Experiences from the Development of Harvest C

Eric W. Sink

Abstract:

This paper describes the development of Harvest C, a full C compiler and linker for the Macintosh. This project has revealed a number of interesting issues, partially due to its unusual size, and partially due to the internal structure of Macintosh applications. The following areas will be considered:

- *Memory management for large abstract data structures*
- *Compiling the usual Macintosh extensions to C*
- *Macintosh application structure and linking*
- *The history of Harvest C and its future directions*
- *Comparisons with commercial C compilers*

Harvest C is freely distributable

Introduction

This paper describes the development of Harvest C, a full C compiler and linker for the Macintosh. Harvest C compiles the full ANSI C language, as well as almost all of the MPW™ Macintosh extensions. All inclusive, the compiler consists of over 100,000 lines of C code and has been in development for 19 months. Harvest C does not represent an advance in compiler technology, but an advance in the availability of development tools for the Macintosh. As this is the case, some of the information presented here will not be particularly revelatory for the experienced student of compiler design. It is hoped that this discussion will be of interest and value for the Macintosh enthusiast, both expert and novice. Some of the material presented here is technical, some is anecdotal. There will be discussion of future directions as well as experience.

History

Harvest C began as a learning exercise in compiler design. Initially, all development was done on a Sun 3, constructing a simple compiler that generated 68020 assembly language, using 68881 instructions for all floating point operations. That compiler advanced to the point that it correctly compiled itself as well as enquire 4.3[1] and GNU Chess 3.1[2]. At this point, Harvest C (then called Ecc) had no Macintosh features. As development continued, much of the Macintosh-specific code was written on the Sun as well. The parser and code generator were modified to handle pascal function declarations. An assembler was added and code was written to dump object files in MPW format. As indicated above, a replacement for the Memory Manager was written to allow Macintosh specific code to be written and tested. In fact, a replacement for much of the File Manager was written as well. These functions had the ability to read and write Macintosh files on a UNIX™ system. The files were manipulated in Binhex format. In those days, Ecc could compile a C source file directly into a Binhexed MPW object file.

The code generator for SANE™ floating point was written on the Sun as well, though it obviously could not be tested there. The entire linker was written **after** porting to the Mac, because a functional equivalent of the Resource Manager would have been a huge and futile task.

In its earlier incarnations, Harvest C sported some features that it now lacks. It was originally planned that Ecc would be a cross compiler, retaining its ability to run on a UNIX system as well as its ability to generate assembler source for the UNIX assembler. At one point, Ecc

[1]Enquire is a program by Steven Pemberton, CWI, Amsterdam(steven@cwi.nl), designed to test floating point accuracy in C compilers.
[2]GNU Chess is distributed by the Free Software Foundation, 657 Massachusetts Ave., Cambridge, MA 02139.

successfully did function inlining like GNU CC[3]. Ecc also had the ability to calculate and report a number of software metrics, including Halstead's [2], McCabe's [3] and others. Ecc also once included support for much of the Objective-C [1] language. None of these features are currently being maintained.

Harvest C was first released in October 1991. Although it was unreliable and had an atrocious user interface, it **was** capable of generating simple Macintosh applications. Version 1.2 is a substantial improvement over previous releases.

Development now takes place in the THINK C™ environment, using the THINK Class Library[4] for user interface implementation. Since development moved from UNIX to the Macintosh, Harvest C no longer compiles itself. At the current time, it would be impossible to do so, as the source code now takes advantage of the object oriented extensions to THINK C, which are not part of the language which Harvest C recognizes. In the future, it may be possible to extend Harvest C's input language to allow for self-compilation once again.

Motivation

It is the author's opinion that a set of free development tools would benefit the Macintosh community. It may be argued that Macintosh programming is far too complex to be placed in the hands of anyone who is not serious enough to invest in a commercial development system. The hope is that the existence of a usable free C compiler will:

- Facilitate the learning of the C language on a wider scale.
- Spark innovation in amateur programmers, resulting in new Macintosh applications.
- Encourage others to produce and distribute related tools.

In general, the response from the Macintosh community has been very positive. It is still unknown whether Harvest C can survive in the presence of inexpensive commercial alternatives.

Technical Information

Overview

As of version 1.2, the user interface is somewhat of a hybrid of MPW and THINK C. Harvest C is designed to be used in the same manner as THINK C. However, an interactive shell is provided, reminiscent of MPW. This will be described later. Harvest C does not provide an integrated editor, but it does support the "project file" metaphor as opposed to Makefiles. Harvest C relies on AppleEvents™ to communicate with stand-alone text editors. This mechanism will be discussed later.

Like any typical compiler for a language like C, Harvest C consists of a preprocessor, lexical analyzer, parser, code generator, assembler, and linker. Each of these components is discussed below.

The preprocessor and lexical analyzer are integrated together. All preprocessor commands are handled "on the fly", including macro expansion. The result is a stream of preprocessed tokens, passed to the parser. The preprocessor/lexer is hand-written, not machine-generated.

The preprocessor exists in two sections. First, the function `GetCharacter` is the routine that passes individual characters to the lexer for the construction of tokens. The stream of characters resulting from multiple consecutive calls to `GetCharacter` corresponds to the source file after trigraph conversion, backslash line splicing, comment removal, and all preprocessor directives except macro expansion. `GetCharacter` is actually the interface to a larger collection of routines.

The interface from the parser to the lexer is a routine called `GetToken`. `GetToken` calls the `GetCharacter` routine, pieces tokens together, and handles macro expansion. The return value is a code indicating the type of token that was found.

The parser in Harvest C is hand-written, not machine-generated. It was written using the grammar provided by the ANSI C committee as a reference. As it parses the source file, it constructs symbol tables and parse trees that reflect the

[3]GNU CC is distributed by the Free Software Foundation.
[4]The THINK Class Library is distributed by Symantec Corporation with THINK C.

semantics of the source file in translation. The parser is responsible for both syntactic and semantic error checking. Error messages are accumulated and presented to the user in a scrolling log, each with its file and line position. The parser is also capable of generating warnings for a number of questionable constructs. These warnings are presented to the user in the same manner as error messages. The data structures created by the parser bear a nearly one-to-one correspondence with the structure of the C language itself.

A simple code generator translates the parse tree data structure to another data structure which directly represents the 68000 family assembly language. No intermediate representation is used. Register allocation is done "on the fly", rather than employing a graph coloring algorithm later.

There is a small peephole "optimizer" that makes a few modifications to the 68k code stream at this point. Harvest C currently does not have a "real" optimizer, as it does not support code motion, basic block optimizations, register coloring, loop induction variable optimizations, or common subexpression elimination.

The 68k data structures are assembled and converted into yet another data structure, representing object code records. Finally, the object code data structures are dumped to an object file. The file format used is identical to that used by the MPW linker and tools.

The Linker

Though somewhat less sophisticated, the Harvest C linker is functionally compatible with its MPW counterpart. It operates as follows:

1. Read all the object files into memory.
2. Resolve all references. This step verifies that no symbol is defined twice. For every reference record, it searches for the symbol being referenced. Pointer links between referenced symbols are constructed. Every referenced symbol is marked as active, so that inactive symbols may be stripped. The linker currently is not smart enough to strip symbols which are only referenced by inactive symbols. Consider Figure 1. Each oval represents a module. An arrow from one module to another represents a reference. For example, from the figure it can be seen that module A references module B. Modules A and E are inactive, because no modules reference them. Module B is active, because it is referenced by module A. A smarter linker would deduce that since B is only referenced by A, and A is inactive, then B is inactive as well. On the other hand, module D must be considered active, even though it is referenced by an inactive module, because it is also referenced by the active module C.

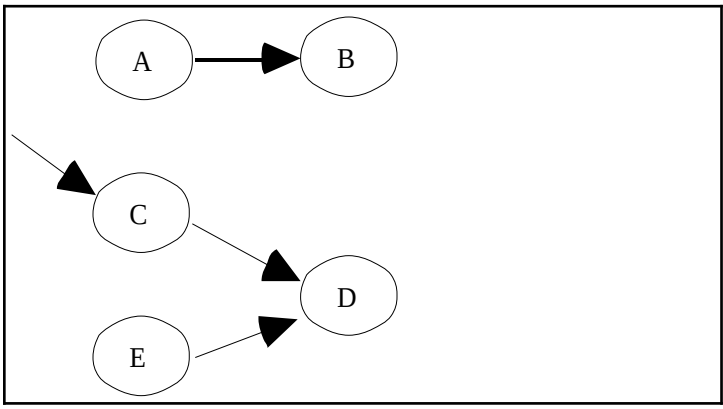

Figure 1

3. Assign A5 offsets. Every global data module is assigned a location in the A5 world. On program startup, a region of memory is allocated for the purpose of holding global program data and the jump table. Register A5 is set to point somewhere in the middle of this block, dividing the region into two parts. The memory at negative offsets from A5 contains global data. The jump table lives above A5. A data module might be assidned an A5 offset of `-2E6` (hex). Code modules get A5 offsets as well, but the offset point to the jump table entry. The jump table is discussed below.
4. Adjust all references. For every reference record, the module containing the reference must be patched with the A5 offset of the referenced symbol.
5. Build the global data initialization segment. In terms of emulating the MPW linker, this was the most difficult aspect of implementation. Basically, all initialized global data must be stored in a compressed table which is appended to the end of the segment called `%A5Init`. A detailed explanation of the data format would be too lengthy to include here.
6. Build the jump table. The jump table uses the same format as described in Inside Macintosh. Harvest C is currently not smart enough to know when it is necessary to generate a jump table entry, so the jump table contains an entry for every function.
7. Generate the application file. This means dumping the CODE segments using `AddResource()`. The application signature is set as well.
8. Add a SIZE resource, configured according to user defined options.
9. Add all resources contained in any resource files present in the project.

Memory Management

Memory Management has been a difficult issue throughout the development of Harvest C. The implementation of a compiler demands a large quantity of complex, dynamically allocated data structures for symbol tables, parse trees, assembler records, and so on. Harvest C not only allocates **many** data structures, it allocates many different kinds of data structures. Certainly Harvest C's data

structures could be more efficiently designed. After all, the project began as a learning endeavor. However, the services offered by the Memory Manager are perhaps too low-level for use in the allocation of individual blocks in a program such as this.

UNIX memory allocation is done using `malloc()`, resulting in pointers to structs. Dynamic memory allocation on the Mac is usually best done using `NewHandle()`, resulting in Handles to structs. This issue was 'handled'[5] by implementing a `NewHandle()` function on the Sun, using `malloc()`.

```
typedef  char *Ptr;
typedef Ptr *Handle;
typedef unsigned long Size;

Handle NewHandle(Size n)
{
       void **master;
       master = malloc(sizeof(Ptr));
       *master = malloc(n);
       return *master;
}
```

The Macintosh Memory Manager simply will not tolerate the level of abuse which may be permissible in a UNIX virtual memory system. The first attempt at a memory allocation strategy was to allocate one relocatable block for each node in a structure. Under this scheme, a simple linked list structure might look like this:

```
struct listNode {
       int data;
       struct listNode **next;
};
```

The Memory Manager offers its functionality on friendly terms with the rest of the operating system. This friendliness does not come without overhead. The ability to relocate a block of memory is very useful, particularly when dealing with the limited memory available in the early Macintosh models. However, under the Macintosh design, each relocatable block requires a master pointer. Harvest C's [admittedly imperfect] design of data structures resulted in the allocation of 30,000 handles or more during the compilation of medium-sized C source files. This means that over 117K of memory is consumed by master pointers alone. The Memory Manager does not provide acceptable performance with such extreme demands.

An alternative to the use of `NewHandle()` for every node, is to use `NewPtr()`. This results in far less overhead due to the lack of master pointers. However, as most Macintosh programmers know, the resulting heap fragmentation provides extraordinarily low performance.

The typical solution to this problem is to implement another layer of memory management on top of the Memory Manager. Many applications allocate very large chunks of memory using `NewPtr()`, and allocate their own structures from larger blocks. A solution of this nature was implemented, but it was later deemed unsatisfactory. Essentially, implementations of this nature are application-specific versions of `malloc()`. Although it seems very un-Mac-like, a well-written `malloc` library seemed the way to go. In fact, this is currently the memory allocation strategy used by Harvest C. Excellent results have been obtained thus far in the use of a `malloc()` written by Tim Endres[6]. This library provides the standard functions `malloc()` and `free()`. The technique used is to allocate large blocks of memory using `NewPtr()` and carve smaller blocks out of it to fill `malloc()` requests. Endres' library manages all the necessary blocks lists and other data structures internally. It also provides support for multiple pools, debugging information, and statistics.

There are certainly other viable strategies for handling memory. Unfortunately, experimenting with various strategies is time consuming, because the necessary changes to the code are extensive. The most important lesson learned here is that on the Macintosh, the best memory management strategy is probably not obvious.

Language Extensions

Harvest C supports a few important extensions to the ANSI C standard. These extensions are important for their support of access to the Macintosh Toolbox.

In the MPW C 3.2 header files, a Toolbox trap function is typically declared somewhat like this:

```
#pragma parameter __D0 ReadDateTime(__A0)
pascal OSErr
ReadDateTime(unsigned long *time) = 0xA039;
```

This declaration contains a great deal of information for the code generator. First of all, the `#pragma` directive specifies the location of the arguments and return value for this function. Using this example, Harvest C would place this function's single argument in register A0 instead of pushing it onto the stack as it usually would.

Furthermore, this declaration informs us that `ReadDateTime` is a trap function. When generating a call to this routine, Harvest C will generate the trap `0xA039` instead of the usual `JSR` instruction.

In order to allow for compilation of filter procedures and hook functions, Harvest C also deals correctly with normal pascal functions. These routines receive their arguments on the stack in the opposite order of that used by C functions. In addition, `char` and `short` parameters occupy two stack bytes for pascal functions and four stack bytes for C functions. Finally, pascal functions remove their own arguments from the stack, and return

[5]The author conveys his apologies for the pun.

their results on the stack.

All the various possibilities which can occur when generating a function call make that aspect of the Harvest C code generator rather complicated. The situation is even more complex when struct valued functions are considered.

Comparison with THINK and MPW

Overview

As commercial C compilers typically have a large staff of developers and testers, it would be completely unrealistic to presume that Harvest C could compete effectively. Nonetheless, as it has been a goal to make the product usable, a comparison to the excellent commercial compilers is indicate of the measure of success it has achieved.

It is not the purpose of Harvest C to compete with the commercial offerings. Both THINK C and MPW are distinct products, each with its own intrinsic value. It is hoped that Harvest C will be viewed in a similar manner, not as an attempt to dethrone its excellent neighbors.

Although Harvest C has not matched either THINK C or MPW in speed, features or reliability, it has aimed at a certain amount of innovation of its own. Harvest C combines many of the features of its commercial counterparts, in rather unique ways.

Tcl Scripting

The development of Harvest C is taking place in coordination with other authors working toward a set of freely distributable tools and applications. We hope to provide a high degree of integration among applications through the use of a common scripting language, the Tool Command Language (Tcl) [4]. Tcl[7] provides a simple but full featured language interpreter designed to be embedded in applications. Although each application typically provides its own extensions to Tcl, all such applications share the same basic scripting mechanisms. Through the use of standard AppleEvents, an application may offer its "scriptability" as a service to other programs.

An example of tool integration through Tcl and AppleEvents is the communication which takes place between Harvest C and Alpha[8]. In the shareware arena, Alpha has shown itself to be a powerful text editor for programmers. Coordination has taken place to allow Harvest C and Alpha to communicate, providing a more complete development environment. Alpha has the ability to accept an arbitrary Tcl script encapsulated in an AppleEvent. Harvest C uses this feature by passing Tcl commands to Alpha for various functions. For example, double-clicking on an error or warning message (in the error log) will send a Tcl script to Alpha which looks something like this:

```
openFile "Disk:Folder:File.c"
set pos [rowcolPos LINENUM 0]
select $pos [nextLineStart $pos]
```

The result is that Alpha opens the given file and selects the given line, ready for editing at the site of the error. Alpha also supports a full shell, offering a wide variety of functions to Harvest C simply through the use of Tcl.

Another application which supports identical Tcl scripting and communication mechanisms is Tickle[9]. Tickle is a shell utility, providing a number of built in conversion utilities as well as the ability to be extended using XTCL resources (similar to HyperCard™ XCMDs).

Harvest C itself is also completely scriptable using Tcl. This allows the user to control the development process in a more automated manner. The user accesses this functionality either through AppleEvents or through an integrated "shell". After opening the Harvest C shell, the user may issue commands using standard Tcl as well as a number of extensions. Harvest C currently supports the following extensions to the base Tcl language:

```
newProject ?PROJNAME?
openProject PROJNAME
closeProject
setOption     OPTION BOOLEAN
              OPTION BOOLEAN
setWarnings   all BOOLEAN
              WARNING BOOLEAN
              WARNING BOOLEAN
setSig OSTYPE
setPartition INTEGER
setSIZEFlags INTEGER
bringUpToDate
buildApplication ?APPNAME?
makeClean
runApplication
addFiles FILENAME(S)
removeFiles FILENAME(S)
compile CFILENAME
listProject
```

In addition, a set of general Macintosh extension commands are available. As an example, a user might execute the following script.

[6]This malloc library may be obtained from the author (e-sink@uiuc.edu). Tim Endres may be reached at time@ice.com.
[7]Throughout this paper, the Tool Command Language is referred to as Tcl and the THINK Class Library as TCL.
[8]Alpha is a shareware text editor for the Macintosh, by Pete Keleher (pete@rice.edu).
[9]Tickle is a freely distributable scripting environment for the Macintosh, by Tim Endres (time@ice.com).

```
cd "MyDisk:Development:myApp"
newProject myApp.π
setSig LASJ
setPartition 500
eval addFiles [glob -t TEXT *.c]
eval addFiles [glob -t rsrc *]
makeClean
bringUpToDate
closeProject
```

The effect of this script is to create a new project, set its partition and signature, add a group of C and resource files, and compile all the C files. Other possibilities include:

- The `setOption` command could be used to compile various files with different option settings.
- The Macintosh extension commands could be used to interact with the user during the build process.
- One could create a procedure to write a formatted list of the contents of the project file.

A Macintosh Tcl distribution is available to allow other developers to integrate Tcl into their applications easily. The Tcl scripting language is appropriate for all sorts of applications. Let us hope that other authors will undertake the writing of simple, freely distributable spreadsheets, databases and other applications, all based on Tcl.

Benchmarks

Table 1 presents a comparison of the performance of Harvest C against THINK C. Results from three test applications are presented. The Bullseye and MiniEdit applications are sample programs which come with THINK C. The StdFile application is the C source from the DTS[10] Sample Code #18. The "Compile" and "Link" columns list absolute times in seconds. For each test, Harvest C's results for each program are presented first, with THINK C's ratings appearing on the following line. The "Code size" column indicates the actual size of the generated code whereas the "App size" column shows the size of the final application, including all linked libraries and resources.

App (Compiler)	Compile (seconds)	Link (seconds)	Code size (bytes)	App size (K bytes)
Bullseye (Harvest)	122	40	2224	20
(THINK)	11	6	1386	
MiniEdit (Harvest)	320	55	8774	29
(THINK)	18	8	5288	
StdFile (Harvest)	271	47	9040	30
(THINK)	12	9	5732	14

Table 1

All timings were recorded on a Macintosh IIsi without FPU, in 32-bit mode. Taking nothing away from the extraordinary speediness of the THINK compiler, it should be noted that Harvest C's lack of support for precompiled header files is responsible for some of the disparity in the timings. The first two tests contain less than 1,000 lines of code, ignoring header files. The StdFile test consists of just over 1,800 lines.

[10]Developer Technical Support (Apple Computer, Inc.)

General Features

Harvest C was written to follow the conventions of MPW C 3.2 and THINK C 5.0. In fact, Harvest C does not provide any libraries or header files. Instead, it makes use of the MPW C headers and libraries without modification. Thus, Harvest C generates object files in MPW format, handles most all of the same extensions to C which are defined by MPW C, and Harvest C's linker is functionally compatible with the MPW linker (although not as powerful).

Harvest C does not generate SADE™ debugging information, nor will it accept object files generated by MPW tools which contain SADE information. With these exceptions, MPW and Harvest C have successfully shared object files without difficulty.

In general, Harvest C is functionally somewhat of an MPW in THINK C clothing:

- Like THINK C, Project files are used instead of Makefiles.
- Like MPW, a single object file (file.o) is generated for each C source file.
- Segmentation is controlled using `#pragma segment` directives in the source code.
- Toolbox traps use `#pragma parameter` directives and inline functions.
- Project files may contain one or more resource files.
- Apple Events are used to communicate with Alpha for text editing, as well as with ResEdit™ for resource editing.

Table 2 briefly summarizes some differing aspects of Harvest C and the two principal commercial compilers. This paper does not attempt to provide an in-depth comparison.

	THINK	MPW	Harvest
Editor	Built-in	Built-in	External
Scripting	None	Shell	Tcl
Build	Project	Makefile	Project
Debugger	Integrated	SADE	None
OOP	C++ subset	Separately	None

Table 2

Conclusions

Since Harvest C was written as a learning experience, a summary of what was learned is fitting. Most certainly, learning compiler design in a proper course is much easier. Much has been learned during code rewrites made necessary by inexperience. A list of proverbs for Macintosh compiler-writers appears below. While

these tidbits are purely the opinion of one author, they do carry a bit of hindsight:

- Use compiler generation tools for the parser and lexer.
- Do not be discouraged if you find that you need to rewrite a section of code. This is “par for the course”.
- Pay very careful attention to the design of your data structures, particularly parse trees and type records.
- Peruse the relevant chapters of a compiler design textbook before you even begin writing the register allocator.
- Do use a class library such as TCL or MacApp for the user interface.
- MacsBug is your best friend.
- Participate in USENET newsgroups and electronic mail. The exchange of ideas is invaluable.

Future Directions

As with any compiler, Harvest C has many areas of potential improvement. Some parts should be rewritten for efficiency and/or cleanliness of design. Currently the front end is being rewritten to use a yacc[11] parser with a lexer/preprocessor generated using flex[12].

In addition, there is much that could be done which has simply not been possible due to time constraints. Possible areas include an optimizer, a smarter linker, and support for object oriented extensions.

Harvest C is still more of a development project than a stable, full-featured alternative to the commercial offerings. In the hope that Harvest C may continue to grow, beyond the efforts of a single developer, the program is available in source form[13]. For Harvest C, the greatest potential for positive impact upon the Macintosh community lies in its widespread usage.

Acknowledgements

Although the responsibility for the content and form of this paper is mine, it would not have been the same without the help of friends. First, thanks to Pete Keleher, who commented on draft versions of this paper. Also, thanks to the many people with whom I communicate through electronic mail and network news. They provide invaluable advice and feedback. Thanks to Waldemar Horwat and all who have helped in the coordination of MacHack. Finally, thanks to my wife Lisa, for her patience and support.

References

1. Cox, B. *Object Oriented Programming : An Evolutionary Approach,* Addison Wesley, Reading, MA (1986).

2. Halstead, M.H. *Elements of Software Science,* Elsevier North-Holland (1977).

3. McCabe, T.J. A Complexity Measure. *IEEE Transactions on Software Engineering SE-2*, 4 (Dec 1976).

4. Ousterhout, J.K. Tcl: An Embeddable Command

[11]Yacc (Yet Another Compiler Compiler) is a compiler generation tool distributed with most UNIX systems.
[12]Flex is a compiler generation tool by Vern Paxson.

[13]Contact the author for details. Eric W. Sink (e-sink@uiuc.edu) 1014 Pomona Drive, Champaign, IL 61821.