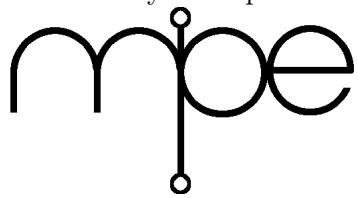
C to Forth compiler

Microprocessor Engineering Limited

C to Forth compiler

C to Forth compiler

Published by Microprocessor Engineering



User manual
Manual revision 1.2
4 January 2013

Software
Software version 1.2

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 C2Forth - Compiling C for a Forth System

With the development of MPE's VFX Forth family of compilers that produce optimised native code from high level Forth, an experiment was performed to use high level Forth as the assembly language of a C compiler. The C2Forth system is the result.

The C2Forth system allows you to compile C source to be run VFX Forth and XC6/7 cross compilers. It consists of a tool chain that produces Forth source from C source, and a build harness for the Forth System.

1.1 Overview

Traditional C compiler chains usually have four stages.

`PreProcess`
: This step takes a C source file and parses it and any included header files to produce a single reference source. Macros are expanded at this stage and conditional compilation structures are evaluated to remove irrelevant information.

`Compile`
: This step takes to output from the preprocess stage and produces an assembler listing for the target CPU.

`Assemble`
: The assembler source from the compile stage is assembled into a binary form along with any relocation information, symbol tables and missing dependancy lists.

`Link`
: All the binaries for a project (as output from the Assemble stage) are merged along with any pre-compiled libraries. Dependant symbols from an object are resolved with exported symbols from the surrounding binaries and libraries to form a stand-alone image.

MPE has developed a C tool chain which targets a virtual machine rather than a specific processor. The compiler is based on the LCC system with the preprocess/assembler/linker derived from other sources.

The MPE virtual machine design is a dual stack, no register architecture design which is similar to a conventional Forth virtual machine. The virtual machine runtime is written in Forth.

1.1.1 C Preprocessor

The preprocessor used comes as part of the LCC toolchain. Only detail changes have been made to this code, mostly to aid compilation.

1.1.2 C Compiler

The compiler is a heavily modified derivative of LCC. The LCC system is designed such that all the processing of C source into DAG lists as well as tree reduction and analysis are common. Only the part of the code which translates the final DAG trees into assembler need retargeting. The "back-end" which performs this job is usually built with lBurg.

lBurg is a code-generator generator system developed by the same authors as LCC. Unfortunately the LCC and lBurg combination was never designed to cope with the kind of architecture our dual stack VM used. For that reason it became necessary to replace the "normal" back-end interface in LCC with our own.

1.1.3 Assembler and Linker

A VFX Forth Kernel replaces the more traditional Assembler and Linker. The output from the compiler for each file in the project is passed into the Forth Kernel which has a support harness. The binary is placed directly into the Forth dictionary and label names are linked either against each other or against the current Forth search-order. Therefore libraries and such are not linkable binaries but Forth definitions built into the system before the converted code.

1.2 Current Status and Goals

This tool chain has reached the stage of proving that a good Forth compiler can be used as a back-end for C. As the usage of C2Forth increases, the amount of carnal knowledge about the target Forth system will decrease. Ultimately the goal is to have output source which is 100% ANS Forth compliant. This is theoretically possible to achieve - just difficult.

No attempt has been made to either implement a harness or port the existing one onto anything other than an MPE VFX Forth system.

1.3 Contents of the distribution

```
\BIN            C compiler binaries
    CPP         C Preprocessor
    FCC         C Compiler
    OMAKE       Make utility
    C2FPOST     Parsing utility to take FCC output and pad
                it into a form which can be safely TFTP'd
                in 512byte packets.
\DOC            C2F Distribution documentation.
                See C2Forth.pdf
\DOC\LIBC       Documentation on sample C library implementation.
\HARNESS        C2F Forth Target Harness, reference source.
                See C2Forth.pdf for details.
\INCLUDE        C Header files for sample C library implementation.
\LIB            Forth source for C library functions.
\SRC            C source code for tools.
\TESTS          Some simple test programs and a suitable makefile
```

2 Limitations

You would have to go a long way to find two langauges so far apart as C and Forth so obviously this system has some limitations.

2.1 Short Data Types

Some C programmers are used to the data type SHORT on a 32bit system being 16 bits. The current C compiler tool treats all SHORTs as INTs.

2.2 Floating Point

There is absolutly no support in this system for floating point numbers at all.

2.3 Namespace Visibility and Linking

At the moment the runtime code on the Forth kernel makes no attempt to obey the C EXTERN directive. All public labels are built in the global name space, i.e. Forth's current)vocabulary. Also since the delivery mechanism for this system is source code to an open Forth, there is no binary library format to link against.

2.4 Standard C Library

Due to the lack of a binary library format there is no method of compiling C libraries from C source code and storing the object code for linking. C2Forth has been designed to link C to Forth; since the Forth search-order is used to resolve symbols during compilation it is possible to write library functions in Forth(with a preceeding underscore in the name) and call that definition from the C code by name. (You will need a header file and prototype for the C compiler.)C2Forth contains a very limited subset of the C standard library as Forth source code along with C header files.

2.5 Target Forth Kernel

This system has been designed to exploit the optimising Forth kernels built by MPE. As such some tradeoffs have been made between portability of output code and speed.

The VFX kernels from MPE all have the following characteristics:

- 32 bit CELL, 8 bit CHAR.
- Native Code Generation.
- An XT (or CFA) is an executable address not some other id.
- Return Stack access is permitted outside of the current definition.
- Headerless definitions supported.
- MPE standard local variable code present.

There is no technical reason why the required kernel support cannot be built on any Forth System. The next section describes the output form of the Forth source and the functionality required above and beyond ANS Forth. The harness/kernel file supplied is for our VFX targets and as such, the nearer a Forth Kernel is to our specification, the easier it will be to port the existing harness.

3 Forth Style Output from Compiler

This section describes the format of the compiler output which is to be built on a Forth System. In many ways the output is Forth but with certain necessary extensions. The local variable format may vary between versions of the C compiler and Forth target.

3.1 Simple C program

```
void main( void )
{
        int a;
        a = 0;
        while( a < 26 )
        {
                putchar( a+65 );
                a++;
        }
        printf(("\n"));
}
```

3.2 Output from the C to Forth Compiler

```
PUBLIC _main
FLABEL _main
:NONAME

        MAKELVS [ #0 w, #1 w, ]
        #0

        [ #-4 lv, ]
        !
        CompileBranch @3
LABEL  @2

        [ #-4 lv, ]
        @
        #65
        +
        CompileCall _putchar
        drop
        #1

        [ #-4 lv, ]
        +!
LABEL  @3

        [ #-4 lv, ]
        @
        #26
        <
        CompileCondBranch @2
        CompileAddrLit @5 #0
        CompileCall _printf
        drop
LABEL  @1

        RELLVS
; drop

EXTERN _printf
EXTERN _putchar
LABEL  @5
                $0A c,
                $00 c,
```

3.3 Binary Generated on VFX Forth

```
Target    Assembler/Binary Data Compiled              Correlation To
Address                                               Original C Source


00469ED6  call    MAKELVS                             void main( void )
00469EDB    Inline data16: 0000                       {
00469EDD    Inline data16: 0001                         int a;

00469EDF  mov     dword Ptr [edi+-04], 00000000         a = 0;

00469EE6  call    LIT                                   while ( a<26 )
00469EEB    Inline data32: 00469F14 ( Label @3 )
00469EEF  push    ebx
00469EF0  mov     ebx, [ebp]
00469EF3  lea     ebp, [ebp+04]
00469EF6  ret

Label @2  mov     edx, [Edi+-04]                        putchar( a+65 )
00469EFA  add     edx, 41
00469EFD  lea     ebp, [ebp+-04]
00469F00  mov     [ebp], ebx
00469F03  mov     ebx, edx
00469F05  call    _PUTCHAR
00469F0A  add     [edi+-04], 01                         a++;
00469F0E  mov     ebx, [ebp]
00469F11  lea     ebp, [edp+04]

Label @3  cmp     [edi+-04], 1a                         .... Test code for
00469F18  jnl/ge  00469F24                              .... WHILE clause
00469F1E  push    00469EF7 ( Label @2 )
00469F23  ret

00469F24  call    LIT                                   printf("\n");
00469F29    Inline data32: 00469F3E ( Label @5 )
00469F2D  call    _PRINTF
00469F32  mov     ebx, [ebp]
00469F35  lea     ebp, [ebp+04]

00469F38  call    RELLVS                              }
00469F3D  ret

Label @5    Inline data8: 0                           .... String Literal
00469F3F    Inline data8: 0
```

The output is largely Forth code with some non-standard words being used. These "strange" definitions are Forth compiler directives built in a harness file on top of the Forth Kernel by exploiting Forth's ease of extensibility.

The compiler aids are split into 4 groups: Label handling, Control flow, data address literals and local frame access. Both the flow control directives and the data address literal directives support forward referencing by name and offset.

Most "traditional" C compilers produce code from top to bottom and assume a back-end assembler which can do 2 pass compilation. The Forth system cannot easily be coerced into a dual pass compiler so any attempt to reference an unresolved label adds a record into a forward reference chain, and generates a form of binary whereby the address can be patched up later. For this reason there are "magic" compiler directives for anything which may be formed by using a label which could be forward references.

3.4 Code Generation Extensions

The following definitions are used to provide required compilation behaviour not standard in Forth. All definitions for data addresses and flow control are `IMMEDIATE`.

For details of how to implement these directives as well as formal stack comments and operation descriptions, please see the section describing the current reference harness.

3.4.1 Label Creation

There are four compiler directives which create and modify labels. All branch targets and data addresses have a label.

`PUBLIC` Parses a <name> and creates a label entry with no specified type or address.

`EXTERN` Currently not used since C2F assumes all labels are global and anything not found in the current C source will be searched for in the Forth dictionary.

`FLABEL` Used to define an address for a function entry point. FLABEL will either patch the address and type of an already defined PUBLIC label (creating a public function label), or will create a new code label for the current address. Note that the address stored by FLABEL is not the current value of HERE, but the value HERE will be at after executing a :NONAME. IE, the label address stored will be the XT of the following headerless code.

`LABEL` Behaves as FLABEL for internal branch targets and data addresses. The value stored will be the value of HERE. Therefore LABEL cannot be used for function entry points.

3.4.2 Local Frame Access

Local frame access is performed by exploiting the implementation of local variables which is common to all MPE targets. For the purposes of the C2F compiler we need three definitions:

`MAKELVS` This definition builds a local frame of a specified size and moves any parameters from the data stack into the local frame. The frame itself is usually created by "dropping" the return stack pointer. MAKELVS is always followed by a 32 bit number where the high 16 bits represent the number of arguments and the low 16 bits represent the size in bytes of the local data space.

`RELLVS` This definition will release the last allocated locals frame and restore the return stack pointer.

`LV,` An internal definition which when executed will take a literal frame offset value and compile the code necessary to put the address of the local frame + offset onto the data stack at runtime.

The local variable format used was designed during the SENDIT and PRACTICAL projects. The frame is built on the return stack.

```
argn
...        paramteters
arg1
----
previous return stack pointer (RSP)
previous locals pointer (LP)
---- new LP points to old LP
uninitialised locals
---- new RSP points here
```

In an environment with interrupts, the code laid or performed by `RELLVS` must restore LP before RSP.

3.4.3 Data Address Literals

There are two definitions which are used whenever a data address literal reference needs to be compiled.

`CompileAddrLit`
Compiles code which will place the address of the label specified on the top of the data stack at runtime.

`CommaAddrLit`
Places the address of the specified label into the next cell in the dictionary.

Both these directives have the same argument syntax. The first token is the label name, and the second token is a signed 32bit offset literal. Forward referencing is built into these two compilers such that, if the label has allready been resolved the address is used directly either as a literal or via , (comma). If the label has not yet been defined or resolved then code generation is completed assuming a value of 0 and a forward reference entry is generated for later resolution. Any generated code which requires a forward reference should be built in such a way that it can have its value changed later. This is implementation specific.

3.4.4 Control Flow

There are three definitions which are used to perform branches.

`CompileCall`
Lay code to CALL the address represented by the given label. If the address is allready known, this is a COMPILE, otherwise some code is compiled whereby the XT is built into the dictionary as a CELL which can be updated when the address for the label has been resolved.

`CompileBranch`
Similar to CompileCall except it behaves as an assembler JUMP instruction. There is no direct Forth equivilant to this operation, on a number of systems it can be

acheived by performing calling a definition which performs >R. This is very implementation specific but common.

`CompileCondBranch`
: Compiles code to perform a branch if top-of-stack is non-zero. Some systems may have the word ?BRAN for this operation, other more ANS compliant targets can define this as using POSTPONE IF POSTPONE COMPILEBRANCH POSTPONE THEN

4 VFX Forth Runtime Harness

C2F_CORE.FTH provides a reference source for the Forth compiler support harness required to use C2F on the MPE VFX Forth Kernels.

This harness provides all the directives which occur in the compiler output source to control forward referencing etc.

Documentation here is in source-code order and should be used as a roadmap when reading the reference code.

4.1 Misc Configurations

```
variable <periodic-resolve>     \ -- addr
```
This variable is used to force the system to attempt to resolve pending forward references each time the PUBLIC directive is encountered. Performing this extra pass on the resolve (rather than the default setting of only resolving at the end of a source) will reduce the amount of heap memory used at compile time, but will increase compilation time.

```
: periodic-resolve?     \ -- flag
```
Returns TRUE if periodic address resolution is required.

4.2 CPU Specific Tools

These words and utilities are used to generate target code. They are isolated here since the required settings and actions may vary from host to host.

These primitives assume an MPE VFX Forth target.

```
5 constant size-of-machine-call
```
The size in address-units of a target call instruction. Primarily used by 'literal'

```
: get-parent-cfa        \ xt -- addr:32
```
When passed the xt of a definition which is a child of a defining word, this returns the xt of the runtime action's(parent. Used in the CONSTANT-XT? definition.

```
: NonameGap     \ loc -- xt
```
This primitive will return the XT that would be generated if :NONAME was invoked whilst HERE was at loc. It is used by the system to record function entry points.

```
code lp@        \ -- *lp
```
This definition returns the local frame pointer. The format of the local frame is defined in the MPE Locals Section of this text.

```
: flush-opt     \ --
```
This directive forces the MPE code optimiser to flush any pending code trees.

```
: [literal]     \ n -- ;
```
This directive lays down the target code necessary to generate the literal value N at runtime.

```
: 'literal'     \ dp n --
```
This directive can change the literal value compiled by [literal] when HERE was at DP. It is used to resolve forward referenced data address literals.

```
: CONSTANT-XT?          \ xt -- flag
```

Return TRUE if XT is a child of CONSTANT. Used when CONSTANTS are defined for addresses.

4.3 Multitasking and PAUSE

The code in this section can be used with Forth systems that have a cooperative multitasker called by the word `PAUSE`. When the C2F compiler lays a call to a known XT via `COMPILECALL` this code is used. If the current value of `HERE` is between the variables `COMP-PAUSE-START` and `COMP-PAUSE-END`, a call to the `PAUSE` is compiled before the normal call.

This arrangement allows you to specifiy a range of the dictionary for which target calls will cause a schedule operation. By default the range is 0 - 0 which turns `PAUSE` generation off. The best settings are usually between the end of the normal Forth kernel and the start of the C compiled source. This will make calls to the larger library definitions have a `PAUSE` but calls to the kernel primitives will not.

`: Set-Comp-Pause-Range  \ start end --`
This definition sets the range of addresses in the dictionary for which a compiled word will be preceeded by `PAUSE`.

`: ?compile-pause        \ where --`
Compiles a `PAUSE` if the specified address is within the range set by `SET-COMP-PAUSE-RANGE`. Note that if there is no multitasker present, nothing is compiled by this definition.

4.4 Tools

`: cksum         \ addr len -- cksum`
The INET Check sum algorithm. Used to provide a checksum of generated code to ensure delivery.

`: icompare      \ c-addr1 u1 c-addr2 u2 -- flag`
Case insensitive version of the ANS word `COMPARE`.

`: upc           \ v -- 'v`
Perform where possible an upper case character conversion.

`: zstrlen       \ addr -- len`
Given a pointer to a zero-terminated string this will return the character length excluding the terminator.

`: zcount        \ zaddr -- zaddr len`
A zero terminated string version of COUNT.

`: >pos          \ n --`
Used in diagnostic display. Move the cursor X position to column N.

`: .dword        \ n --`
Used in diagnostic display. Show the value N as an unsigned 32 bit hexadecimal.

`: c@s           \ addr -- cell`
Fetch character from ADDR and sign extend to a CELL.

`: allot&erase   \ n --`
Allot N bytes from the dictionary and pre-erase it.

`: char>cell             \ signed-byte -- signed-int`
Sign extend the supplied character to a CELL.

4.5 Code Stream Handlers

The **codestream** is an area into which strings are built to `EVALUATE` in order to perform compilation.

`256 buffer: CodeStream`
The temporary buffer into which a string is built.

`: ResetStream \ --`
Re-initialises the `CodeStream` buffer to zero length.

`: EvalStream \ --`
Perform an `EVALUATE` on the code stream.

`: $>stream \ c-addr u --`
Append a string to the code stream buffer.

`: h>stream \ n --`
Add a string to the end of the code stream buffer which represents the hexadecimal number N. The number text is preceeded by a $ character as per MPE practice.

4.6 LABEL Chain Handlers

During compilation of a converted C source file, all labels defined are kept in a linked list. Each list entry holds the name of the label, the address of the labels target if known and a set of flags which describe the label useage. All data pointers and branch targets have a corresponding label.

Label Struct Format - all entries are 1 cell and in order)

`LBL.link` link, pointer to next label record in list or NULL

`LBL.*name`
Pointer to label name which is a counted string

`LBL.flags`
Flags, describe the label type, see Flags

`LBL.address`
Address, the target address the label equates to

Label Flags consists of one or more of:

`LF_PUBLIC`
Label scope is global.

`LF_ADDRESSVALID`
The Address field of the label structure is valid.

`LF_DATALABEL`
The label represents an internal address. This is either a data pointer, or a function local label.

`LF_CODELABEL`
The label represents a function entry point.

`LF_DELETE`
Used internally. The label is marked for deletion

```
0 Value LabelChain
```

Either NULL or a pointer to the first label structure.

```
: .label                \ *LABEL --
```

Given a pointer to a label structure this definition will output a human readable dump of the label information.

```
: .l                    \ --
```

Lists all currently defined labels and their attributes. The output for each label will be the name, address and flags status as according to the header displayed on screen.

```
: DestroyLabelChain     \ --
```

Destroy the entire current label chain.

```
: AddLabel              \ c-addr u flags address --
```

Add a new label entry using the name flags and address supplied.

```
: FindLabel     \ c-addr u -- *LABEL | NULL
```

Look for a label given a name. Returns either a label structure pointer or NULL if the label was not found. Please note that under VFX Forth label names are not case sensitive.

```
: FindResolvedLabel     \ c-addr u -- c-addr u 0 | address true
```

Similar to FindLabel except only a label which has a valid address is accepted. On success the label address and TRUE is returned, if the label does not exist or has not yet been resolved the original name and namelen parameters are returned with a FALSE flag.

```
: GarbageCollectLBL     \ --
```

Walks the label list removing and deallocating any structures marked for delete.

```
: ExportLBL             \ *lbl --
```

Exports a label from a label chain structure supplied into the current Forth definitions wordlist. Code labels are built by generating a : definition of the same name which will pass the compiled codes address to `EXECUTE`. Non-Code labels are built as `CONSTANT`s.

```
: PatchLabel            \ address iflag *LABEL --
```

Modify the supplied label structure to use the supplied address and combine the supplied flags with the existing ones. This is used to resolve a label which had no valid address when defined. A fatal error has occured if any attempt is made to patch a label which already has a valid target address.

```
: NewLabel              \ address iflag c-addr u --
```

This definition is similar to `AddLabel` except the `LF_ADDRESSVALID` flag is automatically added. Used as a shortcut for those times when a known address label is to be added.

```
: (label)               \ address iflag "name" --
```

The internal factor used to create most labels. A label name is parsed from the input buffer and if it exists the address and *iflag* are passed to `PatchLabel`. If the label does not exist it is created by `NewLabel`.

```
: RemoveLocalLabels     \ --
```

Destroys the local label list by marking each entry for deletion and then calling calling `GarbageCollectLBL`.

```
: ExportPublics         \ --
```

Exports all public labels to the Forth namespace. Works by calling `EXPORTLBL` for each local label entry with the public attribute and then marking it for delete.

4.7 Forward Reference Chain Handlers

Address literals and branch targets are referenced by symbol name during compilation. Since C generated code may forward reference such symbols a chain of unresolved targets is kept during the "C" build. Each entry in the chain holds the symbol name, a numeric offset to apply to the symbol, the dictionary pointer for the target code to patch and the type of patch to apply.

```
$00000001 constant FF_CELL      \ -- x
```

This flag when set in the FW.flags field indicates the target code to be patched is a CELL, otherwise the code to be patched is the literal code as generated by [literal].
The FW(Forward reference struct has the following fields:

`FW.link` link, pointer to next record in list or NULL

`FW.*name` Pointer to label name which is a counted string. A zero in this field indicates invalid FW and the records is removed on the next garbage collect.

`FW.here` Target dictionary location to perform patch operation.

`FW.offset`
Offset to apply to address of symbol in *name before patching dictionary.

`FW.flags` Flags, describe the patch type. Either 0 or FF_CELL.

```
0 Value FWChain \ -- addr|0
```

Either NULL or pointer to the first forward reference record.

```
: .fw                   \ *FW --
```

Given a pointer to a FW structure display it in human readable form.

```
: .f                    \ --
```

Display list of all pending forward references, their target location and type of patch to apply.

```
: DestroyFWChain        \ --
```

Destroy the entire forward reference chain.

```
: $AddLitFW             \ c-addr u -- *FW
```

Add a new forward reference record for symbol whose name is passed as *C-ADDR U*. Returns a pointer to the record to allow the addresses and flags to be set by the caller.

```
: Resolve               \ *FW -- res?
```

If possible resolve the forward reference whose structure is at **FW*. Returns TRUE if the reference was found or false if the name does not exist or the address has not been resolved yet.

```
: GarbageCollectFW      \ --
```

Destroy any forward reference records with the FW.*name field at zero.

```
: ResolveFW             \ --
```

Attempt to resolve all forward references and delete any which have succesfully been resolved.

4.8 Compilers

Provides the harness definitions for the compilation of address literals and branches. Each compilation directive will attempt to resolve the required address and generate optimial code, for an unresolved address and forward reference is generated and some less optimal but patchable code is generated.

```
#256 buffer: name-store \ -- addr
```

A buffer to store the first token after a directive, i.e. the target label name. Performed since VFX parses text at `HERE` which will be corrupted during code generation.

`: >name-store \ c-addr u -- 'c-addr u`
Copy the string *C-ADDR U* to the name-store buffer for use later.

`: ParseNumericOffset \ "offset" -- val`
Parse the next token from the input stream and attempt to convert to a signed numeric offset. `ABORT`s if the next token is not numeric. This is used to parse and set the address offset as specified by directives such as `COMMAADDRLIT`.

`: CommaAddr \ "symbol" "offset" --`
Generate code to place the address of "symbol" with "offset" into the dictionary as a CELL via comma. If the address is already known it is simply passed to comma, otherwise a 0 value is compiled and a FF_CELL forward reference added to the FW chain.

`: CompileAddrLit \ "symbol" "offset" --`
Similar to `CommaAddr` except rather than laying the data as a CELL in the dictionary the runtime code for a literal is compiled. As with `CommaAddr` if the label is resolved the literal is used directly (as though it was present in the source) otherwise a patchable literal is compiled via `[LITERAL]` and a forward reference is added.

`: CompileCall \ "symbol" --`
Compile a call to the address named "symbol". If the address is known it is passed to `COMPILE,` otherwise "symbol" is treated as with `COMPILEADDRLIT` and a call to the Forth `EXECUTE` is compiled afterwards. Also note that when compiling a resolved symbol the target address is passed to `?COMPILE-PAUSE` as documented above in the section on multitasking.

`: CompileJump \ --`
Compile code into target to perform a JUMP to the address in TOS.

`: CompileBranch \ "symbol" --`
Compile a jump the address reference by "symbol". This is acheived by compiling a literal as with CompileAddrLit and then calling `COMPILEJUMP` to lay the branch code.

`: CompileCondBranch \ "symbol" --`
Lays code for a conditional variant of `COMPILEBRANCH`. The normal method of achieving this is to postpone IF COMPILEBRANCH THEN.

4.9 Label Definitions

These definitions form part of the language extensions as used by the output from the c2forth translator. They each record a specific kind of label.

`: FLabel \ "name" --`
Adds a new code-entry point label. The actual address represented is the target XT address which will be returned if `:NONAME` is invoked at the current value of `HERE`. If the label already exists it has the `LF_CODELABEL` flag added to its current incarnation.

`: Label \ "name" --`
Adds a new non-entry point label. These labels are either reference points within a definition, e.g. branch targets, or are pointers to data items. As with `FLabel`, an existing label is patched. This time with the `LF_DATALABEL` flag.

`: Public \ "name" --`
Add a new label of type `LF_PUBLIC`. This operation allways adds a new label, and does not specify a type. It will usually be immediately followed by a `FLABEL` or `LABEL` directive to specify type.

```
: Extern                \ "name" --
```
EXTERN <label> will eventually be used to mark external labels. Currently C2F ignores these directives and everything is treated as global.

4.10 Variadic Function Support

The calling method for variadic functions involves placing the known stack depth into the first local variable, building a frame with the specified parameters and using an equivalent to Forth's PICK instruction to grab the variadic parameters. These three definitions are used at runtime to support variadics.

```
: _vm_va_depth          \ -- i
```
Return the contents of the first local variable.)

```
: _VM_depth             \ -- n
```
Return data stack depth. A sanity check, how far down the data stack can a variadic function safely perform PICK.)

```
: _VM_pick              \ n -- c
```
An alias for PICK to complete the isolation layer.

4.11 Outer Harness

The definitions used to control the compilation of C files within a Forth interpreter.

```
variable        c-build-start-dp
```
A variable which records the value of HERE when C compile mode is enabled. Simply used to report the size of generated target code on completion.

```
: [EOF]                 \ --
```
This definition is invoked at the end of each individual source include during a C build. It resolves forward references where possible, deletes the source-file local labels and exports public symbols to the Forth namespace.

```
: "C"                   \ --
```
Begin "C" source code build. Records start HERE position and ensures the local label and forward reference chain are clear.

```
: "FORTH"               \ --
```
End a "C" source code build. Reports unresolved forward references and size/checksum of generated code.

```
: #included             \ c-addr u --
```
A version of the ANS word INCLUDED which automatically calls [EOF] on completion. Used to include each individual C file between the "C" and "FORTH" directives.

```
: #include              \ "name" --
```
A version of the ANS INCLUDE word. Performs as #INCLUDED.

5 Libraries and Calling Forth

Since there is no librarian tool or intermediate binary in the C2Forth system it is necessary for some other mechanism for declaring C library functions such as **printf()** etc.

The C2F system allows code output from the C compiler to reference symbols in the Forth dictionary. This means that the standard C libraries and user functions can be coded in Forth and built into the system before the C build.

Supplied with the kit is a minimal implementation of some of the more useful standard C libs.

5.1 Coding Library Functions in Forth

As an example, here is the library implementation of the standard C library function **malloc()**.

First the function prototype for the C compiler is placed in the header file *stdlib.h*. It has the form:-

```
extern void *malloc( size_t s );
```

In a Forth world this can be seen as:-

```
: malloc        \ size_t -- void*
```

Therefore a suitable definition for `malloc` in Forth would be:-

```
: _malloc       \ size_t -- void*
  allocate
  if drop 0 then
;
```

There are three key things to remember about Forth definitions for C library calls.

1. Each function must have a prototype in a C header file.
2. Names in Forth must have a preceeding underscore character.
3. Input parameters are in reverse order to the C prototype.

5.2 Variadic Functions

Implementing a variadic function in Forth is slightly more complex. With a variadic function you have a fixed formal parameter list, followed by N other items.

In Forth you know the fixed parameter list, the variadic members should be pulled off the data stack using PICK as required. Note that the Forth definition is responsible for keeping the overall stack effect of the C prototype for FORMAL PARAMETERS ONLY. The Forth definition should not `DROP` the variadic members.

Please examine the Forth definition for **_SPRINTF()** in the *STDIO.FTH* library file.

6 C library

C2Forth provides a minimal version of the C standard library. The minimal version is enough for most embedded applications. You can extend the the library yourself in Forth or C.

6.1 ASSERT - Runtime Debugging Exception

The **ASSERT()** mechanism provides runtime error checking facilities. The parameter to **ASSERT()** is a small piece of RT code which when evaluated to 0 at runtime causes the program to halt and display the original source file and line number.

```
void assert( <expression> )
```

If <expression> evaluates to 0 (ZERO)at runtime, cause a halt.

```
: ___assert     \ failedexpr line file --
```
Runtime for a failed **assert()**.

6.2 CONIO - Extended Console IO functions

The conio library holds functions relating to console input and output which, whilst useful, are not part of the ANSI/ISO library standards.

```
int kbhit( void )
```

Returns non-zero is a key has been pressed, otherwise returns 0. This is a non-blocking "peek" operation on stdin.

```
: _kbhit        \ -- flag
```
Return true when a key has been pressed. Equivalent to the Forth word `KEY?`.

6.3 CTYPE - Character Type Query and conversion

This is an implementation of the ANSI "C" library for testing and converting individual characters in the standard character form.

6.3.1 C prototypes

```
char isalpha( char t );
```

Returns non-zero if the character t is either an uppercase or lowercase alphabetic character. ('A'..'Z' or 'a'..'z').

```
char isascii( char t );
```

Returns non-zero if the character t is within the 7 bit ASCII alphabet.

```
char isblank( char t );
```

Is the supplied character a blank?

```
char iscntrl( char t );
```

Is the supplied character a control character?

```
char isdigit( char t );
```

Is the supplied character a numeric digit?

```
char isxdigit( char t );
```

Is the supplied character a hexadecimal digit?

```
char isgraph( char t );
```

Is the supplied character a graphics character?

```
char islower( char t );
```

Is the supplied character a lower case alphabetic?

```
char isupper( char t );
```

Is the supplied character an uppercase alphabetic?

```
char isspace( char t );
```

Is the supplied character a space?

```
char ispunct( char t );
```

Is the supplied character a puncuation mark?

```
char isprint( char t );
```

Is the supplied character printable?

```
char isalnum( char t );
```

Is the supplied character an alphanumeric?

```
char tolower( char t );
```

Convert char to lower case where possible.

```
char toupper( char t );
```

Convert char to upper case where possible.

6.3.2 Forth implementation

Conversion and testing is controlled by a type table.

```
: _U    $01 c, ; immediate          \  UpperCase
: _L    $02 c, ; immediate          \  LowerCase
: _N    $04 c, ; immediate          \  Numeric
: _S    $08 c, ; immediate          \  WhiteSpace
: _P    $10 c, ; immediate          \  Punctuation
: _C    $20 c, ; immediate          \  Control
: _X    $40 c, ; immediate          \  Xdigit
: _B    $80 c, ; immediate          \  Space
: _CS   $28 c, ; immediate          \  Control or WhiteSpace
: _SB   $88 c, ; immediate          \  Space character
```

```
: _UX   $41 c, ; immediate          \  Uppercase or XDigit
: _LX   $42 c, ; immediate          \  Lowercase or XDigit

create ctype-table      \ -- addr
```

Type table for ASCII characters.

```
: [ctype]       \ char -- val
```

Return the type value of a character.

```
: _isalpha      \ char -- flag
```

Returns non-zero if the character t is either an uppercase or lowercase alphabetic character. ('A'..'Z' or 'a'..'z').

```
: _isascii      \ char -- flag
```

Returns non-zero if the character is within the 7 bit ASCII alphabet.

```
: _isblank      \ char -- flag
```

Is the supplied character a blank?

```
: _iscntrl      \ char -- flag
```

Is the supplied character a control character?

```
: _isdigit      \ char -- flag
```

Is the supplied character a numeric digit?

```
: _isxdigit     \ char -- flag
```

Is the supplied character a hexadecimal digit?

```
: _isgraph      \ char -- flag
```

Is the supplied character a graphics character?

```
: _islower      \ char -- flag
```

Is the supplied character a lower case alphabetic?

```
: _isupper      \ char -- flag
```

Is the supplied character an uppercase alphabetic?

```
: _isspace      \ char -- flag
```

Is the supplied character a space?

```
: _ispunct      \ char -- flag
```

Is the supplied character a puncuation mark?

```
: _isprint      \ char -- flag
```

Is the supplied character printable?

```
: _isalnum      \ char -- flag
```

Is the supplied character an alphanumeric?

```
: _tolower      \ char -- 'char
```

Convert char to lower case where possible.

```
: _toupper      \ char -- 'char
```

Convert char to upper case where possible.

6.4 STDIO - Standard IO Model

The standard IO model has been stripped for C2Forth. There is no file support, and the STDIO system relates to the debug serial port on the target. These functions are for DEBUG only and should not be used for runtime code since they affect performance. At runtime communication should be between the target and the host rather than the debug-console.

```
0 constant _stdout      \ -- x
```
The C **stdout** value. Change as required by your host.

```
1 constant _stdin       \ -- x
```
The C **stdin** value. Change as required by your host.

```
2 constant _stderr      \ -- x
```
The C **stderr** value. Change as required by your host.

```
: _puts         \ char* -- len
```
Write a string of text to the the debug output channel

```
  int puts( const char *text );
```

```
: _sprintf      \ .... fmt buff -- .... len
```
Write a string of text using the printf() syntax to a memory array.

```
  int sprintf( char *b, const char *fmt, ... );
```

```
: _printf      \ .... fmt -- .... len
```
Write a formated string to the debug console.

```
  int printf( const char *fmt, ... );
```

```
: _getchar      \ -- char
```
Wait for and return the next key pressed.

```
  int getchar( void );
```

```
: _putchar      \ char -- char
```
Transmit the character.

```
: _sscanf        \ ?????
```
Not implemented.

6.5 STDLIB - Standard Library

```
#32 constant #ExitChain                 \ -- u
```
Maximum number of entries in the exit chain.

```
#ExitChain cells buffer: ExitChain      \ -- addr
```
Buffer for exit chain entries.

```
0 Value ExitChainIdx                    \ -- 0|addr
```
Anchor for exit chain.

```
: _abort        \ --
```
Terminate the current program no matter what.

```
  void abort( void );
```

```
: _exit         \ n --
```
As with **abort()** except with a return status code and the fact that an exit() is not considered abnormal termination.

```
  void exit( int status );
```

```
: _atexit       \ xt -- ior
```
Specify a function which will be executed when **exit()** occurs.
```
  int atexit( void(*func)(void));
```

```
: _malloc       \ size -- ptr | 0
```
Attempt to allocate S bytes of heap and return a pointer.
```
  void *malloc( size_t s );
```

```
: _free         \ ptr --
```
Release **malloc()**ed memory.
```
  void free( void *ptr );
```

```
: _realloc      \ size ptr -- 'ptr | 0
```
Attempt to resize the allocated block P to the new size S.
```
  void *realloc( void *p, size_t s );
```

```
: _atoi         \ s -- n
```
Convert an ascii string to a decimal number.
```
  int atoi( const char *s );
```

6.6 STRING - Simple String Tools

As with all the other libraries, this one has been stripped. Sufficient functionality is provided for most programs and those functions missing can be easily written in C using the ones supplied.

```
: _memset       \ size value source -- source
```
Set memory (bytes).
```
  void *memset( void *, int c, size_t s );
```

```
: _strlen       \ z$ -- len
```
Return the length of the supplied string not including the zero terminator.
```
  size_t strlen( const char *s );
```

```
: _strcpy       \ source dest -- dest
```
Copy the string S to the char array at D. Returns the pointer to the destination.
```
  char *strcpy( char *d, const char *s );
```

```
: _memcpy       \ len source dest -- dest
```
Copy memory.
```
void *memcpy( void *d, const void *s, size_t len );
```

```
: _strtol       \ base **endptr *s -- val
```
Attempt to convert the string S to a long integer using BASE as the radix. ENDPTR points to a CHAR* pointer which will hold the address of the first part of the source string which cannot be converted as a numeric digit.
```
  long int strtol( const char *s, char **endptr, int base );
```

```
: _memcmp       \ len s1 s2 -- v
```
Compare two blocks of memory.
```
int memcmp( const void *s1, const void *s2, size_t n );
```

```
: _strcat       \ source dest -- dest
```
Append string S2 onto the char array at S1.
```
  char *strcat( char *s1, const char *s2 );
```

7 Usage with VFX Forth

This section explains how to compile the supplied test programs and build them on the ProForth VFX system using the reference harness.

7.1 Building the Test Apps

In the distributions \TESTS directory there are three sample applications.

`BANNER` Does large screen pretty printing, similar to the Unix banner program.

`TESTHEAP` Performs a stress test on the system heap.

`VARTEST` Test program for simply variadic functions.

By entering MAKE <testname> in the \TESTS directory you can build each test program. The translated source for each file will have a CXX file extension. This is the Forth style output.

MAKE itself is a batch file which calls OMAKE (in the \BIN folder) and uses the supplied .MAK file which can be used as a template for all builds.

7.2 Running the Test Apps

In order to compile and run the test programs from the Forth sources you need to perform the steps below. For Windows systems, there are batch files (scripts) that simplify the process. If you are using a Unix system, you can use these as the basis of your own scripts.

7.2.1 Windows script

In the root of the C2Forth system, you will find a batch file called *InstallWin.bat*. Run this from a console prompt at the root of the C2Forth directory.

```
C:\Products\C2ForthKit.120>InstallWin
```

The script creates apermanent environment variable called C2FORTH that specifies the root of the C2Forth directory. If you are using Windows XP, you may have to download *setx.exe* from the Microsoft web site as it is required by *InstallWin.bat*.

The **tests** directory contains *banner.bld* which can be compiled directly by VFX Forth for Windows.

7.2.2 Run VFX Forth

You need any version of VFX Forth.

7.2.3 Change to the C2Forth Folder

VFX Forth contains a command CD which behaves the same way as the Unix/Windows CD shell command. Use it to change to the folder into which you installed the C2F system. You can use DIR <cr> to verify you are in the correct directory.

7.2.4 Compile the C2F Reference Harness and Libraries

This distribution contains a sample implementation of the C2F harness and some simple C style libraries. Change into the \HARNESS folder and include the sources using:

```
include c2f_vfx.bld <CR>
```

7.2.5 Change to the \TESTS directory

On the default installation you can change to the \TESTS folder after building the harness by performing:-

```
CD ..\TESTS <cr>
```

7.2.6 Compile your first test app

Compilation of a C build is performed in three steps.

The first is to initialise the C2F harness for a new compile.

```
"C" <cr>                        Initialise C2F Harness
```

Second, include the source files which make up your project:

```
#include banner.cxx <cr>        Include banner source
```

Third, signal the end of the build and switch back to Forth.

```
"FORTH" <cr>                    Back to Forth Mode
```

If all has gone well you should see a report on the compilation. If you perform `WORDS <cr>` on the Forth system you will find that all the C procedures are available. Type _MAIN <CR> to run the program.

7.3 Rebuilding the C toolchain

If you are going to rebuild the compiler using the same tools as MPE, you will need a copy of Visual C++ 6.0. The batch file *setpath.bat*, adds the VC++ include directory to the Windows search order.

If you convert the code to use other compilers, please return your updates and scripts to MPE for the benefit of others.

Index