Yet another target compiler#

Table of Contents

- Yet another target compiler
- FORTH TARGET CODE GENERATOR
- Problems With Code Generation For Target Or Different Systems
- Normal Forth Compiler Behavior Synopsis
- Meta Compilers
- Development On Target Machines
- Multi-pass Target Compiler
- Problems Fixed And Why -
- Interactions Between The Three Passes -
- Pass 1 Description -
- Pass 2 Description -
- Pass 3 Description -
- Data Types And The Code They Generate -
- MODULES
- Forth Compiler Modifications
- Target Compiler Setup And Control
- Converting Definition To Symbolic Code
- Definitions And Data Types Logged -
- Intermediate Level Code Generation -
- PRODUCING AN APPLICATION FOR A TARGET MACHINE
- WHY YATC IS NOT JUST A FANCY DECOMPILER
- CODING TRICKS AND OTHER THINGS THAT WON'T WORK
- GLOSSARY OF EXTERNAL WORDS
- REFERENCES

Yet Another Target Compiler by Eric S. Johansson

January 6, 1987 Yet Another Target Compiler Page 2

FORTH TARGET CODE GENERATOR#

This paper describes an alternative method for generating stand-alone applications in forth. Important aspects of this new code generation method include

- freedom from the ambiguities and restrictions of meta compilers.
- using the same application source code for local and target code generation
- expanding the usefulness of forth as an applicationslanguage.
- providing an easy way to connect forth generated code to code generated with other languages.
- reduces problems with forward references

Problems With Code Generation For Target Or Different Systems#

Forth is a language strongly bound to its local environment. This has the advantage of making the compiler very simple. It also has the disadvantage of making it VERY difficult to create applications for any environment other than the local one.

To date, making forth compile to a target environment has takenone of two paths: using a meta compiler or using the target machine to create an executable image.

Before looking at compiling for target environments, let's review how forth generates code in local environments.

Normal Forth Compiler Behavior Synopsis#

The behavior of a forth compiler has been described in sufficient detail elsewhere so the following description is there to relate the original behavior to the new behavior.

Forth generates code in three different ways. They are colon definitions, immediate words and defining words.

The simplest form of code generation is a colon definition consisting of non-immediate words. In this case there exists a one-to-one correspondence between words in a forth definition and the code generated. Every forth word gets converted into an execution token and placed into the forth workspace or dictionary. An execution token is equivalent to an op-code for a forth cpu.

Things get a little more complicated when immediate words are present in a colon definition. Immediate words generate code that no longer has a one-to-one correspondence to the word used in the colon definition. A common example of immediate definitions are the conditional words (if..else..then, begin..until, etc.).

- "if" compiles a branch-if-0 token into the dictionary and compiles space for the forward branch offset.
- "else" compiles a branch-unconditional token and associated offset space into the dictionary. Then "else" changes the branch offset generated by the proper "if" to point to after the "else" compiled branch.
- "then" doesn't compile any code. It makes the most recent branch offset point to the current dictionary location.

Things get even stranger with defining words. They provide the compile time actions and run time actions for words they create. A classic example of a defining word is the constant definition.

The first part of of the constant definition executes when defining a constant. Defining a constant creates a dictionary entry consisting of the constant name and the value of the constant. The second part of the constant definition executes when the program references constant. It places the value of a constant on the stack.

Forth compilers localize code generation complexity to the definition implementing the function. The nature of the code generation complexity determines which method of code generation isappropriate for the application.

Meta Compilers#

A meta compiler is a single pass compiler written in forth, producing code for the target environment. Major problems with meta compilers revolve around the requirement for two sets of applicationsource code, one set for local compiling and the other for meta compiling. Which source code to use depends on which environment theapplication is compiled for, local or target. Two sources for one application can cause many headaches in application development and maintenance.

Several immediate and defining words create problems in meta compiling. These words include but are not limited to: program flow control constructs, "compile", "[compile]" "create", "."" (dot-quote), and "does>" . Since immediate and defining words refer to and modify the local environment, problems arises when compiling for a target environment.

The usual solution changes words referring to a local environment to refer to a target environment and this fix introduces ambiguities. As an example of these ambiguities let's look at constants.

When a normal constant is created, the constant is placed in the local environment. Placing a constant into a target environment changes the compile time definition of a constant. Now there are two ways of defining constants; one for local environments and another for target environments. And the programmer is left with a mess figuring out how to refer to constants in the target environment for use in the local environments and the reverse.

The problems and ambiguities escalate when creating immediate words. The first generation is not too tough to create. It uses words from the local environment and generates code for the target environment. A second generation word is a little tougher.

Second generation immediate words use a first generation immediate word inside of second generation word. A redefinition of the first generation word must occur before the second generation word can use it. It must be redefined as a normal forth immediate word (i.e. executes in local environment and modifies local environment). Another ambiguous mess is left for a programmer to manually deal with. A programmer can try resolving ambiguities of which immediate word was used or try maintaining two copies of an application; one written for the local environment and another written for the target environment.

Unfortunately there are as many hacks for getting around meta compilation problems as there are forth programmers.

Development On Target Machines#

The numerous problems and ambiguities that arise during meta compilation triggered the search for other solutions. Another popular method of creating applications on target machines uses that target machine as a development environment.

Creating an executable image on a target machine eliminates the source code control problems associated with meta compilation but introduces a few problems of its own. One problem involves how to create code for a different machine other than the one we are running on (i.e. generating 68000 code on an 8086). Another problem is hardware support on the target machine. The target hardware needs to support at least terminal i/o and hopefully a disk. Since the code image generated in the target is taken as-is and preserved, the entire development system is required to provide the needed run time support for all applications. Not very practical for most production environments.

What options does that leave a programmer. Either the programmer puts up with maintaining two very different source code files for one application or recreates the development environment for every target machine worked with. Either way is not very suitable for a production environment.

Multi-pass Target Compiler#

Forth applications and systems development need to take the output of the forth compiler, unbind it from the local environment and then bind it to the target environment. Why should it be so hard?

Most of the problems with meta compilation seem to be a direct result of forth not being a simple incremental compiler. A forth definition can change its contents anytime until the definition completes compiling. Immediate words and defining words make this behavior possible.

Making forth a multi-pass compiler solves problems created by immediate and defining words in a single pass compiler. Target compilation becomes much simpler. The first pass compiles code into the dictionary and keeps track of the data it compiles, the second pass unbinds the output of pass 1 from the local environment, and the third pass binds the output of pass 2 to the target environment.

Problems Fixed And Why -#

Now target code is almost as easy to create and use as local code. But questions arise: Why does it (a multi-pass compiler) work? How does it eliminate the ambiguities and restrictions present in meta compilers? How do forward references get handled? How is it easier to connect forth generated code to code generated with other languages?

This form of a target compiler works because we compile code for and in the local environment. Locally defined words are used in local definitions, local immediate words compile into local definitions, and local defining words create local definitions.

The difference is this compiler produces two outputs for every compiled definition. The two outputs are normal forth compiled code, and intermediate level code suitable for conversion into executable code for a target machine.

The intermediate level code output has a few characteristics that make it reasonably special and useful for creating target machine executable code.

- The output contains nothing but executable forth execution tokens and addresses. All of the words that cause special code generation have executed and done their work.
- Addresses are properly identified. Offsets from symbols are calculated and displayed as part of the address symbolic representation.
- Non address data is properly identified and displayed.
- Output matches pass 3 syntax

Problems with meta-compilation (source code maintenance, immediate words, and defining words) go away. They vanish because the code for the target system is based on the code compiled for and by the current system.

Forward references become easier to deal with. Simply put, all forward referenced words are compiled into the dictionary before they are needed. The target compiler is turned on and the application is compiled including the forward referenced code in its proper place.

```
: moo           <---    define forward reference before
  . . .                 compiling application program
;                       outside of the target compiling stage

far             <---    turn on target compiler.  start of
                        active pass 2 section

: woof
  . . . moo .   <---    "forward" reference
;

: moo           <---    code woof intended to reference
  . . .                 within the target compiling stage
;

near            <---    end of active pass 2 section
```

Handling forward references this way creates the definitions before they are needed and allows use of immediate and defining words before defining them within the context of the application code.

When pass 2 produces symbolic output, it creates symbolic references to symbols defined inside or outside of the active pass 2 section. The real code for forward referenced words is defined later and the proper labels are created. Forward reference symbol resolution is delayed until pass 3. If an assembler or similar toolis used for pass 3, forward references should be no problem because the

tool handles it for you. If pass 3 is user created in forth,forward referenced symbols need to be dealt with.

Linkage to code created by other programming languages becomeseasier when pass 3 is an assembler. Linkage routines can be created in assembler and translate forth's calling sequence to anotherlanguages calling sequence and back. The linkage routines are joined with the application during pass 3.

Interactions Between The Three Passes -#

Traditionally, in a multi-pass compiler, a compiler pass finishes executing before handing control to the next pass. Because forth is an incremental compiler, the three passes interact in the following fashion.

- The first pass occurs when compilation occurs. As stated previously, it is basically normal forth compilation activity with additional work of logging data types.
- The second pass occurs whenever a definition is completed.
- The third pass occurs when all definitions are compiled and production of an executable target image is desired.

The first two passes are tightly coupled and control is passed between the first pass code and the second pass code. The third pass is independent of the first two passes. It will run much after the first two passes have been completed. The third pass is analogous to a linking loader.

Deciding when to start the second pass and generate intermediate code is critical. Start second pass at the wrong time and data might be lost. The second pass must be started when a definition is complete.

Forth definitions are complete when next definition starts. At that time, all changes to the internal structure of a definition are finished. and the second pass is run on the just completed definition.

When the second pass is done, control is returned to the the first pass and next definition is compiled.

Pass 1 Description -#

The first pass of a multi-pass forth compiler is no different from a normal forth compiler except for tracking compiled data and its data type.

Forth normally ignores data types. It knows about 8 bit, 16 bit, and 32 bit integers. But it ignores the type of data associated with its storage allocations. Is that storage a pointer, character string, or literal number? Currently forth can not tell.

It is relatively easy to change forth to record data types. The changes to the compiler are minimal and are described under compiler changes. Most of the time forth correctly records data types but there are a few situations where it won't. At those times the programmer must explicitly state the type of data compiled.

Pass 2 Description -#

The second pass of the compiler "knows" things about the forth environment it is working in. It knows how the compiler generates code (indirect, direct, token, or subroutine threaded code), the dictionary header structure, the target machine structure, and the third pass syntax.

The second pass front end knows about how the compiler generates code and the dictionary header structure. This knowledge allows the front end to drive the back end properly.

The second pass back end is told the current form of the definition by the front end and "knows" how to convert the information to the correct form for the third pass. The output of the back end is a symbolic or intermediate code description of a definition.

The front end changes if the local environment changes. For example, the current pass 2 front end knows about indirect threaded code on a 6502 processor and fig-forth style headers. If any of these local environment attributes are changed, the front end must change to accommodate.

The back end changes if the target environment changes. For example, the current pass 2 back end knows about indirect threaded code and fig-forth style headers and the third pass syntax. If the target environment changes, the back end must change to accommodate them.

Pass 3 Description -#

The third pass converts the output of the second pass into an executable image. The third pass can be either a linker or a macro assembler with the proper macros.

If an assembler exists without macro capability, the second pass back end can be modified to do the macro expansion for the third pass.

Data Types And The Code They Generate -#

The data types logged are 8 bit and 16 bit integers, addresses, execution tokens, and multi-byte storage allocations.

Integers are recovered and printed as numbers. Addresses are transformed into symbols and offsets. Execution tokens are transformed into symbols. Multi-byte storage allocations are transformed into a list of 8-bit integers.

Data types that are not explicitly logged are strings and integers other than 8 and 16 bit integers (i.e. double, quad, or longer integers). Strings are considered multi-byte storage Yet Another Target Compiler Page 9

allocations. Longer integers are either recorded as multi-byte storage or as multi-integer storage. Yet Another Target Compiler Page 10

MODULES#

The changes to a forth compiler can be broken up into several distinct modules. Each one is described in detail below.

Forth Compiler Modifications#

The following definitions are created or modified for the target compiler.

- , changed to log 16-bit integers.

- c, changed to log 8-bit integers.

- a, places addresses into dictionary and logs addresses.

- e, places execution tokens into dictionary and logs tokens

- address_literal creates an embedded address constant.

- compile, interpret, and [compile] change to use e,

- back changes to use a,

- ' changes to use address_literal

- create is renamed to (create)

- define an execution vector named create

The changes can be made to the original compiler in one of two ways. Change the source code and recompile a new kernel or compile all the changes on top of the original compiler. The second method can be made to work in almost any forth system as long as enough is known about the internals of the base compiler. The first method is practical only when the source code of the compiler is available and can be used to regenerate a forth system. Examples of this are fig-forth and F83.

2.1.1 Logging Data Types -

The data type logging function has two halves. A recording half and a playback half. The recording half, "mark_it" records the data types produced by the compiler. The playback half, "get_next_mark", returns the data type and the address that data type refers to.

"mark_it" is used by ",", "c,", "a," "e," and "allot" for recording every data type in a data type log. "allot" also records the amount of storage allocated. Yet Another Target Compiler Page 11

Recording data types uses the word "mark_it". A typical use of "mark_it" follows:

: (i,) integer_mark mark_it (,)

"integer_mark" is a constant used as an integer data type marker in the data type log. "mark_it" places each data type mark in a buffer called "data_type_log". "(,)" compiles the integer into the dictionary.

The other data type marks are: "byte_mark" for logging bytes, "address_mark" for logging addresses, "execute_mark" for marking execution tokens and "storage_mark" for logging storage allocations. All data type marks are single word elements except for "storage_mark" which is a two word element. The second element is the amount of storage allocated in bytes.

Target Compiler Setup And Control#

Basic compiler control is provided. The target compiler can be started, finished and turned on and off. Dictionary headers can be left in or turned off.

start_target_compiler and "start_target_compiler setup the target compiler. "start_target_compiler takes a filename pointer as an argument and opens that file for target compiler output.

End_target_compiler shuts down the target compiler. It finishes output and closes any open files.

Near and far control the output of the target compiler. Far turns on target compiling and near turns it off. Target compiling is a "far" environment. Local compilation is a "near" environment.

Heads and tails control header production. Heads turns on header production. Tails turns off header production leaving the definition code only (the tail end).

Converting Definition To Symbolic Code#

Definitions And Data Types Logged -#

As stated earlier, every definition creates a list of data types associated with that definition. This section shows the relationship of data types logged and the code compiled. Yet Another Target Compiler Page 12

What follows is an example of a simple definition. It multiplies by two by adding the input to itself. The picture describes the compiled code generated by an ITC type of forth compiler and the contents of data type log after compilation is complete.

The data types used in this example are storage ("s"), address ("a")and execution ("e") tokens.

Storage tokens are generated every time any memory is allocated to the forth dictionary. Allot is the source of all storage tokens. Storage token is currently the only token that is a multi-byte token. The token is followed by a byte count giving the size of the storage allocation.

Address tokens are associated with compiled addresses. This example has addresses in the link field and code field only. Addresses can also be generated as part of branches, or address literals.

Execution tokens are generated for forth "instructions". In many implementations of forth, execution tokens are the same as address tokens. Maintaining a difference between address and execution tokens keeps the target compiler flexible enough to work with almost any type of forth compiler.

Tokens not used in this example are the byte and the integer tokens. They represent byte and integer storage allocations. Representing different sized integers explicitly enhances the code's ability to be independent of byte order.

2.3.1.1 Example Of A Simple Definition And Logged Data Types -

: 2* dup + ;

definition data type log

+-------!!! +--!!! header --> | 2 | | s 3 | +-------!!! ~

2	. .
*	~

+-------!!! +--!!! link --> | ptr to | | a | field | prev dfn | | s 2 | +-------!!! +--!!! code --> | ptr to | | a | field | : code | | s 2 | +-------!!! +--!!! body of--> | dup | | e | dfn | token | | s 2 | +-------!!! +--!!!

+		e	
token		s 2	

+-------!!! +--!!!

;s		e	
token		s 2	

+-------!!! +--!!!

The definition breaks into two separate parts, the header and the "executable" code.

Intermediate Level Code Generation -#

Deciding when to convert the code compiled by the first pass into a form usable by the third pass is critical. Too early or late and result is useless for target code production. The right time is when the previous definition is complete and the next has not started compiling. This point in the compilation process is when create is called.

All defining words call create to define their dictionary header. Executing pass two code before calling create will catch all forms of code definition.

Pass two code starts out "knowing" the starting address of a definition and the data types compiled by that definition. Starting at the beginning of the definition and the data type log, the pass two code looks at the data type and converts the data to the proper form for pass 3 consumption.

Actual code output from pass 2 is extremely dependent on the syntax of pass 3 therefor, any specifics of code output are based on the current implementation (6502 itc fig-forth).

Integers generate signed hex numbers, bytes generate unsigned hex numbers. For example, a literal constant generates the following code. +------!!! +--!!! ... 01F ... -- generates --> | literal | | e |

token		s 2	

+------!!! +--!!!

00		i	
1f		s 2	

+------!!! +--!!!

which generates for pass 3,

@resolve.exec zliteral,0 .word $1f

The code consists of a macro invocation of the forth literal "instruction" followed by the integer value.

Control flow constructs generate forth branch instructions and branch addresses. The "if" statement generates the following code.

... if ... -- generates --> +------!!! +--!!!

0branch		e	
token		s 2	

+------!!! +--!!!

branch		a	
address		s 2	

+------!!! +--!!!

which generates for pass 3.

@resolve.exec z0branch,0 @resolve.addr <dfn containing if>, <dfn start-branch addr>

The address following the branch token references an address within the current definition.

The il code generator is broken into two parts. The front end is the machine independent part and the back end is the machine dependent part.

2.3.2.1 Intermediate Level Code Generator Front End -

The il code generator front end is machine independent to the extent that it is independent of the type of cpu it is running on. It is dependent on the type of forth machine it is running on. The il code generator front end needs to know about the dictionary format, and code threading method of the forth compiler it is running on. This knowledge is required by the front end so it can drive the back end properly. The current implementation knows about 6502 itc. The routine that controls the production of the il code is "generate_intermediate_code".

Each data type mark has its own routine to process that data type.

Execution tokens are converted back to symbols by "expand_execution_token". Addresses are converted to symbols by "expand_address". Integers, bytes, and allocated storage are handled by "expand_integer", "expand_byte", and "expand_storage".

2.3.2.2 Intermediate Level Code Generator Back End -

The il code generator back end is a collection of routines that converts numbers and strings into the pass 3 required syntax. The current pass 3 is MAC/65, a macro assembler sold by Optimized Systems Software for the Atari 8-bit line of PC's. Yet Another Target Compiler Page 15

The requirements placed upon the il code generator back end are as follows:

Bytes are printed as ".byte" statements and integers are printed as ".word" statements. Labels and symbols are restricted to starting with an alpha character, "@" or "?" and the rest of the characters are alphanumeric, "@", ".", or "?". Maximum symbol length is 127 characters.

There was no problem in creating routines that printed bytes and integers in the proper format. Problems arose when printing symbols from name fields in the dictionary headers.

The form of symbols in MAC/65 is fairly unrestricted when compared to most assemblers but it is very restricted when compared to forth symbols. Converting forth symbols to MAC/65 symbols required a couple of things. First, every symbol is prefaced with a "z" as an indication of a compiler generated symbol. It also simplifies the handling of forth symbols that start with a number. Second, a translation of non-permitted characters to strings of permitted characters is performed.

When printing a forth symbol as a MAC/65 symbol, every character in the symbol is translated to a "legal" character or characters by passing the characters through a filter. To simplify the translation of non-permitted characters to strings of permitted characters, every "illegal" character gets its own routine. The filter tests to the existence of translatable characters and passes the characters to the proper routines otherwise it just prints the character unchanged.

Other routines in the il code generator back end deal with header format for the target machine and header production control.

A large attempt was made to keep the il code generator back end reasonably independent of MAC/65 or any assembler.

2.3.2.3 Macros Used In Pass 3 -

To make pass 3 processing easier with MAC/65, a series of macros were created. These macros take care of many aspects of forth headers and forth code creation. The macros are described briefly below:

- @mark.link marks a location for the next use of @leave.link.

- @leave.link uses the location marked by @mark.link to create a dictionary link field.

- @fudge.dict adjusts the dictionary to prevent ITC forth on a 6502 processor from having a CFA that straddles a page boundary. Yet Another Target Compiler Page 16

- @resolve.cfa creates a CFA for a definition.

- @resolve.addr creates an pointer in memory.

- @resolve.exec creates an execution token in memory.

Yet Another Target Compiler Page 17

PRODUCING AN APPLICATION FOR A TARGET MACHINE#

The target compiler takes text files as input and produces text files as output. The tool used for pass 3 processing may or may not use text files for input. MAC/65 does not use text files without processing them into tokenized form. This requirement for file conversion adds a couple of steps to the process of producing an executable image on an Atari 8-bit computer.

Producing an application for a target machine is not very difficult. The process follows the steps outlined below.

1. An application is developed in the local forth environment.

2. After it works locally, the target compiler is loaded and the application is compiled with the target compiler.

3. The target compiler produces a file containing the il code form of the application.

4. The il code, a forth kernel, and any external language linkages are run through the pass 3 tool (an assembler and if needed a linker)

5. if an error listing indicates any missing routines, the missing routines are added to the kernel (if assembler) and repeat step 4 or to the application (if forth) and repeat step 2.

6. when a clean pass 3 is achieved, the executable image produced should work exactly as it did in the local (original) environment.

WHY YATC IS NOT JUST A FANCY DECOMPILER#

By just looking at the output of the target compiler, the conclusion could be drawn that this is a fancy decompiler. The major differences between the target compiler and a decompiler are the following:

- A decompiler prints out just the names of the forth definitions.

- A decompiler requires special case handling for all multi-byte forth "instructions" and defining words

- This target compiler knows about all data types compiled and can convert them to symbols.

- This target compiler "learns" about all new defining words can output the code they generate without changes to the target compiler.

CODING TRICKS AND OTHER THINGS THAT WON'T WORK#

Like all language, forth is not immune to programmers creating "tricks" to take advantage of special cases or situations. This target compiler handles most of the tricks found in forth but it misses on a few. Most of the not supported tricks are:

- Negative allocation to permit overwriting of variables or constants storage

- Naming relocatable addresses as constants. (This is ok for non-relocatable addresses).

- Address calculations using [...] literal. (They should change to [...] address.literal).

All of the tricks are either bad or lazy programming practices. They can be remedied easily and changed into something more maintainable.

Another area where the target compiler will not work properly all the time is in the area of vocabularies. Currently, the target compiler restricts the application to one vocabulary.

GLOSSARY OF EXTERNAL WORDS#

This glossary describes externally visible words provided by the target compiler. All other words are internal to the target compiler and should not be depended on by other applications.

start_target_compiler (---)

Start up the target compiler. Headers are produced and target compiler output is directed to the console device.

"start_target_compiler (fn adr ---)

input: fn adr; address of os specific legal filename

Start up the target compiler. Headers are produced and target compiler output is directed to the file specified by the input filename.

end_target_compiler (---)

Clean up target compiling. Print any remaining intermediate level code and close file if open.

near (---)

Stop compiling for far (target) environment and only compile in near (local) environment.

far (---)

Resume compiling for far (target) and near (local) environment.

heads (---)

Turn on header production for target compiled definitions.

tails (---)

Turn off header production for target compiled definitions. Produce only the "tail" end (the code part of a definition.

a, (addr ---)

Allocate space for an address in dictionary and place address found on top of stack in that allocation.

e, (execution token ---)

Allocate space for an execution token in dictionary and place execution token found on top of stack in that allocation. Yet Another Target Compiler Page 21

REFERENCES#

John J. Cassady, "METAFORTH, A metacompiler for fig-forth" Mountain View Press, (1980)

Henry Laxen, "Meta Compiling 1", Forth Dimensions, (Vol 4, Number 6), PP 19-22.

Henry Laxen, "Meta Compiling 2", Forth Dimensions, (Vol 5, Number 2), PP 23-24.

Henry Laxen, 'Meta Compiling 3", Forth Dimensions, (Vol 5, Number 3), PP 31-32.

A. P. Haley, H. P. Oakford, and C. L. Stephens "In Sitsu Development - The ideal complement to Cross Target Compilers", 1985 FORMAL Conferences Proceedings, (October 1985), PP 391-398.

Randy M. Dumse, "The R65F11 and F68K Single-Chip Forth Computers", The Journal of Forth Application and Research, (Volume 2, Number 1, 1984), PP 11-21.

```
;---------------------------------------------------------------
; mac/65 macro set for yatc
;---------------------------------------------------------------
; @mark.link and @leave.link are used by the assembler to build the
; linked list that makes up the forth dictonary structure
.macro  @mark.link
@linktmp .= @nfa
```

```
@nfa .= *
.endm

.macro @leave.link
.word @linktmp
.endm

@linktmp .= 0
@nfa .= 0

; @fudge.dict looks at the size of the header and moves the start iff
; the cfa will cross a page boundary
; NOTE: this is needed iff compiling for 6502 itc
;
; usage: @fudge.dict <n>    where n is the size of the name

.macro @fudge.dict
.if ~[*+%1+3]&$ff=$FF
*=*+1
.endif
.endm

; @resolve.cfa creates a cfa for itc implementations  this will be a
; nop for stc, dtc and ttc implementations.
;
; usage: @resolve.cfa <sym>

.macro @resolve.cfa
.IF *&$FF=$FF
.ERROR "CFA STARTING ON PAGE END"
.ENDIF
.word %1
.endm

; @resolve.addr places an address and offset into the dictonary
;
; usage: @resolve.addr <sym>, <offset>

.macro @resolve.addr
.word %1 + %2
.endm

; @resolve.exec places an execution address and offset into the
; dictonary.  this generates the proper code sequence for any code
; threading technique.
;
; usage: @resolve.exec <sym>, <offset>

.macro @resolve.exec
.word %1 + %2
.endm

.end

 ." D:VECTOR.4TH    LOAD " CR
  " D:VECTOR.4TH    LOAD

( ." D:STACK.4TH     LOAD " CR )
(  " D:STACK.4TH     LOAD )
```

```
 ." D:FSM.4TH       LOAD " CR
  " D:FSM.4TH       LOAD

 ." D:GENFILE.4TH   LOAD " CR
  " D:GENFILE.4TH   LOAD

 ." D:GENSTORE.4TH  LOAD " CR
  " D:GENSTORE.4TH  LOAD

 ." D:LOGDATA.4TH   LOAD " CR
  " D:LOGDATA.4TH   LOAD

 ." D:FINDNAME.4TH  LOAD " CR
  " D:FINDNAME.4TH  LOAD

 ." D:CG65XXI.4TH   LOAD " CR
  " D:CG65XXI.4TH   LOAD

 ." D:CGCOMMON.4TH  LOAD " CR
  " D:CGCOMMON.4TH  LOAD

 ." D:MOD4TH.4TH    LOAD " CR
  " D:MOD4TH.4TH    LOAD

 ." D:GENLINK.4TH   LOAD " CR
  " D:GENLINK.4TH   LOAD

( ------------------------------------------------------------------- )
( vector.4th )
( ------------------------------------------------------------------- )

( vector definition words )
: vector_undefined
  ." undefined vector "
;

: vector
( define a vector word )
( : name vector_undefined ; )
( use : to create the header )
  ~[compile] :

( place the undefined vector into the dictionary )
  compile vector_undefined

( now, finish off the "definition" with a ; )
  ~[compile] ;
;

: make ( --- adr )
( make vector definition become definition )
  ~[compile] '
;

: (become) ( vector ptr, action pfa --- )
  cfa swap !
;

: become ( vector ptr --- )
  ~[compile] '
```

```
  (become)
;

: ~[make] ( --- adr )
  ?comp
  make
; immediate

: ~[become] ( vector adr --- )
  ?comp
  ~[compile] '
  compile (become)
; immediate

( ------------------------------------------------------------------ )
( fsm.4th )
( ------------------------------------------------------------------ )

( finite state machine tool )
( . )
( history )
(  1 nov 86 esj created )

: causes ( state value --- )
( compiles the state table information )
( test for execution compiler state.  will not work if compiled )
( into definition )
  ?exec

( compile in the state value )
  ,

( get the address of the action routine  and convert it to )
( executable form )
  ~[compile] '  cfa ,
; immediate

: initial_action  ( --- )
( define the initial action of the fsm )
( initial actions execute when the fsm is entered )
 0 ~[compile] causes
;

: otherwise_action  ( --- )
( define the otherwise action of the fsm )
( execute if we do not match to any state )
 -1 ~[compile] causes
;

: state_machine  ( --- )
( build a state machine table )
  <builds
    ?exec
  does>
(   move state table address out of the way and do initial action )
(   and get state to evaluate from initial action )
    dup >r 2 + @  execute

(   now that we have the state value to act on, go fine it and do it!)
    begin ( stk = state )
```

```
(     update state table pointer )
      r> 4 + >r

(     get state value for comparison )
      r @ ( stk=state,state1)

(     now, test for state value match or end of table. )
      over over = swap -1 = or  ( stk = state, f1|f2 )

    until
(   get rid of state flag and execute action )
    drop r> 2+ @ execute
; immediate

( ---------------------------------------------------------------- )
( genfile.4th )
( ---------------------------------------------------------------- )
( file io for forth intermediate generator )
( . )
( history )
( 28 dec 86 esj close file iff file iocb is not 1..7 i.e legal iocb# )
(  7 nov 86 esj created )

0 variable link_iocb

: ?legal_iocb ( iocb# --- f )
( check to see if iocb is in the range of 1..7.  tf = yes | ff = no )

  1 - dup 0< swap 6 > or lnot
;


: open_code_file ( fn adr --- )
( open code destination file )
( input: fn adr; addr of file name )
( if any file problems, the link is terminated via abort )
  8 open 1 = lnot
  if ( bad open )
    ." check your filename and try again "  abort
  then

  link_iocb !
;

: close_code_file ( --- )
( close code destination file )

  link_iocb @ ?legal_iocb
  if ( legal )
    link_iocb @ close ( file ) drop ( error code )
  then
  0 link_iocb !  ( restore code out iocb to 0 )
;

: swap_iocb ( --- )
( swap contents of out.iocb and link_iocb )

  link_iocb @ out.iocb @
  link_iocb ! out.iocb !
;
```

```
( ------------------------------------------------------------- )
( genstore.4th )
( ------------------------------------------------------------- )
( constants and storage for the forth intermediate code generator )
( . )
( history )
( 27 nov 86 esj added execution_mark
( 25 oct 86 esj split from main file )

40      constant immed.bit
04      constant storage_data_type_size
06      constant most_data_type_size
14      constant header_data_type_size
ascii A constant address_mark
ascii I constant integer_mark
ascii C constant byte_mark
ascii S constant storage_mark
ascii L constant label_mark
ascii E constant execution_mark
0400    constant data_type_log_size

( current_field contains the pointer to a data element definition. )
( current_address contains the pointer to a data element within )
( a definition. )

0 variable current_field
0 variable current_address
0 variable first_time
0 variable data_type_log  data_type_log_size allot
here constant data_type_log_end

( ------------------------------------------------------------- )
( logdata.4th )
( ------------------------------------------------------------- )
( log data type code)
( . )
( history )
( 28 dec 86 esj vector mark_it
( 20 nov 86 esj add get_next_mark
( 25 oct 86 esj split from main routine )

: reset_data_type_log ( --- )
( setup relocation buffer to start condition )
  data_type_log dup !
;

: ?empty_data_type_log ( --- f )
( test for empty data type log )
( output: f; tf = empty, ff = not empty )
  data_type_log dup @ =
;

vector mark_it

: (mark_it)  ( mark --- )
( enter a chunk of data into the relocation buffer )
( input: mark; token placed into the relocation buffer )
( output: none )

( index next free relocation buffer location )
```

```
  2 data_type_log +!

( test for end of relocation buffer )
  data_type_log @ data_type_log_end u<
  if
(   not at end, place mark into dictionary )
    ( debug )
   ( data_type_log @ space . dup emit )
    data_type_log @ !
  else

(   warn of data type log overflow )
    ." overflow data type log at "
    data_type_log @ 0 d. cr
  then
;


: ?done ( --- f )
( test for linker done. tf = done)
  data_type_log @
  current_field  @
  2dup =
  >r u< r> or
;

: get_next_mark ( --- dfn addr, size, mark )
( return an address and its data type )
( input: none )
( output: dfn addr;  pointer to data of data type specified by mark )
(         size;      size of data represented by mark )
(         mark;      data type mark )
( current_field points to the current data type. return the data type )
( pointed to by current_field )
( current_address points inside of definition. location current_address )

( points to should be same data type as current_field indicated data )
( type mark. )

( setup return values )
  current_address @
  current_field @ @

( move to next data type and matching address )
  dup storage_mark =
  if ( storage mark, offset to current address follows )
    2 current_field +!
    current_field @ @  dup current_address +!
    2 current_field +!

  else ( standard size data type mark )
    6 current_field +!
    2 current_address +!
    2 ( size argument )
  then
  swap ( put size in its proper place )
;

: init_code_out_ptrs ( --- )
```

```
( setup pointers for intermediate code generator )
  data_type_log 2+ current_field !
  latest  current_address !
;

( ------------------------------------------------------------- )
( findname.4th )
( ------------------------------------------------------------- )

: look_up ( adr --- adr, nfa, f )
( look_up searches the current vocabulary for close match )
( to the address passed in.  will not give proper match when )
( vocabularys are mixed within a definition)
(   get nfa of start of search )
    context @ @
    true ( dummy flag )
    begin
(     stk = adr within dfn, nfa of definition, f ~[either dummy or prev] )
(     drop flag because flag is for exit testing )
      drop

(     test for "match" )
      2dup swap u<

(     test for end of dictionary list)
      dup
      if
(       make the stack look like not found condition )
        dup

      else
(       stk = adr1, adr2, f )
        swap pfa lfa @

(       stk = adr1, f, adr3 )
        dup 0=

(       stk = adr1, f, adr3, eov f)
        rot swap over or

(       stk = adr1, adr3, f, f or f )
      then
    until

(   stk = adr, found nfa, f tf=found | ff=not found )

;

( ------------------------------------------------------------- )
( cg65xxi.4th )
( ------------------------------------------------------------- )
( mac65 code version )
( routines that generate the intermediate code text. )
( when generating code for different linking methods, different macro )
( assemblers or a native forth linker, hopefuly these are the only )
( routines that need to be changed )
( . )
( history )
( 28 dec 86 esj limited range of .byte to 0..ff )
( 28 nov 86 esj removed the machine independent parts )
```

```
( 27 nov 86 esj added handling execution mark. )
( 23 nov 86 esj changed code output for mac65 syntax )
( 20 nov 86 esj eliminate need for next_data_type_field )
(  1 nov 86 esj sync data typing table and current address pointer)
( 25 oct 86 esj split from main file )

hex

: .asm_hex ( n --- )
( print a byte as mac65 hex format )
( input: n; integer 0..ffff )

( save current base and convert to hex )
  base @ swap  hex

( do formated output )
  0 ( convert n to unsigned double )
  <# #s ascii $ hold #> type

  base !
;

: .byte ( n --- )
( print n as .byte statement )
( input: n; integer 0..ff )

  ."  .byte "
  0ff and ( limit data to 0..ff range )
  .asm_hex
;

: .word ( n --- )
( print n as .word statement )
( input: n; integer 0..ffff )

  ."  .word "  .asm_hex
;

( the following definitions translate non-alpha characters )
( into alpha char stings. )

: .store
  drop ( char )
  ." store"
;

: .quote
  drop ( char )
  ." quote"
;

: .crosshatch
  drop ( char )
  ." crhatch"
;

: .dollar
  drop ( char )
  ." dollar"
```

```
;

: .percent
  drop ( char )
  ." percent"
;

: .ampersand
  drop ( char )
  ." ampersand"
;

: .tick
  drop ( char )
  ." tick"
;

: .at
  drop ( char )
  ." at"
;

: .oparens
  drop ( char )
  ." oprn"
;

: .cparens
  drop ( char )
  ." cprn"
;

: .minussign
  drop ( char )
  ." msgn"
;

: .equalssign
  drop ( char )
  ." eqsgn"
;

: .lessthan
  drop ( char )
  ." lthn"
;

: .greaterthan
  drop ( char )
  ." grthn"
;

: .underline
  drop ( char )
  ." uline"
;

: .plus
  drop ( char )
```

```
  ." plus"
;

: .star
  drop ( char )
  ." star"
;

: .slash
  drop ( char )
  ." slash"
;

: .cbracket
  drop ( char )
  ." cbrckt"
;

: .obracket
  drop ( char )
  ." obrckt"
;

: .semi
  drop ( char )
  ." semi"
;

: .colon
  drop ( char )
  ." colon"
;

: .dot
  drop ( char )
  ." dot"
;

: .comma
  drop ( char )
  ." comma"
;

: .qmark
  drop ( char )
  ." qmark"
;


state_machine emit_limited ( char --- )
( convert all char that may choke an assembler to ok char )
 initial_action  dup ( char )

  ascii ! causes .store
  ascii " causes .quote
  ascii # causes .crosshatch
  ascii $ causes .dollar
  ascii % causes .percent
  ascii & causes .ampersand
```

```
  ascii ' causes .tick
  ascii @ causes .at
  ascii ( causes .oparens
  ascii ) causes .cparens
  ascii - causes .minussign
  ascii = causes .equalssign
  ascii < causes .lessthan
  ascii > causes .greaterthan
  ascii _ causes .underline
  ascii + causes .plus
  ascii * causes .star
  ascii / causes .slash
  ascii ] causes .cbracket
  ascii ~[ causes .obracket
  ascii ; causes .semi
  ascii : causes .colon
  ascii . causes .dot
  ascii , causes .comma
  ascii ? causes .qmark

  otherwise_action emit


: (common.nfa) ( nfa --- end adr, start adr)
( convert nfa to do loop arguments for all .nfa versions )
( input: nfa; pointer to name field of definition )
( output: end adr; last address of name field )
(         start adr; first address of name field )

( get size of name, and mask out header fields from size )
  count 1f and

( stk = str addr, size | convert to do loop bounds )
  over + swap
;


: .nfa_as_bytes ( nfa --- )
( print the name field as a .byte string )
( input: nfa; printed name field address )

  cr ."  .byte "
  dup c@ .asm_hex
  (common.nfa)
  do
(   place , between fields )
    ." , "

(   get char, mask to 7 bits, and print it. )
    i c@ 7f and .asm_hex

  loop

( turn on the 80 bit for the last char.  note: this method is )
( a mac/65 "feature" )
  ." +$80"
;
```

```
: .nfa ( nfa --- )
( print the name field without the 80 bit showing )
( input: nfa; printed name field address )

  (common.nfa)
  do

(   get char, mask to 7 bits, and print it. )
    i c@ 7f and emit
  loop
;


: .nfa_limited ( nfa --- )
( print the name field without the 80 bit showing or any characters )
( that are not palatable to many assemblers )
( input: nfa; printed name field address )

  ." z" ( many assemblers must have an alpha first char )
  (common.nfa)
  do

(   get char, mask to 7 bits, and print it. )
    i c@ 7f and emit_limited
  loop
;


: generate_resolve_addr ( adr, nfa --- )
( generate the resolve code, label, and offset, if needed. )
( input: adr; an absolute adr ref to be converted to a symbolic adr ref)
(        nfa; name field address of definition that contains adr )
( output: a line in the form of              )
(        @resolve.addr <sym>, <offset> )
( on a new line, print name of address resolver in linker )
  cr  ."  @resolve.addr "

  dup .nfa_limited
  ." ,"  pfa cfa - .asm_hex
;


: generate_resolve_exec ( adr, nfa --- )
( generate the resolve code, label, and offset, if needed. )
( input: adr; an absolute adr ref to be converted to a symbolic adr ref)
(        nfa; name field address of definition that contains adr )
( output: a line in the form of              )
(        @resolve.exec <sym>, <offset> )
( on a new line, print name of address resolver in linker )
  cr  ."  @resolve.exec "

( print execution token as symbol )
  dup .nfa_limited

( calculate and print offset )
  ." ,"  pfa cfa - .asm_hex
;

: fudge_dictionary ( count --- )
```

```
( input: count; size of name field )
( NOTE: this is ONLY NEEDED if 6502 itc because of the 6502 jmp indirect bug.)

( print the macro that corrects the dictionary if the cfa could end)
( up on an xxFF boundary )
  1f and ."   @fudge.dict " .asm_hex

( consume dfn offset field ~[6502 only])
  get_next_mark drop drop drop
;


( header print control )

vector .header

: (.header) ( adr ---  )
( print the header field. can be turned off for headerless code )
( input: adr; pointer to start of name field )
( output:  )
( mac65 version: we could create a macro to build headers BUT mac65)
( doesn't allow " in strings ~[silly people] SO, we gotta build the)
( header and all associated lables for ourselves.  All is not lost.)
( we will still use macros for as many things as possible )

  cr

( get size of header for dictonary adjustment )
  dup c@ fudge_dictionary

( mark link address )
  cr  ."  @mark.link "

( leave the header trail )
  .nfa_as_bytes

( leave link field )
  cr ."  @leave.link"

( consume the proper amount of data and their marks to leave )
( us at the cfa. )
  get_next_mark drop drop drop ( consume lfa )
;

: (noheads) ( adr --- )
( dummy for not dumping headers )
( input: adr; pointer to start of name field )
( method: consume data type marks like (.header but don't dump )
( any header data. )

  drop ( input )
  0 fudge_dictionary ( make 0 length fudge field )
  get_next_mark drop drop drop ( consume lfa field )

( now hopefully, we are at the cfa mark )
;

: heads ( --- )
( turn on headers )
```

```
  ~[make] .header ~[become] (.header)
;

: tails ( --- )
( turn off headers )
  ~[make] .header ~[become] (noheads)
;

( end of header control )

: .cfa ( cfa --- )
( print cfa label for definition at cfa given )
( input: cfa; )
( output: a line with a cfa label.  lable form is z.<dfn name> )
  @  look_up
  drop ( flag because we will "always" find it )
  cr ."  @resolve.cfa  z." .nfa_limited  drop
;

: .label ( --- )
( make a label for the current definition. )
  cr  latest .nfa_limited
;

( ---------------------------------------------------------------- )
( CGCOMMON.4TH )
( ---------------------------------------------------------------- )
( Code Generator COMMON section )
( this code is the ~[hopefully] machine independent part of)
( code generation.  this code knows about itc forth implementations.)
( . )
( history )
( 28 nov esj split codeout.4th into machine dependent/independent parts)
hex


: expand_address ( adr, size ---  )
( expand address into symbolic label )
( input: adr; pointer to contents of a definition )
(        size; size of address field. currently ignored )

  drop ( size )

( is it a branch offset? assume all branches are within 1 byte )
  dup @
  100 over +  200 u<
  if ( branch )
    +   2 - ( adjust branch offset )

  else ( not branch, get rid of 2ed on stack )
    swap drop

  then
  look_up ( and find what definition it belongs to )
  if ( found )
    generate_resolve_addr

  else ( not found )
    2drop ( not found address )
    ." not_found "
```

```
  then
;


: expand_execution_token ( adr, size ---  )
( expand execution token into symbolic label )
( input: adr; pointer to contents of a definition )
(        size; size of token. currently ignored )

  drop ( size )

( assumption: all executable tokens have a 0 offset from start of )
( definition therefor use the following code segment )
  @ dup 2+ nfa true ( we "always" find it. )

( if executable tokens have a variable offset from definition )
( start, use the following code segment )
(  @ look_up ( and find what definition it belongs to )
  if ( found )
    generate_resolve_exec

  else ( not found )
    2drop ( not found address )
    ." not_found "

  then
;

: expand_integer ( adr, size ---  )
( produce integer intermediate code output )
( input: size; size of the integer compiled at address )
(        adr; pointer to contents of a definition )
( note: size is currently ignored. it will be used in the future to)
( handle different sized integers )
  drop ( size)

( produce new line, get integer, and print it with line terminator )
  cr
  @ .word
;

: expand_byte ( adr, size ---  )
( produce byte intermediate code output )
( input: size; size of the integer compiled at address )
(        adr; pointer to contents of a definition )
( note: size is currently ignored. it will be used in the future to)
( handle different sized integers )
  drop ( size)

( produce new line, get byte, and print rage ( adr, size --- )
( produce the storage code output )
( input: size; size of storage area )
(        adr; pointer to contents of a definition )
( marked storage allocation is a special case. a storage marker )
( is generated every time the compiler allots some space.  the )
( storage marker generated after a "," or "c," or "a," is hidden. )

( do nothing if storage allocation is 0 in length )
```

```
  -dup ( if not 0 )
  if
(   loop count not 0, loop through all of the bytes and output them )
    over + swap  ( build loop indexes )
    do
      cr
      i  c@ .byte
    loop
  then
;

: bad_data_type ( data mark, adr --- data mark)
  drop ( dfn ptr )
  cr  ." not found -> "   dup emit
  cr
;

state_machine datatype_switch
( dispatch to proper action for each data mark )
  initial_action       task ( do nothing )

  address_mark   causes expand_address
  integer_mark   causes expand_integer
  byte_mark      causes expand_byte
  storage_mark   causes expand_storage
  execution_mark causes expand_execution_token
  otherwise_action     bad_data_type


: generate_intermediate_code ( --- )
( this is the definition that produces intermediate code output )
( note: produce code iff there is some to be dumped.  the test is )
( if data_type_log is empty or not )
  ?empty_data_type_log lnot
  if
(   redirect io to the output file )
    swap_iocb

    init_code_out_ptrs

(   print header, cfa, and label fields )
    get_next_mark drop drop  .header
    get_next_mark drop drop  .cfa
  .label

(   start converting a forth definition back into symbolic references )
    begin
      get_next_mark

(     stk =  definition pointer, data type mark )
      datatype_switch

      ?done
    until

    cr

(   turn io back to normal output )
    swap_iocb
```

```
  then
;

( ------------------------------------------------------------- )
( MOD4TH.4TH )
( ------------------------------------------------------------- )
( modify a, , and c, to keep data type information when compiling )
( . )
( history )
( 27 nov 86 esj add (e,)
( 25 oct 86 esj split from main program )

: (a,) ( n --- )
( comma with an address data type mark)
( input: address value )

  address_mark mark_it
  (,)
;

: (e,) ( n --- )
( comma with an execution data type mark)
( input: execution token )

  execution_mark mark_it
  (,)
;


: (i,) ( n --- )
( comma with an integer data type mark)
( input: integer value )

  integer_mark mark_it
  (,)
;

: (b,) ( byte --- )
( c comma with an byte data type mark)
( input: byte value )

  byte_mark mark_it
  (c table )
( input: n; number of bytes to allocate in dictionary )

  storage_mark mark_it
  dup          mark_it
  dp ( code that is replaced when allot is hot patched )
;

( ------------------------------------------------------------- )
( GENLINK.4TH )
( ------------------------------------------------------------- )
( forth target compiler.  This program modifies several aspects )
( of a forth compiler. The end goal is to generate an intermediate )
( source code form suitable for a macro assembler or other )
( executable image generating tool)
( . )
( history )
( 15 jly 86 esj created v1.0 )
```

```
( 18 jly 86 esj added link_end and documentation )
(  7 sep 86 esj added limit checking for relocation buf size)
( 21 oct 86 esj reorged and split into separate files )
(  3 nov 86 esj seperated file io into own file )
( 28 dec 86 esj improved compiler control )

hex


: common_pass_2 ( --- )
( common work for all pass 2 definitions.  this is all that is )
( done when the target compiler is first turned on.  Otherwise it )
( is the final work for compleating pass 2 )
  reset_data_type_log
  (create)
;

: do_pass_2 ( --- )
( second pass code generator )
( method: generate intermediate code )

  generate_intermediate_code
(  cr ." rel buf @ "  )
(  data_type_log @ 0 d. )
(  here 0 d.  cr )

  common_pass_2
;

: first_pass_2 ( --- )
( pass 2 code executed first time through pass 2 )
  ~[make] create  ~[become] do_pass_2
  common_pass_2
;

: far ( --- )
( turn on the pass 2 code generator )
  ~[make] create  ~[become] first_pass_2
  ~[make] mark_it ~[become] (mark_it)
  reset_data_type_log
;

: near ( --- )
( turn off the pass 2 code generator )
( method: flush the currently active definition then reset )
( create and mark_it to do-nothing states )

  generate_intermediate_code cr
  ~[make] create  ~[become] (create)
  ~[make] mark_it ~[become] drop
;

: end_target_compiler
( turn off linker code and dump the last definition )
  near  close_code_file
;


: start_target_compiler ( --- )
```

```
( method: turn on pass 2 code generator, reset data )
( type logging, sets initial entry state, and turn on headers )

  far heads
;

: "start_target_compiler ( fn adr --- )
( input: fn adr; address of legal filename )
( method: open output file, turn on pass 2 code generator, reset data )
( type logging, sets initial entry flag using start_target_compiler )
( whenever possible )

( starts generating a link output file)
  open_code_file

( do everything else )
  start_target_compiler
;

." setup world "
( init the relocation buffer so the world starts clean )
reset_data_type_log

( setup compiling locally )
near

make a,     become (a,)
make e,     become (e,)
make ,      become (i,)
make c,     become (b,)
make allot  become (allot)

( abbrevs )
:  stc  start_target_compiler ;
:  etc  end_target_compiler   ;
: "stc "start_target_compiler ;

( notes: )
( near stops the target compiler output and only compiles locally. )
( far lets the target compiler pick up from where it was stopped by )
( near and compile for far away places )
( start_target_compiler and  "start_target_compiler do just )
( that, they start up the target compiler.  the difference is )
( "start_t_c opens a file for code output. )
( end_target_compiler finishes the target compilation process by )

( closing files dumping final code segments and similar things. )
```