()	SetBits()	StoreFloatProtected()
StoreLongProtected()	ToggleBits()	

Accessing the LCD display require the insertion of wait states, and the computer architecture requires that interrupts be disabled while a wait state memory access is in progress. The following routines disable interrupts to insert wait states:

AskKeypad()	AskKeypress()	CharToDisplay()
ClearDisplay()	CommandToDisplay()	DisplayOptions()
InitDisplay()	IsDisplayAddress()	Keypad()
PutCursor()	ScanKeypad()	ScanKeypress()
StringToDisplay()	STRING_TO_DISPLAY()	UpdateDisplay()
UpdateDisplayLine()	UpdateDisplayRam()	

The multitasker mediates access to shared resources and ensures smooth transfer of information among tasks. The routines that manage resource variables and mailboxes must disable interrupts

for short periods of time to ensure proper access to shared resources and messages. Consequently, the following routines temporarily disable interrupts:

```
FRECEIVE()          FReceive()          FSEND()
FSend()             GET()               Get()
RECEIVE()           Receive()           SEND()
Send()              TryToFSend()        TryToGet()
TryToSend()         TRY_TO_FSEND()      TRY_TO_GET()
TRY_TO_SEND()
```

Some of these routines also call Pause() to give other tasks a chance to run while waiting for a resource or message; as explained below, Pause() also disables interrupts. Consult their glossary entries for details.

The following routines temporarily disable interrupts to ensure that a new task is not corrupted while it is being built:

```
BuildTask()       BUILD_C_TASK()
```

This function disables interrupts to ensure that the elapsed time clock is not updated while it is being read:

```
ReadElapsedSeconds()
```

The multitasker is charged with smoothly transferring control among tasks via timeslicing or cooperative task switching. The timeslicer is an interrupt service routine associated with output compare#2. The timeslicer's interrupt service routine disables interrupts for the duration of a task switch which requires 25 microseconds, plus 3.25 microseconds for each ASLEEP task encountered in the task list. The cooperative task switch routine

```
Pause()
```

switches tasks in (27 + 3.25n) microseconds, where n is the number of ASLEEP tasks encountered in the round robin task list. Of this time, interrupts are disabled for (20 + 3.25n) microseconds.

The Pause() routine (which temporarily disables interrupts) is called by the following built-in device drivers:

```
ForthEmit()         ForthKey()          Emit()
Emit1()             Emit2()             Key()
Key1()              Key2()              Keypad()
ScanKeypad()
```

The following device driver routines GET() and RELEASE() resource variables, and so disable interrupts for short periods of time:

```
AD8Multiple()       AD8Sample()         AD8ToBuffer()
AskKey()            AskKey1()           AskKey2()
Emit()              Emit1()             Emit2()
ForthAskKey()       ForthEmit()         ForthKey()
Key()               Key1()              Key2()
PauseOnKey()        ReadWatch()         SetWatch()
```

All of the routines that write to the EEPROM disable interrupts for 20 msec per programmed byte. This results from the 68HC11's design which prohibits any EEPROM locations from being read while other EEPROM locations are being modified. Since all interrupts are vectored through EEPROM, interrupts cannot be serviced while an EEPROM storage operation is in progress. The following fundamental EEPROM storage functions:

```
StoreEEChar()       StoreEEFloat()      StoreEEInt()
StoreEELong()
```

disable interrupts for 20 msec per programmed byte. These routines are smart enough to avoid programming a byte that already has the correct contents. The following routines may modify EEPROM locations:

```
ATTACH()            Attach()            Autostart()
ColdOnReset()       DefaultRegisterInits()DownloadMap()
InitVitalIRQsOnCold() InstallMultitasker()  InstallRegisterInits()
IsDisplay()         NoAutostart()       Serial1AtStartup()
Serial2AtStartup()  StandardMap()       StandardReset()
StartTimeslicer()
```

All of the routines that write to the Flash memory disable interrupts for 20 msec per programmed sector, where a standard sector is 256 bytes. This results from the flash architecture which prohibits any flash locations from being read while other flash locations are being modified. Since interrupts invoke flash-resident code, interrupts cannot be serviced while an flash storage operation is in progress. The following flash routines disable interrupts:

```
PageToFlash()       PageToRam()         ToFlash()
```

The following routines disable interrupts and do not re-enable them:

```
Cold()              DISABLE_INTERRUPTS()  Warm()
```

DISABLE_INTERRUPTS() and its assembly language counterpart SEI explicitly set the I bit in the condition code register. The routines ENABLE_INTERRUPTS and the assembler mnemonic CLI clear the I bit to globally enable interrupts. The restart routines Cold() and Warm() disable interrupts to ensure that the initialization process is not interrupted.

Comparison with Prior QED-Forth Firmware

The QCard, QScreen Controller, and QED Handheld operating system firmware carries the version number V4.4x, where x represents a numeric value. Customers who are familiar with the prior V4.08 operating system software will notice that certain device driver functions have been removed from V4.4x because the corresponding hardware is not present. These functions comprise the hardware drivers for the 12-bit A/D, D/A, PIA, and high current drivers. We recommend the use of the many available WildCard I/O modules and their associated pre-coded kernel extension device drivers to add customized I/O to meet the needs of your application.

The V4.4x keypad scanner and character/graphics display drivers are available to support an optional Keypad/Display WildCard. These kernel-resident drivers work only if the Keypad/Display WildCard is assigned a WildCard module address of 0. On the QCard, this is accomplished by installing the Keypad/Display WildCard on module header 0, and leaving the jumper caps off the

WildCard's two on-board jumpers. The QScreen and Handheld products come complete with a graphical user interface that is controlled by pre-coded drivers.

The **ReadWatch()** and **SetWatch()** functions are backwardly compatible with prior versions, but they rely on different real-time clock hardware as described in their glossary entries.

The V4.4x kernel boots up at 19200 baud, compared to 9600 baud on many earlier products based on the V4.08 kernel.

How To Access the Additional Functions

The following QED-Forth functions have been added to the Forth debugging glossary:

```
ALL.TO.FLASH          CALC.CHECKSUM
CLEAR.BOOT.VECTOR     ENABLE.DOWNLOAD
SET.BOOT.VECTOR
```

The following five C-callable functions have been added:

```
BufferToSPI()          BytesToDisplay()
CalcChecksum()         ClearBootVector()
SetBootVector()
```

No additional software is needed to access the new functions using interactive Forth commands, as they are built into the kernel.

C programmers must include the files named v4_4update.c and v4_4update.h to gain access to the five new C-accessible functions. These files are located in the

\Mosaic\Fabius\Include\Mosaic\v4_4Update

directory in the software distribution CD. Simply #include both the v4_4update.h and v4_4update.c files in one of your source files, and also #include v4_4update.h in any other source files that use these new kernel routines.

Summary of Modified Functions

A set of operating system functions makes it easy to manage the downloading of code into RAM and the transfer of the compiled code to flash. This is performed automatically by the Mosaic C IDE (Integrated Development Environment), so C programmers need not be aware of the details. For those who are interested, the Forth debugging glossary entries for the functions **STANDARD.MAP**, **DOWNLOAD.MAP**, **TO.FLASH**, **PAGE.TO.FLASH** and **PAGE.TO.RAM** describe how the new expanded amounts of flash memory are managed. The QCard has flash at pages 4-7 that swaps with RAM on parallel pages 1-3, plus flash at hex pages 10-17 that swaps with RAM on parallel hex pages 18-1F. In the "download map", flash and RAM are swapped: flash is present on pages 1-3 and 18-1F, and RAM is present on pages 4-6 and 10-17. On the QScreen Controller and Handheld, **TO.FLASH** can program the optional flash memory at pages 0x20-2F. The great majority of applications compile to less than 96 KBytes of code, which fits on pages 4, 5 and 6. For these applications, the new functions **ENABLE.DOWNLOAD** and **ALL.TO.FLASH** greatly simplify the downloading process. Simply insert the command **ENABLE.DOWNLOAD** at the top of the first file to be downloaded to the QCard. This

function makes sure that any previously downloaded code is transferred to RAM, and calls **DOWNLOAD.MAP** to ensure a RAM-based memory map that enables compilation of code. At the end of the last file to be downloaded, insert the command **ALL.TO.FLASH**. This copies the compiled code in pages 4, 5 and 6 to flash, sets the **STANDARD.MAP**, and calls **SAVE** so that the **RESTORE** command can be used to recover after a crash or a COLD restart during the development process.

To support a "bullet-proof" kernel extension to enable firmware upgrades, an optional "boot vector" is implemented using the **SET.BOOT.VECTOR** operating system function. This allows the posting of a function that is executed before the autostart program is run; see the Forth debugging glossary entry of **SET.BOOT.VECTOR** for details. The boot vector and its code are located on page 0x0C, a page that can be hardware write protected with the "page-C write protect" jumper. Removal of the boot vector is accomplished by invoking CLEAR.BOOT.VECTOR or by using the special cleanup mode described in the hardware documentation.

The **AskKeypad()**, **AskKeypress()** and **Keypad()** functions have been upgraded to support more than 20 keys. They work identically to prior versions for keys 0 through 19, but add support for an additional 20 keys with numbers 20 through 39.

InitSPI() now configures the SPI to sample data on the falling trailing edge of the SPI clock. This is consistent with the data transfer protocol of the on-board battery-backed real-time clock. The user may freely change the SPI configuration to meet the needs of the application program.

When using the Forth debugger tools, numbers can now have a leading 0x or 0X to indicate that the numbers are hexadecimal.

Main Glossary of Control-C Library Functions

void Abort(void)

If the CUSTOM_ABORT flag is true (non-zero), executes the abort routine whose xcfa (32-bit extended code field address) is stored in the user variable UABORT, and then returns to the routine that called ABORT. If CUSTOM_ABORT is false (zero), executes the default routine SysAbort() which clears the data and return stacks, and sets the page to the default page (0). If an autostart vector has been installed [see Autostart() and PriorityAutostart()], SysAbort() executes the specified routine; otherwise it executes QUIT which sets the execution mode and enters the QED-Forth monitor. If the stack pointers do not point to common RAM, a COLD restart is initiated.
Type: _forth function; QED-Forth name: ABORT
Header file: qedsys.h

ABS(num)

Returns the absolute value of num; the input can be of any type. This macro is defined as:

```
#define ABS(A)  (((A) >=0) ? (A) : (-A))
```

Type: macro
Header file: utility.h

void Activate(void(*action)(), uint actionPage, xaddr taskBase)

Sets up the specified routine on the specified actionPage as the action function of the task whose TASKBASE address is taskBase, and leaves the specified task AWAKE so that it will be entered on the next pass through the round robin task list. For application programs that are compiled on a single page, the macro form ACTIVATE() is highly recommended; see its glossary entry for a complete description of what this routine does. For programs whose code occupies multiple pages of memory on the QED Board, this functional form of Activate() allows the page of the action routine (actionPage) to be specified.
Type: _forth function; QED-Forth name: ACTIVATE
Header file: mtasker.h

void ACTIVATE(void(*action)(), int* task_base_addr)

Sets up the specified routine as the action function of the task whose TASKBASE address equals task_base_addr, and leaves the specified task AWAKE so that it will be entered on the next pass through the round robin task list. The ACTIVATE() macro may be used in all applications whose code compiles onto a single page of the QED Board; for multi-page applications, see the functional form named Activate() which allows you to specify the page of the action routine.
ACTIVATE() assumes that the specified task has already been declared using a TASK statement and added to the task list by BUILD_C_TASK(). The task's action function is typically either an infinite loop or a finite routine that ends with a Halt() instruction (which is itself an infinite loop). ACTIVATE() buries a call to Halt() in the return stack frame to ensure graceful termination of a finite activation routine. If cooperative multitasking is used exclusively (that is, if the timeslicer is not used), then the loop of

the action function must contain at least one Pause() statement, or invoke a function that in turn executes Pause(). Otherwise, no task switching occurs. If timeslicing is used, incorporation of Pause() statements in the loop of the action function is optional. The typical form of an action function is:

```
void action_name(void)
        {       while(1)
                {       statements to be executed infinitely;
                        Pause();
                        statements to be executed infinitely;
                }
        }
```

or:

```
void action_name(void)
        {       statements to be executed;
                Pause();
                statements to be executed;
                Halt();
        }
```

Once the action routine has been defined, the task can be named, built and activated as follows:

```
TASK  ReadInputTask;            // name and allocate the task area
BUILD_C_TASK(0,0,&ReadInputTask);       // build the task in RAM
ACTIVATE(action_name, &ReadInputTask);  // activate the task
```

Note that action_name is passed without parentheses to the ACTIVATE routine; this tells the compiler that action_name is a pointer to a function. The turnkey application program in the documentation provides working examples of how to define, build and activate tasks in a multitasking program. See TASK and BUILD_C_TASK().
Type: macro; Related QED-Forth function: Activate()
Header file: mtasker.h

void AD8Multiple(xaddr buffer, uint Interval, uint NumSamples, uint channel)

Acquires NumSamples samples from the 8 bit analog to digital (A/D) converter in the 68HC11 and stores the samples as sequential unsigned 8 bit values starting at the specified buffer xaddress. [For an automated approach to storing samples in an array in paged memory, see the glossary entries for DIM_AD8_BUFFER() and AD8ToBuffer()].

The channel parameter specifies the channel number of the A/D (0 <= channel <= 7). To ensure proper operation in a multitasking environment, this routine executes GET(AD8_RESOURCE) before reading the A/D and RELEASE(AD8_RESOURCE) before terminating. If the specified xaddr is in common memory, the first sample is taken after 86 µsec and subsequent samples are taken every (10+2.5*u1) µsec, where u1 is the specified timing parameter passed to this routine. If the specified xaddr is in paged memory, the first sample is taken after 81 µsec and subsequent samples are taken every (32.5+2.5*u1) µsec. Of course, the operation of interrupts (including timesliced multitasking) will affect these sampling times. For a faster version suitable for non-multitasking applications, see FastAD8Multiple(). See also AD8Sample(), FastAD8Sample(), and A/D8.ON.
Type: _forth function; QED-Forth name: A/D8.MULTIPLE
Header file: analog.h

void AD8Off(void)

Turns off the 68HC11's on-chip 8 bit analog to digital (A/D) convertor by clearing the ADPU bit in the processor's OPTION register. The 8 bit A/D is initialized to the off state upon every reset or restart. See AD8On().
Type: _forth function; QED-Forth name: A/D8.OFF
Header file: analog.h

void AD8On(void)

Turns on the 68HC11's on-chip 8 bit analog to digital (A/D) convertor by setting the ADPU bit in the processor's OPTION register, and waits 100 microseconds for the A/D to stabilize. Also initializes AD8_RESOURCE to zero. This routine must be executed after a reset or restart before using the 8 bit A/D. See AD8Off().
Type: _forth function; QED-Forth name: A/D8.ON
Header file: analog.h

uchar AD8Sample(uint channel)

Returns a single sample byte from the specified channel (0 <= channel <= 7) of the 8 bit analog to digital (A/D) converter in the 68HC11. To ensure proper operation in a multitasking environment, this routine executes GET(AD8_RESOURCE) before reading the A/D and RELEASE(AD8_RESOURCE) before terminating. This routine executes in 93 microseconds. For a faster version suitable for non-multitasking applications, see FastAD8Sample(). See also FastAD8Multiple(), AD8Multiple(), and AD8On().
Type: _forth function; QED-Forth name: A/D8.SAMPLE
Header file: analog.h

void AD8ToBuffer(FORTH_ARRAY* array_ptr, uint Interval, uint NumSamples, uint channel)

Acquires NumSamples samples from the 8 bit analog-to-digital (A/D) converter and stores the samples as sequential 8 bit values in the specified Forth array in paged memory. The acquired readings may then be accessed using the ARRAYFETCH() macro. AD8ToBuffer() assumes that DIM_AD8_BUFFER() has already been executed to set up the Forth array buffer. The Interval parameter specifies the timing of the samples (see the glossary entry for AD8Multiple() for detailed timing specifications), and channel specifies the channel number of the A/D (0 <= channel <= 7).
Example of use:

```
// **** Initializations (performed only once at system startup): ****
#define MAXSAMPLES  100
INIT_DEFAULT_HEAP();          // ensure valid heap to hold array
AD8On();
FORTH_ARRAY acquired_data;
DIM_AD8_BUFFER(&acquired_data, MAXSAMPLES); // 1 column
// **** Now perform the conversions ****
AD8ToBuffer(&acquired_data, 0, MAXSAMPLES, 5);
// ***************
```

This example acquires 100 samples at the maximum sampling rate (because the timing interval equals 0) and stores them into the acquired_data array. Note that the AD8ToBuffer() routine can be called repeatedly to update the contents of the array. To fetch the sample whose index is 5 (that is, the 6th sample taken), you can execute:

```
int the_sample;
the_sample = ARRAYFETCH(uchar, 5, 0, &acquired_data);
```

Note that the data index (5) appears as the row index, and 0 appears as the column index in the parameter list for ARRAYFETCH().

Type: C function
Header file: analog.h

AD8_RESOURCE

A constant that returns the address of the resource variable associated with the 8 bit analog to digital (AD8) convertor. Should be accessed by the routines GET(), TRY_TO_GET() and RELEASE(). Initialized to zero by AD8On() and at each reset or restart. AD8_RESOURCE is automatically invoked by many of the AD8 device driver functions and macros. See RESOURCE.

Type: macro constant; Related QED-Forth function: A/D8.RESOURCE
Header file: mtasker.h

xaddr AddXaddrOffset(xaddr baseAddr, long offset)

Adds the specified signed offset to baseAddr and returns the resulting address. Note that in the QED Board's paged memory, the address immediately following 0x7FFF is address 0000 on the following page.

Type: _forth function; QED-Forth name: XD+
Header file: xmem.h

xaddr ARRAYBASE(FORTH_ARRAY* array_ptr)

Returns the base address (that is, the address of the first element) of the specified array. Returns zero if the array is undimensioned. No error checking is performed.

Example of use:

To define an array of unsigned longs named MyArray with 3 rows and 5 columns, execute:

```
FORTH_ARRAY Myarray;
DIM(ulong, 3, 5, &Myarray);
```

Now to assign the base address of the array to a variable, execute:

```
static xaddr Myarray_base_address;
Myarray_base_address = ARRAYBASE(&Myarray);
```

Note that the & (address-of) operator in front of the array's name tells the compiler that a pointer is being passed. If you forget the & operator, the compiler will warn you that you are attempting to pass an entire structure (the array's parameter field structure) as an argument to a function.

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also FORTH_ARRAY, DIM(), ARRAYSTORE() and ARRAYFETCH().

Type: macro; Related QED-Forth function: [0]
Header file: array.h

xaddr ArrayBase(FORTH_ARRAY* array_ptr, uint pfa_page)

A subsidiary Forth function called by the recommended macro ARRAYBASE(); see ARRAYBASE().

Type: _forth function; QED-Forth name: [0]
Header file: array.h

ulong ARRAYFETCH(type, uint row, uint col, FORTH_ARRAY* array_ptr)

Fetches the contents of the element at row, col in the specified 2-dimensional array. The size of the data that is fetched from the array may be 1 byte, 2 bytes, or 4 bytes depending upon the number of bytes per element of the array as specified by the DIM() command. The "type" parameter passed to the function causes the specified type-cast to be performed on the fetched data; this is necessary to inform the compiler of the type of the data stored in the array. Examples of valid "type" parameters are standard identifiers and pre-defined types such as:

int	unsigned int	uint
char	unsigned char	uchar
long	unsigned long	ulong

Typically, the specified type corresponds to the type of the data stored in the array. Note that the uint, uchar, and ulong types are defined in the types.h file. For floating point data, use FARRAYFETCH(). There is an unchecked error if the specified array does not have 2 dimensions or if the number of bytes per element does not equal 1, 2, or 4. If UDEBUG is true (its default state after a COLD startup) and if the indices are out of range, Abort() is called.

Example of use:

To define an array of unsigned longs named MyArray with 3 rows and 5 columns, execute:

```
FORTH_ARRAY  Myarray;
DIM(ulong, 3, 5, &Myarray);
```

Now to fetch the contents of the item at row=1, column=2 into a variable, execute:

```
static ulong  the_contents;
the_contents = ARRAYFETCH(1,2,&Myarray);
```

Note that the & (address-of) operator in front of the array's name tells the compiler that a pointer is being passed. If you forget the & operator, the compiler will warn you that you are attempting to pass an entire structure (the array's parameter field structure) as an argument to a function.

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also FORTH_ARRAY, DIM(), FARRAYFETCH() and ARRAYSTORE().

Type: macro; Related QED-Forth function: ArrayFetch()

Header file: array.h

ulong ArrayFetch(uint row, uint col, FORTH_ARRAY* array_ptr, uint pfa_page)

A subsidiary function called by the recommended ARRAYFETCH() macro; see ARRAYFETCH().

Type: _forth function; QED-Forth name: 2ARRAY.FETCH

Header file: array.h

xaddr ARRAYMEMBER(uint row, uint col, FORTH_ARRAY* array_ptr)

Returns the 32-bit extended address of the element at the specified row and column in the 2-dimensional array specified by array_ptr. If UDEBUG is true (its default state after a COLD startup) and if the indices are out of range, calls Abort(). Example of use:

```
FORTH_ARRAY  Myarray;       // define an array named Myarray
DIM(ulong, 3, 5, &Myarray); // dimension it as 3 rows by 5 columns
```

To calculate the extended address of the element at row=0, column = 1, execute:

```
static xaddr  element_address;
element_address = ARRAYMEMBER( 0, 1, &Myarray);
```

element_address could now be passed as a parameter to FetchLong() or StoreLong() to access the array member. Note that ARRAYFETCH() and ARRAYSTORE() provide a more direct means for fetching from and storing to an array element. Caution: recall that Forth function calls cannot be nested, so it is not legal to use ARRAYMEMBER() as an input parameter for another function such as StoreLong(). Rather, the return value of ARRAYMEMBER() must be saved in a variable which in turn is passed as a parameter to another Forth function.

Type: macro; Related QED-Forth function: M[]

Header file: array.h

xaddr ArrayMember(uint row, uint col, FORTH_ARRAY* array_ptr, uint pfa_page)

A subsidiary function called by the recommended macro ARRAYMEMBER(); see ARRAYMEMBER().

Type: _forth function; QED-Forth name: M[]

Header file: array.h

uint ARRAYSIZE(FORTH_ARRAY* array_ptr)

A macro that returns the number of elements d (not the number of bytes!) in the Forth array designated by array_ptr. An unpredictable result is returned if the array is not dimensioned.

Example of use:

```
FORTH_ARRAY  Myarray;       // define an array named Myarray
DIM(ulong, 3, 5, &Myarray); // dimension it as 3 rows by 5 columns
static uint  size_of_the_array;
size_of_the_array = ARRAYSIZE(&Myarray);
```

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM().

Type: macro; Related QED-Forth name: ?ARRAY.SIZE

Header file: array.h

void ARRAYSTORE(ulong value, uint row, uint col, FORTH_ARRAY* array_ptr)

Stores the specified value at row, col in the 2-dimensional FORTH_ARRAY specified by array_ptr. The size of the data that is stored to the array may be 1 byte, 2 bytes, or 4 bytes depending upon the number of bytes per element of the array as specified by the DIM() command. Valid data types for value include signed and unsigned char, int and long. To store a floating point value, use FARRAYSTORE(). There is an unchecked error if the specified array does not have 2 dimensions or if the number of bytes per element does not equal 1, 2, or 4.

Example of use:

To define an array of unsigned longs named MyArray with 3 rows and 5 columns, execute:

```
FORTH_ARRAY  Myarray;
DIM(ulong, 3, 5, &Myarray);
```

Now to store 0x12345 into the element at row=1, column=2 into a variable, execute:

```
ARRAYSTORE(0x12345, 1,2,&Myarray);
```

Note that the & (address-of) operator in front of the array's name tells the compiler that a pointer is being passed. If you forget the & operator, the compiler will warn you that you are attempting to pass an entire structure (the array's parameter field structure) as an argument to a function.
See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also FORTH_ARRAY, DIM(), FARRAYSTORE() and ARRAYFETCH().
Type: macro; Related QED-Forth function: ArrayStore()
Header file: array.h

void ArrayStore(ulong value, uint row, uint col, FORTH_ARRAY* array_ptr, uint pfa_page)
A subsidiary Forth function called by the recommended macro ARRAYSTORE(); see ARRAYSTORE().
Type: _forth function; QED-Forth name: 2ARRAY.STORE
Header file: array.h

int AskKey(void)
Returns a flag indicating receipt of a character. If flag equals -1, a character has been received; if the flag equals 0 (false), no character has been received. Executes GET(SERIAL) and, depending on the value in SERIAL_ACCESS, may execute RELEASE(SERIAL). AskKey() is a vectored routine that executes the function whose xcfa (32-bit code field address) is stored in the user variable UASK_KEY. Thus the programmer may install a different routine to tailor the behavior of AskKey() to the application's needs. For example, AskKey() could access a serial port other than that on the 68HC11 chip, or different tasks could use different AskKey() routines. See Key(), AskKey1() and AskKey2(), _peekTerminal(), and SERIAL_ACCESS.
Type: C function; related QED-Forth function name: ?KEY
Header file: comm.h

int AskKey1(void)
Returns a flag indicating whether a character has been received on the primary serial port (serial1) associated with the 68HC11's on-chip hardware UART. If a character has been received a flag equal to -1 is returned; otherwise a false flag (=0) is returned. AskKey1() is the default AskKey() routine installed in the UASK_KEY user variable after the special cleanup mode is invoked, or if Seral1AtStartup has been executed. If the value in SERIAL_ACCESS is RELEASE_AFTER_LINE, AskKey1() does not GET(SERIAL1_RESOURCE) or RELEASE(SERIAL1_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, AskKey1() executes GET(SERIAL1_RESOURCE) and RELEASE(SERIAL1_RESOURCE). If SERIAL_ACCESS contains RELEASE_NEVER, AskKey1() GETs but does not RELEASE() the SERIAL1_RESOURCE. See SERIAL_ACCESS, AskKey(), UASK_KEY, AskKey2().
Type: _forth function; QED-Forth name: ?KEY1
Header file: comm.h

int AskKey2(void)
Returns a flag indicating whether a character has been received on the on the secondary serial port (serial2). The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). If a character has been

received a flag equal to -1 is returned; otherwise a false flag (=0) is returned. AskKey2() can be made the default AskKey() routine installed in the UASK_KEY user variable after each reset or restart by executing Serial2AtStartup(). If the value in SERIAL_ACCESS is RELEASE_AFTER_LINE, AskKey2() does not GET(SERIAL2_RESOURCE) or RELEASE(SERIAL2_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, AskKey2() executes GET(SERIAL2_RESOURCE) and RELEASE(SERIAL2_RESOURCE). If SERIAL_ACCESS contains RELEASE_NEVER, AskKey2() GETs but does not RELEASE() the SERIAL2_RESOURCE. See SERIAL_ACCESS, AskKey(), UASK_KEY, AskKey1().
Type: _forth function; QED-Forth name: ?KEY2
Header file: comm.h

ulong AskKeypad(void)
A subsidiary Forth function that is called by ScanKeypad(); see ScanKeypad().
Type: _forth function; QED-Forth name: ?KEYPAD
Header file: intrface.h

ulong AskKeypress(void)
A subsidiary Forth function that is called by ScanKeypress(); see ScanKeypress().
Type: _forth function; QED-Forth name: ?KEYPRESS
Header file: intrface.h

ASLEEP
A constant that returns the value 1. When stored into a task's STATUS user variable, indicates to the multitasking executive that the task is asleep and cannot be entered.
The following example illustrates how to put another task to sleep. Assume that we have pre-defined an infinite loop task function named GatherData(), and that we have named, built and activated a task with the following statements:

```
TASK ReadInputTask;          // name and allocate the task area
BUILD_C_TASK(0,0,&ReadInputTask);      // build the task in RAM
ACTIVATE(GatherData, &ReadInputTask);  // activate the task
```

Now the task is AWAKE and running, and the TASK command has defined &ReadInputTask as a pointer to the task's TASK structure, of which the USER_AREA structure is the first element. Note that the task's STATUS address (whose contents control whether the task is AWAKE) is the first element in the task's USER_AREA structure; its element name is user_status. Thus, to put the task ASLEEP we simply execute

```
ReadInputTask.USER_AREA.user_status = ASLEEP ;
```

If a given task wants to put itself asleep, it can simply execute the commands:

```
STATUS = ASLEEP;
Pause();
```

The Pause() command ensures that the multitasking executive immediately exits the task after it is put ASLEEP. See also AWAKE.
Type: constant; Related QED-Forth function: ASLEEP
Header file: mtasker.h

void ATTACH (void(*action)(), int interrupt_id)

Posts an interrupt handler routine specified by the function pointer (*action)() for the interrupt with identity number interrupt_id (for example, OC1_ID, SWI_ID, etc). This macro form of the Attach() function is recommended for applications whose object code fits on a single page. For example, suppose you have defined a standard C function named OC1_Service() to service the Output Compare #1 interrupt. To install the interrupt handler, simply execute

```
ATTACH( OC1_Service, OC1_ID);
```

Note that the name of the service routine is passed as a parameter without parentheses; this tells the C compiler to pass a pointer to the OC1_Service() function. ATTACH() compiles an 8-byte code sequence at the EEPROM location associated with the specified interrupt. When the interrupt is serviced, the code specified by the function pointer will be executed. The service routine should NOT be declared as an _interrupt function; the ATTACH() routine compiles the needed RTI (return from interrupt) instruction.

Type: macro; Related QED-Forth function: Attach()
Header file: interupt.h

void Attach(void(*action)(), uint actionPage, int interrupt_id)

Posts an interrupt handler routine specified by the function pointer (*action)() whose code is compiled on actionPage for the interrupt with identity number interrupt_id (for example, OC1_ID, OC3_ID, etc). This functional form (as opposed to the ATTACH() macro) is recommended for applications whose object code resides on more than one memory page on the QED Board. For example, suppose you have defined a standard C function named OC1_Service() on QED memory page 5 to service the Output Compare #1 interrupt. (To find out on which page the service routine resides, search for the service routine's name in the .OUT file created by the compiler, and look at the top 2 hexadecimal digits of the hexadecimal function address.) To install the interrupt handler, simply execute

```
Attach( OC1_Service, 5, OC1_ID);
```

Note that the name of the service routine is passed as a parameter without parentheses; this tells the C compiler to pass a pointer to the OC1_Service() function. Attach() compiles an 8-byte code sequence at the EEPROM location associated with the specified interrupt. When the interrupt is serviced, the code specified by the function pointer will be executed. The service routine NOT be declared as an _interrupt function; the Attach() routine compiles the needed RTI (return from interrupt) instruction.

Type: _forth function; QED-Forth name: ATTACH
Header file: interupt.h

void Autostart(void(*action)(), uint actionPage)

Compiles a 6-byte sequence into the EEPROM in the 68HC11. On subsequent restarts and ABORTs, the routine having the specified xcfa will be executed. This allows a finished application to be automatically entered upon power up and resets. CAUTION: If your application is to be put into production and replicated, it is recommended that you use the PriorityAutostart() function which stores the 6-byte autostart sequence in flash memory.

Usage: We recommend that Autostart() and PriorityAutostart() be executed interactively from the QED-Forth monitor. The easiest way to do this is to use Forth

syntax instead of C syntax. After your application program is completed and debugged, simply type from your terminal the command:

```
CFA.FOR MAIN AUTOSTART
```

This writes a pattern into EEPROM that causes MAIN to be executed upon all subsequent resets and restarts.

Implementation detail: At location hex AE00 in EEPROM, AUTOSTART writes the pattern 1357 followed by the four byte xcfa. To undo the effects of this command and return to the default startup action, type the QED-Forth command

```
NO.AUTOSTART
```

from your terminal. To recover from the installation of a buggy autostart routine, use the special cleanup mode as described in the "Programming the QED Board in C" chapter in the "Getting Started" Manual. See PriorityAutostart().

Type: _forth function; QED-Forth name: AUTOSTART
Header file: qedsys.h

AWAKE

A constant that returns the value 0. When stored into a task's STATUS user variable, indicates to the multitasking executive that the task is awake and may be entered.

The following example illustrates how to wake up another task that was earlier put to sleep. Assume that we have pre-defined an infinite loop task function named GatherData(), and that we have named, built and activated a task with the following statements:

```
TASK ReadInputTask;             // name and allocate the task area
BUILD_C_TASK(0, 0, &ReadInputTask);      // build the task in RAM
ACTIVATE(GatherData, &ReadInputTask);    // activate the task
```

If the task has been put ASLEEP by a command such as

```
ReadInputTask.USER_AREA.user_status = ASLEEP ;
```

we can now wake it up with the following command:

```
ReadInputTask.USER_AREA.user_status = AWAKE ;
```

The TASK command has defined &ReadInputTask as a pointer to the task's TASK structure of which the USER_AREA structure is the first element. Note that the task's STATUS address (whose contents control whether the task is AWAKE) is the first element in the task's USER_AREA structure; its element name is user_status. See also ASLEEP.

Type: constant; Related QED-Forth function: AWAKE
Header file: mtasker.h

void Baud1AtStartup(int baud)

Configures the QED Board so that the baud rate of the primary serial port (serial1) supported by the 68HC11's hardware UART will equal the specified standard baud rate upon all subsequent resets and restarts. Standard baud rates for are 150, 300, 600, 1200, 2400, 4800, 9600, and 19200 baud. This function can also be called interactively from the terminal using QED-Forth syntax; simply type the baud rate followed by a space followed by BAUD1.AT.STARTUP. For example, to change the baud rate to 19,200 baud, type from your terminal:

```
DECIMAL 19200 BAUD1.AT.STARTUP
```

Implementation detail: This routine calls InstallRegisterInits() which writes into EEPROM the required contents of INIT (=B8H), the contents of BAUD that corresponds to the specified baud rate, and the contents of OPTION, TMSK2, and BPROT that are

present when this routine is executed. These values are installed in their respective registers upon each subsequent reset and restart. To undo the effects of this command, type from your terminal the command

DEFAULT.REGISTER.INITS

or invoke the special cleanup mode as described in the "Programming the QED Board in C" chapter in the "Getting Started" Manual.

Type: _forth function; QED-Forth name: BAUD1.AT.STARTUP
Header file: comm.h

void Baud2(int baud)

Sets the baud rate of the secondary serial port (serial2) supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). Smooth file transfers can be achieved at up to 4800 baud. The baud rate of serial2 is initialized to 1200 baud by the COLD restart routine. See UseSerial2() and Serial2AtStartup().

Type: _forth function; QED-Forth name: BAUD2
Header file: comm.h

void Beep(void)

Emits the bell character, ascii 07. Setting the user variable QUIET (see user_quiet in the user.h file) to a true (non-zero) value silences the beep.

Type: _forth function; QED-Forth name: BEEP
Header file: comm.h

void Breakpoint(void)

The Breakpoint() function may be edited into any function that is being debugged to set a software breakpoint. It saves the machine state and invokes a FORTH-style interactive text interpreter that can be distinguished from the standard interpreter by the BREAK> prompt displayed at the start of each line. Any valid commands may be executed from within the Breakpoint interpreter. From within the Breakpoint interpreter, typing a carriage return alone on a line exits the Breakpoint mode, restores the machine registers to the values they held just before Breakpoint() was entered, and resumes execution of the program that was running when Breakpoint() was entered. The Breakpoint() routine's preservation of the register state and its ability to execute any valid command make it a very powerful debugging tool. Breakpoint() may be compiled into any definition to stop program flow in order to debug or analyze a function at the point where Breakpoint() was called. Once inside Breakpoint(), variables and memory locations may be displayed or altered using QED-Forth debugging routines such as DUMP, INT, =INT, etc.; see the debugging glossary section of this document. To fully exit from the routine that called Breakpoint(), type ABORT (or any illegal command) from the terminal in response to the BREAK> prompt. Any error encountered while in the BREAK routine executes ABORT which places the programmer back into the standard QED-Forth interpreter (unless ABORT has been revectored to perform some other action; see CUSTOM_ABORT).

Type: _forth function; QED-Forth name: BREAK
Header file: qedsys.h

uint BufferPosition(int line, int column)

Given the specified LCD display line number n1 [0 <= n1 < LinesPerDisplay()] and the specified character position in the display line [0 <= n2 < CharsPerDisplayLine()],

returns the offset of the specified position relative to the base address returned by DisplayBuffer(). Clamps the returned offset to ensure that it is not greater than the size of the buffer. Note that for a graphics-style display the input parameter "line" is interpreted differently depending on whether the display is being used in "text mode" or "graphics mode". In text mode, "line" corresponds to the character line#; in graphics mode, "line" corresponds to the pixel line#. See LinesPerDisplay() for further information.
Type: _forth function; QED-Forth name: BUFFER.POSITION
Header file: intrface.h

void BufferToSPI(uint base_addr, uint base_page, uint numbytes, uint readback)
Sends numbytes of data starting at base_addr on base_page to the SPI (Serial Peripheral Interface). The buffer must not cross a page boundary, and 0 <= numbytes < 32,768. This routine does not GET or RELEASE the SPI.RESOURCE. This is an outgoing transfer only; no incoming data is stored. This routine is optimized for speed, and executes at about 9 microseconds per byte.
Type: _forth function; QED-Forth name: BUFFER>SPI
Header file: V4_4Update.h; V4_4Update.c must be #included in 1 file only

void BuildTask(xaddr heapStart,xaddr heapEnd,xaddr vp,xaddr dp,xaddr np,xaddr tib,xaddr pad,
xaddr pocket,xaddr r0,xaddr s0,xaddr taskBase, int n)
Subsidiary function called by the recommended macro BUILD_C_TASK; see BUILD_C_TASK.
Type: _forth function; QED-Forth name: BUILD.TASK
Header file: mtasker.h

void BUILD_C_TASK(xaddr heapStart, xaddr heapEnd, TASK * taskbase)
Builds a task with a specified heap, locating its task area in a 1 kilobyte block starting at the specified taskbase address in common RAM. The TASK declaration is used to name and allocate the task area. BUILD_C_TASK() assigns the task's stacks, user area, and PAD, POCKET, and TIB buffers to a 1Kbyte block of common RAM starting at the base of the TASK structure. The task is appended to the round-robin task list and left ASLEEP running the default action routine Halt(). heapStart is the 32-bit heap starting address, and xaddr2 is the extended heap end address. For example, the following statements name, allocate and build a task whose heap (which is where arrays reside) extends from location 0000 on page 5 to 0x7FFF on page 6:

```
TASK Taskname;        // name and allocate task space
BUILD_C_TASK( 0x050000, 0x067FFF, &Taskname);
```

TASK creates and allocates the new TASK structure. BUILD_C_TASK() first calls IsHeap() which initializes the heap accordingly. BUILD_C_TASK() then initializes the task's USER_AREA and task buffers in the 1 kilobyte area starting at the task's base address. The user area of the parent task (i.e., the task that is active when this command executes) is copied to create the new task's USER_AREA, so the parent's configuration is initially "inherited" by the new task. The variables that control the memory map of the new task are set so that the return stack extends downward for 512 bytes at &Taskname + 0x400, the Forth data stack extends downward for up to 128 bytes at &Taskname + 0x200, TIB (QED-Forth terminal input buffer) extends upward for 96 bytes starting at &Taskname + 0x180, the page-change stack (bankstack)

extends downward for 128 bytes starting at &Taskname + 0x180, POCKET (used by QED-Forth interpreter) extends upward for 36 bytes starting at &Taskname + 0x200, and PAD (scratchpad area, available for programmer) extends upward for 88 bytes and downward for 35 bytes starting at &Taskname + 0x100. To initialize CURRENT_HEAP without modifying the heap control variables, pass BUILD_C_TASK() a heapStart address that is equal to the heapEnd address; see IsHeap().
Type: macro; Related QED-Forth function: BUILD.TASK
Header file: mtasker.h

void BytesToDisplay(uint base_addr, uint base_page, uint numbytes, uint display_data_addr)
Sends numbytes of data starting at base_addr on base_page to the graphics display specified by display_data_addr. This is a low-level primitive that is typically not useful to the end user.
Type: _forth function; QED-Forth name: BYTES>DISPLAY
Header file: V4_4Update.h; V4_4Update.c must be #included in 1 file only

int CalcChecksum(xaddr base_addr, uint numbytes)
Calculates a 16-bit checksum for the buffer specified by base_addr and numbytes, where numbytes is even and 0 <= numbytes < 32,768. The buffer must not cross a page boundary. The checksum is calculated by initializing a 16-bit accumulator to zero, then adding in turn each 2-byte number in the buffer to the accumulator; the checksum is the final value of the accumulator. Using this routine provides a method of checking whether the contents of an area of memory have changed since a prior checksum was calculated. This routine is optimized for speed, and executes at less than 3 microseconds per byte.
Type: _forth function; QED-Forth name: CALC.CHECKSUM
Header file: V4_4Update.h; V4_4Update.c must be #included in 1 file only

CALENDAR_TIME
This struct typedef defines the bytes that hold the results of a read of the battery-backed real-time clock. The watch_results instance of this structure is initialized each time ReadWatch is executed. A set of macros (WATCH_SECONDS, WATCH_MINUTES, WATCH_HOUR, etc.) have been predefined to facilitate easy access to the watch results; see the glossary entry for ReadWatch().
Type: typedef; QED-Forth name: READ.WATCH
Header file: watch.h

void ChangeBits(uchar data, uchar mask, xaddr address)
At the byte specified by the 32-bit address, modifies the bits specified by 1's in the mask to have the values indicated by the corresponding bits in the data parameter. In other words, mask specifies the bits at xaddr that are to be modified, and the data parameter provides the data which is written to the modified bits. This function is useful for modifying data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). Disables interrupts for 16 cycles (4 microseconds) to ensure an uninterrupted read/modify/write operation. See ClearBits(), SetBits(), and ToggleBits().
Type: _forth function; QED-Forth name: CHANGE.BITS
Header file: xmem.h

void ChangeTaskerPeriod(uint periodFactor)

Sets periodFactor as the period of the timeslice clock (the OC2 interrupt) in units of 100 microseconds. For example, to set the timeslice period to 0.8 msec, execute

```
ChangeTaskerPeriod(8);
```

Note that the default timeslice increment set after a COLD restart is 5 msec. Implementation detail: Based on the prescaler bits PR1 and PR0 in the TMSK2 register, this routine calculates the period of the clock driving the OC2 timer. It then calculates the number of these periods in the requested timeslice period u, and stores the resulting OC2 timer increment in an unnamed system variable called TIMESLICE_INCREMENT. This stored increment sets the period of the OC2 timer. The period of the OC2 timer determines the timeslice period and also the resolution of the elapsed time clock. Aborts if the calculated increment is 0 or is greater than 65,535. See also TIMESLICE_COUNT and ReadElapsedSeconds().

Type: _forth function; QED-Forth name: *100US=TIMESLICE.PERIOD

Header file: mtasker.h

int CharsPerDisplayLine(void)

Returns the number of characters per line in the LCD display as specified by the last execution of IsDisplay(); valid return values are 8, 12, 16, 20, 24, 30, and 40 characters per line. The default return value after executing the "special cleanup mode" is 20, corresponding to the default 4-line by 20-character display. The result returned by this routine is used by BufferPosition(), PutCursor(), UpdateDisplay(), and UpdateDisplayLine().

Type: _forth function; QED-Forth name: CHARS/DISPLAY.LINE

Header file: intrface.h

void CharToDisplay(char c)

Writes the specified data byte c to the LCD display. Does not write to the Display Buffer. If an alphanumeric (character) display is being used, this command writes the specified ascii character at the current cursor position and increments the cursor position. (Caution: the cursor does not always follow a contiguous path as it is incremented; there may be discontinuities at the ends of lines.) If a graphics display is in use, this function must be used in conjunction with a function that specifies the meaning of the data byte. Intermittently disables interrupts for 28 cycles (7 µsec) per byte written to the display to implement clock stretching. See CommandToDisplay() and UpdateDisplay().

Type: _forth function; QED-Forth name: CHAR>DISPLAY

Header file: intrface.h

void ClearBits(uchar mask, xaddr address)

For each bit of mask that is set, clears the corresponding bit of the 8 bit value at the 32bit address. This function is useful for modifying data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). Disables interrupts for ten cycles (2.5 microseconds) to ensure an uninterrupted read/modify/write operation. See ChangeBits(), SetBits(), and ToggleBits().

Type: _forth function; QED-Forth name: CLEAR.BITS

Header file: xmem.h

void ClearBootVector(void)

Removes a boot vector from page 0x0C. Note that the “page C write protect” jumper must be removed for this function to be effective. This function is called during a "factory cleanup", but it is not called by **NO.AUTOSTART**. See SetBootVector(). This function is typically invoked interactively from the QED-Forth prompt; see CLEAR.BOOT.VECTOR in the Forth debugging glossary.
Type: _forth function; QED-Forth name: CLEAR.BOOT.VECTOR
Header file: V4_4Update.h; V4_4Update.c must be #included in 1 file only

void ClearDisplay(void)

Clears (blanks) the LCD display and moves the cursor to home position (at the start of line 0). If a character display is in use [see IsDisplay()], fills the 80 character DISPLAY.BUFFER with ascii blank characters. If a graphics display is being used in text mode, fills the buffer specified by GARRAY_XPFA with ascii blanks if a Hitachi graphics controller is in use, or with zeros if a Toshiba graphics controller is in use. If a graphics display is being used in graphics mode, erases (zeros) the buffer specified by GARRAY_XPFA. Intermittently disables interrupts for 28 cycles (7 µsec) per byte written to the display to implement clock stretching. See InitDisplay().
Type: _forth function; QED-Forth name: CLEAR.DISPLAY
Header file: intrface.h

CLOCK_MONITOR_ID

A constant that returns the interrupt identity code for the clock monitor interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: CLOCK.MONITOR.ID
Header file: interupt.h

void CmoveMany(xaddr source, xaddr dest, long numBytes)

Moves a block of memory. If the 32-bit byte count numBytes is greater than 0, numBytes consecutive bytes are copied from addresses starting at source to addresses starting at dest. The source and destination extended addresses may be located on different pages and the move may cross page boundaries. If the source and destination regions overlap and source is less than dest, CmoveMany() starts at high memory and moves toward low memory to avoid propagation of the moved contents. CmoveMany() always moves the contents in such a way as to avoid memory propagation. Speed is approximately 19 microseconds per byte.
Type: _forth function; QED-Forth name: CMOVE.MANY
Header file: xmem.h

void Cold(void)

Disables interrupts and restarts the QED-Forth system and initializes all of the user variables to their default values. Initializes the following machine registers:

PORTG, DDRG, TMSK2, SPCR, BAUD, SCCR1, SCCR2, BPROT, OPT2, OPTION, HPRIO, INIT, CSCTL.

Initializes the vectors of the vital interrupts if InitVitalIRQsOnCold() has been executed. Calls Abort() which clears the stacks and calls either the QED-Forth interpreter or an autostart routine that has been installed using Autostart() or PriorityAutostart(). If ColdOnReset() has been executed, every reset or power-up will invoke a Cold() as opposed to a Warm() initialization sequence. This function may be called interactively from the terminal by simply typing:

```
    COLD
```

See also Warm().
Type: _forth function; QED-Forth name: COLD
Header file: qedsys.h

void ColdOnReset(void)

Initializes a flag in EEPROM that causes subsequent resets to execute a cold restart (as opposed to the standard warm-or-cold restart). This option is useful for turnkeyed systems that have an autostart routine installed; any error or reset causes a full Cold() restart which initializes all user variables, after which the autostart routine completes the system initialization and enters the application routine. To revert to the standard reset, call StandardReset(). Note that this function can be executed interactively from QED-Forth by typing at the terminal:

```
    COLD.ON.RESET
```

Implementation detail: Initializes location 0xAE1C in EEPROM to contain the pattern 0x13.
Type: _forth function; QED-Forth name: COLD.ON.RESET
Header file: qedsys.h

void CommandToDisplay(char cmd)

Writes the specified cmd byte to the LCD display as a command (as opposed to a data byte to be displayed). Does not modify the contents of the Display Buffer. Intermittently disables interrupts for 28 cycles (7 µsec) per command byte written to the display to implement clock stretching. See CharToDisplay().
Type: _forth function; QED-Forth name: COMMAND>DISPLAY
Header file: intrface.h

void COPYARRAY(FORTH_ARRAY* src_array_ptr, FORTH_ARRAY* dest_array_ptr)

Dimensions the destination array specified by dest_array_ptr and copies the contents of the source array specified by src_array_ptr into the destination. The source and destination can be in the same or different heaps. See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr.
Type: macro; Related QED-Forth function: COPY.ARRAY
Header file: array.h

void CopyArray(FORTH_ARRAY* src_array_ptr, uint pfa_page, FORTH_ARRAY* dest_array_ptr, uint pfa_page)

Subsidiary function called by the recommended macro COPYARRAY(); see COPYARRAY().
Type: _forth function; QED-Forth name: COPY.ARRAY
Header file: array.h

COP_ID

A constant that returns the interrupt identity code for the computer operating properly (COP) interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: COP.ID
Header file: interupt.h

xaddr CountedString(char* stringAddr, uint strPage)

Converts the specified null-terminated string (located at stringAddr on strPage) into a Forth-style counted string with the count in the first byte and the non-null-terminated string in the remaining bytes. Returns the 32-bit address of PAD which is where the converted counted string is located. The resulting string can be moved to any desired location by the StringMove() function. Note that the size of the PAD buffer puts a limit on the string size; the input string length should be less than 86 bytes. See StringMove() and PAD.
Type: _forth function; QED-Forth name: C$>COUNTED$
Header file: xmem.h

void Cr(void)

Causes subsequent output to appear at the beginning of the next line by emitting a carriage return (ascii 13) followed by a line feed (ascii 10). See Emit().
Type: _forth function; QED-Forth name: CR
Header file: comm.h

int CreateRamp(int start_speed, int end_speed, int acceleration, int ticks_per_sec, RAMP_ELEMENT* starting_ramp_addr, int speeds_per_ramp)

Writes speed_per_ramp +1 entries into the RAMP.ARRAY starting at the specified start_ramp_addr to attain the specified starting and ending speeds and acceleration (or deceleration). Returns the number of steps in the created ramp. start_speed, end_speed and acceleration are all interpreted as positive numbers. Speeds are in units of steps per second if the motor is configured for full stepping, or halfsteps per second if the motor is configured for half stepping. The acceleration is in units of (half) steps per second per second. Speeds are clamped to the attainable range (between 0 and ticks_per_second), and the acceleration is clamped such that a maximum of 10 seconds is spent at any one transient speed in a ramp. Each ramp entry comprises a step_limit which specifies the number of steps to be taken at the speed, and a duty_cycle which specifies the speed (see the glossary entry for SpeedToDuty). If the specified speeds_per_ramp = 0, this function simply writes a "final" speed by setting the step_limit to 0. For non-zero speeds_per_ramp, this routine writes the specified number of ramp entries, plus an additional entry at the final speed with the step_limit set to 0 which tells the StepManager function that this is the final speed in the ramp. Note that higher level calling routines can write over the final speed, or concatenate two ramps to achieve a speed profile that ramps up to a steady speed for a specified number of steps, and then smoothly ramps down to a stopped state. See the high level source file steppers.c in the Demos_and_Drivers directory of the distribution.
Type: _forth function; QED-Forth name: CREATE.RAMP
Header file: stepper.h

CURRENT_HEAP

A user variable (member of the currently active TASK.USER_AREA structure) that holds the 32-bit extended address that specifies the end of the current heap. Other heap control variables are stored just below this address in the heap. See IsHeap().
Type: macro; Related QED-Forth function: CURRENT.HEAP
Header file: user.h

CUSTOM_ABORT

A user variable (member of the currently active TASK.USER_AREA structure) that contains a flag. If the flag is 0 (false), then the standard system SysAbort() routine is performed. If the flag is non-zero (true), then the function whose address is stored in the user variable UABORT is executed when Abort() runs. See Abort(), SysAbort(), and UABORT.
Type: macro; Related QED-Forth function: CUSTOM.ABORT
Header file: user.h

CUSTOM_ERROR

A user variable (member of the currently active TASK.USER_AREA structure) that contains a flag. If the flag is 0 (false), then the default error routine is performed in response to every system error. If the flag is non-zero (true), then the function whose address is stored in the user variable UERROR when an error occurs. See UERROR.
Type: macro; Related QED-Forth function: CUSTOM.ERROR
Header file: user.h

DDRA

A macro that returns the contents of the 8 bit DDRA (data direction for PORTA) register at address 0x8001 in the 68HC11. To configure a PORTA pin to be an output, simply use an assignment statement to write a 1 to the corresponding bit position in DDRA. Similarly, to configure a PORTA pin to be an input, write a 0 to the corresponding bit position in DDRA. Note that the software UART that implements the secondary serial port uses bits 3 and 4 of PORTA, so care must be taken not to alter the direction or state of these bits if the secondary serial port is in use.
Type: macro; Related QED-Forth function: PORTA.DIRECTION
Header file: qedregs.h

DDRD

A macro that returns the contents of the 8 bit PORTD register at address 0x8009 in the 68HC11 which sets the data direction of bits 2-5 of PORTD. PORTD implements the primary serial channel on bits 0 and 1, and the serial peripheral interface (SPI) on bits 2-5 which controls the onboard 12 bit A/D and 8 bit D/A. If these SPI-interfaced devices are on the board, the contents of DDRD should be left in their default state.
Type: macro; Related QED-Forth function: PORTD.DIRECTION
Header file: qedregs.h

void DefaultRegisterInits(void)

Undoes the effect of the InstallRegisterInits() command.
Implementation detail: sets the contents of location 0xAE06 in EEPROM to 0xFF to ensure that default initializations will be used after subsequent resets. The default register initializations are:

Register Name	Register Address	Default Value
OPTION	0x8039	0x33
TMSK2	0x8024	0x02
BPROT	0x8035	0x10
BAUD	0x802B	0x31

Note that calling this function restores the baud rate of the primary serial port to 9600 baud upon subsequent resets and restarts; see Baud1AtStartup().

Type: _forth function; QED-Forth name: DEFAULT.REGISTER.INITS
Header file: qedsys.h

DEFAULT_HEAPEND

A 32-bit constant that returns the value 0x0F4600 cast as an xaddr. Used as an argument for IsHeap() to establish the default 14.5 Kbyte heap on page fifteen. See also DEFAULT_HEAPSTART and INIT_DEFAULT_HEAP().
Type: constant
Header file: heap.h

DEFAULT_HEAPSTART

A 32-bit constant that returns the value 0x0F4600 cast as an xaddr. Used as an argument for IsHeap() to establish the default 14.5 Kbyte heap on page fifteen. See also DEFAULT_HEAPEND and INIT_DEFAULT_HEAP().
Type: constant
Header file: heap.h

void DELETED(FORTH_ARRAY* array_ptr)

De-allocates the heap space assigned to the specified Forth array, and clears the parameter field to indicate that the data structure is no longer dimensioned. It is good programming practice to delete arrays that hold temporary data after the data has been used; this frees the space in the heap for use by other arrays. See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr, and see DIM() for a description of how to dimension arrays.
Type: macro; Related QED-Forth function: DELETED
Header file: array.h

void Deleted(FORTH_ARRAY* array_ptr, uint pfa_page)

Subsidiary function called by the recommended macro DELETED(); see DELETED().
Type: _forth function; QED-Forth name: DELETED
Header file: array.h

void DIM(type, uint numrows, uint numcols, FORTH_ARRAY* array_ptr)

Dimensions the array specified by array_ptr to have the specified number of rows and columns, with each array element sized to hold a parameter of the specified type. Examples of valid "type" parameters are standard identifiers and pre-defined types such as:

int	unsigned int	uint
char	unsigned char	uchar
long	unsigned long	ulong

Note that the uint, uchar, and ulong types are defined in the types.h file.
DIM() first executes DELETED() to de-allocate any heap space previously allocated to the array, and then writes the dimensioning information into the array's parameter field in common RAM and allocates the required number of bytes in the heap. Calls Abort() if there is not enough heap space.
Example of use:
To define an array of unsigned longs named MyArray with 3 rows and 5 columns, execute:

```
FORTH_ARRAY Myarray;
```

```
DIM(ulong, 3, 5, &Myarray);
```

Once an array is declared using FORTH_ARRAY, it may be dimensioned "on the fly". BE SURE TO DIMENSION THE ARRAY WITHIN A FUNCTION THAT IS CALLED AT RUNTIME! COMPILE-TIME OR LINK-TIME DIMENSIONING DOES NOT WORK WITH FORTH ARRAYS! Note that the & operator in front of the array's name tells the compiler that a pointer is being passed. If you forget the & operator, the compiler will warn you that you are attempting to pass an entire structure (the array's parameter structure) as an argument to a function.

To store the value 0x123456 at row=0, column = 1, execute:

```
ARRAYSTORE( 0x123456, 0, 1, &Myarray);
```

To fetch the value stored at row=0, column=1 and assign it to a variable, you could execute:

```
static ulong  retrieved_data;
retrieved_data = ARRAYFETCH( 0, 1, &Myarray);
```

Note: the dimensioned FORTH_ARRAY array is not a C array. It must be initialized and accessed via special fetch and store functions such as ARRAYFETCH(), FARRAYFETCH(), ARRAYSTORE() and FARRAYSTORE() as opposed to C-style pointer arithmetic. The dimensioned FORTH_ARRAY is stored in memory in column-primary order; in other words, sequential elements in a column are stored in sequential memory addresses. This is the reverse of standard C-style arrays. The functions related to FORTH_ARRAYs provide a convenient means of storing data in the large paged memory space of the QED Board; the standard 16-bit ANSI C compiler cannot directly address this extended memory without the aid of the Forth heap manager and memory access functions.

See FORTH_ARRAY, ARRAYSTORE(), FARRAYSTORE(), ARRAYFETCH(), FARRAYFETCH(), ARRAYMEMBER(), ARRAYBASE(), DELETED(), FILLARRAY(), and COPYARRAY().

Type: macro; related function: DIMENSIONED()

Header file: array.h

```
void Dimensioned(uint r, uint c, uint Numdims, uint bytesPerElement,
        FORTH_ARRAY* array_ptr, uint pfa_page)
```

Subsidiary function called by the recommended macro DIM(). Dimensioned() initializes the parameter field in common memory and allocates space for the specified memory in the heap. The parameters specify the number of rows, number of columns, number of dimensions (must equal 2), and bytes per element (should be 1, 2, or 4 so the standard ARRAYFETCH() and ARRAYSTORE() functions can access the data). See DIM().

Type: _forth function; QED-Forth name: DIMENSIONED

Header file: array.h

```
void DIM_AD8_BUFFER(FORTH_ARRAY* array_ptr, uint NumSamples)
```

Dimensions a buffer to accept data from the 8 bit A/D via the AD8ToBuffer() function. Dimensions the specified Forth array to have NumSamples rows, 1 column, and 1 byte per element. DIM_AD8_BUFFER() must be executed once before the first call of AD8ToBuffer(). After that, there is no need to call DIM_AD8_BUFFER() unless you wish to change the dimensions of the buffer. For a detailed example of use, see the glossary entry for AD8ToBuffer() .

Type: macro

Header file: array.h

DISABLE_INTERRUPTS(void)

Sets the interrupt mask bit (the "I bit") in the condition code register to globally disable interrupts.
Type: Macro; Related QED-Forth function: DISABLE.INTERRUPTS
Header file: interupt.h

void DisableSerial2(void)

Disables the secondary serial port (serial2) which is supported by QED-Forth's software UART. Implementation detail: Locally disables the serial2 output interrupt OC4 and disconnects the pin control logic associated with the PA4 output. Locally disables the serial2 input interrupt IC4/OC5. Clears (stores 0 into) the resource variable SERIAL2_RESOURCE.
Type: _forth function; QED-Forth name: DISABLE.SERIAL2
Header file: comm.h

xaddr DisplayBuffer(void)

Returns the 32-bit base address (including page) of the buffer that holds the display data. To write all or part of this buffer to the LCD display, call UpdateDisplay() or UpdateDisplayLine(), respectively. If a character display is in use, the returned xaddr is the base address of an 80 character buffer in the system RAM. If a graphics display is in use, the returned xaddr is the starting address of the array associated with the Forth array pointer GARRAY_XPFA. Each byte in the DisplayBuffer represents a character position or graphical byte on the LCD display. To display characters on the LCD display, simply write the desired ascii characters or graphical data into this buffer and execute UpdateDisplayLine() or UpdateDisplay(). UpdateDisplayLine() causes the contents of a specified line in the DisplayBuffer to be written to the corresponding line of the display. UpdateDisplay() causes the contents of all lines in DisplayBuffer to be written to the corresponding lines of the display. See StringToDisplay() and BufferPosition().
Type: _forth function; QED-Forth name: DISPLAY.BUFFER
Header file: intrface.h

void DisplayOptions(int display_on, int cursor_on, int cursor_blink, int text_mode)

Sets the display and cursor options on the LCD display. Each of the input parameters is a flag that takes a false (0) or true (non-zero) value. If display_on is true, the contents of the display are visible; if false, the display appears blank. If cursor_on is true, the cursor is on (typically an underscore character); if false, the cursor is off. If cursor_blink is true, the cursor blinks (typically a flashing box the size of a single character); if false, the cursor blink is turned off. If text_mode is true, the display is operating in "text mode"; if false, it is operating in "graphics mode". Note that graphics mode should only be specified if a graphics display is in use; see IsDisplay(). Note also that the cursor is never visible in graphics mode. The InitDisplay() function (which is executed upon each reset or restart) leaves the display enabled with the cursor off and cursor blink off.
Implementation detail: In addition to writing the appropriate command byte to the display, DisplayOptions() stores the command byte in an unnamed system variable called PRIOR_CURSOR_STATE. This variable is referenced by UpdateDisplayLine()

and UpdateDisplay() to blank the cursor during updates to character displays (to prevent annoying flickering) and restore it to its prior state after the update is complete. It is also used by LinesPerDisplay() to infer whether the display is being operated in text mode or graphics mode, which in turn determines whether LinesPerDisplay() reports the number of character lines or the number of pixel lines in the display. This routine intermittently disables interrupts for 28 cycles (7 µsec) per command byte written to the display to implement clock stretching.
Type: _forth function; QED-Forth name: DISPLAY.OPTIONS
Header file: intrface.h

DISPLAY_HEAP
A constant that returns the extended address 0x0F45FF that points to the top of the heap containing the graphics array. Used as an argument for IsHeap(). The default display heap is located at 0x3000 to 0x45FF on page 0x0F. Caution: adding items to the DISPLAY.HEAP is not recommended.
Type: constant; Related QED-Forth function: DISPLAY.HEAP
Header file: intrface.h

void DownloadMap(void)
Sets a flag in EEPROM and changes the state of a latch in the onboard PALs to put the download memory map into effect. After execution of this routine, and upon each subsequent reset or restart, hex pages 4, 5, 6, and 0x10-17 are addressed in RAM, and pages 1, 2, 3, and 0x18-1F are addressed in flash memory. This allows code (and Forth names) to be compiled into RAM on pages 4, 5 and 6 (and, if a 512K RAM is present, into pages 0x10-17) and then transferred to flash using the PAGE.TO.FLASH function. To establish the standard memory map, see the glossary entry for StandardMap(). Note that the standard map is active after a "factory cleanup" operation.
Type: _forth function; QED-Forth name: DOWNLOAD.MAP
Header file: flash.h

xaddr DupHeapItem(xaddr xhandle)
Given the 32-bit handle (pointer to a pointer) named xhandle1 of a source heap item, creates a duplicate heap item with identical contents in the same heap and returns its handle. Returns zero if xhandle1 is not a valid handle or if there is insufficient memory in the heap. To copy a heap item into a different heap, use TransferHeapItem().
Type: _forth function; QED-Forth name: DUP.HEAP.ITEM
Header file: heap.h

void Emit(uchar c)
Displays character c by sending it via the serial I/O port. Emit() is a vectored routine that executes the routine whose 32-bit execution address is installed in the user variable UEMIT. The default installed routine called is Emit1() which sends the character via the primary serial port (supported by the 68HC11's hardware UART). Emit2() may be installed in UEMIT by UseSerial2 or Serial2AtStarup(); Emit2() sends the character via the secondary serial port (supported by QED Forth's software UART and using pins PA3 and PA4). See Emit1(), Emit2() and _writeChar().
Type: C function; related QED-Forth function name: EMIT
Header file: comm.h

void Emit1(uchar c)

Displays a character by sending it via the primary serial port (serial1) associated with the 68HC11's on-chip hardware UART. Before sending the character, Emit1() waits (if necessary) for the previous character to be sent, and executes Pause() while waiting to allow other tasks (if present) a chance to run. The most significant byte of the input data stack cell is ignored. Emit1() is the default Emit() routine installed in the UEMIT user variable after the special cleanup mode is invoked or if Serial1AtStartup() has been executed. If the value in the user variable SERIAL_ACCESS is RELEASE_AFTER_LINE, Emit1() does not GET(SERIAL1_RESOURCE) or RELEASE(SERIAL1_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, Emit1() GETs and RELEASEs the SERIAL1_RESOURCE. If SERIAL_ACCESS contains RELEASE_NEVER, Emit1() GETs but does not RELEASE the SERIAL1_RESOURCE. See Emit(), UEMIT, Emit2(), SERIAL_ACCESS.
Type: _forth function; QED-Forth name: EMIT1
Header file: comm.h

void Emit2(uchar c)

Writes the specified ascii character c to the output buffer of the secondary serial port (serial2) for subsequent transmission. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). If the serial2 transmitter is idle (and if the serial2 port and its interrupts have been properly initialized) then the character is transmitted immediately. Otherwise the character will be transmitted after the prior characters in the output buffer are transmitted. If the 80 character output buffer is full when Emit2() is executed, Emit2() executes Pause() and waits until room becomes available in the buffer (as a result of a character being sent out). The most significant byte of the input data stack cell is ignored. Emit2() can be made the default Emit() routine installed in the UEMIT user variable after each reset or restart by executing Serial2AtStartup(). If the value in the user variable SERIAL_ACCESS is RELEASE_AFTER_LINE, Emit2() does not GET(SERIAL2_RESOURCE) or RELEASE(SERIAL2_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, Emit2() GETs and RELEASEs the SERIAL2_RESOURCE. If SERIAL_ACCESS contains RELEASE_NEVER, Emit2() GETs but does not RELEASE the SERIAL2_RESOURCE. See Emit(), UEMIT, Emit1(), SERIAL_ACCESS.
Type: _forth function; QED-Forth name: EMIT2
Header file: comm.h

ENABLE_INTERRUPTS(void)

Clears the interrupt mask bit (the "I bit") in the condition code register to globally enable interrupts.
Type: Macro; Related QED-Forth function: ENABLE.INTERRUPTS
Header file: interupt.h

void Execute(void(*action)(), uint actionPage)

Executes (calls) the action function specified by the function pointer; the function code resides on the specified actionPage.
Type: _forth function; QED-Forth name: EXECUTE
Header file: qedsys.h

EXTENDED_ADDR

A union typedef that provides a way of converting a 16-bit address and associated page into a 32-bit xaddr type, or vis versa. The definition is:

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

For example, the following code converts the address of the variable varname in common RAM (which corresponds to a 16 bit address; the effective page = 0) into a 32 bit xaddr in xaddr_of_varname:

```
char varname;
xaddr xaddr_of_varname;       // we want this to hold a 32bit addr
EXTENDED_ADDR temporary; // allocate union to convert type
temporary.sixteen_bit.addr16 = &varname;
temporary.sixteen_bit.page16 = 0;      // common page = 0
xaddr_of_varname = temporary.addr32;          // here's the result
```

See the source code in the TYPES.H file.

Type: typedef

Header file: types.h

FALSE

A constant equal to 0.

Type: constant; Related QED-Forth function: FALSE

Header file: utility.h

float FARRAYFETCH(type, uint row, uint col, FORTH_ARRAY* array_ptr)

Fetches the contents of the floating point element at row#, column# in the specified 2-dimensional array and casts it to the specified type. Typically, the "float" type will be specified when this function is called, although other compatible types may be specified (see the glossary entry for ARRAYFETCH() which handles non-floating-point data). There is an unchecked error if the specified array does not have 2 dimensions or if the number of bytes per element does not equal 4. See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM(), ARRAYFETCH() and FARRAYSTORE().

Type: macro; Related QED-Forth function: ArrayFetch()

Header file: array.h

float FArrayFetch(uint row, uint col, FORTH_ARRAY* array_ptr, uint pfa_page)

A subsidiary function called by the recommended macro FARRAYFETCH(); see FARRAYFETCH().

Type: _forth function; QED-Forth name: 2ARRAY.FETCH

Header file: array.h

void FARRAYSTORE(float value, uint row, uint col, FORTH_ARRAY* array_ptr)

Stores the specified floating point value at row, col in the 2-dimensional FORTH_ARRAY specified by array_ptr. Use ARRAYSTORE() to store non-floating-point data. There is an unchecked error if the specified array does not have 2

dimensions or if the number of bytes per element does not equal 4. See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM() and FARRAYFETCH().
Type: macro; Related QED-Forth function: 2ArrayStore()
Header file: array.h

void FArrayStore(float value, uint row, uint col, FORTH_ARRAY* array_ptr, uint pfa_page)
A subsidiary function called by the recommended macro FARRAYSTORE(); see FARRAYSTORE().
Type: _forth function; QED-Forth name: 2ARRAY.STORE
Header file: array.h

void FastAD8Multiple(xaddr buffer, uint Interval, uint NumSamples, uint channel)
Acquires NumSamples samples from the 8 bit analog to digital (A/D) converter in the 68HC11 and stores the samples as sequential unsigned 8 bit values starting at the specified buffer xaddress. [For an automated approach to storing samples in an array in paged memory, see the glossary entries for DIM_AD8_BUFFER() and AD8ToBuffer()]. The channel parameter specifies the channel number of the A/D (0 <= channel <= 7). To maximize speed, this routine does not GET() or RELEASE() the AD8_RESOURCE. Consequently, this routine should not be used in a multitasking environment where another task might require access to the 8 bit A/D; see AD8Multiple(). If the specified xaddr is in common memory, the first sample is taken after 16 µsec and subsequent samples are taken every (10+2.5*u1) µsec, where u1 is the specified timing parameter passed to this routine. If the specified xaddr is in paged memory, the first sample is taken after 11 µsec and subsequent samples are taken every (32.5+2.5*u1) µsec. Of course, the operation of interrupts (including timesliced multitasking) will affect these sampling times. See FastAD8Sample(), AD8Sample(), AD8Multiple(), DIM_AD8_BUFFER(), AD8ToBuffer() and AD8On().
Type: _forth function; QED-Forth name: (A/D8.MULTIPLE)
Header file: analog.h

uchar FastAD8Sample(uint channel)
Acquires and places on the stack a single sample byte from the specified channel (0 <= channel <= 7) of the 8 bit analog to digital (A/D) converter in the 68HC11. To maximize speed, this routine does not GET() or RELEASE() the AD8_RESOURCE. Consequently, this routine should not be used in a multitasking environment where another task might require access to the 8 bit A/D; see AD8Sample(). This routine executes in 23 microseconds. See AD8Sample(), FastAD8Multiple(), AD8Multiple(), and AD8On().
Type: _forth function; QED-Forth name: (A/D8.SAMPLE)
Header file: analog.h

char FetchChar(xaddr address)
Fetches an 8-bit value from the specified extended address. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().
Type: _forth function; QED-Forth name: C@
Header file: xmem.h

float FetchFloat(fxaddr address)

Fetches a 32-bit floating point number from the specified extended address. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().
Type: _forth function; QED-Forth name: F@
Header file: xmem.h

float FetchFloatProtected(xaddr address)

Fetches a floating point value from the specified extended address. Disables interrupts during the fetch to ensure that an interrupting routine or task does not modify the contents while the fetch is in process. Disables interrupts for 28 cycles (7 microseconds) unless the specified 4 bytes straddle a page boundary, in which case interrupts are disabled for approximately 260 cycles. Note that in paged memory, the address immediately following 0x7FFF is address 0000 on the following page. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). See also StoreFloatProtected().
Type: _forth function; QED-Forth name: |F@|
Header file: xmem.h

int FetchInt(xaddr address)

Fetches a 16-bit number from the memory location specified by address. The high order byte is taken from address and the low order byte from address+1. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER()
Type: Forth function; QED-Forth name: @
Header file: xmem.h

long FetchLong(xaddr address)

Fetches a 32-bit value from the specified extended address. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().
Type: _forth function; QED-Forth name: 2@
Header file: xmem.h

long FetchLongProtected(xaddr address)

Fetches a 32-bit value from the specified extended address. Disables interrupts during the fetch to ensure that an interrupting routine or task does not modify the contents while the fetch is in process. Disables interrupts for 28 cycles (7 microseconds) unless the specified 4 bytes straddle a page boundary, in which case interrupts are disabled for approximately 260 cycles. Note that in paged memory, the address immediately following 0x7FFF is address 0000 on the following page. This function is useful for fetching data from arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). For floating point values, use FetchFloatProtected().
Type: _forth function; QED-Forth name: |2@|
Header file: xmem.h

void FILLARRAY(FORTH_ARRAY* array_ptr, uchar c)

Stores c into each byte of the specified Forth array. For descriptions of how to define and dimension a Forth array that resides in paged memory, see the glossary entries for FORTH_ARRAY and DIM().
Type: macro; Related QED-Forth function: FILL.ARRAY
Header file: array.h

void FillArray(FORTH_ARRAY* array_ptr, uint pfa_page, uchar c)

A subsidiary function called by the recommended macro FILLARRAY(); see FILLARRAY().
Type: _forth function; QED-Forth name: FILL.ARRAY
Header file: array.h

void FillMany(xaddr base, long numBytes, char contents)

The specified byte contents is stored in each of numBytes consecutive addresses beginning at the 32-bit extended address base. The specified address region may cross page boundaries. Does nothing if numBytes = 0.
Type: _forth function; QED-Forth name: FILL.MANY
Header file: xmem.h

FILL_FIELD

A user variable that contains a flag. If the flag is true (non-zero), floating point numbers converted to strings by FPtoString() or printed in FLOATING format by PrintFP() are padded with spaces to yield a constant field width irrespective of whether the number is printed in scientific notation or fixed notation, and numbers printed in fixed notation are decimal aligned. This leads to neat printouts of tabular data. If the flag is false, the field width is not padded out. See FPtoString(), PrintFP() and FLOATING().
Type: macro; Related QED-Forth function: FILL.FIELD
Header file: numbers.h

void FIXED(void)

Sets the default printing format used by FPtoString() and PrintFP() to fixed. Numbers are decimal aligned, and RIGHT_PLACES and LEFT_PLACES determine the field width. See the glossary entry for FPtoString() for more details.
Type: macro; Related QED-Forth function: FIXED
Header file: numbers.h

FLOATING()

Sets the default printing format used by FPtoString() and PrintFP() to floating. This format displays the number in FIXED format if the number can be represented with the same or more significant digits as it would if it were represented in SCIENTIFIC format. Otherwise, it uses SCIENTIFIC format. See the glossary entry for FPtoString() for more details.
Type: macro; Related QED-Forth function: FLOATING
Header file: numbers.h

FMAILBOX

This typedef allocates a 32-bit mailbox in RAM which can be accessed by FSEND(), TRY_TO_FSEND() and FRECEIVE(). The mailbox can hold any floating-point "message"; for non-floating-point messages, use MAILBOX. Example of use:

```
FMAILBOX latest_data;
SEND( 3.14159, &latest_data);
```

Mailboxes are used in multitasked systems to share information between tasks and to synchronize tasks to one another. If the mailbox's contents equal zero, the mailbox is empty; it contains a message if its contents are non-zero. Before its first use, the mailbox must be initialized to zero. After initialization to zero, the only operators that should access the floating point mailbox are FSEND(), TRY_TO_FSEND() and FRECEIVE().

Type: typedef; Related QED-Forth function: MAILBOX:
Header file: mtasker.h

int ForthAskKey(void)

A subsidiary function to AskKey(); see AskKey().
Type: _forth function; QED-Forth name: ?KEY
Header file: comm.h

int ForthEmit(void)

A subsidiary function to Emit(); see Emit().
Type: _forth function; QED-Forth name: EMIT
Header file: comm.h

int ForthKey(void)

A subsidiary function to Key(); see Key().
Type: _forth function; QED-Forth name: KEY
Header file: comm.h

FORTH_TASK

A macro that represents the TASKBASE at address 0x8400 for the default QED-Forth task; defined as:

```
_at(0x8400) TASK FORTH_TASK;
```

See TASK.
Type: macro
Header file: mtasker.h

FORTH_ARRAY

Declares a new Forth-style array. Use as:

```
FORTH_ARRAY array_name;
```

where array_name is any name of your choosing. The declaration allocates a "parameter field" structure in the variable area. This structure is initialized by DIM() to hold the dimensioning information for the array (number of rows and columns, element size, etc.) as well as pointers to the Forth heap and heap item that contain the array data.

Example of use:
To define an array of unsigned longs named MyArray with 3 rows and 5 columns, execute:

```
FORTH_ARRAY Myarray;
```

```
DIM(ulong, 3, 5, &Myarray);
```

Note that the & (address-of) operator in front of the array's name tells the compiler that a pointer is being passed. If you forget the & operator, the compiler will warn you that you are attempting to pass an entire structure (the array's parameter field structure) as an argument to a function.

To store the value 0x123456 at row=0, column = 1, execute:

```
ARRAYSTORE( 0x123456, 0, 1, &Myarray);
```

To fetch the value stored at row=0, column=1 and assign it to a variable, you could execute:

```
static ulong  retrieved_data;
retrieved_data = ARRAYFETCH( 0, 1, &Myarray);
```

Note: the dimensioned FORTH_ARRAY array is not a C array. It must be initialized and accessed via special fetch and store functions such as ARRAYFETCH(), FARRAYFETCH(), ARRAYSTORE() and FARRAYSTORE() as opposed to C-style pointer arithmetic. The dimensioned FORTH_ARRAY is stored in memory in column-primary order; in other words, sequential elements in a column are stored in sequential memory addresses. This is the reverse of standard C-style arrays. The functions related to FORTH_ARRAYs provide a convenient means of storing data in the large paged memory space of the QED Board; the standard 16-bit ANSI C compiler cannot directly address this extended memory without the aid of the Forth heap manager and memory access functions.

See DIM(), ARRAYSTORE(), FARRAYSTORE(), ARRAYFETCH(), FARRAYFETCH(), ARRAYMEMBER(), ARRAYBASE(), DELETED(), FILLARRAY(), and COPYARRAY().

Type: macro; related function: ARRAY:

Header file: array.h

char* FPtoString(float ansi_fp_num)

Converts the specified ansi_fp_num floating point number to a standard null-terminated ascii string, and returns the address of the string. If the conversion fails, returns 0. The specified number is converted into one of three formats: FIXED, SCIENTIFIC, or FLOATING. To set the default format, execute one of the macros FIXED(), SCIENTIFIC(), or FLOATING(). FLOATING format is the default after a COLD restart. Each format is described in detail here:

FIXED()

If FIXED() has been executed, FPtoString() converts the input number into a text string using the following format: (an optional) sign, LEFT_PLACES digits, a decimal point, RIGHT_PLACES digits, and a trailing space, as

```
-xxx.yyy
```

If the user variable NO_SPACES is false (the default condition), the field size equals LEFT_PLACES + RIGHT_PLACES + 3 and numbers are decimal aligned. The size of the string is clamped to a maximum of 32 characters. Setting the user variable TRAILING_ZEROS true displays all trailing zeros to the right of the decimal point, to a maximum specified by the contents of the user variable RIGHT_PLACES. If the input number cannot be represented as an ascii string in FIXED format (that is, if the values of LEFT_PLACES and RIGHT_PLACES won't allow the number to be represented in FIXED format) then FPtoString() returns 0.

SCIENTIFIC()

If SCIENTIFIC() has been executed, FPtoString() converts the input number into a text string using the following format: (an optional) sign, single digit, decimal point, MANTISSA.PLACES digits, E, exponent sign, 2-digit exponent, and a trailing space, as

```
-1.xxxxE-yy
```

The field size is 8 plus the contents of the user variable MANTISSA_PLACES. The string includes a trailing space unless NO_SPACES is true. Any valid floating point number can be represented in the SCIENTIFIC format, so a valid string pointer is always returned.

FLOATING()

If FLOATING() has been executed, FPtoString() selects FIXED format unless the number can be displayed with greater resolution using scientific notation, in which case SCIENTIFIC() format is used. If the user variable FILL_FIELD equals zero (the default condition), the string is displayed using the minimum possible field size, and numbers are not decimal aligned. If FILL_FIELD is true (non-zero), the field size of the string is always equal to the scientific field size, which is MANTISSA_PLACES+8, and numbers are decimal aligned for neat display of tabular data. The string includes a trailing space unless NO_SPACES is true. A valid string address is always returned because any valid floating point number can be represented in the FLOATING format.

See also PrintFP().

Type: _forth function; QED-Forth name: FPtoString
Header file: numbers.h

ulong FP_CtoQ(float ansi_fp_num)

Converts the ANSI/IEEE-standard formatted input floating point number into the QED-Forth floating point format as described in the QED-Forth Software Manual. The returned QED-formatted float is declared as an unsigned long to prevent the C compiler from corrupting the value. Converts denormalized input numbers to zero; that is, if the biased exponent = 0, the returned QED-formatted floating point number = zero. NAN (not a number) inputs are converted to +/- infinity depending on their sign bit. The least significant bit (lsb) of the mantissa is not rounded, resulting in up to 1 lsb error during the conversion.

Type: _forth function; QED-Forth name: FP_CtoQ
Header file: numbers.h

FP_FORMAT

A user variable (member of the currently active user structure) that specifies the format to be used by subsequent executions of PrintFP() and FPtoString. FP_FORMAT is typically accessed indirectly by means of the format specifiers FIXED(), FLOATING(), and SCIENTIFIC(); see their glossary entries for more details.

Type: macro
Header file: numbers.h

float FP_QtoC(ulong qed_fp_number)

Converts the QED-Forth formatted input floating point format into an ANSI/IEEE-standard formatted floating point number. The input QED-formatted float is declared as an unsigned long to prevent the C compiler from corrupting the value.

Type: _forth function; QED-Forth name: FP_QtoC
Header file: numbers.h

float FRECEIVE(float* mailboxAddr)

If mailboxAddr is empty (ie., if it contains 0.0), executes Pause() until the mailbox contains a message. If xmailbox contains a message (that is, if it does not contain zero), returns the floating point contents of mailboxAddr and stores 0.0 into mailboxAddr to indicate that the message has been received and that the mailbox is now empty. To receive and send non-floating-point messages, use RECEIVE() and SEND(). To ensure that the state of the mailbox is correctly determined, RECEIVE() disables interrupts for 26 to 61 cycles (6.5 to 15.25 microseconds). See FSEND() and MAILBOX.
Type: macro; Related QED-Forth function: RECEIVE
Header file: mtasker.h

float FReceive(float* mailboxAddr, uint mailboxPage)

A subsidiary function called by the recommended macro FRECEIVE(); see FRECEIVE().
Type: _forth function; QED-Forth name: RECEIVE
Header file: mtasker.h

xaddr FromHeap(ulong size)

If size bytes are available in the heap, allocates them and returns a 32-bit xhandle (pointer to a pointer) that indirectly points to the 32-bit base xaddress of the allocated heap item. Adjusts size upward so that it is an even multiple of 4, and allocates the heap item so that its base address is an even multiple of 4. Returns 0 if there is not enough heap space to perform the allocation, or if the allocated handle is within 5 bytes of the bottom of CURRENT_HEAP's page (handles must be on the same page as CURRENT_HEAP).
Type: _forth function; QED-Forth name: FROM.HEAP
Header file: heap.h

void FSEND(float message, float * mailboxAddr)

Executes Pause() until the mailbox with extended address mailboxAddr is empty (contains 0.0) and then stores the 32-bit floating point message in mailboxAddr. The message can be any 32-bit floating point number except zero; use SEND() to send a non-floating-point value as a message. To ensure that the state of the mailbox is correctly determined, FSEND() disables interrupts for 16 to 50 cycles (4 to 12.5 microseconds). See TRY_TO_FSEND, FRECEIVE() and MAILBOX.
Type: macro; Related QED-Forth function: SEND
Header file: mtasker.h

void FSend(float message, float * mailboxAddr, uint mailboxPage)

A subsidiary function called by the recommended macro FSEND(); see FSEND().
Type: _forth function; QED-Forth name: SEND
Header file: mtasker.h

GARRAY_XPFA

A macro that returns the 32-bit xpfa (extended parameter field address) that specifies the graphics data array; an xpfa is also referred to as an "array_ptr" in function prototypes. This otherwise unnamed array is dimensioned by InitDisplay() if a graphics

display has been selected using IsDisplay(). UpdateDisplayLine() and UpdateDisplay() write the contents of this array to the graphics display. DisplayBuffer() returns the xaddr of the first element in this array if a graphics display is in use. See the graphics extension routines that are supplied in source code form to augment the ROM libraries; these routines provide examples of how to access information in the graphics array.
Type: macro; Related QED-Forth function: GARRAY.XPFA
Header file: intrface.h

void GET(xaddr* resourceAddr)
Used in a multitasking system to gain access to a shared resource. Executes Pause() until the resource variable whose address is resourceAddr is available, and then GETs the resource by storing the task ID (i.e., the base address of the TASK structure) of the requesting task into the resourceAddr. A 32-bit zero in resourceAddr indicates that the resource is available, and a non-zero value that is not equal to the requesting task's ID indicates that another task controls the resource. To ensure that the state of the resource is correctly determined, GET() disables interrupts for 27 to 57 cycles (6.75 to 14.25 microseconds). See TRY_TO_GET(), RELEASE(), TASK and RESOURCE.
Type: macro; Related QED-Forth function: GET
Header file: mtasker.h

void Get(xaddr* resourceAddr, uint resourcePage)
A subsidiary function called by the recommended macro GET(); see GET().
Type: _forth function; QED-Forth name: GET
Header file: mtasker.h

void Halt(void)
An infinite loop whose action is to put the calling task ASLEEP and execute Pause(). Typically used to terminate a task action that is not itself an infinite loop.
Type: _forth function; QED-Forth name: HALT
Header file: mtasker.h

IC1_ID
A constant that returns the interrupt identity code for input capture 1 which is associated with port bit PA2. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: IC1.ID
Header file: interupt.h

IC2_ID
A constant that returns the interrupt identity code for input capture 2 which is associated with port bit PA1. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: IC2.ID
Header file: interupt.h

IC3_ID
A constant that returns the interrupt identity code for input capture 3 which is associated with port bit PA0. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: IC3.ID
Header file: interupt.h

IC4_OC5_ID

A constant that returns the interrupt identity code for input capture 4/ output compare 5. This interrupt can control the action of port bit PA3. Note that the optional secondary serial port uses IC4/OC5 and PA3. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: IC4/OC5.ID
Header file: interupt.h

ILLEGAL_OPCODE_ID

A constant that returns the interrupt identity code for the illegal opcode interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: ILLEGAL.OPCODE.ID
Header file: interupt.h

void InitDisplay(void)

Initializes the liquid crystal display (LCD) interface. If a graphics-style display has been specified by IsDisplay(), initializes the DISPLAY_HEAP and dimensions GARRAY_XPFA to point to an appropriately sized array in that heap; the base address of this array is returned by DisplayBuffer(). If a character-style (alphanumeric) display has been specified by IsDisplay(), then the display buffer is located in the system RAM and the DISPLAY_HEAP and GARRAY_XPFA are not initialized. If the dimensions specified by IsDisplay() call for a graphics array that is larger than the available Room() in the DISPLAY_HEAP, then InitDisplay() will not dimension the array; see the glossary entry of DISPLAY_HEAP. InitDisplay() calls ClearDisplay() to clear the DisplayBuffer() and write the blank data to the LCD display. Homes the cursor to the start of line 0, and leaves the display enabled with the cursor off and not blinking. See ClearDisplay(). Intermittently disables interrupts for 28 cycles (7 µsec) per byte transmitted to the display to implement clock stretching.
Type: _forth function; QED-Forth name: INIT.DISPLAY
Header file: intrface.h

void InitElapsedTime(void)

Initializes the 32-bit contents of the system variable TIMESLICE_COUNT to zero. See ReadElapsedSeconds(), StartTimeslicer() and StopTimeslicer().
Type: _forth function; QED-Forth name: INIT.ELAPSED.TIME
Header file: mtasker.h

void InitRS485(void)

Calls InitPIA() to configure the peripheral interface adaptor (PIA) so that it is consistent with operation of the RS485 circuitry, and then sets the RS485 transceiver to receive mode. Recall that InitPIA() expects to see two input parameters: the first parameter is true if PPA is to be an output, and the second is true if upper PPC is to be an output. InitRS485() sets the first parameter so as to leave the data direction of PPA unchanged, and sets the second parameter passed to InitPIA() to TRUE to configure upper PPC as an output. PPC bit 4 controls the direction of the RS485 data transfer: when bit 4 of PPC is high, the RS485 port is in transmit mode, and when bit 4 of PPC is low, the RS485 port is in receive mode. (NOTE: Make sure that the onboard RS485/RS232 jumper is properly set before attempting to use the RS485 interface). See InitPIA(), RS485Receive(), and RS485Transmit().
Type: _forth function; QED-Forth name: INIT.RS485

Header file: comm.h

void InitSerial2(void)

Initializes the secondary serial port (serial2) which is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). Clears the contents of resource variable SERIAL2_RESOURCE to zero, initializes PARITY to OFF (no parity), initializes the transmit and receive buffers (80 characters each, located in the reserved system RAM), initializes the data directions of PA3 and PA4 as input and output, respectively, and locally enables the required interrupts associated with PA3 and PA4. Does not globally enable interrupts. The programmer must separately execute the Baud2() command (to set the baud rate) and execute ENABLE_INTERRUPTS (to globally enable interrupts) before using the serial2 port. See UseSerial2() and DisableSerial2().
Type: _forth function; QED-Forth name: INIT.SERIAL2
Header file: comm.h

void InitSPI(void)

Configures and enables the serial peripheral interface (SPI) so that it can transfer data to and from the on-board battery-backed real-time clock. The SPI uses bits 2-5 of PORTD. Initializes the 68HC11 as the SPI "master" with 2 MHz data transfer, with valid data present/sampled on the falling trailing edge of the SPI clock. Initializes the contents of DDRD (PORTD direction register) to be compatible with being the master of the SPI (that is, PD2/MISO = input, PD3/MOSI = output, PD4/SCK = output, PD5/SS = output). Also initializes the contents of the resource variable SPI_RESOURCE to zero.
Type: _forth function; QED-Forth name: INIT.SPI
Header file: analog.h

void InitVitalIRQsOnCold(void)

Undoes the effect of the NoVitalIRQInit() command, and causes subsequent cold restarts to perform the default action of checking the interrupt vectors for the COP, clock monitor, illegal opcode and OC2 interrupts and initializing them if they do not contain the standard interrupt service vectors. Implementation detail: sets location 0xAE1B in EEPROM to 0xFF.
Type: _forth function; QED-Forth name: INIT.VITAL.IRQS.ON.COLD
Header file: qedsys.h

INIT_DEFAULT_HEAP()

A macro that initializes a 14.5 Kbyte heap in page 0x0F at addresses 0x4600 to 0x7FFF. This is the default heap located in page fifteen RAM just above the reserved graphics heap. See also IsHeap(), DEFAULT_HEAPSTART and DEFAULT_HEAPEND.
Type: macro
Header file: heap.h

void InstallMultitasker(void)

Installs the timeslice multitasker timer by initializing the interrupt vector of the output compare 2 (OC2) timer. This command is automatically executed upon a COLD restart (unless the command NoVitalIRQInit() has been executed) and by the command

StartTimeslicer(). Because the interrupt vector is in non-volatile EEPROM, it is usually not necessary to invoke this command unless the OC2 interrupt vector has been modified.
Type: _forth function; QED-Forth name: INSTALL.MULTITASKER
Header file: mtasker.h

void InstallRegisterInits(uchar option, uchar tmsk2, uchar bprot, uchar baud)

Compiles a 6-byte sequence into the EEPROM that specifies the contents to be loaded into the "protected registers" plus the BAUD register after subsequent resets. The protected registers are those that must be initialized within 64 machine cycles after a reset; after that their contents cannot be changed. They are INIT, OPTION, TMSK2, and BPROT. The BAUD register controls the BAUD rate of the primary serial communications interface (serial1), and is included so that a user-specified baud rate can be set upon every restart [see also Baud1AtStartup()]. The INIT register controls the location of the on-chip RAM and the registers. This value is set to 0xB8 (on-chip RAM at 0xB000, and registers at 0x8000); other values are not compatible with QED-Forth. The contents of the other 4 registers may be specified by the user. Once INSTALL.REGISTER.INITS is executed, subsequent resets will cause 0xB8 to be stored in INIT, byte1 in OPTION, byte2 in TMSK2, byte3 in BPROT, and byte4 in BAUD. To undo the effects of this function and return to the default contents of the protected registers use the DefaultRegisterInits() command; see its glossary entry for a list of the default values for each of the registers.
Implementation detail: InstallRegisterInits() writes the pattern 0x13 at location 0xAE06 in the EEPROM. The five bytes following the pattern contain the specified contents of INIT (=0xB8), OPTION, TMSK2, BPROT, and BAUD, respectively.
Type: _forth function; QED-Forth name: INSTALL.REGISTER.INITS
Header file: qedsys.h

IRQ_ID

A constant that returns the interrupt identity code for the external interrupt request interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: IRQ.ID
Header file: interupt.h

void IsDisplay(int numRows, int numCols, int textMode, int charDisplay, int hitachi)

Based on the specified number of rows, number of columns, and flags that indicate text or graphics mode, character versus graphics display, and Hitachi versus Toshiba graphics controller chip, this routine saves the display configuration in EEPROM so that the LCD display is properly initialized upon subsequent restarts and resets by the InitDisplay() routine which is automatically executed at startup. The encoded information is accessible via the routines CharsPerDisplayLine() and LinesPerDisplay(). When IsDisplay() is executed, numRows and numCols should be expressed as the number of 8x6- or 8x8-pixel characters that the screen can accommodate. The standard width font for Toshiba graphics displays is set by hardware inputs on the display module to either 6 or 8 pixels wide. The standard width font for Hitachi graphics displays is 8 pixels in graphics mode, and can be set to either 6 pixels or 8 pixels wide in text mode. The allowed values of numRows are 2, 4, 8 or 16 lines per display. The allowed values of numCols are 8, 12, 16, 20, 24, 30, and 40 characters or bytes per line. The textMode input parameter selects between text

mode (if textMode is true/non-zero) and graphics mode (if textMode is false/zero) for graphics displays; character displays always operate in text mode. The charDisplay input parameter selects between a strictly alphanumeric character display if charDisplay is true, and a graphics display if the charDisplay is false. The hitachi input parameter specifies the type of controller that drives the graphics display module. If hitachi is true, a Hitachi 61830 controller chip is assumed; if hitachi is false, we assume a Toshiba 6963 graphics controller chip. NOTE that if a graphics display is specified (charDisplay is false) but the text mode is specified (textMode is true), the data buffer created by InitDisplay() in the DisplayHeap() will be too small to accommodate graphical data. Thus if you want to use both the text and graphics modes of a graphics display, declare a graphics mode display (i.e., with a false textMode flag), and use the DisplayOptions() routine to convert to and from text mode. Then the dimensioned buffer will be large enough for either character or graphical data. The following appropriately named function calls make it easy to specify the most commonly used displays:

```
void Character4x20(void) { IsDisplay( 4,20,-1,-1,-1 ); }
void HitachiGraphics128x240(void) { IsDisplay( 16,30,0,0,-1 ); }
void HitachiText128x240(void) { IsDisplay( 16,40,-1,0,-1 ); }
void ToshibaGraphics128x240(void) { IsDisplay( 16,40,0,0,0 ); }
void ToshibaText128x240(void) { IsDisplay( 16,40,-1,0,0 ); }
void HitachiGraphics128x128(void) { IsDisplay( 16,16,0,0,-1 ); }
void HitachiText128x128(void) { IsDisplay( 16,20,-1,0,-1 ); }
```

The 4x20 character display is the default type that is established by the "special cleanup mode". Remember to execute InitDisplay() after executing IsDisplay() the first time. Note that because IsDisplay() saves the configuration information in EEPROM, you need not execute it each time the board starts up. InitDisplay() is automatically executed each time the QED Board starts up.

Implementation detail: This routine encodes the configuration information in a single byte that is saved at location 0xAE1E in EEPROM.

Type: _forth function; QED-Forth name: IS.DISPLAY

Header file: intrface.h

void IsDisplayAddress(uint RamAddress)

Configures a graphics display so that the next data write will occur at the specified RamAddress in the display RAM. This routine can be used in conjunction with UpdateDisplayRam() to write data to the "off-screen" RAM that is typically present on a graphics display module. Then modifying the "home address" (upper left location) of the display allows scrolling of data across the display; see the source code of the graphics extension source code file for more details. IsDisplayAddress() has no effect if a character display is installed.

Type: _forth function; QED-Forth name: IS.DISPLAY.ADDRESS

Header file: intrface.h

void IsHeap(xaddr start, xaddr end)

Initializes the heap control variables to set up a heap starting at the specified 32-bit start address and ending 1 byte below the specified 32-bit end address. All of the bytes between start and end must be modifiable RAM. The size of the heap and of individual heap items is limited only by available memory. If the specified heap size (end - start) is greater than or equal to 16 bytes, IsHeap() initializes the user variable

CURRENT_HEAP to end, and initializes heap variables (located near the top of the heap) to indicate that the specified memory region can be used for the heap and that there are no allocated heap items. If the specified heap size (end - start) is less than 16 bytes, only the user variable CURRENT_HEAP is initialized, and the heap control variables that are stored in the heap itself are not initialized. This allows tasks to share a heap which has already been initialized without disturbing the values of the heap control variables. Caution: sharing a heap among tasks may lead to hard-to-diagnose multitasking failures. Consult the chapters on multitasking and re-entrant coding in the Software Manual when designing multitasking programs. See also INIT_DEFAULT_HEAP(), DEFAULT_HEAPSTART, and DEFAULT_HEAPEND.
Type: _forth function; QED-Forth name: IS.HEAP
Header file: heap.h

uchar Key(void)

Waits (if necessary) for receipt of a character from the serial port and places the character on the data stack. Key() is a vectored routine that executes the routine whose xcfa is stored in the user variable UKEY. The default installed routine called is Key1() which receives the character from the primary serial port (supported by the 68HC11's hardware UART). Key2() may be installed in UKEY by UseSerial2() or Serial2AtStartup(); Key2() receives the character from the secondary serial port (supported by QED Forth's software UART and using pins PA3 and PA4). See Key1(), Key2() and _readTerminal().
Type: C function; related QED-Forth function name: KEY
Header file: comm.h

uchar Key1(void)

Waits (if necessary) for receipt of a character from the primary serial port (serial1) and returns the received character. Key1() does not echo the character. The serial1 port is associated with the 68HC11's on-chip hardware UART. Key1() is the default Key() routine installed in the UKEY user variable if Serial1AtStartup() has been executed (and after the special cleanup mode is invoked). If the value in SERIAL_ACCESS is RELEASE_AFTER_LINE, Key1() does not execute GET(SERIAL1_RESOURCE) or RELEASE(SERIAL1_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, Key1() GETs and RELEASEs the SERIAL1_RESOURCE. If SERIAL_ACCESS contains RELEASE_NEVER, Key1() GETs but does not RELEASE the SERIAL1_RESOURCE. See Key(), UKEY, Key2(), and SERIAL_ACCESS.
Type: _forth function; QED-Forth name: KEY1
Header file: comm.h

uchar Key2(void)

Waits (if necessary) for receipt of a character from the secondary serial (serial2) port, removes the character from the serial2 input buffer and returns the received character. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). Key2() does not echo the received character. Key2() can be made the default Key() routine installed in the UKEY user variable after each reset or restart by executing Serial2AtStartup(). If the value in SERIAL_ACCESS is RELEASE_AFTER_LINE, Key2() does not execute GET(SERIAL2_RESOURCE) or RELEASE(SERIAL2_RESOURCE). If SERIAL_ACCESS contains RELEASE_ALWAYS, Key2() GETs and RELEASEs the SERIAL2_RESOURCE. If

SERIAL_ACCESS contains RELEASE_NEVER, Key2() GETs but does not RELEASE the SERIAL2_RESOURCE. See Key(), UKEY, Key1(), and SERIAL_ACCESS.
Type: _forth function; QED-Forth name: KEY2
Header file: comm.h

int Keypad(void)

Scans keypad or touchscreen having up to 8 rows and 5 columns and waits for a keypress. Executes Pause() while waiting to give other tasks (if present) a chance to run. Waits until the key is released, then returns the key number on the data stack. Disables interrupts for 12 microseconds each time a row is scanned. The keymap is as follows:

```
39  35  31  27  23
38  34  30  26  22
37  33  29  25  21
36  32  28  24  20
19  15  11   7   3
18  14  10   6   2
17  13   9   5   1
16  12   8   4   0
```

Note that the behavior with respect to 4-row by 5-column keypads is unchanged, so legacy 20-key hardware operates as it did under prior kernel versions. The support for keys 20 through 39 enables the use of larger keypads on the Handheld product. See ScanKeypad() and ScanKeypress().
Type: _forth function; QED-Forth name: KEYPAD
Header file: intrface.h

void Kill(TASK* taskBase, uint taskPage)

Puts ASLEEP and removes from the round robin multitasking loop the task whose TASKBASE address is taskBase. The task to be killed must be installed in the round robin loop when Kill() is called. If it isn't, or if a task attempts to KILL itself, the results are unpredictable. Aborts if taskBase is not in common RAM. Note that input parameter taskPage always equals 0, indicating that the task is located in common RAM. See TASK and TASKBASE.
Type: _forth function; QED-Forth name: KILL
Header file: mtasker.h

LEFT_PLACES

A user variable that specifies the number of digits to be displayed to the left of the decimal point when a floating point number is displayed in FIXED format. See FPtoString(), PrintFP() and FIXED().
Type: macro; Related QED-Forth function: LEFT.PLACES
Header file: numbers.h

int LinesPerDisplay(void)

Returns the number of lines in the LCD display. For character displays and for graphics displays being operated in "text mode", the result n equals the number of character lines (rows) in the display (the allowed values are 2, 4, 8 or 16 lines per display). For graphics displays being operated in "graphics mode", the result n equals the number of horizontal pixels on the display (which in turn is 8 times the number of

character lines on the display). The type of display and the display mode (text mode vs. graphics mode) are determined by the most recent execution of DisplayOptions() or InitDisplay() (which implements the configuration specified by IS.DISPLAY). The default value of n after executing the "special cleanup mode" is 4, corresponding to the default 4-line by 20-character display. The result returned by this routine is used by BufferPosition(), PutCursor(), UpdateDisplay(), and UpdateDisplayLine().
Type: _forth function; QED-Forth name: LINES/DISPLAY
Header file: intrface.h

MAILBOX

This typedef allocates a 32-bit mailbox in RAM which can be accessed by SEND(), TRY_TO_SEND() and RECEIVE(). The mailbox can hold any non-floating-point "message" up to 32 bits in size; for floating point messages, use FMAILBOX. Example of use:

```
MAILBOX  comm_flag;
SEND( 0x12345, &comm_flag);
```

Mailboxes are used in multitasked systems to share information between tasks and to synchronize tasks to one another. If the mailbox's contents equal zero, the mailbox is empty; it contains a message if its contents are non-zero. Before its first use, the mailbox must be initialized to zero. After initialization to zero, the only operators that should access the mailbox are SEND(), TRY_TO_SEND() and RECEIVE().
Type: typedef; Related QED-Forth function: MAILBOX:
Header file: mtasker.h

MANTISSA_PLACES

A user variable that holds the number of digits to be displayed in the mantissa when a floating point number is displayed in SCIENTIFIC format. See FPtoString(), PrintFP() and FLOATING().
Type: macro; Related QED-Forth function: MANTISSA.PLACES
Header file: numbers.h

MAX(num1, num2)

Returns the greater of num1 and num2; the inputs can be of any compatible type. This macro is defined as:

```
#define MAX(A,B)  (((A) > (B)) ? (A) : (B))
```

Type: macro
Header file: utility.h

void MicrosecDelay(uint numMicroseconds)

Enters a software timing loop for the specified number of microseconds. The function can time to within 2 microseconds resolution for 16 <= u <= 65535 microseconds. Note that the elapsed time will be increased by the duration of any interrupt routines that are serviced while MicrosecDelay() is running. Consequently, this routine does not guarantee accurate timing when the timesliced multitasker is running.
Type: _forth function; QED-Forth name: MICROSEC.DELAY
Header file: mtasker.h

MIN(num1, num2)

Returns the lesser of num1 and num2; the inputs can be of any compatible type. This macro is defined as:

```
#define MIN(A,B)  (((A) < (B)) ? (A) : (B))
```

Type: macro
Header file: utility.h

NEXT_TASK

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 16-bit TASKBASE address of the next task in the round-robin task list; in other words, NEXT_TASK contains the base address of the next task's user area. Before building all of the tasks in the top level routine of a multitasking application, the command

```
NEXT_TASK = TASKBASE;
```

must be executed to empty the round-robin task loop (by making NEXT_TASK point to its own TASKBASE address). This is detailed in the commentary accompanying the "Turnkey Application Program" in the QED "Getting Started" book. See TASK and TASKBASE.
Type: macro; Related QED-Forth function: NEXT.TASK
Header file: user.h

void NoAutostart(void)

Undoes the effect of the Autostart() and PriorityAutostart() commands and attempts to ensure that the standard QED-Forth interpreter will be entered after subsequent resets. This command can be executed interactively using QED-Forth syntax by typing from the terminal:

```
NO.AUTOSTART
```

Implementation detail: Erases the 0x1357 pattern at location 0xAE00 [put there by Autostart()] in EEPROM, and erases the 0x1357 pattern at location 0x047FFA [put there by PriorityAutostart()] in page 4 of paged memory. Note that the priority_autostart vector at 0x047FFA cannot be erased if the memory is write-protected when NoAutostart() is executed. NoAutostart() is invoked by the special cleanup mode.
Type: _forth function; QED-Forth name: NO.AUTOSTART
Header file: qedsys.h

void NoVitalIRQInit(void)

Writes a pattern into EEPROM so that subsequent cold restarts will not initialize the COP, clock monitor, illegal opcode, and OC2 interrupt vectors. This option is provided for programmers interested in installing their own interrupt service routines in any of these four vectors. Can be undone by InitVitalIRQsOnCold(). This function can be interactively executed using QED-Forth syntax by typing from the terminal:

```
NO.VITAL.IRQ.INIT
```

Implementation detail: Initializes location 0xAE1B in EEPROM to contain the pattern 0x13.
Type: _forth function; QED-Forth name: NO.VITAL.IRQ.INIT
Header file: qedsys.h

NO_SPACES

A user variable that contains a flag. If the flag is true (non-zero), leading and trailing spaces are not printed when a floating point number is displayed. If the flag is false (zero), the spaces are printed. See FPtoString() and PrintFP().
Type: macro; Related QED-Forth function: NO.SPACES
Header file: numbers.h

uint NUMCOLUMNS(FORTH_ARRAY* array_ptr)

A macro that returns the number of columns in the Forth array designated by array_ptr. An unpredictable result is returned if the array is not dimensioned.
Example of use:

```
FORTH_ARRAY  Myarray;       // define an array named Myarray
DIM(ulong, 3, 5, &Myarray); // 3 rows x 5 columns of unsigned longs
static uint  number_of_columns;
number_of_columns = NUMCOLUMNS(&Myarray);
```

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM().
Type: macro
Header file: array.h

uint NUMDIMENSIONS(FORTH_ARRAY* array_ptr)

A macro that returns the number of dimensions in the Forth array designated by array_ptr. The result is typically 2, because the DIM() macro specifies 2-dimensional arrays. An unpredictable result is returned if the array is not dimensioned.
Example of use:

```
FORTH_ARRAY  Myarray;       // define an array named Myarray
DIM(ulong, 3, 5, &Myarray); // 3 rows x 5 columns of unsigned longs
static uint  number_of_dimensions;
number_of_dimensions= NUMDIMENSIONS(&Myarray);
```

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM().
Type: macro
Header file: array.h

int NumInputChars(void)

Returns the number of characters in the input queue of the secondary serial port (serial2). In other words, returns the number of characters that have been received by the serial2 input interrupt service routine that have not yet been removed from the circular input buffer by Key2(). The default serial2 input buffer holds 80 characters and is located in the system RAM. The serial2 port is supported by a software UART using hardware pins PA3 (input) and PA4 (output).
Type: _forth function; QED-Forth name: #INPUT.CHARS
Header file: comm.h

int NumOutputChars(void)

Returns the number of characters in the output queue of the secondary serial port (serial2). In other words, returns the number of characters that have been placed in the output buffer by Emit2() that have not yet been removed from the circular output buffer by the serial2 output interrupt service routine. The default serial2 output buffer

holds 80 characters and is located in the system RAM. The serial2 port is supported by the software UART using hardware pins PA3 (input) and PA4 (output).
Type: _forth function; QED-Forth name: #OUTPUT.CHARS
Header file: comm.h

uint NUMROWS(FORTH_ARRAY* array_ptr)

A macro that returns the number of rows in the Forth array designated by array_ptr. An unpredictable result is returned if the array is not dimensioned.
Example of use:

```
FORTH_ARRAY  Myarray;       // define an array named Myarray
DIM(ulong, 3, 5, &Myarray);  // 3 rows x 5 columns of unsigned longs
static uint  number_of_rows;
number_of_rows = NUMROWS(&Myarray);
```

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM().
Type: macro
Header file: array.h

OC1_ID

A constant that returns the interrupt identity code for output compare 1. This interrupt can control the action of port bits PA3-PA7. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: OC1.ID
Header file: interupt.h

OC2_ID

A constant that returns the interrupt identity code for output compare 2. This interrupt can control the action of port bit PA6. Used as an argument for ATTACH(). Note that the OC2 interrupt is used by the timeslice multitasker; if you wish to use it for another purpose, make sure that you do not need any of the services of the timeslicer or elapsed-time clock. See also InitVitalIRQsOnCold() and InstallTimeslicer().
Type: constant; Related QED-Forth function: OC2.ID
Header file: interupt.h

OC3_ID

A constant that returns the interrupt identity code for output compare 3. This interrupt can control the action of port bit PA5. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: OC3.ID
Header file: interupt.h

OC4_ID

A constant that returns the interrupt identity code for output compare 4. This interrupt can control the action of port bit PA4. Used as an argument for ATTACH(). Note that OC4 and PA4 are used by the optional secondary serial port supported by the QED-Forth software UART; if you are not using the secondary serial port, you may use freely use OC4 and PA4.
Type: constant; Related QED-Forth function: OC4.ID
Header file: interupt.h

PAD

A macro that returns the 16-bit start address of the PAD scratchpad area in the active task's task-private area in common RAM. The 32 bytes below PAD are used for floating point and integer string/number conversion, and the 88 bytes above PAD are available as scratchpad memory for the programmer (Note that the QED-Forth routines ASK.NUMBER, ASK.FNUMBER, INPUT.STRING, and RECEIVE.HEX write text strings into the PAD buffer; however, this should not be a problem for C-programmed applications).
Type: macro; Related QED-Forth function: PAD
Header file: user.h

void PageToFlash(int source_page)

Transfers the 32 Kbyte contents of the specified RAM source page to the parallel page in flash. If the current memory map is the "download map", then valid source pages are 4, 5, or 6, (and, if a 512K RAM is installed, pages 0x10-17). Page 4 RAM is transferred to page 1 flash, page 5 RAM is transferred to page 2 flash, page 6 RAM is transferred to page 3 flash, and pages in the range 0x10-17 are transferred to parallel flash pages in the range 0x18-1F. If the current memory map is the "standard map", then valid source pages are 1, 2, or 3 (and, if a 512K RAM is installed, pages 0x18-1F). Page 1 RAM is transferred to page 4 flash, page 2 RAM is transferred to page 5 flash, page 3 RAM is transferred to page 6 flash, and pages in the range 0x18-1F are transferred to parallel flash pages in the range 0x10-17. An "invalid input parameter" error is issued if an invalid source page is specified. A "can't program flash" error is issued if the flash cannot be programmed. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user.
Type: _forth function; QED-Forth name: PAGE.TO.FLASH
Header file: flash.h

void PageToRAM(int source_page)

Transfers the 32 Kbyte contents of the specified flash source page to the parallel page in RAM. If the current memory map is the "download map", then valid source pages are 1, 2, or 3 (and, if a 512K RAM is installed, pages 0x18-1F). Page 1 flash is transferred to page 4 RAM, page 2 flash is transferred to page 5 RAM, page 3 flash is transferred to page 6 RAM, and pages in the range 0x18-1F are transferred to parallel RAM pages in the range 0x10-17. If the current memory map is the "standard map", then valid source pages are 4, 5, or 6 (and, if a 512K RAM is installed, pages 0x10-17). Page 4 flash is transferred to page 1 RAM, page 5 flash is transferred to page 2 RAM, page 6 flash is transferred to page 3 RAM, and pages in the range 0x10-17 are transferred to parallel RAM pages in the range 0x18-1F. An "invalid input parameter" error is issued if an invalid source page is specified.
Type: _forth function; QED-Forth name: PAGE.TO.RAM
Header file: flash.h

PAGE_LATCH

A constant that returns 0x8002 which is the address of the page latch whose contents indicate the current page. See THIS_PAGE which returns the contents of the PAGE_LATCH. In general, the PAGE_LATCH may be read but not written to by

application programs; only routines that are located in common memory (addresses above 0x8000) are allowed to write to the PAGE_LATCH.
Type: constant; QED-Forth name: (PAGE.LATCH)
Header file: types.h

PARITY

The PARITY variable is set by the programmer to specify the behavior of the secondary serial port (serial2) supported by QED-Forth's software UART. If PARITY is TRUE (non-zero), a parity bit is appended to each transmitted character and a parity bit is expected in each incoming character. The level of the transmitted parity bit is set by the system variable PARITY_OUT, and the value of the parity bit of the most recently received character is stored in the system variable PARITY_IN. PARITY is initialized to FALSE (zero) by InitSerial2() and UseSerial2() and at each reset or restart.
Type: macro; Related QED-Forth function: PARITY
Header file: comm.h

PARITY_IN

A system variable that equals the value of the parity bit of the character most recently received by the secondary serial port (serial2, supported by the software UART) if PARITY is true (non-zero). If the incoming parity bit was high, PARITY_IN equals 1; otherwise it equals 0. The contents are available to the programmer if parity checking of incoming data is required; the software UART does not check for correct parity. See PARITY.
Type: macro; Related QED-Forth function: PARITY.IN
Header file: comm.h

PARITY_OUT

A system variable that specifies the value of the parity bit of the character to be sent next by the secondary serial port (serial2, supported by the software UART) if PARITY is true. If the contents of PARITY are TRUE (non-zero) and the least significant byte of PARITY_OUT is non-zero, then the parity bit of the next outgoing character is set to one. If the contents of PARITY are TRUE and the least significant byte of PARITY_OUT is zero, then the parity bit of the next outgoing character is set to zero. The value of PARITY_OUT is not modified by the serial2 routines, and the application program must perform any required parity calculations. See PARITY.
Type: macro; Related QED-Forth function: PARITY.OUT
Header file: comm.h

void Pause(void)

Stacks the state of the current task and passes control to the next AWAKE task in the round-robin task list. You can embed calls to Pause() in any task when you wish to give other tasks a chance to run. Pause() may be used in multitasked systems whether or not the timeslicer is active. Pause() switches tasks in (27 + 3.25n) microseconds, where n is the number of ASLEEP tasks encountered in the round robin task list. Of this time, interrupts are disabled for (20 + 3.25n) microseconds.
Type: _forth function; QED-Forth name: PAUSE
Header file: mtasker.h

void PauseOnKey(void)

Suspends execution of the calling function when a character is received and, with the exceptions noted below, resumes execution of the calling function when a second character is received. Typically coded into a loop structure to allow control of execution during debugging, or to control a data dump. PauseOnKey() checks whether a character has been received. If no character has been received, it does nothing. If a character has been received and it is a carriage return, executes Abort() which clears the stacks and returns to the QED-Forth interpreter or the autostart routine (if installed). If the character received is a . (dot) executes QUIT which returns to the QED-Forth interpreter without clearing the data stack. If any other character is received, suspends execution until another character other than carriage return or . is received. This function effectively responds to XON/XOFF flow Control-Characters from a host terminal; a function running on the QED Board that dumps data and calls PauseOnKey() repeatedly will pause when the XOFF is received and resume when XON is received. PauseOnKey() does not know that the XON/XOFF characters are special; it just stops when receiving the first and resumes after the second.
Type: _forth function; QED-Forth name: PAUSE.ON.KEY
Header file: comm.h

PORTA

A macro that returns the contents of the 8 bit PortA register at address 0x8000 in the 68HC11. This port is available to the user and is associated with various counting and timing functions. To read the input pins of PORTA, simply use the PORTA macro in an expression or as the right hand side of an assignment statement. If no output compare interrupts are controlling the states of the PORTA pins, PORTA can be used as the left-hand side of a simple assignment statement to control the PORTA outputs. However, if output compare interrupts are controlling the states of the PORTA pins or if the secondary UART is in use, uninterruptable operators [such as the Forth functions SetBits() and ClearBIts(), or corresponding C functions defined with the _protect keyword] must be used to modify the available port bits; otherwise, unpredictable results can occur. Note that the software UART that implements the secondary serial port uses bits 3 and 4 of PORTA, so care must be taken not to alter the direction or state of these bits if the secondary serial port is in use. The DDRA register sets the data direction of the pins in PORTA.
Type: macro; Related QED-Forth function: PORTA
Header file: qedregs.h

PORTD

A macro that returns the contents of the 8 bit PORTD register at address 0x8008 in the 68HC11. This port implements the primary serial channel on bits 0 and 1, and the serial peripheral interface (SPI) on bits 2-5 which controls the onboard 12 bit A/D and 8 bit D/A. See InitSPI() and SPIOff().
Type: macro; Related QED-Forth function: PORTD
Header file: qedregs.h

PORTE

A macro that returns the contents of the 8 bit PORTE register at address 0x800A in the 68HC11. This port can either be used as an 8 channel 8 bit A/D convertor, or as an octal digital input port. See AD8On() and AD8Off().
Type: macro; Related QED-Forth function: PORTE

Header file: qedregs.h

void PrintFP(float ansi_fp_num)

Displays the specified floating point number using the format specified by the most recent execution of FIXED(), SCIENTIFIC(), or FLOATING(). FLOATING format is the default after a COLD restart. If the specified format is FIXED() and if the ansi_fp_num does not fit in the allowed number of LEFT_PLACES and RIGHT_PLACES, PrintFP() prints the string "won't fit". See the glossary entry for FPtoString() for a detailed description of the FIXED(), SCIENTIFIC() and FLOATING() formats.

Type: _forth function; QED-Forth name: PrintFP

Header file: numbers.h

void PriorityAutostart(void(*action)(), uint actionPage)

Compiles a 6-byte sequence at locations 0x7FFA-7FFF on page 4 so that upon subsequent restarts and Abort()s, the action routine having the specified execution address will be automatically called. This allows a finished application to be automatically entered upon power up and resets. This function is typically executed from the terminal using QED-Forth syntax by typing:

```
CFA.FOR MAIN PRIORITY.AUTOSTART
```

after a C program has been downloaded as described in the "Turnkey Application Program" example in the documentation.

In contrast to the EEPROM-based Autostart() function, the PriorityAutostart() vector is located in paged memory which is in flash memory in turnkeyed "production" boards. Thus PriorityAutostart() facilitates the autostarting of flash-based systems. PRIORITY.AUTOSTART is Flash smart; it writes to page 4 whether page 4 addresses RAM or Flash at the time. In the standard map PRIORITY.AUTOSTART writes directly to Flash in page 4. In the download memory map it also writes to page 4, now RAM. Subsequently page 4 can be copied to Flash and the Flash readdressed onto page 4 in the standard map.

Implementation detail: At location 0x7FFA on page 4, PriorityAutostart() writes the pattern 1357 followed by the four byte xcfa; make sure that page 4 is not write protected when executing PriorityAutostart(). Upon every reset, restart or runtime error, the Abort() function is called. Abort() checks the priority autostart vector first and executes the specified routine (if any). If no priority autostart routine is posted or if the specified routine terminates, Abort() then checks the EEPROM-based autostart vector and executes the specified routine (if any). If no autostart routine is posted or if the specified routine terminates, Abort() then invokes QUIT which is the QED-Forth interpreter. To undo the effects of this command and return to the default startup action, make sure that page 4 is un-write-protected RAM and call NoAutostart() which clears both the priority autostart and the EEPROM-based autostart vectors. To recover from the installation of a buggy priority autostart routine if page 4 is RAM, make sure that page 4 is not write-protected and invoke the special cleanup mode (consult the Manual). See Autostart() and NoAutostart().

Type: _forth function; QED-Forth name: PRIORITY.AUTOSTART

Header file: qedsys.h

PULSE_EDGE_ID

A constant that returns the interrupt identity code for the pulse accumulator input edge detector which is associated with port bit PA7. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: PULSE.EDGE.ID
Header file: interupt.h

PULSE_OVERFLOW_ID

A constant that returns the interrupt identity code for the pulse accumulator overflow detector which is associated with port bit PA7. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: PULSE.OVERFLOW.ID
Header file: interupt.h

void PutCursor(int line, int column)

Positions the LCD display cursor at the line number specified by line and the character number specified by column. The next character or graphical byte sent to the display by the CharToDisplay() routine will appear at the specified cursor position, and then the cursor position will automatically increment. The input parameters line and column are 0 based (that is, the top line on the display is line#0, and the left-most character on each line is column#0). PutCursor() clamps line to one less than LinesPerDisplay(), and clamps n2 to one less than CharsPerDisplayLine(). The line number follows the same rules explained in the description of BufferPosition(): for a graphics-style display the line number is interpreted differently depending on whether the display is being used in "text mode" or "graphics mode". In text mode, line corresponds to the character line number; in graphics mode, line corresponds to the pixel line number which is 8 times the character line number. Note that the cursor may not be visible, and is never visible in graphics mode; see DisplayOptions(). Also note that after the cursor reaches the end of a line it may skip to the start of a line elsewhere on the display. This routine intermittently disables interrupts for 28 cycles (7 µsec) per command byte to implement clock stretching.
Type: _forth function; QED-Forth name: PUT.CURSOR
Header file: intrface.h

int Random(void)

Generates and returns a pseudo-random 16bit integer. The result is also stored in the user variable RANDOM_SEED.
Type: _forth function; QED-Forth name: RANDOM
Header file: numbers.h

RANDOM_SEED

A user variable that equals the last 16-bit number generated by Random(). Storing a specific integer (a "seed") into RANDOM_SEED leads to the generation of a reproducible series of pseudo-random numbers by repeated calls to Random(). This may be useful for debugging functions that use random numbers. See Random().
Type: macro; Related QED-Forth function: RANDOM#
Header file: numbers.h

ulong ReadElapsedSeconds(void)

Returns the elapsed number of seconds since the timeslice clock was initialized to zero by InitElapsedTime(). See TIMESLICE_COUNT, StartTimeslicer(), and ChangeTaskerPeriod().

Type: _forth function; QED-Forth name: READ.ELAPSED.SECONDS
Header file: mtasker.h

void ReadWatch(void)

Reads the battery-operated real-time clock (if present), storing the time, day, and date in the 8-byte watch_results structure located at address 0xB3F8. The stack items, their allowed ranges, and the structure elements that hold the specified contents are as follows:

description	range	result is in structure element:
year	0 - 99	WATCH_YEAR
month	1 - 12	WATCH_MONTH
date	1 - 31	WATCH_DATE
day of week	1 - 7	WATCH_DAY
hour of day	0 - 23	WATCH_HOUR
minute after the hour	0 - 59	WATCH_MINUTE
seconds after the minute	0 - 59	WATCH_SECONDS
hundredths of seconds	0	WATCH_HUNDREDTH_SECONDS

Example of use:

```
int hour, day, date;    // static variables to hold current time
ReadWatch(); // get current time into watch_results structure
hour = WATCH_HOUR;          // assign from the structure into variables
day  = WATCH_DAY;
date = WATCH_DATE;
```

Due to a hardware limitation, the hundredths of second parameter always reads as 0; it is included in the structure to maintain backward compatibility with prior code. Once correctly set, the watch handles the differing numbers of days in each month, and correctly handles leap years. ReadWatch() uses the top 16 bytes of on-chip RAM at 0xB3F0-B3FF as a scratchpad buffer. See SetWatch().
Type: _forth function; QED-Forth name: READ.WATCH
Header file: watch.h

long RECEIVE(long* mailboxAddr)

If mailboxAddr is empty (ie., if it contains a 32-bit zero), executes Pause() until the mailbox contains a message. If mailboxAddr contains a message (that is, if it does not contain zero), returns the contents of mailboxAddr and stores a zero into mailboxAddr to indicate that the message has been received and that the mailbox is now empty. To receive and send floating point messages, use FRECEIVE() and FSEND(). RECEIVE() disables interrupts for 26 to 61 cycles (6.5 to 15.25 microseconds) to ensure that the state of the mailbox is correctly determined. See SEND() and MAILBOX.
Type: macro; Related QED-Forth function: RECEIVE
Header file: mtasker.h

long Receive(long* mailboxAddr, uint mailboxPage)

A subsidiary function called by the recommended macro RECEIVE(); see RECEIVE().
Type: _forth function; QED-Forth name: RECEIVE
Header file: mtasker.h

void RELEASE(xaddr* resourceAddr)

If the current task owns the resource variable referenced by resourceAddr (that is, if resourceAddr contains the current task's TASKBASE address), releases the resource by storing zero in xresource. Otherwise, does nothing; this prevents a task from RELEASEing a resource controlled by another task. Interrupts are not disabled and Pause() is not executed. See GET() and RESOURCE.
Type: macro; Related QED-Forth function: RELEASE
Header file: mtasker.h

void Release(xaddr* resourceAddr, uint resourcePage)

A subsidiary function called by the recommended macro RELEASE(); see RELEASE().
Type: _forth function; QED-Forth name: RELEASE
Header file: mtasker.h

RELEASE_AFTER_LINE

A constant which is the default value stored into the SERIAL_ACCESS user variable. If stored into (assigned to) SERIAL_ACCESS, prevents the low level I/O functions Key() Emit() and AskKey() from executing GET() or RELEASE() on the active serial resource variable. Rather, the task program installed by ACTIVATE() is responsible for executing GET() before each line is received and RELEASE() after each line is received. This SERIAL_ACCESS method is used by the QED-Forth interpreter to virtually eliminate the overhead required to GET and RELEASE during downloads, and allows the interpreter to run at a sustainable 19.2 Kbaud.
CAUTION: In multitasking systems using both serial ports Serial1 and Serial2, the application code should include the command

```
SERIAL_ACCESS = RELEASE_ALWAYS;
```

or SERIAL_ACCESS = RELEASE_NEVER;
before building the tasks. This prevents contention that can occur if the default RELEASE_AFTER_LINE option is installed in the SERIAL_ACCESS user variable. See SERIAL_ACCESS, RELEASE_NEVER, and RELEASE_ALWAYS.
Type: constant; Related QED-Forth function: RELEASE.AFTER.LINE
Header file: mtasker.h

RELEASE_ALWAYS

A constant. Returns a value that, when stored into the SERIAL_ACCESS user variable, causes the low level I/O functions Key() Emit() and AskKey() to always RELEASE the serial resource variable after each I/O operation. This is useful if the task that has control over the serial line wants to share access to the serial port. See SERIAL_ACCESS, RELEASE_NEVER, and RELEASE_AFTER_LINE.
CAUTION: You may find that storing RELEASE_ALWAYS into the QED-Forth task's SERIAL_ACCESS variable decreases the sustainable download baud rate. To assure the highest sustainable download baud rate, it is recommended that RELEASE_AFTER_LINE be stored in the QED-Forth task's SERIAL_ACCESS variable during program development.
CAUTION: In multitasking systems using both serial ports Serial1 and Serial2, the application code should include the command

```
SERIAL_ACCESS = RELEASE_ALWAYS;
```

or SERIAL_ACCESS = RELEASE_NEVER;

before building the tasks. This prevents contention that can occur if the default RELEASE_AFTER_LINE option is installed in the SERIAL_ACCESS user variable.
Type: constant; Related QED-Forth function: RELEASE.ALWAYS
Header file: mtasker.h

RELEASE_NEVER

A constant. Returns a value that, when stored into the SERIAL_ACCESS user variable, prevents the low level I/O functions Key() Emit() and AskKey() from executing the command RELEASE(SERIAL). This is useful if the task that has control over the serial line does not want to share access to the serial port. See SERIAL_ACCESS, RELEASE_ALWAYS, and RELEASE_AFTER_LINE.
CAUTION: You may find that storing RELEASE_NEVER into the QED-Forth task's SERIAL_ACCESS variable decreases the sustainable download baud rate. To assure the highest sustainable download baud rate, it is recommended that RELEASE_AFTER_LINE be stored in the QED-Forth task's SERIAL_ACCESS variable during program development.
CAUTION: In multitasking systems using both serial ports Serial1 and Serial2, the application code should include the command

```
SERIAL_ACCESS = RELEASE_ALWAYS;
```

or SERIAL_ACCESS = RELEASE_NEVER;
before building the tasks. This prevents contention that can occur if the default RELEASE_AFTER_LINE option is installed in the SERIAL_ACCESS user variable.
Type: constant; Related QED-Forth function: RELEASE.NEVER
Header file: mtasker.h

RESOURCE

This typedef allocates a 32-bit resource variable in the common RAM. Use as:

```
RESOURCE <name>;
```

where <name> is any name of your choosing. Resource variables are used in multitasked systems to control access to shared resources (for example, an A/D converter, serial port, block of memory, etc.) When the resource associated with <name> is available, <name> contains zero. When it is controlled by a task (and hence unavailable to other tasks), it contains the TASKBASE address of the controlling task; see the glossary entries for TASK and TASKBASE. Before its first use, the resource variable must be initialized to zero. After initialization to zero, the only operators that should access the resource variable are GET() TRY_TO_GET() and RELEASE(). The following resource variables are pre-defined in the mtasker.h file:

```
AD8_RESOURCE        SPI_RESOURCE
SERIALSERIAL1_RESOURCE SERIAL2_RESOURCE
```

See their glossary entries and consult the Multitasking chapter in the Software Manual for further descriptions and examples of use.
Type: typedef; Related QED-Forth function: RESOURCE.VARIABLE:
Header file: mtasker.h

RIGHT_PLACES

A user variable that holds the number of digits to be displayed to the right of the decimal point when a floating point number is printed in FIXED format. See FPtoString(), PrintFP() and FIXED().
Type: macro; Related QED-Forth function: LEFT.PLACES

Header file: numbers.h

ulong Room(void)

Returns the number of bytes available in the HEAP. NOTE: because there is some overhead (up to 12 bytes) associated with adding an item to the heap, you may not be able to dimension a new heap item that requires the exact number of bytes returned by Room().
Type: _forth function; QED-Forth name: ROOM
Header file: heap.h

void RS485Receive(void)

Clears bit 4 in PPC (of the PIA) to the logic 0 state. If upper PPC has been configured as an output port, this places the RS485 transceiver in the receive mode. NOTE: Make sure that the onboard RS485/RS232 jumper is properly set before attempting to use the RS485 interface. See InitRS485() and RS485Transmit().
Type: _forth function; QED-Forth name: RS485.RECEIVE
Header file: comm.h

void RS485Transmit(void)

Sets bit 4 in PPC (of the PIA) to the logic 1 state. If upper PPC has been configured as an output port, this places the RS485 transceiver in the transmit mode. NOTE: Make sure that the onboard RS485/RS232 jumper is properly set before attempting to use the RS485 interface. See InitRS485() and RS485Receive().
Type: _forth function; QED-Forth name: RS485.TRANSMIT
Header file: comm.h

RTI_ID

A constant that returns the interrupt identity code for the real time interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: RTI.ID
Header file: interupt.h

int ScanKeypad(void)

Scans keypad or touchscreen having up to 8 rows and 5 columns. If a key is being depressed, PAUSEs and waits until the key is released, then returns the key number under a true flag. If no key is depressed, returns a false flag. Consult the Keypad() glossary entry for a detailed description of keypad orientation. Disables interrupts for 12 µseconds each time a row is scanned. See ScanKeypress() and Keypad().
Type: C function; Related QED-Forth function: ?KEYPAD
Header file: intrface.h

int ScanKeypress(void)

Scans keypad or touchscreen having up to 8 rows and 5 columns. If a key is being depressed, returns the key number under a true flag; unlike ?KEYPAD, ?KEYPRESS does not wait for the key to be released. If no key is depressed, returns a false flag. Consult the Keypad() glossary entry for a detailed description of keypad orientation. Disables interrupts for 12 microseconds each time a row is scanned. See ScanKeypad() and Keypad().
Type: C function; Related QED-Forth function: ?KEYPRESS

Header file: intrface.h

void SCIENTIFIC(void)

Sets the default printing format used by PrintFP() and FPtoString() to scientific. For more details, see the glossary entry for FPtoString().
Type: macro; Related QED-Forth function: SCIENTIFIC
Header file: numbers.h

SCI_ID

A constant that returns the interrupt identity code for the asynchronous serial communications interface. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: RTI.ID
Header file: interupt.h

void SEND(long message, long * mailboxAddr)

Executes Pause() until the mailbox with extended address mailboxAddr is empty (contains zero) and then stores the 32-bit message in mailboxAddr. The message can be any 32-bit quantity except zero; use FSEND() to send a floating point value as a message. For example, the message can be an array address returned by ARRAYMEMBER() that points to a block of data. To ensure that the state of the mailbox is correctly determined, SEND() disables interrupts for 16 to 50 cycles (4 to 12.5 microseconds). See TRY_TO_SEND(), RECEIVE() and MAILBOX.
Type: macro; Related QED-Forth function: SEND
Header file: mtasker.h

void Send(long message, long * mailboxAddr, uint mailboxPage)

A subsidiary function called by the recommended macro SEND(); see SEND().
Type: _forth function; QED-Forth name: SEND
Header file: mtasker.h

SERIAL

A constant that returns the address of the resource variable associated with the primary serial I/O port. A synonym for SERIAL1_RESOURCE. See SERIAL1_RESOURCE.
Type: macro constant; Related QED-Forth function: SERIAL
Header file: mtasker.h

void Serial1AtStartup(void)

Initializes a flag in EEPROM which causes the initialization software to install the primary serial port (serial1) as the default serial port used by the QED-Forth interpreter after each reset or restart. The serial1 port is supported by the 68HC11's on-chip hardware UART.
Implementation detail: Sets the contents of address 0xAE1D in EEPROM to 0xFF. Upon each reset or restart, the QED-Forth startup routine checks this byte, and contents of 0xFF cause the UseSerial1() routine to be executed. See UseSerial1() and Serial2AtStartup().
Type: _forth function; QED-Forth name: SERIAL1.AT.STARTUP
Header file: comm.h

SERIAL1_RESOURCE

A constant that returns the address of the resource variable associated with the primary serial I/O port. This resource variable mediates access to the primary serial port (serial1) associated with the 68HC11's on-chip hardware UART. SERIAL1_RESOURCE should be accessed only by the functions GET(), TRY_TO_GET() and RELEASE(). Initialized to zero by UseSerial1() and UseSerial2() and at each reset or restart. See RESOURCE.
Type: macro constant; Related QED-Forth function: SERIAL1.RESOURCE
Header file: mtasker.h

void Serial2AtStartup(int baud)

Initializes a flag in EEPROM which causes the initialization software to install the secondary serial port (serial2) at the specified baud rate as the default serial port used by the QED-Forth interpreter after each reset or restart. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). The specified baud rate must a power of 2 times 75 baud, up to a maximum of 9600 baud. Thus the allowed baud rates for this routine are 75, 150, 300, 600, 1200, 2400, 4800, and 9600 baud. The effect of this routine is canceled by executing Serial1AtStartup(). Note that the serial2 port can support many more baud rates, but the options have been limited to facilitate setting a reasonable startup baud rate based on a simple implementation as described below. Note also that the maximum baud rate that can be sustained by the serial2 port is 4800 baud; see the glossary entry for Baud2().
Implementation detail: Sets the contents of address 0xAE1D in EEPROM equal to baud/75. Upon each reset or restart, the QED-Forth startup routine checks this byte, and contents equal to an exact power of two cause the UseSerial2() routine to be executed before control is passed to the interpreter or to an autostart routine. Note that UseSerial2() globally enables interrupts during the startup process. If you wish to use the secondary serial port while avoiding this side-effect and maintaining control over the global enabling of interrupts, don't execute Serial2AtStartup(). Rather, have your autostart routine explicitly call UseSerial2() after ensuring that all interrupt service routines are properly initialized.
Type: _forth function; QED-Forth name: SERIAL2.AT.STARTUP
Header file: comm.h

SERIAL2_RESOURCE

A constant that returns the address of the resource variable that mediates access to the secondary serial port (serial2). The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). SERIAL2_RESOURCE should be accessed only by the functions GET(), TRY_TO_GET() and RELEASE(). Initialized to zero by UseSerial1() and UseSerial2() and at each reset or restart. See RESOURCE.
Type: macro constant; Related QED-Forth function: SERIAL2.RESOURCE
Header file: mtasker.h

SERIAL_ACCESS

SERIAL_ACCESS is a user variable (member of the currently active TASK.USER_AREA structure) that contains a flag that controls when a task GETs and RELEASEs access to the serial resource. If more than one task needs access to the serial I/O port, this flag can help specify which task (if any) gets priority use. If

SERIAL_ACCESS contains the value RELEASE_ALWAYS, then each I/O operation by Key() Emit() or AskKey() will GET() the active serial resource before each I/O operation and RELEASE() the active serial resource after each character I/O operation is complete. If SERIAL_ACCESS contains the value RELEASE_NEVER, then I/O operations called by the task always GET() but never RELEASE() the serial resource variable. If SERIAL_ACCESS contains the value RELEASE_AFTER_LINE, then Key() Emit() and AskKey() never GET() or RELEASE() the serial resource. Rather, the QED-Forth interpreter GET()s the serial resource before each line is received and RELEASE()s the serial resource after each line is interpreted. This virtually eliminates the overhead required to GET() and RELEASE() during downloads, and allows the interpreter to run at a sustainable 19200 baud. The default value stored in SERIAL_ACCESS after a COLD restart is RELEASE_AFTER_LINE.

CAUTION: In multitasking systems using both serial ports SERIAL1 and SERIAL2, the application code should include the command

```
SERIAL_ACCESS = RELEASE_ALWAYS;
```

or `SERIAL_ACCESS = RELEASE_NEVER;`

before building the tasks. This prevents contention that can occur if the default RELEASE_AFTER_LINE option is installed in the SERIAL_ACCESS user variable. See SERIAL1_RESOURCE, SERIAL2_RESOURCE, GET(), RELEASE(), Key(), Emit() and AskKey().

Type: macro; Related QED-Forth function: SERIAL.ACCESS

Header file: user.h

void SetBits(uchar mask, xaddr address)

For each bit of mask that is set, sets the corresponding bit of the 8 bit value at address. Disables interrupts for ten cycles (2.5 microseconds) to ensure an uninterrupted read/modify/write operation. See also ClearBits(), ToggleBits(), ChangeBits() and PIASetBits().

Type: _forth function; QED-Forth name: SET.BITS

Header file: xmem.h

void SetBootVector (uint fn_addr, uint fn_page)

Compiles a 6-byte sequence at locations 0x7FFA-0x7FFF on page 0x0C so that upon subsequent restarts and ABORTs, the function having the xcfa (execution address) specified by fn_addr and fn_page will be executed BEFORE any other autostart routines are executed. The execution order at startup is: boot_vector, then priority_autostart, then autostart. Note that the “page C write protect” jumper must be removed for this function to be effective. The boot vector is most useful for extending the kernel in a "bullet-proof" way that cannot be overwritten unless the page C write protect jumper is removed. For example, suppose that you want to allow fail-safe field firmware upgrades using Compact Flash (CF) cards via Mosaic's CF Wildcard. This can be accomplished by removing the page C hardware write protect jumper, loading the CF Wildcard kernel extension on page 0x0C, and compiling a startup function on page C that checks for the presence of an "AUTOEXEC.QED" file that will be automatically executed (loaded) if present. Using SetBootVector, the startup function can be declared as a boot vector, and then the page C write protect jumper can be installed. The boot vector will be able check for the presence of a firmware upgrade file, and the hardware write protection of page C prevents the erasure of the boot vector or its code. To remove the boot vector, take off the page C write protect jumper and call

ClearBootVector (CLEAR.BOOT.VECTOR in Forth), or perform a "factory cleanup". We recommend that this function be invoked interactively from the QED-Forth prompt. Assume that a function called Page_C_Startup has been defined. Forth programmers can just execute:

```
CFA.FOR Page_C_Startup SET.BOOT.VECTOR
```

C programmers can use the *.map file generated by the C compiler to look up the compilation address and page of the Page_C_Startup function, or the function can be defined using the _Q prefix as:

```
_Q void Page_C_Startup( void ) { function body goes here }
```

Then from the QED-Forth prompt, type

```
CFA.FOR Page_C_Startup SET.BOOT.VECTOR
```

Make sure that the page containing the debug headers is included in your final runtime system.

Type: _forth function; QED-Forth name: SET.BOOT.VECTOR
Header file: V4_4Update.h; V4_4Update.c must be #included in 1 file only

void SetWatch(hundredth_seconds, seconds, minute, hour, day, date, month, year)

Sets the battery-operated real-time clock (if present) to the time, day, and date specified by the input parameters. The input parameters and their allowed ranges are:

description	range
year	0 - 99
month	1 - 12
date	1 - 31
day	1 - 7
hour	0 - 23
minute after the hour	0 - 59
seconds after the minute	0 - 59
hundredth_seconds	0 - 99

Due to a hardware limitation, the hundredths of second parameter is ignored; it is included in the parameter list to maintain backward compatibility with prior code. Once correctly set, the watch handles the differing numbers of days in each month, and correctly handles leap years. SetWatch() uses the top 16 bytes of on-chip RAM at B3F0-B3FF as a scratchpad buffer. See ReadWatch().

Type: _forth function; QED-Forth name: SET.WATCH
Header file: watch.h

uint SIZEOFMEMBER(FORTH_ARRAY* array_ptr)

A macro that returns the number of bytes per element in the Forth array designated by array_ptr. An unpredictable result is returned if the array is not dimensioned.

Example of use:

```
FORTH_ARRAY Myarray;      // define an array named Myarray
DIM(ulong, 3, 5, &Myarray); // 3 rows x 5 columns of unsigned longs
static uint size_of_each_member;
size_of_each_member = SIZEOFMEMBER(&Myarray);
```

See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also DIM().

Type: macro; Related QED-Forth name: BYTES/ELEMENT
Header file: array.h

int SpeedToDuty(int steps_per_second, int ticks_per_second)

Returns an integer representation of a duty cycle which specifies the step rate of the stepper motor. The first input parameter is the integer number of steps per second if full stepping, or the number of halfsteps per second if half stepping. The second input parameter is the integer number of clock ticks per second; the default is 1000 ticks per second. The integer output parameter can be interpreted as a fraction with the radix point to the left of the most significant bit. A 100% duty cycle is represented by hex 0xFFFF, and this tells the STEP.MANAGER to output a new step pattern on every tick of the interrupt clock (e.g., once per millisecond, corresponding to 1000 (half) steps per second). A duty cycle of hex 0x8000 means a new step pattern is written to the motor port every other clock tick; a duty cycle of hex 0x0100 dictates one step every 256 clock ticks; and a duty cycle of 0000 means corresponds to a stopped state with no step pattern updates. See the high level source file steppers.c in the Demos_and_Drivers directory of the distribution.
Type: _forth function; QED-Forth name: SPEED.TO.DUTY
Header file: stepper.h

void SPIOff(void)

Disables the serial peripheral interface (SPI) by clearing the SPI enable (SPE) bit in the SPI control register (SPCR). Note that the SPI communicates with the onboard 12 bit A/D and 8 bit DAC; if a custom QED Board has been ordered without these devices installed, then after execution of this routine, PORTD pins PD2-PD5 may be used as standard digital I/O subject to the data direction specified in the DDRD register. See InitSPI().
Type: _forth function; QED-Forth name: SPI.OFF
Header file: analog.h

SPI_ID

A constant that returns the interrupt identity code for the synchronous serial peripheral interface (SPI). Used as an argument for ATTACH(). Note that the SPI communicates with the onboard 12 bit A/D and 8 bit DAC if they are installed. See InitSPI() and SPIOff().
Type: constant; Related QED-Forth function: SPI.ID
Header file: interupt.h

SPI_RESOURCE

A constant that returns the address of the resource variable associated with the serial peripheral interface (SPI) which is used for data transfer to and from the 12 bit analog to digital convertor (AD12) and 8 bit digital to analog convertor (DAC). Should be accessed only by the functions GET(), TRY_TO_GET() and RELEASE(). Initialized to zero by InitSPI() and InitAD12andDAC() and at each reset or restart. SPI_RESOURCE is automatically invoked by many of the AD12 and DAC device driver routines. See RESOURCE.
Type: macro constant; Related QED-Forth function: SPI.RESOURCE
Header file: mtasker.h

void StandardMap(void)

Sets a flag in EEPROM and changes the state of a hardware latch to put the standard memory map into effect. After execution of this routine, and upon each subsequent

reset or restart, hex pages 4, 5, 6, and 0x10-17 are addressed in flash memory, and pages 1, 2, 3, and 0x18-1F are addressed in RAM. After code is downloaded to RAM and transferred to flash using the PAGE.TO.FLASH function, establishing the standard map allows code resident on pages 4, 5 and 6 (and pages 0x10-17) to be executed. To establish the download memory map, see the glossary entry for DOWNLOAD.MAP. Note that the standard map is active after a "factory cleanup" operation.
Type: _forth function; QED-Forth name: STANDARD.MAP
Header file: flash.h

void StandardReset(void)

Undoes the effect of the ColdOnReset() command so that subsequent resets will result in the standard warm-or-cold startup sequence. Note that this function can be executed interactively using QED-Forth syntax by typing from the terminal:

```
STANDARD.RESET
```

Implementation detail: sets the flag at location 0xAEC in EEPROM to 0xFF.
Type: _forth function; QED-Forth name: STANDARD.RESET
Header file: qedsys.h

void StartTimeslicer(void)

Starts the timeslice clock based on the Output Compare 2 (OC2) interrupt and begins timeslice multitasking. Initializes the OC2 interrupt vector (if it wasn't already initialized) so that the multitasking executive/elapsed-time clock routine services the interrupt. Enables the OC2 interrupt mask and globally enables interrupts by clearing the I bit in the condition code register of each built task.
Notes:
1. The default timeslice clock period of 5 msec can be changed with the command ChangeTaskerPeriod().
2. StartTimeslicer() does not initialize the value in TIMESLICE_COUNT; execute InitElapsedTime() if you wish to initialize the clock count to zero.
3. After a restart, the system is configured so that timeslice multitasking can begin at any time; if no other tasks have been built, the FORTH_TASK (which is also the task that calls main) is the only task in the task loop.
4. The timeslicer's interrupt service routine disables interrupts for the duration of a task switch which requires 25 microseconds plus 3.25 microseconds for each ASLEEP task encountered in the task list.
Type: _forth function; QED-Forth name: START.TIMESLICER
Header file: mtasker.h

STATUS

STATUS is the first element in the currently active task's TASK.USER_AREA structure; its contents specify whether the task is ASLEEP or AWAKE. The following example shows how to access the status address of another task that has been defined using the TASK directive:

```
TASK AnyTask;          // name and allocate the new task
AnyTask.USER_AREA.user_status = ASLEEP;  // put the task asleep
```

See the user.h file for a detailed description of all of the user variables, and consult the glossary entries for the constants ASLEEP and AWAKE.
Type: macro; QED-Forth name: STATUS
Header file: user.h

void StepManager(void)

Expects the base address of the STATUS.ARRAY in the Y register. Based on the information in the status array and the ramp array (defined in high level source file steppers.c in the Demos_and_Drivers directory of the distribution), for each enabled motor StepManager writes the appropriate step pattern at the specified duty cycle to the motor port to attain the speed specified in the motor's ramp array. This function is meant to be called from a periodic interrupt service routine typically associated with an output compare (OC) interrupt; the default time base is once per millisecond generated by the OC3 interrupt, with a resulting maximum speed of 1000 full- or half-steps per second. This assembly coded routine executes in approximately 120 µs per enabled stepper motor. Thus running four stepper motors at a maximum speed of 1000 full- or half-steps per second requires approximately half of the 68HC11's available time (480 µs interrupt service time every 1000 µs). See the high level source file steppers.c in the Demos_and_Drivers directory of the distribution. CAUTION: The presence of other interrupt service routines can affect the timing of the step manager, and may affect the smoothness of the stepper motor operation.
Type: _forth function; QED-Forth name: STEP.MANAGER
Header file: stepper.h

void StopTimeslicer(void)

Stops the multitasker's timeslice clock by disabling the local Output Compare 2 (OC2) timer interrupt mask. Cooperative task switching involving Pause() is not affected. See StartTimeslicer(). Note that this command also stops the elapsed-time clock; see TIMESLICE_COUNT and ReadElapsedSeconds().
Type: _forth function; QED-Forth name: STOP.TIMESLICER
Header file: mtasker.h

void StoreChar(char value, xaddr address)

Stores the 8-bit value at the specified extended address. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().
Type: _forth function; QED-Forth name: C!
Header file: xmem.h

void StoreEEChar(char value, int * addr)

Stores the specified 8-bit value at the specified addr in EEPROM. If addr already contains the specified contents it is not re-programmed; this helps lengthen the lifetime of the EEPROM. Requires 20 msec per programmed byte. Disables interrupts during the programming of each byte. Caution: the prolonged disabling of interrupts by (EEC!) can adversely affect real-time servicing of interrupts including those associated with the secondary serial line.
Example of use:

```
#pragma  option data=.eeprom // relocate data section to .eeprom
static uchar numsamples;     // define a "variable" in eeprom
#pragma  option data=.data   // restore data area
StoreEEChar( 8, &numsamples);      // init numsamples to equal 8
```

Note that this function can be called interactively to initialize EEPROM from the terminal using QED-Forth syntax. For example, assuming that the C program containing the

definition of numsamples has been downloaded to the QED Board, you can interactively set numsamples to equal 8 by typing from your terminal:

```
8 numsamples (EEC!)
```

Type: _forth function; QED-Forth name: (EEC!)
Header file: xmem.h

void StoreEEFloat(float value, int * addr)

Stores the specified 32-bit floating point value at the specified addr in EEPROM. The most significant word of val is stored at addr and the least significant word is stored at addr+2. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. Requires 20 msec per programmed byte. Disables interrupts during the programming of each byte. Caution: the prolonged disabling of interrupts by (EE2!) can adversely affect real-time servicing of interrupts including those associated with the secondary serial line.
Example of use:

```
#pragma option data=.eeprom // relocate data section to .eeprom
static float coefficient; // define a "variable" in eeprom
#pragma option data=.data // restore data area
StoreEEFloat( 123.4, &coefficient); // init coefficient to equal 123.4
```

Note that this function can be called interactively to initialize EEPROM from the terminal using QED-Forth syntax. For example, assuming that the C program containing the definition of coefficient has been downloaded to the QED Board, you can interactively set coefficient to equal 123.4 by typing from your terminal:

```
123.4 FP_QtoC coefficient (EEF!)
```

We inserted the FP_QtoC command to convert from QED-Forth floating point representation to the ANSI-C representation.
Type: _forth function; QED-Forth name: (EEF!)
Header file: xmem.h

void StoreEEInt(int value, int * addr)

Stores the specified 16-bit value at the specified addr in EEPROM. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. Requires 20 msec per programmed byte, independent of the clock speed. Disables interrupts during the programming of each byte. Caution: the prolonged disabling of interrupts by StoreEEInt() can adversely affect real-time servicing of interrupts including those associated with the secondary serial line.
Example of use:

```
#pragma option data=.eeprom // relocate data section to .eeprom
static int numsamples; // define a "variable" in eeprom
#pragma option data=.data // restore data area
StoreEEInt( 1234, &numsamples); // init numsamples to equal 1234
```

Note that this function can be called interactively to initialize EEPROM from the terminal using QED-Forth syntax. For example, assuming that the C program containing the definition of numsamples has been downloaded to the QED Board, you can interactively set numsamples to equal 1234 by typing from your terminal:

```
1234 numsamples (EE!)
```

Type: _forth function; QED-Forth name: (EE!)
Header file: xmem.h

void StoreEELong(long value, int * addr)

Stores 32-bit value at the specified addr in EEPROM. The most significant word of val is stored at addr and the least significant word is stored at addr+2. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. Requires 20 msec per programmed byte. Disables interrupts during the programming of each byte. Caution: the prolonged disabling of interrupts by (EE2!) can adversely affect real-time servicing of interrupts including those associated with the secondary serial line.

Example of use:

```
#pragma option data=.eeprom // relocate data section to .eeprom
static int numsamples; // define a "variable" in eeprom
#pragma option data=.data     // restore data area
StoreEELong( 1234, &numsamples);  // init numsamples to equal 1234
```

Note that this function can be called interactively to initialize EEPROM from the terminal using QED-Forth syntax. For example, assuming that the C program containing the definition of numsamples has been downloaded to the QED Board, you can interactively set numsamples to equal 1234 by typing from your terminal:

```
DIN 1234 numsamples (EE2!)
```

where the DIN command tells QED-Forth to interpret the following number as a 32-bit long.

Type: _forth function; QED-Forth name: (EE2!)

Header file: xmem.h

void StoreFloat(float value, xaddr address)

Stores a 32-bit floating point value at the specified extended address. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().

Type: _forth function; QED-Forth name: F!

Header file: xmem.h

void StoreFloatProtected(float value, xaddr address)

Stores the floating point value at the specified extended address. Disables interrupts during the store to ensure that an interrupting routine or task will read valid data. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). Disables interrupts for 28 cycles (7 microseconds) unless the specified 4 bytes straddle a page boundary, in which case interrupts are disabled for approximately 260 cycles. The most significant word of val is stored at address and the least significant is stored at address+2. Note that in paged memory, the address immediately following 0x7FFF is address 0000 on the following page. See also FetchFloatProtected().

Type: _forth function; QED-Forth name: |F!|

Header file: xmem.h

void StoreInt(int value, xaddr address)

Stores a 16-bit value at the specified extended address. The high order byte is stored at address and the low order byte at address+1. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().

Type: _forth function; QED-Forth name: !

Header file: xmem.h

void StoreLong(long value, xaddr address)

Stores a 32-bit value at the specified extended address. The most significant word of val is stored at address and the least significant is stored at address+2. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER().
Type: _forth function; QED-Forth name: 2!
Header file: xmem.h

void StoreLongProtected(long value, xaddr address)

Stores the 32-bit value at the specified extended address. Disables interrupts during the store to ensure that an interrupting routine or task will read valid data. This function is useful for storing data in arrays located in paged memory, where the extended address is returned by ARRAYMEMBER(). Disables interrupts for 28 cycles (7 microseconds) unless the specified 4 bytes straddle a page boundary, in which case interrupts are disabled for approximately 260 cycles. The most significant word of val is stored at address and the least significant is stored at address+2. Note that in paged memory, the address immediately following 0x7FFF is address 0000 on the following page. For floating point values, use StoreFloatProtected().
Type: _forth function; QED-Forth name: |2!|
Header file: xmem.h

void StringMove(xaddr countedStrAddr, xaddr destination, long numBytes)

Moves the contents of the counted string specified by countedStrAddr to the specified 32-bit destination address. Does not move the count byte. The number of characters moved is clamped to a maximum of numBytes bytes. To use this function to move a null-terminated C-style string into paged memory, first convert the string to a Forth-style counted string using CountedString(), and then move it with this function. As explained in the glossary entry for CountedString(), the string should contain less than 86 characters.
Example of use: The following code copies a null-terminated "C-style" string to a buffer in paged memory:

```
#define DESTINATION = ( (xaddr) 0x071000)    // buffer in page 7 RAM
#define MAX_STRING_CHARS  85
char*  source_string = "This is the string we will move";
xaddr  counted_string = CountedString(source_string, THIS_PAGE);
StringMove(counted_string, DESTINATION, MAX_STRING_CHARS);
```

See also CountedString().
Type: _forth function; QED-Forth name: $MOVE
Header file: xmem.h

void StringToDisplay(char* string, uint stringPage, int linenum, int column)

For most programs, the macro form named STRING_TO_DISPLAY() is recommended; see its glossary entry. The function StringToDisplay() should be used when the specified string resides on a different memory page than the routine that invokes StringToDisplay(), for example in an application that is compiled on multiple pages on the QED Board. Pass the function the correct stringPage, and set the upper

byte of the stringPage = 0xFF to signal that the string is a null-terminated "C-style" string as opposed to a counted "Forth-style" string.
Type: _forth function; QED-Forth name: $>DISPLAY
Header file: intrface.h

void STRING_TO_DISPLAY(char* string, int linenum, int column)
A macro that calls the _forth function:
void StringToDisplay(char* string, uint stringPage, int linenum, int column)
The macro supplies the parameter stringPage = the current page, and sets the upper byte of the page = 0xFF to signal the routine that the string is a null-terminated "C-style" string. The routine moves the contents of the counted string specified by string to the location in DisplayBuffer() starting at the specified character number 'column' on the specified line number 'linenum'. Confines the string to the specified line in Display Buffer by clamping the number of characters moved to a maximum equal to the number of character positions remaining after the specified position on the specified line. The line number 'linenum' should be less than the value returned by LinesPerDisplay(), and the character number n2 should be less than the value returned by CharsPerDisplayLine(). Does not modify the contents of the LCD display; this will occur upon the next execution of UpdateDisplayLine() or UpdateDisplay(). If a Toshiba graphics display has been declared by IsDisplay(), subtracts 0x20 from each ascii character in the string to accommodate the encoding of the Toshiba graphics controller's character ROM.
NOTE: While this macro always works properly if your application resides on a single page of memory on the QED Board, this macro cannot be used if the specified string resides on a memory page that is different from the page of the calling routine. If the string resides on a different page from the calling function, use the function StringToDisplay() and pass it the proper string page, remembering to set the upper byte of the page = 0xFF to signal that the string is null-terminated.
Type: macro; Calls function: StringToDisplay; QED-Forth name: $>DISPLAY
Header file: intrface.h

void SWAPARRAYS(FORTH_ARRAY* array1_ptr, FORTH_ARRAY* array2_ptr)
Interchanges the contents of the parameter fields of the two specified arrays and leaves the heap undisturbed, thus rapidly swapping the two arrays. See the FORTH_ARRAY glossary entry for a description of how to define an array and its corresponding array_ptr. See also FORTH_ARRAY, DIM(), ARRAYFETCH() and ARRAYSTORE().
Type: macro; Related QED-Forth function: SWAP.ARRAYS
Header file: array.h

void SwapArrays(FORTH_ARRAY* array1_ptr, uint pfa_page, FORTH_ARRAY* array2_ptr, uint pfa_page,)
A subsidiary function called by the recommended macro SWAPARRAYS(); see SWAPARRAYS().
Type: _forth function; QED-Forth name: SWAP.ARRAYS
Header file: array.h

SWI_ID
A constant that returns the interrupt identity code for the software interrupt (SWI). Used as an argument for ATTACH().

Type: constant; Related QED-Forth function: SWI.ID
Header file: interupt.h

void SysAbort(void)

The default abort routine called by Abort() if the flag in CustomAbort is false. Clears the data and return stacks, and sets the page to the default page (0). If an autostart vector has been installed [see Autostart() and PriorityAutostart()], SysAbort() executes the specified routine; otherwise it executes QUIT which sets the execution mode and enters the QED-Forth monitor. If the stack pointers do not point to common RAM, a COLD restart is initiated.
Type: _forth function; QED-Forth name: (ABORT)
Header file: qedsys.h

TASK

A structure typedef that names and allocated a 1 Kbyte TASK structure in common RAM. In other words, TASK is used to name and locate a new task. To declare a new task named AnyTask, use the statement:

```
TASK AnyTask;
```

See the glossary entries for BUILD_C_TASK() and ACTIVATE() for a discussion of how to build and activate the new task; see also FORTH_TASK.
Type: typedef; Related QED-Forth function: TASK:
Header file: user.h

TASKBASE

A pointer to the current task's TASK structure which holds the task's USER_AREA structure and its task-private stacks and buffers. The USER_AREA is a structure defined in the user.h file that contains task-private operating system pointers and variables. Each task in a multitasking application has a unique 1 Kbyte TASK area named and allocated by the TASK statement, and the multitasker periodically updates the contents of UP (the User Pointer) to point to the current user area. TASKBASE is a macro defined as:

```
((TASK *) *UP)
```

In other words, TASKBASE returns the contents of UP, and TASKBASE is cast as a pointer to the TASK structure. The base address returned by TASKBASE is also referred to as the "task identifier" or "task id"; it is the address in common memory used to identify a particular task. It is passed as a parameter to BUILD_C_TASK() and ACTIVATE().
NOTE: before building the tasks in a multitasked application, the current value returned by TASKBASE should be stored into the NEXT_TASK user variable to effectively empty the task loop and kill any extraneous tasks that may be running. This can be accomplished by executing the statement:

```
NEXT_TASK = TASKBASE;
```

before invoking BUILD_C_TASK(). An example of this technique is presented in the "Turnkeyed Application Program" in the QED "Getting Started" book.
Type: macro; related QED-Forth function: STATUS
Header file: user.h

THIS_PAGE

A macro that returns the contents of the PAGE_LATCH which indicates the current page. THIS_PAGE is equivalent to:

```
* PAGE_LATCH
```

In general, the PAGE_LATCH may be read but not written to by application programs; only routines that are located in common memory (addresses above 0x8000) are allowed to write to the PAGE_LATCH.
Type: macro; Related QED-Forth function: THIS.PAGE
Header file: types.h

TIB

A macro that returns the 16-bit start address of the Terminal Input Buffer. The default size of the terminal input buffer is 96 bytes; it is used by the QED-Forth interpreter. If Forth is not running in a given task and if high-level Forth serial I/O routines such as EXPECT, QUERY and INTERPRET are not being executed, a C application may use the TIB as a task-private buffer.
Type: macro; Related QED-Forth function: TIB
Header file: user.h

TIMER_OVERFLOW_ID

Returns the interrupt identity code for the free-running timer overflow interrupt. Used as an argument for ATTACH().
Type: constant; Related QED-Forth function: TIMER.OVERFLOW.ID
Header file: interupt.h

TIMESLICE_COUNT

Returns the 32-bit count of the number of clock ticks on the timeslicer clock. The count is set to zero by InitElapsedTime(), and the period of the clock is set by ChangeTaskerPeriod(); the default is 5 milliseconds (ms). To determine the elapsed time between two events in units of ms, simply subtract the corresponding counts and multiply by the number of milliseconds per count. The following example calculates the elapsed time and stores it in the variable elapsed_ms:

```
#define  MS_PER_COUNT      5
static long  start_count, end_count, elapsed_ms;
start_count = TIMESLICE_COUNT;      // start timing
        // now perform the actions that you want to time
end_count = TIMESLICE_COUNT;      // finish timing
elapsed_ms = (end_count  -  start_count) * MS_PER_COUNT;
```

See ReadElapsedSeconds() and InitElapsedTime().
Type: macro; Related QED-Forth function: TIMESLICE.COUNT
Header file: mtasker.h

void ToggleBits(uchar mask, uchar address)

For each bit of mask that is set, reverses the state of the corresponding bit of the 8 bit value at addr. Disables interrupts for ten cycles (2.5 microseconds) to ensure an uninterrupted read/modify/write operation. See also PIAToggleBits().
Type: _forth function; QED-Forth name: TOGGLE.BITS
Header file: xmem.h

int ToFlash(xaddr source, xaddr dest, uint numBytes)

Transfers numbytes (0 <= numbytes <= 65,535) starting at the specified source extended address, to the specified destination extended address in flash. The source may be anywhere in memory; it may even be in the flash which is being programmed. The destination must be in flash. Returns a flag equal to -1 if the programming was successful, or 0 if the programming failed. Reasons for failure include write protected flash (e.g., attempting to program page 0x0C while the page C write protect jumper is installed), or a destination that is not in a programmable page in flash memory. (If any locations in the flash are programmed more than 10,000 times, the cell may wear out causing a failure flag to be returned). Assuming that the standard 512 Kbyte flash is present on the board, writable flash pages include pages hex 4, 5, 6, 7, 0xC, 0xD, and 0x10-17 in the standard map, and pages 1, 2, 3, 7, 0xC, 0xD, and 0x18-1F in the download memory map. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user. Caution: the prolonged disabling of interrupts by TO.FLASH can adversely affect real-time servicing of interrupts including those associated with the secondary serial line. See PAGE.TO.FLASH and ALL.TO.FLASH in the Forth debugger glossary.
Type: _forth function; QED-Forth name: TO.FLASH
Header file: flash.h

int ToHeap(xaddr xhandle)

If xhandle is a valid 32-bit handle (pointer to a pointer) in the current heap, the heap item associated with the xhandle is returned to the heap (de-allocated), the heap is compacted, and a true flag is returned. If xhandle is not a valid handle in the current heap, no action is taken and a false flag is returned. ToHeap() is automatically invoked by DELETED().
Type: _forth function; QED-Forth name: TO.HEAP
Header file: heap.h

xaddr TO_XADDR(uint address, int page)

This C macro combines the specified 16-bit address and page into a single 32-bit extended address. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.
Type: macro
Header file: types.h

TRAILING_ZEROS

A user variable that contains a flag. If the flag is false, trailing zeroes are not printed when a floating point number is displayed in fixed or floating format by FPtoString() and PrintFP(). If true, trailing zeros are displayed. See FPtoString(), PrintFP(), FIXED() and FLOATING().
Type: macro; Related QED-Forth function: TRAILING.ZEROS
Header file: numbers.h

xaddr TransferHeapItem(xaddr xhandle, xaddr HeapEnd)

Copies the heap item specified by xhandle in the current heap into the heap whose CURRENT_HEAP is equal to HeapEnd. If the operation is successful, returns the 32-

bit handle of the new heap item; if unsuccessful, does nothing and returns zero. To copy a heap item within a single heap, see DupHeapItem().
Type: _forth function; QED-Forth name: TRANSFER.HEAP.ITEM
Header file: heap.h

TRANSMITTING
A system variable that is true (non-zero) if the secondary serial port (serial2) is in the process of transmitting a character. If the serial2 transmitter is active, the TRANSMITTING flag stays true until the serial2 output buffer is empty. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output).
Type: macro; Related QED-Forth function: TRANSMITTING
Header file: comm.h

TRUE
A constant equal to 1.
Type: constant; Related QED-Forth function: TRUE
Header file: utility.h

int TryToFSend(float message, float * mailboxAddr)
A subsidiary Forth function that is called by the recommended macro TRY_TO_FSEND(); see TRY_TO_FSEND().
Type: Forth function; QED-Forth name: ?SEND
Header file: mtasker.h

int TryToGet(xaddr * resourceAddr, uint resourcePage)
This function performs the actions of the macro TRY_TO_GET(); the macro version is recommended. The function expects to be passed the parameter resourcePage, which must equal 0 for programs coded in C. See TRY_TO_GET().
Type: _forth function; QED-Forth name: ?GET
Header file: mtasker.h

int TryToSend(long message, long * mailboxAddr)
A subsidiary Forth function that is called by the macro TRY_TO_SEND(); see TRY_TO_SEND().
Type: Forth function; QED-Forth name: ?SEND
Header file: mtasker.h

int TRY_TO_FSEND(float message, float * mailboxAddr)
If the mailbox with the specified mailboxAddress is empty (i.e., contains a floating point zero), this routine stores the 32-bit floating point "message" in mailboxAddr and returns a flag = -1. If xmailbox is not empty, this routine returns a flag = 0 and does not store the message. Does not execute Pause(). The message can be any 32-bit floating point quantity except 0; use TRY_TO_SEND to send a non-floating-point value as a message. To ensure that the state of the mailbox is correctly determined, TRY_TO_FSEND() disables interrupts for 16 to 50 cycles (4 to 12.5 microseconds). See FSEND(), FRECEIVE(), and MAILBOX.
Type: macro; Related QED-Forth function: ?SEND
Header file: mtasker.h

int TRY_TO_GET(xaddr * resourceAddr)

Checks the resource variable resourceAddr. If the resource is available (i.e., if it contains 0\0 or the current task's user base address), TRY_TO_GET() claims the resource by storing the current task's 32-bit base address in resourceAddr, and returns a flag equal to -1. Otherwise, TRY_TO_GET() returns a false (0) flag. Does not execute Pause(). To ensure that the state of the resource is correctly determined, TRY_TO_GET() disables interrupts for 27 to 57 cycles (6.75 to 14.25 microseconds). See GET(), RELEASE(), and RESOURCE.
Type: macro; Related function: TryToGet()
Header file: mtasker.h

int TRY_TO_SEND(long message, long * mailboxAddr)

If the mailbox with the specified mailboxAddress is empty (ie., contains a 32-bit 0 value), this routine stores the 32-bit message in mailboxAddr and returns a flag = -1. If xmailbox is not empty, this routine returns a flag = 0 and does not store the message. Does not execute Pause(). The message can be any 32-bit quantity except 0; use TRY_TO_FSEND to send a floating point value as a message. For example, the message can be an array address returned by ARRAYMEMBER() that points to a block of data. To ensure that the state of the mailbox is correctly determined, TRY_TO_SEND() disables interrupts for 16 to 50 cycles (4 to 12.5 microseconds). See SEND(), RECEIVE(), and MAILBOX.
Type: macro; Related QED-Forth function: ?SEND
Header file: mtasker.h

TWO_INTS

A union typedef that provides a way of converting two 16-bit integers into a 32-bit long, or vis versa. The definition is:

```
typedef union   {       ulong int32;
                struct  {       int     msInt;
                                int     lsInt;
                        } twoNums;
        } TWO_INTS;
```

For example, the following code splits a 32-bit result in longvar into two 16-bit integers in lower16bits and upper16bits:

```
ulong longvar;
int lower16bits, upper16bits;   // we want to set these
TWO_INTS temporary; // allocate union to convert type
temporary.int32 = longvar;
lower16bits = temporary.twoNums.lsInt;
lower16bits = temporary.twoNums.msInt;
```

See the source code in the TYPES.H file.
Type: typedef
Header file: types.h

UABORT

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 32-bit code field address of the user-supplied abort routine that is executed if the CUSTOM_ABORT flag is true (non-zero). If CUSTOM_ABORT is false

(zero), Abort() executes the default SysAbort() routine. UABORT is initialized by COLD to contain the code field address of SysAbort(). See Abort(), SysAbort(), and CUSTOM_ABORT.
Type: macro; Related QED-Forth function: UABORT
Header file: user.h

UASK_KEY

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 32-bit code field address of the AskKey() routine. See AskKey().
Type: macro; Related QED-Forth function: U?KEY
Header file: user.h

UDEBUG

A user variable (member of the currently active TASK.USER_AREA structure) that contains a flag. If the flag is non-zero then error checking and diagnostic stack printing are enabled in the interactive QED-Forth interpreter/compiler.
Type: macro; Related QED-Forth function: DEBUG
Header file: user.h

UEMIT

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 32-bit code field address of the Emit() routine. See Emit().
Type: macro; Related QED-Forth function: UEMIT
Header file: user.h

UERROR

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 32-bit code field address of the error routine that is executed if the CUSTOM_ERROR flag is true (non-zero). If CUSTOM_ERROR is false (zero), all system errors call the default system error routine which prints descriptive error messages. UERROR is initialized by COLD to contain the code field address of a simple default error handler that prints the hexadecimal system error number and executes Abort(). See CUSTOM_ERROR and Abort(), and consult the error message appendix in the Software Manual.
Type: macro; Related QED-Forth function: UERROR
Header file: user.h

UKEY

A user variable (member of the currently active TASK.USER_AREA structure) that contains the 32-bit code field address of the Key() routine. See Key().
Type: macro; Related QED-Forth function: UKEY
Header file: user.h

UP

A pointer to TASKBASE, which in turn is a pointer to the base of the current task's user area. UP returns an address whose contents is the TASKBASE address of the current task. See TASKBASE.
Type: macro; Related QED-Forth function: UP
Header file: user.h

UPAD

User variable (member of the currently active TASK.USER_AREA structure) that holds the 32-bit base address of PAD. See PAD.
Type: macro; Related QED-Forth function: UPAD
Header file: user.h

void UpdateDisplay(void)

Writes the contents of the DisplayBuffer() to the LCD display. When finished, leaves the display cursor pointing at the first position in the first line. For character displays, the cursor is turned off during the write to the display and is restored to its prior state after the update is complete, thus avoiding "flickering" of the cursor. Intermittently disables interrupts for 28 cycles (7 µsec) per byte to implement clock stretching.
Type: _forth function; QED-Forth name: UPDATE.DISPLAY
Header file: intrface.h

void UpdateDisplayLine(int lineNum)

Writes the contents of the specified lineNum in the DisplayBuffer() to the LCD display. lineNum is zero-based, and is clamped to a maximum of 1 less than the value returned by LinesPerDisplay(). UpdateDisplayLine() writes CharsPerDisplayLine() characters to the display. When finished, leaves the display cursor pointing at the first position in the line following lineNum. For character displays, the cursor is blanked during the write to the display and is restored to its prior state after the update is complete, thus avoiding "flickering" of the cursor. The lineNum follows the same rules explained in the description of BufferPosition(): for a graphics-style display the lineNum is interpreted differently depending on whether the display is being used in "text mode" or "graphics mode". In text mode, lineNum corresponds to the character line number; in graphics mode, lineNum corresponds to the pixel line number which is 8 times the character line number. Intermittently disables interrupts for 28 cycles (7 µsec) per byte to implement clock stretching.
Type: _forth function; QED-Forth name: UPDATE.DISPLAY.LINE
Header file: intrface.h

void UpdateDisplayRam(void)

Writes the contents of the Display Buffer to the LCD display. Unlike the related UpdateDisplay() function, UpdateDisplayRam() does NOT put the cursor and the display ram pointer to the "home position" at the upper left corner before writing to the display. When used with graphics displays, this function can be called after IsDisplayAddress() to write display data into "off-screen" display RAM, and then data can be scrolled onto the screen by changing the display's "home address". When finished, UpdateDisplayRam() leaves the display cursor and the display ram pointer at the first position in the first line. Intermittently disables interrupts for 28 cycles (7 µsec) per byte to implement clock stretching. See IsDisplayAddress() and UpdateDisplay().
Type: _forth function; QED-Forth name: (UPDATE.DISPLAY)
Header file: intrface.h

USER_AREA

A struct typedef that declares the user area structure containing task-specific variables and pointers. In turn, the USER_AREA structure is the first element in the TASK

structure. For a full definition of this structure, see the source code in the user.h file. See also TASK.
Type: typedef
Header file: user.h

void UseSerial1(void)

Installs the primary serial port (serial1) as the serial link called by Emit(), AskKey(), and Key() used by the default task that runs at startup. The serial1 port is associated with the 68HC11's on-chip hardware UART. Stores the code address of Key1() in UKEY, the code address of AskKey1() in UASK_KEY, and the code address of Emit1() in UEMIT. Thus the vectored routines Key(), AskKey(), and Emit() will automatically execute the serial1 routines Key1(), AskKey1(), and Emit1() respectively. Initializes the resource variable SERIAL1_RESOURCE to zero, and initializes the resource variable associated with the prior serial channel in use (typically either SERIAL1_RESOURCE or SERIAL2_RESOURCE) to zero. Does not disable the serial2 port.
Type: _forth function; QED-Forth name: USE.SERIAL1
Header file: comm.h

void UseSerial2()

Installs the secondary serial port (serial2) as the serial link called by Emit(), AskKey(), and Key() used by the default task that runs at startup. Calls InitSerial2() to initialize the serial2 port, and globally enables interrupts to allow the serial2 port to operate. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). UseSerial2() stores the code address of Key2() in UKEY, the code address of AskKey2() in UASK_KEY, and the code address of Emit2() in UEMIT. Thus the vectored routines Key(), AskKey(), and Emit() will automatically execute the serial1 routines Key2(), AskKey2(), and Emit2() respectively. Initializes the resource variable SERIAL2_RESOURCE to zero, and initializes the resource variable associated with the prior serial channel in use (typically either SERIAL1_RESOURCE or SERIAL2_RESOURCE) to zero. Does not disable the serial1 port. See Baud2() and Serial2AtStartup().
Type: _forth function; QED-Forth name: USE.SERIAL2
Header file: comm.h

UTIB

User variable (member of the currently active TASK.USER_AREA structure) that holds the 32-bit base address of the Terminal Input Buffer (TIB). See TIB.
Type: macro; Related QED-Forth function: UTIB
Header file: user.h

void Warm(void)

Restarts the QED-Forth system and clears the data and return stacks and executes Abort(). Unlike Cold(), Warm() does not initialize all of the user variables to their default values. Note that this function can be called interactively from the terminal by typing:

```
WARM
```

Calling WARM interactively before downloading a program is good practice, as it aborts any active multitasking program that may be in progress. See also COLD.

Type: _forth function; QED-Forth name: WARM
Header file: qedsys.h

WATCH_DATE

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_DATE returns the current date (1...31). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_DAY

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_DAY returns the current day (1...7). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_HOUR

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_HOUR returns the current hour (0...23). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_HUNDREDTH_SECONDS

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_HUNDREDTH_SECONDS returns the current hundredth-seconds (0...99) since the last integral second. (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_MINUTE

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_MINUTE returns the current minute (0...59). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_MONTH

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_MONTH returns the current month (1...12). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

watch_results

This structure located at 0xB3F8 is an instance of the CALENDAR_TIME typedef that defines the bytes that hold the results of a read of the battery-backed real-time clock. A set of macros (WATCH_SECONDS, WATCH_MINUTES, WATCH_HOUR, etc.) have been pre-defined to facilitate easy access to the watch results; see the glossary entry for ReadWatch().
Type: structure instance
Header file: watch.h

WATCH_SECONDS

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_SECONDS returns the current seconds (0...59). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

WATCH_YEAR

A structure element whose contents were updated by the most recent execution of ReadWatch(). When used as the right-hand side of an assignment statement, WATCH_YEAR returns the current year (0...99). (Of course, this requires that a properly set battery-operated real-time clock is installed on the QED Board.) See the glossary entry for ReadWatch() for an example of use.
Type: macro
Header file: watch.h

int WhichMap(void)

Returns a 0 if the current memory map is the "standard map", and returns a 1 if the current map is the "download map". If the standard map is active, pages 4, 5, and 6 and 0x10-0x17 are addressed as flash, and pages 1, 2, and 3 (and pages 0x18-0x1F, if present) are addressed in as RAM. If the download map is active, pages 4, 5, and 6 (and 0x10-0x17, if present) are addressed as RAM, and pages 1, 2, and 3 and pages 0x18-0x1F are addressed as flash memory. This routine allows a user or program to verify which map is currently being used. After a "factory cleanup" operation, the standard map is active. See STANDARD.MAP and DOWNLOAD.MAP. See StandardMap() and DownloadMap().
Type: _forth function; QED-Forth name: WHICH.MAP
Header file: flash.h

long XaddrDifference(xaddr addr1, xaddr addr2)

Subtracts addr2 from addr1 to yield the signed double number result d. There is an unchecked error if one of the xaddresses is in common memory (addr >= 0x8000) and the other is in paged memory (addr <= 0x7FFF on any page). Note that in paged memory, the address immediately following 0x7FFF is address 0x0000 on the following page.
Type: _forth function; QED-Forth name: X1-X2>D
Header file: xmem.h

uint XADDR_TO_ADDR(xaddr xaddress)

This C macro converts the specified 32-bit extended address into its constituent 16-bit address. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.
Type: macro
Header file: types.h

int XADDR_TO_PAGE(xaddr xaddress)

This C macro converts the specified 32-bit extended address into its constituent page. It is present in \fabius\include\mosaic\types.h starting with V1.2 of the types.h file.
Type: macro
Header file: types.h

XIRQ_ID

A constant that returns the interrupt identity code for the external non-maskable interrupt called XIRQ. Used as an argument for ATTACH(). The XIRQ interrupt is activated by an active-low signal on the XIRQ input pin and is enabled by the X bit in the condition code register.
Type: constant; Related QED-Forth function: XIRQ.ID
Header file: interupt.h

uchar _peekTerminal(void)

A serial I/O primitive called by printf() and other C printing functions. Checks the serial1 input port. If an input character is present, adds it to a 10-byte input buffer located near the top of the 1K on-chip RAM. _peekTerminal() returns the number of pending characters in the input buffer. See _readTerminal to access the contents of the input buffer. See also AskKey(), _readChar() and _writeChar().
Type: C function
Header file: comm.h

uchar _readChar(void)

A serial I/O primitive called by printf() and other C printing functions. Calls _readTerminal to retrieve the latest input character from the current serial port, and then calls _writeChar to echo the character. Returns the input character. Does not execute Pause() while waiting for the input character. See also Key(), _readTerminal(), _peekTerminal() and _writeChar().
Type: C function
Header file: comm.h

uchar _readTerminal(void)

A serial I/O primitive called by printf() and other C printing functions. Removes one character from the input buffer associated with the current serial input port [see _peekTerminal()] and returns the character. If the input buffer is empty, waits until the next input character appears. See also _readChar() and _writeChar().
Type: C function
Header file: comm.h

void _writeChar(uchar chr)

A serial I/O primitive called by printf() and other C printing functions. Writes the specified character chr to the current serial port. If chr is a linefeed (ascii 10), writes a carriage return (ascii 13) to the serial port before writing the linefeed. Does not execute Pause(). See also Emit(), _peekTerminal(), _readChar() and _writeTerminal().
Type: C function
Header file: comm.h

_Q

A macro whose default definition is simply:

```
_pascal
```

Used in front of a function declaration or definition, it instructs the C compiler to label the function as a pascal type and to use the pascal calling convention (push leftmost parameter first, pass rightmost parameter in registers). This macro is used to enable the interactive debugging features offered by the QED Board. Functions declared with the _Q specifier may be individually executed in an interactive fashion from your terminal; this greatly speeds debugging. Without the interactive capability, a compiled C program can only execute one function (main), and if you want to test individual functions in the program you must continually recode main to call the function and then recompile the program. Using the _Q debugging feature offers a simpler alternative. Instead of having to revector the "main" routine to execute the function of interest, you can interactively execute each function in the program with a variety of input parameters of your choice. This lets you test each function to isolate bugs. There is no significant performance penalty for using the _Q specifier; in fact, the _pascal functions are more memory-efficient than the standard functions (because the "stack cleanup" code is compiled in the function itself, rather than in each calling function.)
Example of use:

```
_Q float Square( float input)
{       return (input * input);
}
```

After this function is compiled in a program and the .TXT download file is sent to the QED Board, you can test the function with any input. For example, to test the Square() function with an input of 3.5, simply execute from the terminal

```
Square( float 3.5)
```

Note that there is no space between the "e" and the "(", and there must be at least one space between the "(" and the next character. The "float" keyword is required to tell the interpreter that the input parameter is a floating point number. The QED Board will respond with a summary of the return value in several formats:

```
Rtn: 16708   0 =0x41440000=fp: 12.25
```

We know that this routine returns a floating point number, so we see that the return value is 12.25, which is the correct answer. For more details, see the "Interactive

Debugging Routines" section of this glossary, as well as the debugging chapter of the "Getting Started" booklet.
Type: macro
Header file: types.h

Interactive Debugger Glossary

Introductory Notes

This glossary summarizes functions and keywords that can be interactively typed at the terminal and interpreted by the QED-Forth debugger and operating system. Note that Forth tokens are delimited by spaces, and the tokens themselves may contain any printable non-space character, including parentheses, commas, periods, and exclamation points, etc. Also note that Forth is case-insensitive, so that any keyword can be typed in all capitals, all small letters, or any combination.

To improve the readability of this glossary, all of the QED-Forth interactive debugger functions are CAPITALIZED to distinguish them from the type parameters that specify the input and output parameters. This is especially important for glossary entries such as "INT", where the name of the QED-Forth function is spelled the same as the type designator "int" that we use to describe the input and output parameters.

Many of the entries in this glossary refer to entries in the Main Glossary. For example, the interactive QED-Forth function PRIORITY.AUTOSTART which sets up a program to automatically start each time the QED Board turns on is fully described in the Main Glossary entry for Priority-Autostart().

We use some non-ANSI notation when specifying the input parameters of these functions. For example, when an input parameter can be one of several types, we use the notation:

function_name ([type1] or [type2] or [type3])

Forth compilers are more forgiving than C compilers when it comes to accepting input parameters of differing types.

Glossary Entries

(Alphabetized in ASCII Order)

void (EE!) (int val, int* addr)
Stores val at the specified addr in EEPROM. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. See StoreEEInt() in the Main Glossary for an example of how to define a "variable" in EEPROM and interactively initialize it.
Pronunciation: "paren-e-e-store"

void (EE2!) (long val, long* addr)

Stores val at the specified addr in EEPROM. The most significant word is stored at addr, and the least significant word is stored at addr+2. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. See StoreEELong() in the Main Glossary for an example of how to define a "variable" in EEPROM and interactively initialize it.
Pronunciation: "paren-e-e-two-store"

void (EEC!) (char val, char* addr)

Stores val at the specified addr in EEPROM. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. See StoreEEChar() in the Main Glossary for an example of how to define a "variable" in EEPROM and interactively initialize it.
Pronunciation: "paren-e-e-c-store"

void (EEF!) (float val, float* addr)

Stores val at the specified addr in EEPROM. The most significant word is stored at addr, and the least significant word is stored at addr+2. Any byte that already contains the specified contents is not re-programmed; this helps lengthen the lifetime of the EEPROM. See StoreEEFloat() in the Main Glossary for an example of how to define a "variable" in EEPROM and interactively initialize it.
Pronunciation: "paren-e-e-f-store"

void . (int)

Prints the input parameter as a signed integer. Number conversion is performed in the current number base set by the most recent execution of DECIMAL or HEX.
Pronunciation: "dot"

void =CHAR (LHS: [addr] or [xaddr] ; RHS: [char] or [int] or [float] or [addr] or [xaddr])

=CHAR is a QED-Forth function that acts as an interactive assignment operator. It is used in the form:

<destination> =CHAR <char_specifier>

where <destination> is a 16-bit address left on the data stack by a variable name, or a 32-bit xaddress left on the data stack by a FORTH_ARRAY element. <char_specifier> is either a valid number or a variable name or FORTH_ARRAY element that contains a byte. =CHAR assigns the byte specified by the right-hand-side (RHS) to the memory location specified by the left-hand-side (LHS).
Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static char  data;
FORTH_ARRAY buffer8;
DIM(char, 3, 4, &buffer8);
```

Now all of the following interactive commands may be typed from the terminal:

Interactive Command	Result
data =CHAR 0x44	Assigns hex 44 to the variable data
buffer8[1, 2] =CHAR 9	Assigns 9 to array element at row=1, col=2

data =CHAR buffer8[1,2] Assigns contents of array element to data
buffer8[2,3] =CHAR data Assigns contents of data to array element

In summary, the syntax is similar to a C assignment statement. Note that =CHAR must be typed as one word and must be preceded and followed by spaces.

void =FLOAT (LHS: [addr] or [xaddr] ; RHS: [float] or [addr] or [xaddr])

=FLOAT is a QED-Forth function that acts as an interactive assignment operator. It is used in the form:

<destination> =FLOAT <float_specifier>

where <destination> is a 16-bit address left on the data stack by a variable name, or a 32-bit xaddress left on the data stack by a FORTH_ARRAY element. <float_specifier> is either a valid floating point number or a variable name or FORTH_ARRAY element that contains a floating point number. =FLOAT assigns the float specified by the right-hand-side (RHS) to the memory location specified by the left-hand-side (LHS).

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static float radius;
FORTH_ARRAY fbuffer;
DIM(float, 3, 4, &fbuffer);
```

Now all of the following interactive commands may be typed from the terminal:

Interactive Command	Result
radius =FLOAT 1.2E3	Assigns 1,200. to the variable radius
fbuffer[1, 2] =FLOAT 2.3	Assigns 2.3 to array element at row=1,col=2
radius =FLOAT fbuffer[1,2]	Assigns contents of array element to radius
fbuffer[2,3] =FLOAT radius	Assigns contents of radius to array element

In summary, the syntax is similar to a C assignment statement. Note that =FLOAT must be typed as one word and must be preceded and followed by spaces.

void =INT (LHS: [addr] or [xaddr] ; RHS: [char] or [int] or [float] or [addr] or [xaddr])

=INT is a QED-Forth function that acts as an interactive assignment operator. It is used in the form:

<destination> =INT <integer_specifier>

where <destination> is a 16-bit address left on the data stack by a variable name, or a 32-bit xaddress left on the data stack by a FORTH_ARRAY element. <integer_specifier> is either a valid number or a variable name or FORTH_ARRAY element that contains an integer. =INT assigns the integer specified by the right-hand-side (RHS) to the memory location specified by the left-hand-side (LHS).

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static int radius;
FORTH_ARRAY buffer16;
DIM(int, 3, 4, &buffer16);
```

Now all of the following interactive commands may be typed from the terminal:

Interactive Command	Result
radius =INT 5	Assigns 5 to the variable radius
buffer16[1, 2] =INT 0x44	Assigns hex 44 to array element at row=1,col=2
radius =INT buffer16[1,2]	Assigns contents of array element to radius
buffer16[2,3] =INT radius	Assigns contents of radius to array element

In summary, the syntax is similar to a C assignment statement. Note that =INT must be typed as one word and must be preceded and followed by spaces.

void =LONG (LHS: [addr] or [xaddr] ; RHS: [char] or [int] or [addr] or [xaddr])

=LONG is a QED-Forth function that acts as an interactive assignment operator. It is used in the form:

<destination> =LONG <long_specifier>

where <destination> is a 16-bit address left on the data stack by a variable name, or a 32-bit xaddress left on the data stack by a FORTH_ARRAY element. <long_specifier> is either a valid number or a variable name or FORTH_ARRAY element that contains an long. =LONG assigns the long specified by the right-hand-side (RHS) to the memory location specified by the left-hand-side (LHS).

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static long data = 56789;
FORTH_ARRAY buffer32;
DIM(long, 3, 4, &buffer32);
```

Now all of the following interactive commands may be typed from the terminal:

Interactive Command	Result
data =LONG 54321	Assigns 54321 to the variable data
buffer32[1, 2] =LONG 0x44	Assigns hex 44 to array element at r=1, c=2
data =LONG buffer32[1, 2]	Assigns contents of array element to data
buffer32[2, 3] =LONG data	Assigns contents of data to array element

In summary, the syntax is similar to a C assignment statement. Note that =LONG must be typed as one word and must be preceded and followed by spaces.

void ABORT (void)

Aborts the current operation; to use, simply type at the terminal:

ABORT

If the CUSTOM_ABORT flag is true (non-zero), executes the abort routine whose xcfa (32-bit extended code field address) is stored in the user variable UABORT, and then returns to the routine that called ABORT. If CUSTOM_ABORT is false (zero), executes the default routine SysAbort() which clears the data and return stacks, and sets the page to the default page (0). If an autostart vector has been installed [see Autostart() and PriorityAutostart()], SysAbort() executes the specified routine; otherwise it executes QUIT which sets the execution mode and enters the QED-Forth monitor. If the stack pointers do not point to common RAM, a COLD restart is initiated. See the entry in the Main Glossary entry for Abort().

void AUTOSTART (xaddr)

Expects on the data stack a 32-bit code field xaddress (xcfa) of a function. Compiles a 6-byte sequence into the EEPROM in the 68HC11. On subsequent restarts and ABORTs, the routine having the specified xcfa will be executed. This allows a finished application to be automatically entered upon power up and resets.

CAUTION: To put your application into production, it is recommended that you use the PRIORITY.AUTOSTART function which stores the 6-byte autostart sequence in flash memory.

Usage: We recommend that Autostart() and PriorityAutostart() be executed interactively from the QED-Forth monitor. The easiest way to do this is to use Forth

syntax instead of C syntax. After your application program is completed and debugged, simply type from your terminal the command:

CFA.FOR MAIN AUTOSTART

This writes a pattern into EEPROM that causes MAIN to be executed upon all subsequent resets and restarts.

Implementation detail: At location hex AE00 in EEPROM, AUTOSTART writes the pattern 1357 followed by the four byte xcfa. To undo the effects of this command and return to the default startup action, type the QED-Forth command

NO.AUTOSTART

from your terminal. To recover from the installation of a buggy autostart routine, use the special cleanup mode as described in the "Programming the QED Board in C" chapter in the "Getting Started" Manual. See PRIORITY.AUTOSTART, and see the entry in the Main Glossary entry for Autostart().

void BAUD1.AT.STARTUP (int baud)

Configures the QED Board so that the baud rate of the primary serial port (serial1) supported by the 68HC11's hardware UART will equal the specified standard baud rate upon all subsequent resets and restarts. Standard baud rates are 150, 300, 600, 1200, 2400, 4800, 9600, and 19200 baud.

Example of use: To set the baud rate of the serial1 port to 19,200 baud, type from your terminal:

DECIMAL 19200 BAUD1.AT.STARTUP

Be sure to modify your terminal's baud rate setting to match the new baud rate. The new rate will take effect upon all subsequent resets and restarts.

Implementation detail: This routine calls InstallRegisterInits() which writes into EEPROM the required contents of INIT (=$B8_H$), the contents of BAUD that corresponds to the specified baud rate, and the contents of OPTION, TMSK2, and BPROT that are present when this routine is executed. These values are installed in their respective registers upon each subsequent reset and restart. To undo the effects of this command, type from your terminal the command

DEFAULT.REGISTER.INITS

or invoke the special cleanup mode as described in the "Programming the QED Board in C" chapter in the "Getting Started" Manual. See the entry in the Main Glossary entry for Baud1AtStartup().

void BAUD2 (int baud)

Sets the baud rate of the secondary serial port (serial2) supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). Smooth file transfers can be achieved at up to 4800 baud. The baud rate of serial2 is initialized to 1200 baud by the COLD restart routine. See UseSerial2().

Example of use: To set the baud rate of the serial2 port to 4800 baud, type from your terminal:

DECIMAL 4800 BAUD2

Be sure to modify your terminal's baud rate setting to match the new baud rate, and make sure that DIP switch#4 on the QED Board is in the ON position when using the serial2 port. See the entry in the Main Glossary entry for Baud2().

int CALC.CHECKSUM (xaddr xbase, int numbytes)

Calculates a 16-bit checksum for the buffer specified by xaddr and +n, where xaddr is the starting address, and +n is the number of bytes (0 <= +n < 32,768). The buffer must not cross a page boundary, and n must be an even number of bytes. The checksum is calculated by initializing a 16-bit accumulator to zero, then adding in turn each 2-byte number in the buffer to the accumulator; the checksum is the final value of the accumulator. Using this routine provides a method of checking whether the contents of an area of memory have changed since a prior checksum was calculated. This routine is optimized for speed, and executes at less than 3 microseconds per byte.

void CALL.CFN (xaddr <input_parameter_list>)

A low-level function inserted by the "Make" utility in the .TXT download file. When properly inserted in a QED-Forth function, enables interactive calls to C functions that were declared using the _Q keyword. CALL.CFN expects on the data stack a 32-bit xaddress representing the execution address of the function to be called. CALL.CFN removes from the input stream a list of comma-delimited parameters terminated by the) character. It then sets up the proper stack frame for a "pascal" type function (i.e., a function declared using the _Q or _pascal keyword) that has been compiled by the Control-C compiler. CALL.CFN calls the designated function, then prints the return values (passed in the D and Y registers):

in the current number base as two 16-bit integers;
as a 32-bit hexadecimal number; and,
as a floating point number.

It is up to the programmer to decide which (if any) of these return value summaries is relevant based on the declared type of the called function's return value.

Example of use: Assume that a function with the following prototype has been compiled and downloaded as part of the GETSTART.C program:

```
static uint radius;
_Q float CalcArea( uint radius);
```

Then the GETSTART.TXT download file created by the Control-C compiler will include the following QED-Forth declarations:

```
HEX
8E1B  CONSTANT radius
: CalcArea(  DIN 040182  CALL.CFN ;
```

While the exact numbers may vary, in this case DIN 040182 is the 32-bit execution address of the CalcArea() function. If we interactively type at the terminal the following commands to initialize the radius variable to 5 and then call the CalcArea() function :

```
radius =INT  5
CalcArea( int  radius)
```

QED-Forth executes the function and prints the following summary of the return value:

```
Rtn: 17053 5242  =0x429D147A =fp: 78.54  ok
```

We know that CalcArea() returns a floating point (fp) result, so we identify the return value as 78.54 (and indeed this is the area of a circle that has a radius equal to 5). Consult the "Getting Started" book for a full discussion of interactive function calling.

Pronunciation: "call-c-function"

xaddr CFA.FOR (<function_name>)

Removes <function_name> from the input stream and leaves its extended code field address (cfa) on the data stack. An error occurs if no <function_name> is given or if

<function_name> cannot be found in the QED-Forth dictionary. Typically used in conjunction with AUTOSTART or PRIORITY.AUTOSTART. For example:

```
CFA.FOR MAIN PRIORITY.AUTOSTART
```

configures the QED Board so that the MAIN function is executed upon each subsequent reset or restart.

Pronunciation: "c-f-a-for"

char CHAR([char] or [int] or [float] or [addr] or [xaddr])

CHAR is a QED-Forth function that examines the next token; if it is a valid number such as 5 or 3.2, CHAR simply converts it to the nearest 8-bit byte. There is an unchecked error if the input is not in the range 0-255 (unsigned char) or -128 to _127 (signed char). If the next token is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), CHAR extracts the 8-bit contents stored at the specified memory location. CHAR is also used to specify the type of an input parameter when interactively calling a function.

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static char ascii_code = 55;
FORTH_ARRAY buffer8;
DIM(char, 3, 4, &buffer8);
_Q void SaveByte( char val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SaveByte(55, 1, 2, &buffer8);
```

Now all of the following interactive commands typed from the terminal will result in the number "55" being printed by QED-Forth:

```
CHAR 55 U.
CHAR 55.45 U.
CHAR ascii_code U.
CHAR buffer8[ 1, 2] U.
```

Moreover, to initialize the element of buffer8 at row=0, col=1 to 34, we could interactively type at the terminal:

```
SaveByte( CHAR 34, INT 0, INT 1, buffer16)
```

This demonstrates the use of the CHAR keyword in specifying the input parameter types of interactive function calls; the syntax is similar to a C function prototype.

char CHAR* ([addr] or [xaddr])

CHAR* is a QED-Forth function that examines the next token; if it is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), CHAR* extracts the 16-bit pointer stored at the specified memory location, and in turn extracts the 8-bit byte pointed to by the pointer.

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static char data = 5;
char* data_ptr = &data;
FORTH_ARRAY buffer16;
DIM(char*, 3, 4, &buffer16);
_Q void SavePtr(char* val, int row, int col, FORTH_ARRAY* array_ptr)
```

```
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
    SavePtr(&radius, 1, 2, &buffer16);
```

Now all of the following interactive commands typed from the terminal will result in the number "5" being printed by QED-Forth:

```
CHAR data U.
CHAR* data_ptr U.
CHAR* buffer16[ 1, 2] U.
```

void CLEAR.BOOT.VECTOR (void)

Removes a boot vector from page 0x0C. Note that the “page C write protect” jumper must be removed for this function to be effective. This function is called during a "factory cleanup", but it is not called by **NO.AUTOSTART**. See SET.BOOT.VECTOR. This function is typically invoked interactively from the QED-Forth prompt.

void COLD (void)

Disables interrupts and restarts the QED-Forth system and initializes all of the user variables to their default values. To use, simply type at your terminal the command:

```
COLD
```

Initializes the following machine registers:

```
PORTG, DDRG, TMSK2, SPCR, BAUD, SCCR1, SCCR2, BPROT,
OPT2, OPTION, HPRIO, INIT, CSCTL.
```

Initializes the vectors of the vital interrupts if InitVitalIRQsOnCold() has been executed. Calls Abort() which clears the stacks and calls either the QED-Forth interpreter or an autostart routine that has been installed using Autostart() or PriorityAutostart(). If ColdOnReset() has been executed, every reset or power-up will invoke a Cold() as opposed to a Warm() initialization sequence. See the entry in the Main Glossary entry for Cold().

void COLD.ON.RESET (void)

Initializes a flag in EEPROM that causes subsequent resets to execute a cold restart (as opposed to the standard warm-or-cold restart). This option is useful for turnkeyed systems that have an autostart word installed; any error or reset causes a full Cold() restart which initializes all user variables, after which the autostart routine completes the system initialization and enters the application routine. To use, simply type at your terminal:

```
COLD.ON.RESET
```

See STANDARD.RESET, and see the entry in the Main Glossary entry for ColdOnReset().

Implementation detail: Initializes location hex AE1C in EEPROM to contain the pattern 13.

void D. (long)

Prints the input parameter as a signed long. Number conversion is performed in the current number base set by the most recent execution of DECIMAL or HEX.

Pronunciation: "d-dot"

void DECIMAL (void)

Sets the number conversion base to decimal. See HEX. The number conversions occur when inputting numbers typed at the terminal, and when QED-Forth prints numbers to the terminal.

void DEFAULT.REGISTER.INITS (void)

Undoes the effect of the INSTALL.REGISTER.INITS command; to use, simply type at the terminal:

```
DEFAULT.REGISTER.INITS
```

Implementation detail: sets the contents of location 0xAE06 in EEPROM to 0xFF to ensure that default initializations will be used after subsequent resets. The default register initializations are:

Register Name	Register Address	Default Value
OPTION	0x8039	0x33
TMSK2	0x8024	0x02
BPROT	0x8035	0x10
BAUD	0x802B	0x31

See the entry in the Main Glossary entry for DefaultRegisterInits().

xaddr DO[] (addr <input_parameter_list>)

A low-level function inserted by the "Make" utility in the .TXT download file. When properly inserted in a QED-Forth function, enables interactive examination and modification of FORTH_ARRAY elements. Expects on the data stack a 16-bit address representing the pfa (parameter field address) of a FORTH_ARRAY. DO[] removes from the input stream a row specifier, a comma, a column specifier, and a terminating]. It leaves on the stack the 32-bit xaddress of the specified element in the specified FORTH_ARRAY.

Example of use: Assume that the following declaration has been compiled and downloaded as part of a C program:

```
FORTH_ARRAY circle_parameters;
```

Then the .TXT download file created by the Control-C compiler will include a QED-Forth definition of the form:

```
: circle_parameters  8E03 DO[] ;
```

While the exact numbers may vary, in this case 8E03 is the 16-bit pfa of the circle_parameters array. Now debugger expressions such as:

```
float circle_parameters[ 0,0] PrintFP
circle_parameters[1,0] =FLOAT 3.1416
```

can be interpreted and executed by the interactive debugger.

Pronunciation: "do-brackets"

void DOWNLOAD.MAP (void)

Sets a flag in EEPROM and changes the state of a latch in the onboard PALs to put the download memory map into effect. After execution of this routine, and upon each subsequent reset or restart, hex pages 4, 5, 6, and 0x10-17 are addressed in RAM, and pages 1, 2, 3, and 0x18-1F are addressed in flash memory. This allows code (and Forth names) to be compiled into RAM on pages 4, 5 and 6 (and, if a 512K RAM is present, into pages 0x10-17) and then transferred to flash using the PAGE.TO.FLASH function. To establish the standard memory map, see the glossary entry for

STANDARD.MAP. Note that the standard map is active after a "factory cleanup" operation.

void DUMP (xaddr start, uint numbytes)

Displays the contents of numbytes bytes starting at the specified start xaddr. The contents are dumped as hexadecimal bytes regardless of the current number base, and the ascii equivalent contents are also displayed. For example, to display 0x40 bytes starting at address 0x1000 on page 1, execute:

```
HEX 1000 1  40  DUMP
```

and to display the last 0x10 bytes on page 1 and the first 0x20 bytes on page 2, execute

```
7FF0 1 30 DUMP
```

DUMP calls the function PAUSE.ON.KEY, so the dump responds to XON/XOFF handshaking and can be aborted by typing a carriage return; see PauseOnKey() in the Main Glossary.

void DUMP.INTEL (char* start, uint start_page, uint reported_start, uint numbytes)

The parameters start and start_page specify the location of the first byte to be dumped, reported_start specifies the starting address reported in the dump, and numbytes is the number of bytes to be dumped. Dumps the contents of numbytes bytes starting at start on start_page using the standard ascii Intel hex format which is useful for transferring data between devices. The line format is:

```
:{#bytes}{reported.addr}{00}{byte}{byte} ...{byte}{checksum}
```

All numbers are in hexadecimal base. Each line starts with a : character, followed by a 2-digit number of bytes (20, indicating that the contents of 0x20 bytes are displayed per line), followed by a 4-digit starting address for the line, followed by 00, followed by the contents of the memory locations (2 hex digits per byte), and concluding with a checksum followed by a carriage return/linefeed. The checksum is calculated by summing each of the bytes on the line into an 8-bit accumulator and negating (two's complementing) the result. The hex dump ends with the line

```
:00000001FF
```

For example, to dump 0x40 bytes starting at QED Board address 0x1000\1 so that the bytes reside at the beginning of a target memory device, execute:

```
HEX  1000  01  0000  40  DUMP.INTEL
```

which specifies 0x1000\1 as the starting address, 0000 as the reported base address in the memory device, and 0x40 as the number of bytes to be dumped. To dump the last 0x20 bytes on page 1 and the first 0x40 bytes on page 2 so that they reside at locations 0x7FE0 through 0x803F in the target memory device, execute

```
7FE0  1  7FE0  60  DUMP.INTEL
```

The complementary word RECEIVE.HEX loads QED memory starting at any location based on a received Intel or Motorola hex file. DUMP.INTEL calls the word PAUSE.ON.KEY, so the dump responds to XON/XOFF handshaking and can be aborted by typing a carriage return. See DUMP.S1, DUMP.S2, RECEIVE.HEX and PAUSE.ON.KEY.

void DUMP.S1 (char* start, uint start_page, uint reported_start, uint numbytes)

The parameters start and start_page specify the location of the first byte to be dumped, the 16-bit reported_start specifies the starting address reported in the dump, and numbytes is the number of bytes to be dumped. Dumps using the standard ascii

Motorola S1 hex format which is useful transferring data between devices. Motorola S1 records report 16 bit addresses. (To report full 24 bit addresses, see DUMP.S2.) Outputs an S0 header record which is

S00900004845414445524D

then as many S1 data records as required, followed by an S9 termination record which is

S9030000FC

The Motorola S1 hex line format is:

S1{#bytes}{16bit.reported.addr}{byte}...{byte}{chksum}

All numbers are in hexadecimal base. Each line starts with a the record type (S1 in this case), followed by a 2-digit number of bytes (23, which equals 0x20 bytes per line plus 3 bytes for the reported address and checksum), followed by a 4-digit starting address for the line, followed by the contents of the memory locations (2 hex digits per byte), and concluding with a checksum. The checksum is calculated by summing each of the bytes on the line (excluding the record type) into an 8-bit accumulator and (one's) complementing the result.

For example, to dump 0x40 bytes starting at QED Board address 0x1000\1 so that the bytes reside at the beginning of a target memory device, execute:

HEX 1000 01 0000 40 DUMP.S1

which specifies 0x1000\1 as the starting address, 0000 as the reported base address in the memory device, and 0x40 as the number of bytes to be dumped. To dump the last 0x20 bytes on page 1 and the first 0x40 bytes on page 2 so that they reside at locations 0x7FE0 through 0x803F in the target memory device, execute:

7FE0 1 7FE0 60 DUMP.S1

The complementary word RECEIVE.HEX loads QED memory starting at any location based on a received Motorola or Intel hex file. DUMP.S1 calls the word PAUSE.ON.KEY, so the dump responds to XON/XOFF handshaking and can be aborted by typing a carriage return. See DUMP.S2, DUMP.INTEL, RECEIVE.HEX and PAUSE.ON.KEY.

void DUMP.S2 (char* start, uint start_page, long reported_start, uint numbytes)

The parameters start and start_page specify the location of the first byte to be dumped, the 32-bit reported_start specifies the starting address reported in the dump, and numbytes is the number of bytes to be dumped. Dumps using the standard ascii Motorola S2 hex format which is useful for transferring data between devices. Motorola S2 records report 24 bit addresses. (To report 16 bit addresses, see DUMP.S1.) Dumps an S0 header record which is

S00900004845414445524D

then as many S2 data records as required, followed by an S9 termination record which is

S9030000FC

The Motorola S2 hex line format is:

S2{#bytes}{24bit.reported.addr}{byte}...{byte}{chksum}

All numbers are in hexadecimal base. Each line starts with a the record type (S2 in this case), followed by a 2-digit number of bytes (24, which equals 0x20 byte per line plus 4 bytes for the reported address and checksum), followed by a 6-digit starting address for the line, followed by the contents of the memory locations (2 hex digits per byte), and concluding with a checksum. The checksum is calculated by summing each of the bytes on the line (excluding the record type) into an 8-bit accumulator and (one's)

complementing the result. DUMP.S2 calls the word PAUSE.ON.KEY, so the dump responds to XON/XOFF handshaking and can be aborted by typing a carriage return. See DUMP.S1, DUMP.INTEL, RECEIVE.HEX and PAUSE.ON.KEY.

Example of use: Assume that you have created an application program in pages 4, 5, and 6, and used PRIORITY.AUTOSTART to configure a flash-based autostart vector so that the application runs automatically upon each power-up and restart. To dump a complete application program that resides on pages 4, 5 and 6, so that the bytes reside at the beginning of a flash memory device, execute:

```
HEX
0000 04  DIN  000000   8000  DUMP.S2
0000 05  DIN  008000   8000  DUMP.S2
0000 06  DIN  010000   8000  DUMP.S2
```

Now you can edit the resulting file, concatenate the 3 dumps into 1 large S-record by removing all but the first and last S0 (header) and S9 (termination) records, and re-save the file. To transfer the application to a new QED Board, simply execute

```
DOWNLOAD.MAP
0 4 RECEIVE.HEX       <send the captured file>
4 PAGE.TO.FLASH
5 PAGE.TO.FLASH
6 PAGE.TO.FLASH
STANDARD.MAP
```

This is a time-effective method of mass producing QED-based products running a “turnkeyed” autostart program.

float FLOAT ([char] or [int] or [float] or [addr] or [xaddr])

FLOAT is a QED-Forth function that examines the next token; if it is a valid integer or QED-formatted floating point number such as 5 or 3.2, FLOAT simply converts it to an ANSI-C-formatted floating point number. If the next token is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), FLOAT extracts the 32-bit (float) contents stored at the specified memory location. FLOAT is also used to specify the type of an input parameter when interactively calling a function.

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static float radius = 5.25;
FORTH_ARRAY fbuffer;
DIM(float, 3, 4, &fbuffer);
_Q void FSave( float val, int row, int col, FORTH_ARRAY* array_ptr)
        {       FARRAYSTORE(val, row, col, array_ptr};
        }
FSave(5.25, 1, 2, &fbuffer);
```

Now all of the following interactive commands typed from the terminal will result in the number "5.25" being printed by QED-Forth:

```
FLOAT 5.25 PrintFP
FLOAT radius PrintFP
FLOAT fbuffer[ 1, 2] PrintFP
```

Moreover, to initialize the element of fbuffer at row=0, col=1 to 12.34, we could interactively type:

```
FSave( FLOAT 12.34, INT 0, INT 1, fbuffer)
```

This demonstrates the use of the FLOAT keyword in specifying the input parameter types of interactive function calls; the syntax is similar to a C function prototype.

float FLOAT* ([addr] or [xaddr])

FLOAT* is a QED-Forth function that examines the next token; if it is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), INT* extracts the 16-bit pointer stored at the specified memory location, and in turn extracts the 32-bit float pointed to by the pointer.

Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static float radius = 5.6;
float* radius_ptr = &radius;
FORTH_ARRAY buffer16;
DIM(float*, 3, 4, &buffer16);
_Q void SavePtr( float* val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SavePtr(&radius, 1, 2, &buffer16);
```

Now all of the following interactive commands typed from the terminal will result in the number "5.6" being printed by QED-Forth:

```
FLOAT radius PrintFP
FLOAT* radius_ptr PrintFP
FLOAT* buffer16[ 1, 2] PrintFP
```

float FP_CtoQ (float ansi_fp_num)

Converts the ANSI/IEEE-standard formatted input floating point number into the QED-Forth floating point format. Converts denormalized input numbers to zero; that is, if the biased exponent = 0, the returned QED-formatted floating point number = zero. NAN (not a number) inputs are converted to +/- infinity depending on their sign bit. The least significant bit (lsb) of the mantissa is not rounded, resulting in up to 1 lsb error during the conversion. See the entry in the Main Glossary entry for FP_CtoQ().

float FP_QtoC (float qed_fp_number)

Converts the QED-Forth formatted input floating point format into an ANSI/IEEE-standard formatted floating point number. See the entry in the Main Glossary entry for FP_QtoC().

void ENABLE.DOWNLOAD (void)

If the download map is set, this function does nothing. If the standard map is set, this command prepares for a code download by copying pages 4, 5, and 6 to RAM, then setting the download map. When combined with ALL.TO.FLASH, this function ensures that an application program up to 96 Kbytes long compiled on pages 4, 5, and 6 is properly transferred to flash after a download. Usage: When paired with the ALL.TO.FLASH command, this function simplifies the loading of a Forth program. Place the ENABLE.DOWNLOAD command at the top of the first file to be loaded, and place ALL.TO.FLASH at the end of the last file to be loaded. This ensures proper compilation of code into RAM pages 4, 5 and 6 in the download map, followed by transfer to flash and setting of the standard map. It is also possible to put ENABLE.DOWNLOAD at the top of each source code file, and ALL.TO.FLASH at the bottom of each source file. This

technique ensures proper compilation of any given source code file during the development process. Of course, this command may also be typed at the QED-Forth prompt before code is downloaded.

void HEX (void)
Sets the number conversion base to hexadecimal. See DECIMAL. The number conversions occur when inputting numbers typed at the terminal, and when QED-Forth prints numbers to the terminal.

void INIT.VITAL.IRQS.ON.COLD (void)
Undoes the effect of the NO.VITAL.IRQ.INIT command, and causes subsequent cold restarts to perform the default action of checking the interrupt vectors for the COP, clock monitor, illegal opcode and OC2 interrupts and initializing them if they do not contain the standard interrupt service vectors. To use, simply type at your terminal:

```
INIT.VITAL.IRQS.ON.COLD
```

Implementation detail: sets location 0xAE1B in EEPROM to 0xFF.
See the entry in the Main Glossary entry for InitVitalIRQsOnCold().
Pronunciation: "init-vital-i-r-qs-on-cold

void INSTALL.REGISTER.INITS (char option, char tmsk2, char bprot, char baud)
Compiles a 7-byte sequence into the EEPROM that specifies the contents to be loaded into the "protected registers" plus the BAUD register after subsequent resets. The protected registers are those that must be initialized within 64 machine cycles after a reset; after that their contents cannot be changed. They are INIT, OPTION, TMSK2, and BPROT. The BAUD register controls the BAUD rate of the primary serial communications interface (serial1), and is included so that a user-specified baud rate can be set upon every restart. The INIT register controls the location of the on-chip RAM and the registers. This value is set to 0xB8 (on-chip RAM at 0xB000, and registers at 0x8000); other values are not compatible with QED-Forth. The contents of the other 4 registers may be specified by the user. Once INSTALL.REGISTER.INITS is executed, subsequent resets will cause $B8_H$ to be stored in INIT, byte1 in OPTION, byte2 in TMSK2, byte3 in BPROT, and byte4 in BAUD. To undo the effects of this word and return to the default contents of the protected registers use the DEFAULT.REGISTER.INITS command; see its glossary entry for a list of the default values for each of the registers.
Example of use: To set OPTION = 0x33, TMSK2 = 0x02, BPROT = 0x10, and BAUD = 0x31, type from the terminal:

```
HEX 33 02 10 31 INSTALL.REGISTER.INITS
```

Implementation detail: INSTALL.REGISTER.INITS writes the hex pattern 13 at location hex AE06 in the EEPROM. The five bytes following the pattern contain the specified contents of INIT (=$B8_H$), OPTION, TMSK2, BPROT, and BAUD, respectively.
See the entry in the Main Glossary entry for InstallRegisterInits().

int INT ([char] or [int] or [float] or [addr] or [xaddr])
INT is a QED-Forth function that examines the next token; if it is a valid integer or floating point number such as 5 or 3.2, INT simply converts it to the nearest integer. If the next token is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), INT extracts the 16-bit

contents stored at the specified memory location. INT is also used to specify the type of an input parameter when interactively calling a function.
Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static int radius = 5;
FORTH_ARRAY buffer16;
DIM(int, 3, 4, &buffer16);
_Q void SaveElement( int val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SaveElement(5, 1, 2, &buffer16);
```

Now all of the following interactive commands typed from the terminal will result in the number "5" being printed by QED-Forth:

```
INT 5 U.
INT 5.45 U.
INT radius U.
INT buffer16[ 1, 2] U.
```

Moreover, to initialize the element of buffer16 at row=0, col=1 to 1234, we could interactively type at the terminal:

```
SaveElement( INT 1234, INT 0, INT 1, buffer16)
```

This demonstrates the use of the INT keyword in specifying the input parameter types of interactive function calls; the syntax is similar to a C function prototype.

int INT* ([addr] or [xaddr])
INT* is a QED-Forth function that examines the next token; if it is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), INT* extracts the 16-bit pointer stored at the specified memory location, and in turn extracts the 16-bit integer pointed to by the pointer.
Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static int radius = 5;
int* radius_ptr = &radius;
FORTH_ARRAY buffer16;
DIM(int*, 3, 4, &buffer16);
_Q void SavePtr( int* val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SavePtr(&radius, 1, 2, &buffer16);
```

Now all of the following interactive commands typed from the terminal will result in the number "5" being printed by QED-Forth:

```
INT radius U.
INT* radius_ptr U.
INT* buffer16[ 1, 2] U.
```

long LONG([char] or [int] or [float] or [addr] or [xaddr])
LONG is a QED-Forth function that examines the next token; if it is a valid number such as 5 or 1234567 or 453.2, LONG simply converts it to the nearest 32-bit long number. If the next token is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), LONG extracts the

32-bit (long) contents stored at the specified memory location. LONG is also used to specify the type of an input parameter when interactively calling a function.
Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static long data = 56789;
FORTH_ARRAY buffer32;
DIM(long, 3, 4, &buffer32);
_Q void SaveLong( long val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SaveLong(56789, 1, 2, &buffer32);
```

Now all of the following interactive commands typed from the terminal will result in the number "56789" being printed by QED-Forth:

```
LONG 56789 D.
LONG data D.
LONG buffer32[ 1, 2] D.
```

To initialize the element of buffer32 at row=0, col=1 to 12345, we can interactively type:

```
SaveElement( LONG 12345, INT 0, INT 1, buffer32)
```

This demonstrates the use of the LONG keyword in specifying the input parameter types of interactive function calls; the syntax is similar to a C function prototype.

long LONG* ([addr] or [xaddr])

LONG* is a QED-Forth function that examines the next token; if it is a named 16-bit address (such as a variable name) or a 32-bit xaddress (such as a FORTH_ARRAY element xaddress), LONG* extracts the 16-bit pointer stored at the specified memory location, and in turn extracts the 32-bit long pointed to by the pointer.
Example of use: Assume that the following C code has been compiled and downloaded as part of a program:

```
static long data = 12345;
long* data_ptr = &data;
FORTH_ARRAY buffer16;
DIM(long*, 3, 4, &buffer16);
_Q void SavePtr( long* val, int row, int col, FORTH_ARRAY* array_ptr)
        {       ARRAYSTORE(val, row, col, array_ptr};
        }
SavePtr(&data, 1, 2, &buffer16);
```

Now all of the following interactive commands typed from the terminal will result in the number "12345" being printed by QED-Forth:

```
LONG data D.
LONG* data_ptr D.
LONG* buffer16[ 1, 2] D.
```

void MAIN (void)

Executes the main() function which is located at address 0x0000 on page 0x04. Each compiled program must contain one and only one definition of the main() function.

void NO.AUTOSTART (void)

Undoes the effect of the AUTOSTART and PRIORITY.AUTOSTART commands and attempts to ensure that the standard QED-Forth interpreter will be entered after subsequent resets. This command is typically executed interactively using QED-Forth syntax by typing from the terminal:

```
NO.AUTOSTART
```

Implementation detail: Erases the 0x1357 pattern at location 0xAE00 [put there by Autostart()] in EEPROM, and erases the 0x1357 pattern at location 0x047FFA [put there by PriorityAutostart()] in page 4 of paged memory. Note that the priority_autostart vector at 0x047FFA cannot be erased if the memory is write-protected when NO.AUTOSTART is executed. NO.AUTOSTART is invoked by the special cleanup mode.

void NO.VITAL.IRQ.INIT (void)

Writes a pattern into EEPROM so that subsequent cold restarts will not initialize the COP, clock monitor, illegal opcode, and OC2 interrupt vectors. This option is provided for programmers interested in installing their own interrupt service routines in any of these four vectors. Can be undone by INIT.VITAL.IRQS.ON.COLD. To use, simply type from the terminal:

```
NO.VITAL.IRQ.INIT
```

Implementation detail: Initializes location hex AE1B in EEPROM to contain the pattern 13. See the entry in the Main Glossary entry for NoVitalIRQInit().
Pronunciation: "no-vital-i-r-q-init"

void PAGE.TO.FLASH (int source_page)

Transfers the 32 Kbyte contents of the specified RAM source page to the parallel page in flash. If the current memory map is the "download map", then valid source pages are 4, 5, or 6, (and, if a 512K RAM is installed, pages 0x10-17). Page 4 RAM is transferred to page 1 flash, page 5 RAM is transferred to page 2 flash, page 6 RAM is transferred to page 3 flash, and pages in the range 0x10-17 are transferred to parallel flash pages in the range 0x18-1F. If the current memory map is the "standard map", then valid source pages are 1, 2, or 3 (and, if a 512K RAM is installed, pages 0x18-1F). Page 1 RAM is transferred to page 4 flash, page 2 RAM is transferred to page 5 flash, page 3 RAM is transferred to page 6 flash, and pages in the range 0x18-1F are transferred to parallel flash pages in the range 0x10-17. An "invalid input parameter" error is issued if an invalid source page is specified. A "can't program flash" error is issued if the flash cannot be programmed. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user.

void PAGE.TO.RAM (int source_page)

Transfers the 32 Kbyte contents of the specified flash source page to the parallel page in RAM. If the current memory map is the "download map", then valid source pages are 1, 2, or 3 (and, if a 512K RAM is installed, pages 0x18-1F). Page 1 flash is transferred to page 4 RAM, page 2 flash is transferred to page 5 RAM, page 3 flash is transferred to page 6 RAM, and pages in the range 0x18-1F are transferred to parallel RAM pages in the range 0x10-17. If the current memory map is the "standard map", then valid source pages are 4, 5, or 6 (and, if a 512K RAM is installed, pages 0x10-17). Page 4 flash is transferred to page 1 RAM, page 5 flash is transferred to page 2 RAM, page 6 flash is

transferred to page 3 RAM, and pages in the range 0x10-17 are transferred to parallel RAM pages in the range 0x18-1F. An "invalid input parameter" error is issued if an invalid source page is specified.

void PrintFP (float)

Prints the input ANSI-C floating point parameter using the format specified by the most recent execution of FIXED, SCIENTIFIC, or FLOATING. See the entry in the Main Glossary entry for PrintFP().

void PRIORITY.AUTOSTART (xaddr)

Expects on the data stack a 32-bit code field xaddress (xcfa) of a function. Compiles a 6-byte sequence at locations 0x7FFA-7FFF on page 4 so that upon subsequent restarts and ABORTs, the routine having the specified xcfa will be automatically executed. This allows a finished application to be automatically entered upon power up and resets. In contrast to the EEPROM-based AUTOSTART function, the PRIORITY.AUTOSTART vector is located in paged memory which is in flash memory in turnkeyed "production" boards. Thus PRIORITY.AUTOSTART facilitates the autostarting of flash-based systems. ABORT (which is called by the error handler and upon every reset or restart) checks the priority autostart vector first and executes the specified routine (if any). If no priority autostart routine is posted or if the specified routine terminates, ABORT then checks the EEPROM-based autostart vector (see AUTOSTART) and executes the specified routine (if any). If no autostart routine is posted or if the specified routine terminates, ABORT then invokes QUIT which is the QED-Forth interpreter.

Usage: We recommend that Autostart() and PriorityAutostart() be executed interactively from the QED-Forth monitor. The easiest way to do this is to use Forth syntax instead of C syntax. After your application program is completed and debugged, simply type from your terminal the command:

```
CFA.FOR MAIN PRIORITY.AUTOSTART
```

This writes a pattern into EEPROM that causes MAIN to be executed upon all subsequent resets and restarts.

Implementation detail: At location $7FFA_H$ on page 4, PRIORITY.AUTOSTART writes the pattern 1357 followed by the four byte xcfa; make sure that page 4 is not write protected when executing PRIORITY.AUTOSTART. To undo the effects of this command and return to the default startup action, make sure that page 4 is un-write-protected RAM and call NO.AUTOSTART (which clears both the priority autostart and the EEPROM-based autostart vectors). To recover from the installation of a buggy priority autostart routine if page 4 is RAM, make sure that page 4 is not write-protected and invoke use the special cleanup mode (consult the QED-Forth manual). See AUTOSTART, and see the Main Glossary entry for PriorityAutostart().

void RECEIVE.HEX (xaddr <text> --)

Accepts a download in standard Intel hex or Motorola S1 or S2 or S3 hex formats and initializes the memory locations starting at the specified xaddr accordingly. The first address specified in the <text> hex dump is stored in memory at xaddr, and all subsequent bytes are stored in QED memory preserving the relative spacing of data specified in the <text> hex dump. If the xaddress equals -1 (=0xFFFFFFFF), the download is stored in the addresses as specified in the hex dump itself. The QED paged memory is treated as a contiguous memory space; recall that the location

following 7FFF on a given page is location 0000 on the following page. Accepts empty lines. If a format or checksum error is detected, emits an 'X' character to signal the error, but does not abort. Aborts with a "Missing delimiter" message if the first character on a line is not a : or S character. Terminates when an end-of-file record is received; the final line of an Intel hex dump is

```
:00000001FF
```

and the standard final line of a Motorola hex dump is

```
S9030000FC
```

although any S7, S8, or S9 termination record will terminate reception. Motorola S0 header records are accepted and ignored. Each input text line is temporarily stored at PAD. Be sure that the PAD buffer is large enough to accommodate a full line (decimal 80 bytes or more is safe). See the glossary entries for DUMP.INTEL, DUMP.S1, and DUMP.S2 for descriptions of Intel and Motorola hex formats.

Implementation detail: RECEIVE.HEX calculates an offset as the specified xaddr minus the first address specified in the <text> file. This offset is then added to every byte's file address (specified in the <text> file) to calculate the QED destination address. If the specified xaddr = -1 (0xFFFFFFFF), the offset is set to zero. This scheme allows the data in a <text> hex dump file with arbitrarily reported addresses to be loaded starting at any desired location in the QED memory space.

Pronunciation: "receive-hex"

void RESTORE (void)

Restores the memory map user variables stored by the last execution of SAVE to their respective user variables. To use, simply type at your terminal:

```
RESTORE
```

See SAVE.

void SAVE (void)

Saves the current memory map so that it may be restored later. Saves the QED-Forth dictionary pointer DP, names pointer NP, variable pointer VP, last xnfa in the FORTH vocabulary, and CURRENT.HEAP in a reserved area in EEPROM (0xAE0C to 0xAE1A). RESTORE fetches these quantities and places them in the appropriate user variables to restore the saved state. Useful for dictionary management and for recovery from crashes. To use, simply type from the terminal:

```
SAVE
```

after sending a .TXT download file created by the WinEdit "Make" or "Rebuild" utilities. Consult the Debugging chapter in the "Getting Started" book for more information.

void SERIAL1.AT.STARTUP (void)

Initializes a flag in EEPROM which installs the primary serial port (serial1) as the default serial port used by the QED-Forth interpreter after each reset or restart. The serial1 port is supported by the 68HC11's on-chip hardware UART. To use, simply type at the terminal:

```
SERIAL1.AT.STARTUP
```

Implementation detail: Sets the contents of address 0xAE1D in EEPROM to 0xFF. Upon each reset or restart, the QED-Forth startup routine checks this byte, and contents of 0xFF cause the USE.SERIAL1 routine to be executed. See the entry in the Main Glossary entry for Serial1AtStartup().

Pronunciation: "serial-one-at-startup"

void SERIAL2.AT.STARTUP (uint baud_rate)

Initializes a flag in EEPROM which installs the secondary serial port (serial2) at the specified baud_rate as the default serial port used by the QED-Forth interpreter after each reset or restart. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). The specified baud rate u must a power of 2 times 75 baud up to a maximum of 9600 baud. Thus the allowed baud rates for this routine are 75, 150, 300, 600, 1200, 2400, 4800, and 9600 baud. The effect of this routine is canceled by executing SERIAL1.AT.STARTUP. Note that the serial2 port can support many more baud rates, but the options have been limited to facilitate setting a reasonable startup baud rate based on a simple implementation as described below. Note also that the maximum baud rate that can be sustained by the serial2 port is less than 9600 baud; see the glossary entry for BAUD. For example, to specify that the serial2 port is to be used at startup with a baud rate of 2400 baud, type from the terminal:

```
DECIMAL 2400 SERIAL2.AT.STARTUP
```

Be sure to modify your terminal's baud rate setting to match the new baud rate, and make sure that DIP switch#4 on the QED Board is in the ON position when using the serial2 port. See the entry in the Main Glossary entry for Serial2AtStartup().

void SET.BOOT.VECTOR (xaddr xcfa --)

Compiles a 6-byte sequence at locations 0x7FFA-0x7FFF on page 0x0C so that upon subsequent restarts and ABORTs, the function having the xcfa (execution address) will be executed BEFORE any other autostart routines are executed. The execution order at startup is: boot_vector, then priority_autostart, then autostart. Note that the “page C write protect” jumper must be removed for this function to be effective. The boot vector is most useful for extending the kernel in a "bullet-proof" way that cannot be overwritten unless the page C write protect jumper is removed. For example, suppose that you want to allow fail-safe field firmware upgrades using Compact Flash (CF) cards via Mosaic's CF Wildcard. This can be accomplished by removing the page C hardware write protect jumper, loading the CF Wildcard kernel extension on page 0x0C, and compiling a startup function on page C that checks for the presence of an "AUTOEXEC.QED" file that will be automatically executed (loaded) if present. Using SetBootVector, the startup function can be declared as a boot vector, and then the page C write protect jumper can be installed. The boot vector will be able check for the presence of a firmware upgrade file, and the hardware write protection of page C prevents the erasure of the boot vector or its code. To remove the boot vector, take off the page C write protect jumper and call ClearBootVector (CLEAR.BOOT.VECTOR in Forth), or perform a “factory cleanup”. We recommend that this function be invoked interactively from the QED-Forth prompt. Assume that a function called Page_C_Startup has been defined. Forth programmers can just execute:

```
CFA.FOR Page_C_Startup SET.BOOT.VECTOR
```

C programmers can use the *.map file generated by the C compiler to look up the compilation address and page of the Page_C_Startup function, or the function can be defined using the _Q prefix as:

```
_Q void Page_C_Startup( void ) { function body goes here }
```

Then from the QED-Forth prompt, type

```
CFA.FOR Page_C_Startup SET.BOOT.VECTOR
```

Make sure that the page containing the debug headers is included in your final runtime system.

void SP! (. . .)
Clears all items off the QED-Forth data stack and resets the data stack pointer to its default location.
Pronunciation: "s-p-store"

void STANDARD.MAP (void)
Sets a flag in EEPROM and changes the state of a hardware latch to put the standard memory map into effect. After execution of this routine, and upon each subsequent reset or restart, hex pages 4, 5, 6, and 0x10-17 are addressed in flash memory, and pages 1, 2, 3, and 0x18-1F are addressed in RAM. After code is downloaded to RAM and transferred to flash using the PAGE.TO.FLASH function, establishing the standard map allows code resident on pages 4, 5 and 6 (and pages 0x10-17) to be executed. To establish the download memory map, see the glossary entry for DOWNLOAD.MAP. Note that the standard map is active after a "factory cleanup" operation.

void STANDARD.RESET (void)
Undoes the effect of the COLD.ON.RESET command so that subsequent resets will result in the standard warm-or-cold startup sequence. To use, simply type at the terminal:

```
STANDARD.RESET
```

Implementation detail: sets the flag at location 0xAE1C in EEPROM to 0xFF.
See the entry in the Main Glossary entry for StandardReset().

int TO.FLASH (xaddr source, xaddr destination, uint numbytes)
Transfers numbytes (0 <= numbytes <= 65,535) starting at the specified source extended address, to the specified destination extended address in flash. The source may be anywhere in memory; it may even be in the flash which is being programmed. The destination must be in flash. Returns a flag equal to -1 if the programming was successful, or 0 if the programming failed. Reasons for failure include write protected flash (e.g., attempting to program page 0x0C while the page C write protect jumper is installed), or a destination that is not in a programmable page in flash memory. (If any locations in the flash are programmed more than 10,000 times, the cell may wear out causing a failure flag to be returned). Assuming that the standard 512 Kbyte flash is present on the board, writable flash pages include pages hex 4, 5, 6, 7, 0xC, 0xD, and 0x10-17 in the standard map, and pages 1, 2, 3, 7, 0xC, 0xD, and 0x18-1F in the download memory map. This function uses the 68HC11's on-chip RAM at hex B200 to B3CF to manage the write to the flash (the real-time clock and C/Forth interrupt stack reserve the bytes at B3D0 to B3FF). The remaining on-chip RAM at B000 to B1FF remains available to the user. Caution: the prolonged disabling of interrupts by TO.FLASH can adversely affect real-time servicing of interrupts including those associated with the secondary serial line. See PAGE.TO.FLASH and ALL.TO.FLASH.

void U. (int)
Prints the input parameter as an unsigned integer. Number conversion is performed in the current number base set by the most recent execution of DECIMAL or HEX.
Pronunciation: "u-dot"

void USE.SERIAL1 (void)

Installs the primary serial port (serial1) as the serial link called by Emit(), AskKey(), and Key(). The serial1 port is associated with the 68HC11's on-chip hardware UART. Stores the xcfa of Key1() in UKEY, the xcfa of AskKey1() in UASK_KEY, and the xcfa of Emit1() in UEMIT. Thus the vectored routines Key(), AskKey(), and Emit() will automatically execute the serial1 routines Key1(), AskKey1(), and Emit1() respectively. Initializes the resource variable SERIAL1.RESOURCE to zero, and initializes the resource variable associated with the prior serial channel in use (typically either SERIAL1.RESOURCE or SERIAL2.RESOURCE) to zero. Does not disable the serial2 port. To use, simply type at the terminal:

USE.SERIAL1

See the entry in the Main Glossary entry for UseSerial1().

void USE.SERIAL2 (void)

Installs the secondary serial port (serial2) as the serial link called by Emit(), AskKey(), and Key(), calls INIT.SERIAL2 to initialize the serial2 port, and globally enables interrupts to allow the serial2 port to operate. The serial2 port is supported by QED-Forth's software UART using hardware pins PA3 (input) and PA4 (output). Stores the xcfa of Key2() in UKEY, the xcfa of AskKey2() in UASK_KEY, and the xcfa of Emit2() in UEMIT. Thus the vectored routines Key(), AskKey(), and Emit() will automatically execute the serial2 routines Key2(), AskKey2(), and Emit2() respectively. Initializes the resource variable SERIAL2.RESOURCE to zero, and initializes the resource variable associated with the prior serial channel in use (typically either SERIAL1.RESOURCE or SERIAL2.RESOURCE) to zero. Does not disable the serial1 port. To use, simply type at the terminal:

USE.SERIAL2

See BAUD2, and see the entry in the Main Glossary entry for UseSerial2().

void WARM (void)

Restarts the QED-Forth system and clears the data and return stacks and executes ABORT. Unlike COLD, WARM does not initialize all of the user variables to their default values. To use, simply type at the terminal:

WARM

See the entry in the Main Glossary entry for Warm().

int WHICH.MAP (void)

Returns a 0 if the current memory map is the "standard map", and returns a 1 if the current map is the "download map". If the standard map is active, pages 4, 5, and 6 and 0x10-0x17 are addressed as flash, and pages 1, 2, and 3 (and pages 0x18-0x1F, if present) are addressed in as RAM. If the download map is active, pages 4, 5, and 6 (and 0x10-0x17, if present) are addressed as RAM, and pages 1, 2, and 3 and pages 0x18-0x1F are addressed as flash memory. This routine allows a user or program to verify which map is currently being used. After a "factory cleanup" operation, the standard map is active. See STANDARD.MAP and DOWNLOAD.MAP.

void WORDS (void)

Prints all words in the CURRENT QED-Forth vocabulary; this can be a useful way of reminding yourself of which recently defined QED-Forth function names can be

interactively typed. WORDS incorporates PAUSE.ON.KEY, so the printout can be terminated by typing a carriage return or . (dot); it can be suspended and resumed by typing other characters, and it responds to XON/XOFF handshaking (see PAUSE.ON.KEY). Each word is printed left justified in a field of 16 or 32 characters, 3 names per line. Characters that are not saved in the headers are represented by the appropriate number of _ characters. To use, simply type at the terminal:

```
WORDS
```

and then type an additional carriage return to stop the printout when you have seen enough.