Compiler Conventions
for the ORCA/M Development System
14 January 1986

Introduction

This technical note tells you how to install a compiler, interpreter, assembler, or some combination in the ORCA programming environment. The description is valid for both the Apple // version of ORCA and the upcomming Cortland version, which is called the Cortland Programmer's Workshop. This is an overview of some of the "how" and "why" behind the interface between ORCA and the compiler. For a description of the GET_LINFO and SET_LINFO calls refered to in this document, see the ORCA/M reference manual (for Apple // implementations) or the Cortland Programmer's Workshop ERS (for Cortland implementations). The only difference between the two is that pointers on the Apple // are two bytes, while the fields used for pointers on the Cortland require four bytes each.

Input Format and Conventions

Compilers must be capable of accepting ASCII files as input. These files consist of the standard ASCII character set with the high bit off. Lines are separated by the CR ($0D) character. The files themselves are stamped with a language number, described below. Whether or not control characters are allowed in the input file (other then $0D) is up to the implementor, but it is recomended that at least the form feed ($0C) be accepted if it appears in column 1 of an otherwise blank line. A possible future version of the ORCA editor would generate such lines. The effect of the line should be to do a form feed on a printed output, which is accomplished by simply sending the $0C character to the printer.

As a matter of convention, all source lines are assumned to be 80 columns long on the Apple // ProDOS based version of ORCA, and 255 columns on the Cortland version. The current system editor cannot generate longer lines, so this will not generally be a problem.

Entry and Exit Control

Control is passed to the compiler via a JSR instruciton from ORCA.HOST. The compiler starts by doing a GET_INFO call to ORCA.HOST to read the input parameters. On normal completion (no terminal errors) the compiler returns control to ORCA.HOST with a SET_INFO call. For maximum portability between versions of ORCA, the GET_INFO and SET_INFO calls should be done using the macros provided in the subroutine library. For terminal errors, include a displacement into the text file that was being processed. This displacement is used by the system editor, which will display the line that caused the problem at the top of the screen. Calling the editor is done automatically by ORCA.HOST whenever control is returned with the maximum error level encountered showing a negative value. Since the error code is returned in a one byte field, any value over 127 is considered negative. Currently, terminal errors are always flagged with a value of $FF.

Several inputs are provided by the system. With two exceptions, these are self explanatory. The compiler is responsible for reading and using these inputs, and for updating any of the values that have changed via the SET_INFO call when the comilation has completed. The KEEP parameter is covered in more detail in a later section.

The second parameter which bears special comments is the command type entry, TYPE. This entry tells the operaing system which functions need to be done. The three least significant bits are currently used. On entry to a compiler, the least significant bit will always be set; this bit indicates that a compile is to be performed. The next bit indicates that a link edit is needed. Finally, the bit in position 2 indicates that the finished program is to be executed. There are four conditions that can be in effect on exit from a compiler. The first is a terminal error, with a negative error code. The second is a normal error. In both of these cases, the setting of the TYPE byte is immaterial, since ORCA.HOST will not continue with further processing. (In one case it enters the EDITOR, in the other it enters the MONITOR.) The third case is the normal completion of a compile, in which case the least significant bit of the TYPE byte should be cleared. This signals to ORCA.HOST that the compile is finished. It is not the responsibility of the compiler to set the next two bits; this was done by the monitor in response to the specifiec command that started the compile. Finally, there is the case where the compile stops due to encountering an unrecognized language (covered in detail later). In that case, the TYPE byte should not be changed.

Identifing the Language

Each language used under ORCA is assigned a unique number by the Byte Works. This number is associated with the type of language in use. In the to determine the largest alphabetical suffix now in use, and use the next one. For example, the normal situation is for the second language used in a two language compile to find a .A file created by the first language, so a .B file would be created.

Keep in mind that the purpose of the root file is to hold the first code segment to be executed. In many languages, like assembly language, BASIC and Fortran, the first subroutine compiled is the first one to be executed. In languages like C and Pascal, this is not true. If the compiler is careful, it is reasonable to open the .A file immediately, opening the root file when the entry code is actually compiled. (For Pascal, this would be the program block. For C, it would be the function "main".)

Naming Conventions in Object Modules

The format for object modules varies between the Apple // and Cortland implementations. This will change when we release V4.1 of ORCA/M for the Apple //, which will conform to the convention used on the Cortland.

In both versions, it is important to follow several conventions when writting names to object modules. If the source language is case insensitive, like Pascal, always use uppercase letters in identifiers. For fixed length names, be sure and padd unneeded characters with spaces. This will make the object modules created by the compiler compatible with those created by ORCA/M.

Partial Compiles

The discussion in the last section probably seems like a mess. It is complicated for a reason. That reason is partial compiles.

Let's face it, no matter how fast a computer gets, there will be people trying to develop programs that push the resources of the machine to the limit. On an Apple //, compiling a 500 line program is quite an undertaking with the compilers available at the moment. You certainly don't make a casual change to a program using Manx C on the Apple // - it takes too long to compile. If you are using macros extensively, the same is true under ORCA/M. Partial compiles provide some relief from long compile times by allowing the compiler to recompile only the parts of a program that have changed.

To accomplish this, the compiler should split the program into individual code segments. The ORCA/M assembler does this at the beginning of each code segment. Small-C does so for each function definition. The only critical requirement that must be met is that the order of these units should not matter. Since it is not possible to "drop through" the end of a C function, C, for example, fulfills this requirement.

When a partial compile is done, the compiler scans the source code looking for the start of code segments. When one is found, it is compiled, and the code segment is sent to the object module. If the code segment is the first code segment that will be executed when the program is run, delete the old root file and create a new one. Otherwise, open a new output file using the next available alphabetical suffix. All new code segments go in that object module.

When the linker does its thing, it starts with the root module, and links that segment. It then scans all of the object modules, and starts with the one with the highest alphabetical suffix. It then works its way back to the .A file, ignoring any segments which it has already found. This way, it always uses the most recent version of each segment.

To see how to pick up the list of names for a partial compile, see the documentation for GET_LINFO.

Interpreters

Up until this point, the fact that a language might be an interpreter (or at least might not need the services of a link editor) has been ignored. Interpreters can be executed under ORCA, with some limitations. The most severe is that the interpreter could never make calls to code compiled by a compiler, since the link editor cannot be used to combine the interpreted code with the compiled code.

The only special requirement to make an interpreter work properly is to remember to clear all three bits of the TYPE byte before doing a SET_LINFO to return control to ORCA.HOST. This is because the interpreter itself executes the program, so linking and seperate execution are not needed.

command processor, the number is coupled with the name of the compiler used to translate the files. For example, ASM6502 is the name of the ORCA/M macro assembler, as configured to assemble 6502 code. This is also the name of the command used to tell the system that new files should be sent to this program when they are assembled, and is the auxiliary field name that shows up in the directory when the file is cataloged.

This number must appear as the fourth byte of an assembler or compiler, in case the number must be changed by the user for some reason. For example, there may be two pascal compilers available some day, and for some reason, someone may wish to uniquely identify both compilers on the same system. Perhaps more to the point is the current situation where two versions of the advanced linker are available.

All languages should have a meta-command that corresponds to the ORCA/M APPEND directive; this is a command that transfers control to another file. Whenever this command is used, the compiler must check the first byte of the auxility field of the appended file to make sure that the value of that byte matches the fourth byte of itself. If the bytes do not match, the new file is of another language. The compiler should then close the object module it is generating, and do a SET_INFO command to transfer control back to ORCA.HOST.

The following table shows the language numbers which have been assigned so far. Inclusion of a language in this table does not indicate that there is any intentionon the part of any party to produce a compiler for that language. If you are writing a compiler for a language not listed here, please apply to the Byte Works for a language number. We will be happy to assign one, insuring that your finished product will not have future conlicts with other languages.

Number	Name	Use
0	PRODOS	ProDOS TXT file
1	TEXT	Text formatter
2	ASM6502	6502 assembler
3	ASM65816	65816/65802 assembler
4	BASIC	BASIC compiler
5	PASCAL	Pascal compiler
6	EXEC	Command files
7	C	C compiler
8	FORTRAN	FORTRAN compiler
9	ADA	ADA compiler

10	FORTH	FORTH interpreter
11	CFORTH	FORTH compiler
12	IBASIC	BASIC interpreter
13	LISP	LISP interpreter
14	MODULA	Modula 2 compiler
15	PROLOG	Prolog interperter
16	PILOT	Pilot interpreter
17	LOGO	Logo interpreter
18	COBOL	COBOL compiler
19	ASMZ80	Z80/8080 assembler
20	ASMZ8000	Z8000 assembler
21	ASM6800	6800 assembler
22	ASM8085	8085 assembler
23	ASM65C02	65C02 assembler (usually combined with 1)
24	ASM68000	68000 assembler
25	ASM8086	8086/8088 assembler

The Relocatable Output File

The output from a compiler under ORCA should be in the form of one or more relocatable files, which are then processed by the link editor. The KEEP variable passed to and from ORCA HOST via SET_LINFO and GET_LINFO calls is the key to determinaing the names and number of files to generate.

Basically, each program consists of a relocatable file that ends with the characters ".ROOT".. If, for example, the program is saved under the name MYFILE, the program consists of the relocatable file MYFILE.ROOT. The root file, as it is called, contains the entry point for the program, and is always placed at the beginning of the executable file by the link editor. If a program uses any subroutines, these appear in subsequent files with ascending alphabetical endings, like MYFILE.A and MYFILE.B.

This is all keyed off of the KEEP number. If the output file is to be kept at all, the KEEP variable will be 1, 2 or 3 on entry to the compiler. A value of 1 indicates that no output files have been opened, so the compiler should start by producing a root file from the first subroutine. A value of 2 indicates that the root file was created by a previous language, and that the compiler should start by creating a file with the suffix ".A". This is a fairly rare occurrance, since the normal situation is for the first language used to create both a root file and a .A file. In that case, the keep number will be 3. When the keep number is 3, the compiler should search the output directory